Karlsruher Institut für Technologie

Automatische Auszeichnung von Semantik in natürlichsprachlichen Spezifikationen

Diplomarbeit
von

Mathias Landhäußer

An der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dipl.-Inform. Sven J. Körner

Bearbeitungszeit: 1. Dezember 2009 – 31. Mai 2010

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 28. Mai 2010

Mathias Landhäußer

Kurzfassung

Requirements Engineering ist eine der wichtigsten Aufgaben bei der Softwareentwicklung. Aus diesem iterativen Prozess ist natürliche Sprache für die Dokumentation der Anforderungen nicht wegzudenken. Ungenauigkeiten in natürlichsprachlichen Dokumenten können jedoch im weiteren Softwareentwicklungsprozess zu Problemen führen, weswegen sie früh gegen formale(re) Modelle ersetzt werden. SAL$_E$ mx stellt eine Methode bereit, mit der aus (annotierten) natürlichsprachlichen Texten UML-Modelle systematisch und automatisch erzeugt werden können.

Die zu SAL$_E$ mx gehörende Annotationssprache SAL$_E$ ermöglicht es dem Analysten, die Semantik eines Textes mithilfe von thematischen Rollen exakt aufzuschreiben. Die notwendige, manuelle Annotation stellt jedoch einen Flaschenhals bezüglich des Aufwandes dar – insbesondere dann, wenn der Analyst über keine oder wenig Erfahrung mit SAL$_E$ verfügt. Dies führt dazu, dass SAL$_E$ mx noch nicht so praxistauglich ist, wie es sein sollte.

Bestehende Ansätze zur Ermittlung von Semantik in einem Text beziehen sich auf Rollensysteme, die nicht mit dem von SAL$_E$ übereinstimmen. Darüber hinaus verfügt SAL$_E$ über weitere Konzepte, die von den bestehenden Ansätzen nicht oder nur teilweise unterstützt werden. Ein statistischer Ansatz scheitert, da die Trainingsbasis fehlt.

Die vorliegende Arbeit stellt AutoAnnotator vor. Ausgehend von einem natürlichsprachlichen Text in englischer Sprache ermittelt AutoAnnotator mithilfe von bestehenden NLP-Programmen die Struktur und teilweise die Semantik von Texten. Hierzu bedient er sich verschiedener Techniken und Programmen aus dem NLP und führt diese so zusammen, dass eine gültige Annotation ermittelt werden kann. Da die strukturelle Analyse der NLP-Programme nicht ausreichend ist, um die Semantik eines Textes zu erschließen, werden Ontologien verwendet, um spezielle Beziehungen zwischen Satzelementen zu ermitteln. Darüber hinaus werden ermittelte Beziehungen gegengeprüft und Unstimmigkeiten gegebenenfalls an den Benutzer gemeldet.

Da für die Annotationsergebnisse mit SAL$_E$ keine Goldstandards existieren, wurde der Ansatz mit mehreren kleinen Fallstudien evaluiert. Hierzu dienen ein kurzes, selbst erstelltes Beispiel, ein Beispiel von Fillmore sowie zwei Texte, die in der Literatur für die Bewertung von automatischen Modellierungsprogrammen gedient haben. Die Ergebnisse der Evaluation legen nahe, dass der verfolgte Ansatz im Prinzip funktioniert; es können aber auch Grenzen und Verbesserungsmöglichkeiten aufgezeigt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltexte-Verzeichnis

Glossar

Corpus Ein Corpus ist eine Sammlung von Texten, die gemeinsam verarbeitet werden sollen.

Genauigkeit Die Genauigkeit – oder *Accuracy* – ist ein Begriff aus dem Information Retrieval. Sie ist das Verhältnis aller korrekten Elemente zu allen Elementen. Mit anderen Worten ist die Genauigkeit das Verhältnis von *true positives* + *true negatives* zur *Gesamtmenge*.

Precision Die Precision – oder die Präzision – ist ein Begriff aus dem Information Retrieval. Sie ist das Verhältnis zwischen den korrekten gefundenen Elementen und den gesamten gefundenen Elementen. Mit anderen Worten ist die Precision das Verhältnis von *true positives* zu *true positives* + *false positives*.

Recall Der Recall – oder die Trefferquote – ist ein Begriff aus dem Information Retrieval. Er ist das Verhältnis zwischen den korrekten gefundenen Elementen und den gesamten korrekten Elementen. Mit anderen Worten ist der Recall das Verhältnis von *true positives* zu *true positives* + *false negatives*.

Stakeholder Ein Stakeholder ist eine Person, die ein Interesse an der zu erstellenden Software hat. Es kann sich hierbei neben den zukünftigen SW-Benutzern auch um den Kunden handeln, der das Produkt bezahlt (selbst wenn er es selbst nicht benutzt). Weitere klassiche Stakeholder sind Domänenexperten, Legacy-System-Benutzer oder Produktmanager auf Herstellerseite.

Abkürzungsverzeichnis

CASE Computer Aided Software Engineering.

CDIF CASE Data Interchange Format.

CIM Computation Independent Model.

MDA Model Driven Architecture.

MOF Meta Object Facility.

NER Named Entity Recognizer.

NLP Natural Language Processing.

OMG Object Management Group.

PIM Platform Independent Model.

POS Part-of-Speech.

PSM Platform Specific Model.

RAP Resolution of Anaphora Procedure.

RE Requirements Engineering.

RECAA Requirements Engineering Complete Automation Approach.

RUP Rational Unified Process.

Sal$_e$ Semantic Annotation Language for English.

Sense Software Engineers Natural language Semantics Encoding.

TimeML Time Markup Language.

UML Unified Modeling Language.

XMI XML Metadata Interchange.

1. Einleitung

Requirements Engineering (RE) ist eine der wichtigsten Aufgaben bei der Erstellung von Software. Es wird hierbei oft folgendermaßen definiert:

> [Requirements engineering is] the systematic approach of developing requirements through an iterative cooperative process of analysing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained.[LK95, Seite 13]

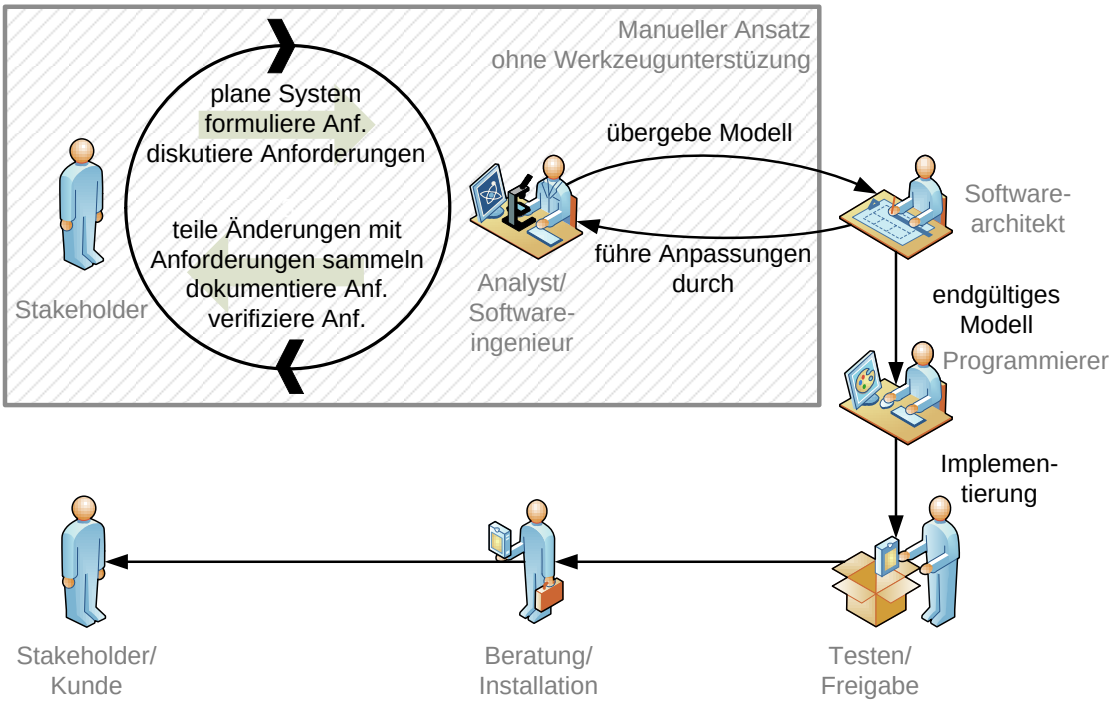


Abbildung 1: Der Softwareentwicklungsprozess.

Wie Abbildung 1 zeigt, liegt der zentrale Punkt in einem kooperativen Prozess, der von Analysten mit Kunden, Anwendern, (Domänen-) Experten und sonstigen Beteiligten (diese Menge von Menschen wird als *Stakeholder* bezeichnet) durchlaufen wird. An seinem Ende steht ein dokumentierter, gemeinsamer Wissensstand.

Cheng und Atlee stellen in [CA07] fest, dass die natürliche Sprache nicht nur in der Vergangenheit, sondern auch für die (absehbare) Zukunft eine der Hauptquellen im RE sein wird. Der Analyst muss die Dokumente des Kunden und damit seine Bedürfnisse, die Domäne, das Problem und die mögliche Lösung verstehen. Seine Aufgabe ist dann, dieses Verständnis so zu dokumentieren, dass es im anschließenden Entwicklungsprozess eine solide Basis darstellt und sowohl von den Stakeholdern, als auch von den Entwicklern verstanden werden kann.

Hasegawa et al. bechreiben, dass Analysten während der Anforderungsanalyse große Mengen an Texten lesen und verarbeiten müssen [HKKS09]. Während dieser Phase erstellen sie ein (inneres) Modell der Domäne und der Anforderungen. Dieser Prozess wiederholt sich bei jedem Projekt, unabhängig davon, ob bereits andere Projekte in ähnlichen Domänen durchgeführt wurden. Die Dokumente, die den Analysten zur Verfügung stehen, könnte man aber auch mit einer Software (vor-) verarbeiten und so den Analysten entlasten.

1.1. Motivation

SAL$_{\text{E}}$ MX stellt für den Softwareentwicklungsprozess eine Methode bereit, mit der aus natürlichsprachlichen Texten Domänenmodelle in der Unified Modeling Language (UML) erstellt werden können. Die automatische Modellextraktion ermöglicht hierbei einen schnellen Start in den Entwicklungsprozess. Gleichzeitig gibt sie dem Analysten die Möglichkeit, jederzeit eine aktualisierte Fassung der Domänenmodelle mit den Stakeholdern zu besprechen. Die Erstellung der Modelle, die in Kapitel 2.4 ausführlich beschrieben wird, erfordert jedoch, dass die Semantik der verwendeten Texte explizit aufgeschrieben wird. Diese Explizierung stellt einen Flaschenhals im Sinne des notwendigen Aufwandes dar. Zusätzlich muss bei Anpassungen am Text – oder am Modell – der neue oder geänderte Teil der Spezifikation erneut annotiert werden. Mit einer (semi-) automatischen Annotation, wäre der SAL$_{\text{E}}$ MX-Prozess ein Stück näher an einer praxistauglichen Lösung. Darüber hinaus ist derzeit nicht ausreichend evaluiert, wieviel Zeit für die manuelle Annotation tatsächlich anfällt. In [Gel10, Kapitel 7.3] wird eine qualitative Fallstudie beschrieben, bei der in 5 Stunden und 20 Minuten eine vierseitige Spezifikation annotiert wurde. Das Ergebnis wurde mit Modellen der selben Spezifikation verglichen, die von 54 Studentenpaaren im Rahmen einer Übung manuell erstellt wurden und denen ein ähnlicher zeitlicher Aufwand zu Grunde liegt. Das erzeugte Modell konnte mit den studentischen Lösungen mithalten; kein anderes Modell hatte mehr richtige Punkte, weniger fehlende Punkte und zwei Studentenpaare hatten mehr Fehler, als die mit SAL$_{\text{E}}$ MX erzeugte Lösung.

1.2. Zielsetzung der Arbeit

Ausgangspunkt der anzufertigenden Arbeit ist ein natürlichsprachlicher Text in englischer Sprache, in welchem die semantischen Informationen expliziert werden sollen. Der bisher notwendige manuelle Annotationsvorgang soll von einem Computersystem vorgenommen oder unterstützt werden, sodass eine automatische Modellerzeugung aus Text möglich ist.

Zunächst ist zu untersuchen, in wie weit grammatikalische Informationen als Ausgangspunkt für die Annotation dienen können. Gelhausen zeigt in [Gel10, Kapitel 3.6], dass ein grammatikalisches Modell nicht ausreichend ist, um die Semantik eines Satzes zu erfassen. Eine grammatikalische Analyse eröffnet jedoch einen sinnvollen Startpunkt für eine automatische semantische Analyse. Anschließend dient eine Ontologie dazu, die Beziehungen der genannten Elemente zu ermitteln, beziehungsweise zu untermau-

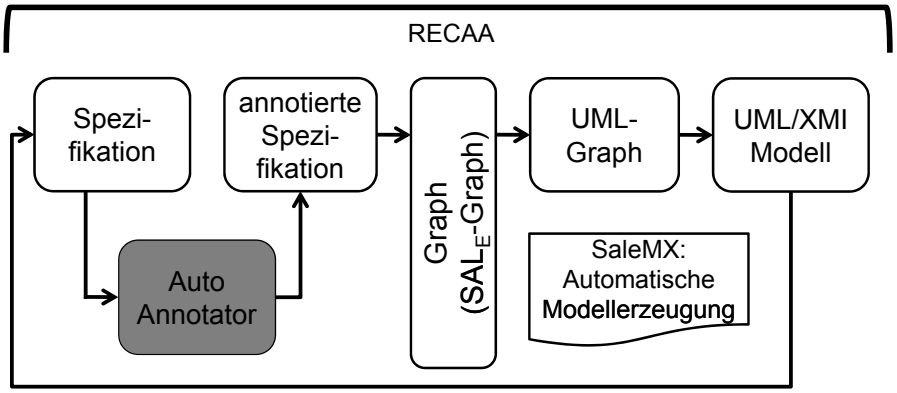


Abbildung 2: Die Position von AutoAnnotator im RECAA-Projekt.

ern. Werkzeuge der Koreferenzanalyse sollen herangezogen werden, um Referenzen und Personalpronomina aufzulösen. Um diese Vorgänge durchzuführen, wurde im Rahmen dieser Arbeit das Programm AutoAnnotator entwickelt. Abbildung 2 verdeutlicht die Position von AutoAnnotator im Modellextraktionsprozess mit Sal_E MX.

1.3. Gliederung der Arbeit

Zunächst werden im 2. Kapitel die Grundlagen und im darauf folgenden Kapitel die verwandten Arbeiten beschrieben. In Kapitel 4 wird die Aufgabenstellung analysiert. Kapitel 5 beschreibt den Enturf des erstellten Programms und geht auf die Implementierung und Erweiterbarkeit des Ansatzes ein. In Kapitel 6 wird der vorgestellte Ansatz an Fallbeispielen getestet und die Ergebnisse ausgewertet. Mögliche Verbesserungen und Erweiterungen werden im 7. Kapitel aufgezeigt. Eine Zusammenfassung schließt die Arbeit in Kapitel 8 ab.

2. Grundlagen

Autor(en)	Thema	Seite
Dawson & Swatman	Modellierungspraxis – ad hoc Transformation	4
Eman et al.	Kundenbeteiligung	4
Mich et al.	Marktstudie RE	5
Juzgado et al.	Manuelle Modellextraktionsmethode	5
Moreno & van de Riet	Modellextraktion aus nat. Sprache	5
Ryan	Natural Language Processing (NLP) im RE	5
Kof	Überwachte Ontologieextraktion	6
OMG	Model Driven Architecture	7
Gelhausen	SENSE, SAL$_E$, SAL$_E$ MX	7
Körner	RECAA	9

Tabelle 1: Übersicht über die behandelten Grundlagen.

Dieses Kapitel bescheibt die Grundlagen des vorgestellten Ansatzes und verweist auf die theoretischen Grundlagen der verwendeten Komponenten. Tabelle 1 fasst die angesprochenen Arbeiten zusammen. Eine Diskussion über verschiedene Lösungsansätze findet sich in Kapitel 5.

2.1. Anforderungsanalyse in der Praxis

Ein Großteil der Anforderungsdokumente sind in natürlicher Sprache verfasst. Im Bereich der Softwareentwicklung bietet sich insbesondere eine Beschreibung der Anforderungen mit formalen Methoden an, da mit ihnen die Software vollständig modelliert und verfiziert werden kann. Verfügt man über eine formale Spezifikation, kann man nach Abschluss der Entwicklung formal prüfen, ob die Software die Spezifikation korrekt umsetzt. Problematisch ist dann jedoch der Rückweg zum Kunden, denn dieser muss die Spezifikation verstehen können – zumal sie Teil des Vertragswerkes werden wird. Nur selten dürfte der Kunde versiert genug sein, um eine formale Spezifikation zu verstehen [DS99]. Die aus den Anforderungsdokumenten erstellten Modelle werden jedoch selten mit dem Kunden diskutiert. Stattdessen erstellen die Analysten zwei Modelle: Ein formales Modell, welches im Softwareentwicklungsprozess verwendet wird, und ein Modell für den Kunden. Die Übersetzung von einer auf die andere Seite erfolgt hierbei oft ad hoc, und muss bei Änderungen erneut duchgeführt werden. Dieses Vorgehen birgt neben dem erhöhten Aufwand das Risiko, dass sich die Modelle nach und nach voneinander entfernen.

Den Kunden überhaupt nicht oder zumindest weniger zu beteiligen, ist keine gangbare Lösung. Eman et al. zeigten in [EQM96], dass die Kundenbeteiligung im RE-Prozess qualitätssteigernd ist, und zwar umso mehr, je unsicherer[1] das Projekt ist. Die Ergebnisse

[1] Unsicherheit wird von den Autoren durch Merkmale wie unklare oder sich widersprechende Anforderungen und unklare oder instabile Arbeitsabläufe (*business process and management stability*) beschrieben.

legen nahe, dass die Beteiligung von Kunden die Folgen der Unsicherheit für das Projekt mildert.

Mich et al. führten 1999 eine Marktstudie durch, in der sie sich mit Anforderungsermittlung auseinandersetzen [MFI04]. Die Teilnehmer berichteten, dass 79 Prozent der Anforderungen in natürlicher Sprache erfasst werden (*common natural language*) und stammen hauptsächlich vom Kunden selbst oder aus Interviews mit dem Kunden. Darüber hinaus entfielen weitere 16 Prozent auf strukturierte natürliche Sprache (mit Schablonen oder Sprachbeschränkungen). Dies macht deutlich, wie wichtig die Verarbeitung von natürlicher Sprache für die Anforderungsermittlung ist. Mich et al. sehen daher ein großes Unterstützungspotential für Analysten durch NLP-Systeme, die unter anderem auch eine semantische Analyse durchführen sollten. Das Modellieren von Anforderungen stand für die Studienteilnehmer an dritter Stelle, wenn sie nach den wichtigsten Aktivitäten der Softwareentwicklung gefragt wurden. Drei Viertel der Befragten würden eine automatisierte Lösung für diese Aufgabe dem Outsourcing oder der internen Delegation vorziehen. Insgesamt ist Automatisierung für ca. 64 Prozent das Mittel der Wahl für eine Effizienzsteigerung. Beides wird als starker Indikator für eine mögliche Marktnachfrage nach NLP-Unterstützung im Computer Aided Software Engineering (CASE) angesehen. Hierbei ist der Wunsch nach Automatisierung so groß, dass bspw. Auslassungen oder Defekte im Modell in Kauf genommen werden, solange das Modell automatisch erzeugt und manuell angepasst werden kann. Abschließend stellen Mich et al. fest, dass ein derartiges NLP-CASE-Tool auch zur Ausbildung von Analysten genutzt werden kann; auf der einen Seite könnte man Analysten an den objektorientierten Entwurf heranführen und gleichzeitig die Fähigkeit der Modellierung und der Modellbewertung stärken.

Juzgado et al. beschreiben, dass bestehende RE-Methoden keine formalen Anleitungen zur Erarbeitung von objektorientierten Modellen geben [JML00]. Daher basiert der Erfolg dieser Methoden auf den Fähigkeiten und der Erfahrung des Analysten und ist somit nicht wiederholbar. In ihrer Publikation beschreiben sie eine manuelle Methode zur Modellextraktion. Ihre exakten Arbeitsanweisungen machen den Prozess wiederholbar und weniger abhängig vom ausführenden Analysten.

2.2. Informationsextraktion

Moreno und van de Riet legten 1997 einen Grundstein für die Modellextraktion aus natürlichsprachlichen Texten [Mor97, MR97]. Sie zeigten, dass Konzepte in der linguistischen Welt L durch bestimmte Muster repräsentiert werden. Dieselben Konzepte werden in der konzeptuellen Welt C mithilfe von Diagrammen dargestellt. Wie Abbildung 3 zeigt, wird die Verbindung beider Welten in einer dritten, der mathematischen Welt M hergestellt, indem L auf prädikatenlogische Ausdrücke (FOL) und C mithilfe der Mengentheorie (ST) abgebildet werden. In M kann gezeigt werden, dass (und welche) prädikatenlogische Ausdrücke mit welchen mengentheoretischen Aussagen äquivalent sind. Anschließend zeigen sie eine Menge von Sprachmustern und entsprechenden objektorientierten Entwürfen sowie deren Äquivalenz.

Ryan beschreibt in [Rya93], dass NLP-Programme auf statistischen Verfahren beruhen, die eine Fehlerrate haben können, welche die eines Menschen übersteigt. Deshalb

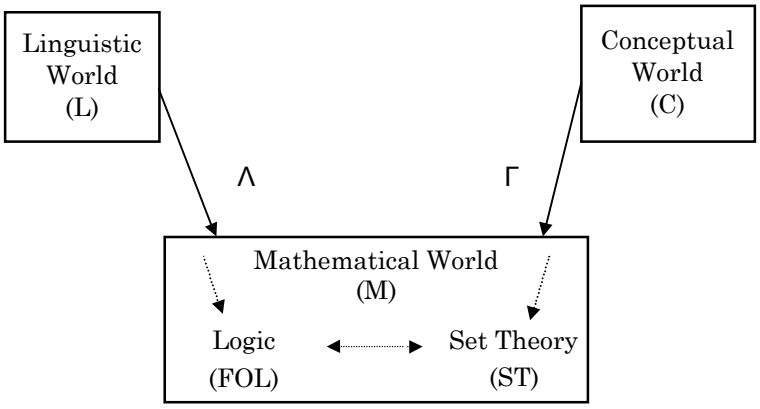


Abbildung 3: Die Abbildung von Sprache auf konzeptionelle Modelle [MR97].

sollen NLP-Programme keinesfalls menschliche Analysten ersetzen, sondern lediglich zu deren Unterstützung eingesetzt werden. Er führt außerdem aus, dass NLP nicht damit gleichzusetzen ist, dass ein Rechner einen Text *verstehen* kann; NLP könne aber dazu eingesetzt werden, Texte systematisch aufzubereiten und zu verarbeiten.

Kof stellt einen überwachten Prozess vor, mit dem auf Spezifikationen basierend Ontologien erstellt werden. Zunächst werden Terme aus den Spezifikationstexten extrahiert und anschließend zu Clustern zusammengefasst. Aus diesen können dann Taxonomien erstellt werden. Am Ende des Prozesses werden die ermittelten Begriffe zueinander in Beziehung gesetzt. Das Verfahren wurde zunächst in einer Machbarkeitsstudie anhand eines Textes von sechs Seiten mit positivem Ergebnis evaluiert [Kof04a]. Kof zeigt in [Kof04b] anhand einer größeren Fallstudie, dass sein NLP-basierter Ansatz auch für größere Dokumente genutzt werden kann. Für die Bearbeitung des 80-seitigen Dokumentes wurden fünf Arbeitstage benötigt, wobei allein ein Tag notwendig war, um die Spezifikation zu lesen. Diese Zeitspanne wurde – gemessen am Ergebnis – als akzeptabel und lohnenswert eingestuft. Zusammenfassend argumentiert Kof, dass der Ansatz nicht nur für die Verbesserung der Dokumentation verwendet werden kann, sondern auch dazu geeignet ist, eine Brücke zwischen den Anforderungsdokumenten und einer domänenspezifischen Ontologie zu schlagen. In [Kof05] zeigt er, dass seine Kombination verschiedener NLP-Techniken zu besseren Ergebnissen führt, als die Techniken für sich betrachtet. Seine Kombination bezieht ihre Stärke daraus, dass die Stärken einer Technik dazu genutzt werden können, die Schwäche(n) einer anderen auszugleichen. Trotz der notwendigen manuellen Eingriffe ist seiner Meinung nach NLP reif genug, um im Bereich der Ontologieextraktion für das RE eingesetzt zu werden.

Für die Aufgabe, eine automatische semantische Annotation durchzuführen, und somit für die automatische Modellerzeugung sind verschiedene Grundlagen vorauszusetzen. Zum Einen benötigt man Werkzeuge, die den gelieferten Text in eine zweckmäßige Datenstruktur überführen, zum Anderen benötigt man verschiedene Analysewerkzeuge. Nach der Analyse steht die Interpretation der Ergebnisse sowie die Überprüfung der ermittelten Informationen. Abschließend muss der annotierte Text in maschinenlesbarer

Form ausgegeben werden.

2.3. Die Model Driven Architecture

Die Model Driven Architecture (MDA) ist ein systematisches Vorgehen, um Software stufenweise zu erstellen; sie wurde von der Object Management Group (OMG) spezifiziert und als [MM03] herausgegeben. Die MDA ist hierbei nur als Rahmen anzusehen und zeigt, wie Modelle und Modelltransformationen in einem kontrollierten und effizienten Softwareentwicklungsprozess eingesetzt werden sollen. Abbildung 4(a) verdeutlicht das Vorgehen: Ausgehend von einer Idee oder einem Problem wird eine Softwarelösung angestrebt; diese Idee bestimmt relevante Einflussfaktoren aus der Realität, die sich im Domänenmodell (Computation Independent Model (CIM)) wiederfinden. Ab hier wird das Modell stufenweise über das Platform Independent Model (PIM) hin zum Platform Specific Model (PSM) transformiert; am Ende der Transformationskette steht ein ausführbares Programm.

Die Transformationsschritte (vergleiche Abbildung 4(b)) sollen dafür sorgen, dass bei einer Änderung an einem höhergelegenen Modell die folgenden Änderungen direkt bis nach unten zum Quellcode „durchgereicht“ werden können. Dies bedeutet jedoch nicht, dass es *das eine* MDA-Werkzeug gibt, welches die Modelltransformationen vornimmt. Stattdessen müssen die Transformationsregeln unter Umständen für jede Stufe und jedes Projekt neu definiert werden. Diese Transformationsregeln können in einer systematischen Arbeitsanweisung bestehen oder in Programmen, welche die Transformationen automatisch vornehmen. Hierbei wird ein möglichst hoher Automatisierungsgrad angestrebt. Die leere Box auf der rechten Seite von Abbildung 4(b) symbolisiert die Zusatzinformationen oder Werkzeuge, die für einen Transformationsschritt benötigt werden. Die (aufgeschriebenen) Transformationsanweisungen sollen hierbei zu einer Wiederholbarkeit führen und können optimalerweise in beide Richtungen angewendet werden, was zum sogenannten *roundtrip engineering* führen soll. Änderungen, die *von oben* kommen, sollen automatisch durchgereicht werden können; Änderungen, die *unten* vorgenommen werden, sollen sich in den übergeordneten Modellen wiederfinden.

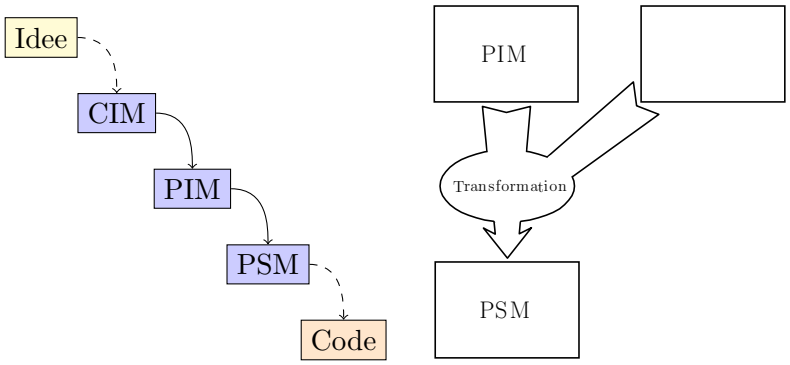
Durch diese Konzeption enthält die MDA ein hohes Maß an Automatisierung. Diese beginnt nach der Erstellung des CIM und endet mit dem ausführbaren Programm. Das CIM bleibt nach wie vor Aufgabe der Analysten, die das Modell aus den Ergebnissen der Anforderungsermittlung erstellen müssen.

2.4. SENSE, SAL_E und SAL_E MX

Gelhausen beschreibt Software Engineers Natural language Semantics Encoding (Sense) in [Gel10, Kapitel 4] folgendermaßen:

> [Sense ist ein] Formalismus für die Codierung der Semantik eines natürlichsprachlichen Textes – aus Sicht eines Softwareingenieurs.

Ziel der Entwicklung von Sense war die Erzeugung von Modellen aus natürlicher Sprache. So beschreibt Sense zunächst, wie die Semantik verschiedener Wortklassen und



(a) Der MDA Prozess. (b) Model Transformation [MM03].

Abbildung 4: Die Model Driven Architecture.

Grammatikstrukturen erfasst werden kann. Als Nebenprodukt liefert die explizit aufgeschriebene Semantik eine exakte Beschreibung des Verständnisses des Analysten, die vom (kundigen) Stakeholder geprüft werden kann [Gel10, Kapitel 3.6]. Die Semantic Annotation Language for English (SAL$_E$) ist die zu SENSE entwickelte Annotationssprache, die es ermöglicht, thematische Rollen so in einen natürlichsprachlichen Text einzufügen, dass die semantischen Informationen durch einen Rechner verarbeitet werden können. SAL$_E$ MX stellt als Modellextraktionsprogramm einen systematischen Startpunkt für die MDA dar. Es erzeugt Domänenmodelle aus natürlichsprachlichen Spezifikationen und verlegt somit den Automatisierungsprozess der MDA um eine Stufe nach vorne [Gel10]. Üblicherweise sind die Modelle auf der CIM-Stufe natürlichsprachliche Texte; diese können mit SAL$_E$ MX in UML-Diagramme überführt werden. Der Prozessablauf von SAL$_E$ MX ist in Abbildung 5 dargestellt und wird im Folgenden erläutert.

Vor einer automatischen Modellerzeugung in der UML steht die manuelle Kodierung der semantischen Informationen (1). Dies geschieht mithilfe der Annotationssprache SAL$_E$ [2]. Diese stellt u. a. thematische Rollen bereit, welche die konkrete Bedeutung bzw. Funktion der Satzelemente ausdrücken. Die thematischen Rollen müssen von einem Annotierer in den Text eingefügt werden.

Anschließend wird das entstandene SAL$_E$-Dokument mit dem SAL$_E$-Compiler in eine Graphspezifikation für das Graphersetzungssystem GRGEN.NET übersetzt (2). Der hieraus entstehende Spezifikationsgraph kann von den verschiedenen Werkzeugen von SAL$_E$ MX verarbeitet werden. Ein Werkzeug transformiert den Spezifikationsgraphen in ein UML-Modell, das im XML Metadata Interchange (XMI)-Format [3] (3) serialisiert wird. Es geht schlussendlich als CIM in die MDA ein (4).

[2] Die Funktionsweise von SAL$_E$ ist nicht Gegenstand der Betrachtung. Sie wird in Anhang B anhand eines kurzen Beispiels beschrieben.

[3] Das XMI-Format ist ein Standard der OMG und dient dem Austausch von Modellen auf der Basis von Meta-Modellen der Meta Object Facility (MOF). Neben UML-Modellen lassen sich folglich alle Modelle austauschen, solange die zugehörigen Meta-Modelle mit der MOF ausgedrückt werden können.

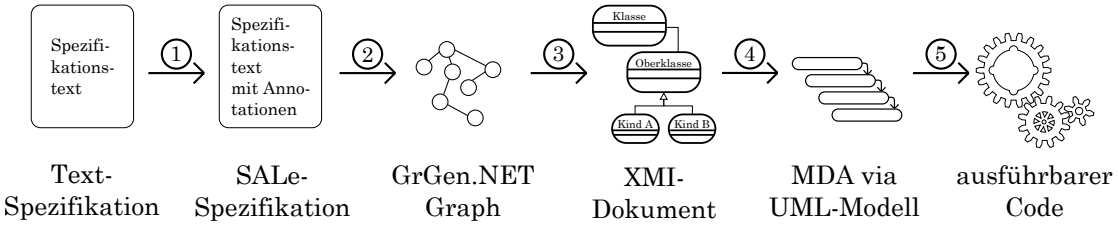


Abbildung 5: Der SAL$_E$ MX-Prozess – Von der Spezifikation über UML zum ausführbaren Programm.

Betrachtet man die weiteren Stufen des Softwareentwicklungsprozesses, so folgt die MDA, welche ein ausführbares Programm liefert (5).

Die Kodierung der Semantik mittels Annotation mit thematischen Rollen erfolgte vollständig manuell. Hieraus resultierte ein potentiell sehr langwieriger Prozess – insbesondere dann, wenn der annotierende Analyst wenig Erfahrung mit der Syntax und den in SAL$_E$ verwendeten thematischen Rollen (siehe [KDGL, Lan08]) hat.

2.5. RECAA

Requirements Engineering Complete Automation Approach (RECAA) [KDGL] ist ein Forschungsprojekt, das die Verbesserung des RE-Prozesses zum Ziel hat. Hierbei definiert RECAA keinen neuen RE-Prozess, sondern versucht die Schwächen vorhandener, bewährter Prozesse zu identifizieren und zu beheben. Dabei spielt die Analyse von textuellen Spezifikationen eine zentrale Rolle, aber auch die Modellerzeugung, wie sie mit SAL$_E$ MX geschieht, soll unterstützt werden.

Diverse Probleme textueller Spezifikationen wurden bereits vor RECAA von Forschern und Praktikern identifiziert und es wurden Strategien zu ihrer Vermeidung entwickelt (siehe beispielsweise [Rup09]). RECAA versucht nicht, diese Probleme durch Arbeitsanweisungen oder Verhaltensregeln zu umgehen, sondern diese Probleme automatisch zu identifizieren und – wo möglich – zu beheben. Hierbei sollen die bereits bestehenden Techniken wie statistische Parser genutzt und durch digitalisiertes Allgemeinwissen (*common sense*) in Form von Ontologien bereichert werden. Die Stärke des Ansatzes liegt insbesondere darin, sich nicht auf eine Technologie festzulegen, sondern verschiedene Technologien zu nutzen, die sich gegenseitig ergänzen. Im Laufe der Forschungsarbeiten wurde das Programm RESI entwickelt, welches Textschwächen wie unvollständig spezifizierte Prozesswörter erkennt und dem Analysten anzeigt [KB09a, KB09b, Bru09].

3. Verwandte Arbeiten

Für die vorliegende Arbeit wurden bestehende Technologien kombiniert, um die Struktur und Semantik eines Textes zu erschließen. AutoAnnotator nutzt bewährte NLP-Werkzeuge und Ontoligen, um eine Informationsbasis zu erstellen, welche dann ausgewertet wird. Im Folgenden werden die verschiedenen Werkzeuge und Verfahren vorgestellt. Tabelle 2 fasst diesen Abschnitt zusammen. Am Ende des Kapitels werden ähnliche Lösungsansätze aus der Forschung diskutiert.

Autor(en)	Thema	Seite	integr.
UPenn	Penn POS-Tags	10	ja
Liu	Montylingua	11	nein
Stanford NLP Group	POS-Tagger	11	ja
Stanford NLP Group	NL Parser	11	ja
Universität Stuttgart	TreeTagger	12	nein
Ge & Mooney	Scissor	13	nein
Ambriola & Gervasi	CIRCE	13	nein
Ambriola & Gervasi	CICO	13	nein
University of Sheffield	GATE	14	nein
Miller	WordNet	15	ja
Pazienza et al.	Kombi. v. WordNet. VerbNet, PropBank	15	nein
Cycorp Inc.	Research Cyc	15	ja
Dimitrov et al.	Leichtgewichtige Coreferenzanalyse	16	nein
Soon et al.	Maschinelles Lernen von Coreferenzen	17	nein
Qiu et al.	JavaRAP	17	ja

Tabelle 2: Übersicht über die betrachteten Werkzeuge und Konzepte.

3.1. NLP-Konzepte und Werkzeuge

Dieser Unterabschnitt befasst sich mit den grundlegenden Analyseprogrammen. Sie werden eingesetzt, um eine syntaktische Wissensbasis zu erstellen.

3.1.1. Part-Of-Speech-Tags

Die von AutoAnnotator ausgewerteten Part-of-Speech (POS)-Tags sind sogenannte Penn-Tags [San90]. Sie stammen aus der Annotationsarbeit, die in den neunziger Jahren an der Universität von Pennsylvania durchgeführt wurde und in der Penn-Treebank [Uni99] mündete. Die Penn-Treebank ist ein Corpus, der Texte aus vier verschiedenen Themengebieten (Wall Street Journal, The Brown Corpus, Switchboard sowie ATIS) enthält. Zunächst wurde der Corpus automatisch annotiert und anschließend manuell korrigiert; der Corpus umfasst rund 4,5 Millionen Wörter. Das Tag-Set basiert auf dem des Brown Corpus [MSM93] und wurde demgegenüber verkleinert, weswegen es

nur noch 36 POS-Tags sowie 12 andere Tags für Zeichensetzung und Währungssymbole enthält. Eine Liste der Tags findet sich in Anhang D.

3.1.2. Montylingua

Montylingua ist ein NLP-Programm, welches 2004 von Liu veröffentlicht wurde [Liu04]. Es verarbeitet einen englischen Text, auf welchem verschiedene Operationen ausgeführt werden können (Tokenisierung, POS-Tagging, Phrasenextraktion, Extraktion von Subjekt-Prädikat-Objekt-Tripeln (SPO-Tripeln), Lemmatisierung sowie Erstellung von Textzusammenfassungen). Die Verarbeitung basiert nicht nur auf syntaktischen Informationen, sondern wurde durch semantische Informationen angereichert. Somit können laut Liu viele Interpretationsfehler vermieden werden.

Betrachtet man beispielsweise den Satz *The mosquito bit the boy*, so ist die Zerlegung in zwei Phrasen `NP(The mosquito bit) NP(the boy)`[4] inhaltlich nicht korrekt. Richtig wäre folglich die Ausgabe `NP(The mosquito) VP(bit) NP(the boy)`, bei der *bit* das Prädikat des Satzes und nicht Teil der Phrase *the mosquito bit* ist [Liu04].

Leider war es in vertretbarer Zeit nicht möglich, Montylingua lauffähig zu installieren und es zu integrieren.

3.1.3. Der Stanford'sche POS-Tagger

Die Stanford Natural Language Processing Group [MJ] der Stanford University ist führend auf dem Gebiet des NLP. Die Forschungsarbeiten führten zu einem vielfältigen Programmangebot an führenden probabilistischen Parsern, die auf kontextfreien Grammatiken aufbauen. In [KM03] zeigen Klein und Danning, dass unlexikalisierte (*unlexicalized*) Parser mit frühen Implementierungen von lexikalisierten (*lexicalized*) Parsern mithalten können. Die gewonnenen Einsichten (siehe auch [KM02]) können teilweise als Verbesserungen in lexikalisierte Parser zurückgeführt werden.

AutoAnnotator verwendet derzeit als POS-Tagger den *Stanford Log-linear Part-of-Speech Tagger*, welcher unter [The09a] zu finden ist. Dieser Tagger zeichnet sich nicht nur durch seine Leistungsfähigkeit aus, sondern ist darüber hinaus in Java implementiert. Hierdurch kann er vollständig über seine API angesprochen und somit ohne Weiteres in AutoAnnotator eingebunden werden.

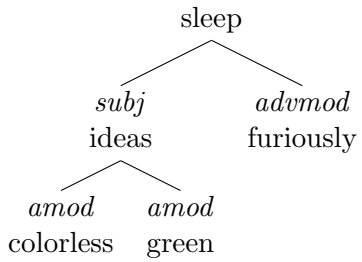
3.1.4. Der Stanford'sche Parser für natürliche Sprache

WordNet ist eine lexikalische Ontologie, die es Nutzern ermöglicht, Informationen wie Synonyme und Grundformen von Worten zu ermitteln (für Details siehe 3.2). Derartige Informationen gehören zum allgemeinen Sprachgebrauch und sind einfach zu verstehen. In [MM08b] stellen Marneffe und Manning fest, dass WordNet in der Praxis viel stärker genutzt wird, als beispielsweise die Penn-Treebank oder Framenet [LBF97]. Dies begründen sie mit der höheren Komplexität, die die anderen Konzepte aufweisen.

[4] Das POS-Tag *NP* steht für Nominalphrase; VP steht für Verbalphrase.

Colorless/JJ green/JJ ideas/NNS sleep/VBP furiously/RB ./.

(a) Ein vom Stanford POS-Tagger verarbeiteter Satz.



(b) Ein vom Stanford Parser erstellter Abhängigkeitsbaum.

Abbildung 6: Ein Satz mit POS-TAGs im Vergleich zum zugehörigen Abhängigkeitsbaum.

Um die Forschungsergebnisse im NLP-Bereich einer breiteren Nutzerschaft zu erschließen, entwickelten sie das Konzept der typisierten Abhängigkeiten (*typed dependencies*). Die typisierten Abhängigkeiten wurden mit dem Ziel entwickelt, eine direkte Beschreibung von grammatikalischen Relationen zu liefern. Hierbei zielen die Autoren darauf ab, ein nicht zu feinkörniges Analyseergebnis bereitzustellen um in der Praxis einsetzbar zu sein.

In [MM08a] gehen sie näher auf das Konzept ein und erläutern Aufbau und Funktion der typisierten Abhängigkeiten. Typisierte Abhängigkeiten werden definiert als binäre Abhängigkeiten `reln(gov, dep)`; hierbei steht `reln` für den Typ der Abhängigkeit zwischen dem Governor `gov` und dem Dependant `dep`. Es gibt 55 grammatikalische Relationen, die in dieser Form vom Parser ausgegeben werden können. Eine Liste der Abhängigkeiten findet sich in Anhang E.

Die Abhängigkeiten sind derzeit für Englisch, Chinesisch, Deutsch und Arabisch verfügbar; eine Anpassung für weitere Sprachen ist möglich. Der von der Stanford NLP Group bereitgestellte Parser [The09b] versieht die Satzelemente nicht nur mit einem POS-Tag, sondern ordnet den Satz in eine Baumstruktur. Hieraus können weit mehr Informationen abgeleitet werden, als aus POS-Tags; Abbildung 6 verdeutlicht den Unterschied.

3.1.5. Der Stuttgarter Tree-Tagger

Die Universität Stuttgart gibt den Stuttgarter *TreeTagger* heraus, welcher auf Markov-Modellen basiert.

In der Regel können die benötigten Wahrscheinlichkeiten für die Zustandsübergänge nicht gut geschätzt werden, wenn nur ein kleines Trainingsset zur Verfügung steht (sogenanntes *Sparsity-Problem*). Dies stellt für englische Texte oft kein Problem dar, da hierfür ausreichend Gold-Standards existieren; will man jedoch einen Tagger für eine „unpopuläre“ Sprache trainieren, hat man oft nicht genügend Informationen für ein gutes Trainingsset.

Der TreeTagger nutzt Entscheidungsbäume für die Schätzung von Wahrscheinlichkeiten [Sch94, Sch95]. Hierdurch wird das Sparsity-Problem der anderen Ansätze umgangen und es kann eine Genauigkeit auf der Penn-Treebank erreicht werden, die mit 96 Prozent besser ist, als bei einem normalen Trigram-Tagger. Für den TreeTagger stehen Parameterdateien für verschiedene Sprachen zur Verfügung.

Der Tagger selbst wird als ausführbares Programm ausgeliefert; für eine Integration in Java steht eine Bibliothek bereit, die von Richard Eckart de Castilho [5] betreut wird [Cas09].

3.1.6. Der SCISSOR-Parser

Ge und Mooney stellen in [GM05] den Parser Scissor *(Semantic Composition that Integrates Syntax and Semantics to get Optimal Representations)* vor. Hierbei handelt es sich um einen integrierten Parser, der zugleich Syntax und Semantik betrachtet. „Integriert“ bedeutet, dass semantische Informationen genutzt werden, wenn die Syntax nicht eindeutig geparst werden kann und umgekehrt. Hierdurch entstehen semantisch angereichterte Syntaxbäume (*Semantically Augmented Parse Trees*), deren Knoten nicht nur syntaktische, sondern auch semantische Annotationen besitzen.

Die Autoren evaluieren den vorgestellten Ansatz anhand von zwei Beispielen: Zum Einen an der CLang (RoboCup Coach Language), zum Anderen anhand von Anfragen an eine Geodatenbank mit Prolog. Ausgehend von manuell erstellten natürlichsprachlichen Sätzen (also „Übersetzungen“ von formalen Anfragen aus den zugehörigen Corpora) sollten wieder die formalen Anfragen erstellt werden. Hierbei schnitt Scissor besser ab, als die ihm gegenübergestellten Systeme.

3.1.7. CIRCE und CICO

Ambriola und Gervasi stellen in [AG97, AG01] CIRCE vor, das die parallelen (gleichzeitig von mehreren Personen durchgeführten) Anpassungen an Softwarespezifikationen erleichtern soll. *CIRCE* ist eine Anwendung, die es Analysten erlaubt, aus einfachen (d.h. unpräzisen) Anforderungen mithilfe von sogenannten Definitionen schrittweise exakte Anforderungen zu erhalten. Die Definitionen führen zu einer Textersetzung im Anforderungsdokument. Gleichzeitig werden UML-Modelle aus den natürlichsprachlichen Anforderungen erzeugt bzw. verfeinert.

Diese Modellerzeugung basiert auf den Ergebnissen des *CICO-Parsers* [AG99, Ger01], welcher aus dem Anforderungstext und einem Glossar einen domänenspezifischen Parsebaum erzeugt. Das Glossar dient einerseits dazu, eine 1:1 Beziehung zwischen Begriffen und (Realwelt-) Objekten zu erhalten. Andererseits erlaubt es eine Verbesserung der Parserergebnisse, da die Semantik der Begriffe bereits im Glossar geklärt wurde. Die Syntaxbäume verschiedener Anforderungssätze werden dann in eine sogenannte kanonische Form überführt und miteinander kombiniert um eine Baumstruktur zu erhalten. Aus

[5] Die Bibliothek wird unter `http://www.annolab.org/tt4j` angeboten. Eine Unterstützung des Entwicklers durch die Universität Stuttgart ist (abgesehen von der Verlinkung) nicht erkenntlich.

dieser erzeugen nachgelagerte Programme verschiedene Artefakte wie Klassendiagramme oder Datenflussdiagramme, die dem Softwareentwicklungsprozess zufließen. Da die UML keine Weiterentwicklung unterstützt, setzt man bei CIRCE auf eine Verfeinerung der Anforderungen und eine automatische Anpassung der UML-Diagramme.

3.1.8. GATE

Die *General Architecture for Text Engineering (GATE)* ist eine Sammlung von Java Programmen, die seit 1995 an der Universität von Sheffield entwickelt wird [CMBT02]. GATE legt Wert auf die Wiederverwendbarkeit der einzelnen Komponenten und besteht aus den folgenden Teilen: Basis für den Entwickler ist die IDE *GATE Developer*, welche den Zugriff auf die verschiedenen Komponenten bündelt. Das Framework *GATE Embedded* bietet eine Klassenbibliothek, welche zum Ansteuern der GATE-Komponenten aus Programmen heraus verwendet werden kann. Daneben bietet *GATE Teamware* eine Web-Oberfläche, welche einen verteilten Annotationsprozess durch mehrere Benutzer ermöglicht. Nicht zuletzt liefert GATE mit den vielen vorgefertigten Prozessen und Leitfäden eine strukturierte Umgebung für die Entwicklung von NLP-Systemen. Ein Wiki und die *GATE Cloud* sind derzeit in der Entwicklung begriffen. Von einer Verwendung von GATE für die vorliegende Arbeit wurde aufgrund der benötigten SAL_E-Datenstrukturen abgesehen.

Neben GATE gibt es weitere Linguistische Programmsammlungen, die von verschiedenen Organisationen (Firmen, Universitäten etc.) gepflegt werden. Die Liste der verfügbaren Programme ist so groß, dass eine Aufzählung an dieser Stelle weder vollständig noch hilfreich sein könnte. Eine Übersicht findet sich bspw. unter [Ali10b].

3.2. Ontologien

Ontologien für Allgemeinwissen (oder domänenspezifische Ontologien) sind nicht auf eine Art von Informationen spezialisiert. Sie enthalten Konzepte und Beziehungen zwischen Konzepten. Objekte in der Realwelt sind für Menschen eindeutig beschrieben – das kennzeichnende Wort ist fest in unserem Gehirn verankert. Nimmt man beispielsweise die Raubkatze *Jaguar*, so hat ein Menschen sofort das agile Tier vor Augen; der Begriff *Jaguar* selbst ist jedoch kontextabhängig: Es könnte sich auch um ein Auto handeln oder um ein Kampfflugzeug. Diese Mehrdeutigkeit des Symbols *Jaguar* wird in Ontologien mithilfe von Konzepten abgebildet: Für das Symbol *Jaguar* sollten also mindestens die Konzepte $Jaguar_{Raubkatze}$, $Jaguar_{Flugzeug}$ und $Jaguar_{Auto}$ vorhanden sein. Diese Konzepte sind dann sind mithilfe von typisierten Beziehungen verwoben. So könnte man sich eine Beziehung $Jaguar_{Raubkatze} \xrightarrow{istEin} Fleischfresser$ vorstellen.

Während es eine Standardaufgabe für WordNet ist, die Grundform eines Verbes zu bestimmen, zeichnen sich allgemeine Ontologien dadurch aus, dass sie Informationen über die Verbindungen von Konzepten enthalten. So macht es für eine lexikalische Ontologie durchaus Sinn, über alle Flexionen und Tempora eines Verbes Bescheid zu wissen (*essen*, *ich aß*, *er wird essen* ...). Betrachtet man hingegen eine allgemeine Ontologie, so erfährt man beispielsweise, dass *Menschen Lebensmittel essen* und dass eine *Birne*

ein *Lebensmittel* – eine *Glühbirne* aber kein *heißes Obst* ist (und daher nicht gegessen wird). Diese Verknüpfungen sind auf Konzeptebene hinterlegt; es macht keinen Sinn, diese Informationen für alle Flexionen zu speichern.

3.2.1. WordNet

WordNet [Mil09, Mil95] ist eine manuell erstellte lexikalische Datenbank, die für die englische Sprache entwickelt wurde. Die Entwicklung der WordNet-Datenbank begann 1985 und umfasst mittlerweile über 155'000 Wörter, die in über 206'000 Wort-Sinn-Paaren (sogenannten *Synsets*) beschrieben sind.

WordNet enthält Worte der offenen Wortklassen der Nomen, Verben, Adjektiven und Adverbien. Jedes Synset beschreibt ein festgelegtes Konzept, sodass Wörter mit mehreren Bedeutungen in verschiedenen Synsets verzeichnet sein können. Synsets sind derart miteinaner verbunden, dass man nicht nur Synonyme identifizieren kann (*sind beide Wörter im selben Synset?*) sondern auch Antonyme und ähnliche Beziehungen. WordNet ist somit auch als semantisches Lexikon anzusehen. Es wurde mit dem Ziel entwickelt, ein Wörterbuch für Programme zu sein. Daher lässt es sich einfach dazu verwenden, Wortbedeutungen und Grundformen zu bestimmen.

Um WordNet in AutoAnnotator zu verwenden, wurde der in [Bru08][Kapitel 5.4] erweiterte Serverdienst von Oliver Steele wiederum erweitert und eingebunden. Die Hauptaufgabe von WordNet ist hierbei die Bestimmung der Grundform von Wörtern um eine Parametrisierung von Anfragen in späteren Schritten zu ermöglichen.

3.2.2. Kombinieren von WordNet, VerbNet und PropBank

Pazienza et al. beschreiben in [PPZ06], wie sie die Informationen von WordNet, VerbNet [Sch06] und der PropBank [KP03] verknüpfen. Durch die Verbindung von Informationen über die Bedeutung von Worten (enthalten in WordNet), die *Frame*-Informationen aus VerbNet und die *corpus information* aus der PropBank entsteht eine mächtige Informationsquelle.

Leider lassen sich nur 20 Prozent des WordNet-Inhalts auf VerbNet abbilden und nur 43 Prozent von VerbNet auf die PropBank. So entsteht zwar eine interessante Informationskombination, jedoch umfasst sie nur wenige Worte (ca. 1000).

3.2.3. Cyc

Cyc ist eine englischsprachige Ontologie, welche seit 1984 gepflegt und weiterentwickelt wird. Sie enthält viele Konzepte des Alltagswissens mit dem Zweck, dieses für die maschinelle Verarbeitung zu erschließen und Schlussfolgerungen aus Fakten ziehen zu können. Für das Ziehen von Schlussfolgerungen steht eine leistungsstarke Deduktionsmaschine (*inference engine*) zur Verfügung, welche mit der LISP-artigen Anfragesprache CycL angesprochen werden kann.

Cyc wird seit 1995 kommerziell vertrieben und steht für wissenschaftliche Zwecke als ResearchCyc [Cycb] zur Verfügung.

Cyc wird in AutoAnnotator hauptsächlich dazu eingesetzt, ermittelte semantische Rollengefüge zu überprüfen. Beispiele hierfür finden sich in Abschnitt 5.4.5.

3.3. Auflösen von Anaphern

Eine *Anapher* ist ein Wort oder eine Phrase eines Satzes, das sich auf eine Entität bezieht, die bereits vorher genannt wurde (das sogenannte *Antezedens*): *Mathias spielt Basketball. Er trifft den Korb.* Im voranstehenden Beispiel bezieht sich die Anapher (das Personalpronomen) *er* auf das Antezedens *Mathias*. In einem SAL_E-Dokument können derartige Verweise in Form von Zusicherungen ausgedrückt werden. Sie bestimmen, dass ein Begriff A mit einem anderen Begriff B gleichzusetzen ist (siehe Abschnitt über Zusicherungen in Anhang B). Nachfolgend werden einige Ansätze erläutert; einen (nicht mehr ganz aktuellen) Überblick gibt Mitkov in [Mit99].

3.3.1. Ein leichtgewichtiger Ansatz

Dimitrov et al. beschreiben in [DBCM05] einen leichtgewichtigen Ansatz für die Koreferenzanalyse; sie schränken hierbei das Problem auf die Ermittlung von Antezedenzien ein, die benannte Entitäten (*named entities*) sind. Ihr Ansatz basiert nicht auf ausgefeilter Linguistik oder Domänenwissen, sondern nur auf POS-Tags und Informationen über benannte Entitäten.

Grundlage für die Entwicklung und die Evaluation des Ansatzes ist der ACE-Corpus, welcher sich aus drei Quellen speist: Nachrichtensendungen (Transkriptionen von ABC News, CNN und anderen), Zeitungsartikeln (hauptsächlich von der Washington Post) und Texten von Nachrichtenagenturen. Vor der Entwicklung ihrer Programmmodule analysierten die Autoren den Corpus, um die Häufigkeitsverteilung verschiedener Anapher-Kandidaten zu bestimmen. Anschließend wurde von der Betrachtung seltener Gruppen abgesehen, da sie selbst bei korrekter Behandlung nicht stark zur Güte des Algorithmus beitragen würden. Aufgrund der Ergebnisse werden vom vorgestellten Ansatz nur ein Teil der Pronomina und Possessiv-Adjektive (meins, deins, ...) aufgelöst; sehr selten vorkommende Pronomen wie bspw. Possessiv- und Reflexivpronomen wurden nicht berücksichtigt [6].

Der Suchraum für Bezugspunkte ist in vielen Fällen mit vier Sätzen groß genug; eine Betrachtung des aktuellen Satzes und über drei seiner Vorgänger hinaus verbessert das Ergebnis nicht wesentlich. Das Ergebnis der Evaluation mit einer Precision von 66 Prozent und einem Recall von 46 Prozent legen nahe, dass für eine einfache Koreferenzanalyse ein leichtgewichtiger Ansatz ausreicht und keine ausgefeilten NLP-Techniken notwendig sind. Da die Autoren jedoch nur wenig Spielraum für Verbesserungen sehen, empfehlen sie, für eine Weiterentwicklung zusätzliche Informationen einzubeziehen.

[6] Possessivpronomen kamen in zwei der drei betrachteten Quellen gar nicht vor, im dritten nur zwei mal. Reflexivpronomen stellten lediglich 1,5 Prozent der Pronomen im Corpus.

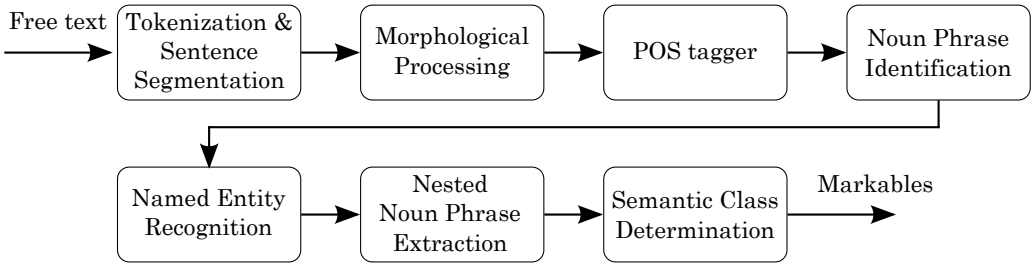


Abbildung 7: Die Systemarchitektur einer NLP-Pipeline [SNL01].

3.3.2. Ein Ansatz aus dem maschinellen Lernen

Soon et al. zeigen in [SNL01] einen Ansatz aus dem maschinellen Lernen für die Koreferenzanalyse in unbeschränkten natürlichsprachlichen Texten. Das Antezedens und die Anapher eines Koreferenzpaares bestehen aus sogenannten *Markables*, welche ausdrücklich nicht beschränkt sind (im Falle der Anapher bspw. auf Personalpronomina), sondern allgemeine *noun phrases* sein können. Aus einem manuell mit Koreferenzinformationen versehenen Corpus werden die möglichen Markables ermittelt und für Paare Eigenschaftsvektoren mit zwölf Dimensionen erstellt; betrachtet werden u. a. der Abstand, Übereinstimmung in Genus oder Numerus. Zusammen mit den Koreferenzinformationen wird dann ein Klassifizierer erstellt, welcher weitere Markable-Paare basierend auf ihrem Eigenschaftsvektor bewerten kann. Die Verarbeitungskette in Abbildung 7 zeigt das Vorgehen, mit dem die Markables aus dem Text extrahiert werden.

Die Ergebnisse der Evaluation zeigen, dass für die betrachteten Corpora MUC-6 und MUC-7 eine Genauigkeit erreicht werden kann, die ähnlich gut ist, wie die von nichtlernenden Ansätzen.

3.3.3. Resolution of Anaphora Procedure (RAP)

Lappin und Leass veröffentlichten 1994 den Resolution of Anaphora Procedure (RAP) Algorithmus [LL94]. Er ermittelt aufgrund syntaktischer Informationen zu Personalpronomina in der dritten Person sowie lexikalischen Anaphern das Antezedens.

Qiu et al. veröffentlichten 2004 in [QKC04] eine Referenzimplementierung des RAP-Algorithmus. Ihre Implementierung *JavaRAP* erreicht eine Genauigkeit von 57,9 Prozent und bezieht die syntaktischen Informationen vom ebenfalls frei verfügbaren Parser von Eugene Charniak [Cha00]. Charniaks Parser versieht die eingegebenen Sätze nicht nur mit POS-Tags, sondern erzeugt den für RAP notwendigen Syntaxbaum.

JavaRAP wurde in AutoAnnotator integriert und seine Ausgabe in Zusicherungen umgewandelt. Für nähere Informationen zur Installation und dem Zusammenspiel JavaRAP und Charniaks Parser siehe Anhang H.

Autor(en)	Thema	Seite
Gildea & Jurafsky	Ermitteln von semantischen Beziehungen	18
Harmain & Gaizauskas	CM-Builder	19
Overmyer et al.	Linguistic assistant for Domain Analysis (LIDA)	19
Montes et al.	Extraktion von OASIS-Code aus Text	20
Deeptimahanti & Sanyal	SUGAR, UMGAR	20
Hasegawa et al.	Anforderungsmodelle aus Text	21
Verma et al.	Requirements Analysis Tool	21

Tabelle 3: Übersicht über ähnliche Ansätze.

3.4. Ähnliche Lösungsansätze

Während der Analysephase stehen dem Analysten viele Dokumente zur Verfügung, die auf eine Verarbeitung warten. Teilweise liegen diese Dokumente als handschriftliche Notizen vor, andere sind bereits digialisiert. Alle diese Dokumente können als Eingabe in ein Softwaresystem dienen, welches Modelle oder andere Artefakte für den Softwareentwicklungsprozess aus ihnen erzeugt. Im Folgenden werden ähnliche Lösungsansätze vorgestellt; Tabelle 3 fasst die Ansätze zusammen.

3.4.1. Ein statistischer Semantik-Tagger

Gildea und Jurafsky stellen in [GJ02] ein System vor, mit dem semantische Beziehungen von Konstituenten mit einer Genauigkeit von rund 80 Prozent ermittelt werden können. Das System basiert auf statischen Informationen, die aus dem FrameNet-Corpus gewonnen wurden. Verwendet wurden die Argumentrollen der Frames, die weder so abstrakt sind wie die neun Fillmore'schen Rollen, noch so speziell wie die (vielen Tausend) möglichen verbspezifischen Rollen. Um gegen eine Abhängigkeit des Ansatzes vom Rollensystem zu prüfen, führten die Autoren Testläufe mit einem abgewandelten Rollensystem durch. Sie erstellten hierzu eine Abbildung der urspünglichen Rollen auf ein allgemeines Rollensystem (mit Rollen wie *Agens*, *Patiens* usw.). Hierbei ließ sich häufig entweder eine 1:1-Beziehung ermitteln oder keine „passende" Rolle finden (so bspw. für die Rolle *Degree*). Deswegen wurden weitere Rollen wie *Topic* eingeführt. Das so erstellte allgemeine Rollensystem umfasst 18 Rollen. Bei der Auswertung zeigte sich, dass die Güte des Ergebnisses sinkt, sobald Rollengeneralisierungen eingeführt werden, die eher voneinander entfernte Rollen zusammenfassen. Eine starke Abhängigkeit der Gesamtgüte vom Rollensystem konnte jedoch nicht gezeigt werden; die ermittelten Ergebnisse unterscheiden sich nur um zwei Prozentpunkte zugunsten des allgemeineren Rollensystems. Während der umfangreichen Evaluation versuchten die Autoren, die Güte des Gesamtsystems dadurch zu erhöhen, dass Semantik und Syntax für die Erzeugung des Parse-Baumes integriert wurden (vergl. hierzu den Abschnitt zu Ge und Mooney weiter oben). Die Erweiterung des probabilistischen Systems führte jedoch zu keinen nennenswerten Verbesserungen. Dies führten sie auf die Robustheit zurück, über die bereits einfache probabilistische Systeme verfügen.

3.4.2. CM-Builder

Harmain und Gaizauskas beschrieben 2000 das *CM-Builder*-System [HG00]. CM-Builder ist modular aufgebaut und führt eine domänenunabhängige linguistische Analyse eines Dokuments durch. Basierend auf dieser Analyse werden Listen von Kandidatenklassen und zugehörigen Beziehungen erstellt. Darüber hinaus erstellt CM-Builder ein Begriffsmodell im CASE Data Interchange Format (CDIF), das für den Import in CASE-Programme bestimmt ist.

Die Arbeitsweise mit CM-Builder ist grob in vier Schritte unterteilt:

1. Erfassen der funktionalen Anforderungen oder einer Domänenbeschreibung in natürlicher Sprache.

2. Analyse der Eingabedaten auf syntaktischer und semantischer Ebene. Erstellen eines internen Diskursmodells.

3. Extrahieren von Klassen, Attributen und Beziehungen zwischen den Klassen aus dem Diskursmodell.

4. Erstellen eines ersten statischen Modells des Systems aus den extrahierten Elementen. Verwenden eines CASE-Programms zur manuellen Verfeinerung bzw. Anpassung des Modells.

Die NLP-Analyse beginnt mit einer lexikalischen Vorverarbeitung (Tokenisierung, Aufspalten des Dokuments in Sätze, POS-Tagging sowie eine morphologische Analyse). Anschließend werden mithilfe eines Parsers prädikatenlogische Aussagen erstellt und eine semantische Interpretation durchgeführt. Hierzu werden die erstellten Aussagen zusammen mit ontologischem Wissen zu einem Diskursmodell verschmolzen. Dieses Diskursmodell wird dann für die objektorientierte Analyse herangezogen, welche in der CDIF-Ausgabe mündet.

3.4.3. LIDA

2001 stellten Overmeyer et al. den *Linguistic assistant for Domain Analysis (LIDA)* vor [OLR01]. LIDA soll den Analysten helfen, objektorientierte Modelle einer Domäne zu erstellen. Manuell aus bestehenden Dokumenten Klassen und deren Eigenschaften zu extrahieren (Substantivanalyse etc.), erscheint den Analysten als „natürlicher“ Ansatz. LIDA unterstützt dieses Vorgehen, indem es die Wortanalyse mithilfe von NLP-Programmen automatisch vornimmt. Jetzt kann der Analyst die erzeugten Wortlisten in einer graphischen Benutzeroberfläche der manuellen Prüfung unterziehen.

Zunächst erhält er eine Liste von Nomen und zusammengesetzten Begriffen, die er als Klassen markieren oder verwerfen (als Synonyme deklarieren, entfernen, als Attribut markieren) kann. Anschließend erhält er eine Adjektivliste, aus der er die (zu den identifizierten Klassen gehörenden) Attribute auswählen muss. Danach wird eine Verbliste angezeigt, aus der Methoden, Rollen und Hierarchien entnommen werden können. Nach diesen Schritten ist vorgesehen, dass der Analyst weitere (nicht automatisch erkannte)

Modellelemente zum Modell hinzufügen kann. Eine nachgelagerte Komponente erzeugt aus dem erstellten Modell mithilfe von Schablonen eine textuelle Modellbeschreibung, die zur Verifikation herangezogen werden kann. Um mit anderen Programmen weiterarbeiten zu können, ist ein Export des Modelles jederzeit möglich.

3.4.4. Objektorientierte Modelle mithilfe von NLP

Montes et al. stellen in [MPEP08] ein Verfahren vor, mit welchem aus Texten objektorientierte Softwaremodelle erzeugt werden. Der vorgestellte Ansatz basiert stark auf NLP-Verfahren, was ihn in den Augen der Autoren über Ansätze wie LIDA oder CM-Builder erhebt.

Aufgrund der sprachlichen Analyse werden die Elemente von objektorientierten Modellen (Klassen, Methoden, Beziehungen und Eigenschaften) extrahiert. Als Eingabe dienen (spanische) Anwendungsfälle (Szenarios), die zunächst in einzelne Sätze aufgeteilt und dann weiterverarbeitet werden. Die darauf folgende semantische Analyse soll Mehrdeutigkeiten eliminieren (Wortumschreibungen, zusammengesetzte (Klassen-) Namen usw.). Basierend auf einem nun eindeutigen Szenario, erstellen sie eine Zwischenrepräsentation als Graph. Dieser Graph enthält nicht nur die rein textuellen Informationen, sondern auch statistische Kennzahlen und sprachliche Informationen (Morphologie etc.). Abschließend wird der Graph traversiert und ein OASIS-Code[7] erzeugt, welcher im Softwareentwicklungsprozess genutzt werden kann.

3.4.5. SUGAR und UMGAR

Deeptimahanti und Sanyal stellen in [DS08, DS09] ihren *Static UML model Generator from Analysis of Requirements (SUGAR)* vor. Ihr Ansatz basiert auf dem Rational Unified Process (RUP) [Kru03] und folgt somit einer anerkannten Technik zur Modellextraktion aus Texten.

Zunächst werden die Anforderungen erfasst. Hierbei werden mithilfe von NLP-Werkzeugen Sätze identifiziert, die nicht den einfachen Subjekt-Prädikat- oder Subjekt-Prädikat-Objekt-Schemata entsprechen; diese Sätze müssen vom Benutzer umformuliert werden. Anschließend werden die Substantive und die dazugehörigen Verbalkonstruktionen aus den Sätzen extrahiert und mit einem vordefinierten Glossar abgeglichen. Danach werden Akteure und deren Anwendungsfälle identifiziert. Anschließend werden Klassen und deren Eigenschaften (Attribute und Methoden) ermittelt. Die sich ergebende Liste wird gemäß des RUP zu einem Klassendiagramm weiterverarbeitet. Da der Ansatz jedoch ausschließlich mit Subjekt-Prädikat- und Subjekt-Prädikat-Objekt-Sätzen arbeitet, ist er eher im Bereich der eingeschränkten Sprache oder der schablonenbasierten Ansätze anzusiedeln.

In [DB09] stellen sie ihren *UML Model Generator from Analysis of Requirements (UMGAR)* vor, der eine Weiterentwicklung von SUGAR zu sein scheint. Er verwendet die-

[7] OASIS ist eine Beschreibungssprache für objektorientierte Modelle. Sie unterstützt neben der Definition von Objekten auch die Erzeugung prädikatenlogischer Ausdrücke, die zur Spezifikation äquivalent sind. Für weitere Informationen siehe [LHB92].

selben Konzepte und Herangehensweisen wie SUGAR. Da neben RUP auch Konzepte aus dem ICONIX-Prozess[RS07] verwendet werden, ist UMGAR jedoch nicht auf Anwendungsfall- und Klassendiagramme beschränkt. Hinzugekommen sind Kollaborationsdiagramme sowie eine Unterscheidung zwischen Klassendiagrammen für die Analyse und das Design. Darüber hinaus verfügt UMGAR über einen Java- und XMI-Export.

3.4.6. Erzeugen von Anforderungsmodellen

Hasegawa et al. beschreiben ein automatisches Werkzeug, welches sogenannte Anforderungsmodelle (*requirements models*) aus (japanischen) natürlichsprachlichen Texten extrahiert [HKKS09]. Anforderungsmodelle werden hierbei als abstrakte Modelle beschrieben, die während der Anforderungsanalyse erstellt werden. Es spielt keine Rolle, ob man objektorientierte UML-Modelle erzeugt oder einem featureorientierten Entwicklungsprozess folgt. Das Verfahren ist letztendlich gleich, unabhängig vom erstellten Modelltyp.

Um unabhängig vom Typ des Ausgabemodells zu sein, erstellen sie ein internes Datenmodell, welches Konzeptgraph (*conceptual graph*) genannt wird. Er enthält neben den Konzepten ihre Beziehungen und deren Eigenschaften. Ein Konzept ist *keine* Klasse im objektorientierten Sinn. Funktionen, Objekte (bekannte und benennbare Instanzen eines Konzeptes) und Umgebungen sind bspw. ebenfalls Konzepte.

Der Konzeptgraph wird nicht nur aufgrund einer NLP-Analyse erstellt, sondern es werden auch Data-Mining-Verfahren angewendet. Um unbedeutende Konzepte auszuschließen, werden Werte wie Termhäufigkeit (TF), inverse Termhäufigkeit (ITF) und Entropie der Begriffe betrachtet. Neben diesen Verfeinerungen gehen die Autoren davon aus, dass sich semantisch nahestehende Begriffe oft auch gemeinsam auftreten. Der Zusammenhang, in dem dies geschieht, wird für die Extraktion von Beziehungen herangezogen. Der erstellte Konzeptgraph kann dann von verschiedenen Übersetzern in spezielle Modelle (bspw. UML- oder Featuremodelle) überführt werden. Darüber hinaus steht er den Analysten und Entwicklern im weiteren Softwareentwicklungsprozess als Domänenontologie zur Verfügung.

3.4.7. Das Requirements Analysis Tool (RAT)

Neben den wissenschaftlichen Ansätzen gibt es auch (pragmatische) Ansätze, die NLP und Ontologiekonzepte in der RE-Praxis einsetzen. Das *Requirements Analysis Tool (RAT)* von Verma et al. ist ein solcher Ansatz. RAT ist ein Programm, das den Analysten während der Anforderungsermittlung unterstützen soll. Es führt verschiedene Analysen durch, die auf Erfahrungswerten (*best practices*) basieren [VK08].

Der von Verma et al. verfolgte Ansatz basiert auf benutzerdefinierten Glossaren und eingeschränkter Sprache. RAT verwendet vier Glossare: Je ein Glossar für Handlungen/Aktionen und Handelnde/Agenten, eines für Modalworte und eines für „problematische Begriffe“. Ein problematischer Begriff ist bspw. „täglich“, da nicht klar wird um wieviel Uhr etwas täglich geschehen soll.

Für das Parsen wird ein Verfahren eingesetzt, das auf deterministischen endlichen Automaten basiert, welche Sätze entsprechend den vier vordefinierten Mustern akzeptieren

[JVKV09, VKV+10]. Endet die Verarbeitung einer Anforderung in einem Fehlerzustand, so kann eine Meldung erzeugt werden, die dem Analysten Rückmeldung über mögliche Verbesserungen gibt.

RAT ist als Plug-In für Microsoft Word konzipiert und integriert sich so einfach in den üblichen Arbeitsablauf der Analysten. Ähnlich wie die gewohnte Rechtschreibkorrektur werden die verschiedenen Analyseergebnisse angezeigt oder als Kommentare im Dokument hinterlegt. Durch das sofortige Feedback kann der Analyst unmittelbar auf etwaige Defekte reagieren und die Glossare anpassen. Gerade diese enge Rückkopplung (zusammen mit den vorgeschlagenen Verbesserungen) führten in einer Pilotphase zu einer hohen Akzeptanz des Programms.

Die erzielten Ergebnisse zeigen einerseits eine Verringerung des Zeitaufwandes für die Erfassung der Anforderungen nach ihrer Ermittlung in Interviews (Zeitersparnis zwischen 10 Prozent und 30 Prozent). Andererseits berichten Teamleiter während der Pilotphase von einer Zeitersparnis von 30 Prozent bis 50 Prozent für die Durchsicht der erstellten Dokumente. Neben der Textanalyse enthält RAT eine Komponente, die aus den Glossaren und Anforderungen einen semantischen Graphen erstellt. Dieser kann dazu genutzt werden, Abhängigkeiten zwischen Anforderungen, Konflikte und Ähnliches zu ermitteln.

4. Analyse

In diesem Kapitel wird beschrieben, wie ein natürlichsprachlicher Text halbautomatisch mit SAL$_E$-Annotationen versehen werden kann, sodass der Annotationsprozess im Rahmen der Anforderungsanalyse beschleunigt wird. Zunächst wird vorgestellt, wie ein SAL$_E$-Dokument aufgebaut ist und welche Informationen es enthält. Dann wird gezeigt, wie diese Informationen von einem Computersystem ermittelt werden können.

4.1. Der Aufbau eines SAL$_E$-Dokuments

Linguistische Struktur	**Erklärung**
AG *agens*	Handelnde Person oder Sache, die eine Aktion ausführt.
PAT *patiens*	Person oder Sache, die von einer Handlung beeinflusst wird.
STAT (+AG +PAT) *actus*	Eine Beziehung zwischen AG und PAT.
OPUS *opus*	Ein Werk (+) oder etwas Zerstörtes (−)[8].
FIN (+FIC) *fingens* und *fictum*	Das FIN spielt die Rolle eines oder verhält sich wie ein FIC.

Tabelle 4: Die thematischen Rollen von SAL$_E$ (Auszug).

SAL$_E$ verwendet thematische Rollen, um die Semantik von Sätzen zu kodieren: Betrachtet man als Beispiel *Mike Tyson likes chillies*, so möchte man kennzeichnen, dass *Mike Tyson* eine aktive Rolle spielt, die *chillies* die passive Rolle und *likes* die Beziehung zwischen beiden darstellt (nicht jedoch eine aktive Handlung wie bspw. *essen*). Wie in Tabelle 4 zu sehen, werden aktive (nicht im grammatikalischen Sinn!) Elemente in SAL$_E$ mit `AG` gekennzeichnet, die zugehörigen passiven/behandelten mit `PAT`. Die Verbindung in unserem Beispiel ist eine Beziehung, welche mit `STAT` gekennzeichnet wird. Somit ergibt sich die folgende Annotation:

```
1 [ Mike_Tyson|AG likes|STAT chillies|PAT ].
```

Die Annotation bezieht sich immer auf die Sachlage, die im Text steht. Betrachtet man den Satz *Mike loves Lakiha*, so gelangt man zur Annotation in Zeile eins:

```
1 [ Mike|AG loves|STAT Lakiha|PAT ]. #{RICHTIG!}
2 [ Mike|PAT loves|STAT Lakiha|AG ]. #{FALSCH!}
```

Man könnte zwar annehmen, dass auch die Umkehrung *Lakiha loves Mike* gilt, dies berechtigt aber nicht dazu, die Annotation wie in Zeile zwei umzukehren. Kodiert wird *grundsätzlich* nur das, was im Satz steht.

Soll ein Text manuell in ein gültiges SAL$_E$-Dokument[9] überführt werden, so muss zunächst der Text vorbereitet werden. In einem SAL$_E$-Dokument müssen alle Sätze mit

[8] Vorzeichenbehaftete Rollen werden im SAL$_E$-Dokument mit einem `P` für + und einem `M` für − gekennzeichnet. Ohne Vorzeichen können sie nicht verwendet werden.

[9] Für eine detaillierte Beschreibung von SAL$_E$ siehe Anhang B.

eckigen Klammern versehen werden. Ist dies geschehen, müssen etwaig enthaltene Subphrasen ebenfalls mit eckigen Klammern versehen werden; erst dann können diese mit einer thematischen Rolle versehen werden. Wurden im Text Elemente genannt, die aus mehreren Worten bestehen (bspw. *Mike Tyson* oder *WHOIS server*), so müssen diese verbunden werden. Anschließend werden die thematischen Rollen vergeben und Füllwörter auskommentiert. Ist dies erfolgt, werden Modifikatoren[10] von Worten als Attribute oder als Multiplizitäten gekennzeichnet. Da sich Modifikatoren zunächst immer auf das direkt (rechts) folgende Wort beziehen, müssen sie in den übrigen Fällen verschoben werden. Hierfür müssen die Richtung des Verschiebens und die Anzahl der zu überspringenden Worte angegeben werden (Kommentare werden für die Zählung übersprungen). Werden im Satz Worte verwendet, die sich auf ein vorher Genanntes beziehen (wie bspw. Personalpronomina), so sind entsprechende Zusicherungen einzufügen, um diesen Sachverhalt klarzustellen.

SAL$_E$ kann den Unterschied zwischen *dem selben* und *dem gleichen* ausdrücken: Wird im Verlaufe des Textes ein Wort zweimal gebraucht, z. B. ein Name, so handelt es sich um die *selbe* Entität, wenn die zweite Nennung als Referenz im SAL$_E$-Dokument aufgeführt ist. Wird zweimal dasselbe Wort verwendet, ohne eine Referenz zu benutzen, so handelt es sich um zwei *gleiche* Entitäten. Diese Feinheit der Semantik kann zwar mit SAL$_E$ dargestellt werden, wird das Dokument dann mit SAL$_E$ MX verarbeitet, werden für gleiche Entitäten separate Klassen im UML-Diagramm angelegt. Dies erscheint zunächst verwirrend, sind Klassen doch die Schablonen, aus denen *gleiche* Instanzen erzeugt werden. Da es sich beim erstellten Modell jedoch um ein Domänenmodell (nicht um ein klassisches Klassendiagramm) handelt, muss dargestellt werden, dass es verschiedene Klassen sein könnten – wenn die gleichnamingen Klassen beispielsweise unterschiedliche Funktionen erfüllen.

Um ein sinnvoll strukturiertes SAL$_E$-Dokument ausgeben zu können, stehen folglich die nachstehenden Aufgaben an:

1. Teilen des Dokuments in Sätze
2. Identifizieren von Subphrasen in Sätzen
3. Verbinden von mehrwortigen Elementen
4. Auskommentieren von bedeutungsleeren Füllworten
5. Ermitteln von Modifikatoren sowie deren Bezugswort
6. Zuweisen von thematischen Rollen zu Worten und Subphrasen
7. Ermitteln von Referenzen (bspw. das Auflösen von Personalpronomina)

[10] Modifikatoren heißen in SAL$_E$ *Properties*. Von der Verwendung einer Übersetzung wurde in dieser Arbeit abgesehen, da der Begriff *Eigenschaft* mit dem Begriff *Attribut* in der UML verwechselt werden könnte.

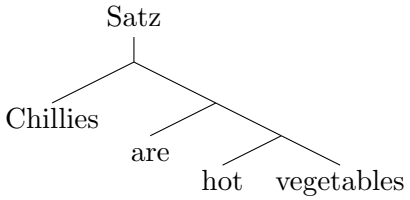


Abbildung 8: Der Syntaxbaum von *Chillies are hot vegetables.*

4.2. Syntax vs. Semantik

Auf unterster Ebene ist die Syntax die Menge der strukturellen Beziehungen, die sich aus der Anordnung von Zeichen ergeben [Goo00, Kapitel 1.1.5]. Diese Syntax ist rein aus den Daten selbst beobachtbar und messbar. Weitergehende Beziehungen zwischen den Daten werden als Semantik bezeichnet, welche sich aus einer Interpretationsvorschrift ergeben. Betrachtet man auf dieser Ebene *ein Haus*, so hat man – rein syntaktisch – drei Buchstaben, ein Leerzeichen und weitere vier Buchstaben. Dass es sich um Worte handelt und das Leerzeichen zum Trennen von Worten verwendet wird, ist streng genommen bereits Semantik.

Betrachtet man die Syntax und die Semantik von natürlicher Sprache, so bildet die Syntax einen Teil der Grammatik; neben ihr gehören hierzu noch Wortbildungsregeln und Ähnliches. Die Syntax beschreibt, wie Worte zu größeren Einheiten zusammengefasst werden und wie aus diesen Einheiten Phrasen gebildet werden, die wiederum zu Sätzen zusammengeführt werden. Diese Satzelemente werden als *Konstituenten* bezeichnet – sie sind die Bausteine, aus denen Sätze bestehen. Oft werden Sätze als Syntaxbaum (auch: Strukturbaum oder Stemma) aufgeschrieben, um die Struktur eines Satzes und den Aufbau seiner Konstituenten zu beschreiben. Betrachtet man den Syntaxbaum in Abbildung 8 von unten nach oben, so sieht man, dass die Worte *hot vegetables* zusammengenommen einen Konstituenten bilden. Dieser Konstituent bildet das (attributierte) Prädikatsnomen zu *are*. Die Konstituenten *Chillies* und *are hot vegetables* bilden dann zusammen den Satz.

Die Semantik in diesem Kontext ist *die Bedeutung* der Worte; wir *wissen*, dass eine *chili* ein *vegetable* ist und das *hot* ein Attribut ist. Für diese Interpretation ist jedoch Zusatzwissen nötig; die Information, die im Satz steckt, ist für ein Verständnis allein nicht ausreichend. Verfügt man jedoch über diese Zusatzinformationen, so kann man die Semantik mithilfe von SAL$_E$ explizit aufschreiben. In unserem Beispiel müsste man zuerst *are* auskommentieren, da es sich zwar um ein Verb, jedoch nicht um irgendeine Aktion handelt – *sein* ist keine Handlung (im weiteren Sinn). Der Umstand, dass sich eine Sache so verhält, wie eine andere Sache – oder anders ausgedrückt: dass eine Sache eine andere Sache *ist*, lässt sich mit den thematischen Rollen *fingens* (FIN) und *fictum* (FIC) beschreiben. *Hot* ist ein Attribut von *vegetables* und wird daher mit einem `$` versehen. In diesem Fall sind die *chillies* das *fingens* und nehmen die Rolle eines *hot vegetables* ein:

```
1 [ Chillies|FIN #are $hot vegetables|FIC ].
```

Diese beiden Rollen lassen sich in diesem Fall über das kopulative Konstrukt ermitteln. Ebenso lässt sich das Attribut *hot* über die Grammatik eindeutig auf *vegetables* beziehen.

4.3. Erzeugen von SAL_E aus natürlicher Sprache

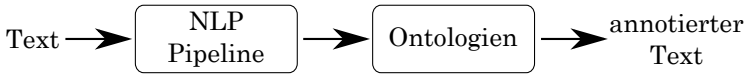


Abbildung 9: Der Prozess von AutoAnnotator.

In den voranstehenden beiden Abschnitten wurde dargelegt, welche Elemente in einem SalE-Dokument vorkommen können und was uns die Syntax eines Satzes liefern kann. Um den manuellen Teil der Annotation so gering wie möglich zu halten, müssen zunächst alle Anpassungen am Text vorgenommen werden, die aufgrund der Syntax festgestellt werden können.

Das Aufspalten eines Textes in Sätze ist eine Standardaufgabe und in vielen (NLP-) Programmen bereits realisiert [Ali10a, CMBT02]. Nachdem der Text in einzelne Sätze aufgespalten ist, können wir uns mit der syntaktischen Zusammenstellung der einzelnen Sätze beschäftigen.

Analysiert man die Struktur eines Satzes und konzentriert sich hierbei auf die Verben, so kann man oft Subjekt-Prädikat-Objekt-Konstruktionen feststellen. Ein Ansatz wäre, das Subjekt zum Handelnden, das Verb als Handlung und das Objekt als Behandeltes zu kennzeichnen. Ein Beispiel für eine solche Subjekt-Prädikat-Objekt-Konstruktion (SPO-Konstruktion) wäre *Der Hausmeister öffnet die Tür*[11]. Das Subjekt ist *der Hausmeister*, *öffnet* ist Prädikat und *die Tür* ist das Objekt. Man könnte also zu folgender (zufällig korrekter) Annotation kommen:

```
1 [ #Der Hausmeister|AG öffnet|ACT #die Tür|PAT ].
```

Formuliert man denselben Sachverhalt jedoch im Passiv, sieht es anders aus: *Die Tür wird vom Hausmeister geöffnet.* Hier ist *die Tür* das Subjekt, *vom Hausmeister* ein Objekt und *wird geöffnet* das Prädikat. Selbst wenn man das Hilfsverb *wird* „korrekt“ auskommentiert, erhält man folgende (falsche) Annotation:

```
1 [ #Die Tür|AG #wird #vom Hausmeister|PAT geöffnet|ACT ].
```

Möchte man also derartige SPO-Konstruktionen direkt auf semantische Rollen abbilden, wird man um eine Beschränkung der Sprache nicht umhinkommen (vergleiche den Absatz zu Deeptimahanti und Sanyal in Abschnitt 3.4).

Weitergehende Fragen entstehen, wenn man sich die semantischen Beziehungen detaillierter ansieht. Betrachtet man das Beispiel *Michelangelo malt ein Bild*, so kann man zunächst zu folgender Annotation gelangen:

```
1 [ Michelangelo|AG malt|ACT #ein Bild|PAT ].
```

[11] Vergleiche hierzu Abschnitt 6.2.

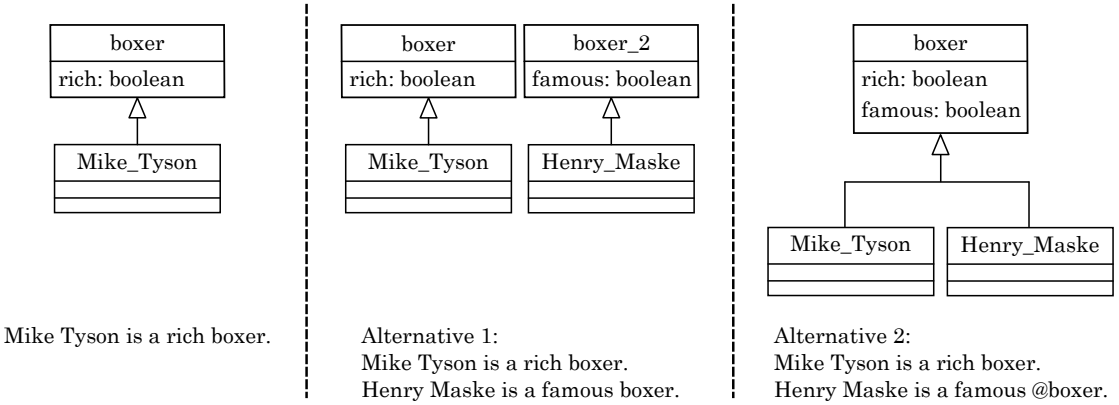


Abbildung 10: Das Boxer-Beispiel für Referenzen.

Diese Annotation ist an sich korrekt: *Michelangelo* ist zweifelsohne der Handelnde, *malt* eine Handlung und das *Bild* ist zweifelsfrei das Behandelte. Die semantische Beziehung der *Erschaffung von etwas* kann jedoch noch mit einer anderen SAL$_E$-Rolle genauer beschrieben werden: Verwendet man statt dem allgemeinen `PAT` die Rolle `OPUSP` (siehe Tabelle 4), so kann man ausdrücken, dass es sich um ein Erschaffen handelt:

```
1 [ Michelangelo|AG malt|ACT #ein Bild|OPUSP ].
```

Kennzeichnet man ein Satzelement mit `OPUSP`, so kann SAL$_E$ MX im Modellextraktionsprozess nicht nur die zugehörige Methode anlegen, sondern sogar den Typ des Rückgabewertes angeben.

Um derartige „Spezialrollen“ verwenden zu können, werden mehr Informationen benötigt, als sich aus der Satzstruktur herauslesen lassen. Um dies mit (statistischen) NLP-Programmen zu erreichen, benötigt man einen großen Trainingscorpus, welcher alle Rollen von SAL$_E$ und die zugehörigen Worte ausreichend abdeckt. Da ein solcher Corpus für SAL$_E$ nicht existiert, versucht AUTOANNOTATOR zunächst, die SPO-Konstruktion allgemein zu ermitteln (zu diesem Zeitpunkt werden teilweise Platzhalterrollen vergeben). In einem zweiten Schritt wird versucht, Ontologien zur Bestimmung der tatsächlichen Rollengefüge zu verwenden. Der sich ergebende Ablauf ist in Abbildung 9 zusammengefasst.

Da die SPO-Konstrukte und Ähnliches keinen direkten Rückschluss auf semantische Beziehungen erlauben, können sie nur als Ausgangspunkt für weitergehende Analysen verwendet werden. Welche Strukturen in welcher Weise für die Extraktion herangezogen werden, wird im folgenden Kapitel beschrieben.

4.4. Erfassen von Entitäten

Wie oben beschrieben, unterscheidet SAL$_E$ zwischen dem Selben und dem Gleichen. Handelt es sich um dasselbe Element, so muss in SAL$_E$ eine Referenz verwendet werden, um die Identität auszudrücken.

SAL$_E$ MX legt für gleiche Entitäten verschiedene Klassen an. Ebenso erzeugt es für Instanzen Klassen. Lässt man den Satz *Mike Tyson is a rich boxer* von SAL$_E$ MX model-

lieren, so erhält man zunächst eine Klasse *boxer* mit dem Attribut *rich*. Da es sich bei *Mike Tyson* (zumindest für einen Mensch) zweifelsfrei um eine Instanz handelt, erwartet man im Klassendiagramm, welches im linken Teil von Abbildung 10 gezeigt wird, eigentlich keine Klasse *Mike_Tyson*. Dennoch wird sie von SAL$_{\text{E}}$ mx erzeugt und eine Vererbungsbeziehung zwischen ihr und dem *boxer* angelegt. Wird nun im weiteren Verlauf ein anderer Boxer genannt, bspw. *Henry Maske is a famous boxer*, so erhalten wir nur dann ein zusätzliches Attribut in der Klasse *boxer*, wenn wir eine Referenz auf das erste Vorkommen von *boxer* im Text gesetzt haben. Andernfalls erhalten wir eine zweite Klasse *boxer_2*, die das Attribut *famous* trägt. Der mittlere und rechte Teil von Abbildung 10 verdeutlicht den Unterschied – die Referenz wurde durch ein @ gekennzeichnet. Ist nun in einer Spezifikation häufig von gleichen Entitäten die Rede, wird das erzeugte Klassendiagramm (oft) unnötig groß aufgebläht. Aus diesem Grund wurde bei der Arbeit mit SAL$_{\text{E}}$ mx oftmals eine Referenz nur deswegen gesetzt, weil man im Klassendiagramm nur eine Klasse erzeugt haben wollte. Diese – eigentlich falsche – Kodierung der Semantik führte zu einem intuitiv lesbaren und übersichtlichen Domänenmodell. Aufgrund der Herangehensweise von SAL$_{\text{E}}$ mx an diese Problematik ergeben sich verschiedene Optionen für den Umgang mit Referenzen. Je nach dem, welche Option gewählt wird, erzeugt SAL$_{\text{E}}$ mx ein unterschiedlich kompaktes Modell.

Ein sehr kompaktes Modell ergibt sich, wenn alle gleichnamigen Elemente auf ein Grundelement referenzieren. Hierbei können in einem der thematischen Annotation nachgelagerten Schritt alle Worte, die eine thematische Rolle einnehmen, auf ihre Grundform hin untersucht werden. Dann die „gleichen“ Elemente zu verschmelzen, kann leicht durchgeführt werden. Auf diese Weise erhält man – unabhängig von der Flexion – nur eine Klasse für ein Wort im Domänenmodell. Die Verschmelzung der Worte lässt sich in SAL$_{\text{E}}$ über Zusicherungen realisieren, wodurch der eigentliche Spezifikationstext nicht verändert werden muss. Diese Herangehensweise verbessert die Lesbarkeit des erstellten Domänenmodells, da es deutlich kompakter und intuitiver wird. Die Güte der Annotation wird jedoch verringert.

Die Betrachtung von Named Entities ermöglicht eine Annäherung an eine korrekte Umsetzung. Per Definition ist eine named entity ein Vorkommen einer textuellen Repräsentation einer Entität[12] (d.h. eines Individuums) in einem Text. Diese Tatsache lässt den Schluss zu, dass wann immer eine named entity mehrfach in einem Text gebraucht wird, eine Referenz zu setzen ist.

Die korrekte Semantik kann von AUTOANNOTATOR für Referenzen nicht ermittelt werden. Leider konnte im Verlauf dieser Arbeit kein Ansatz zum Ermitteln der *korrekten*

[12] Die Definition ist etwas weiter gefasst. Je nach Aufgabenstellung, wird unterschieden zwischen Personen (Mike), Organisationen (World Boxing Council), Zeitpunkten (19.12.2004), Beträge und Geldeinheiten (1500 USD) und Ähnlichem. Je nach Klassifizierung erhält die named entity eine Kennung, die einen Rückschluss auf die Klassifikation ermöglicht. Diese Klassifizierung muss bei der Auswertung berücksichtigt werden; verschmolzen werden bspw. Namen von Personen, nicht jedoch Beträge mit der zugehörigen Währung.

Referenzen gefunden oder entwickelt werden. Solange keine sinnvolle Umsetzung für dieses Problem in SAL$_E$ MX besteht, bleibt es im Ermessen des Benutzers, welche Stufe der Modellkompression gewählt werden soll.

Neben den obenstehenden Vorschlägen wäre es darüber hinaus möglich, in einem Glossar alle (System-) Komponenten aufzuführen, die im Modell nur ein Mal erzeugt werden sollen oder dürfen. Über dieses (benutzerdefinierte) Glossar könnte dann ähnlich wie bei der Auflösung über die named entities in einem nachgelagerten Schritt alle Nennungen von Glossareinträgen mit Referenzen versehen werden. Die Glossareinträge könnten während dem Annotationsprozess aus den Nominalphrasen gewonnen werden.

4.5. Zusammenfassung

Wie in den obigen Abschnitten gezeigt wurde, stellt eine Syntaxanalyse einen sinnvollen Startpunkt für eine automatische Annotation mit SAL$_E$ dar. Die notwendigen semantischen Informationen müssen teilweise aus anderen Quellen bezogen werden. Hierfür eignen sich allgemeine, wie auch domänenspezifische Ontologien. Das folgende Kapitel geht auf die konkrete Umsetzung des besprochenen Ansatzes durch AUTOANNOTATOR ein und beschreibt detailliert die einzelnen Schritte.

5. Entwurf und Implementierung

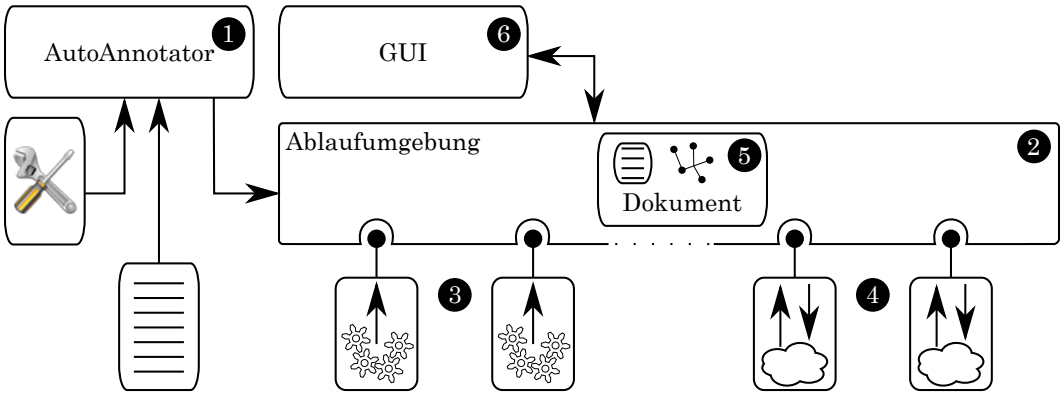


Abbildung 11: AutoAnnotator – Eine Übersicht.

AutoAnnotator wurde als interaktives Java-Programm implementiert. Die eingebundenen Programme sind unter Linux und Windows lauffähig, sodass AutoAnnotator auf beiden Plattformen verwendet werden kann. Dieses Kapitel beschreibt zunächst den Grobentwurf des Programms, geht dann auf die internen Datenmodelle ein und erläutert die einzelnen Komponenten in der Verarbeitungskette. Anschließend wird die Informationsextraktion motiviert und für die verwendeten Programme bzw. die von ihnen bereitgestellten Informationen erläutert.

5.1. Grobentwurf

Die Verarbeitung eines Textes kann nicht als atomare Aufgabe angesehen und mit einem Schritt gelöst werden. Um die Informationsgewinnung und -verarbeitung möglichst effizient zu gestalten, wurden verschiedene NLP-Werkzeuge miteinander kombiniert und anschließend die ermittelten Ergebnisse gegen digitales Allgemeinwissen in Form von Ontologien geprüft.

AutoAnnotator besteht im Wesentlichen aus sechs Komponenten (-gruppen). Abbildung 11 verdeutlicht den Aufbau: Die Klasse `AutoAnnotator` (1) läd die Konfiguration[13] und den Eingabetext. Anschließend initialisiert sie die übrigen Komponenten und startet die Ablaufumgebung. Die Ablaufumgebung (2) ist die zentrale Komponente der Programmarchitektur; sie verwaltet nicht nur das zu bearbeitende Dokument, sondern ermöglicht auch das sukzessive Aufrufen verschiedener Komponenten. Die eingebundenen Programme werden kettenartig hintereinander gereiht und ausgeführt; so stehen in einem Verarbeitungsschritt nicht nur die Basisinformationen zur Verfügung, sondern bereits die Ergebnisse der vorangegangenen Schritte. Informationslieferanten (3) und -verarbeiter (4) müssen lediglich eine Schnittstelle implementieren. Sie können dann in

[13] Die von AutoAnnotator verwendeten Komponenten müssen parametrisiert werden. Darüber hinaus lassen sich einige Feineinstellung für die Informationsverarbeitung durch den Benutzer festlegen. Details zur Konfiguration findet sich in Anhang G sowie im aktuellen Kapitel.

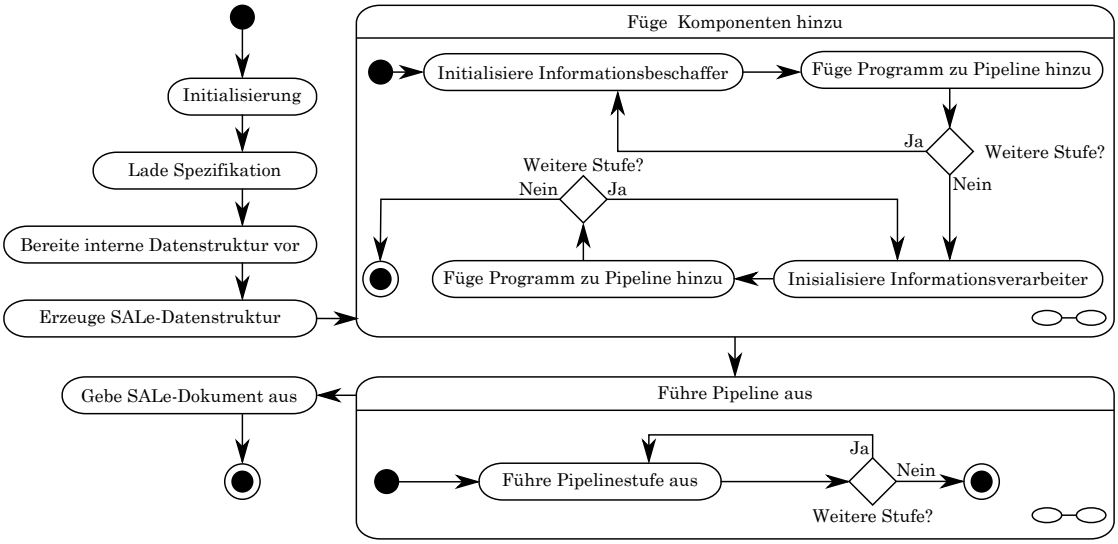


Abbildung 12: AutoAnnotator – Der Programmablauf.

die Ablaufumgebung eingebunden werden, verfügen automatisch über eine Anbindung zum Dokument und werden von der Ablaufumgebung aufgerufen. Das Dokument (5) umfasst den eingelesenen Spezifikationstext sowie das zu erzeugende SAL$_E$-Dokument. Die grafische Benutzerobefläche (6) wird für Rückfragen und Fehlermeldungen verwendet.

Abbildung 12 beschreibt den generellen Programmablauf: Zunächst wird die gesamte Umgebung initialisiert und die Eingabedatei eingelesen. Die Informationsquellen und -verarbeiter werden nacheinander initialisiert und der Ausführungsumgebung hinzugefügt. In dieser werden den einzelnen Komponenten das bis dahin aufgebaute Dokument übergeben, damit diese weitere Informationen hinzufügen bzw. Analysen durchführen können. Einige Verarbeitungsstufen fügen hierbei neue Informationen hinzu, andere verarbeiten diese Informationen oder prüfen die Ergebnisse. Am Ende der Verarbeitungskette steht die Ausgabe der Analyseergebnisse als SAL$_E$-Dokument.

5.2. Datenmodelle

In der Welt von SAL$_E$ MX dominieren Graphen als Informationsspeicher und Arbeitsgrundlage. Als Graphersetzungssystem dient SAL$_E$ MX das am Karlsruher Institut für Technologie entwickelte GrGen.NET [GBG+06]. Anfänglich war die Entwicklung von AutoAnnotator darauf ausgerichtet, zunächst eine Zwischenrepräsentation zu erstellen und diese dann in GrGen.NET zu laden; anschließend sollte der Graph (mit zusätzlichen Informationen angereichert) als Grundlage für Abfragen dienen und am Ende serialisiert werden. Im Laufe der Entwicklung zeigte sich jedoch, dass neben dem zu erstellenden SAL$_E$-Graphen auch ein flexibler Datenspeicher für die Analyseergebnisse der eingebundenen Programme benötigt wurde. Da GrGen.NET ein vordefiniertes Modell der zu erzeugenden Graphen benötigt, müsste entweder für jede Informationsquelle das Graphmodell entsprechend erweitert werden oder ein hinreichend generisches Modell er-

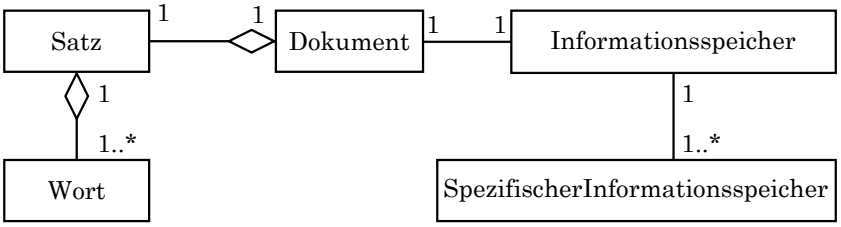


Abbildung 13: Das Klassendiagramm des Dokuments (vereinfacht).

stellt werden. Beide Möglichkeiten schienen zu wartungsintensiv und nicht zielführend, da die entwickelte Lösung dann auf GrGen.NET als Datenspeicher angewiesen wäre. Eine Anbindung von GrGen.NET an ein Programm wäre mithilfe seiner API möglich; diese steht jedoch nur für Microsoft .NET [Mic] zur Verfügung. Da alle eingebundenen Programme und AutoAnnotator selbst in Java implementiert sind, würde dies zu einer Verkomplizierung der Architektur führen. Das Vorhaben, für AutoAnnotator ebenfalls einen Graphen als internes Modell zu verwenden, wurde aufgrund dieser Schwierigkeiten fallengelassen. Stattdessen kommen zwei Datenstrukturen zum Einsatz, die mithilfe von Java implementiert wurden. Neben der hohen Flexibilität und der einfachen Erweiterbarkeit führt dies dazu, dass mit Java-Bordmitteln eine Verbindung zwischen Anforderungsdokument und Sal_E-Modell hergestellt werden konnte (vergleiche dazu Abbildung 15).

Das erste Datenmodell, zu sehen in Abbildung 13 und im weiteren *Dokument* genannt, erfasst zunächst die Spezifikation selbst. Im Laufe der Verarbeitung werden dann die verschiedenen gewonnenen Informationen zum Informationsspeicher des Dokuments hinzugefügt. Dies führt insbesondere dazu, dass bereits ermittelte Informationen allen folgenden Verarbeitungsschritten zur Verfügung stehen – und zwar nicht nur den Informationsverarbeitern, sondern auch den Informationsquellen. So können beispielsweise die von einem POS-Tagger ermittelten Tags als zusätzliche Eingabe für einen Parser dienen und müssen nicht doppelt ermittelt werden. Das Dokument entspricht einem Diskursmodell oder einer Tiefenstruktur.

Das zweite Datenmodell, zu sehen in Abbildung 14, ist das Sal_E-Modell, welches im Wesentlichen einen Sal_E-Graphen repräsentiert (vergleiche [GT07] oder Anhang B). Hier werden Sal_E-spezifische Informationen wie *shifting* und Referenzen abgebildet. Dieses Sal_E-Modell wird im Laufe der Verarbeitung auf- und umgebaut und schlussendlich als Sal_E-Dokument ausgegeben; es entpricht einem externen Modell bzw. einer Oberflächenstruktur. Die Klassen des Sal_E-Modells entsprechen weitestgehend den verfügbaren Sal_E-Konzepten. Wie Abbildung 15 zeigt, verfügen sie zusätzlich zu den Sal_E-Attributen über eine bidirektionale Verbindung zu den Elementen des Dokuments, aus denen sie erstellt wurden. Dies ermöglicht eine effiziente Navigation zwischen beiden Welten und somit eine einfache Zuordnung von neuen Informationen.

Das Sal_E-Modell erlaubt es darüber hinaus, einen Knoten auszutauschen oder Knoten zu verschmelzen. Die Informationen (wie korrespondierende Elemente im Dokument etc.) werden dann im neuen Knoten zusammengeführt. Nach Abschluss der Informati-

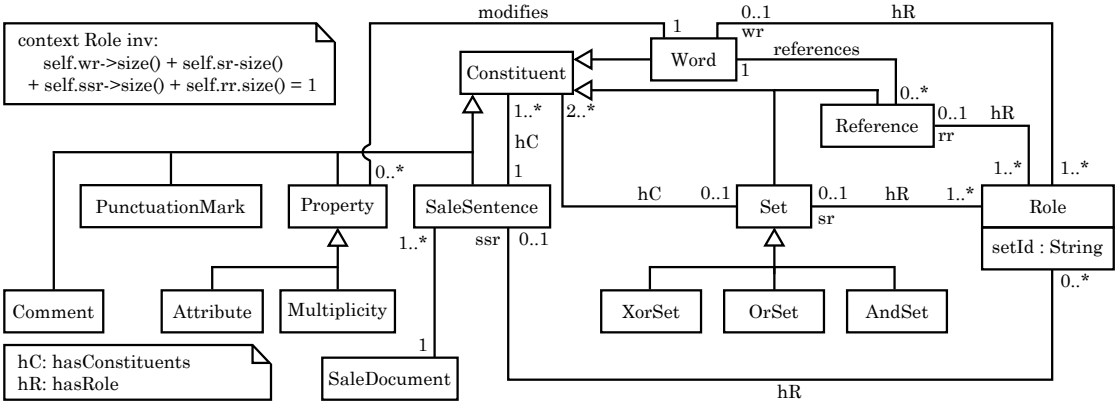


Abbildung 14: Das Klassendiagramm des SAL$_E$-DOM (vereinfacht).

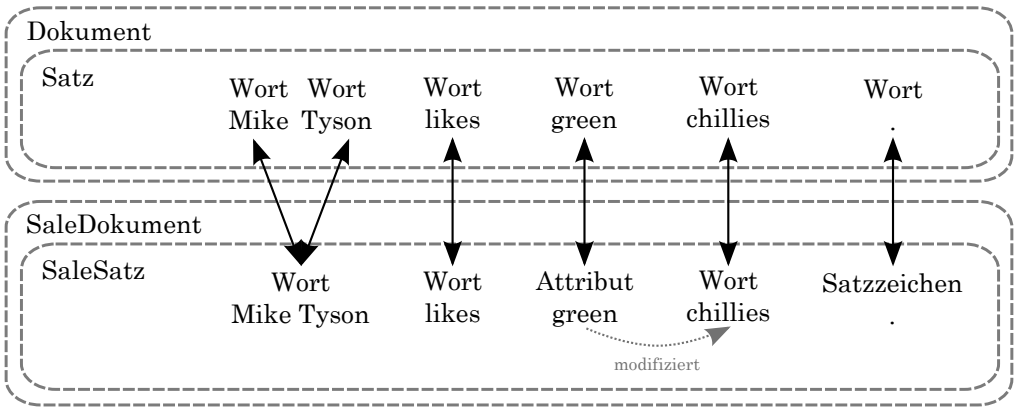


Abbildung 15: Ein Beispiel für ein Dokument mit zugehörigem SAL$_E$-Dokument.

onsverarbeitung wird das Modell serialisiert und in einer Textdatei gespeichert.

5.3. Komponenten in der Pipeline

In diesem Abschnitt wird die in Abschnitt 5.1 und Abbildung 16 grob skizzierte Pipeline genauer erläutert. AUTOANNOTATOR verfügt über eine Verarbeitungsumgebung, in welche die verwendeten Komponenten geladen und darin nacheinander ausgeführt werden. Hierbei stellt die Verarbeitungsumgebung sicher, dass die Programme in der gegebenen Reihenfolge ausgeführt werden und über die Informationen der vorangegangenen Stufen verfügen. Grundsätzlich werden die Informationen von Programmpaaren hinzugefügt und verarbeitet – eine Information wird an (mindestens) zwei Stellen in der Pipeline verwendet. Sie wird zunächst von einem Informationsbeschaffer ermittelt, der entweder die Information selbst bereitstellen (wie bspw. der Sentence Chunker), ein Programm über seine API verwenden (bspw. die Schnittstellenklasse zum Stanford'schen Parser) oder einen externen Server abfragen (bspw. WordNet) kann. Wo verfügbar, wurden in

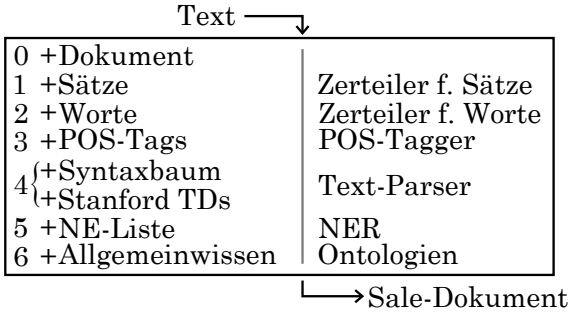


Abbildung 16: Die Verarbeitungspipeline von AutoAnnotator.

AutoAnnotator serverbasierte Programme angebunden, um den lokalen Rechner zu entlasten und Programmkonflikten aus dem Weg zu gehen. Wenn keine Serverimplementierung vorhanden war, wurde vorzugsweise eine Java-API verwendet, um das (oft für menschliche Leser aufbereitete) Ergebnis eines Analyseprogramms nicht erneut parsen zu müssen. Nachdem alle Informationsbeschaffer ihre Arbeit abgeschlossen haben, werden die zugehörigen Informationsverarbeiter aufgerufen.

AutoAnnotator ist nicht auf spezielle Programme angewiesen, sondern kann flexibel angepasst werden. Die POS-Tags werden beispielsweise mithilfe des Stanford POS-Taggers gewonnen. Solange ein anderes Programm die gleichen Informationen (oder mehr) bereitstellen kann, ist ein Austausch problemlos möglich. Da die Stufen jedoch teilweise aufeinander aufbauen, ist bei einer Modifikation darauf zu achten, dass der gewonnene Informationsumfang nicht beschnitten wird. Durch eine Verzahnung der Komponenten konnte teilweise Rechenzeit gespart werden: Für das Erstellen eines Syntaxbaums benötigt der Stanford'sche Parser die POS-Tags. Diese werden üblicherweise vor der Verarbeitung automatisch bestimmt. Da diese jedoch schon bekannt sind, können sie dem Parser übergeben werden, der daraufhin diesen Schritt überspringt.

Für die vorliegende Arbeit wurden die angebotenen Standardversionen der Programme verwendet. Parser und Tagger wurden nicht speziell trainiert und sowohl WordNet als auch ResearchCyc enthalten ihre Standard-Informationen. Eine denkbare Anpassung wäre das Ausweichen auf speziell trainierte Modelle. Gegebenenfalls lassen sich durch die Verwendung von Domänenontologien bessere Ergebnisse erzielen. Eine Evaluation dieser Anpassungen ist jedoch nicht Bestandteil dieser Arbeit.

In den folgenden Abschnitten werden die eingesetzten Programme und die mit ihrer Hilfe gewonnenen Informationen beschrieben; eine Übersicht über die eingesetzten Komponenten gibt Tabelle 5. Im Anschluss wird gezeigt, wie das SalE-Modell erstellt bzw. angepasst wird. Als durchlaufendes Beispiel dienen die Sätze: *Chillies are hot vegetables. Mike Tyson likes green chillies. Last week he ate five of them.*

5.3.1. Aufbau des Dokuments

Am Anfang steht das Einlesen des Textdokuments, welches gewissen Einschränkungen unterliegt: Sowohl der vorgestellte Ansatz, als auch SalE MX ist auf die Verarbeitung von

Programm	**Plattform**	**Verfügbar für Sprache(n)**	**Voraussetzungen Besonderheiten**	**Quellen**
Sentence Chunker	Java	–	keine	
Word Chunker	Java	–	keine	
Stanford POS-Tagger	Java 1.5+, API-Zugriff	Englisch, Arabisch, Chinesisch und Deutsch	trainierbar	[MJ, The09a]
Stanford Parser	Java 1.5+, API-Zugriff	Englisch, Chinesisch, Deutsch und Arabisch	trainierbar	[MJ, The09b, MM08a, MM08b]
Stanford NER	Java 1.5+ Server verfügbar	sprachunabhängig	trainierbar	[MJ, FGM05]
JavaRAP	Java 1.5	Englisch	benötigt den Charniak Parser	[QKC04]
Charniak Parser	C++, Linux nativ, Windows via cygwin	Englisch		[Cha00]
WordNet	Server + Java-API	Englisch		[Mil09, Mil95, Fel98]
Research Cyc	Server + Java-API	Englisch		[Cycb, Cyca]
Tree Tagger	Standalone Java-Wrapper	Deutsch, Englisch, Französisch u.a.m.	trainierbar	[Sch94, Sch95, Cas09]

Tabelle 5: Übersicht über die integrierten Programme.

Text beschränkt. Daher ist das Eingabeformat ein einfaches Textdokument. Da Grafiken oder Abbildungen weder später verarbeitet, noch in die Prozesse eingegeben werden können, sollte der textuelle Inhalt ohne Informationen aus (eventuell den Anforderungsdokumenten zusätzlich beigefügten) Abbildungen auskommen. Ebenso werden Aufzählungen und dergleichen bei der Verarbeitung nicht berücksichtigt; um unerwünschte Ergebnisse zu vermeiden, sollten derartige Elemente vorher aus dem Text entfernt werden. Da der Ansatz auf der Analyse von Sätzen basiert, sind Aufzählungen von Stichpunkten und Ähnliches nicht sinnvoll.

Da sich die meisten Informationen auf Worte beziehen, oder im Kontext eines Satzes gelten, muss der Text zunächst aufgespalten werden. Zuerst wird der eingelesene Text mithilfe des Java-BreakIterators in Sätze gespalten. Der BreakIterator ist Bestandteil der Java-Bibliothek `java.text` und verfügt über verschiedene sprachabhängige Implementierungen, die bei der Initialisierung durch die Wahl einer `locale` ausgewählt werden können. Alternativ könnte man in der Eingabedatei pro Satz genau eine Zeile verwenden und somit über die Zeilen eine Aufspaltung in Sätze erreichen. Die ermittelten Sätze werden dann zusammen mit ihrer Position im Dokument gespeichert.

Anschließend werden die Sätze in einzelne Worte aufgespalten. Hierzu kommt ebenfalls der Java-BreakIterator zum Einsatz, der neben einer Aufspaltung in Sätze auch eine Aufspaltung von Sätzen in einzelne Worte vornehmen kann. Die ermittelten Worte werden dann in die jeweiligen Sätze eingegliedert.

Für beide Spaltungsoperatoren ist kein Training der Software notwendig; möchte man statt englischer Texte andere Sprachen verarbeiten, muss lediglich eine andere `locale` gewählt werden.

Der folgende Quelltext verdeutlicht die Struktur des Dokuments in unserem Beispiel[14]:

```
Dokument
 |--> Satz: Chillies are hot vegetables.
 | |--> Worte: Chillies, are, hot, vegetables, .
 |--> Satz: Mike Tyson likes green chillies.
 | |--> Worte: Mike, Tyson, likes, green, chillies, .
 |--> Satz: Last week he ate five of them.
 | |--> Worte: Last, week, he, ate, five, of, them, .
```

Neben der internen Repräsentation des Textes wird ein initiales SAL$_E$-Dokument erzeugt. Hierzu werden lediglich die SAL$_E$-Sätze angelegt, dem SAL$_E$-Dokument hinzugefügt und Referenzen auf die ursprünglichen Sätze registriert. Anschließend werden alle Worte aus dem Dokument als SAL$_E$-Kommentare in die SAL$_E$-Sätze eingefügt und die jeweiligen Worte als Referenz registriert. So entsteht eine eng gekoppelte SAL$_E$-Repräsentation des Dokuments, die im weiteren Verlauf der Informationsextraktion umgebaut bzw. erweitert werden kann.

[14] Satzzeichen sind ebenfalls Worte in einem Satz; der Begriff *Wort* ist an dieser Stelle synonym mit dem Begriff *token* verwendet.

Würde man das SALE-Dokument an dieser Stelle ausgeben, erhielte man den folgenden Quelltext:

```
#{Chillies are very hot vegetables.}
[ #Chillies #are #very #hot #vegetables ].

#{Mike Tyson likes green chillies.}
[ #Mike #Tyson #likes #green #chillies ].

#{Last week, he ate five of them.}
[ #Last #week, #he #ate #five #of #them ].
```

Quelltext 1: Mike Tyson Beispiel (Schritt 1, nur Kommentare).

5.3.2. POS-Tagger

Als erstes Programm in der Pipeline wird ein POS-Tagger aufgerufen, welcher die grundlegenden syntaktischen Informationen zu den Worten hinzufügt. Zum Einsatz kommt der Stanford Log-linear Part-Of-Speech Tagger, welcher für beliebige Sprachen trainiert werden kann [MJ, The09a]. Zum Download stehen derzeit trainierte Modelle für Englisch, Arabisch, Chinesisch und Deutsch bereit. Da der Tagger vollständig als Java-Implementierung vorliegt, kann er über seine Schnittstellen direkt im Programmcode von AutoAnnotator verwendet werden. Die zusätzliche Verarbeitung einer textuellen Ausgabe entfällt somit gänzlich. Die POS-Tags werden direkt in den Wort-Elementen des Dokuments gespeichert und stehen so unmittelbar zur Verfügung.

Unsere Beispielsätze werden mit den folgenden Informationen angereichert:

```
Chillies/NNS are/VBP very/RB hot/JJ vegetables/NNS ./.
Mike/NNP Tyson/NNP likes/VBZ green/JJ chillies/NNS ./.
Last/JJ week/NN ,/, he/PRP ate/VBD five/CD of/IN them/PRP ./.
```

Die in diesem Beispiel auftretenden POS-Tags sind in Tabelle 6 aufgeführt; in Anhang D findet sich eine vollständige Liste des Penn Tagsets. Betrachtet man die Ausgabe des zweiten Satzes, so erkennt man zuerst die beiden Namensbestandteile, die mit dem POS-Tag NNP als Nomen im Singular gekennzeichnet wurden. Das Verb *likes* ist mit `VBZ` gekennzeichnet, was für ein Verb in der dritten Person Singular des Präsens steht. *green* ist ein Adjektiv und daher mit `JJ` gekennzeichnet. Die *chillies* sind ein Nomen, welches im Plural steht; daher sind sie mit `NNS` gekennzeichnet. Der Punkt am Satzende ist ebenfalls mit einem Tag, dem Punkt (`.`) versehen.

5.3.3. Parser für natürliche Sprache

Im folgenden Schritt werden die Sätze von einem Parser verarbeitet, der nicht nur eine Ausgabe mit POS-Tags erzeugt, sondern Sätze in eine Baumstruktur überführt. Hierfür kommt der statistische Stanford Parser zum Einsatz [MJ, The09b]. Dieser erstellt neben der Baumstruktur auch eine Liste von sogenannten *Typed Dependencies*, welche die Informationen im Satz repräsentieren [MM08a, MM08b].

Tag	Bedeutung	Tag	Bedeutung
CD	Kardinalzahl	RB	Adverb
IN	Präposition oder Konjunktion	VBD	Verb, Vergangenheit (past tense)
JJ	Adjektiv	VBP	Verb, nicht 3. Pers. Sing. Präsens
NNP	Nomen, Singular	VBZ	Verb, 3. Pers. Sing. Präsens
NNS	Nomen, Plural	,	Komma
PRP	Personalpronomen	.	Punkt am Satzende

Tabelle 6: Penn POS-Tags (Auszug).

Die ermittelten Informationen sind in Tabelle 7 zusammengefasst. Im Folgenden wird das Verarbeitungsergebnis anhand des ersten Satzes erläutert. Zunächst sieht man auf der linken Seite der Tabelle den Syntaxbaum, der vom Parser erstellt wurde. Unterhalb des Wurzelknotens (`ROOT`) befindet sich ein Satzknoten (`S`), in welchem die Worte des Satzes als Phrasen enthalten sind. Die erste Phrase ist eine Nominalphrase[15] (engl. *Noun Phrase*, `NP`), die aus dem Nomen *Chillies* im Plural besteht (`NNS`). Die zweite Phrase ist eine geschachtelte Verbalphrase[16] (engl. *Verb Phrase*, `VP`). Sie enthält zunächst das Verb *are*, welches mit der Kennzeichnung `VBP` für Verbenversehen ist, die nicht in der dritten Person Singular des Präsens stehen. Weiterhin sind innerhalb der Verbalphrase die *very hot vegetables* enthalten. Diese stehen in einer Nominalphrase, welche neben den *vegetables* aus einer Adjektivphrase besteht. Die Adjektivphrase selbst besteht aus dem Adverb *very* und dem Adjektiv *hot* (versehen mit den Tags `RB` bzw. `JJ`). Als nächstes Kind des Satzknotens findet sich der Punkt als Satzzeichen. Auf der rechten Seite der Tabelle sieht man die Typed Dependencies, welche die Satzstrukturen indirekt wiedergeben; die Zahlen hinter den Worten geben die Position des Wortes im Satz an: Zunächst betrachten wir die Abhängigkeit `amod(vegetables-5, hot-4)`; sie kennzeichnet ein Adjektiv *hot* und das dadurch beschriebene Element *vegetables*. Die Abhängigkeit `advmod(hot-4, very-3)` beschreibt die Beziehung zwischen *hot* und dem Adverb *very*. Die oberen beiden Abhängigkeiten beschreiben eine Kopula[17]: Dass *Chillies vegetables* sind, lässt sich der Abhängigkeit `nsubj(vegetables-5, Chillies-1)` entnehmen, welche ein nominales Subjekt beschreibt. Die Abhängigkeit `cop(vegetables-5, are-2)` verbindet das kopulative Verb mit dem nominalen Prädikat *vegetables*.

[15] Eine Nominalphrase (NP) ist eine Phrase, deren Kopf ein Nomen (ein Substantiv oder nominalisiertes Verb oder Adjektiv) ist. Neben dem Substantiv kann die Nominalphrase auch Adjektive etc. enthalten und wird meist von einem Determinativ (das/ein/mein/dieses ... Haus) eingeleitet [ZHS97a, B2 1.1][ZHS97c, G1]. Beispiel (Nominalphrase kursiv): *Das alte Haus* wird abgerissen.

[16] Eine Verbalphrase (VP) oder Verbalgruppe bezeichnet eine Phrase, deren Kopf ein Verb ist. Zu ihr gehören neben dem Verb auch seine Argumente und gegebenenfalls Supplemente (Adjektive etc.) dieser Argumente [ZHS97a, B2 2]. Beispiel (Verbalphrase kursiv): Ich freue mich, weil *Hans mir sein neues Auto leiht.*

[17] Eine Kopula ist ein Prädikatsausdruck. Sie schreibt einer Sache eine Eigenschaft zu oder ordnet sie in eine Gruppe ein [ZHS97a, D3 8][ZHS97b, E2 2.4.3]. Beispiele (Kopula kursiv): Das Auto *ist blau.* Das Auto *ist ein Ferrari.*

Chillies are very hot vegetables.	

(ROOT
  (S
  (NP (NNS Chillies))
  (VP (VBP are)
  (NP
  (ADJP (RB very) (JJ hot))
  (NNS vegetables)))
  (. .)))
	nsubj(vegetables-5, Chillies-1)
	cop(vegetables-5, are-2)
	advmod(hot-4, very-3)
	amod(vegetables-5, hot-4)

Mike Tyson likes green chillies.	

(ROOT
  (S
  (NP (NNP Mike) (NNP Tyson))
  (VP (VBZ likes)
  (NP (JJ green) (NNS chillies)))
  (. .)))
	nn(Tyson-2, Mike-1)
	nsubj(likes-3, Tyson-2)
	amod(chillies-5, green-4)
	dobj(likes-3, chillies-5)

Last week he ate five of them.	

(ROOT
  (S
  (NP (JJ Last) (NN week))
  (NP (PRP he))
  (VP (VBD ate)
  (NP
  (NP (CD five))
  (PP (IN of)
  (NP (PRP them)))))
  (. .)))
	amod(week-2, Last-1)
	tmod(ate-4, week-2)
	nsubj(ate-4, he-3)
	dobj(ate-4, five-5)
	prep(five-5, of-6)
	pobj(of-6, them-7)

Tabelle 7: Die Ergebnisse des Stanford'schen Parsers.

5.3.4. Named Entity Recognition

Als erstes *externes* Programm wird der Named Entity Recognizer (NER) aufgerufen. Die Standardversion, die von der Stanford NLP Group herausgegeben wird, enthält bereits eine Serverversion. Sie kann über eine einfache Socketverbindung angesprochen werden. Der NER basiert auf linearen CRF Sequenzmodellen (*linear chain Conditional Random Field (CRF) sequence models*). Die verfügbaren Modelle wurden mit englischen Texten trainiert; es können jedoch mit entsprechenden Trainingsdaten Modelle für andere Sprachen erzeugt werden. Zwar liegt dieses Programm ebenfalls als Java-Implementierung vor, jedoch überschneiden sich die Implementierungen der Stanford'schen Programme derart, dass ein gemeinsames Laden und Ausführen der drei Programme in einer virtuellen Maschine nicht möglich ist. Da der NER eine einfache Ausgabe und eine integrierte Serverversion besitzt, wurde er extern eingebunden, um diesen Konflikt aufzulösen.

Der NER versieht die Elemente in den Sätzen erneut mit Tags; diese sind `Person`, `Organization`, `Location` und `O` (keine Entität erkannt). Der einzige Beispielsatz, der neben `O` andere Tags enthält ist der zweite:

```
Mike/PERSON Tyson/PERSON likes/O green/O chillies/O ./O
```

Diese Information wird aggregiert, als Liste zusammengefasst und im Dokument gespeichert.

5.3.5. Auflösen von Anaphern

JavaRAP ermöglicht das Auflösen von (satzübergreifenden) Anaphern. Es steht als plattformunabhängige Java-Implementierung bereit und wurde über seine API eingebunden. Nach der Ausführung stellt das Programm eine Liste von Anapher-Antezendens-Paaren bereit. Diese werden dann im Dokument ermittelt und die sich ergebenden Zusicherungen zum SAL_{E}-Dokument hinzugefügt. Unglücklicherweise decken sich die Elemente der Paare nicht notwendigerweise mit den Konstituenten des Dokuments:

```
(1,0) Mike Tyson <-- (2,3) he
(1,3) green chillies <-- (2,7) them
```

Der Ausgabe lässt sich entnehmen, dass sich *he* auf *Mike Tyson* bezieht und *them* auf *green chillies*. Im Falle von *Mike Tyson* ist dies äußerst praktisch, da *Mike Tyson* ohnehin aus den beiden Einzelworten zusammenzufassen ist. Bei den *green chillies* verhält es sich jedoch anders: Hier handelt es sich nicht um einen Konstituenten, der aus mehreren Worten besteht, sondern um zwei getrennt zu behandelnde Einheiten. Hier ist es nicht zielführend, einen Konstituenten zu suchen, der aus den Elementen *green* und *chillies* besteht.

JavaRAP benötigt den Parser von Eugene Charniak [Cha00]; Details zu dessen Installation und der Konfiguration von JavaRAP finden sich in Anhang H.

5.3.6. Ermitteln von Wortstämmen

Die erste Ontologie, die in AutoAnnotator eingebunden wird, ist WordNet. Anfragen an allgemeine Ontologien, die nicht in der Lage sind, gebeugte Worte auf deren Grundform zurückzuführen, können nur dann zielführend formuliert werden, wenn man die Grundform der betroffenen Worte kennt und verwendet.

WordNet enthält nur Worte der sogenannten offenen Wortklassen: Nomen, Verben, Adjektive und Adverben. Um eine Grundform zu ermitteln, kann man eine Anfrage mit der gebeugten Form des Wortes an WordNet stellen; man erhält als Antwort eine Liste von Grundformen. Da es zu Mehrdeutigkeiten kommen kann, ist es sinnvoll, den Suchraum einzuschränken: Im Falle von *ate* in unserem Beispiel würde eine unparametrisierte Suche nach der Grundform zu mehreren Ergebnissen führen: Zum einen erhält man die erwartete Grundform *to eat* des Verbs; andererseits liefert WordNet *Ate* als Grundform eines Nomens. Dies mag auf den ersten Blick überraschend sein; bei genauerer Betrachtung dieses Falles stellte sich jedoch heraus, dass Ate als eine griechische Göttin [18] in WordNet registriert ist. Parametrisiert man die Anfrage zusätzlich mit der Wortart, so kann der Suchraum eingeschränkt und derartige Ergebnisse vermieden werden. Da WordNet nur Wörter der offenen Wortklassen enthält, verfügt es über eigene POS-Tags. Die zuvor ermittelten POS-Tags können daher nicht direkt verwendet werden. Eine einfache Abbildung der POS-Tags auf WordNet-Tags ermöglicht jedoch die Parametrisierung der Anfragen: Da bekannt ist, dass *ate* ein Verb in der Vergangenheitsform ist (POS-Tag `VBD`), liefert WordNet lediglich die Grundform des Verbes als Antwort auf unsere Abfrage.

Die Grundform wird für alle Worte ermittelt, für die ein gültiges [19] WordNet-Tag ermittelt werden kann.

5.4. Informationsextraktion

Nachdem die Informationslieferanten abgearbeitet wurden, steht die Verarbeitung der gelieferten Informationen an. Die Zerteilung des Eingabetextes in einzelne Sätze ist der erste Schritt zum gültigen Sal_{E}-Dokument. Da nun alle Sätze zur Verfügung stehen, können die Informationen aus dem internen Modell ausgewertet werden; das Sal_{E}-Dokument wird nur noch umgebaut und angereichert.

5.4.1. Auskommentieren von Füllworten

Nach dem Dokumentaufbau sind alle Worte (außer den Satzzeichen) als Kommentare im Sal_{E}-Dokument hinterlegt. Diese Vorgehensweise führt dazu, dass – streng genommen – keine Kommentare, sondern nicht-Kommentare gefunden werden müssen. Auf der anderen Seite wird dadurch selbst bei einer unvollständigen Erkennung der Satzelemente ein syntaktisch korrekter Sal_{E}-Satz ausgegeben. AutoAnnotator verfügt über keine Komponente zur Identifikation von „sicheren“ Kommentaren, Kommentare

[18] Ate ist als griechische Göttin erfasst. Sie ist eine Tochter des Zeus und stürzt Menschen ins Unheil.

[19] Aufgrund der Einschränkung auf die vier Wortklassen bleiben die übrigen Worte ohne erkanntes WordNet-Tag.

sind vielmehr Nebenprodukte bei der Erkennung anderer Strukturen. Betrachtet man beispielsweise den ersten Beispielsatz, so sieht man in Tabelle 7 die Struktur der Kopula: `nsubj(vegetables-4, Chillies-1)` und `cop(vegetables-4, are-2)`. Nachdem AutoAnnotator festgestellt hat, dass es sich um eine Kopula handelt, wird die Information mithilfe der Rollen `FIN` und `FIC` kodiert. Das Verb *are* trägt danach keine Information mehr und kann als sicherer Kommentar gekennzeichnet werden.

5.4.2. Ermitteln von Subphrasen

Durch die grammatikalische Struktur, die vom Parser erzeugt wurde, können Subphrasen ermittelt werden. Betrachtet man das Beispiel *If the sun shines, plants grow*, so erhält man den folgenden Syntaxbaum:

```
(ROOT
  (S
    (SBAR (IN If) (S (NP (DT the) (NN sun)) (VP (VBZ shines))))
    (, ,)
    (NP (NNS plants))
    (VP (VBP grow))
    (. .)))
```

Der Knoten, der mit `SBAR` gekennzeichnet ist, enthält den Teilbaum mit der Bedingung für das Pflanzenwachstum. Die zugehörigen Blätter dieses Teilbaumes sind *If*, *the*, *sun* und *shines* und bilden die Konstituenten der Subphrase.

AutoAnnotator geht auf der Suche nach Subphrasen den gesamten Syntaxbaum ab. Trifft er bei der Analyse eines Satzes auf eine derartige Konstruktion, wird eine Subphrase angelegt, die alle Blätter des identifizierten Teilbaumes enthält. Die Analyse wird dann mit den anderen Knoten des Baumes weitergeführt, bis er vollständig betrachtet wurde.

Da Subphrasen in SalE Konstituenten sind, müssen sie eine Rolle einnehmen. AutoAnnotator versucht zunächst zu ermitteln, ob die Rollen für Bedingungen (`SUM`) oder Begründungen (`CAU`) anwendbar sind. Hierfür wird geprüft, ob die Teilphrase mit bestimmten Worten (bspw. *if*) oder Wortfolgen (bspw. *even though*) beginnt. Kann auf diese Weise keine Rolle ermittelt werden, wird der Benutzer wie in Abbildung 17 gebeten, der Subphrase eine Rolle zuzuweisen. Wird auch hier keine Rolle vergeben, so verwendet AutoAnnotator eine Platzhalterrolle, die von SalE MX nicht ausgewertet wird und keinen Syntaxfehler in SalE erzeugt.

5.4.3. Verbinden mehrwortiger Elemente

Da Bezeichner in SalE keine Leerzeichen enthalten dürfen, müssen Begriffe verbunden werden, die aus mehreren Worten bestehen.

Auf der einen Seite handelt es sich hierbei um zusammengesetzte Nomen und Eigennamen (Vor- und Nachnamen). Derartige Nomen werden von den typed dependencies

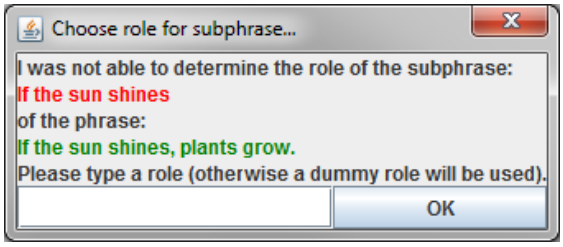


Abbildung 17: Der Zuweisungsdialog für die Eingabe einer Rolle für eine Subphrase.

als sogenannte *compound nouns* erfasst und mit der Abhängigkeit `nn(a,b)` gekennzeichnet. Betrachtet man *Mike Tyson* im zweiten Beispielsatz, so sieht man, dass der Name folgendermaßen verbunden wird : `nn(Tyson-2, Mike-1)` (vergleiche Tabelle 7). AUTOANNOTATOR verschmilzt darauf hin die beiden Worte im SAL$_E$-Dokument. Betrachtet man weiterhin den zweiten Beispielsatz, so wird deutlich, dass sich weitergehende Informationen nicht auf *Mike Tyson* beziehen, sondern nur auf das einzelne Wort *Tyson*:

```
nn(Tyson-2, Mike-1)
nsubj(likes-3, Tyson-2)
amod(chillies-5, green-4)
dobj(likes-3, chillies-5)
```

Die Information, aus welchen Worten des Dokuments das neue SAL$_E$-Wort *Mike Tyson* entstanden ist, bleibt dabei erhalten: Es handelt sich um die Einzelworte *Mike* und *Tyson*, was auch in Abbildung 15 zu sehen ist. So ist es möglich, zu einem späteren Zeitpunk das richtige SAL$_E$-Wort zu einem ursprünglichen Wort zu ermitteln.

Andere Begriffe, die zusammengefasst werden müssen, sind Mengenangaben. Betrachtet man beispielsweise ein Kuchenrezept, so könnte man lesen: *One takes 5 cups of sugar.* In diesem Fall möchte man erreichen, dass die Annotation aussagt, dass man *sugar* verwendet, und zwar genau *5 cups*:

```
1 [ One|AG takes|AG *5_cups #of sugar|PAT ].
```

Auch in diesem Fall gibt es eine typed dependency (`num(cups-4, 5-3)`), die den Umstand ausdrückt, dass *5 cups* eine zusammengesetzte Mengenangabe ist. Diese typed dependency wird ebenso aufgelöst, wie `nn(a,b)` oben, nämlich durch die Konkatenation beider Elemente mit einem Unterstrich.

Würde man das SAL$_E$-Dokument an dieser Stelle ausgeben, erhielte man den folgenden Quelltext:

```
1 #{Chillies are very hot vegetables.}
2 [ #Chillies #are #very #hot #vegetables ].
3
4 #{Mike Tyson likes green chillies.}
5 [ #Mike_Tyson #likes #green #chillies ].
6
7 #{Last week, he ate five of them.}
```

```
8 [ #Last #week, #he #ate #five #of #them ].
```

Quelltext 2: Mike Tyson Beispiel (Schritt 2, mehrwortige Elemente zusammengefasst).

Geändert hat sich bis zu diesem Zeitpunkt lediglich *Mike Tyson*, der nun mit einem _ zu einem Wort verbunden wurde.

5.4.4. Ermitteln von Modifikatoren sowie den Bezugsworten

Modifikatoren im Sinne von SALE sind Multiplizitäten und Attribute, analog zu den Konzepten in UML. Um einen Modifikator korrekt in einem SALE-Dokument abbilden zu können, muss man zunächst unterscheiden, ob es sich um eine Multiplizität handelt, oder um ein Attribut. Dabei nur darauf zu achten, ob es sich um eine Ziffernfolge handelt oder nicht, ist jedoch nicht ausreichend. So sind *500g* als Multiplizität zu sehen und *fünf* ebenso.

Unbestimmte Modifikatoren treten häufig als Determinativ (engl. *determiner*) auf. Ist dies der Fall, werden sie von der typed dependency `det(a,b)` erfasst: Zu *Every pallet must be returned after transport* liefert der Parser die Abhängigkeit `det(pallet-2, Every-1)`. AUTOANNOTATOR prüft daraufhin, ob es sich um eine registrierte Mengenangabe handelt. Falls ja, wird eine Multiplizität erzeugt und mit dem zugehörigen Element verknüpft. Kann das Determinativ nicht auf eine Mengenangabe zurückgeführt werden, wird es als sicherer Kommentar markiert. Dies ist häufig dann der Fall, wenn es sich um einen (un-) bestimmten Artiken handelt, wie *the* oder *a*.

Betrachtet man hingegen eine Apposition wie in *Bill Gates, 55, is a co-founder of Microsoft*, so ist *55* keinesfalls als Multiplizität zu betrachten, sondern als (verkürzte) Altersangabe. In einem solchen Fall wird ein Attribut erzeugt und mit *Bill Gates* verknüpft.

Handelt es sich bei der Altersangabe wie in *Bill Gates is 55* um eine Kopula, so wird ebenfalls ein Attribut erzeugt. Bei der Betrachtung von Kopulae wird darüber hinaus berücksichtigt, ob es sich um eine Konstruktion mit einem Adjektiv handelt; ist dies der Fall, wird ebenfalls ein Attribut erzeugt (für Kopulae, die nicht mithilfe von Modifikatoren aufgelöst werden können, vergleiche den Abschnitt zu Kopulae in 5.4.5). Ist das Ziel des Attributes nur als Kommentar im SALE-Dokument hinterlegt, so kann keine Attributierung angelegt werden (Kommentare können in SALE nicht mit Modifikatoren versehen werden). Statt das erkannte Attribut zu verwerfen, wird das Attributziel zu einem Wort mit einer Platzhalterrolle umgewandelt. Die Platzhalterrolle `COP` wird jedoch von SALE MX nicht betrachtet, was im Zweifel dazu führt, dass das Element nicht als Klasse angelegt und das Attribut verlorengeht. Um dieses Problem zu umgehen, kann die Platzhalterrolle vom Benutzer vorgegeben werden. Eine mögliche Rolle wäre die Rolle `OBJECTROLE`, welche keine Semantik kodiert, bei der Modellextraktion jedoch zu einer Klasse führt. Da diese Rolle nicht im Rollensystem von SALE, sondern nur im zugehörigen GRGEN.NET-Graphmodell vorhanden ist, wäre ihre Verwendung strenggenommen falsch. Möglich wäre auch die Verwendung der Rolle des *agens* – quasi als konkrete Umsetzung der `OBJECTROLE`. Wählt man jedoch *agens* als Rolle, so wird für

den Anwender erschwert, erkannte Semantik von Platzhalterrollen zu unterscheiden. Eine andere Lösung wäre, die Platzhalterrolle `COP` (bspw. als Unterrolle von `OBJECTROLE`) in das Graphmodell aufzunehmen.

Im dritten Beispielsatz sieht man eine präpositionale Verknüpfung von *five* über die Präposition *of* mit *them* (den *chillies*). In derartig gelagerten Konstruktionen kann die Präposition sicher auskommentiert und eine Multiplizität erzeugt werden. Da die Multiplizität keine thematische Rolle einnehmen kann, muss gleichzeitig die Objektbeziehung vom Zahlwort *five* auf das präpositionale Objekt *them* verschoben werden.

Betrachtet man allgemeine Attribute, so sieht man in allen drei Beispielsätzen, dass Adjektive samt Bezugswort in der typed dependency `amod(a,b)` abgebildet werden. Diese Adjektive werden als Attribut gekennzeichnet und direkt mit den Elementen verbunden. Ähnlich wie bei Kopulae wird eine Platzhalterrolle (`ADJ`) verwendet, wenn das attributierte Element als Kommentar im SalE-Dokument erfasst ist.

5.4.5. Ermitteln von thematischen Rollen

In diesem Abschnitt wird beispielhaft erläutert, wie aufgrund der Informationsbasis Schlüsse auf thematische Rollen gezogen werden. Eine Liste der von AutoAnnotator betrachteten Konstruktionen findet sich in Anhang A.

Eine Konstruktion, die im Abschnitt zu den Modifikatoren bereits beleuchtet wurde, ist die Kopula. Die Kopula im ersten Beispielsatz lautet *Chillies are vegetables*. Die zugehörige Abhängigkeit ist `cop(Chillies, vegetables)`. Da die *vegetables* weder auf ein Zahlwort zurückgeführt werden können, noch als Adjektiv markiert sind (ihr POS-Tag ist `NNS` für ein Nomen im Plural), wählt AutoAnnotator eine Umsetzung über thematische Rollen. Vergeben werden in derartigen Fällen die Rollen *fingens* und *fictum*. Sie kodieren die Aussage der Kopula. Das Kopulaverb *are* trägt nach der Kodierung keine zusätzliche Semantik mehr und wird als sicherer Kommentar markiert.

Neben Spezialkonstruktionen wie der eben genannten Kopula versucht AutoAnnotator zunächst Verbalkonstruktionen über Subjekte, Prädikate und Objekte zu identifizieren. Hierzu dienen die Abhängigkeiten `nsubj(a,b)` und `dobj(a,b)`; ähnliche Abhängigkeiten existieren für passive Konstruktionen. Nachdem eine Verbalkonstruktion ermittelt wurde, werden etwaig zusätzlich vorhandene Abhängigkeiten betrachtet, wie beispielsweise `temp(a,b)`, die den Zeitpunkt mit einem anderen Konzept verbindet (vergleiche Tabelle 7). Zu diesem Zeitpunkt werden teilweise Platzhalterrollen vergeben, die in späteren Stufen genauer betrachtet werden.

Nach den vorangegangenen beiden Verarbeitungsstufen erhält man den folgenden Quelltext:

```
1 #{Chillies are very hot vegetables.}
2 [ Chillies|FIN #{are} $very $hot vegetables|FIC ].
3
4 #{Mike Tyson likes green chillies.}
5 [ Mike_Tyson|AG likes|METHODROLE $green chillies|PAT ].
6
7 #{Last week, he ate five of them.}
8 [ $Last week|TEMP, he|AG ate|METHODROLE *five #of them|PAT ].
```

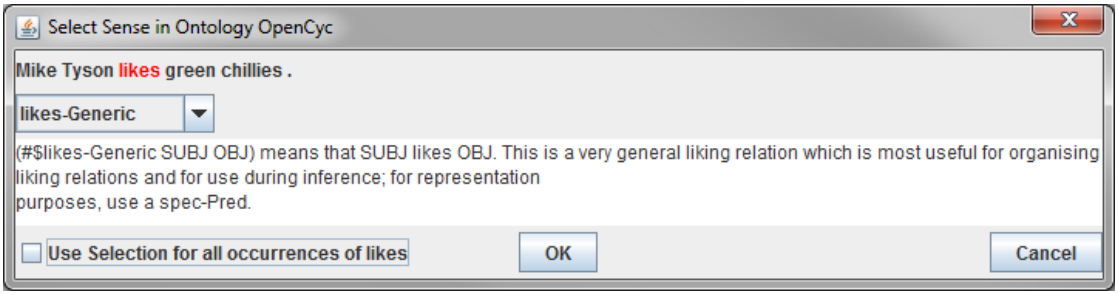


Abbildung 18: Die Auswahl eines Wortes in OpenCyc.

Quelltext 3: Mike Tyson Beispiel (Schritt 3, mit ersten thematischen Rollen und Modifikatoren).

In Zeile 2 kann man den sicheren Kommentar *are* erkennen (gekennzeichnet duch geschwungene Klammern). Ebenso wurde hier die oben genannte Kopula gekennzeichnet. Neben den Modifikatoren wurden einige thematischen Rollen vergeben. In allen drei Sätzen wurden bis hier Platzhalterrollen oder allgemeine Rollen wie `PAT` vergeben; insbesondere ist zu beachten, das `METHODROLE` keine Rolle aus dem Rollensatz von SAL_{E} ist. Sie wird verwendet, um im nächsten Schritt die Prädikate in die Untergruppen `ACT`, `STAT` und `TRANS` einzugruppieren.

Um Anfragen an eine Ontologie zu parametrisieren, benötigt man zunächst die Grundform des Wortes, über das man Informationen erhalten möchte. Doch auch mit der Grundform ist eine eindeutige Zuordnung eines Wortes auf ein Konzept nicht immer möglich (vergleiche Abschnitt 3.2). Um das gewünschte Konzept auswählen zu können, wird der Benutzer aufgefordert, aus den verfügbaren Konzepten das gewünschte auszuwählen. Hierzu wird ein Auswahldialog angezeigt, wie er in Abbildung 18 zu sehen ist. Der Dialog enthält neben einer Auswahlliste die Beschreibung des Konzeptes in der Ontologie. In Abbildung 18 wurde bereits das korrekte Konzept ausgewählt – die Beschreibung passt zur Verwendeung von *like* im Beispiel. Soll für dieses Wort in der aktuellen Spezifikation immer das gewählte Konzept verwendet werden, kann der Benutzer dies AutoAnnotator mitteilen.

Ist das verwendete Konzept in der Ontologie enthalten, werden verschiedene Abfragen durchgeführt, um das bisher ermittelte Rollengefüge zu verfeinern. Die in dieser Arbeit verwendeten Ontologiekonzepte sind in Anhang F genauer beschrieben. Im Beispiel müssen *likes* und *ate* genauer betrachtet werden, da sie noch mit Platzhalterrollen versehen sind. Bei Verben muss unterschieden werden, ob es sich um eine allgemeine Handlung (Rolle: `ACT`), einen Zustandsübergang (Rolle: `TRANS`) oder eine Beziehung handelt (Rolle: `STAT`).

Zunächst wird *likes* betrachtet. Nachdem das korrekte Konzept ausgewählt wurde, werden die Eigenschaften dieses Konzeptes betrachtet. Eine Abfrage an ResearchCyc liefert die folgenden Informationen (gekürzt):

```
Predicate: likes-Generic
  isa:   Predicate Relation
         TemporalExistencePredicate TemporalExistenceRelation
  arity: 2
  (argIsa likes-Generic 1 Agent-Generic)
  (argIsa likes-Generic 2 Thing)
```

Dieser Ausgabe kann man entnehmen, dass es sich bei *like* um ein zweistelliges Prädikat handelt. Neben diesen Informationen sieht man, dass die beiden Argumente von *like* ein allgemeiner Agent (`Agent-Generic`) und ein Ding (`Thing`) sind. Ein Ding ist in ResearchCyc als „etwas, das existiert“ definiert; es handelt sich hierbei um alle Dinge, unabhängig davon, ob sie physisch greifbar sind, oder es sich um abstrakte Dinge (wie eine mathematische Menge) handelt. Diesen Informationen kann man entnehmen, dass es sich bei *like* um eine Beziehung handelt, welche mit `STAT` gekennzeichnet wird (siehe dazu auch [KG08]).

In der nächsten Abfrage wird *ate* betrachtet. Eine Abfrage mit der Grundform *eat* liefert das Konzept `EatingEvent` mit den folgenden Eigenschaften (gekürzt):

```
Collection: EatingEvent
  isa:   Collection ConsumingFood-Food-Topic
         CumulativeEventType DurativeEventType
  genls: Action TemporalThing
         TemporallyExistingThing TemporallyExtendedThing
```

Aus diesen Informationen lässt sich ablesen, dass es sich bei *eat* um eine Handlung (`Action`) handelt, die von `TemporallyExtendedThing` generalisiert wird. TemporallyExtendedThings sind Dinge, die sich über einen bestimmten Zeitraum erstrecken und dabei nicht vollständig zu einem Zeitpunkt *da sind*. Diese Eigenschaft erfüllen Handlungen, da sie zu einem Zeitpunkt anfangen und zu einem anderen (späteren) Zeitpunkt enden. Zu einem bestimmten Zeitpunkt innerhalb dieses Zeitintervalls sind sie nicht vollständig präsent. Ein Gegenbeispiel wäre eine Person; sie wird zwar zu einem bestimmten Zeitpunkt geboren und stirbt zu einem bestimmten Zeitpunkt, allerdings ist sie zu jedem Zeitpunkt innerhalb ihrer Lebenszeit vollständig anwesend. Aus diesen Informationen lässt sich schließen, dass es sich bei *ate* um eine allgemeine Handlung handelt und daher mit der Rolle `ACT` versehen wird (siehe dazu auch [KG08]).

Weitere Unklarheiten ergeben sich bei dem Beispiel nicht, weswegen keine weiteren Anfragen an ResearchCyc gestellt werden. Der folgende Quelltext wird ausgegeben, wenn man nicht mit dem Extraktionsprozess fortfährt:

```
1 #{Chillies are very hot vegetables.}
2 [ Chillies|FIN #{are} $very $hot vegetables|FIC ].
3
4 #{Mike Tyson likes green chillies.}
5 [ Mike_Tyson|AG likes|STAT $green chillies|PAT ].
6
7 #{Last week, he ate five of them.}
8 [ $Last week|TEMP, he|AG ate|ACT *five #of them|PAT ].
```

Quelltext 4: Mike Tyson Beispiel (Schritt 4, nach der Verfeinerung der Rollen).

Die beiden Verfeinerungen gegenüber dem 3. Quelltext sind in den Zeilen 5 und 8 zu sehen.

5.4.6. Ermitteln von Referenzen

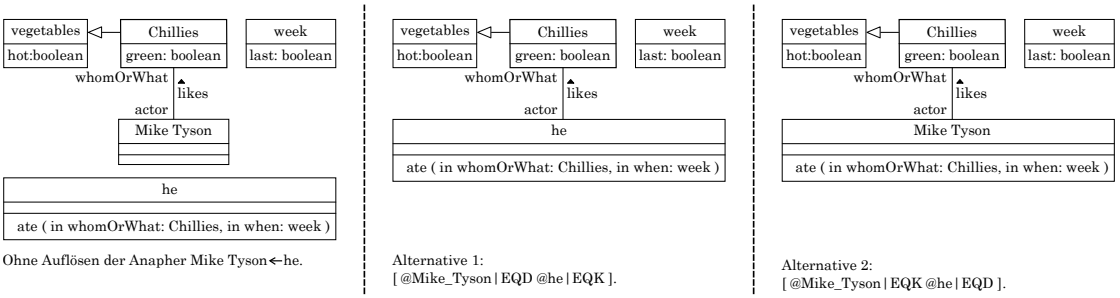


Abbildung 19: Die Auswirkung einer Auflösung einer Anapher.

Aufgelöste Referenzen führen in SAL$_E$ zu Zusicherungen. Zusicherungen sind kurze Sätze, in denen die beiden identischen Elemente verbunden werden. Hierbei wird das Element mit `EQK` (Kurzform für *equals keep*) gekennzeichnet, welches im Folgenden noch gebraucht wird. Das zweite Element wird mit `EQD` (Kurzform für *equals drop*) gekennzeichnet; es wird entfernt und das andere Element an seiner Stelle verwendet.

Die ersten Referenzen, die aufgelöst werden, sind die, die mithilfe von JavaRAP ermittelt wurden. Wie oben zu sehen ist, kennt man nun Anapher und Antezedens samt Satz und Position im Satz, sodass das jeweilige Element im SAL$_E$-Dokument ermittelt werden kann. Nach dem Satz, in dem die Anapher auftritt, wird nun eine Zusicherung eingefügt, die das Paar miteinander verbindet. Hierbei wird immer das Antezedens mit `EQK` gekennzeichnet. Die Anapher wird mit `EQD` gekennzeichnet, wodurch sie im weiteren Prozess von SAL$_E$ mx aus dem Diagramm verschwindet. Dieses Vorgehen wird dadurch gestützt, dass sich JavaRAP auf Personalpronomina konzentriert. Abbildung 19 verdeutlicht den Unterschied im Domänenmodell anhand des Anapher-Antezedens-Paares *Mike Tyson* und *he* aus den Sätzen zwei und drei des Beispiels. Sobald man ein Programm integriert, welches vollständige Koreferenzketten erstellt, können sich andere Schlussfolgerungen ergeben.

Darüber hinaus ist zu beachten, dass die gelieferten Konstituenten aus mehreren einzelnen Konstituenten bestehen können. Dann müssen mehrere Referenzen aufgelöst werden. Betrachtet man als Beispielsatz *Mike Tyson has a red Ferrari and a yellow Lamborghini. He likes them*, so ermittelt JavaRAP die folgenden Anapher-Antezedens-Paare:

```
(0,0) Mike Tyson <-- (1,0) He,
(0,3) a red Ferrari and a yellow Lamborghini <-- (1,2) them
```

Der Konstituent *a red Ferrari and a yellow Lamborghini* ist dann so zu behandeln, dass die beiden Nomen extrahiert und für die Zusicherung verwendet werden. Die zusätzlichen Attribute sind nicht zu wiederholen, da sie bereits bei der ursprünglichen Nennung im Text zugewiesen wurden und in der Zusicherung nicht zu wiederholen sind. In diesem Fall muss eine Menge verwendet werden:

```
[ Mike_Tyson|POSS #has
  #a $red Ferrari|HAB #and #a $yellow Lamborghini|HAB ].
[ He|AG likes|STAT them|PAT ].
[ @He|EQD @Mike_Tyson|EQK ].
[ {@Ferrari AND @Lamborghini }|EQK @them|EQD ].
```

Je nach Benutzerwunsch werden in einem nachgelagerten Schritt alle named entities auf ihr erstes Vorkommen im Text referenziert. Welche Tags hierfür ausgewertet werden, ist ebenfalls konfigurierbar; standardmäßig werden alle drei Tags (`LOCATION`, `ORGANIZATION`, `PERSON`) zur Auflösung verwendet. Der Beispieltext lautet nun: *Chillies are hot vegetables. Mike Tyson likes green chillies. Last week Mike Tyson ate five of them.* Setzt man in einem solchen Fall keine Referenz, so legt SAL$_E$ MX zwei Klassen namens *Mike Tyson* an und versieht sie mit einer Nummer. Da *Mike Tyson* zweimal mit den Kennzeichen `PERSON` versehen wurde – es sich also zweimal um dieselbe named entity handelt – wird das zweite Vorkommen auf das erste referenziert.

Ein noch kompakteres Diagramm wird dann mit SAL$_E$ MX erzeugt, wenn alle Worte, die eine Rolle einnehmen und dieselbe Grundform haben, auf ihr jeweils erstes Vorkommen referenziert werden. Dieses Vorgehen führt zu einer schwammigeren Semantik, da nicht mehr zwischen dem Selben und dem Gleichen unterschieden wird. Insbesondere wenn der Text viele Personalpronomina enthält, die nicht von JavaRAP aufgelöst werden können, ist die Wiedergabe im Domänenmodell nicht eindeutig. Aus diesem Grund ist diese Option ebenfalls vom Benutzer ein- oder auszuschalten; standardmäßig wird sie nicht eingesetzt.

Referenzen können nur für Worte gebildet werden, die im SAL$_E$-Dokument eine Rolle einnehmen. Soll folglich in dieser Prozessstufe ein Referenzpaar aufgelöst werden, von dem ein Element ein Kommentar im SAL$_E$-Dokument ist, so ist vermutlich ein Fehler aufgetreten: Entweder hätte das Kommentarelement eine Rolle einnehmen müssen, oder das ermittelte Referenzpaar ist falsch. Da an dieser Stelle nicht entschieden werden kann, welcher Fehler aufgetreten ist, wird eine Warnung erzeugt, die der Benutzer auswerten muss (siehe Abbildung 20)

Nach dem Auflösen der Referenzen, erhält man die folgende Ausgabe:

```
#{Chillies are very hot vegetables.}
[ Chillies|FIN #{are} $very $hot vegetables|FIC ].

#{Mike Tyson likes green chillies.}
[ Mike_Tyson|AG likes|STAT $green chillies|PAT ].

[ @Chillies|EQK @chillies|EQD ].

#{Last week, he ate five of them.}
[ $Last week|TEMP, he|AG ate|ACT *five #of them|PAT ].
```

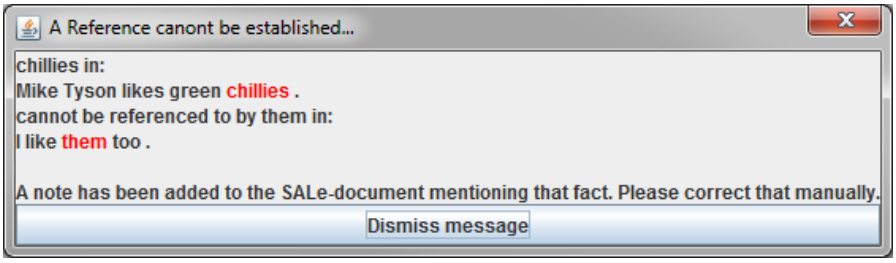


Abbildung 20: Die Meldung eines Fehlers bei der Referenzauflösung.

```
11
12 [ @them|EQD @chillies|EQK ]. [ @he|EQD @Mike_Tyson|EQK ].
```

Quelltext 5: Mike Tyson Beispiel (Schritt 5, fertiges Dokument mit Referenzen).

Wie in Zeile 12 zu sehen ist, wurden die Pronomina *he* und *them* von JavaRAP aufgelöst und das Auflösungsergebnis in Zusicherungen umgewandelt[20]. Verwendet man die Referenzauflösung über die Grundform der Worte, so wird zusätzlich die Zusicherung in Zeile 7 erzeugt.

5.4.7. Ausgabe des SAL_E-Dokuments

Nach den oben beschriebenen Schritten wird das SAL_E-Dokument serialisiert. Hierzu werden nacheinander die einzelnen Sätze ausgewertet und entsprechend ihrem Inhalt ausgegeben. Die Funktionsweise ist ähnlich der des SAL_E-Compilers, da in beiden Fällen ein SAL_E-Graph serialisiert werden muss.

Bei der Ausgabe von Modifikatoren wird in dieser Verarbeitungsstufe das Shifting berechnet. Bezieht sich ein Modifikator nicht auf das direkt folgende Wort, so muss er verschoben werden. Die Verschiebung wird als Anzahl der Verschiebungsschritte angegeben und kann folglich erst dann ermittelt werden, wenn keine Veränderungen am SAL_E-Dokument mehr vorgenommen werden (siehe auch Anhang B).

SAL_E kennt sowohl Einzelkommentare (gekennzeichnet durch Voranstellen von `#`), als auch Kommentare aus mehreren Worten (gekennzeichnet durch `#{` am Anfang und `}` am Ende). Diese Unterscheidung wird genutzt, um ohne eine Erweiterung der SAL_E-Syntax die beiden verschiedenen Kommentartypen kenntlich zu machen: Die initial eingefügten Kommentare werden als Einzelkommentare ausgegeben. Sichere Kommentare werden als Mehrwortkommentare ausgegeben; wo mehrere Worte hintereinander sicher auskommentiert sind, werden sie zu einem Mehrwortkommentar zusammengefasst. So kann der Analyst einfach feststellen, was von AutoAnnotator gewollt auskommentiert wurde und was nicht. Diese Vorgehensweise macht es möglich, gezielt nach Auswertungslücken oder Auswertungsfehlern zu suchen.

[20] Die Zusicherungen erhalten in der Ausgabe normalerweise jeweils eine eigene Zeile. Aus Platzgründen wurde hier die Ausgabe für den Druck angepasst.

5.5. Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Erkenntnisse aus Kapitel 4 umgesetzt werden können. Neben der Programmstruktur von AutoAnnotator wurden die einzelnen Pipelinestufen und die Informationsextraktion anhand eines Beispiels erläutert. Nach Abschluss der syntaktischen Analyse wurde gezeigt, wie man mithilfe von Ontologieabfragen provisorisch ermittelte Rollengefüge genauer ermitteln kann.

6. Evaluation und Abgrenzung

In diesem Kapitel werden einige Beispiele mit AutoAnnotator annotiert und die Ergebnisse qualitativ bewertet. Für keines der Beispiele besteht ein belastbarer Goldstandard. Das Mike Tyson Beispiel wurde in [KL10] erstmalig veröffentlicht; die Annotationen wurden innerhalb des RECAA-Projektes diskutiert und können aufgrund der einfachen Sätze als korrekt angesehen werden. Die Beispiele von SUGAR/UMGAR und CIRCE wurden von deren Autoren zusammen mit einer Lösung publiziert. Die Programme zielen jedoch nicht darauf ab, eine semantische Annotation durchzuführen, sondern direkt Modelle zu erstellen.

Für das Beispiel von SUGAR wurden zusätzlich zur Gütebewertung aus den veröffentlichten Modellen die zu extrahierenden Modellelemente ermittelt und geprüft, ob diese auch bei einer Modellextraktion mit Sal_E MX erstellt werden würden.

In der annotierten Spezifikation von CIRCE wird ein Problem aufgezeigt, das AutoAnnotator derzeit nicht lösen kann.

Neben der qualitativen Bewertung werden die Programmlaufzeiten genannt, welche einen groben Eindruck über die Arbeitsgeschwindigkeit geben sollen. Diese Zeitangaben hängen jedoch stark vom Benutzer ab, da die Hauptlaufzeiten im interaktiven Teil von AutoAnnotator entstehen.

6.1. Mike Tyson

Vorgang	Dauer	Vorgang	Dauer	Vorgang	Dauer
Init Quellen	7330 ms	POS-Tagger	281 ms	Typed Deps	188 ms
Init Extrakt.	2388 ms	Parser	1101 ms	Ref.: Anapher	n.m.
Satz-Splitter	15 ms	NER	39 ms	Cyc-Anfragen	6870 ms
Wort-Splitter	17 ms	WordNet	63 ms	Ref.: NE	63 ms
JavaRAP	27847 ms	Subphrasen	23 ms	Gesamt	46737 ms

Tabelle 8: Durchschnittliche Verarbeitungszeiten für das Mike Tyson Beispiel (gekürzt).

Das erste Beispiel das zur Beurteilung von AutoAnnotator dienen soll, ist das Beispiel aus [KL10]. Es handelt sich hierbei um das durchgehende Beispiel aus Kapitel 5:

```
1 Chillies are very hot vegetables.
2 Mike Tyson likes green chillies.
3 Last week, he ate five of them.
```

Quelltext 6: Das Mike Tyson Beispiel.

AutoAnnotator benötigte durchschnittlich 47 Sekunden zum Verarbeiten dieses Textes. Hierbei entfielen die in Tabelle 8 genannten Zeiten auf die einzelnen Verarbeitungsstufen.

Das von AutoAnnotator erzeugte Sal_E-Dokument ist nachfolgend zu sehen:

```
1  #{Chillies are very hot vegetables.}
2  [ Chillies|FIN #{are} $very $hot vegetables|FIC ].
3
4  #{Mike Tyson likes green chillies.}
5  [ Mike_Tyson|AG likes|STAT $green chillies|PAT ].
6
7  [ @Chillies|EQK @chillies|EQD ].
8
9  #{Last week, he ate five of them.}
10 [ $Last week|TEMP, he|AG ate|ACT *five #of them|PAT ].
11
12 [ @them|EQD @chillies|EQK ]. [ @he|EQD @Mike_Tyson|EQK ].
```

Quelltext 7: Das Mike Tyson Beispiel (annotiert).

Da das Beispiel bereits ausführlich besprochen wurde, wird an dieser Stelle auf eine detaillierte Betrachtung verzichtet. Wie erwartet, wurde *Mike Tyson* zusammengefasst (zu erkennen am Unterstrich in Zeile 2). Ebenso wurde erkannt, dass *like* ein (lang andauernder) Zustand ist und keine Handlung. Die ermittelten Koreferenzen führten zu den Zusicherungen in den Zeilen 12 und 13. Die Zusicherung in Zeile 7 wird nur erzeugt, wenn die Referenzauflösung über die Grundformen der Wörter durchgeführt wird. Die Tatsache, dass es sich bei *Chillies* und *chillies* um dasselbe handelt, wird von den eingesetzten Programmen nicht erkannt. Im Folgenden werden keine Referenzen mehr über die Grundformen aufgelöst, um die Lesbarkeit der SAL$_E$-Texte zu verbessern.

6.2. Fillmore – Türenbeispiel

Vorgang	Dauer	Vorgang	Dauer	Vorgang	Dauer
Init Quellen	8531 ms	POS-Tagger	211 ms	Typed Deps	96 ms
Init Extrakt.	1469 ms	Parser	2158 ms	Ref.: Anapher	n.m.
Satz-Splitter	11 ms	NER	564 ms	Cyc-Anfragen	38858 ms
Wort-Splitter	68 ms	WordNet	281 ms	Ref.: NE	18 ms
JavaRAP	28217 ms	Subphrasen	82 ms	GESAMT	80609 ms

Tabelle 9: Durchschnittliche Verarbeitungszeiten für das Türenbeispiel (gekürzt).

Fillmore zeigt in [Fil69, S.363] ein Beispiel für vier unterschiedliche Sätze, die denselben Vorgang ausdrücken:

```
1 The door will open.
2 The janitor will open the door.
3 The janitor will open the door with this key.
4 This key will open the door.
```

Quelltext 8: Fillmores Türenbeispiel.

Er zielt dabei darauf ab, dass eine syntaktische Analyse nicht zu korrekten semantischen Beziehungen führen kann – sonst müsste in allen vier Sätzen der *key* das Hilfsmittel, der *janitor* der Handelnde und die *door* das Behandelte sein.

AutoAnnotator benötigte durchschnittlich 81 Sekunden zum Verarbeiten dieses Textes. Hierbei entfielen die in Tabelle 9 genannten Zeiten auf die einzelnen Verarbeitungsstufen.

AutoAnnotator gelangt nach der Analyse zu folgender Annotation:

```
#{The door will open.}
[ #{The} door|AG #will open|ACT ].

#{The janitor will open the door.}
[ #{The} janitor|AG #will open|ACT #{the} door|PAT ].

#{The janitor will open the door with this key.}
[ #{The} janitor|AG #will open|ACT #{the}door|PAT #with
  #{this} key|INSTP ].

#{This key will open the door.}
[ #{This} key|AG #will open|ACT #{the} door|PAT ].
```

Quelltext 9: Fillmores Türenbeispiel (annotiert).

Betrachtet man den ersten Satz isoliert, fehlt ein Hinweis, dass sich dir Tür nicht selbst öffnet (bspw. bei einem Fahrstuhl); die ermittelte Annotation der Tür als Handelnde ist also durchaus nachvollziehbar – und das nicht nur aufgrund der (technischen) Überlegungen der vorangegangenen Kapitel.

Die Annotation der beiden folgenden Sätze ist nachvollziehbar und korrekt. Die Annotation des letzten Satzes ist jedoch nicht richtig: In dieser Aussage ist der *key* kein Handelnder; die Tatsache, dass ein bestimmter Schlüssel eine Tür (genauer: ein passendes Schloss) öffnet, ist eher eine Eigenschaft des Schlüssels. Hätte man den Satz manuell annotiert, hätte man daher statt der Rolle `ACT` die Rolle `STAT` gewählt. Korrekt kann man derartige Sätze nur annotieren, wenn man die Betrachtung ausweitet. Denkbar wäre eine kontextabhängige Betrachtung des Verbes bei den Ontologieabfragen.

Tabelle 10 fasst die Annotationsergebnisse zusammen. Für jeden Satz sind jeweils die korrekten Annotationen ($\surd$) den fehlerhaften (!) und den fehlenden (?) gegenübergestellt. Bei den fehlenden Annotationen handelt es sich um die nicht als sicher gekennzeichneten Kommentare *will* und *with*; diese haben jedoch keinen Einfluss auf die Güte des Ergebnisses.

6.3. SUGAR/UMGAR – Musical Store

Deeptimahanti und Sanyal evaluieren ihren Ansatz zur Modellextraktion (vergleiche Abschnitt 3.4.5) in [DS09] mit dem folgenden Text. Es handelt sich hierbei um eine abgewandelte (vereinfachte) Fassung der Spezifikation, die von Overmyer für die Beschreibung von LIDA verwendet wurde (vergleiche Abschnitt 3.4.3 und [OLR01]). Die Spezifikation umfasst 17 Sätze. AutoAnnotator benötigte durchschnittlich 129 Sekunden zum

Satz	√	!	?
1	3	0	1
2	5	0	1
3	7	0	2
4	4	1	1

Tabelle 10: Qualitative Beurteilung der Annotation des Türenbeispiels.

Vorgang	Dauer	Vorgang	Dauer	Vorgang	Dauer
Init Quellen	5694 ms	POS-Tagger	408 ms	Typed Deps	252 ms
Init Extrakt.	1281 ms	Parser	3297 ms	Ref.: Anapher	3 ms
Satz-Splitter	14 ms	NER	232 ms	Cyc-Anfragen	90628 ms
Wort-Splitter	362 ms	WordNet	493 ms	Ref.: NE	7 ms
JavaRAP	21256 ms	Subphrasen	201 ms	Gesamt	128813 ms

Tabelle 11: Durchschnittliche Verarbeitungszeiten für das Musical Store Beispiel (gekürzt).

Verarbeiten dieses Textes. Hierbei entfielen die in Tabelle 11 genannten Zeiten auf die einzelnen Verarbeitungsstufen.

```
1  The musical store receives tape requests from customers.
2  The musical store receives new tapes from the Main office.
3  Musical store sends overdue notice to customers.
4  Store assistant takes care of tape requests.
5  Store assistant update the rental list.
6  Store management submits the price changes.
7  Store management submits new tapes.
8  Store administration produces rental reports.
9  Main office sends overdue notices for tapes.
10 Customer request for a tape.
11 Store assistant checks the availability of requested tape.
12 Store assistant searches for the available tape.
13 Store assistant searches for the rental price of available tape.
14 Store assistant checks status of the tape to be returned by customer.
15 Customer can borrow if there is no delay with return of other tapes.
16 Store assistant records rental by updating the rental list.
17 Store assistant asks the customer for his address.
```

Quelltext 10: Musical Store Spezifikation (SUGAR/UMGAR).

Aufgrund der strikten Subjekt-Prädikat- oder Subjekt-Prädikat-Objekt-Struktur der Sätze, ergibt sich eine gleichförmige Annotation. Der Text enthält außer Präpositionalkonstruktionen keine speziellen Schwierigkeiten. AutoAnnotator gelangt zu folgender Annotation:

```
1  #{The musical store receives tape requests from customers.}
2  [ #{The} $musical store|AG receives|ACT tape_requests|PAT
3    #{from} customers|OBJECTROLE ].
```

```
4
5  #{The musical store receives new tapes from the Main office.}
6  [ #{The} $musical store|AG receives|ACT $new tapes|PAT
7    #{from the} Main_office|OBJECTROLE ].
8
9  #{Musical store sends overdue notice to customers.}
10 [ $Musical store|AG sends|ACT $overdue notice|PAT #to
11   customers|RECPLDEST ].
12
13 #{Store assistant takes care of tape requests.}
14 [ Store_assistant|AG takes|ACT care|PAT #of #tape_requests ].
15
16 #{Store assistant update the rental list.}
17 [ Store_assistant|AG update|ACT #{the} $rental list|PAT ].
18
19 #{Store management submits the price changes.}
20 [ Store_management|AG submits|ACT #{the} price_changes|PAT ].
21
22 #{Store management submits new tapes.}
23 [ Store_management|AG submits|ACT $new tapes|PAT ].
24
25 #{Store administration produces rental reports.}
26 [ Store_administration|AG produces|ACT $rental reports|OPUSP ].
27
28 #{Main office sends overdue notices for tapes.}
29 [ Main_office|AG sends|ACT $overdue notices|PAT #{for}
30   tapes|OBJECTROLE ].
31
32 #{Customer request for a tape.}
33 [ Customer|AG request|ACT #{for a} tape|OBJECTROLE ].
34
35 #{Store assistant checks the availability of requested tape.}
36 [ Store_assistant|AG checks|ACT #{the} availability|PAT
37   #of $requested tape|ADJ ].
38
39 #{Store assistant searches for the available tape.}
40 [ Store_assistant|AG searches|ACT #{for the} $available
41   tape|OBJECTROLE ].
42
43 #{Store assistant searches for the rental price of available tape.}
44 [ Store_assistant|AG(S14) searches|ACT(S14) #{for the} $rental
45   price|HAB(S15) #of $available tape|POSS(S15) ].
46
47 #{Store assistant checks status of the tape to be returned by customer.}
48 [ Store_assistant|AG checks|ACT status|PAT #of #{the}
49   tape|OBJECTROLE #to #be_returned #{by} customer|OBJECTROLE ].
50
51 #{Customer can borrow if there is no delay with return of other tapes.}
52 [ Customer|AG #can borrow|ACT
53   [ #{if there is} *no delay|OBJECTROLE #{with} return|HAB #of $other
54     tapes|POSS ]|SUM
55 ].
56
57 #{Store assistant records rental by updating the rental list.}
```

```
58 [ Store_assistant|AG records|ACT rental|PAT #by
59   #updating #{the} $rental list|ADJ ].
60
61 #{Store assistant asks the customer for his address.}
62 [ Store_assistant|AG(S20) asks|ACT(S20) #{the} customer|PAT(S20) #{for}
63   his|POSS(S21) address|HAB(S21) ].
64
65 [ @his|EQD @Store_assistant|EQK ].
```

Quelltext 11: Musical Store Spezifikation (SUGAR/UMGAR) (annotiert).

Deeptimahanti geben an, dass ihr Ansatz die in Tabelle 12 aufgeführten Klassen und zugehörigen Attribute sowie Methoden extrahiert. In der Tabelle sind neben den Klassennamen die Attribute in Klammern angegeben, anschließend die Methodennamen. Ob SUGAR neben den Name der Methoden auch ihre Parameter extrahiert, lässt sich [DS09] nicht entnehmen.

Im Modell von SUGAR sind zusätzlich die folgenden (unbenannten) Beziehungen vorhanden:

- main office $\longleftrightarrow$ musical store
- musical store $\longleftrightarrow$ customer
- customer $\longleftrightarrow$ store assistant

Im von SUGAR erzeugten Anwendungsfalldiagramm sind die Subjekte der Spezifikationssätze die Akteure. Die Anwendungsfälle sind die jeweils folgenden Satzenenden.

Neben den Klassen und Methoden, die in Tabelle 12 aufgeführt sind, erzeugt AutoAnnotator Annotationen, aus denen Sal_{E} MX die Klassen aus Tabelle 13 erzeugt. Zusätzlich werden die fogenden Beziehungen gekennzeichnet:

- tape $\xrightarrow{\text{has}}$ availability
- tape $\xrightarrow{\text{has}}$ price
- Store_assistant $\xrightarrow{\text{has}}$ address

In der letzten Zeile des Annotierten Txtes sieht man das Ergebnis eines Auflösungsfehlers. Die Anapher *his* wurde fälschlicherweise auf den *store assistant* bezogen. Dies führt dazu, dass die Kundenadresse im Modell fälschlicherweise dem *store assistant* zugeteilt wird.

Tabelle 14 fasst die qualitative Beurteilung der Annotation zusammen. Für jeden Satz sind jeweils die korrekten Annotationen ($\surd$) den fehlerhaften (!) und den fehlenden (?) gegenübergestellt. Z Steht für die Zusicherung, die aufgrund der Anapher eingefügt wurde.

Die Klassen, die von SUGAR ermittelt werden, werden auch von AutoAnnotator ermittelt. Die erzeugte Annotation erzeugt darüber hinaus die Methodenparameter, die in [DS09] nicht ersichtlich sind.

SUGAR		AutoAnnotator	
customer	borrow request	customers	borrow request(tape)
main office (main)	sends	Main_office	sends(notices)
musical store (musical)	sends receives	store (musical)	sends(notices) receives(tapes) receives(tape_requests)
store administration	produces	Store_administration	produces(reports)
store assistant	takes records checks update searches	Store_assistant	takes(care) records(rental) checks(availability) update(list) searches(tape) searches(price) ask(customer)
store management	submits	Store_management	submits(price_changes) submits(tapes)

Tabelle 12: Identifizierte Klassen im Musical Store.

address	list (rental)	price_changes
care	notice (overdue)	reports (rental)
delay	price (rental)	return
tape (available, new, other, requested)		

Tabelle 13: Zusätzlich von AutoAnnotator im Musical Store erkannte Klassen.

Satz	√	!	?	Satz	√	!	?
1	7	0	1	10	4	0	1
2	9	0	1	11	6	0	2
3	4	2	0	12	6	0	1
4	5	1	2	13	9	0	2
5	6	0	0	14	6	0	6
6	6	0	0	15	12	0	3
7	5	0	0	16	6	0	3
8	5	0	0	17	7	1	1
9	6	0	1	Z	0	1	0

Tabelle 14: Qualitative Beurteilung der Annotation des Musical Store Beispiels.

6.4. Die Spezifikation von CIRCE

Vorgang	Dauer	Vorgang	Dauer	Vorgang	Dauer
Init Quellen	6026 ms	POS-Tagger	2520 ms	Typed Deps	214 ms
Init Extrakt.	1523 ms	Parser	3325 ms	Ref.: Anapher	n.m.
Satz-Splitter	18 ms	NER	331 ms	Cyc-Anfragen	41958 ms
Wort-Splitter	31 ms	WordNet	542 ms	Ref.: NE	25 ms
JavaRAP	31808 ms	Subphrasen	344 ms	GESAMT	88715 ms

Tabelle 15: Durchschnittliche Verarbeitungszeiten für die Spezifikation von CIRCE (gekürzt).

Ambriola und Gervasi beschreiben ihr System CIRCE in [AG97] mit einer kurzen Spezifikation, die 12 Sätze enthält.

```
1  The system is made of the Web interface, of Cico, of the view modules,
       and of the view selector.
2  The Web interface receives from the user requirements and glossary.
3  Requirements contain data on the team, on the author and on the revision.
4  The Web interface transmits to Cico requirements and glossary.
5  If the project is cooperative, the Web interface sends requirements and
       glossary to the repository, too.
6  Cico computes abstract requirements using requirements, glossary, MAS-
       rules, predefined glossary and team data.
7  If the project is cooperative, Cico requests team data to the repository.
8  The view modules receive abstract requirements from Cico.
9  The view modules can be dedicated to modeling, validation or metrication.
10 From abstract requirements, view modules compute a view.
11 The view module sends the view to the view selector.
12 The user requests a view to the view selector.
```

Quelltext 12: Die Spezifikation von CIRCE.

Die von AUTOANNOTATOR erzeugte Annotation findet sich in folgendem Quelltext. Auf die einzelnen Punkte wird an dieser Stelle nicht eingegangen. Stattdessen wird ein markantes Problem (Verarbeitung fehlerhafter Informationen) aufgegriffen und erläutert.

Ein Satz dieses Beispiels zeigt deutlich, was passiert, wenn der Syntaxbaum des Satzes fehlerhaft aufgebaut wurde. Satz 7 der Spezifikation lautet *If the project is cooperative, Cico requests team data from the repository.* Er wird vom Parser zu folgendem Syntaxbaum verarbeitet:

```
(ROOT
  (S
    (SBAR (IN If)
      (S
        (NP (DT the) (NN project))
        (VP (VBZ is)
          (ADJP (JJ cooperative)))))
```

```
(, ,)
(NP (NNP Cico) (NNS requests))
(VP (VB team)
  (NP (NNS data))
  (PP (IN from)
    (NP (DT the) (NN repository))))
(. .)))
```

Hier zeigt sich, warum *team* fälschlicherweise als *actus* gekennzeichnet wurde und *Cico requests* als *agens*. Bereits im Syntaxbaum wurden die Konstituenten fehlerhaft zusammengefügt. Diesen Umstand erkennt man an den beiden folgenden Knoten:

```
(NP (NNP Cico) (NNS requests))
(VP (VB team)
```

Dieser Fehler im Syntaxbaum führt zu einer falschen Umsetzung von AutoAnnotator. Zu sehen ist dies in Zeile 37 des annotierten Textes. Um derartige Fehler aufzudecken, müsste man die Syntaxbäume des Textes mehrfach mit verschiedenen Parsern (oder unterschiedlich trainierten Modellen) erstellen und die Ergebnisse vergleichen, bevor mit dem Verarbeitungsprozess begonnen wird.

```
1  #{The system is made of the Web interface, of Cico, of the view modules,
2    and of the view selector.}
3  [ #{The} system|OBJECTROLE is_made|HAB(S1) #of #{the}
4    Web_interface|{POSS(S1), HAB(S2), HAB(S3)}, #of Cico|POSS(S2),
5    #of #{the} view_modules|POSS(S3),
6    #and #{of the} view_selector|OBJECTROLE ].
7
8  #{The Web interface receives from the user requirements and glossary.}
9  [ #{The} Web_interface|AG(S4) receives|ACT(S4) #{from the}
10   user_requirements|OBJECTROLE #and #glossary ].
11
12 #{Requirements contain data on the team, on the author and on the
13   revision.}
14 [ Requirements|AG(S5) contain|ACT(S5) data|PAT(S5) #{on the}
15   team|OBJECTROLE, #{on the} author|OBJECTROLE #and #{on the}
16   revision|OBJECTROLE ].
17
18 #{The Web interface transmits to Cico requirements and glossary.}
19 [ #{The} Web_interface|AG transmits|ACT #to Cico_requirements|RECPLDEST
20   #and #glossary ].
21
22 #{If the project is cooperative, the Web interface sends requirements
23   and glossary to the repository, too.}
24 [ [ #{If the} project|COP #{is} $cooperative<<1 ]|SUM,
25   #{the} Web_interface|AG sends|ACT requirements|PAT #and #glossary #to
26   #{the} repository|RECPL, $too<<3 ].
27
28 #{Cico computes abstract requirements using requirements, glossary,
29   MAS-rules, predefined glossary and team data.}
30 [ Cico|AG(S10) computes|ACT(S10) $abstract requirements|PAT(S10) #using
```

```
31    #requirements, #glossary, #MAS-rules, $predefined glossary|ADJ #and
32    #team_data ].
33
34  #{If the project is cooperative, Cico requests team data to the
35  repository.}
36  [ [ #{If the} project|COP #{is} $cooperative<<1 ]|SUM,
37    Cico_requests|AG team|ACT data|PAT #to #{the} repository|RECP ].
38
39  #{The view modules receive abstract requirements from Cico.}
40  [ #{The} view|AG(S13) modules|ACT(S13) #receive $abstract
41    requirements|ADJ #{from} Cico|OBJECTROLE ].
42
43  #{The view modules can be dedicated to modeling, validation or
44    metrication.}
45  [ #{The} view_modules|OBJECTROLE #can
46    be_dedicated|{ACT(S14), ACT(S15), ACT(S16)} #to
47    modeling|PAT(S14), validation|PAT(S15) #or metrication|PAT(S16) ].
48
49  #{From abstract requirements, view modules compute a view.}
50  [ #{From} $abstract requirements|OBJECTROLE, view_modules|AG
51    compute|ACT #{a} view|PAT ].
52
53  #{The view module sends the view to the view selector.}
54  [ #{The} view_module|AG sends|ACT #{the} view|PAT #to #{the}
55    view_selector|RECP ].
56
57  #{The user requests a view to the view selector.}
58  [ #{The} user|AG requests|ACT #{a} view|PAT #to #{the}
59    view_selector|RECP ].
```

Quelltext 13: Die Spezifikation von CIRCE (annotiert).

6.5. Laufzeit

Vorgang	Dauer	Vorgang	Dauer	Vorgang	Dauer
Init Quellen	6689 ms	POS-Tagger	1138 ms	Typed Deps	223 ms
Init Extrakt.	1613 ms	Parser	3087 ms	Ref.: Anapher	n.m.
Satz-Splitter	16 ms	NER	283 ms	Cyc-Anfragen	64204 ms
Wort-Splitter	201 ms	WordNet	469 ms	Ref.: NE	18 ms
JavaRAP	25960 ms	Subphrasen	229 ms	Gesamt	90728 ms

Tabelle 16: Durchschnittliche Verarbeitungszeiten pro Satz (gekürzt).

In Tabelle 16 sind die Zeitspannen aufgeführt, die die jeweiligen Verarbeitungsstufen benötigten. Wo möglich, wurde die durchschnittliche Zeitdauer bezogen auf einen Satz berechnet. Da die Programminitialisierung (inklusive der Initialisierung der eingesetzten Programme) nicht von der Anzahl der Sätze im Dokument abhängt, wurde hier die durchschnittliche Initialisierungsdauer über die Anzahl der Beispiele berechnet. Dasselbe gilt für die Gesamtdauer.

Die Zeiten wurden auf einem Rechner mit einem Core2 DUO Prozessor mit 2, 40GHz, 2GB Arbeitsspeicher und Windows 7 (64bit) als Betriebssystem gemessen. Hierbei wurden alle integrierten Programme lokal ausgeführt; hiervon ausgenommen ist die Ontologie Cyc, welche auf einer virtuellen Maschine des RECAA-Projektes zur Verfügung gestellt wurde. Cyc stehen hierbei 4GB Arbeitsspeicher und ein Opteron Prozessor zur Verfügung.

Ein großer Anteil des Zeitaufwandes entsteht im Bereich der Ontologieabfragen. Hierbei ist jedoch nicht die Ontologie (-anbindung) der kritische Zeitfaktor, sondern die Antwortzeit des Benutzers. In dieser Verarbeitungsphase werden Rückfragen gestellt, deren Beantwortung nur möglich ist, wenn man die Konzepte von Cyc kennt und/oder die Konzeptbeschreibungen liest.

6.6. Abgrenzung

Im Rahmen der Entwicklung von AutoAnnotator und der Evaluation zeichnete sich ab, dass AutoAnnotator nicht uneingeschränkt funktioniert. Die Qualität (Struktur) der Texte nimmt einen großen Einfluss auf die Verarbeitungsgüte der eingebundenen NLP-Programme. Wo diese eine fehlerhafte Ausgabe liefern, kann AutoAnnotator keine richtigen Schlussfolgerungen mehr ziehen. Aufgrund der Tatsache, dass für jeden Teil der Datengrundlage bis jetzt nur ein Programm eingebunden wurde, kann AutoAnnotator fehlerhafte Informationen nicht erkennen.

Zu prüfen ist außerdem, ob der verwendete Ansatz auf andere Sprachen übertragbar ist. Hierfür müssten die verwendeten NLP-Komponenten für andere Sprachen parametrisiert werden. Darüber hinaus benötigt man eine (oder mehrere) Ontologien in der gewählten Sprache. Da der Einsatz von Sal_{E} und Sal_{E} MX für einige andere Sprachen vorgestellt wurde (siehe [Gel10]), wären diese Sprachen vorrangig zu betrachten. Da im Rahmen dieser Arbeit nur eine begrenzte Anzahl von Programmen betrachtet und eingebunden werden konnte, sind diese Schritte Teil einer zukünftigen Weiterentwicklung von AutoAnnotator.

AutoAnnotator darf nicht als Programm für einen Endkunden gesehen werden. Es wurde zu dem Zweck entwickelt, Analysten bei ihrer Arbeit mit natürlichsprachlichen Spezifikationen zu unterstützen.

6.7. Zusammenfassung

Die Annotierungen der ersten beiden Beispiele zeigen, dass der vorgestellte Ansatz grundsätzlich in der Lage ist, eine Annotation durchzuführen oder vorzubereiten. Im vierten Beispiel ist die Qualität der Annotation nicht mehr mustergültig und gibt Anregungen zur Verbesserung der Erkennungsrate. Im Vergleich mit einem ähnlichen Ansatz zur Modellerzeugung liefert die erzeugte Annotation jedoch einen guten Ausgangpunkt für weitere Arbeiten.

Rückfragen des Programmes helfen, um Mehrdeutigkeiten zu eliminieren und Konzepte in der Ontologie zu ermitteln. Die Verarbeitungszeiten der Ontologie sind dabei

ausreichend gering, sodass der Analyst bei der Arbeit mit AutoAnnotator nicht warten muss.

7. Ausblick und weitere Schritte

Neben den Hinweisen, die sich aus der Evaluation ergeben haben, gibt es weitere mögliche Erweiterungen, um die Güte der Annotation zu steigern. Darüber hinaus gibt es Problembereiche, die AutoAnnotator bis jetzt nicht abdeckt. Im Folgenden werden mögliche Verbesserungen und Erweiterungen angesprochen.

7.1. Prüfen von Attributen

Die in Abschnitt 5.4.4 beschriebene Auflösung von Adjektiven ist nicht immer eindeutig. Betrachtet man ein Beispiel aus [KB09a], so sieht man die Grenzen dieses Ansatzes. Der erste Teil des Beispiels ist *Tom saw the plane flying.* Hier gelangt AutoAnnotator aufgrund der gesammelten Informationen zu dem Schluss, dass das *plane* ein Attribut *flying* bekommen sollte. Betrachtet man den zweiten Satz des Beispiels *Tom saw the mountains flying*, so muss AutoAnnotator aufgrund der strukturellen Gleichheit zum selben Schluss kommen: Die *mountains* erhalten ein Attribut *flying.* Dieses Ergebnis ist höchstwahrscheinlich nicht erwünscht, da Berge selten fliegen und sich somit *flying* eher auf *Tom* bezieht, der zum Beobachtungszeitpunkt im Flugzeug saß. Diese sprachliche Mehrdeutigkeit lässt sich nur aufgrund einer Syntaxanalyse mit NLP-Programmen nicht beheben. In einem (dem allgemeinen Analyseteil nachgelagerten) Schritt könnten alle Attributierungen nochmals auf Plausibilität geprüft werden. Wird ein unplausibles Attribut erkannt, müsste im aktuellen Satz nach anderen (passenden) Elementen gesucht werden. Sofern andere Bezugspunkte ermittelt werden können, müssten sie dem Benutzer zur Auswahl angeboten werden. Ansätze, wie man diese Art von Anfragen ausführt, finden sich in [Bru09].

7.2. Auswertung der Synsets von WordNet

WordNet wird bisher in AutoAnnotator ausschließlich als Informationslieferant – für die Bestimmung der Grundformen der Worte – verwendet. Die Informationen, die in den Synsets kodiert sind, könnten jedoch auch als Entscheidungsgrundlage für die Zuweisung von thematischen Rollen verwendet werden. Sicher dürfte die Betrachtung von „typischen“ Spezifikationsworten zu einer guten Abdeckung der Spezifikationstexte führen. Diese Information steht grundsätzlich ebenfalls zur Verfügung – ob die Auswertung aller Synsets sinnvoll ist, muss noch geprüft werden.

7.3. Verwendung eines Glossars für Referenzen

Die Erkennung von Entitäten zur Referenzierung ist verbesserungswürdig. Ein Glossar mit Systemkomponenten, die im Domänenmodell nur einmal auftauchen sollten, würde diese Aufgabe erleichtern. Das Glossar könnte einerseits a priori vom Benutzer angelegt werden. Andererseits könnte man im Verlauf der Textverarbeitung dem Benutzer Vorschläge für die Aufnahme von Begriffen in das Glossar machen. Nominalphrasen, die häufig im Text auftreten, eignen sich als Kandidaten für Glossareinträge (ähnlich funktioniert bspw. die Glossarentwicklung im Requirements Analysis Tool, das in Abschnitt

3.4.7 vorgestellt wurde). Ist ein Textelement, das in diesem Glossar steht, einzigartig (wie bspw. ein *WHOIS Server*), so steigt nicht nur die Lesbarkeit des erzeugten Domänenmodells, sondern gleichzeitig auch die Annotationsgüte (vergl. Abschnitt 4.4).

7.4. Koreferenzketten

AutoAnnotator verwendet für die Analyse von Referenzen JavaRAP. JavaRAP erstellt jedoch keine vollständigen Koreferenzketten, sondern löst Anaphern auf. Dies führt zwar zu einer Verbesserung der Annotation, jedoch wäre die Integration einer vollständigen Koreferenzanalyse wünschenswert. Hierdurch könnten nicht nur Platzhalter wie Personalpronomina aufgelöst werden, sondern weitere Informationen über Entitäten erhalten werden. Betrachtet man beispielsweise die Sätze *Der deutsche Außenminister besuchte die USA. Es war Westerwelles erste Auslandsreise in diesem Jahr*, so könnte man mit einer korrekten Koreferenzkette schließen, dass *Westerwelle* ein *deutscher Außenminister* ist (oder umgekehrt).

7.5. Zeitliche Abhängigkeiten

AutoAnnotator betrachtet derzeit keine zeitlichen Abhängigkeiten zwischen einzelnen Aktionen, Zuständen oder Beziehungen. Diese semantischen Beziehungen können jedoch in Sal_{E} kodiert und in Sal_{E} MX genutzt werden (bspw. für die Erzeugung von Aktivitätsdiagrammen).

Die Analyse von Zeitreihen ist ein breites Forschungsgebiet und es existieren verschiedenste Ansätze zu ihrer Ermittlung. Bethard et al. beschreiben, dass es verhältnismäßig einfach ist, Ereignisse in Texten zu identifizieren [BMK07]. Die zeitlichen Abhängigkeiten zwischen den gefundenen Ereignissen lassen sich jedoch sehr viel schwieriger identifizieren: In einer vereinfachten Aufgabe konnten Zeitreihen lediglich mit ca. 50-60 Prozent Korrektheit erzeugt werden. Mithilfe von relativ wenigen Trainingsdaten lassen sich Support Vector Machines (SVM) erzeugen, die neu hinzukommende zeitliche Abhängigkeiten mit einer Genauigkeit von fast 90 Prozent identifizieren können. Hierbei wird jedoch auf die grammatikalischen Zeiten der Verben im Text zurückgegriffen. Gilt die Prämisse von Gelhausen, dass Spezifikationstexte in der dritten Person Singular im Präsens geschrieben sind, so wird die Aufgabe noch schwieriger. Weicht man die Prämisse auf und lässt verschiedene Zeiten zu, so ist dennoch zu erwarten, dass in Spezifikationstexten nicht das gesamte Zeitarsenal einer Sprache verwendet wird, sondern allenfalls drei: eine Vergangenheit, die Gegenwart und eine Zukunft. Nachstehend werden einige Konzepte und Ansätze vorgestellt, die zukünftig in AutoAnnotator integriert werden könnten.

Time Markup Language (TimeML) ist eine XML-basierte Markupsprache, die zur Auszeichnung zeitlicher Informationen in Texten entwickelt wurde. TimeML kann gut maschinell verarbeitet werden und es wurden bereits Werkzeuge entwickelt, die sowohl die Annotationsarbeiten erleichtern, als auch die Ergebnisse grafisch aufbereiten können [PCI+03]. Wird ein Programm in AutoAnnotator integriert, das den Text mit TimeML anreichert, so könnten diese Informationen für die Annotation genutzt werden.

Pustejovsky et al. stellen in [PHS+03] den Timebank Corpus vor. Hierbei handelt es

sich um einen Textcorpus, der mit der TimeML annotiert wurde. Der Corpus umfasst 183 Artikel von Nachrichtenagenturen, die sorgfältig manuell annotiert wurden. Die Timebank war zunächst nur als Demonstrationscorpus für TimeML gedacht, kann aber nach mehreren Revisionen als Goldstandard für die Auszeichnung von zeitlichen Informationen angesehen werden.

Bethard und Martin stellen in [BM06] das *System for Textual Event Parsing (STEP)* vor. Die Autoren konzentrieren sich auf das Identifizieren von Ereignissen und setzen diese dann zueinander in Beziehung. Der entwickelte Algorithmus wurde mit 90 Prozent der Daten aus der Timebank trainiert und erreicht bei der Ermittlung von Ereignissen auf den verbleibenden 10 Prozent eine Precision von rund 86 Prozent und einen Recall von ca. 90 Prozent für Verben und von 72 Prozent bei Nomen. Die Evaluation des Ansatzes zeigte eine steile Lernkurve, sodass mit rund 10 Prozent der Trainingsdaten bereits eine Genauigkeit von über 93 Prozent erreicht werden kann. In [BNM+07] beschreiben sie, wie Zeitreihen erstellt werden können. Ihre Idee basiert darauf, dass die Ereignisse und die Akteure als Knoten eines Graphen abgebildet werden können. Zu Beginn ist der Graph leer; nach und nach werden dann die Informationen verschiedener Dokumente zum bestehenden Graphen hinzugefügt. Die Repräsentation ist hierbei ähnlich wie bei einem SalE-Graphen: Knoten repräsentieren Akteure und Ereignisse, die Kanten des Graphen drücken thematische Rollen aus. Zunächst besteht der Graph aus isolierten Teilgraphen, da die semantischen Informationen nicht über Satzgrenzen hinausgehen. Handelt es sich bei zwei Knoten um dieselbe Entität, so werden diese Knoten verschmolzen, um die Kohärenz des Graphen zu steigern. Kann die zeitliche Abfolge von Ereignissen ermittelt werden, so werden zusätzliche Kanten in den Graphen eingefügt, wodurch ebenfalls die Kohärenz erhöht wird.

Mani et al. beschreiben in [MVW+06] einen Ansatz aus dem maschinellen Lernen, mit dem sie Ereignisse in natürlicher Sprache zeitlich ordnen. Zunächst erzeugen sie eine (leere) Zeitachse. Dann versuchen sie, die Ereignisse verschiedener Dokumente auf dieser gemeinsamen Achse zu verankern. Der Algorithmus lernt hierbei von manuellen Ergebnissen und erreichte eine Genauigkeit von 93 Prozent mit einem nicht angepassten *Maximum Entropy classifier*.

7.6. Persistenter Speicher für Ontologiekonzepte

Während der Arbeit mit AutoAnnotator für die Evaluation wurde schnell klar, dass die Benutzerinteraktion verbessert werden kann. Vor einer ontologiebasierten Analyse muss zunächst das korrekte Konzept in der Ontologie bestimmt werden. Setzt man voraus, dass das Vokabular der Spezifikation kontrolliert verwendet wird, so werden für verschiedene Konzepte auch verschiedene Begriffe verwendet. Dann können vom Benutzer ausgewählte Wort-Konzept-Paare als Anlage zum Spezifikationstext gespeichert werden. Eine Integration dieser Anlage in die Ontologiekomponente von AutoAnnotator scheint einfach realisierbar, da die Ontologieanbindung bereits über einen Cache verfügt. Dieser Cache könnte vor dem Verbindungsaufbau mit der Ontologie bereits gefüllt werden.

Insbesondere wenn man den eingangs beschriebenen iterativen Prozess zugrundelegt,

wird der Spezifikationstext – wenn auch teilweise modifiziert – mehrfach mit AutoAnnotator bearbeitet. Verfügt man nach der ersten Iteration über eine Sammlung von Wort-Konzept-Paaren, kann der Annotationsprozess beschleunigt werden, da keine (oder zumindest weniger) Rückfragen an den Benutzer gestellt werden müssen. Dies sollte auch dazu führen, dass bei vielen Iterationen die Fehleranzahl bei der Konzeptauswahl sinkt, da diese weniger oft durchgeführt werden muss. Werden die Wort-Konzept-Paare gespeichert, können sie auch dazu verwendet werden, automatisch ein Glossar zu erstellen, das auf den Konzeptbeschreibungen in der Ontologie basiert.

8. Zusammenfassung

Das Erstellen von Modellen aus natürlichsprachlichen Anforderungen ist ein wichtiger Aufgabenbereich des RE. Praktiker wünschen sich Unterstützung bei der Modellentwicklung durch automatisierte Prozesse. SAL$_E$ MX kann aus annotierten Spezifikationen automatisch Softwaremodelle in der UML erzeugen. Zeitaufwändig ist hierbei das manuelle kodieren der Semantik mithilfe von SAL$_E$.

Diese Arbeit untersuchte, welche Informationen mithilfe von NLP-Programmen ermittelt und für eine automatische Annotation verwendet werden können. Die Analyse der verwandten Arbeiten zeigte, dass eine Kombination verschiedener Programme genutzt werden sollte. Aus diesem Grund erstellt AutoAnnotator aus den Analyseergebnissen verschiedener NLP-Programme zunächst eine breite Datenbasis. Aufgrund dieser Datenbasis werden anschließend semantische Annotationen ermittelt. Teilweise kann die Semantik nicht ausschließlich mit den Strukturinformationen ermittelt werden, die NLP-Programme liefern. Aus diesem Grund setzt AutoAnnotator in einem weiteren Analyseschritt auf digitalisiertes Allgemeinwissen, welches in Form von Ontologien verfügbar ist. Die Textanalyse ist dabei nicht auf eingeschränkte Sprache (Schablonen etc.) oder spezielle syntaktische Konstruktionen (bspw. Subjekt-Prädikat-Objekt-Sätze) beschränkt.

In der Evaluation zeigte sich, dass AutoAnnotator schnell eine grundlegende Annotation von Spezifikationen ermitteln kann. Die Informationen, die aus Ontologien bezogen werden können, tragen dabei stark zur Güte der ermittelten Annotation bei. Die Arbeit mit dem Programm zeigte aber auch, dass AutoAnnotator noch verbessert und benutzerfeundlicher gestaltet werden kann.

Da AutoAnnotator erweitert werden kann, können weitere Informationsquellen und Analysemethoden einfach eingebunden werden. Die Integration weiterer Informationsquellen – auch für Informationen, die bereits verfügbar sind – sollte geeignet sein, die ermittelte Annotation zu verbessern. Wie die Anbindung von domänenspezifischen Ontologien das Analyseergebnis verbessern kann, ist eine Frage, die in Zukunft untersucht werden sollte.

Im RECAA-Projekt soll AutoAnnotator zusammen mit weiteren Komponenten zu einem Produkt verschmolzen werden. Dieses Produkt soll den Anforderungsanalysten im gesamten Analyseprozess unterstützen: Angefangen vom Schreiben guter Spezifikationen (RESI [Bru09]) über die Modellextraktion (SAL$_E$ MX [Gel10]) hin zum Bewerten von Spezifikationsänderungen ([Tur10]) und Rückkoppeln von Modelländerungen in die Spezifikation (REFS [Der10]).

A. Ermittlung thematischer Rollen

Die von den NLP-Programmen ermittelten Informationen können dazu herangezogen werden, auf die thematischen Rollen von SalE zu schließen. Mithilfe von Ontologieabfragen können die ermittelten Rollengefüge genauer erfasst werden.

Im 5. Kapitel wurden einige Konstruktionen beispielhaft vorgestellt. Nachstehend finden sich die Konstruktionen, die AutoAnnotator derzeit erkennt. Die genannten Paket- und Klassennamen befinden sich unterhalb des Paketes `de.uka.ipd.autoAnnotator`.

Subphrasen

Verarbeitende Klasse: `reasoner.SubphraseExtractor`

Subphrasen können dem Syntaxbaum entnommen werden. Eine ausführliche Beschreibung findet sich in Abschnitt 5.4.2.

Rollen von Subphrasen

Verarbeitende Klasse: `reasoner.SubphraseRoleFixer`

Um Rollen von Subphrasen zu ermitteln, verwendet AutoAnnotator Signalwörter, die in der Zentralen Konfiguration vorgegeben werden können. Die folgenden Signalwörter werden derzeit verwendet:

- `CAU`: because
- `SUM`: if
- `INT`: while, whenever
- `TEMP`: to

Kann keine Rolle ermittelt werden und gibt der Benutzer keine Rolle an, so wird der Platzhalter `SP` verwendet.

Sichere Kommentare

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`

Die meisten sicheren Kommentare sind Nebenprodukte anderer Extraktionsprozesse. Es gibt jedoch auch Konstruktionen, die ausschließlich auf Kommentare abgebildet werden können.

Betrachtete typed dependency: `expl(a,b)`. Keine Besonderheiten.

Verbinden mehrwortiger Elemente

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`

Betrachtete typed dependencies: `nn(a,b)`, `auxpass(a,b)`. Keine Besonderheiten.

Betrachtete typed dependency: `num(a,b)`. Steht die Zahl im Satz nach dem anderen Element, so werden die Elemente verbunden (bspw. in *I stay in room 101.*).

Betrachtete typed dependency: `prt(a,b)`. Keine Besonderheiten.

Attribute

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependencies: `amod(a,b)`, `advmod(a,b)`, `not(a,b)`. Keine Besonderheiten.

Betrachtete typed dependency: `appos(a,b)`. Handelt es sich bei der Apposition um eine numerische Angabe (bspw. in *Tom, 30, ist Doktor geworden*), so wird ein Attribut erzeugt.

Betrachtete typed dependencies: `cop(a,b)` & `nsubj(a,c)`. Handelt es sich bei `a` um ein Adjektiv (Prüfung über POS-Tag `JJ*`; bspw. in *Das Auto ist grün*) oder einen numerischen Determinativ (bspw. in *Tom ist 30*), so wird ein Attribut erzeugt und verknüpft. `b` ist ein sicherer Kommentar.

Multiplizitäten

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependency: `det(a,b)`. Handelt es sich um ein Zahlwort, wird eine Multiplizität erzeugt. Andernfalls ist `b` sicher ein Kommentar.

Betrachtete typed dependency: `dep(a,b)`. Handelt es sich um ein Zahlwort, wird eine Multiplizität erzeugt.

Betrachtete typed dependency: `num(a,b)`. Steht die Zahl im Satz vor dem anderen Element, so wird eine Multiplizität erzeugt (bspw. in *I need 500 ml of milk.*).

FIN & FIC

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependencies: `cop(a,b)` & `nsubj(a,c)`. Handelt es sich bei `a` nicht um ein Adjektiv (Prüfung über POS-Tag nicht `JJ*`), so muss eine Rollenbeziehung eingeführt werden. `a` erhält die Rolle `FIC` und `c` die Rolle `FIN`. `b` ist ein sicherer Kommentar.

INST und MOD

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependency: `prep(a,'with')`, `pobj('with',b)`. Besitzt `a` bereits eine `METHODROLE`, so wird `b` mit der Platzhalterrolle `INSTMODP` gekennzeichnet. Auf allen

Worten mit der Rolle INSTMODP werden einige Anfragen durchgeführt, um das weitere Vorgehen zu bestimmen:

- Wenn b ein Instrument ist, wird es mit INSTP gekennzeichnet.
- Wenn b eine Handlungsweise ist, wird es mit MOD gekennzeichnet.

Betrachtete typed dependency: prep(a,'without'), pobj('without',b). Besitzt a bereits eine METHODROLE, so wird b mit der Platzhalterrolle INSTMODM gekennzeichnet. Anschließend werden dieselben Abfragen durchgeführt, wie im obigen Fall des INSTMODP. Vergeben wird dann jedoch die negative Rolle INSTM anstatt INSTP. Bei MOD wird without als Attribut gekennzeichnet, das sich auf b bezieht.

LDEST

Verarbeitende Klasse: reasoner.TypedDependencySentenceVisitor

Betrachtete typed dependency: prep(a,'to') & pobj('to',b). Wenn a die Rolle METHODROLE trägt, wird die Verarbeitung für b fortgeführt. War a ein Kommentar, wird er durch ein Wort mit der Rolle METHODROLE ersetzt und ebenso verfahren.

- Wenn b ein Ort ist, erhält es die Rolle LDEST.
- Wenn b eine Organisation ist, erhält es die Rolle RECP.
- Wenn b eine Person ist, erhält es die Rolle RECP.

Die Typprüfung für b wird aufgrund der Informationen des NER durchgeführt.

METHODROLE, ACT, STAT, TRANS

Verarbeitende Klasse: reasoner.TypedDependencySentenceVisitor

Für das Identifizieren einer METHODROLE wird nach Subjekt-Prädikat-Objekt-Konstruktionen gesucht, die weder eine Kopula (cop()) enthalten, noch ein Füllwort sind (expl()).

Betrachtete typed dependencies: nsubj(p,s), dobj(p,o), tmod(p,t). Das Subjekt erhält die Rolle AG, das Prädikat wird mit der Platzhalterrole METHODROLE versehen. Etwaig verfügbare Objekte erhalten die Rolle PAT und temporale Modifikatoren werden mit TEMP gekennzeichnet. Die Verfeinerung erfolgt mithilfe von Ontologieabfragen.

Verarbeitende Klasse: reasoner.OpenCyc Auf allen Worten mit der Rolle METHODROLE werden einige Anfragen durchgeführt, um das weitere Vorgehen zu bestimmen. Spezielle Anfragen werden vor allgemeinen gestellt, um eine möglichst spezielle Annotation zu erhalten.

- Erzeugende Handlung: Das Prädikat erhält die Rolle ACT; das Objekt die Rolle OPUSP.
- Zerstörende Handlung: Das Prädikat erhält die Rolle ACT; das Objekt die Rolle OPUSM.

- Verb kennzeichnet Teil-Ganzes-Beziehung: Das Prädikat wird sicher auskommentiert. Ganzes erhält die Rolle `OMN`, der Teil die Rolle `PARS`.
- Übergebende Handlung: Das Prädikat erhält die Rolle `ACT`. Kennzeichnen von *donor*, *recipiens* und *habitum*.
- Handlung: Das Prädikat erhält die Rolle `ACT`.
- Beziehung: Prädikat erhält die Rolle `STAT`.
- Zustandsübergang: Prädikat erhält die Rolle `TRANS`.
- Fallback auf die Rolle `ACT` (auch: wenn kein Konzept in Cyc gefunden werden kann).

Für Konstruktionen im Passiv gibt es vergleichbare typed dependencies die analog aufgelöst werden können.

OBJECTROLE

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependency: `pobj(a,b)`. Wird die Präposition nicht an einer anderen Stelle aufgelöst, wird hier eine Platzhalterrolle vergeben. Zugleich wird die Präposition als sicherer Kommentar markiert.

POSS und HAB

Verarbeitende Klasse: `reasoner.TypedDependencySentenceVisitor`
Betrachtete typed dependency: `poss(a,b)`. `a` erhält die Rolle `HAB`, `b` die Rolle `POSS`.

Betrachtete typed dependency: `prep(a,'of')` & `pobj('of',b)`. `a` erhält die Rolle `HAB`, `b` die Rolle `POSS`.

Referenzen

Verarbeitende Klassen:

- `reasoner.JavaRapReferencer`
- `reasoner.NamedEntityReferencer`
- `reasoner.WordReferencer`

Für die Auflösung von Referenzen gibt es verschiedene Optionen. Diese werden in den Abschnitten 4.4 und 5.4.6 ausführlich besprochen.

Ontologieanfragen

Verarbeitende Klasse: `reasoner.Opencyc`
Für Details zu den genannten Ontologiekonzepten, siehe Anhang F.

ACT

Eine `METHODROLE` kann durch `ACT` ersetzt werden, wenn das zugehörige Element ein *TemporallyExtendedThing* ist (vergleiche [KG08]).

INST

Ein `INSTMOD` kann durch `INST` ersetzt werden, wenn es sich bei dem zugehörigen Element um eine Spezialisierung eines *InanimateObject*s handelt.

MOD

Ein `INSTMOD` kann durch `MOD` ersetzt werden, wenn es sich bei dem zugehörigen Element um eine Spezialisierung eines *FeelingAttribute*s handelt.

OPUSP und OPUSM

Für die Bestimmung, ob ein Konzept um eine erschaffende Handlung ist, genügt die Abfrage, ob das Konzept eine Spezialisierung eines *CreationEvent*s ist.

Eine derart einfache Abfrage ist für die Rolle des Zerstörten `OPUSM` nicht möglich. Zwar gibt es das Konzept *DestructionEvent* in Cyc, jedoch ist *essen* ebenfalls eine Spezialisierung davon. Auf den ersten Blick mag dies unsinnig erscheinen, für das Lebensmittel jedoch, ist *essen* durchaus ein zerstörerischer Vorgang. Um die Rolle `OPUSM` eindeutig identifizieren zu können, sind daher noch weitere Ontologieabfragen notwendig.

STAT

Eine `METHODROLE` kann durch `STAT` ersetzt werden, wenn das Konzept in der Ontologie ein *Predicate* ist (vergleiche [KG08]).

TRANS

Eine `METHODROLE` kann durch `STAT` ersetzt werden, wenn das zugehörige Element ein *AtemporalThing* ist (vergleiche [KG08]).

B. Kurzeinführung in SAL$_E$

Tabelle 17: Linguistische Strukturen in SAL$_E$ (Auszug).

Linguistische Struktur	**Erklärung**
AG *agens*	Handelnde Person oder Sache, die eine Aktion ausführt.
PAT *patiens*	Person oder Sache, die von einer Handlung beeinflusst wird.
ACT (+AG +PAT) *actus*	Eine Handlung, ausgeführt von AG an/auf/mit PAT.
STAT (+AG +PAT) *status*	Eine Beziehung zwischen AG und PAT.
FIN (+FIC) *fingens* und *fictum*	Das FIN spielt die Rolle eines oder verhält sich wie ein FIC.
TEMP (+ACT) *tempus*	Eine zeitliche Beschreibung eines ACT.
EQD (+EQK) *equals drop*	Kennzeichnen eines Elements, das ersetzt werden soll.
EQK (+EQD) *equals keep*	Kennzeichnen des Elements, das statt EQD verwendet werden soll.

Thematische Rollen können dazu verwendet werden, Domänenmodelle aus natürlichsprachlichen Texten zu extrahieren [GT07]. Die Semantic Annotation Language for English (SAL$_E$) enthält derzeit 67 thematische Rollen (für eine vollständige Liste siehe [Gel10] und Anhang C) die auf den Arbeiten von Fillmore und Anderen basieren [Bru08, Fil69, Kri05, Rau88]. Für das Beispiel dieser Kurzeinführung reichen jedoch die Rollen in Tabelle 17 aus.

Als Beispiel werden hier die folgenden Sätze verwendet, sie sind [KL10] entnommen:

```
Chillies are very hot vegetables.
Mike Tyson likes green chillies.
Last week, he ate five of them.
```

Verwendet man SAL$_E$ als Annotationssprache, so muss man zunächst Sätze und Subphrasen mit `[` und `]` umschließen, anschließend muss man die Elemente auszeichnen. Elemente können hierbei einzelne Worte sein, aber auch zusammengesetzte Begriffe wie `Mike Tyson`. In diesem Fall müssen die Worte des Elements mit einem `_` verbunden werden. Nicht benötigte Elemente können mit `#` ausgelassen werden; sie werden im Diskursmodell noch als Kommentar geführt, haben aber keine Funktion mehr. Multiplizitäten werden mit einem `*` gekennzeichnet und Attribute mit einem `$`. Alle Elemente eines Satzes müssen entweder eine thematisch Rolle (oder mehrere) haben, ein Modifikator (ein Attribut oder eine Multiplizität) sein, oder auskommentiert werden.

Annotiert man die Beispielsätze, erhält man Folgendes:

```
1 [ Chillies|FIN #are $very $hot vegetables|FIC ].
2 [ Mike_Tyson|AG likes|STAT $green chillies|PAT ].
3 [ $Last week|TEMP, he|AG ate|ACT *five #of them|PAT ].
4 [ @he|EQD @Mike_Tyson|EQK ]. [ @them|EQD @Chillies|EQK ].
5 [ @chillies|EQD @Chillies|EQK ].
```

Die thematische Rolle *fingens* (`FIN`) wird verwendet um eine Person oder Sache zu kennzeichnen, die eine Funktion ausführt (eine Rolle spielt); umgekehrt wird *fictum* (`FIC`) verwendet um eben diese Funktion oder Rolle zu kennzeichnen. Das Verb `are` wird in der Beziehung *fingens*/*fictum* bereits kodiert und kann daher ausgelassen werden. `very` und `hot` sind Attribute; das erste bezieht sich auf `hot`, das zweite auf die `vegetables`. Ein Attribut bezieht sich immer auf das rechts folgende Element (das kein Kommentar ist); mittels Shifting können jedoch auch andere Worte attributiert werden. [21]

Im zweiten Satz kennzeichnet die Rolle *agens* (`AG`), dass das markierte Element die aktive Komponente der Aussage ist (aktiv nicht im grammatikalischen Sinne!). Das *agens* in diesem Fall ist `Mike_Tyson`, ein Element, welches durch Konkatenation aus `Mike` und `Tyson` entstand. Die Rolle *actus* (`ACT`) wird für Handlungen wie *von A nach B laufen* verwendet, während *status* (`STAT`) für generelle Aussagen wie *A wohnt in B* verwendet wird. Da `like` eine allgemeine Aussage ist, handelt es sich um einen *status*. Zuletzt betrachten wir die `chillies`. Sie sind die Sache, die vom *status* von `Mike_Tyson` beeinflusst wird; daher sind sie das *patiens* (`PAT`) in dieser Phrase.

Bei der Rolle `TEMP` handelt es sich um eine Zeitangabe, ein Datum oder eine Zeitspanne. Sie modifiziert die Rollen, mit denen sie verwendet wird. Im dritten Satz bezieht sich `week` auf den *actus* und `last` attributiert die `week`. `he` ist das *agens* des dritten Satzes und gehört zur Handlung `ate`. `them` sind auch dieses Mal das *patiens*, denn sie werden gegessen. `five` ist eine Multiplizität und bezieht sich auf `them`. Zuletzt können wir `of` auslassen.

Da wir als Analysten wissen, dass sich `he` auf `Mike_Tyson` bezieht und `them` auf `chillies`, fügen wir die Zusicherungen in Zeile 4 hinzu. SAL$_{\text{E}}$ MX ersetzt dann im Diskursmodell das Element, das mit EQD gekennzeichnet wurde durch das Element, das mit EQK gekennzeichnet wurde. Um im UML-Diagramm nicht `chillies` und `Chillies` vorzufinden, ersetzen wir das erste durch das zweite mithilfe der Zusicherung in Zeile 5. Die Normalisierung von Elementen würde zwar derartige Zusicherungen überflüssig machen, sie ist derzeit in SAL$_{\text{E}}$ MX jedoch noch nicht eingebunden.

Gibt man den so annotierten Text in SAL$_{\text{E}}$ MX ein, so wird das UML-Klassendiagramm erzeugt, das in Abbildung 21 zu sehen ist.

[21] In diesem Beispiel gibt es keine Attribute, die geshiftet werden müssen. Daher wird im weiteren Verlauf nicht mehr darauf eingegangen.

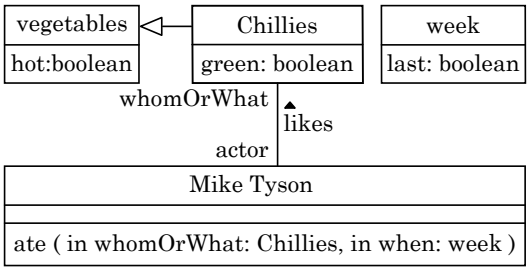


Abbildung 21: Ein von SAL$_{E}$ MX erzeugtes UML-Klassendiagramm.

C. Die thematischen Rollen von SAL$_E$

Die nachstehende Liste ist [Gel10] entnommen und enthält sämtliche in SAL$_E$ verfügbaren Rollen. Vorzeichenbehaftete Rollen (gekennzeichnet mit +/–) können nicht ohne das Vorzeichen verwendet werden. In der Annotation wird ein + mit `P` gekennzeichnet, ein − mit `M` (bspw.`OPUSP` oder `OPUSM`).

ACT	*actus*	eine Handlung
AG	*agens*	ein Handelnder
BEN+/–	*beneficiens*	der Nutznießer einer Handlung
CAU+/–	*causa*	ein Grund
COM	*comes*	ein Begleiter
COMP	*comparand*	etwas, mit dem verglichen wird
COMPII	*compariens*	etwas, das verglichen wird
CONS+/–	*consequentia*	das, was gilt, wenn eine Bedingung erfüllt ist
CONT	*contrarius*	ein Gegner
CONTII	*contrariens*	jemand, der einen Gegner hat
CREA	*creator*	Synonym für *agens*, vor allem im Zusammenhang mit *opus*
CRIT	*criterium*	ein Vergleichskriterium
CUR	*currens*	das Gegenwärtige, Laufende
DON	*donor*	jemand, der etwas gibt
DUX	*dux*	jemand, der begleitet wird
EQD	*equal-drop*	technische Rolle
EQK	*equal-keep*	technische Rolle
EXP	*experior*	jemand, der etwas erfährt
FAU+/–	*fautor*	jemand, der einem einen Vorteil verschafft
FAV+/–	*favor*	ein Vorteil
FIC	*fictum*	eine Funktion oder Rolle, die jemand oder etwas spielt
FIN	*fingens*	jemand, der eine Rolle oder Funktion einnimmt
FREQ	*frequens*	eine zeitlich Häufigkeit
HAB	*habitum*	etwas, das jemand hat oder das ihm gehört
INST+/–	*instrumentum*	ein Hilfsmittel bei einer Tätigkeit
INT	*intentio*	eine Absicht
IUS+/–	*ius*	ein Recht (Anrecht), das jemand hat
IUSII	*iurens*	jemand, der ein Recht hat, der etwas darf
LDEST	*locus destinatio*	ein Ort, wo etwas hin geht oder bis zu dem es reicht
LDIM	*locus dimensio*	eine Strecke (Länge, Höhe, Breite, ...)
LIM	*limes*	ein Pfad, ein Weg, den etwas nimmt
LOC	*locus*	ein Ort
LORIG	*locus origo*	ein Ort, wo etwas herkommt oder wo es anfängt
LTRANS	*locus transitum*	etwas, das sich durch dem Raum bewegt (oder bewegt wird)
MAG	*magister*	ein Lehrer, jemand, der einen anderen in die Lage versetzt, etwas zu tun
MOD	*modus*	die Art, wie jemand etwas tut
NOT	*notio*	eine Erfahrung oder eine Empfindung, die jemand macht
OBL	*obligens*	jemand, der jemand anderen verpflichtet, etwas zu tun
OMN	*omnium*	ein/das Ganze
OPUS+/–	*opus*	ein Werk, etwas, das geschaffen (+) oder zerstört (-) wird

PAR	*pars*	ein Teil eines Ganzen
PAT	*patiens*	ein Behandelter, das Ziel einer Handlung
PERM	*permitens*	jemand, der etwas erlaubt
POSS	*possesor*	der Besitzer einer Sache, der Halter, der „Haber“
POT+/–	*potentia*	ein Können, eine Fähigkeit
POTII	*potens*	jemand, der etwas kann
PRAE	*praecedens*	der Vorgänger, Vorläufer, was zuvor war
PROP	*proportiens*	ein Kriterium, an dem die Größe eines Vergleichskriteriums bestimmt wird
QUAL	*qualitas*	eine Qualität, eine Beschaffenheit
QUALII	*qualificatus*	ewas, das eine Qualität hat, dessen Beschaffenheit beschrieben wird
RECP	*recipient*	jemand, der etwas bekommt oder erhält
REQ+/–	*requisitum*	eine Anforderung, eine Pflicht
REQII	*requirens*	jemand, der eine Pflicht hat, an den eine Anforderung gestellt ist
STAT	*status*	ein Zustand
STIM	*stimulus*	jemand oder etwas, das eine Erfahrung oder Empfindung stimuliert
SUB	*substituens*	etwas, das ersetzt wird
SUBII	*substitutus*	etwas, das etwas anderes ersetzt
SUCC	*succedens*	der Nachfolger, etwas, das danach kommt
SUM	*sumtio*	eine Bedingung, eine Voraussetzung
TDEST	*tempus destinatio*	ein Zeitpunkt bis zu dem etwas ist, etwas war oder sein wird
TDIM	*tempus dimensio*	eine Zeitspanne, ein Zeitraum
TEMP	*tempus*	ein Zeitpunkt
THE	*thema*	ein Thema, der Inhalt
THEII	*thematus*	etwas, das ein Thema hat
TORIG	*tempus origo*	ein Zeitpunkt, seit dem etwas ist oder ab dem etwas war oder sein wird
TRANS	*transitus*	ein Zustandsübergang
TTRANS	*tempus transitum*	etwas, das sich durch die Zeit bewegt oder eine zeitliche Erstreckung hat
VOL+/–	*voluntas*	ein Wille, etwas, das jemand will
VOLII	*volens*	jemand, der etwas will

D. Das Penn-Tagset

Das Penn-Tagset mit den POS-Tags CC bis WRB sowie den zwölf Sonderkennzeichnungen für Zeichen und Symbole:

CC	Coordinating conjunction	TO	*to*
CD	Cardinal number	UH	Interjection
DT	Determiner	VB	Verb, base form
EX	Existential *there*	VBD	Verb, past tense
FW	Foreign word	VBG	Verb, gerund or present participle
IN	Preposition or subordinating conjunction	VBN	Verb, past participle
JJ	Adjective	VBP	Verb, non-3rd person singular present
JJR	Adjective, comparative	VBZ	Verb, 3rd person singular present
JJS	Adjective, superlative	WDT	*wh*-determiner
LS	List item marker	WP	*wh*-pronoun
MD	Modal	WP$	Possessive *wh*-pronoun
NN	Noun, singular or mass	WRB	*Wh*-adverb
NNS	Noun, plural	#	Pound sign
NP	Proper noun, singular	$	Dollar sign
NPS	Proper noun, plural	.	Sentence-final punctuation
PDT	Predeterminer	,	Comma
POS	Possessive ending	:	Colon, semi-colon
PP	Personal pronoun	(	Left bracket character
PP$	Possessive pronoun	)	Right bracket character
RB	Adverb	"	Straight double quote
RBR	Adverb, comparative	‘	Left open single quote
RBS	Adverb, superlative	“	Left open double quote
RP	Particle	’	Right close single quote
SYM	Symbol	”	Right close double quote

Tabelle 19: Das Penn-Tagset aus [MSM93].

E. Typed Dependencies

dep	dependent	**amod**	adjectival modifier
aux	auxiliary	**appos**	appositional modifier
auxpass	passive auxiliary	**advcl**	adverbial clause modifier
cop	copula	**purpcl**	purpose clause modifier
arg	argument	**det**	determiner
agent	agent	**predet**	predeterminer
comp	complement	**preconj**	preconjunct
acomp	adjectival complement	**infmod**	infinitival modifier
attr	attributive	**partmod**	participial modifier
ccomp	clausal compl. with int. subj.	**advmod**	adverbial modifier
xcomp	clausal compl. with ext. subj.	**neg**	negation modifier
compl	complementizer	**rcmod**	relative clause modifier
obj	object	**quantmod**	quantifier modifier
dobj	direct object	**tmod**	temporal modifier
iobj	indirect object	**measure**	measure-phrase modifier
pobj	object of preposition	**nn**	noun compound modifier
mark	marker (word introd. an advcl)	**num**	numeric modifier
rel	relative (word introd. a rcmod)	**number**	element of compound number
subj	subject	**prep**	prepositional modifier
nsubj	nominal subject	**poss**	possession modifier
nsubjpass	passive nominal subject	**possessive**	possessive modifier ('s)
csubj	clausal subject	**prt**	phrasal verb particle
csubjpass	passive clausal subject	**parataxis**	parataxis
cc	coordination	**punct**	punctuation
conj	conjunct	**ref**	referent
expl	expletive (expletive there)	**sdep**	semantic dependent
mod	modifier	**xsubj**	controlling subject
abbrev	abbreviation modifier		

F. ResearchCyc-Konstanten

In diesem Anhang finden sich die Erklärungen zu den verwendeten Ontologiekonstanten. Die Beschreibungen sind Cyc entnommen.

AtemporalThing

A specialization of Intangible (q.v.); the collection of all things that are „timeless“ in the sense of having no „location“ in time. It makes no sense to ask of an atemporal thing (e.g.) „When did it begin (or cease) to exist?“ Examples of atemporal things include sets, collections, numbers, vectors, and certain „abstract structures“ (such as the structure of a partial ordering); see SetOrCollection, Number-General, and VectorInterval. Note that while all atemporals are intangible, the converse is not true. Novels (see Novel-CW), languages, and geographical borders, for instance, are all intangible (i.e. they are not composed of or encoded in matter) but not atemporal, as they have a beginning and (in many cases) an end, and thus a temporal extent. See also the specialization AbstractThing; and cf. TemporalThing.

CreationEvent

A specialization of CreationOrDestructionEvent. In each instance of CreationEvent, at least one instance of SomethingExisting (q.v.) is brought into existence (see outputsCreated).

DestructionEvent

A collection of events and a specialization of CreationOrDestructionEvent. In each instance of this collection at least one instance of SomethingExisting (the inputsDestroyed) ceases to exist. Examples include deleting a computer file, chopping down a tree and breaking an agreement. For cases where the item destroyed is a material thing (i.e. an instance of PartiallyTangible), see the more specialized PhysicalDestructionEvent.

FeelingAttribute

A specialization of ScalarQuantity (q.v.). FeelingAttribute consists of all the varying degrees of such things as Happiness, Sadness and other types of emotions. Each instance is a scalar interval that consists of a number and emotional dimension. Rarely, if ever, are these things assigned specific values or ranges of values. But if certain emotional scales came to be used and it proved useful to represent them we could easily assign numeric values to these quantities. Generally, the instances of this collection will be denoted by terms formed by applying the GenericValueFunctions to specializations of this collection: e.g. (HighAmountFn Anger).

InanimateObject

A subcollection of PartiallyTangible. Each instance of InanimateObject is an (at least partly) tangible thing that is not currently a living structure. Things that were never alive, dead organisms, and dead (or completely non-functioning) organism parts are included in this collection. Examples: YaleUniversity, a piece of Meat, a dead armadillo, the StatueOfLiberty, and a pile of Sawdust. Two important specializations of this collection are InanimateObject-Natural and InanimateObject-NonNatural.

Predicate

A specialization of TruthFunction (q.v.). Each instance of Predicate is either a property of things (see UnaryPredicate) or a relationship holding between two or more things. Like other truth-functions, predicates, or rather the expressions that represent or denote them, are used to form sentences. More precisely, any CycL expression that denotes an instance of Predicate (and only such an expression) can appear in the „0th“ (or „arg0“) position (i.e. as the term following the opening parenthesis) of a CycLAtomicSentence (q.v.). Important specializations of Predicate include UnaryPredicate, BinaryPredicate, TernaryPredicate, QuaternaryPredicate, and QuintaryPredicate. Note that, despite its name, Predicate is a collection of relations, and not a collection of expressions that represent or denote such relations.

TemporallyExtendedThing

A specialization of TemporalThing that is by definition (see rewriteOf) the union of Situation and TimeInterval (qq.v.); it is disjoint with SomethingExisting. TemporallyExtendedThing is the collection of all things that are „extended in time“, as opposed to being „wholly present at a time“.

For example, an event is a temporally-extended thing, as it is extended in time; it is not wholly present at any interval that is properly subsumed by its temporal extent. Similarly for a time-interval, such as a particular CalendarYear. Conversely, a person is not a temporally-extended thing, as s/he exists at different times and is wholly present at each such time.

As the above examples illustrate, it is not the case that everthing that has a temporal extent (see temporalExtent) is a TemporallyExtendedThing.

G. Konfiguration

AutoAnnotator bietet dem Benutzer einige Wahlmöglichkeiten bezüglich der Verarbeitung des Textes. Diese Optionen können in der zentralen Konfigurationsdatei gesetzt werden. Die Komponenten, die AutoAnnotator benutzt, können ebenfalls parametrisiert werden. Die GUI speichert einige Informationen in einer eigenen Datei. Im Ressourcenverzeichnis der Java-Implementierung befinden sich beide Dateien:

- `config.props` Zentrale Konfiguration von AutoAnnotator und den externen Komponenten.
- `guiconfig.props` Konfigurationsdatei der GUI.

Sollen die Konfigurationen angepasst werden, sind diese Dateien in das Basisverzeichnis von AutoAnnotator zu kopieren und dort anzupassen.

Nachstehend sind die möglichen Parameter neben einer kurzen Erklärung aufgeführt. Jede Komponente verfügt über einen Namensaraum, welcher zuerst genannt wird. Die jeweiligen Parameter sind in der Konfigurationsdatei mit einem Punkt an den Namensraum anzuschließen; der zugewiesene Wert ist mit einem Gleichheitszeichen vom Parameternamen zu trennen. Beispiel: `Programm.4711.Namensraum.Parametername=Parameterwert`. Boole'sche Parameter sind mit einem Stern (*) versehen; mögliche Werte sind `true` und `false`. Kommentare können mit `#` eingeleitet werden.

AutoAnnotator-Konfiguration

Allgemeine Einstellungen
de.uka.ipd.autoAnnotator.AutoAnnotator

- stopAfterCollection* Nur Informationen sammeln, kein Reasoning (für Debugging)
- useNeReferences* Löse Referenzen über NE-Typen auf
- useWordReferences* Löse alle Worte mit Referenzen auf, die eine Rolle haben
- useOpenCyc* Spreche OpenCyc an um Unklarheiten zu betrachten

Referenzierung über Named Entities
de.uka.ipd.autoAnnotator.AutoAnnotator.reasoner.NamedEntityReferencer

- usedNeTypes Kommagetrennte Liste von NE-Typen, die zur Referenzauflösung verwendet werden sollen (`PERSON, ORGANIZATION, ...`)

Subphrasenextraktion
de.uka.ipd.autoAnnotator.AutoAnnotator.reasoner.SubphraseExtractor

- subphraseLabels Kommagetrennte Liste von Labeln, die im Syntaxbaum eine Phrase kennzeichnen
- signalCau Kommagetrennte Liste von Signalwörtern für begründende Sätze (because)
- signalInt Kommagetrennte Liste von Signalwörtern für Angaben eines Zieles (to)
- signalSum Kommagetrennte Liste von Signalwörtern für bedingende Sätze (if)
- signalTemp Kommagetrennte Liste von Signalwörtern für Zeitangaben (while)

Extraktion von Rollen via Typed Dependencies
de.uka.ipd.autoAnnotator.AutoAnnotator.reasoner.TypedDependencySentenceVisitor

- adjectiveRole Rolle, die für das beschriebene Element bei Adjektiven verwendet wird, wenn dieses Element noch keine Rolle hat und daher kein Attribut erhalten kann (default: `ADJ`)
- appositionRole Rolle, die für das beschriebene Element bei Appositionen verwendet wird, wenn dieses Element noch keine Rolle hat und daher kein Attribut erhalten kann (default: `APPOS`)
- copulaRole Rolle, die für das beschriebene Element in Kopulae verwendet wird, wenn dieses Element noch keine Rolle hat und daher kein Attribut erhalten kann (default: `COP`)
- objectRole Rolle für ein allgemeines Objekt (default: `OBJECTROLE`)
- recpLdesRole Platzhalterrolle für die Ermittlung von Empfängern oder Zielen (default `RECPLDEST`)

Bereinigung
de.uka.ipd.autoAnnotator.AutoAnnotator.reasoner.DummyRoleRemover

- dummyRoles Kommagetrennte Liste der Properties, die Platzhalterrollen enthalten, die entfernt werden sollen

Konfiguration der externen Komponenten

Stanford POS-Tagger
de.uka.ipd.autoannotator.tagger.stanfordpostagger

- modelFile Pfad zur Modelldatei, die verwendet werden soll

Stanford Parser
de.uka.ipd.autoAnnotator.tagger.StanfordParser

- parserFile Pfad zur Modelldatei, die verwendet werden soll
- parserOpts Parser Optionen
- redoPosTagging* Erneutes Taggen der Sätze oder Verwenden der bereits vorhandener POS-Tags

Stanford NER Client
de.uka.ipd.autoAnnotator.tagger.stanfordNamedEntityRecognizerClient

- server Name des Servers, der angesprochen werden soll
- port Port, auf dem der Server hört

Stanford NER Server
de.uka.ipd.autoAnnotator.tagger.StanfordNamedEntityRecognizerServer

- serverPort Port auf dem der Server hört
- serverJar Pfad zum JAR
- maxMem Maximale Menge an Arbeitsspeicher, die der Server verwenden soll (default: `500m`)
- mainClass Name der Hauptklasse (default: `edu.stanford.nlp.ie.NERServer`)

WordNet
de.uka.ipd.autoAnnotator.tagger.WordNetBaseformTagger

- server Servername des WordNet-Servers
- server_port* Port, auf dem der WordNet-Server hört
- dictfolder Pfad zum Dictionary
- uselocal* Soll ein lokales WordNet verwendet werden oder ein Server
- nonWordList Kommagetrennte Liste von Worten, die nicht geprüft werden

Java RAP
edu.nus.comp.nlp.tool.anaphoraresolution.JavaRAP

- substitution* (default `true`)
- referenceChain* (default `true`)
- mode (default: `TagPresent`)
- keepLog* (default `false`)
- displayLog* (default `true`)
- evaluationVerbose* (default `false`)
- displayResolvingResults* (default `false`)
- displaySubstitutionResults* (default `false`)
- writeSubstitutionResults* (default `false`)
- writeResolvingResults* (default `false`)

Java RAP - Charniak
edu.nus.comp.nlp.tool.anaphoraresolution.JavaRAP

- removeOldParse* Sollen vorherige Ergebnisse gelöscht werden (auf `true` setzen, wenn sich die Datei, nicht aber ihr Name ändert)
- parserHomeDir Basisverzeichnis des Parsers
- parserExecutableName Dateiname des Parsers (konfigurierbar, da plattformspezifisch)
- outputDir Ausgabeverzeichnis (Ausgabedatei wird nicht dauerhaft benötigt)
- tmpDir Temporäres Verzeichnis
- dataPath Datenverzeichnis
- parserOptions Optionen, die dem Parser übergeben werden sollen (default: `-l399`)

Stuttgart Tree Tagger
treetagger

- home Pfad zum Basisverzeichnis des TreeTaggers
- model Dateiname (inkl. abs. Pfad) zur Modelldatei

GUI-Konfiguration

- startFileDialogueInFolder Startpfad für den Dialog „Datei öffnen“

H. Installationen

In diesem Anhang sind Hinweise zur Installation und Anpassung verschiedener benötigter Komponenten angeführt.

Stanford Programme

Die Stanfordschen Programme basieren auf derselben Datengrundlage und entstammen einem einzigen Entwicklungsbaum. Tagger, Parser und NER wurden zu verschiedenen Zeitpunkten veröffentlicht und enthalten verschiedene Implementierungen von bestimmten Klassen. Daher ist es nicht möglich, alle JARs gleichzeitig in einer Java-VM zu laden und auszuführen.

Die Version vom 28. September 2008 des Taggers ist mit der Version vom 26. Oktober 2008 des Parsers kompatibel, sodass diese zeitgleich verwendet werden können.

Der NER erzeugt einen Konflikt mit allen getesteten Versionen der anderen beiden Programme. Da der NER jedoch von Haus aus über eine Serverimplementierung verfügt, wurde diese eingebunden. Die Serverversion der Version vom 16. Januar 2009 lässt sich mit dem folgenden Kommando starten:

```
java -mx500m -cp stanford-ner-with-classifier.jar \
  edu.stanford.nlp.ie.NERServer \
  -loadJarClassifier ner-eng-ie.crf-3-all2008.ser.gz \
  -port 19201
```

Vor einem derartigen Start ist jedoch ein JAR zu erstellen, welches neben dem Programm auch den gewählten Klassifizierer enthält:

```
cp stanford-ner.jar stanford-ner-with-classifier.jar
jar -uf stanford-ner-with-classifier.jar \
  classifiers/ner-eng-ie.crf-3-all2008.ser.gz
```

JavaRAP

JavaRAP verfügt über einige Konfigurationsparameter, welche teilweise über Kommandozeilenparameter gesetzt werden können. Darüber hinaus gab es einige fest hinterlegte Eigenschaften (wie die Dateinamen benötiger externer Komponenten), die nicht vom Benutzer gewählt werden konnten. Das Quellprogramm von JavaRAP wurde so modifizert, dass die Konfigurationsparameter über Java-Properties eingestellt werden können; darüber hinaus wurden für die fest hinterlegten Eigenschaften neue Konfigurationsparameter eingefügt. Hierdurch ist eine Anpassung der Parameter auch über die API möglich. JavaRAP kann unter `http://wing.comp.nus.edu.sg/~qiu/NLPTools/JavaRAP.html` bezogen werden; die angepasste Version findet sich unter `https://svn.ipd.uni-karlsruhe.de/repos/koerner/mx/projects/JavaRAP/`.

Charniak Parser

Für die Verwendung von JavaRAP wird der Parser von Eugene Charniak benötigt. Dieser steht in verschiedenen Versionen als Quellprogramm zum Download unter `ftp://ftp.cs.brown.edu/pub/nlparser/` zur Verfügung. Vorgesehen scheint eine Installation unter Linux. Wie in den FAQ auf der JavaRAP-Homepage beschrieben, kann die Version vom 22. Juni 2006 unter Cygwin (`http://www.cygwin.com/`) auch unter Windows kompiliert werden. Für eine Ausführung außerhalb einer Cygwin-Shell (also auch via Java-Systemaufrufe) ist es nötig, dass die DLL `cygwin1.dll` im selben Verzeichnis liegt, wie der Parser. Diese DLL ist im Installationsumfang

von Cygwin enthalten. Durch die oben beschriebenen Anpassungen von JavaRAP ist es (entgegen den Einträgen in den FAQ von JavaRAP) nicht notwendig, weitere Anpassungen am Parser vorzunehmen.

Literatur

[AG97] Ambriola, Vincenzo ; Gervasi, Vincenzo: Processing natural language requirements. In: *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–8186–7961–1, 36–46

[AG99] Ambriola, Vincenzo ; Gervasi, Vincenzo: Experiences with Domain-Based Parsing of Natural Language Requirements. In: *Proceedings of the Fourth International Conference on Applications of Natural Language to Information Systems, number 129 in OCG Schriftenreihe (Lecture Notes*, 1999, S. 145–148

[AG01] Ambriola, Vincenzo ; Gervasi, Vincenzo: On the parallel refinement of NL requirements and UML diagrams. In: *Proc. of the ETAPS 2001 Workshop on Transformations in UML*. Genova, Italy, April 2001

[Ali10a] Alias-i Inc.: *LingPipe*. http://alias-i.com/lingpipe, 2010. – Zuletzt besucht am 28.05.2010

[Ali10b] Alias-i Inc.: *LingPipe - Competition*. http://alias-i.com/lingpipe/web/competition.html, 2010. – Zuletzt besucht am 28.05.2010

[BM06] Bethard, Steven ; Martin, James H.: Identification of event mentions and their semantic class. In: *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, 2006, 146–154

[BMK07] Bethard, Steven ; Martin, James H. ; Klingenstein, Sara: *Timelines from Text: Identification of Syntactic Temporal Relations*. http://dx.doi.org/10.1109/ICOSC.2007.4338327. Version: 2007. – ICSC 2007. International Conference on 17-19 Sept. 2007 Page(s): 11–18

[BNM+07] Bethard, Steven ; Nielsen, Rodney ; Martin, James H. ; Ward, Wayne ; Palmer, Martha: *Semantic Integration in Learning from Text*. https://www.aaai.org/Library/Symposia/Spring/2007/ss07-06-004.php. Version: 2007

[Bru08] Brumm, Torben: *Erstellung eines Systems thematischer Rollen mit Hilfe einer multiplen Fallstudie*. http://www.ipd.uka.de/Tichy/uploads/arbeiten/135/StudienarbeitBrumm.pdf. Version: 2008. – Studienarbeit am Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Universität Karlsruhe (TH)

[Bru09] Brumm, Torben: *Rechnergestützte Verbesserung von textbasierten Softwarespezifikationen mit Hilfe von Ontologien*, Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Universität Karlsruhe (TH), Diplomarbeit, Juli 2009. http://www.ipd.uka.de/Tichy/theses.php?id=162

[CA07] Cheng, Betty H. C. ; Atlee, Joanne M.: Research Directions in Requirements Engineering. In: *Proc. Future of Software Engineering FOSE '07*, 2007, S. 285–303

[Cas09] Castilho, Richard E.: *TreeTagger for Java*. online. http://www.annolab.org/tt4j/index.html. Version: September 2009. – Zuletzt besucht am 15.05.2010

[Cha00] Charniak, Eugene: A Maximum-Entropy-Inspired Parser. In: Wiebe, Janyce (Hrsg.): *Proceedings of the First Conference of the North American Chapter of the Association for Computational Linguistics (NAACL 2000)*. Seattle, Washington :

Morgan Kaufmann Publishers, San Francisco, CA, USA, April 29 - May 04 2000, 132–139

[CMBT02] Cunningham, Hamish ; Maynard, Diana ; Bontcheva, Kalina ; Tablan, Valentin: GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA*, 2002

[Cyca] Cycorp Inc.: *OpenCyc*. http://www.opencyc.org/. – Zuletzt besucht am 28.05.2010

[Cycb] Cycorp Inc.: *ResearchCyc*. http://research.cyc.com/. http://research.cyc.com/. – Zuletzt besucht am 28.05.2010

[DB09] Deeptimahanti, Deva K. ; Babar, Muhammad A.: An Automated Tool for Generating UML Models from Natural Language Requirements. In: *ASE*, IEEE Computer Society, 2009. – ISBN 978–0–7695–3891–4, 680-682

[DBCM05] Dimitrov, Marin ; Bontcheva, Kalina ; Cunningham, Hamish ; Maynard, Diana: A Light-weight Approach to Coreference Resolution for Named Entities in Text. In: Branco, Antonio (Hrsg.) ; McEnery, Tony (Hrsg.) ; Mitkov, Ruslan (Hrsg.): *Anaphora Processing: Linguistic, Cognitive and Computational Modelling*, John Benjamins, 2005

[Der10] Derre, Emin B.: *Rückkopplung von Softwaremodelländerungen in textuelle Spezifikationen*, Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Karlsruher Institut für Technologie (KIT), Diplomarbeit, Juni 2010. http://www.ipd.uka.de/Tichy/theses.php?id=163

[DS99] Dawson, Linda ; Swatman, Paul A.: The use of object-oriented models in requirements engineering: a field study. In: *ICIS '99: Proceedings of the 20th international conference on Information Systems*, Association for Information Systems, 1999, 260–273

[DS08] Deeptimahanti, Deva K. ; Sanyal, Ratna: Static UML Model Generator from Analysis of Requirements (SUGAR). In: *ASEA '08: Proceedings of the 2008 Advanced Software Engineering and Its Applications*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3432–9, S. 77–84

[DS09] Deeptimahanti, Deva K. ; Sanyal, Ratna: An Innovative Approach for Generating Static UML Models from Natural Language Requirements. In: *Advances in Software Engineering*. Berlin, Heidelberg : Springer, 2009. – ISBN 978–3–642–10241–7 (Print) 978–3–642–10242–4 (Online)

[EQM96] Emam, Khaled E. ; Quintin, Soizic ; Madhavji, Nazim H.: User Participation in the Requirements Engineering Process: An Empirical Study. In: *Requirements Engineering* 1 (1996), März, Nr. 1, 4–26. http://dx.doi.org/10.1007/BF01235763. – DOI 10.1007/BF01235763. – ISSN 0947–3602 (Print) 1432–010X (Online)

[Fel98] Fellbaum, Christiane (Hrsg.): *WordNet: An Electronic Lexical Database*. Cambridge, MA : MIT Press, 1998. – ISBN 978–0–262–06197–1

[FGM05] Finkel, Jenny R. ; Grenager, Trond ; Manning, Christopher D.: Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In:

Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005). University of Michigan, USA : The Association for Computer Linguistics, Juni 2005, 363–370

[Fil69] Fillmore, Charles J.: Toward a modern theory of case. In: Reibel, D. A. (Hrsg.) ; Schane, S. A. (Hrsg.): *Modern Studies in English*. Prentice Hall, 1969, S. 361–375

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. (2006), S. 383 – 397

[Gel10] Gelhausen, Tom: *Modellextraktion aus natürlichen Sprachen*, Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Karlsruher Institut für Technologie (KIT), Diss., 2010

[Ger01] Gervasi, Vincenzo: The Cico Domain-Based Parser / University of Pisa, Dipartimento di Informatica. Version: November 2001. http://circe.di.unipi.it/~gervasi/main/publications.html. 2001 (TR-01-25). – Forschungsbericht

[GJ02] Gildea, Daniel ; Jurafsky, Daniel: Automatic labeling of semantic roles. In: *Computational Linguistics* 28 (2002), September, Nr. 3, S. 245–288. http://dx.doi.org/10.1162/089120102760275983. – DOI 10.1162/089120102760275983. – ISSN 0891–2017

[GM05] Ge, Ruifang ; Mooney, Raymond: A Statistical Semantic Parser that Integrates Syntax and Semantics. In: *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*. Ann Arbor, Michigan : Association for Computational Linguistics, June 2005, 9–16

[Goo00] Goos, Gerhard: *Vorlesungen über Informatik – Band 1: Grundlagen und funktionales Programmieren*. 3. überarbeitete Auflage. Berlin, Heidelberg : Springer, 2000. – ISBN 3–540–67270–2

[GT07] Gelhausen, Tom ; Tichy, Walter F.: Thematic Role based Generation of UML Models from Real World Requirements. In: *First IEEE International Conference on Semantic Computing (ICSC 2007)*. Irvine, CA, USA : IEEE Computer Society, September 2007, 282–289

[HG00] Harmain, Harmain M. ; Gaizauskas, Robert J.: CM-Builder: An Automated NL-Based CASE Tool. In: *ASE*, 2000, S. 45–54

[HKKS09] Hasegawa, Ryo ; Kitamura, Motohiro ; Kaiya, Haruhiko ; Saeki, Motoshi: Extracting Conceptual Graphs from Japanese Documents for Software Requirements Modeling. In: Kirchberg, Markus (Hrsg.) ; Link, Sebastian (Hrsg.): *APCCM* Bd. 96. Wellington, New Zealand : Australian Computer Society, Januar 2009 (CRPIT). – ISBN 978–1–920682–77–4, 87–96

[JML00] Juzgado, Natalia J. ; Moreno, Ana M. ; López, Marta: How to Use Linguistic Instruments for Object-Oriented Analysis. In: *IEEE Software* 17 (2000), Nr. 3. http://dx.doi.org/10.1109/52.896254. – DOI 10.1109/52.896254

[JVKV09] Jain, Prateek ; Verma, Kunal ; Kass, Alex ; Vasquez, Reymonrod G.: Automated review of natural language requirements documents: generating useful warnings with user-extensible glossaries driving a simple state machine. In: Deshpande, Kiran (Hrsg.) ; Jalote, Pankaj (Hrsg.) ; Rajamani, Sriram K. (Hrsg.): *ISEC*, ACM, 2009. – ISBN 978–1–60558–426–3, 37-46

[KB09a] Körner, Sven J. ; Brumm, Torben: Improving Natural Language Specifications with Ontologies. In: *SEKE*, Knowledge Systems Institute Graduate School, Juli 2009. – ISBN 1–891706–24–1, 552-557

[KB09b] Körner, Sven J. ; Brumm, Torben: RESI - A Natural Language Specification Improver. In: *Proceedings of the IEEE ICSC 2009* IEEE, 2009

[KDGL] Körner, Sven J. ; Derre, Bugra ; Gelhausen, Tom ; Landhäusser, Mathias: *RECAA – The Requirements Engineering Complete Automation Approach.* online. https://svn.ipd.uni-karlsruhe.de/trac/mx. – Zuletzt besucht am 28.05.2010

[KG08] Körner, Sven J. ; Gelhausen, Tom: Improving Automatic Model Creation using Ontologies. In: Knowledge Systems Institute (Hrsg.): *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*, 2008, 691–696

[KL10] Körner, Sven J. ; Landhäusser, Mathias: Semantic Enriching of Natural Language Texts with Automatic Thematic Role Annotation. In: Hopfe, C. J. (Hrsg.): *International Conference on Applications of Natural Language to Information Systems, NLDB 2010,Cardiff, UK, June 23-25* Bd. 6177. Berlin, Heidelberg : Springer, Juli 2010 (Lecture Notes in Computer Science), 92–99

[KM02] Klein, Dan ; Manning, Christopher D.: Fast Exact Inference with a Factored Model for Natural Language Parsing. In: Becker, Suzanna (Hrsg.) ; Thrun, Sebastian (Hrsg.) ; Obermayer, Klaus (Hrsg.): *Advances in Neural Information Processing Systems 15 - Neural Information Processing Systems, NIPS 2002*, MIT Press, 2002. – ISBN 0–262–02550–7, 3–10

[KM03] Klein, Dan ; Manning, Christopher D.: Accurate Unlexicalized Parsing. In: *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics.* Morristown, NJ, USA : Association for Computational Linguistics, 2003, S. 423–430

[Kof04a] Kof, Leonid: *An Application of Natural Language Processing to Requirements Engineering – A Steam Boiler Case Study.* 2004. – Contribution to SEFM 2004

[Kof04b] Kof, Leonid: Natural Language Procesing for Requirements Engineering: Applicability to Large Requirements Documents. In: Russo, Alessandra (Hrsg.) ; Garcez, Artur (Hrsg.) ; Menzies, Tim (Hrsg.): *Automated Software Engineering, Proceedings of the Workshops.* Linz, Austria, September 2004. – In conjunction with the 19th IEEE Internationl Conference on Automated Software Engineering

[Kof05] Kof, Leonid: Natural Language Processing: Mature Enough for Requirements Documents Analysis? In: Montoyo, Andrés (Hrsg.) ; Muñoz, Rafael (Hrsg.) ; Métais, Elisabeth (Hrsg.): *NLDB* Bd. 3513. Berlin, Heidelberg : Springer, Juni 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26031–5, S. 91–102

[KP03] Kingsbury, Paul ; Palmer, Martha: PropBank: The next level of TreeBank. In: *Proceedings of Treebanks and Lexical Theories.* Växjö, Sweden, 2003

[Kri05] Krifka, Manfred: *Thematische Rollen.* http://amor.rz.hu-berlin.de/~h2816i3x/GK_Semantik_10_ThematischeRollen.pdf. Version: Juni 2005

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction.* 3. Auflage. Boston, MA : Addison-Wesley, 2003. – ISBN 0201707101

[Lan08] Landhäusser, Mathias: *Automatische Erzeugung von Prüflisten zur spezifikationsbezogenen Beurteilung der Vollständigkeit von UML-Modellen.* Version: Juni 2008. http://www.ipd.uka.de/Tichy/theses.php?id=156 Studienarbeit am Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Universität Karlsruhe (TH)

[LBF97] Lowe, John B. ; Baker, Collin F. ; Fillmore, Charles J.: A frame-semantic approach to semantic annotation. In: *Proceedings of ACL SIGLEX Workshop on Tagging Text with Lexical Semantics.* Washington, D.C., 1997, S. 18–24

[LHB92] In: Lopez, Oscar P. ; Hayes, Fiona ; Bear, Stephen: *Lecture Notes in Computer Science.* Bd. 593: *OASIS: An Object-Oriented Specification Language.* Berlin, Heidelberg : Springer, 1992. – ISBN 978–3–540–55481–3, 348–363

[Liu04] Liu, Hugo: *Montylingua: An end-to-end natural language processor with common sense.* http://web.media.mit.edu/~hugo/montylingua. Version: 2004

[LK95] Loucopoulos, Pericles ; Karakostas, Vassilios: *System Requirements Engineering.* New York, NY, USA : McGraw-Hill, Inc., 1995 (McGraw-Hill international series in software engineering). – ISBN 0–07–707843–8

[LL94] Lappin, S. ; Leass, H.J.: An algorithm for Pronominal Anaphora Resolution. In: *Computational Linguistics* 20 (1994), Dezember, Nr. 4, 535–561. http://www.ldc.upenn.edu/acl/J/J94/J94-4002.pdf

[MFI04] Mich, Luisa ; Franch, Mariangela ; Inverardi, Pierluigi N.: Market research for requirements analysis using linguistic tools. In: *Requirements Engineering* 9 (2004), Februar, Nr. 1, 40–56. http://dx.doi.org/10.1007/s00766-003-0179-8. – DOI 10.1007/s00766–003–0179–8. – ISSN 0947–3602 (Print) 1432–010X (Online)

[Mic] Microsoft Corporation: *.NET Framework.* http://www.microsoft.de/net. – Zuletzt besucht am 28.05.2010

[Mil95] Miller, George A.: WordNet: A lexical database for English. In: *Communications of the ACM* 38 (1995), Nr. 1, S. 39–41. http://dx.doi.org/10.1145/219717.219748. – DOI 10.1145/219717.219748

[Mil09] Miller, George A.: *WordNet - About Us.* online. http://wordnet.princeton.edu/. Version: 2009. – Zuletzt besucht am 28.05.2010

[Mit99] Mitkov, Ruslan: *Anaphora Resolution: The state of the art.* http://clg.wlv.ac.uk/papers/mitkov-99a.pdf. Version: 1999. – Basiert auf den COLING'98/ACL'98 Tutorien zum Auflösen von Anaphern

[MJ] Manning, Chris ; Jurafsky, Dan: *The Stanford Natural Language Processing Group.* online. http://nlp.stanford.edu. – Zuletzt besucht am 28.05.2010

[MM03] Miller, Joaquin ; Mukerji, Jishnu: *MDA Guide Version 1.0.1.* http://www.omg.org/cgi-bin/doc?omg/03-06-01. Version: Juni 2003

[MM08a] Marneffe, Marie-Catherine de ; Manning, Christopher D. ; Stanford NLP Group, Stanford University (Hrsg.): *Stanford typed dependencies manual.* Stanford, CA: Stanford NLP Group, Stanford University, September 2008. http://nlp.stanford.edu/software/dependencies_manual.pdf

[MM08b] Marneffe, Marie-Catherine de ; Manning, Christopher D.: The Stanford typed dependencies representation. In: *COLING Workshop on Cross-framework and Cross-domain Parser Evaluation*, 2008, 1–8

[Mor97] Moreno, Ana M.: Object-Oriented Analysis from Textual Specifications. In: *In Proc. of 9 th International Conference on Software Engineering and Knowledge Engineering (SEKE 97)*, 1997

[MPEP08] Montes, Azucena ; Pacheco, Hasdai ; Estrada, Hugo ; Pastor, Oscar: Conceptual Model Generation from Requirements Model: A Natural Language Processing Approach. In: Kapetanios, Epaminondas (Hrsg.) ; Sugumaran, Vijayan (Hrsg.) ; Spiliopoulou, Myra (Hrsg.): *NLDB* Bd. 5039. Berlin, Heidelberg : Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–69857–9, 325-326

[MR97] Moreno, Ana M. ; Riet, Reind P. d.: *Justification Of The Equivalence Between Linguistic And Conceptual Patterns For The Object Model*. 1997

[MSM93] Marcus, Mitchell P. ; Santorini, Beatrice ; Marcinkiewicz, Mary A.: Building a Large Annotated Corpus of English: The Penn Treebank. In: *Computational Linguistics* 19 (1993), Juni, Nr. 2, S. 313–330

[MVW+06] Mani, Inderjeet ; Verhagen, Marc ; Wellner, Ben ; Lee, Chong M. ; Pustejovsky, James: Machine learning of temporal relations. In: *ACL-44: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Morristown, NJ, USA : Association for Computational Linguistics, 2006, 753–760

[OLR01] Overmyer, Scott P. ; Lavoie, Benoit ; Rambow, Owen: Conceptual Modeling through Linguistic Analysis Using LIDA. In: *ICSE*. Washington, DC, USA : IEEE Computer Society, Mai 2001. – ISBN 0–7695–1050–7, 401–410

[PCI+03] Pustejovsky, James ; Castano, José ; Ingria, Robert ; Saurí, Roser ; Gaizauskas, Robert ; Setzer, Andrea ; Katz, Graham ; Radev, Dragomir: TimeML: Robust Specification of Event and Temporal Expressions in Text. In: *fifth International Workshop on Computational Semantics (IWCS-5)*, 2003. – Also published in AAAI TechReport SS-03-07

[PHS+03] Pustejovsky, James ; Hanks, Patrick ; Saurí, Roser ; See, Andrew ; Gaizauskas, Robert ; Setzer, Andrea ; Sundheim, Beth ; Radev, Dragomir ; Day, David ; Ferro, Lisa ; Lazo, Marcia: *The TIMEBANK Corpus*. http://ucrel.lancs.ac.uk/publications/CL2003/papers/pustejovsky.pdf. Version: 2003

[PPZ06] Pazienza, Maria T. ; Pannecchiotti, Marco ; Zanzotto, Fabio M.: Mixing WordNet, VerbNet and PropBank for studying verb relations. In: *Proceedings of LREC 2006*. Genova, Italy, Mai 2006

[QKC04] Qiu, Long ; Kan, Min-Yen ; Chua, Tat-Seng: A Public Reference Implementation of the RAP Anaphora Resolution Algorithm. In: *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)* 1 (2004), 291–294. http://dblp.uni-trier.de/db/journals/corr/corr0406.html#cs-CL-0406031

[Rau88] Rauh, Gisa: *Tiefenkasus, thematische Relationen und Thetarollen*. Tübingen, Germany : Gunter Narr Verlag, 1988. – ISBN 3878083696

[RS07] Rosenberg, Doug ; Stephens, Matt: *Use Case Driven Object Modeling with UML – Theory and Practice*. Apress, 2007. – ISBN 1590597745

[Rup09] Rupp, Chris: *Requirements-Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis*. 5., aktualisierte und erw. Aufl. Hanser Fachbuchverlag, 2009 http://www.hanser.de/buch.asp?isbn=978-3-446-41841-7. – ISBN 3–446–40509–7,978–3–446–40509–7,978–3–446–41841–7

[Rya93] Ryan, Kevin: The role of natural language in requirements engineering. In: *Proceedings of IEEE International Symposium on Requirements Engineering* IEEE, 1993, 240-242

[San90] Santorini, Beatrice: Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision) / University of Pennsylvania Department of Computer and Information Science. Version: 1990. http://repository.upenn.edu/cis_reports/570. 1990 (MS-CIS-90-47). – Forschungsbericht

[Sch94] Schmid, Helmut: Probabilistic Part-of-Speech Tagging Using Decision Trees. In: *Proceedings of International Conference on New Methods in Language Processing*, 1994

[Sch95] Schmid, Helmut: Improvements In Part-of-Speech Tagging With an Application To German. In: *Proceedings of the EACL SIGDAT-Workshop*. Dublin, 1995, S. 47–50

[Sch06] Schuler, Karin K.: *VerbNet: A Broad-Coverage, Comprehensive Verb Lexicon*, University of Pennsylvania, Diss., 2006. http://verbs.colorado.edu/~kipper/Papers/dissertation.pdf

[SNL01] Soon, Wee M. ; Ng, Hwee T. ; Lim, Daniel Chung Y.: A machine learning approach to coreference resolution of noun phrases. In: *Computational Linguistics* 27 (2001), Nr. 4, 521–544. http://dx.doi.org/10.1162/089120101753342653. – DOI 10.1162/089120101753342653. – ISSN 0891–2017

[The09a] The Stanford Natural Language Processing Group: *Stanford Log-linear Part-Of-Speech Tagger*. http://nlp.stanford.edu/software/tagger.shtml. http://nlp.stanford.edu/software/tagger.shtml. Version: Dezember 2009. – Zuletzt besucht am 28.05.2010

[The09b] The Stanford Natural Language Processing Group: *The Stanford Parser: A statistical parser*. http://nlp.stanford.edu/software/lex-parser.shtml. http://nlp.stanford.edu/software/lex-parser.shtml. Version: Juli 2009. – Zuletzt besucht am 28.05.2010

[Tur10] Turek, Alexander M.: *Automatische Analyse der Auswirkungen von Änderungen an natürlichsprachlichen Spezifikationen auf Softwaremodelle*, Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Programmiersysteme Prof. Dr. Walter F. Tichy, Fakultät für Informatik, Karlsruher Institut für Technologie (KIT), Diplomarbeit, Juni 2010. http://www.ipd.uka.de/Tichy/theses.php?id=164

[Uni99] University of Pennsylvania Department of Computer and Information Science: *The Penn Treebank Project*. online. http://www.cis.upenn.edu/~treebank. Version: Februar 1999. – Zuletzt besucht am 28.05.2010

[VK08] Verma, Kunal ; Kass, Alex: Requirements Analysis Tool: A Tool for Automatically Analyzing Software Requirements Documents. In: Sheth, Amit P. (Hrsg.) ; Staab,

Steffen (Hrsg.) ; Dean, Mike (Hrsg.) ; Paolucci, Massimo (Hrsg.) ; Maynard, Diana (Hrsg.) ; Finin, Timothy W. (Hrsg.) ; Thirunarayan, Krishnaprasad (Hrsg.): *International Semantic Web Conference* Bd. 5318. Berlin, Heidelberg : Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–88563–4, 751–763

[VKV+10] Verma, Kunal ; Kass, Alex ; Vasquez, Reymonrod G. ; Sarkar, Santonu ; Sharma, Vibhu: *Aligning Requirements Documents to Industry-Specific Process Models*. 2010. – Eingereicht zur Konferenz RE2010.

[ZHS97a] Zifonun, Gisela ; Hoffmann, Ludger ; Strecker, Bruno ; Eroms, Hans-Werner (Hrsg.) ; Stickel, Gerhard (Hrsg.) ; Zifonun, Gisela (Hrsg.): *Schriften des Instituts für Deutsche Sprache ; Bd 7,1*. Bd. 1: *Grammatik der deutschen Sprache*. Berlin ; New York : De Gruyter, 1997

[ZHS97b] Zifonun, Gisela ; Hoffmann, Ludger ; Strecker, Bruno ; Eroms, Hans-Werner (Hrsg.) ; Stickel, Gerhard (Hrsg.) ; Zifonun, Gisela (Hrsg.): *Schriften des Instituts für Deutsche Sprache ; Bd 7,2*. Bd. 2: *Grammatik der deutschen Sprache*. Berlin ; New York : De Gruyter, 1997

[ZHS97c] Zifonun, Gisela ; Hoffmann, Ludger ; Strecker, Bruno ; Eroms, Hans-Werner (Hrsg.) ; Stickel, Gerhard (Hrsg.) ; Zifonun, Gisela (Hrsg.): *Schriften des Instituts für Deutsche Sprache ; Bd 7,3*. Bd. 3: *Grammatik der deutschen Sprache*. Berlin ; New York : De Gruyter, 1997