Eine Architektur für Programmsynthese aus natürlicher Sprache

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

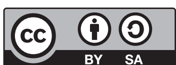
Dissertation

von

Mathias Landhäußer

aus Karlsruhe

Tag der mündlichen Prüfung:	22.04.2016
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Tanja Schultz



Dieses Werk ist lizenziert unter einer Creative Commons Namensnennung –
Weitergabe unter gleichen Bedingungen 3.0 Deutschland Lizenz
(CC BY-SA 3.0 DE): http://creativecommons.org/licenses/by-sa/3.0/de/

Zusammenfassung

Diese Arbeit entwirft eine domänenunabhängige Architektur, die Programmsynthese aus natürlichsprachlichen Texten ermöglicht.

Die Erkenntnisse der verwandten Arbeiten zeigen, dass eine Programmsynthese aus natürlicher Sprache mit zwei Einschränkungen erreicht werden kann: Man kann erstens den Umfang der erlaubten Eingabesprache einschränken, d.h. die Grammatik und das Vokabular. Man kann zweitens das anzusprechende System so stark einschränken, dass damit die sprachliche Komplexität automatisch geringer wird. Unabhängig von der Herangehensweise wird meist die Analyse der Eingabe mit der Code-Erzeugung bzw. Steuerung derart verwoben, dass sie untrennbar oder getrennt voneinander nicht nutzbar sind. Domäneninformationen werden dabei häufig in die Textanalyse mit einbezogen, wodurch die Ergebnisse gut aber die Analysen nicht übertragbar sind. Ein Domänenwechsel ist damit gleichbedeutend mit einer Neuentwicklung.

Bestehende Lösungen betrachten die Eingabetexte meist aus der Sicht der Programmierung (z.B. „Wie sollte man for-Schleifen beschreiben?") und setzen die identifizierten Sprachmuster dann um. Die in der vorliegenden Arbeit vorgestellte Architektur betrachtet das Problem vom entgegengesetzten Standpunkt aus: Wir fragen uns bspw. „Wie drückt ein Mensch Wiederholungen aus?" und ermitteln die zugehörigen sprachlichen Phänomene. Dabei wurden keine neuen computerlinguistischen Verfahren entwickelt (z.B. Wortartmarkierer), sondern wir stützen unsere Analysen auf bewährte Werkzeuge der Computerlinguistik und interpretieren deren Ergebnisse in unserem Kontext. Erst danach findet eine Abbildung auf Programmierkonstrukte statt. Dadurch entkoppeln wir die Sprachanalysen von der Programmsynthese. Einige Sprachanalysen benötigen jedoch Informationen über die anzusteuernde API (z.B. die Namen der Klassen, deren Methoden usw.). Hier erreichen wir die Entkopplung dadurch, dass wir den Analysen das Wissen über die anzusteuernde API in Form eines Modells bereitstellen. Es enthält eine sprechende Beschreibung der Ziel-API sowie weitere sprachliche Informationen wie z.B. Synonyme für Methodennamen.

Die vorgestellte Architektur, der Natural Language Command Interpreter (NLCI), bearbeitet den Text mit einem Fließband (engl. *pipeline*), wobei jede Textanalyse eine Stufe

des Fließbands darstellt. Alle Analysestufen sind als Einschübe (engl. *plug ins*) für die vorgestellte Rahmenarchitektur konzipiert und sind somit austauschbar. Sie können bei ihrer Analyse auf Ergebnisse vorangegangener Stufen zugreifen und/oder weitere Informationen heranziehen. Der vorgestellte Prototyp enthält Einschübe für die Identifikation der angesprochenen API-Klassen, API-Methoden und von Kontrollstrukturen sowie für die Rekonstruktion der Aktionsreihenfolge. Das API-Modell steht allen Einschüben zur Verfügung. Die Code-Erzeugung ist die letzte Stufe des Fließbands und hat somit Zugriff auf die Ergebnisse der Sprachanalysen und muss diese lediglich auswerten. Die Code-Erzeugung ist wiederverwendbar, da sie nur von der Programmiersprache abhängig ist, nicht jedoch von der betrachteten API, da diese über das Modell zugreifbar ist.

Die prototypische Implementierung von NLCI wurde in zwei Fallstudien untersucht und die erzielten Ergebnisse ausgewertet. Die erste Anwendung ist openHAB, eine zentrale Steuerung für intelligente Häuser. NLCI erzeugt in dieser Fallstudie aus Befehlen Steuerkommandos, z.B. zum Einschalten eines Lichts. Die zweite Fallstudie ist Alice, eine Animationssoftware mit der man kurze Trickfilme erzeugen kann. NLCI erzeugt hier aus Drehbüchern die Befehle zur Erzeugung von Animationen. Zur Forschungslenkung und zur Evaluation wurde ein Textkorpus erstellt, der derzeit 146 Drehbücher enthält.

Die Ergebnisse sind auch ohne Einschränkung der Eingabesprache zufriedenstellend, hängen jedoch von den Ergebnissen der verwendeten computerlinguistischen Werkzeuge ab. NLCI erzeugt in der umfangreicheren und komplexeren Domäne von Alice die gewünschten Methodenaufrufe inkl. den benötigten Methodenargumenten mit einer Ausbeute von 71% und einer Präzision von 87% ($F_1 = 78\%$). Die Evaluation zeigt auch, dass die Sprachanalysen von der Domäne entkoppelt wurden. Der Aufwand für einen Domänenwechsel ist klein und die Analysen konnten für beide Anwendungen verwendet werden.

Vorwort

„Every successful individual knows that his or her achievement depends on a community of persons working together.“

Paul Ryan

Es kommt auf den Kontext an. *Immer.* Meine Promotion war eine große Herausforderung, die ich nur meistern konnte, weil mein Kontext stimmte.

Mein größter Dank gilt Daniela und Sophie. Daniela danke ich dafür, dass sie immer für mich da ist und mir den Kontextwechsel vom Elfenbeinturm in die Realität leicht gemacht, mich aber manchmal vor genau diesem Wechsel bewahrt hat. Ganz besonders im letzten Jahr. Sophie danke ich dafür, dass sie jeden Tag aufs Neue (meine) Prioritäten setzt. Seit sie bei uns ist, hat mein Leben eine Farbe mehr. Danken möchte ich auch meinen Eltern und meiner gesamten Familie – ohne sie wäre ich nicht der, der ich bin.

Großer Dank gebührt Walter: Er bot mir die Möglichkeit, an seinem Lehrstuhl zu promovieren – obwohl ich etwas Verrücktes machen wollte (oder gerade deswegen?). In diesem Kontext möchte ich auch Tom und Sven erwähnen, ohne die sich diese Gelegenheit nie eröffnet hätte. Von und mit ihnen habe ich viel gelernt. Danken möchte ich allen meinen Kollegen, die mir am Anfang, in der Mitte und am Ende meiner Promotionszeit sehr geholfen haben; ohne die freundliche Atmosphäre am Lehrstuhl, das Competence Center, gemeinsame Fachsimpeleien und die meist konstruktive aber immer ehrliche Kritik wäre es nicht gegangen. Hildegard, Ruth und Heinz danke ich dafür, dass es im administrativen Kontext immer einen sehr kurzen Dienstweg gab. Tanja danke ich für die hilfreichen Kommentare zu meiner Dissertation.

Einen erheblichen Teil meiner Bruttozeit am Lehrstuhl habe ich im Kontext der Lehre verbracht. Und rückblickend muss ich feststellen: Nein, Halten von Vorlesungen und Gestalten von Übungen bringt einen in der Promotion nicht weiter. Wirklich nicht. Dennoch bin ich froh, dass ich das volle Spektrum miterleben durfte: Lehre im Kontext von dünn besetzten Seminaren ist etwas gänzlich Anderes als bei intensiven Vertiefungsveran-

staltungen oder bei Anfängervorlesungen mit 600 Teilnehmern. Meistens hat das großen Spaß gemacht – und objektiv betrachtet habe ich auch dabei viel gelernt. Ein wichtiger Bestandteil der Lehre sind die Abschlussarbeiter, von denen ich 27 (!) (mit-)betreuen durfte. Viele von ihnen haben sehr gute Ergebnisse erzielt, fünf Abschlussarbeiten führten zu Veröffentlichungen und einige meiner Abschlussarbeiter streben nach Abschluss ihres Studiums eine Promotion an. Ich wünsche ihnen allen von Herzen Erfolg für die Zukunft!

In den vergangenen fünfeinhalb Jahren lief vieles gut, manches exzellent aber bei Weitem nicht alles perfekt. Betrachtet man eine vermeintlich verfahrene Situation isoliert von ihrem Kontext oder aus einem anderen Blickwinkel, so ergeben sich oft neue Ideen und Ansätze. Mein besonderer Dank gilt daher Thomas und Thomas. Sie haben ein Talent dafür, Situationen zu analysieren, Probleme zu sezieren und Dinge ins rechte Licht zu rücken. Die Gespräche mit ihnen haben mir oft geholfen.

Es kommt auf den Kontext an. *Immer.* Manchmal ist es aber auch gut, Aussagen völlig aus ihrem Kontext zu reißen. Daher einfach nur:

Danke!

Karlsruhe im Juni 2016

Inhaltsverzeichnis

1 Die Vision der Programmierung in natürlicher Sprache 1

1.1 Von der Rechenmaschine zum intelligenten Gesprächspartner 2
1.2 Abstraktionsniveau und Zielgruppe 3
1.3 Eine Architektur für natürlichsprachliche Programmierung 5
1.4 Einschränkung der Domäne 8
1.5 Ziele und Thesen dieser Arbeit 9
1.6 Struktur der Arbeit 10

2 Grundlagen 11

2.1 Computerlinguistik 11
2.1.1 Wortarten 12
2.1.2 Syntaxanalyse mit Syntaxbäumen 13
2.1.3 Syntaxanalyse mit Abgängigkeitsgraphen 16
2.2 Das Annotationswerkzeug GoldenGATE 18
2.3 Die linguistische Datenbank WordNet 20
2.4 Ontologien 22

3 Systematik und Überblick 27

3.1 Charakteristika der NLCI-Architektur und des bestehenden Prototyps . . 28
3.2 Anforderungen an die API 33
3.3 Empirische Grundlage 34
3.3.1 Aufbau des NLCI-Korpus 35
3.3.2 Verwendete Animationen 37
3.3.3 Ideal-Drehbuch für Animationen 38
3.3.4 Ablauf der Texterstellung 41
3.3.5 Aufbereitung der Texte 43
3.3.6 Zusammenfassung 45
3.4 Zusammenfassung 45

4 Verwandte Arbeiten 47
4.1 Programmieren natürlicher Sprache 47
4.2 Sprachverarbeitung in der Softwaretechnik 55
4.3 Verwandte Arbeiten zu den Textanalysen 58
4.3.1 Datenbankabfragen mit natürlicher Sprache 58
4.3.2 Linguistische Analyse von zeitlichen Abläufen in Texten 59
4.3.3 Aufbau von API-Modellen 62
4.4 Zusammenfassung 65

5 Die NLCI-Architektur im Detail 67
5.1 Umsetzung der NLCI-Architektur 67
5.2 Die Domänenontologie – das Herz der NLCI-Architektur 69
5.2.1 Die Struktur der NLCI-Ontologie 69
5.2.2 Befüllen der Ontologiestruktur 72
5.2.3 Vorverarbeitung und Aufbereitung der API-Bestandteile 72
5.2.4 Anreicherung der API mit Synonymen 75
5.3 NLP-Analysen 78
5.3.1 Verknüpfung von Text und API 78
5.3.1.1 Atomare Sätze 79
5.3.1.2 Verknüpfung atomarer Konstituenten mit der API . . . 81
5.3.1.3 Ermitteln von Methodenparametern 86
5.3.1.4 Auswahl der auszuführenden Methoden 90
5.3.2 Korrektur der Reihenfolge 91
5.3.2.1 Temporalausdrücke 92
5.3.2.2 Muster mit Temporalausdrücken 93
5.3.2.3 Mustersuche und Korrektur der Reihenfolge 94
5.3.3 Ermitteln von Kontrollstrukturen 98
5.3.3.1 Kontrollstruktur-unabhängige Verarbeitung des Abhängigkeitsgraphen 100
5.3.3.2 Betrachtete Kontrollstrukturen 101
5.3.4 Weitere domänenspezifische Analysen & Zusammenfassung . . . 103
5.4 Programmsynthese 104
5.5 Zusammenfassung 105

6 Fallstudien und Evaluation 107
6.1 Steuerung eines intelligenten Hauses mit openHAB 108
6.1.1 Aufbau der Ontologie 109

6.1.2 Auswertung 110
6.1.3 Bewertung der Ergebnisse 112
6.2 3D-Animationen mit Alice 113
6.2.1 Die Programmierumgebung Alice 114
6.2.2 Aufbau der Ontologie 117
6.2.2.1 Der Ontologie-Erzeuger 117
6.2.2.2 Bewertung der erzeugten Ontologie 120
6.2.3 API-Verknüpfung 121
6.2.3.1 Vorbereitung der Evaluation 121
6.2.3.2 Bewertung der Ergebnisse 123
6.2.4 Auswahl der auszuführenden Methoden 125
6.2.4.1 Vorbereitung der Evaluation 126
6.2.4.2 Bewertung der Ergebnisse mit vollständiger Ontologie . 126
6.2.4.3 Bewertung der Ergebnisse mit eingeschränkter Ontologie 129
6.2.5 Korrektur der Reihenfolge 129
6.2.5.1 Vorbereitung der Evaluation 129
6.2.5.2 Bewertung der Ergebnisse 130
6.2.6 Ermitteln von Kontrollstrukturen 132
6.2.6.1 Vorbereitung der Evaluation 132
6.2.6.2 Bewertung der Ergebnisse 134
6.2.7 Programmsynthese 136
6.3 Zusammenfassung 137

7 Fazit und Ausblick **139**
7.1 Die Thesen dieser Arbeit 141
7.2 Zukünftige Arbeiten 143

A Details zur openHAB-Fallstudie **147**
A.1 Konfiguration des Demo-Haushalts 147
A.2 Eingabetexte der Fallstudie 150

B Fragebogen **151**
B.1 Statistische Informationen 151
B.2 Basisaktionen 151
B.3 Aufgabenbeschreibung 151
B.4 Animationen 153

C Alice-Korpus 157
C.1 Animationen und Beispieltexte . . . 158
C.2 Selbsteinschätzung der Probanden . . . 170
C.3 Angaben zur Muttersprache der Probanden . . . 171

D Annotationen von NLCI 173
D.1 NLP-Vorverarbeitung . . . 173
D.2 Allgemeine Annotationen von NLCI . . . 174
D.3 Ergebnisse der Reihenfolgeanalyse . . . 176
D.4 Ergebnisse der Kontrollstrukturenanalyse . . . 176

E Syntaxregeln für die Bestimmung von Methodenparametern 179
E.1 Richtungsangaben . . . 179
E.2 Numerische Werte . . . 180

F Wortartmarkierungen nach PENN 181

G Typisierte Abhängigkeiten nach Stanford 183

Abbildungsverzeichnis IX

Tabellenverzeichnis XI

Literaturverzeichnis XIII

Kapitel 1

Die Vision der Programmierung in natürlicher Sprache

„The only way a person can truly concentrate on his problem and solve it without constraints imposed on him by the communication problem are if he is able to communicate directly with the computer without having to learn some specialized intermediate language.“

Jean E. Sammet, 1966

Programmierung von Computern ist ein Privileg – ein Privileg für wenige, gut ausgebildete Menschen. Sie verstehen es, eine Problemlösung so zu formulieren, dass eine Maschine sie umsetzen kann: Die Programmierung von Rechnern erfolgt mithilfe von synthetischen Programmiersprachen, deren Syntax und Semantik beherrscht werden muss. Diese Arbeit schlägt vor, Rechner in natürlicher Sprache zu programmieren, um die Programmierbarkeit – und damit die Kernfähigkeit von Rechnern – allgemein zugänglich zu machen. Anstatt Menschen darin auszubilden, sich rechnerkonform auszudrücken, sollten wir Rechnern beibringen, die natürliche Ausdrucksweise von Menschen zu verstehen. Die nachfolgenden Abschnitte begründen die obige Forderung und erläutern, warum das Unterfangen, dem Rechner Sprachkompetenz beizubringen, nicht nur erstrebenswert, sondern auch erreichbar ist.

1.1 Von der Rechenmaschine zum intelligenten Gesprächspartner

Die ersten Rechenmaschinen waren von Experten für Experten gebaut worden, um spezielle Rechenprobleme zu lösen. Heute sind Rechner allgegenwärtig und erschwinglich geworden; sie lösen nicht mehr nur einzelne mathematische Probleme, sondern sind durch ihre Programmierbarkeit flexibel einsetzbar geworden. Hersteller von Rechnern liefern heute die Hardware weitestgehend an Endbenutzer aus, ohne dass diese an eine konkrete Programmierung gebunden ist; der Endbenutzer installiert die Software, mit der er arbeiten möchte.

Dieser hohe Grad der Flexibilität, die fallenden Preise für Rechenleistung und Speicher und die Kommerzialisierung von Heimcomputern haben die Nutzerzahlen von Rechnern stark ansteigen lassen. Einer Schätzung der internationalen Telekommunikationsuntion (ITU) aus dem Jahre 2013 zufolge gibt es weltweit 2,7 Milliarden Internetnutzer [Int14] – jeder von ihnen hat unmittelbaren oder zumindest mittelbaren Zugriff auf einen Rechner. Nahezu alle Bereiche unseres Alltags sind von Computern beeinflusst und selbst „einfache“ Tätigkeiten werden heute von Computern unterstützt, gesteuert, überwacht oder Ähnliches. Kurz: Nahezu jeder muss mit Computern umgehen, mit ihnen arbeiten, vielleicht sie am besten programmieren können.

Die Politik und Interessenvertretungen der Informatik haben diesen Umstand erkannt und fordern, dass Programmierung oder zumindest eine Grundausbildung im IT-Bereich jedem angediehen werden sollte: Vinton Cerf stellte als Präsident der Association for Computing Machinery (ACM) 2013 fest: „Everyone would benefit from exposure to some form of programming.“ [Cer13] Im selben Jahr eröffnete US-Präsident Barack Obama die *Computer Science Education Week* mit einem Appell an junge Amerikaner, sich im Bereich der Informatik auszubilden und zu engagieren [Oba13]. Eng damit verbunden ist die *hour of code*, die einstündige Programmiereinführungen in über 40 Sprachen anbietet. Anfang 2016 wurde zudem das Programm „Computer Science For All“ angekündigt, das mit über 4 Mrd. US$ die IT-Ausbildung aller Schüler in der USA vorantreiben soll [Oba16]. Doch trotz dieser medienwirksamen Bemühungen ist es unwahrscheinlich, dass wir jeden zum Programmierer ausbilden können.

Daher fordert diese Arbeit, dass Rechner genügend Sprachkompetenz erhalten, sodass sie ihre Nutzer auch dann verstehen können, wenn diese keine Programmiersprache sondern eine natürliche Sprache sprechen. Diese Forderung ist nicht so weit hergeholt, wie es zunächst klingt: Die Betrachtung der verwandten Arbeiten und der computerlinguistischen Grundlagen dieser Arbeit zeigt, dass die Bausteine für derartige Systeme prinzipiell

bereitstehen; oftmals sind die Systeme aber nicht übertragbar oder auf bestimmte Anwendungsfälle zugeschnitten.

Mit unserer Forderung kommen wir den heutigen Benutzern entgehen: Sie sind bereit ihre Geräte per Sprache zu steuern – möglicherweise werden sie es sogar bald fordern: Immer mehr mobile Geräte können einfache Sprachbefehle entgegennehmen und beantworten die Anfragen ihrer Benutzer in gesprochener Sprache. Dabei funktionieren die Steuerungsprogramme wie Apples Siri [Bel14] immer besser und werden positiv von den Nutzern angenommen. Kurz- bis mittelfristig werden Computernutzer erwarten, alle Aspekte ihrer Geräte mit Sprache ansteuern zu können – und es ist nicht ersichtlich, warum eine derartige Steuerung auf das Mobiltelefon beschränkt sein sollte.

Unsere Forderung können wir aber mit Systemen wie Siri nicht nachkommen, denn sie sind eher darauf ausgelegt, aus kurzen Eingaben einen Befehl zu generieren und dann direkt rückzufragen bzw. das Ergebnis zu melden. Darüber hinaus ist ihre Verarbeitungslogik oft regelbasiert und handgeschrieben und erlaubt somit eine Steuerung nur in genau den Bereichen, die vom Hersteller vorgesehen und implementiert wurden. Eine allgemeine Programmierung in natürlicher Sprache wird diese vorgezeichneten Pfade aber regelmäßig verlassen: Nutzer werden nicht zwischen unterstützten und nicht unterstützten Anwendungsfällen unterscheiden wollen. Ebenso ist es wünschenswert, dass Benutzer neue Funktionalität beschreiben und verwenden können – unabhängig von der Herkunft der zugrundeliegenden Implementierung. Softwareingenieure werden daher ihre Programme mit größerer Sprachkompetenz und besseren sprachlichen Schnittstellen ausstatten müssen. Damit das effizient passieren kann, müssen unterstützende (und vereinheitlichende) Architekturen entworfen werden.

Die in dieser Arbeit vorgestellte Architektur für textuelle natürlichsprachliche Programmierung erfüllt einen Teil unserer Forderungen. Erschließt man die natürliche Sprache als Programmiersprache, so macht man einen Schritt hin zu universeller nutzbaren Rechnern. Der Benutzer wird weiter in den Mittelpunkt rücken – und zwar nicht deswegen weil Entwickler besonderen Wert auf die Bedürfnisse der späteren Benutzer legen, sondern weil er selbst ein Stück weit zum Entwickler wird.

1.2 Abstraktionsniveau und Zielgruppe

Bereits 1966 forderte Jean E. Sammet, von der auch das einleitende Zitat stammt, eine einfache Programmierbarkeit von Rechnern. Damalige Rechner „verstanden" mathematische Notationen und Programmiersprachen. In ihrer Vision beschreibt sie einen Rechner, der zusätzlich dazu natürliche Sprache verstehen und mit ihren „Schwächen" umgehen

kann, sodass er gemäß dem jeweiligen Einsatzzweck möglichst effizient programmiert werden kann.

Am Ende ihres Vortrags schätzt Sammet, dass die Vollendung dieser Vision – dieses Traumes – viele Jahre in Anspruch nehmen wird. Die Beschäftigung mit den verwandten Arbeiten zeigt, dass sie damit Recht behalten sollte: Viele Versuche sind nicht verallgemeinerbar oder scheiterten. In der Folge entstand keine kohärente Forschung zur Programmierung in natürlicher Sprache – jedoch ergaben sich insbesondere mit den stark gestiegenen Rechenkapazitäten viele Fortschritte bei der Verarbeitung natürlicher Sprache [JM09, Kapitel 1.6]. Diese Fortschritte machen wir uns zu Nutze, um eine Architektur zu gestalten, die die sprachliche Programmierung ermöglicht.

Die einleitende Motivation zeigt, auf welche Nutzergruppe und welches Abstraktionsniveau bzgl. der Programmierung wir abzielen: Laien bilden die Zielgruppe – Menschen ohne Programmierkenntnisse, die entweder Programmieren nicht lernen wollen oder können. Sprachliche Programmierung bedeutet in diesem Kontext also immer auch Programmierung durch den Endbenutzer.

Die Abstraktionsebene, auf der programmiert werden soll, ist höher als bei klassischer Programmierung: Natürlichsprachliche Programme werden typischerweise nicht hardwarenah sein, sondern in den domänenspezifischen Begriffen formuliert werden. Das angestrebte Computersystem muss dann diese Domänenbegriffe auf Programmstrukturen abbilden können. Das Erzeugen von Datenstrukturen und Ausführen primitiver Operationen auf diesen zum Erreichen eines übergeordneten Ziels ist nicht im Sinne der oben genannten Laien: sollten Sie sich auf einer niedrigen Abstraktionsebene mit der Programmierung beschäftigen müssen, so wird die ursprüngliche Intention ad absurdum geführt. Denn dann spielt es keine Rolle mehr, ob sie ihr Programm in einer natürlichen Sprache formulieren oder einer (synthetischen) Programmiersprache.

Angestrebt wird folglich nicht die Programmierung hochsensibler oder komplexer Systeme oder gar von Betriebssystemen. Für diese Art der Programmierung wird man nach wie vor klassische Programmiersprachen haben, da sie präzise und exakt testbar sind. In diesen Bereichen – und in allen sicherheitskritischen Bereichen – sind keine Laien tätig; den ohnehin gut ausgebildeten Experten möchten wir keine zusätzliche Programmierebene vorsetzen oder gar vorschreiben. Sie sollen weiterhin die bewährten Mittel zur Programmierung, zum Testen und zur Programmverifikation verwenden können. Sollte sich zeigen, dass in bestimmten Bereichen (z.B. das Formulieren von funktionalen Testfällen durch Mitarbeiter aus Fachabteilungen) eine natürlichsprachliche Programmierung sinnvoll und möglich ist, so ist der Einsatz im Bereich des Softwaretestens natürlich nicht ausgeschlossen.

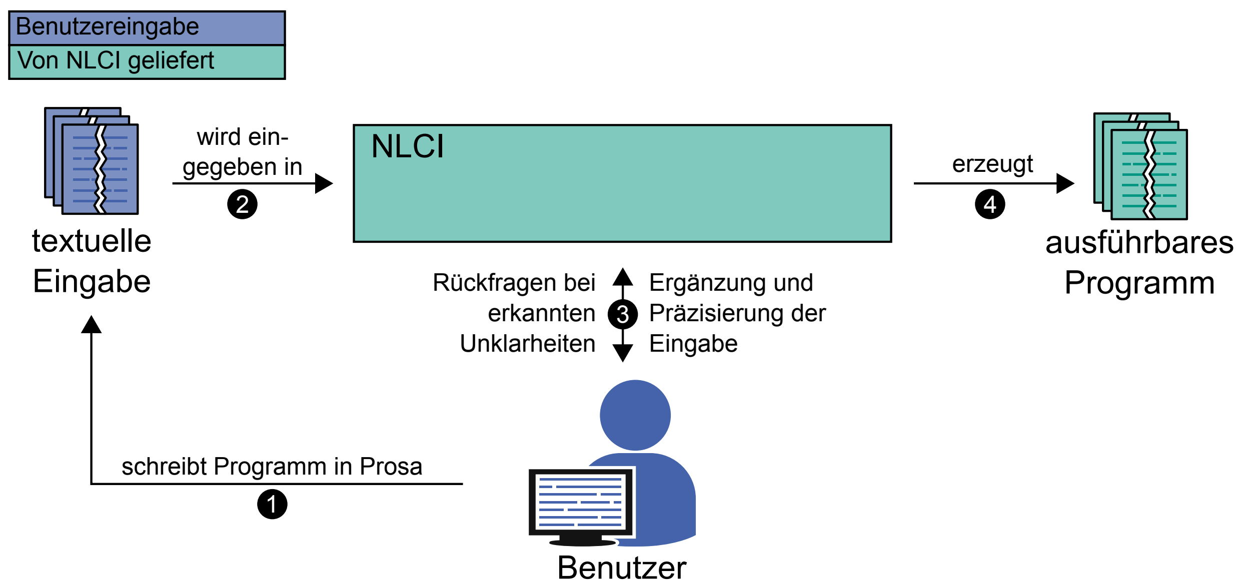


Abbildung 1.1: Die Verarbeitung soll für den Benutzer transparent geschehen: Er interagiert textuell in natürlicher Sprache mit der Maschine.

Über dieser Arbeit steht das Fernziel, einen Rechner zu erschaffen, der uns Menschen versteht. Diese Arbeit versteht das Ziel als Alternative zur derzeit angestrebten Ausbildung aller Menschen zu Programmierern. Anwendungen individuell mit ausreichend linguistischem Wissen (sprich: Eingabe- und Ausgabeschnittstellen, domänenspezifischen linguistischen Analysen und Rückfragemöglichkeiten usw.) auszustatten ist jedoch zu aufwendig – daher entwirft diese Arbeit eine wiederverwendbare flexible Architektur, die Anwendungen sprachlich erschließen soll.

1.3 Eine Architektur für natürlichsprachliche Programmierung

Ein Rechner, der mit natürlicher Sprache programmiert werden kann, muss mannigfaltige Aufgaben ausführen und dabei verschiedenste computerlinguistische Probleme bewältigen. Kern und Namensgeber der hier vorgestellten Architektur ist der Natural Language Command Interpreter, kurz NLCI. Er ist dafür zuständig, die vom Benutzer ausgedrückten Anweisungen auf vom darunter liegenden Rechner interpretierbare Befehle abzubilden. Technisch wird hierbei das Eingabedokument von verschiedenen Analysemodulen aufbereitet und gewonnene Informationen (z.B. welche API-Methoden aufzurufen sind und in welcher Reihenfolge) als Annotation im Dokument hinterlegt.

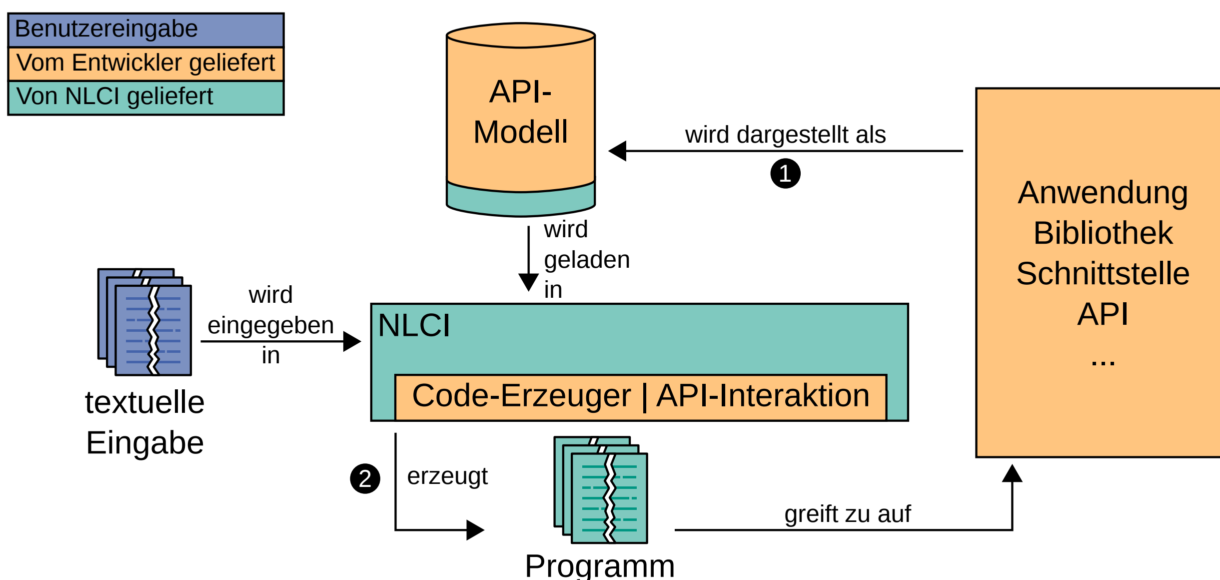


Abbildung 1.2: Vieles ist in der Architektur vorgegeben. Entwickler steuern neben der API nur die API-Beschreibung und ggf. den Quelltext-Erzeuger bei.

Für den Benutzer soll diese Maschinerie möglichst transparent sein – er interagiert textuell in natürlicher Sprache mit der Maschine: Er formuliert sein Programm in natürlicher Sprache (1,2), wird in natürlicher Sprache zum Ergänzen und Korrigieren aufgefordert (3) und erhält am Ende das ausführbare Programm bzw. dessen Ergebnis (4). Abbildung 1.1 verdeutlicht diesen Entwurf. Dem Entwickler soll ein Rahmen gegeben werden, mit der sich Anwendungen einfach sprachlich erschließen lassen soll: NLCI ist eine Architektur, mithilfe derer er Anwendungen um Sprachkompetenz erweitert kann, ohne dass er selbst Kenntnisse im Bereich der Sprachverarbeitung haben muss.

Wie Abbildung 1.2 verdeutlicht, muss der Entwickler neben seiner API zusätzlich deren Beschreibung in Form eines Modells bereitstellen; die Struktur des Modells wird dabei von NLCI vorgegeben. Die API muss im Modell sprechend benannt sein und sich auf der Abstraktionsebene des Benutzers befinden. Die Erstellung des Modells (in Abbildung 1.2 mit (1) gekennzeichnet) kann manuell erfolgen; bei großen APIs ist jedoch eine Automatisierung hilfreich. In den Fallstudien wird gezeigt, dass sowohl ein Modell-Generator als auch der manuelle Aufbau verhältnismäßig einfach möglich sind. Zusätzlich zum API-Modell muss NLCI mit einer Komponente zur Quelltext-Erzeugung ausgestattet werden (in Abbildung 1.2 mit einer (2) gekennzeichnet). Sofern die anzusteuernde API in einer unterstützten Programmiersprache vorliegt, ist hier nichts weiter zu tun. Existiert keine vorbereitete Anbindung, so muss der Entwickler eine Komponente beisteuern, die Quelltext in der gewünschten Programmiersprache erzeugen kann. Hierbei

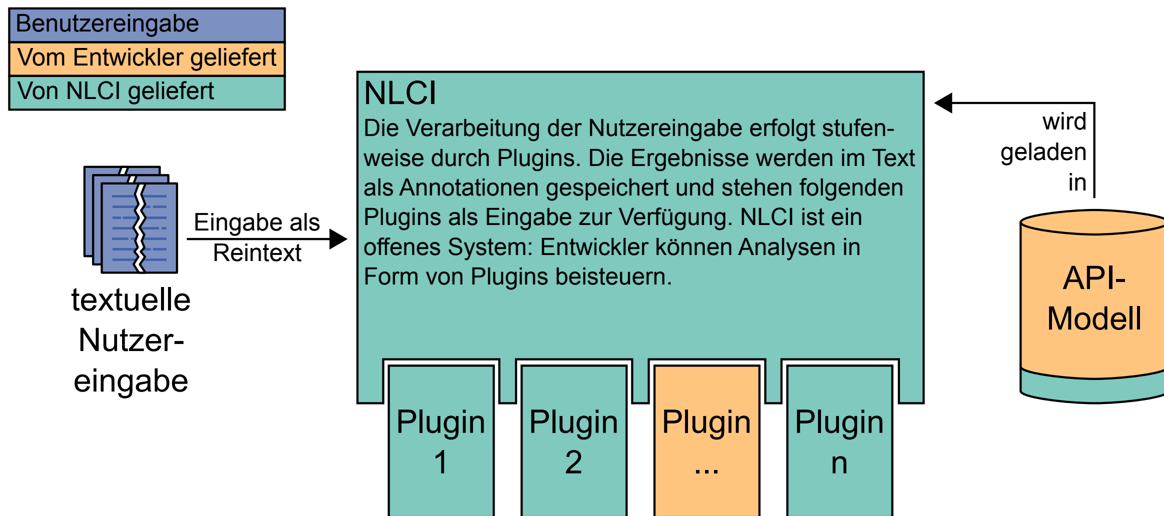


Abbildung 1.3: Der interne Aufbau der NLCI-Architektur. Zusätzliche Analysen können in die Verarbeitungskette integriert werden, indem Entwickler einen Einschub bereitstellen; NLCI bietet dann Zugriff auf den Eingabetext und die vorhandenen Analyseergebnisse. Die Struktur des API-Modells wird von NLCI vorgegeben, Entwickler müssen nur den Inhalt liefern.

kann die Komponente auf die Informationen der Analysestufen zugreifen, ohne dass die darunterliegenden Analysen selbst bekannt bzw. verstanden sein müssen.

Die vorgestellte Architektur zeichnet sich dadurch aus, dass NLCI als *Black Box* betrachtet werden kann. Die einzigen Schnittstellen zur Ziel-API ist das API-Modell und die Quelltext-Erzeugung. Letztere muss auf die gewünschte Programmiersprache zugeschnitten werden, ist jedoch unabhängig von der eigentlichen Ziel-API. Das Wissen, welche Funktionalität in der Ziel-API verfügbar ist, wird dem System in Form des API-Modells eingespeist. An dieser Stelle reicht es aus, sich dieses Modell als Telefonbuch für die Ziel-API vorzustellen. Die dahinter liegende Idee ist der Wunsch, die Ziel-Plattform, also die Domäne, austauschbar zu machen ohne die Black Box anpassen zu müssen; das erlaubt letztlich die Anpassung von NLCI auf andere Domänen. NLCI gibt hier die Struktur des Modells vor; so muss ein Entwickler lediglich beschreiben, welche Elemente seine API bereitstellt und wie diese angesprochen werden können.

Betrachtet man die Interna unserer Black Box in Abbildung 1.3, so sieht man, dass es sich um eine Fließband-Architektur handelt: Die Analysemodule schreiben ihre Ergeb-

nisse direkt als Annotation in den Eingabetext. So werden Analyseergebnisse unmittelbar für die folgenden Module sichtbar. Zudem haben alle Analysestufen Zugriff auf das API-Modell.

Die verschiedenen Analysen laufen nacheinander ab und können aufeinander aufbauen (z.B. kann eine Stufe die Annotationen eines Vorgängers weiterverarbeiten). Die derzeit verfügbaren Analysestufen erlauben die Übersetzung von „imperativen natürlichsprachlichen Programmen“. Eine Verbesserung der einzelnen Analysestufen oder eine gänzlich neue Analyse kann nun in völliger Isolation entwickelt, getestet und letztendlich in NLCI integriert werden – ohne Kenntnis oder Berücksichtigung der übrigen Analysestufen. Diese Entkopplung der Sprachverarbeitungskomponenten ist ein weiterer Vorteil der vorgestellten Architektur.

1.4 Einschränkung der Domäne

Frühere Ansätze des Programmierens in natürlicher Sprache beschränkten sich auf bestimmte Domänen und ermöglichten so bereits beim Entwurf des Systems die notwendigen Prüfungen zu hinterlegen. So entwarfen Lieberman und Liu ein System, das Programmrümpfe in Python erzeugen kann [LL06] (siehe dazu auch den Abschnitt 4.1 im Kapitel über verwandte Arbeiten).

Die Kernidee des hier vorgestellten Ansatzes ist eine Wissensdatenbank, in der die Programmierkonzepte des Zielsystems modelliert sind. NLCI ist jedoch so entworfen, dass die Wissensdatenbank ausgetauscht werden kann, ohne dass die Sprachanalysekomponenten angepasst werden müssen. Lediglich der Quelltext-Erzeuger muss auf das neue Zielsystem angepasst werden – ähnlich wie ein Frontend eines Übersetzers – wenn man die Programmiersprache wechselt; behält man die Programmiersprache beim Domänenwechsel, so kann der Quelltext-Erzeuger weiterverwendet werden.

Die eingangs erwähnte Einschränkung eines Eingabesystems auf eine Domäne ist kurzfristig ein notwendiges Übel. Die Idealvorstellung, mit seinem Rechner einen verständigen Gesprächspartner zu haben, ist zu weit hergeholt. Noch weiter weg von der Realität ist ein Rechner, der nicht nur versteht, sondern auch reflektiert, zurück- und zwischenfragt und sich so selbst ein Bild von der gestellten Aufgabe macht. Ein derartiger Rechner müsste nicht nur verstehen, was zu tun ist – er müsste zudem auch die Pragmatik der Aufgabe verstehen und automatisch eine Zielvorstellung entwickeln, deren Erreichung er selbstständig prüfen kann. Unser System versucht Rückfragen zu stellen, wenn Teile unklar sind; jedoch können nur Rückfragen gestellt werden, wenn bestimmte, erwartete Informationen fehlen. Welche Informationen das sind – und wie man ihre Abwesenheit

feststellen kann – ist damit bereits zur Implementierungszeit von NLCI festgelegt und unabhängig vom späteren Anwendungsfall.

1.5 Ziele und Thesen dieser Arbeit

Diese Arbeit verfolgt das Ziel, Programmieren in natürlicher Sprache zu ermöglichen. Da es derzeit nicht möglich scheint, eine vollständig domänenunabhängige Lösung zu formulieren, soll der Wechsel in eine neue, bisher unbekannte Domäne möglichst einfach gestaltet werden. Dazu wird eine Architektur entworfen und daran die folgenden Thesen untersucht:

1. Die vorgestellte Architektur kann genutzt werden, um natürlichsprachlich ausgedrückte, imperative Programme in Quelltext zu übersetzen. Endanwender ohne Programmierkenntnisse bilden die Zielgruppe, weswegen der Fokus auf skriptartige Ablaufbeschreibungen und nicht auf komplexe Algorithmen gelegt wird.

2. Es ist dafür nicht notwendig, die zulässige Eingabesprache stark einzuschränken; die Ergebnisse heutiger NLP-Analysen sind als Grundlage gut genug.

3. Das in der Architektur vorgesehene API-Modell für das Domänenwissen kann ausgetauscht und so die Domäne gewechselt werden. Das API-Modell kann verhältnismäßig einfach erzeugt werden (bei geeigneten Quellen sogar automatisch).

4. Nicht-Sequenzialität der Beschreibung schadet nicht: Die Reihenfolge der Beschreibung spielt eine untergeordnete Rolle, solange die zeitlichen Beziehungen zwischen den Aktionen genannt werden. Dann kann die chronologisch korrekte Reihenfolge der Aktionen rekonstruiert werden.

5. Kontrollstrukturen (bspw. einfache Wiederholungen oder Parallelität) können aus natürlichsprachlichen Beschreibungen synthetisiert werden.

Um diese Thesen zu untersuchen wurde die NLCI-Architektur prototypisch implementiert. Der Prototyp ist verwendbar für englische Eingaben in Textform und wurde für zwei unterschiedliche Ziel-APIs umgesetzt:

- Steuerung eines *smart homes* mit openHAB: openHAB ist ein hersteller- und plattformunabhängiges System zur Steuerung von intelligenten Häusern [ope]. openHAB kennt hierfür Sensoren (die bspw. die Frage beantworten können, ob eine Tür

geöffnet ist) und Aktoren (die bspw. einen Rolladen öffen oder schließen). Die Integration von NLCI und openHAB erlaubt nun die Ansteuerung dieser Aktoren mit natürlicher Sprache.

- 3D-Animationen mit Alice: Alice ist eine Plattform zur Erstellung von 3D-Animationen [Con97, CAB^{+}00]. Es umfasst neben der reinen Animationserzeugung eine Entwicklungsumgebung für Programmierer, die darin die verwendeten 3D-Objekte mit Java-artigem Quelltext steuern können. Grundsätzlich eignet sich Alice nicht nur für die Erstellung „statischer" Animationen, sondern sogar für interaktive Sequenzen und erlaubt so auch die Erzeugung von Spielen. Dieser interaktive Teil wurde bei der Umsetzung für NLCI jedoch nicht betrachtet.

Die Fallstudien in Kapitel 6 zeigen, wie die beiden Ziel-APIs für NLCI aufbereitet wurden und welche Möglichkeiten der Programmierung sich dadurch ergeben. Zudem wird evaluiert, wie gut die Übersetzung einer natürlichsprachlichen Eingabe in die jeweilige tatsächliche Programmierung funktioniert.

1.6 Struktur der Arbeit

Zunächst werden in den folgenden Kapiteln die technischen sowie computerlinguistischen Grundlagen und verwandte Arbeiten zusammengefasst und diskutiert. Das vierte Kapitel beschreibt die Architektur, die hinter NLCI steht, einmal aus Sicht des Anwenders und einmal aus Sicht des NLCI-Architekten. Das darauf folgende Kapitel beschreibt die Bestandteile (d.h. Analysekomponenten und Datenstrukturen) der Architektur genauer und geht auf die Entwurfsentscheidungen ein, die während der Umsetzung von NLCI getroffen wurden. Im sechsten Kapitel werden Fallstudien vorgestellt, die aufzeigen wie und wie erfolgreich der Prototyp von NLCI eingesetzt werden kann. Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen. Es greift die obigen Thesen noch einmal auf und beurteilt, welche durch die vorliegende Arbeit gestützt werden.

Kapitel 2

Grundlagen

„Programming in natural language might seem impossible, because it would appear to require complete natural language understanding and dealing with the vagueness of human descriptions of programs. But we think that several developments might now make programming in natural language feasible.“

Henry Lieberman & Hugo Liu, 2006

Dieses Kapitel behandelt die für die vorliegende Arbeit nötigen Grundlagen. Zunächst werden Konzepte der Computerlinguistik vorgestellt, sowie die benötigten linguistischen Werkzeuge und deren Ergebnisse erläutert. Danach beschäftigt sich das Kapitel mit (linguistischen) Ressourcen: Zum einen wird WordNet, ein maschinenlesbares Wörterbuch, zum anderen Ontologien vorgestellt.

2.1 Computerlinguistik

Die Verarbeitung natürlicher Sprache erfolgt meist mit einem Fließband, bei dem in jeder Stufe eine bestimmte Analyse durchgeführt wird. Üblicherweise beginnt das Fließband mit dem Einlesen des Textes und der Tokenisierung. Danach finden die eigentlichen linguistischen Analysen wie Wortartmarkierung und das Zerteilen der Sätze statt.

Alle Analysen, die in dieser Arbeit vorgestellt werden, verwenden Ergebnisse computerlinguistischer Analysen. Im Rahmen von NLCI wurden keine grundlegenden computerlinguistischen Werkzeuge oder Analysen entwickelt, sondern NLCI verwendet die bestehenden Werkzeuge und nutzt sie zur Vorverarbeitung der Eingabe. Die nachfolgend vorgestellten linguistischen Begriffe und Werkzeuge werden daher in dieser Arbeit immer wieder verwendet. Weniger wichtige Details werden bei Bedarf an Ort und Stelle

Tabelle 2.1: Ein Auszug aus dem Markierungssatz der PENN-Tree-Bank [San95].

Markierung	Bedeutung
NN	Nomen (Singular) oder Stoffname
PRP$	Possesivpronomen
RB	Adverb
VBG	Verb, Gerundiv-Form oder Partizip-Präsens
VBZ	Verb, dritte Person Singular, Präsens
.	Satzzeichen (Punkt)

eingeführt und kurz erklärt. Für eine ausführliche Einführung in die Computerlinguistik sei auf das Lehrbuch von Jurafsky und Martin [JM09] verwiesen.

2.1.1 Wortarten

Die erste Analyse nach dem Tokenisieren des Eingabetexts ist in vielen NLP-Fließbändern das Markieren der Wortarten. Jedes Token (normalerweise jedes Wort, alle Satzzeichen und je nach Anwendungszweck auch Zeilenumbrüche) erhält dabei eine Markierung, welche die Art des Tokens angibt. So werden bspw. Verbformen (z.B. Verb, dritte Person Singular, Präsens) angegeben und zwischen Nomen in Singular und Plural unterschieden.

Zum Auszeichnen der verschiedenen Wortarten existieren verschiedene Markierungssätze (engl. *tag sets*), die auf einen konkreten Anwendungsfall zugeschnitten sein können. Weit verbreitet ist der Markierungssatz der PENN-Tree-Bank, der 36 verschiedene Markierungen umfasst [San95]. Wortarten werden oft direkt hinter den Wörtern tiefgestellt oder abgetrennt mit einem Schrägstrich angegeben. Häufig werden die Wortarten auch unter den Wörtern notiert, z.B. wenn sie in der Abbildung eines Syntaxbaums angegeben werden. Eine vollständige Liste der Markierungen der PENN-Tree-Bank ist in Anhang F aufgeführt, Tabelle 2.1 zeigt einen Auszug daraus.

Beispiel

Gegeben sei der Satz „My dog also likes eating sausage.“, der wie folgt mit Wortarten annotiert wird:

My/PRP$ dog/NN also/RB likes/VBZ eating/VBG sausage/NN ./.

> Hier ist gekennzeichnet, dass es sich bei „dog“ und „sausage“ um Nomen (Markierung NN) handelt, bei „my“ um ein Possesivpronomen (Markierung PRP$) und so weiter. In Tabelle 2.1 finden sich die weiteren Markierungen mit ihrer Bedeutung.

Die automatische Markierung von Wortarten wird mit einem Wortartmarkierer (engl. *Part-of-speech tagger* oder *POS tagger*) vorgenommen. Zum Markieren können Wortlisten und Regelsätze verwendet werden. Aktuelle Markierer verwenden jedoch statistische Verfahren, die auf einem (bereits markierten) Korpus trainiert werden. Beim Training werden Wahrscheinlichkeiten für Markierungsfolgen bei gegebenen Wortfolgen gelernt; je nach statistischem Modell können hierbei auch verschiedene Kontextinformationen betrachtet werden (z.B. die Markierung der umgebenden Wörter, die Wörter selbst oder morphologische Informationen). Sehr oft kommt beim Training das Wall-Street-Journal-Korpus (WSJ-Korpus) zum Einsatz, welches in der PENN-Tree-Bank mit Syntaxbäumen und mit Wortarten versehen ist [MSM93].

In der Vorverarbeitungsstufe von NLCI kommt der statistische Wortartmarkierer der NLP-Gruppe aus Stanford zum Einsatz [TM00, TKMS03], der Teil der Bibliothek CoreNLP ist [MSB+14]. Er kann auf einem Korpus trainiert und mit verschiedenen statistischen Modellen konfiguriert werden. Im Rahmen von NLCI findet kein Training des Markierers statt, sondern es wird ein Standardmodell für Englisch verwendet, das auf dem WSJ-Korpus trainiert ist. Die Wahl fiel auf die Werkzeuge aus Stanford, da sie in der Literatur als sehr leistungsfähig beschrieben werden. Eine nachträgliche Untersuchung der Genauigkeit von neun Wortartmarkierern auf dem NLCI-Korpus zeigte, dass der Wortartmarkierer aus Stanford mit einer Genauigkeit von $96,2\%$ am besten abschneidet; der zweitbeste Markierer ist der aus Berkeley mit einer Genauigkeit von $95,9\%$ [Kie16].

Aus den Wortarten alleine lässt sich nur schwer ableiten, welche grammatikalische Struktur ein Satz besitzt. Eine Darstellung von Sätzen, die deren Aufbau zeigt, sind Syntax- oder Konstituentenbäume.

2.1.2 Syntaxanalyse mit Syntaxbäumen

Syntaxbäume sind geordnete Bäume, deren Blätter die Token eines Satzes sind. Die Wurzel des Baums repräsentiert den Satz und die inneren Knoten zeigen seinen strukturellen Aufbau. Syntaxbäume entstehen, wenn man Sätze mit einer Grammatik zerteilt und die Ableitungen zur Erstellung der Knoten und Kanten heranzieht. Eine Grammatik (S,N,Σ,R) besteht aus einem Startsymbol S, einer Menge N an Nicht-Terminalen, einem Lexikon Σ mit Terminalsymbolen sowie einer Menge R an Ableitungsregeln.

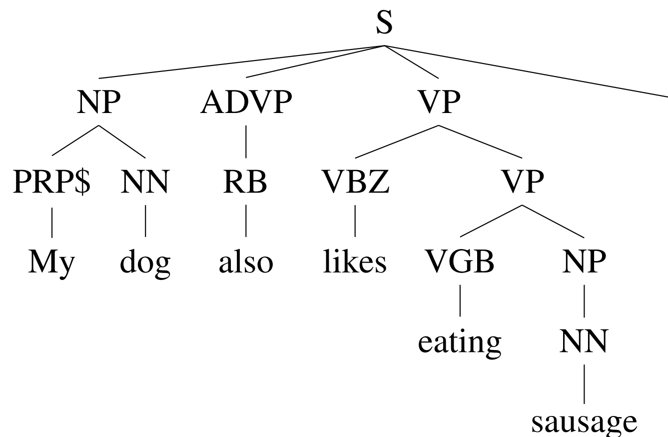


Abbildung 2.1: Der Syntaxbaum des Satzes „My dog also likes eating sausage."

Das Startsymbol repräsentiert beim Zerteilen von natürlichsprachlichen Texten einen Satz. Die Nicht-Terminalsymbole entsprechen den Konstituenten der Sätze und die Terminalsymbole sind die Wörter und Satzzeichen. Konstituenten werden im Baum von unten nach oben gesehen immer weiter zusammengefasst, bilden so Phrasen und schließlich den ganzen Satz. Die Phrasen sind nach ihrem jeweiligen Kopf benannt. Zum Beispiel bezeichnet man eine Phrase, deren Kopf ein Nomen ist, als Nominalphrase. Eine Verbalphrase enthält folglich mindestens ein Verb, sie kann jedoch durch Ergänzungen wie Hilfsverben, Objekte und Modifikatoren erweitert sein.

NLCI verwendet die Definition des Phrasenkopfes von Miller: „The basic criteria for recognizing the head of a syntactic construction are that the head is obligatory and controls certain other possible constituents (words) called modifiers." [Mil11] Mit anderen Worten: Der Kopf einer Phrase ist das Wort der Phrase, das sie *kontrolliert*; d.h. der Kopf bestimmt die Bedeutung der Phrase und die anderen Wörter der Phrase modifizieren ihn lediglich oder beschreiben ihn genauer [Cho95, Mil11]. Ohne den Kopf ergibt die Phrase in der Regel keinen Sinn – oder einen völlig anderen. Die Beispielphrase „the old man" hat den Kopf „man" und „the" und „old" sind seine Modifikatoren. Den Kopf kennzeichnet man mit einem vorangestellten Zirkumflex oder Dach (ˆ) und schreibt „the old ˆman".

Phrasenstrukturgrammatiken sind kontextfreie Grammatiken; sie enthalten Ableitungsregeln der Form $\alpha \rightarrow \beta, \alpha \in N, \beta \in \{N \cup \Sigma\}^+$, d.h. dass auf der linken Seite einer Regel immer nur ein Nicht-Terminalsymbol steht, auf der rechten Seite beliebig viele Symbole. Bei Phrasenstrukturgrammatiken unterscheidet man zudem zwischen lexikalischen Regeln der Form $\alpha \rightarrow \beta, \alpha \in N, \beta \in \Sigma$ und Phrasenstrukturregeln der Form $\alpha \rightarrow \beta, \alpha \in N, \beta \in NN^+$.

Beispiel

Abbildung 2.1 zeigt den Syntaxbaum des Satzes „My dog also likes eating sausage.“

Es handelt sich um einen Aussagesatz, der aus drei Phrasen besteht: Einer Nominalphrase, einer Adverbialphrase und einer Verbalphrase; das Satzzeichen ist das vierte Kind des Wurzelknotens. Die Nominalphrase ist das Subjekt des Satzes und besteht aus dem Possessivpronomen „my“ sowie dem Nomen „dog“.

Die Verbalphrase ist komplizierter, da sie das Verb „likes“ mit seinem Objekt verbindet. Dieses Objekt ist selbst wiederum eine Verbalphrase, die weiter aufgespalten wird.

Die jeweils unterste Ableitung vor den Blättern entspricht der Wortartmarkierung, wie sie im vorangegangenen Abschnitt eingeführt wurde.

Ein Zerteiler (engl. *parser*) erzeugt mit einer Grammatik für eingegebene Sätze die zugehörigen Syntaxbäume. Dazu werden die Ableitungsregeln wiederholt angewendet, um eine Folge von Ableitungen zu erstellen, mit der sich aus einem Startsymbol der Eingabesatz erzeugen lässt. Beim Entwurf einer Grammatik muss gegeneinander abgewogen werden, dass kleine Grammatiken nur über einen stark eingeschränkten Sprachumfang verfügen und große, flexible Grammatiken nicht mehr eindeutig sind.

Beim Zerteilen mit einer mehrdeutigen Grammatik kann es dazu kommen, dass aus einem Satz verschiedene Syntaxbäume abgeleitet werden können. Diese Bäume zeigen dann unterschiedliche Phrasenstrukturen und stehen für unterschiedliche semantische Bedeutungen. Da diese Bäume mit der Grammatik erzeugt wurden, sind sie gültige Sätze der Sprache. Ein Zerteiler muss in der Regel entscheiden, welche Ableitung verwendet und welcher Syntaxbaum zurückgegeben werden soll. Um dieses Problem zu lösen, müsste der Zerteiler über Kontextwissen verfügen. Da Kontextwissen dem Zerteiler nicht zur Verfügung steht und kein Diskursmodell während des Zerteilens eines Textes erzeugt wird, behilft man sich mit probabilistischen Grammatiken.

Bei probabilistischen Grammatiken (S, N, Σ, R, P) wird zusätzlich eine Wahrscheinlichkeitsverteilung P angegeben, die jeder Ableitungsregel eine Wahrscheinlichkeit zuordnet. Die Wahrscheinlichkeitsverteilung wird mithilfe eines Trainingskorpus bestimmt.

Die Gesamtwahrscheinlichkeit eines Baumes erhält man, wenn man das Produkt aller Wahrscheinlichkeiten der angewendeten Regeln bestimmt. Meistens wird nur der wahrscheinlichste Baum ausgegeben; es gibt jedoch auch Zerteiler, die eine TOP-N an Bäumen mit den zugehörigen Konfidenzen liefern.

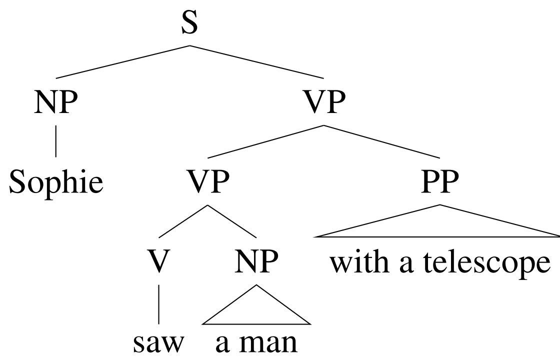


Abbildung 2.2: Ein Syntaxbaum für den Satz „Sophie saw a man with a telescope.“

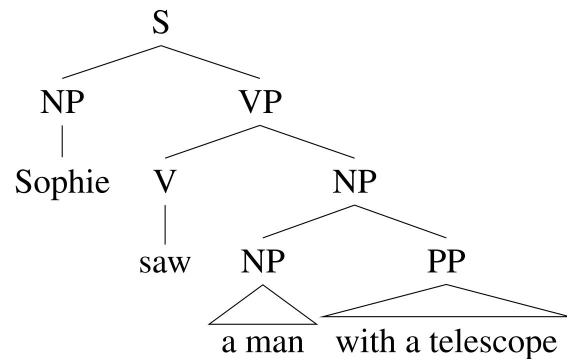


Abbildung 2.3: Ein Syntaxbaum für den Satz „Sophie saw a man with a telescope.“

Beispiel

Die Abbildungen 2.2 und 2.3 zeigen zwei Syntaxbäume für den Satz „Sophie saw a man with a telescope.“ Man kann sehen, dass sich der Satz aus zwei Hauptbestandteilen zusammensetzt: Es handelt sich um einen Aussagesatz, bestehend aus einer Nominalphrase (NP) und einer Verbalphrase (VP). Die Verbalphrase wird noch weiter aufgespalten, bevor man zu den Wörtern des Satzes gelangt.

Ist nur dieser Satz ohne Kontext gegeben, so ist unklar, wer das Teleskop besitzt: Sophie oder der Mann. Die Syntaxbäume zeigen diese beide Interpretationen mit dem unterschiedlichen Aufbau der Verbalphrase: Der Syntaxbaum in Abbildung 2.2 drückt aus, dass das Sehen mit dem Fernrohr geschieht. „saw a man“ und „with a telescope“ sind in einer Verbalphrase (VP) zusammengefasst. Der Baum in Abbildung 2.3 zeigt hingegen, dass „a man“ und „with a telescope“ zu einer Nominalphrase (NP) zusammengefasst wurden und diese Nominalphrase zusammen mit „saw“ die Verbalphrase bildet; in dieser Interpretation hat folglich der Mann ein Fernrohr.

Bei NLCI kommt der probabilistische Zerteiler der CoreNLP-Bibliothek aus Stanford zum Einsatz [SBMN13]; ebenso wie der Wortartmarkierer wurde er nicht gesondert trainiert, sondern es wird die Standardkonfiguration verwendet.

2.1.3 Syntaxanalyse mit Abgängigkeitsgraphen

Die unterschiedlichen Interpretationen des Beispielsatzes aus dem vorigen Abschnitt lassen sich in den Syntaxbäumen bereits erkennen. Abhängigkeitsgraphen ermöglichen jedoch einen einfacheren Zugriff auf diese Information.

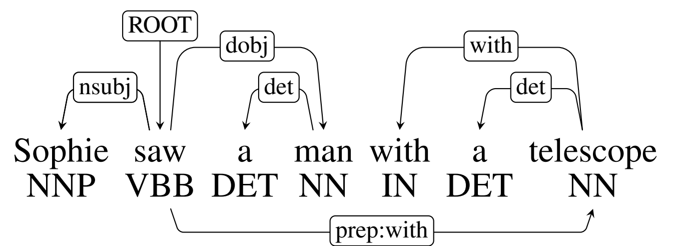


Abbildung 2.4: Ein Abhängigkeitsgraph für den Satz „Sophie saw a man with the telescope“. Die Wortarten sind unterhalb der Wörter angegeben. Der Abhängigkeitsgraph zeigt die Variante aus Abbildung 2.2.

Ein Abhängigkeitsgraph stellt typisierte Abhängigkeiten (engl. *typed dependencies*) zwischen den Wörtern im Satz dar. Die Beziehungen zwischen den Wörtern lassen sich mithilfe von Regeln aus Syntaxbäumen ableiten und sind gemäß der NLP-Gruppe aus Stanford die zu bevorzugende Repräsentation, wenn man die Bedeutung eines Satzes untersuchen möchte [dM08].

Eine Abhängigkeit `reln(gov, dep)` hat immer einem Typ (die *relation* `reln`) und verbindet einen *governor* `gov` mit einem *dependent* `dep`. Der Abhängigkeitsgraph entsteht, wenn man governor und dependent als Knoten auffasst und die Abhängigkeiten als gerichtete Kante mit dem Typ `reln` vom governor zum dependent im Graph zieht. Da ein Wort gleichzeitig in verschiedenen Abhängigkeiten der dependent bzw. governor sein kann, ergibt sich im Ergebnis nicht wie beim Zerteilen ein Baum, sondern ein Graph. Der Knoten, der nur ausgehende aber keine eingehenden Kanten besitzt – meist das Hauptverb des ersten Hauptsatzes – wird als Wurzel (engl. *root*) des Graphen bezeichnet.

Die Abhängigkeitsgraphen werden in drei unterschiedlichen Varianten angeboten: normal, gefaltet (engl. *collapsed*) sowie gefaltet mit aufgelösten Konjunktionen (engl. *collapsed CC-processed*). Beim Falten werden Präpositionen aufgelöst und die beteiligten Knoten direkt verknüpft. Das Auflösen von Konjunktionen hat bspw. zur Folge, dass bei einer Ellipse das Verb direkt mit den beiden Objekten verbunden wird anstellte mit dem ersten Objekt und der Konjunktion.

Abhängigkeitsgraphen abstrahieren von der syntaktischen Struktur eines Satzes. Betrachtet man zum Beispiel einen Satz im Aktiv „Sophie saw a man.“, dann steht das Subjekt im Syntaxbaum ganz links. Formuliert man denselben Satz im Passiv („A man was seen by Sophie.“), tauschen Subjekt und Objekt im Syntaxbaum die Seiten. Im Abhängigkeitsgraph ändert sich an der Struktur dabei nur sehr wenig.

Beispiel

In Abbildung 2.4 ist der Abhängigkeitsgraph zum Satz „Sophie saw a man with a telescope." abgebildet. Die Wurzel des Satzes „saw" ist mit „ROOT" gekennzeichnet.

Die beiden Kanten `nsubj` und `dobj` ausgehend von „saw" kennzeichnen das Subjekt bzw. das direkte Objekt des Verbs „saw". Das „telescope" ist mit einer `prep:with`-Kante mit dem Verb verbunden; sie drückt aus, dass das „telescope" in einer Präpositionalphrase mit „with" die Bedeutung des Verbs beeinflusst. Der Abhängigkeitsgraph zeigt folglich die Variante aus Abbildung 2.2: Sophie besitzt das Teleskop.

Es handelt sich um die gefaltete Darstellung, was dazu führt, dass die Präposition „with" nicht mit dem Verb verbunden ist, sondern die `prep:with`-Kante direkt zum „telescope" zeigt. In der ungefalteten Fassung würde diese Kante als zwei Kanten dargestellt werden: `prep(saw, with)` und `pobj(with, telescope)`.

Die beiden unbestimmten Artikel „a" sind mit einer `det`-Kante mit den dazugehörigen Substantiven verbunden. Für die Analyse des Satzes sind die Artikel jedoch von untergeordneter Bedeutung. Daher zeigt bspw. die `dobj`-Kante direkt auf „man". Im Syntaxbaum bildet „man" zusammen mit dem Artikel eine Nominalphrase, die bei der Satzanalyse genauer betrachtet werden müsste; im Abhängigkeitsgraph gelangt man direkt vom Verb zum Objekt.

Typisierte Abhängigkeiten gibt es für viele syntaktische Konstruktionen und mit ihrer Hilfe lassen sich viele semantische Zusammenhänge erschließen. Eine Liste der Abhängigkeiten nebst Erläuterungen ist in Anhang G zu finden. Im technischen Bericht [dM15] der Stanford'schen Computerlinguisten findet sich eine ausführliche Erklärung der einzelnen Relationen mit vielen Beispielen.

Die Abhängigkeiten werden vom selben Werkzeug erzeugt, wie die Syntaxbäume, die im vorigen Abschnitt beschrieben wurden: dem Zerteiler aus dem CoreNLP-Paket [MSB+14].

2.2 Das Annotationswerkzeug GoldenGATE

GoldenGATE ist eine Rahmenarchitektur für das linguistische Annotieren von Texten. Es bietet einerseits einen XML-Editor, der Annotationen als XML-Markup darstellt, und andererseits vielfältige Möglichkeiten zum manuellen, teil- und voll-automatischen Annotieren von Texten. GoldenGATE wurde – inspiriert von der bekannten NLP-Rahmenarchitektur GATE – entwickelt, um das manuelle Annotieren, die Nachbearbeitung und Kontrolle von Verarbeitungsergebnissen zu vereinfachen [SBA07].

GoldenGATE zeichnet sich vor allem dadurch aus, dass es den annotierten Text in einem Editorfenster als XML-Dokument darstellt, im Hintergrund jedoch ein Token-basiertes Datenmodell verwendet. Der XML-Markup kann je nach Bedarf eingeblendet oder für einzelne oder alle Annotationstypen ausgeblendet werden. Durch das Datenmodell wird auch sichergestellt, dass auch bei manuell eingefügten Annotationen immer ganze Token annotiert werden, unabhängig davon ob das gesamte Token markiert wurde, oder nur ein Teil. Zudem unterstützt das Modell Annotationen, die nicht XML-konform sind, weil sie sich überlappen.

Beispiel

Gegeben sei die Folge von Token $t_0,\ldots,t_3$. GoldenGATE erlaubt es, $t_0,\ldots,t_2$ mit einer Annotation a_0 zu versehen und die Folge $t_1,\ldots,t_3$ mit der Annotation a_1.

XML fordert, dass Annotationen korrekt geschachtelt sind. In diesem Beispiel haben t_1 und t_2 jedoch beide Annotationen und somit lässt sich die Annotation nicht als wohlgeformten XML-Baum darstellen. Im XML-Editor wird immer gültiges XML angezeigt, d.h. werden beide Annotationen als XML-Markup eingeblendet, wird die Annotation verfälscht dargestellt. Das Datenformat von GoldenGATE kann diese Annotation jedoch korrekt verarbeiten, weswegen die Annotation wie beschrieben gespeichert und verwendet werden kann.

GoldenGATE verfügt darüber hinaus über eine umfangreiche API und eine Schnittstelle zur Erweiterung mit NLP-Werkzeugen. So können Einschübe (engl. *plug ins*) entwickelt werden, die den Text verarbeiten und Annotationen automatisch hinzufügen. Die zugehörige Software-Bibliothek erlaubt es zudem, das Datenmodell GAMTA von GoldenGATE für andere Werkzeuge zu verwenden [Sau11]. Das Datenmodell gibt keine Annotationsschemata vor, alle Annotationen und deren Attribute können frei gewählt werden. Die GAMTA-Bibliothek enthält die gängigen Methoden zur Ein- und Ausgabe von GAMTA-Dateien, weswegen NLP-Fließbänder auf GoldenGATE-Basis auch ohne den Editor ausgeführt werden können.

Die Verarbeitungsstufen, die für NLCI entwickelt wurden, sind Einschübe für GoldenGATE und somit können alle Stufen in GoldenGATE ausgeführt und ihre Ergebnisse überprüft werden. Die Annotationsschemata der NLCI-Verarbeitungsstufen sind in Anhang D angegeben.

2.3 Die linguistische Datenbank WordNet

WordNet ist eine linguistische Datenbank, die Einträge für Substantive, Verben, Adjektive und Adverbien enthält. Präpositionen und Artikel sind nicht in WordNet enthalten. WordNet wurde dazu entwickelt, von Programmen automatisiert oder automatisch verwendet zu werden [Mil95, Fel98]. Es wird bei NLCI dazu verwendet, um ein größeres sprachliches Spektrum abzudecken.

WordNet enthält für jede *Bedeutung* eines Wortes einen Eintrag, der Synset genannt wird. Ein Synset fasst alle Wörter gleicher Wortart zusammen, welche dieselbe Bedeutung haben; jedes Synset verfügt darüber hinaus über eine kurze Beschreibung, Glosse genannt, und ggf. über Beispielsätze zur Verwendung. Da Wörter mehrere, verschiedene Bedeutungen haben können, können sie gleichzeitig in verschiedenen Synsets erfasst sein. Synsets existieren für alle vier syntaktischen Kategorien; eine Suche in WordNet kann auf eine Kategorie eingeschränkt werden.

Beispiel

(n.1) **boat**, gravy boat, gravy holder, sauceboat: *a dish (often boat-shaped) for serving gravy or sauce*

In diesem Beispiel ist das erste Substantiv-Synset (gekennzeichnet mit *(n.1)*) für das Wort „boat“ abgedruckt; dieses Synset ist nicht mit Beispielsätzen zur Verwendung versehen. Nach dem Typ des Synsets und seiner Nummer, hier *(n.1)* für Substantiv (engl. *noun*), stehen alle Wortbedeutungen, zu denen diese Bedeutung von „boat“ synonym ist. Die Beschreibung der Bedeutung ist nach dem Doppelpunkt angegeben: Es handelt sich bei diesem Eintrag für „boat“ also nicht um ein kleines Schiff, sondern um eine Sauciere.

Im Folgenden wird eine Bedeutung durch das Wort, den Typ und die Nummer des Synsets gekennzeichnet; die Bedeutung von „boat“ im obigen Beispiel erhält den Bezeichner `boat.n.1`; diese Bezeichner werden sowohl für die konkrete Bedeutung eines Wortes als auch für das Synset verwendet.[1] Aus dieser Benennung folgt, dass ein Synset mehrere Bezeichner hat: aus jedem enthaltenen Begriff entsteht ein Bezeichner für das Synset.

Beispiel

(n.0) boat: *a small vessel for travel on water*

(n.1) boat, gravy boat, gravy holder, sauceboat: *a dish (often boat-shaped) for serving gravy*

[1] WordNet verwendet einen anderen eindeutigen Bezeichner für das Synset. Für unsere Betrachtung können wir dieses Detail jedoch vernachlässigen.

or sauce
(v.0) boat: *ride in a boat on water*

Hier sind zwei weitere Synsets von „boat“ abgedruckt; das erste ist ein weiteres Substantiv-Synset, das die Bedeutung „kleines Schiff“ hat. Das letzte ist ein Verb-Synset, welches das Fahren mit einem Boot meint.

WordNet enthält zwei verschiedene Arten von Beziehungen zwischen den Einträgen: lexikalische Relationen und Synset-Relationen. Lexikalische Relationen verbinden Wörter miteinander, z.B. die *Antonym*-Relation zwischen „light“ und „dark“. Ebenso gibt es morphologische Beziehungen, bspw. zwischen einem Substantiv und einem Verb. Diese Relationen werden nicht zwischen Synsets aufgespannt, da sie zunächst nur für einzelne Wörter und nicht für das gesamte Synset gelten müssen. Synset-Relationen verknüpfen Synsets miteinander und spannen so ein semantisches Netzwerk auf. WordNet enthält unter anderem Relationen für Ober- und Unterbegriffe (Hyperonym und Hyponym) und Teil und Ganzes (Meronym und Holonym). Die Hyperonym-/Hyponym-Beziehungen zwischen den Synsets in WordNet begründen eine Taxonomie von Synsets, an deren Wurzel „entity“ steht. Mit dieser Taxonomie lassen sich z.B. auch Ähnlichkeitsmaße zwischen Synsets oder Wörtern bestimmen. Zusätzlich zu dieser vertikalen Ordnung der Begriffe ergeben sich aus den Meronym-/Holonym-Beziehungen horizontale Verknüpfungen. Das so aufgespannte Netzwerk ist sehr nützlich für die Verarbeitung von natürlichsprachlichen Eingaben (siehe hierzu auch Abschnitt 5.2.4).

Beispiel

In den obigen Beispielen bestehen verschiedene Relationen: Zwischen den Wörtern „to boat“ aus `boat.v.0` und „boat“ aus `boat.n.0` besteht eine morphologische Verknüpfung: „to boat“ ist die Verbform zu „a boat“.

Ausgehend von `boat.n.0` gibt es eine Kette von Hyperonymen, die über „watercraft“, „craft“ und weitere zunächst zu „physical entity“ und schließlich hin zum allgemeinsten Begriff „entity“ reicht.

Die Bedeutungen `souceboat.n.0` und `boat.n.1` sind implizit Synonyme, da sie im selben Synset stehen; das Synset trägt beide Bezeichnungen.

WordNet wurde für die Englische Sprache entwickelt; es enthält derzeit 155.287 Wörter in 117.659 Synsets mit insgesamt 206.941 Wort-Bedeutungspaaren (Stand: Version 3.1 von 2012); ein Großteil der Einträge entfällt auf Substantive.

NLCI verwendet WordNet und ist demnach auf die englische Sprache eingeschränkt. Es gibt jedoch Projekte, die gleich oder ähnlich strukturierte lexikalische Datenbanken

für andere Sprachen pflegen [BP12]. Für Deutsch könnte bspw. auf GermaNet [HF97, HH10] zurückgegriffen werden. Es ist mit 131.814 Begriffen, 101.371 Synsets und 4.199 Relationen noch etwas weniger umfangreich als sein englisches Pendant[2].

2.4 Ontologien

Der Begriff „Ontologie“ stammt aus der Philosophie und bezeichnet die Lehre aller existierenden Dinge. In der Informatik gibt es hingegen *viele* Ontologien. Es gibt händisch gepflegte und automatisch gelernte Ontologien für Weltwissen (bspw. Cyc [Len95] bzw. YAGO [SKW07] und KnowItAll [ECD+04]). Meistens modelliert eine Ontologie jedoch einen einzelnen, abgegrenzten Gegenstandsbereich.

Ontologien werden dazu verwendet, das Wissen über einen Gegenstandsbereich (d.h. einer Domäne) maschinenlesbar zu kodieren: „An ontology is an explicit specification of a shared conceptualization.“ [Gru93] Eine Ontologie beschreibt demnach formal die Konzepte einer Domäne und die Beziehungen, die zwischen den Konzepten bestehen [GOS09]. Ähnlich wie WordNet können Ontologien hierarchisch strukturierte Modelle sein; im Folgenden wird der Begriff „Ontologie“ sowohl für das Konzept als auch für ein konkretes Exemplar verwendet.

Ontologien ähneln objektorientierten Modellen nach Booch [Boo94]: Sie enthalten Konzepte (sie entsprechen Klassen), Individuen (sie entsprechen Instanzen) und Beziehungen zwischen diesen (sie entsprechen Assoziationen). Die Struktur der Ontologie wird mithilfe der Konzepte modelliert. Die Beziehungen können mit einer beliebigen Semantik definiert werden und sind uni- oder bidirektional. Eine häufige Beziehung ist die *istEin*-Beziehung. Gibt es eine *istEin*-Beziehung, so bildet sie über Generalisierung und Spezialisierung eine Taxonomie [Wel09] und damit das strukturelle Rückgrat der Ontologie; dabei ist Mehrfachvererbung ausdrücklich erlaubt. Weitere Beziehungen können vom Ontologie-Autor angelegt und mit einem zulässigen Quell- und Zielbereich versehen werden; das bedeutet, dass eine Beziehung nur zwischen bestimmten Konzepten (bzw. deren Spezialisierungen oder Instanzen) hergestellt werden darf. Davon abgesehen gibt es unidirektionale Beziehungen zwischen Konzepten oder Instanzen und konkreten Werten von Datentypen bspw. Zahlen.

Ontologien sind keine passiven Informationsspeicher, sondern können mit Regeln erweitert werden. Regeln bestehen hierbei aus zwei Teilen: Einer linken und einer rechten Seite. Links steht die Regelbedingung, rechts die Konsequenzen, die eintreten wenn die Bedingung erfüllt ist. Regeln können Bedingungen an die Ontologie stellen, bspw. dass

[2] Quelle: http://www.sfs.uni-tuebingen.de/GermaNet/, Stand: Mai 2015, zuletzt besucht am 05.02.2016.

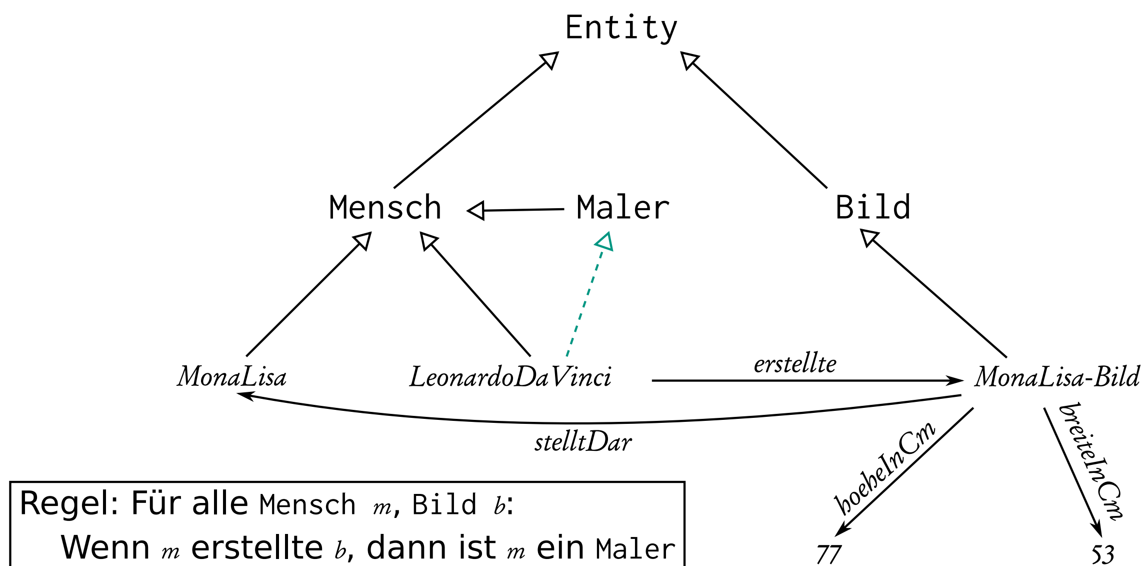


Abbildung 2.5: Beispiel für eine Ontologie. Konzepte sind in einer Festbreitenschrift abgebildet, Instanzen in einer Kursivschrift.

eine bestimmte Beziehung vorhanden sein muss (z.B. „für alle Instanzen *m* der Klasse *Mensch* muss gelten: Zwischen *m* und einer beliebigen Instanz von *Ort* gibt es genau eine Beziehung *istGeborenIn*“). Ebenso können Regeln dazu verwendet werden, Informationen implizit zu hinterlegen (z.B. „wenn ein *Mensch* ein *Bild* gemalt hat, ist er automatisch eine Spezialisierung von *Maler*“).

Inferenzmaschinen (engl. *reasoner*) können dazu verwendet werden, Anfragen an Ontologien auszuwerten. So ist es möglich, auf nicht explizit in der Ontologie hinterlegtes implizites Wissen zuzugreifen. Regeln können dabei sowohl in der Ontologie hinterlegt (wie im voranstehenden Absatz) oder in der Inferenzmaschine eingebaut sein (z.B. die Auswertung einer speziellen Beziehung, die Konzeptidentität kodiert). Eine Ontologie sollte gültig (d.h. widerspruchsfrei) sein, da Anfragen auch mithilfe von logischer Inferenz beantwortet werden und aus widersprüchlichen Voraussetzungen nichts Sinnvolles gefolgert werden kann.[3] Inferenzmaschinen prüfen, ob eine geladene Ontologie gültig ist, bevor sie Anfragen beantworten. Liebig beschreibt Inferenzmaschinen und ihre Funktionsweise ausführlich in seinem technischen Bericht [Lie06] und gibt zudem eine Übersicht über moderne Inferenzmaschinen.

[3] Es gibt Ontologien, die Widersprüche in unterschiedlichen Wissensbereichen zulassen; dann muss bei einer Anfrage jedoch angegeben werden, welcher Bereich für die Beantwortung der Anfrage verwendet werden soll.

Beispiel

In Abbildung 2.5 ist eine Ontologie beispielhaft abgebildet. Die Beziehungen sind als Pfeile dargestellt, wobei der Typ der Beziehung als Beschriftung am Pfeil angebracht ist; eine Ausnahme stellt die *istEin*-Beziehung dar, die durch einen Pfeil mit geschlossener, nicht ausgefüllter Spitze dargestellt wird (genau so wie bei UML-Klassendiagrammen). Die zur Ontologie definierte Regel ist im Kasten unter der Ontologie angegeben.

`Entity`, `Mensch`, `Maler` und `Bild` sind Konzepte. Das Konzept `Thing` ist das allgemeinste Konzept und steht an der Wurzel der Taxonomie; alle anderen Konzepte sind direkte oder indirekte Spezialisierungen. *MonaLisa*, *LeonardoDaVinci* und *MonaLisa-Bild* sind Instanzen der jeweiligen Konzepte. *77* und *53* sind Werte vom Datentyp Integer und beschreiben entsprechend der verwendeten Beziehung die Maße des Bildes. Die horizontal gezeichneten Beziehungen kennzeichnen, wer das Bild *MonaLisa-Bild* erstellt hat und wen es zeigt.

Die Regel besagt, dass alle Menschen, die ein Bild gemalt haben, Maler sind. Die gestrichelte `istEin`-Beziehung zwischen `LeonardoDaVinci` und `Maler` ist eine abgeleitete Information, die nicht explizit in der Ontologie gespeichert ist, sondern von einer Inferenzmaschine aufgrund der Regel ermittelt werden kann.

Stellt man einer Inferenzmaschine die Anfrage, alle Menschen zu ermitteln, so antwortet sie mit {`MonaLisa, LeonardoDaVinci`}. Fragt man die Inferenzmaschine nach Malern, erhält man die Antwort {`LeonardoDaVinci`}.

Ontologien können unter anderem in dem vom *World Wide Web Consortium (W3C)* vorgeschlagenen Austauschformat OWL abgespeichert werden [BVH$^+$04]. OWL basiert auf der RDF-Syntax, die es erlaubt, logische Aussagen über Ressourcen zu beschreiben. OWL ist Grundlage vieler Entwicklungen im semantischen Internet (engl. *semantic web*) und mittlerweile zum Quasi-Standard geworden [Stu09, S. 148]. OWL kann daher von vielen Programmen eingelesen, dargestellt und bearbeitet werden.

Protégé ist ein graphischer Editor, der die Erstellung und Bearbeitung von OWL-Ontologien komfortabel gestaltet [KFNM04].[4] Zudem werden viele Analysewerkzeuge bereitstellt, die direkt aus der Oberfläche heraus verwendet werden können. So wurden auch verschiedene Inferenzmaschinen in Protégé integriert, wodurch Entwickler durch die Anzeige impliziter Informationen unterstützt werden [Stu09, S. 99]; so kann bspw. fortlaufend geprüft werden, ob die Ontologie gültig ist. Abbildung 2.6 zeigt ein Bildschirmfoto

[4] Protégé kann von http://protege.stanford.edu/ bezogen werden. Zuletzt besucht am 20.10.2015.

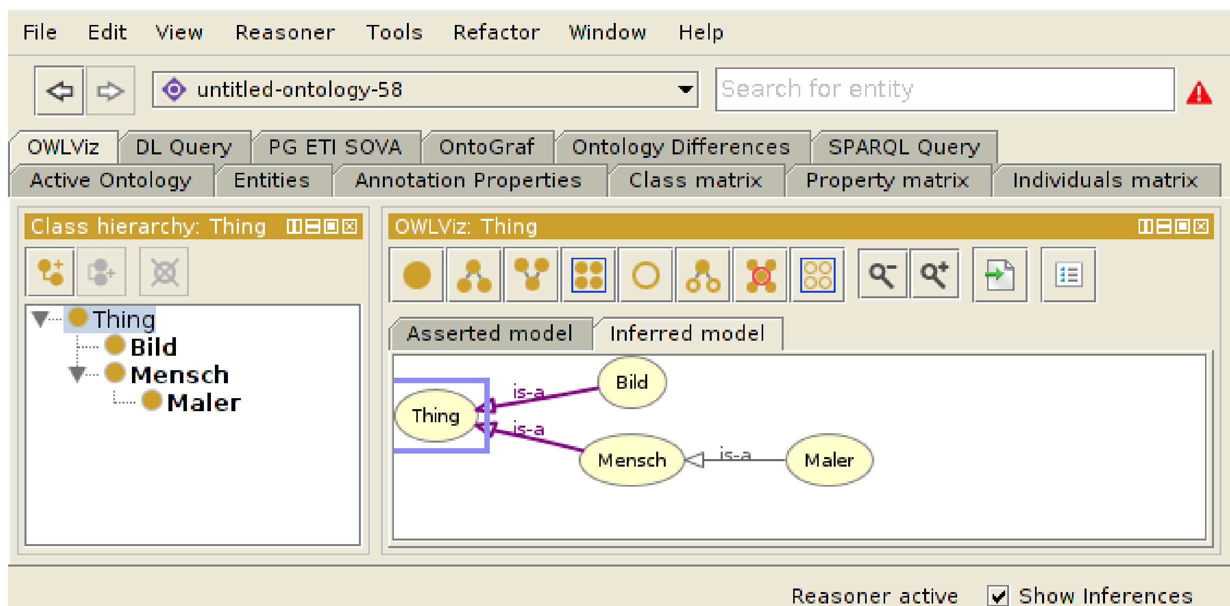


Abbildung 2.6: Beispiel für eine einfache Ontologie geöffnet im Editor Protégé.

von Protégé mit einem Ontologie-Ausschnitt, die in diesem Fall als Graph dargestellt wird.

Einen Überblick über RDF, OWL und das Semantic-Web gibt Horrocks in Referenz [Hor08]. Eine ausführliche Einführung von Ontologien im Allgemeinen, dem manuellen und automatisierten Aufbau, Diskussionen über verschiedene (teilweise automatische) Konsistenzprüfungen sowie diverse Ontologiewerkzeuge gibt Stuckenschmidt im oben genannten Ontologiebuch [Stu09, SS. 98, 129, 148].

Kapitel 3

Systematik und Überblick

„I believe that architecture, as anything else in life, is evolutionary. Ideas evolve; they don't come from outer space and crash into the drawing board.“

Bjarke Ingels, 2009

Die Vorstellung der Grundlagen hat gezeigt, dass ein breites Spektrum an hochwertigen computerlinguistischen Werkzeugen frei verfügbar ist (für eine Übersicht über Projekte und Ansätze zur Programmierung in natürlicher Sprache siehe Kapitel 4). Die obere Hälfte von Abbildung 3.1 zeigt den in der Vergangenheit häufig verfolgten Ansatz zur Programmierung in natürlicher Sprache. In einem Schritt wird die textuelle Eingabe verarbeitet und eine interne Darstellung von den Nutzerwünschen bzw. -befehlen erzeugt (1). Im zweiten Schritt wird diese interne Darstellung dann auf die Zielplattform übertragen und Quelltext erzeugt oder Aktionen ausgeführt (2). Diese Zweiteilung besteht teilweise nur Gedanklich, sodass es auch Ansätze gibt, direkt auf Elemente der Zielplattform abzubilden; Modellerstellung und Abbildung erfolgen dann in einem Schritt.

Der untere Teil von Abbildung 3.1 zeigt im Gegensatz dazu, dass bei NLCI zuerst die Zielplattform explizit modelliert wird (1) und NLCI den Text darauf aufbauend modelliert (2). Ergebnisse von plattform- und domänenunabhängigen Analysen (wie z.B. die Erkennung von wiederholt ausgeführten Aktionen) werden als Zusatzinformation im Text annotiert, ebenso die Verknüpfung der Textelemente mit der API (3). Im letzten Schritt werden die Annotierungen analysiert und der Quelltext erzeugt (4). Erst in diesem Schritt müssen die Eigenheiten der Zielplattform (Syntax, Sprachumfang, etc.) beachtet werden.

Um die Qualität des erzeugten Programms bewerten zu können, müssen wir die Fragen beantworten, ob die Wünsche des Benutzers korrekt umgesetzt wurden und wie vollständig die Umsetzung ist. Das bedeutet, dass wir den generierten Quelltext mit dem

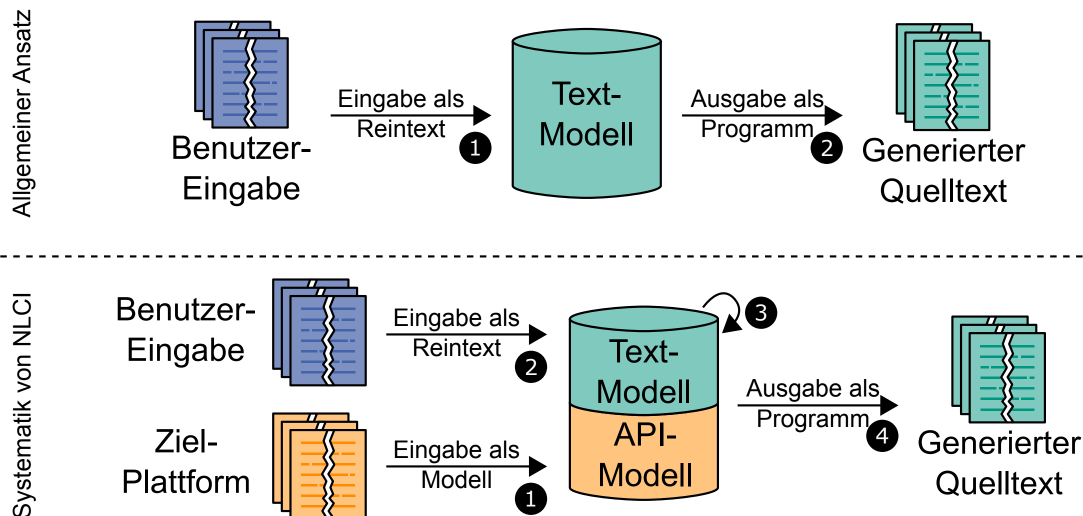


Abbildung 3.1: Der allgemeine Ansatz im Vergleich zur NLCI-Systematik.

erwarteten vergleichen. Um dies effizient durchführen zu können, benötigen wir also eine Sammlung von Eingabetexten mit zugehörigen Erwartungswerten. Diese Textsammlung ist das im folgenden Unterkapitel vorgestellte NLCI-Korpus. Es dient im Projekt zur Aufgabendefinition für die Analyseschritte und zur Evaluation.

Basierend auf dem Kurzüberblick in diesem Kapitel betrachten wir in Abschnitt 3.1, welche Anforderungen an NLCI gestellt werden, welche Charakteristiken die NLCI-Architektur aufweist und welche Vorteile ihr Aufbau bietet. In Abschnitt 3.2 werden die Anforderungen beschrieben, die wir umgekehrt an eine anzusteuernde API stellen. Diese Arbeit beschreibt nicht nur die abstrakte Struktur von NLCI, sondern stellt auch eine prototypische Implementierung vor und beschreibt ihre Leistungsfähigkeit. Abschnitt 3.3 beschreibt die empirischen Grundlagen dieser Arbeit und der Evaluation.

3.1 Charakteristika der NLCI-Architektur und des bestehenden Prototyps

Die prototypische Implementierung der NLCI-Architektur enthält eine Reihe verschiedener Analyseverfahren, die gemeinsam das Ziel der Programmsynthese aus natürlichsprachlichen Eingabetexten erfüllen. Abbildung 3.2 zeigt den derzeitigen Stand dieses Prototyps und die integrierten Analyseverfahren.

Die folgende Aufzählung geht kurz auf die Anforderungen an ein System für Programmierung in natürlicher Sprache ein, die zur Entwicklung bzw. Einbindung der Analysen geführt haben.

- Eine schrittweise Verarbeitung der Eingabe ist wünschenswert. So werden die einzelnen Analysen voneinander entkoppelt und zusätzliche Analysen können leicht hinzugefügt werden.

- Der erste, offensichtliche Schritt, ist die Identifikation der im Text genannten Konzepte (Klassen bzw. Instanzen und Methoden). Diese Identifikation basiert auf einem Modell der Zielplattform (d.h. der anzusprechenden API) und stellt einen expliziten Zusammenhang zwischen dem Eingabetext und der Zielplattform her.

- Nachdem die aufzurufenden Methoden identifiziert wurden, müssen die benötigten Argumente im Text identifiziert werden. Teilweise handelt es sich um primitive Datentypen, teilweise um Klassen oder Instanzen der Zielplattform.

- Der heutige Sprachgebrauch und z.B. der Schulunterricht trainieren Menschen darauf, nicht immer dasselbe Wort zu verwenden, sondern „abwechslungsreich“ zu formulieren. Das System muss daher damit umgehen können, wenn in Texten Begriffe verwendet werden, die so nicht in der API definiert wurden. Ebenso kann es vorkommen (es ist sogar sehr wahrscheinlich), dass für ein Konzept unterschiedliche Begriffe im selben Text verwendet werden. Um festzustellen, dass zwei Begriffe im Text dasselbe Konzept oder dieselbe Sache meinen, sollte eine Korreferenzanalyse durchgeführt werden. Mindestens muss eine Analyse für synonyme Begriffe durchgeführt werden.

- Menschen halten sich beim Beschreiben von Abläufen nicht streng an die chronologisch richtige Reihenfolge. Daher kann man bei der Programmsynthese die Texte nicht streng von vorne nach hinten lesen und interpretieren, wie man es mit einem Stück Quelltext machen würde; die textuelle Reihenfolge kann sich von der gewünschten Ausführungsreihenfolge unterscheiden: Die Aktionen müssen in eine chronologisch korrekte Reihenfolge gebracht werden.

- Beschreibt man eine Aktionsfolge, so möchte man Aktionen z.B. mehrfach ausführen können, ohne sie Wort für Wort noch einmal zu nennen. Ebenso wird es vorkommen, dass Aktionen für oder mit eine(r) Menge von Konzepten ausgeführt werden soll (z.B. „schalte den Fernseher und das BluRay-Gerät ein“ oder „schalte alle

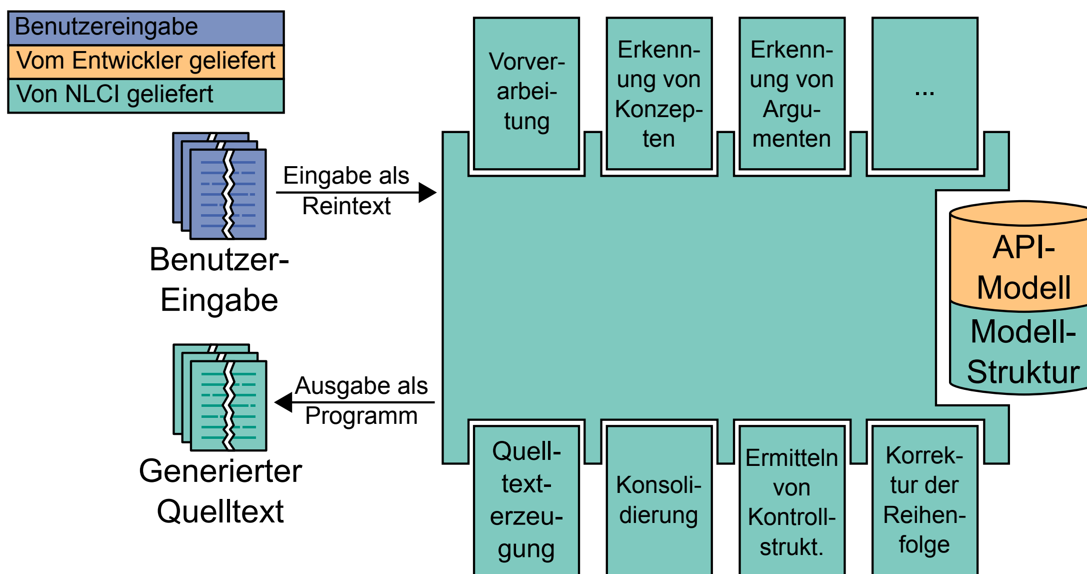


Abbildung 3.2: Eine detailliertere Übersicht über NLCI mit den bereitgestellten Einschüben.

HiFi-Geräte ein“). Aus Sicht eines Programmes müssen für derartige Ausdrücke Schleifen angelegt werden.

- Aktionsfolgen müssen nicht sequentiell verlaufen, sondern der Benutzer könnte auch Aktionen nennen, die zeitgleich ablaufen. Der Text muss also auch darauf untersucht werden, ob Parallelität benötigt wird.

Zur Erfüllung dieser Anforderungen benötigen wir für die einzelnen Analyseverfahren zudem computerlinguistische Grundlagen, z.B. Parser oder Wortartmarkierer. Diese Grundlagen sind in Abbildung 3.2 als „Vorverarbeitung“ zusammengefasst. Einzelne Einschübe für NLCI bearbeiten den Eingabetext mit Standardwerkzeugen der Computerlinguistik (wie bspw. dem Stanford'schen Wortartmarkierer) und annotieren die Ergebnisse im Eingabetext. Die übrigen Einschübe können dann auf diese Grundlagen zugreifen.

Der Prototyp folgt dabei der integrierten Architektur von NLCI, d.h. alle Analysekomponenten sind nach demselben Prinzip aufgebaut und ins System integriert. Für den Datenaustausch gibt es eine flexible aber wohldefinierte Schnittstelle: den Eingabetext. In ihm werden die Analyseergebnisse direkt annotiert. Zudem können alle Analysekomponenten auf das API-Modell zugreifen; es muss zwar vom API-Entwickler geliefert werden, aber NLCI schreibt die Modellstruktur vor, sodass die benötigten Informationen an wohldefinierten Stellen stehen.

Es gibt verschiedene Gründe und Vorteile für die Wahl einer derartigen Architektur:

- Grundlegende Anforderungen einer Analyse (wie beispielsweise Zugriff auf das Modell der API, Ein- und Aufgabeformate, Speicherung der Ergebnisse usw.) werden von der Rahmenarchitektur bereitgestellt. Die Schnittstellen eines Analysemoduls nach außen hin sind somit leicht verständlich und kompakt.
- Viele andere Ansätze, die über reines Anwenden von NLP-Werkzeugen herausgehen, verzahnen die Analysen eng miteinander oder beziehen Domänenwissen direkt in die Sprachanalyse ein. So ist es schwer möglich, einzelne Bausteine separat zu verwenden und weiterzuentwickeln. Umgekehrt ist es oft eine große Herausforderung, eine neue Analysestufe in ein bestehendes System zu integrieren. Der verwendete Entwurf entkoppelt die verschiedenen Analysen, erlaubt aber gleichzeitig, eine Analyse auf eine bestehende aufzubauen und so eine Abhängigkeit einzuführen.
- Aufgrund der sehr lockeren Kopplung der Analysekomponenten können diese nun einzeln betrachtet, verbessert oder ganz ausgetauscht werden: So lange die Informationen, die ein Einschub (z.B. ein syntaktischer Parser) bereitstellt, in derselben Form an den Eingabetext annotiert werden, profitieren die darauf aufbauenden Analysen direkt von den Verbesserungen. Nach dem Austausch kann die Performanz des Gesamtsystems erneut (extrinsisch) evaluiert werden, um zu beurteilen, ob der neue Ansatz besser funktioniert als ein bereits integrierter.

Vergleicht man diese Architektur beispielsweise mit der DeepQA-Architektur von IBM, die im Watson-Projekt [FBCC+10] entwickelt wurde, so fallen Ähnlichkeiten und Unterschiede auf: Ähnlich wie bei DeepQA versteht NLCI die Analysemodule als unabhängig voneinander und macht keine Einschränkungen, welche Ergebnisse erzeugt, wie sie ermittelt und wie sie den übrigen Modulen bereitgestellt werden müssen (d.h. es gibt kein vordefiniertes Annotationsschema). Ebenso verbietet NLCI nicht, dass es mehrere Einschübe gibt, die dasselbe Analyseziel haben – lediglich ihre Annotationen müssen unterscheidbar bleiben. Zu einer Annotationsart können aber auch mehrere Einschübe existieren. Es ist z.B. denkbar, dass ein Einschub die Ergebnisse eines vorangegangenen analysiert und korrigiert; für später nachfolgende Einschübe wäre die Korrektur transparent.

Bei DeepQA wird im Falle von Jeopardy! mithilfe der bekannten Aufgaben aus der Vergangenheit gelernt, welche Lösungsmodule wann verwendet werden sollten. Eine solche Lernkomponente gibt es bei NLCI nicht, da es naturgemäß keine Datengrundlage für ein Lernverfahren gibt; wir verfügen über kein Korpus aus natürlichsprachlichen Formulierungen und zugehörigen implementierten Programmen. Ein interaktives System könnte

aus Rückmeldungen vom Benutzer lernen, mit dessen Formulierungen umzugehen; das Lernen einer Übersetzung von Text nach Programm ist aber nahezu ausgeschlossen. Die Aufgabe, die Analyseergebnisse zu aggregieren und zu interpretieren, übernehmen bei NLCI die abschließenden Einschübe direkt vor der Quelltexterzeugung am Ende des Prozesses.

Betrachtet man NLCI als Bibliothek für die Erstellung natürlichsprachlicher Benutzerschnittstellen, so entstehen bezüglich der seiner pragmatischen Verwendung ebenfalls Vorteile:

- Das System kann von „in NLP untrainierten“ Entwicklern verwendet werden, um eine Anwendung oder API sprachlich ansprechbar zu machen. Hierfür ist keine tiefer gehende Expertise nötig – lediglich das Domänenwissen in Form des API-Modells muss bereitgestellt werden.

- Die Codeerzeugungskomponente ist unabhängig von der Ziel-API; d.h. sofern nur die API aber nicht die Programmiersprache geändert werden soll, kann die Codeerzeugungskomponente unverändert eingesetzt werden.

- Die sprachliche Erschließung von Programmen ist damit auch für kleinere Zielgruppen/Anwendungen möglich und ggf. wirtschaftlich tragbar.

- Die fertige Plattform kann auch dazu genutzt werden, neue Ideen zu testen; wir zielen in dieser Arbeit zwar auf eine (relativ hoch-abstrahierte) Programmierung ab – aber möglicherweise lässt sich die NLCI-Architektur mit ihren hier vorgestellten Komponenten auch für einen anderen Anwendungsfall verwenden. So ist zum Beispiel sehr gut denkbar, dass ein Kunde (d.h. ein Domänenexperte) Abnahmetests für eine Anwendung in natürlicher Sprache formuliert (ähnlich einer *User Story* bei *Scrum*) und dass diese Tests dann automatisch in Quelltext übersetzt werden.

Bei der Entwicklung des Prototyps haben wir auf verschiedene externe Lösungen (wie bspw. den probabilistischen Wortartmarkierer aus Stanford) zurückgegriffen. Sofern diese externen Lösungen trainierbar oder anpassbar sind, haben wir uns dazu entschlossen, keine Anpassung – und insbesondere kein Training anhand unseres Korpus’ – durchzuführen. Obwohl ein Training einfach durchgeführt werden kann, wird aus verschiedenen Gründen darauf verzichtet: Zum einen wird für das Training ein umfangreiches Korpus benötigt. Das Training des Wortartmarkierers aus CoreNLP wird mit über 910’000 Wörtern durchgeführt, wobei das NLCI-Korpus lediglich etwas mehr als 20’000 umfasst.

Zum anderen sind die Ergebnisse der Evaluation dann nur erreichbar, wenn die NLP-Vorverarbeitung auch trainiert wird. Ohne Training ermöglicht die Evaluation eine Einschätzung, wie gut NLCI ohne Training funktioniert. Daraus folgt:

- Die Analysen können nur so gut werden, wie die externen Lösungen auf denen sie aufbauen. Teilweise sind sie verbesserungswürdig.
- Die Ergebnisse des Prototyps sind als untere Schranke zu interpretieren. Ein Training mit typischen Texten einer Domäne sollte die Ergebnisse verbessern.

3.2 Anforderungen an die API

Aufgrund der eingangs genannten Zielgruppe und der erwarteten Komplexität der Programme, müssen wir einige Anforderungen an die anzusprechende API stellen.

Zunächst gibt NLCI vor, dass ein objektorientiertes Modell der API bereitgestellt werden muss. Diese Einschränkung ist technischer Natur und man kann sich leicht vorstellen, dass Module nicht-objektorientierter Systeme in ähnlicher Weise modelliert werden können. Darüber hinaus müssen sich die APIs für eine Programmierung durch den Endbenutzer eignen, d.h. hinreichend weit von internen Datenstrukturen abstrahieren. Naheliegend sind APIs von Systemen, die sich über eine grafische Benutzeroberfläche steuern lassen. Verfügt eine derartige Anwendung nicht über eine entsprechende API, so lässt sich meist eine einfach anzusprechende Fassade definieren, die einige oder alle Funktionen der grafischen Benutzeroberfläche anbietet.

Behält man die natürlichsprachlichen Formulierungen im Hinterkopf, so ergibt sich eine weitere Anforderung an die API: Die Funktionalitäten müssen dort angesiedelt sein, wo sie augenscheinlich auch passieren.

Beispiel

Betrachten wir als Beispiel einen Videorekorder: Üblicherweise verfügt ein Rekorder über eine Kanalwahl sowie über Funktionen zum (geplanten) Aufzeichnen.

Eine API, die den Java-Programmierkonventionen entspricht, würde z.B. ein neues Aufnahmeobjekt mit einer Referenz auf den Rekorder erzeugen und darüber die Aufnahme starten; in etwa so:

```
new Record(VCR.getInstance().setChannel(13)).start(int minutes).
```

Dass man die Aufnahme genau so konstruieren muss und der Rekorder vorher auf den richtigen Kanal geschaltet werden muss, kann man aus diesem Ausschnitt

der API nur herauslesen, wenn man ihre Semantik begreift. Eine direkte Abbildung der Beschreibung auf ein derart entworfenes System scheint derzeit unerreichbar – zumindest, wenn es keine alternativen Schnittstellen anbietet. (Möchte man auf einem Rekorder nur den Kanal wählen, ist die skizzierte API jedoch brauchbar.)

Eine NLCI-geeignete API (oder eine entsprechende Fassade) sollte also in etwa eine Methode `VCR.record(Channel c, int minutes)` anbieten, um eine Aufnahme zu erzeugen. Diese Darstellung lässt sich leicht sprachlich ansprechen, indem man sagt „The VCR records on channel 13 for 15 minutes“. Möchte man zusätzlich auf dem aktuell gewählten Kanal aufnehmen können, oder ohne die Angabe einer Laufzeit, dann muss die API entsprechende Methoden dafür bereitstellen.

3.3 Empirische Grundlage

Die empirische Überprüfbarkeit der entwickelten Werkzeuge ist eine zentrale Anforderung an das vorgestellte Forschungsvorhaben. Daher wurde bereits vor und während der Entwicklung ein Textkorpus aufgebaut [Ham12]. Es sollte die Entwicklung lenken und am Ende zur Evaluation genutzt werden können.

Ein Korpus bezeichnet in der Sprachwissenschaft eine Sammlung von Sprachartefakten und stellt ein bewährtes Mittel der Sprachanalyse dar [McE05]. In der Computerlinguistik werden Korpora seit langem verwendet, um Verfahren der automatischen Sprachanalyse (z.B. einen neuen Ansatz zum syntaktischen Zerteilen) zu evaluieren. Verfahren, die auf Statistik basieren, benötigen zudem Trainingsdaten; diese Trainingsdaten werden häufig in Korpora zusammengefasst und so wiederverwendbar gemacht. Dann enthalten die Korpora nicht nur die Texte, sondern auch Annotationen bspw. für Wortarten oder Syntaxbäume. Beispiele für Korpora, deren Inhalt und Strukturierung finden sich in den Referenzen [FK79, Bur07, JAGL86, MSM93]. Für weitere Informationen zur Rolle von Korpora in der (Computer-)Linguistik siehe Referenz [CEE+10].

Die Texte in unserem Korpus, dem NLCI-Korpus, beschreiben Aktionsfolgen, die von einem Computersystem in ausführbare Programme übersetzt werden sollen. Um möglichst realitätsnahe Eingabetexte zu erhalten, wurden sie von verschiedenen Probanden verfasst, die über unterschiedliche technische Vorkenntnisse verfügen.

Wie Abbildung 3.3 zeigt, diente das Korpus zuerst der Problemdefinition, da die Texte auf Herausforderungen hin untersucht werden konnten. Während der Entwicklung der einzelnen Textanalysen konnten zudem echte Beispiele aus dem Korpus verwendet werden. Für den nachgelagerten Evaluationsschritt wurden weitere Texte von Probanden ge-

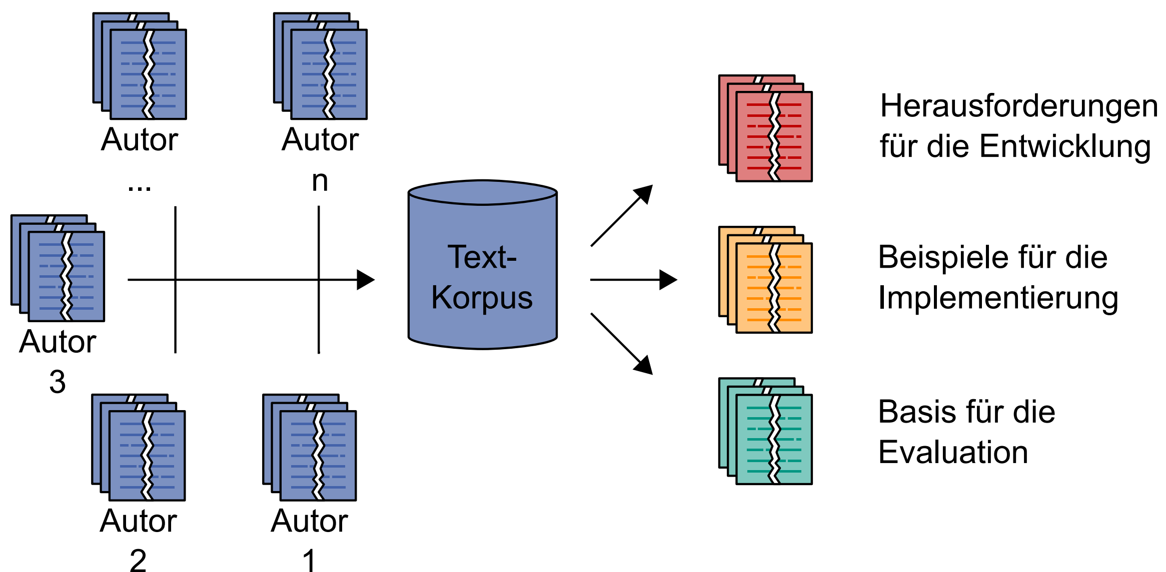


Abbildung 3.3: Empirisch erhobene Texte steuern den Forschungsprozess.

sammelt, um die Leistungsfähigkeit der jeweiligen Analyse mit echten Eingabetexten zu prüfen.

Dieses Vorgehen ist methodisch sehr wichtig: Die Problemstellung wurde vor Projektbeginn (möglichst) realitätsnah festgelegt. Während der Forschung und Implementierungsphase kann somit mit echten Problemen gearbeitet werden, anstatt sich künstliche (möglicherweise leichtere oder realitätsferne) Beispiele hierfür selbst zu formulieren. Ebenso erlaubt dieses Vorgehen die einzelnen Analysekomponenten gezielt zu verbessern; wird von eine bestimmten Analyse ein (nahezu) perfektes Ergebnis erzielt, sollte das Korpus erweitert und um schwierigere Eingabetexte ergänzt werden.

Die folgenden Unterkapitel beschreiben, wie das Korpus aufgebaut wurde und welche Aktionsfolgen von den Probanden beschrieben werden sollten.

3.3.1 Aufbau des NLCI-Korpus

Das Korpus enthält von Probanden verfasste Aktionsbeschreibungen, deren Be- oder Verarbeitung durch NLCI untersucht werden soll. Um zu beurteilen, ob ein Analyseergebnis korrekt ist oder nicht, muss zunächst der Erwartungshorizont festgelegt werden. Dieser Erwartungshorizont hängt naturgemäß davon ab, was der Proband beschrieben hat und was er als Reaktion vom System erwartet. Gibt man den Probanden nicht vor, was sie beschreiben sollen, dann muss dieser Erwartungshorizont für jeden einzelnen Text festgelegt werden. Dieses Vorgehen ist zwar möglich, jedoch ist es zu aufwändig.

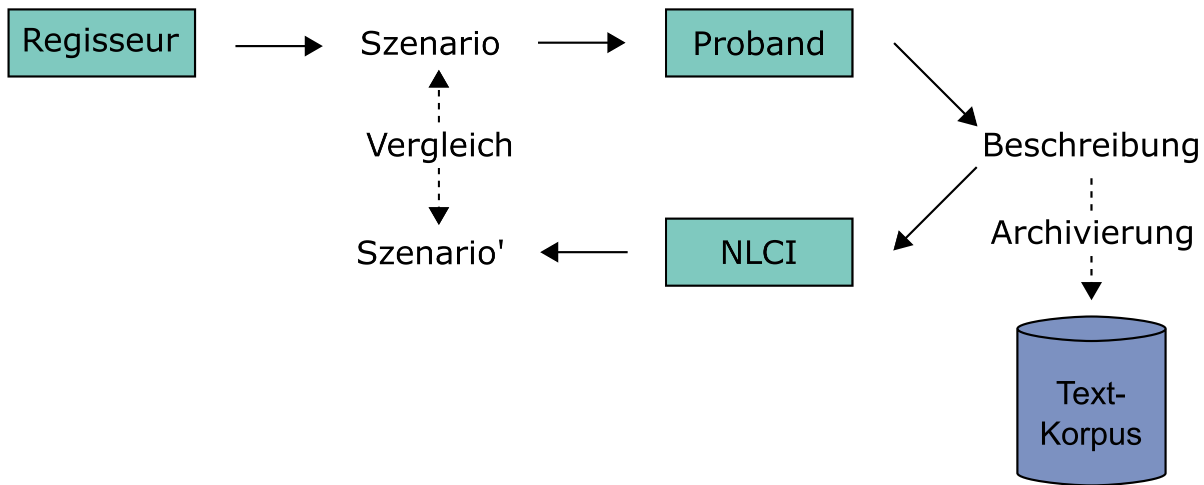


Abbildung 3.4: Erstellung des NLCI-Korpus. Von Probanden geschriebene Texte können zur Evaluation herangezogen werden.

Eine Möglichkeit den Erwartungshorizont einzuschränken wäre es, sich eine Funktionalität zu überlegen (z.B. Auflösen von Wiederholungen als Schleifen) und diese den Probanden zu erklären. Anschließend sollten die Probanden dann Texte verfassen, die die beschriebene Funktionalität nutzen bzw. benötigen. Auf diese Art würde man vermutlich ein System entwerfen und danach festlegen, wie es anzusprechen ist. Dieses Vorgehen würde die Probanden zu sehr einschränken genauso wie die Natürlichkeit ihrer Texte.

Aus diesen Gründen ist die Textsammlung für das NLCI-Korpus mithilfe von Szenarien durchgeführt worden; Abbildung 3.4 skizziert das Vorgehen. Die Entwicklung des NLCI-Prototyps erfolgte anhand der Erzeugung von 3D-Animationen und folglich enthält das NLCI-Korpus Drehbücher für 3D-Animationen. Probanden wurde zunächst erklärt, dass es ein System (NLCI) gebe, das eine natürlichsprachliche Beschreibung in eine Animation übersetzen kann. Für dieses System sollten nun Testeingaben erstellt werden. Eine Beschreibung sollte dann so sein, dass der gezeigte Programmablauf daraus erstellt werden kann. Um den Probanden eine Aktionsfolge vorzugeben, wurde zunächst von einem Regisseur ein Szenario vorbereitet; Probanden betrachten das Szenario und beschrieben es mit ihren eigenen Worten. Die Beschreibung wurde dann im NLCI-Korpus zusammen mit dem dazugehörigen Szenario erfasst. So stellen wir sicher, dass die Probanden den Text in ihren eigenen Worten verfassen und nicht auf Formulierungen von uns angewiesen sind. Andererseits erhalten wir durch die Vorgabe eines Szenarios einen festen Erwartungshorizont: Wenn der Proband das Szenario so beschreibt, wie es stattgefunden

Tabelle 3.1: Selbsteinschätzung der 66 Autoren des NLCI-Korpus

Englisch-	Programmierkenntnisse					
kenntnisse	keine	wenig	mittel	viel	k.A.	Gesamt
keine	2					2
wenig	3		1			4
mittel	7	4	7	3		21
viel	2	3	4	17		26
k.A.				1	12	13
Gesamt	14	7	12	21	12	66

bzw. er es beobachtet hat, dann sollte aus der Beschreibung wieder das Szenario (oder zumindest eine Animation mit identischer Optik) erzeugt werden können.

Die Texte im Korpus sind von verschiedenen Probanden geschrieben worden, die über verschiedene Vorkenntnisse im Bereich des Programmierens verfügen; die Auswahl der Probanden sollte möglichst breit gestreut sein, ist jedoch pragmatisch gehalten: Doktoranden unseres Lehrstuhls stellen einen Teil der Probanden, Studenten einen weiteren. Diese Gruppe verfügt naturgemäß über teilweise umfangreiche Programmierkenntnisse und die Englischkenntnisse sind erwartungsgemäß recht hoch. Die zweite Gruppe der Probanden besteht aus Nicht-Informatikern verschiedener Herkunft. Sie stehen stellvertretend für die Zielgruppe von NLCI. In beiden Gruppen befinden sich insgesamt drei Englisch-Muttersprachler, der überwiegende Teil ist jedoch Deutsch-Muttersprachler. Tabelle 3.1 fasst die Probanden zusammen. Die demographischen Daten haben wir von den Probanden mit einem Fragebogen abgefragt. Ebenso haben wir die Probanden gebeten, die Texte unter einem Pseudonym zu schreiben.

3.3.2 Verwendete Animationen

Die Animationen, die für die Erstellung des Korpus genutzt werden sollten, unterliegen einigen Einschränkungen: Zunächst sollte eine Animation nicht zu lange sein, damit die Probanden die Bearbeitung der Aufgabe nicht abbrechen. Ebenso sollte die Anzahl der enthaltenen Objekte (und damit auch der verfügbaren Aktionen) überschaubar sein. Vorüberlegungen und Probeläufe am Lehrstuhl zeigten, dass Animationen, die länger sind als 60 Sekunden, für die Benutzerstudien ungeeignet sind. Darüber hinaus möchten wir verschiedene Drehbücher für identische Animationen erhalten, damit der Vergleich der

erzeugten Animationen mit der vorgegebenen Animation einfach und objektiv möglich ist.

Daher sollen die Animationen keine interaktiven Elemente enthalten, da andernfalls die Animationen und die zugehörigen Drehbücher von den Entscheidungen der Probanden abhängen. Nicht zuletzt sollten – um eine Beeinflussung der Probanden durch uns auszuschließen oder zumindest zu minimieren – die Animationen von unbeteiligten Dritten erstellt worden sein.

Die Animationen wurden mit Alice erzeugt, einem in der Lehre weit verbreiteten Programmiersystem, für das es viele Kurse für jugendliche Programmieranfänger gibt. Daher kann man im Internet viele, von verschiedenen Personen erstellte Alice-Animationen finden und herunterladen. Sie spiegeln jedoch häufig den Kenntnisstand am Ende eines Programmierkurses wieder und sind daher oft länger als 60 Sekunden, bestehen aus verschiedenen Szenen oder enthalten Interaktivität. Damit erfüllen sie unsere Anforderungen leider nicht und sind damit für den Aufbau eines Korpus ungeeignet.

Die Animationen sollten zwar den Anforderungen genügen, aber auch nicht selbst gewählt sein. Daher entstanden die Animationen der Basis von Animationen Dritter, jedoch wählten wir die Handlung selbst und kontrollierten so Schwerpunkte, Animationskomplexität und -länge.

Im Verlaufe des Projekts entwarfen acht Regisseure auf diese Art zwölf Animationen sowie zwei Szenerien (d.h. Startpunkte für Animationen jedoch ohne Animation). Tabelle 3.2 fasst sie kurz zusammen, eine detaillierte Aufstellung ist in Anhang C gegeben.

3.3.3 Ideal-Drehbuch für Animationen

Für jede Animation wurde ein ideales Drehbuch (Ideal-Drehbuch) erstellt und dem Korpus hinzugefügt. Es orientiert sich an der tatsächlichen Implementierung der jeweiligen Animation. So ist für jede Aktion im Skript ein Satz im Drehbuch zu finden. Daher dienen die Ideal-Drehbücher auch der Identifikation von Aktionen, die im Allgemeinen von den Probanden als implizit vorhanden angenommen werden und daher nicht beschrieben werden. Die Ideal-Drehbücher dienten auch als Beispiel für neue Projektteilnehmer sowie als einfache Startpunkte bei der Implementierung von Analysen. Die Analyseergebnisse, die von NLCI erzeugt werden sollen, bezeichnen wir als Musterlösung; je nach Betrachtungsweise handelt es sich um Zwischenergebnisse oder die tatsächliche Implementierung der Animation. Die Ideal-Drehbücher haben folgende Eigenschaften:

- Die Ideal-Drehbücher verzichten auf komplizierte Sprachkonstrukte, Anaphern und andere Referenzen,

Tabelle 3.2: Eine Übersicht über die Animationen im NLCI-Korpus. Animationen, die nur eine Szenenbeschreibung enthalten, sind mit einem Stern (*) gekennzeichnet.

Animation	Regisseur	Länge	LOC	#Objekte	#Methoden
Alice	A	32 s	31	3	23
Beachday	B	20 s	28	9	11
Cheerleader	C	19 s	12	2	11
Cowboy	D	17 s	13	2	6
Dragon	C	16 s	11	4	9
Farm	B	40 s	39	9	15
Moon	A	38 s	36	3	25
Rabbit	E	23 s	21	3	15
Mummy*	F	–	–	6	–
Saloon*	F	–	–	4	–
Wizard	G	10 s	13	6	8
Woman	H	29 s	22	3	12

- sie halten sich an die von Alice vorgegebene Nomenklatur für die Bezeichnung der Akteure und Aktionen,
- sie liefern für alle Aktionen die benötigten Argumente,
- sie enthalten für jede Aktion einen (Teil-)Satz und
- sie beschreiben in chronologischer Reihenfolge genau das, was in der Animation zu sehen ist.

Beispiel

Gegeben ist eine Szenerie, in der ein Mann `m` und eine Frau `f` mit 10 Metern Abstand voneinander stehen und in die Kamera blicken. Die Animation zeigt dann, wie sich die Frau zum Mann dreht und mit der linken Hand winkt; anschließend geht sie zwei Meter auf ihn zu. Daraufhin dreht sich der Mann zur Frau hin und läuft zu ihr.

Die API stellt dazu die folgenden Methoden bereit:

- `turnTo(Object o)` zum Drehen zu einem Objekt `o`,
- `wave(Hand h)` zum Winken mit einer Hand `h`,

- `walk(int m)` um `m` Meter weit zu gehen sowie
- `goTo(Object z)` um zu einem Ziel `z` zu laufen.

Die Methode `goTo(...)` sei dabei in der API so implementiert, dass zunächst `turn- To(...)` aufgerufen und anschließend die Laufbewegung ausgeführt wird; so stellt die API sicher, dass sich ein Modell zuerst zu seinem Ziel hin dreht und dann beginnt loszulaufen.

Ideallösung

Eine Ideallösung beschreibt den Verlauf der Handlungen z.B. folgendermaßen:

- The woman turns to face the man.
- The woman waves with her left hand.
- The woman walks two meters.
- The man turns to face the woman.
- The man goes to the woman.

Jede (sichtbare) Aktion der Animation wird exakt beschrieben und alle benötigten Argumente für die zu generierenden Methodenaufrufe sind vorhanden.

Nicht-ideale Beschreibung

Eine nicht-ideale Beschreibung könnte folgendermaßen aussehen:

- The woman waves.
- Before that, she turns to face the man.
- He turns to face the woman.
- The woman walks two meters and he to her.

Diese Beschreibung ist ebenfalls verständlich, enthält jedoch neben der unchronologischen Reihenfolge weitere Schwächen: Die Beschreibung weicht in der Aktionsreihenfolge von der Animation ab, da die Frau zuerst zwei Meter auf den Mann zugeht, bevor dieser sich umdreht.

Damit der Methodenaufruf für das Winken erzeugt werden kann, muss eine Hand angegeben werden – da die Beschreibung keine solche Angabe macht, kann die erste Aktion nicht in einen Methodenaufruf übersetzt werden. Hier müsste das System entweder den Benutzer fragen, oder sofern möglich die möglichen Varianten ermitteln und daraus eine auswählen.

Der letzte Satz enthält eine Ellipse; das Prädikat „walks“ des ersten Teilsatzes wird im zweiten wieder benötigt. Hierbei handelt es sich um ein Stilmittel, dessen Auflösung schwierig sein kann.

Zuletzt enthält die Beschreibung Personalpronomen; da in diesem Text lediglich zwei Personen (unterschiedlichen Geschlechts) vorkommen, ist eine Auflösung jedoch einfach. Ist die Animation komplexer und enthält mehr Teilnehmer, so ist die Auflösung für einen Algorithmus nicht mehr einfach möglich.

In der Evaluation werden die Ideal-Drehbücher als Referenz für die Bewertung von gesammelten Texten verwendet. Ursprünglich war geplant, diese Ideal-Drehbücher manuell mit den Analyseergebnissen zu annotieren und als Gold-Standard für die Evaluation des Systems zu benutzen. Bei der Analyse der Drehbücher der Probanden stellte sich jedoch früh heraus, dass sie teilweise nicht die vorgelegte Animation beschrieben, sondern Aktionen übersprangen oder zum Beispiel einen zu hohen bzw. niedrigen Detaillierungsgrad verwendeten. Daher wurde bei der Evaluation für jedes Drehbuch eine eigene Musterlösung erstellt; das entsprechende Ideal-Drehbuch diente dabei als Ausgangspunkt. Eine Musterlösung enthält die Annotationen (z.B. für Methodenaufrufe) genau so, wie der Proband die Animation beschrieben hat – unabhängig davon, wie sie tatsächlich abläuft.

3.3.4 Ablauf der Texterstellung

Der Korpusaufbau erfolgte über einen größeren Zeitraum und die insgesamt 66 Probanden wurden von verschiedenen Personen angeleitet. Um allen Probanden einen möglichst gleichen Startpunkt zu schaffen, haben wir auf eine ausführliche mündliche Erklärung verzichtet, sondern den Probanden eine schriftliche Aufgabenbeschreibung vorgelegt. Anfangs sollte die Aufgabenstellung handschriftlich bearbeitet werden, im Projektverlauf sind wir dazu übergegangen, die Anleitung online bereitzustellen und direkt mit einem Fragebogen zu verknüpfen. Inhaltlich unterscheiden sich die Papier- und die Onlinevariante nicht. Lediglich die Animationen sind in der Online-Variante direkt eingebunden und können innerhalb eines Fragebogens direkt abgespielt werden; in der Papiervariante wurden die Animationen auf den Rechnern der Probanden zur Verfügung gestellt bzw. auf einem Speicherstift ausgehändigt. Ein Beispiel für einen Online-Fragebogen findet sich in Anhang B.

Allen Probanden wurde zunächst das Ziel von NLCI erläutert und anhand eines Beispiels gezeigt, welche Eingaben in das System gegeben werden sollen und welches Ergebnis daraus folgt. Dazu wurde den Probanden eine kurze Animation mit wenigen Akteuren und wenigen Aktionen vorgespielt. Anschließend wurde das Drehbuch dazu besprochen. Das Drehbuch zur Beispielanimation ist zudem Teil des (ausgedruckten) Fragebogens, so-

Tabelle 3.3: Eine Übersicht über die Texte im NLCI-Korpus. Animationen, die nur eine Szenenbeschreibung enthalten, sind mit einem Stern (*) gekennzeichnet.

	Texte		Sätze		Wörter		
Animation	∑T	↯	∑S	S/T	∑W	W/T	W/S
Alice	4	0	82	20	652	163	8
Beachday	15	1	263	17	2 551	170	10
Cheerleader	12	1	154	12	1 428	119	10
Cowboy	12	4	132	11	1 541	128	11
Dragon	12	0	157	13	1 671	139	11
Farm	17	1	387	22	3 649	214	10
Moon	4	1	73	18	753	188	11
Rabbit	23	1	409	17	3 389	147	9
Mummy*	15	0	92	6	1 366	91	10
Saloon*	15	0	98	6	903	60	10
Wizard	11	4	142	12	1 072	97	8
Woman	6	0	119	19	1 136	189	9
Gesamt	146	13	2 130	14	20 111	138	9

dass die Probanden die Aktionen in der Animation mit dem Drehbuch Schritt für Schritt vergleichen konnten. Die Handlungen in der Beispielanimation sind (auch zeitlich) klar voneinander getrennt und laufen streng sequentiell ab.

Anschließend wurde den Probanden die eigentliche Animation gegeben, die sie eigenständig (d.h. ohne Ideal-Drehbuch oder Vorgaben) beschreiben sollten. Der Fragebogen enthielt hierfür eine Aufgabenbeschreibung sowie eine Liste der Akteure und deren verfügbaren Methoden. Jeder Proband arbeitete dabei an einem eigenen Rechner und konnte die Animation beliebig oft abspielen und pausieren.

Im ersten Durchlauf entstanden auf diese Weise 14 Drehbücher von stark unterschiedlicher Länge und Güte (bzgl. Sprachniveau, Detaillierungsgrad, Vollständigkeit und so weiter). Die beschriebene Animation ist sehr einfach, enthält ebenso wie die Beispielanimation nur wenige Akteure sowie Aktionen und verzichtet auf Parallelität und Ähnliches. Die Animation ist bewusst einfach gehalten, da zunächst eine einfache, imperative Programmierung in natürlicher Sprache ohne zusätzliche Schwierigkeiten untersucht werden sollte. Aus diesen Gründen – und um den Korpus weiter zur vergrößern – wurden in darauf folgenden Arbeiten weitere (komplexere) Animationen erstellt. Je nach For-

schungsfrage enthalten die hierbei entworfenen Animationen Schwierigkeiten bzw. geben zusätzliche Freiheitsgrade wie beispielsweise Parallelität. In den dazugehörigen Aufgabenstellungen wurde hierauf explizit hingewiesen, um den Probanden die Möglichkeit zu verdeutlichen. Auf diese Weise konnten wir die Struktur der Texte und damit die daraus folgenden Schwierigkeiten bei der Textanalyse provozieren.

Manche Probanden beschrieben mehr als eine Animation, aber kein Proband verfasste mehr als einen Text zu einer gegebenen Animation. Der aktuelle Umfang und die Eigenschaften des Korpus ist in Tabelle 3.3 zusammengefasst. Die Tabelle führt alle Animationen auf und dazu jeweils die Anzahl der erfassten Texte ($\sum$T) und die davon von der Verarbeitung ausgeschlossenen (↯). Die folgenden zwei Spalten geben an, wie viele Sätze für diese Animation insgesamt geschrieben wurden ($\sum$S), und wie viele Sätze die Texte im Durchschnitt haben (S/T). Die letzten drei Spalten geben an, wie viele Wörter pro Animation geschrieben wurden ($\sum$W), wie viele Wörter ein Text im Durchschnitt umfasst (W/T) und wie lang ein Satz im Schnitt ist (W/S). In der letzten Zeile sind die Summen bzw. Durchschnitte für den gesamten Korpus angegeben.

Im späteren Projektverlauf wurde ein unterstützender Editor entwickelt, mit dessen Hilfe weitere Drehbücher entstanden sind. Der Editor verhält sich minimal-invasiv und gibt Vorschläge bzw. Anmerkungen zum aktuellen Stand des Drehbuchs (ähnlich wie die Prüfung von Rechtschreibung und Grammatik in einem Textverarbeitungsprogramm) [Bes14]. So sollte es ermöglicht werden, gängige Fehler in Drehbüchern zu erkennen und den Autor direkt auf sie hinzuweisen. Der Editor kann beispielsweise prüfen, ob alle im Text verwendeten Entitäten in der Szenenbeschreibung des Drehbuchs genannt wurden. Die hierbei entstandenen Texte lassen sich etwas besser verarbeiten, jedoch enthalten auch sie Mängel: der Editor findet einerseits nicht alle Fehler und gibt andererseits nur Hinweise, schreibt aber keine Formulierungen vor und verhindert bzw. behebt keine Mängel aktiv.

3.3.5 Aufbereitung der Texte

Die gesammelten Texte enthalten naturgemäß Tipp- und Grammatikfehler. Um die Verarbeitung nicht unnötig zu erschweren, wurden die Texte nach der Erhebung von Fehlern befreit. Hierunter fallen hauptsächlich Tippfehler und Fehler, die von Nicht-Muttersprachlern gemacht werden, wie z.B. falsche Personalpronomen (*he* oder *she* anstelle von *it*). Inhaltliche Fehler oder Mängel (bspw. fehlende Informationen) wurden bei dieser Bereinigung nicht behoben.

Texte, die zu weit von der Aufgabenstellung abweichen oder bspw. extrem viele Aktionen aus der Animation auslassen oder neue hinzufügen, wurden entsprechend markiert und von der weiteren Verarbeitung ausgeschlossen.

Die Originaltexte sind nach wie vor im Korpus enthalten, wurden jedoch nicht für die Evaluation verwendet. Dass eine automatische Korrektur grundsätzlich möglich ist, legen nicht nur unsere Arbeiten am unterstützenden Editor nahe [Bes14], sondern auch die verfügbaren Rechtschreib- und Grammatikkorrekturen in aktuellen Textverarbeitungsprogrammen. Eine Kombination aus diesen beiden Korrekturansätzen könnte derartige Fehler erheblich reduzieren. Darüber hinaus sollte sich ein interaktives System mit Rückfragemöglichkeit für das Programmiersystem und Korrekturmöglichkeit für den Benutzer relativ einfach erstellen lassen.

Beispiel

In der Animation „Cowboy“ steht ein Kamel vor einer Pyramide. Ein Cowboy kommt hinzu, springt und klatscht. Am Ende der Animation nicken beide mit dem Kopf.

Der Aktionsteils des Ideal-Drehbuchs lautet: „The camel nods twice and at the same time the cowboy walks two meters. Then the cowboy turns right by a quarter revolution. The cowboy jumps and claps his hands three times. The cowboy and the camel nod four times.“

Ausschluss aufgrund mangelhafter Beschreibung

Eine Beschreibung wurde ausgeschlossen, da sie fast ausschließlich Aktionen verwendet, die von der API nicht angeboten werden. Hinzu kommt, dass die gewünschte Bedeutung des zweiten Satzes im folgenden Auszug völlig unklar ist.

„The cowboy steps in the middle of the picture. His arms turns [sic] from up to down. The camel eats grass from the ground an then looks up to the cowboy.“

Ausschluss aufgrund zu feingranularer Beschreibung

Ein Drehbuch enthält zu feingranulare Beschreibungen, die nicht unbedingt falsch sind, jedoch nicht verarbeitet werden können: „It’s [the camel] looking at the left hand side of the screen, from where a cowboy emerges. He is wearing a black hat, brown duster, grey pants[,] and black shoes and is visible from the side. He’s showing stubble. As the cowboy advances towards the animal, it bows down until its muzzle touches the sandy ground.“

Korrektur von Tippfehlern und der Grammatik

Manche Drehbücher enthalten Grammatikfehler, die aufgrund mangelnder Englischkenntnisse entstehen, jedoch mit einer einfachen Grammatikkorrektur erkannt und behoben werden können, z.B.: „The camel swishes with his [sic] tail. “ und „The cowboy clapps [sic] three times. “

Die Texte im Korpus enthalten darüber hinaus keine Vorverarbeitungsergebnisse oder Annotationen, die mithilfe von NLP-Werkzeugen erzeugt wurden. Das Korpus dient in erster Linie der Performanzmessung des gesamten NLCI-Systems; hierbei soll die Performanz der Vorverarbeitung durch NLP-Werkzeuge mit einfließen. Soll ermittelt werden, wie gut die Analysen von NLCI unabhängig von diesen NLP-Werkzeugen sind, könnte das Korpus (teil-)manuell mit computerlinguistischen Annotationen wie Wortartmarkierungen oder syntaktischen Strukturen versehen und so ein idealer, fehlerfreier Startpunkt für die weiteren Analysen geschaffen werden. Teilweise wurden NLP-Fehler für einzelne Evaluationen entfernt, um Performanz zu bestimmen, die bei optimalen Eingabedaten erreicht werden kann. Die Ergebnisse der Evaluation werden in Abschnitt 6.2 diskutiert.

3.3.6 Zusammenfassung

Das NLCI-Korpus ist ein monolinguistisches, wachsendes Korpus. Es besteht derzeit aus 146 Texten auf Englisch, die insgesamt zwölf verschiedene Alice-Animationen und zwei Szenerien beschrieben. Bei den Texten handelt es sich um Schriftsprache und nicht um Transkriptionen von gesprochener Sprache. Die Texte wurden von Tipp- und einfachen Grammatikfehlern befreit, die Originaltexte archiviert. Die korrigierten Texte wurden einheitlich formatiert, bevor sie im Korpus gespeichert wurden.

In Anhang C finden sich ausführliche Übersichten über die Probanden, die Animationen sowie die Texte im Korpus. Die Fragebögen sind in Anhang B zusammengefasst. Das Korpus ist online unter https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Corpus verfügbar und auszugsweise in Anhang C abgedruckt.

3.4 Zusammenfassung

Dieses Kapitel gab einen Überblick über die Architektur von NLCI. NLCI sieht einen horizontal entkoppelten Bearbeitungsablauf vor und ermöglicht so, dass neue Analysekomponenten einfach hinzugefügt werden können. Kapitel 5 geht detailliert auf die entwickelten Analysen ein. Die im Prototypen bereits enthaltenen Komponenten basieren

großteils auf frei verfügbaren computerlinguistischen Programmen, die im Projektverlauf nicht auf NLCI angepasst bzw. trainiert wurden. Informationen über die Ziel-API werden NLCI als Modell bereitgestellt. Diese Ziel-API muss objektorientiert sein und sich für die Programmierung durch Endbenutzer eignen.

Die Entwicklung von NLCI – sowohl der Architektur, als auch des Prototyps – erfolgte auf Basis des NLCI-Korpus, das Eingabetexte bereitstellt. So stellen wir an die Analysen keine synthetischen sondern echte Anforderungen und die Komponenten können direkt mit echten Eingabedaten getestet werden.

Kapitel 4

Verwandte Arbeiten

„We think it is time to take another look at an old dream – that one could program a computer by speaking to it in natural language.“

Henry Lieberman & Hugo Liu, 2006

Das Ziel dieser Arbeit, Rechner natürlichsprachlich zu programmieren oder Programme und APIs anzusprechen, wurde in der Vergangenheit bereits aus verschiedenen Blickwinkeln betrachtet. Dieses Kapitel stellt zu Beginn Arbeiten mit verwandten Zielen vor. Die Palette an Ansätzen reicht von den Anfängen in den 1970er Jahren bis hin zu jüngeren Arbeiten. Der zweite Teil stellt Arbeiten vor, die sich mit der Sprachverarbeitung in der Softwaretechnik auseinandersetzen. Der dritte Teil des Kapitels betrachtet Arbeiten, die sich mit speziellen Teilaufgaben beschäftigen, die von Analysen von NLCI übernommen werden.

4.1 Programmieren natürlicher Sprache

Programmieren in natürlicher Sprache wird in der Forschung seit den 1960er Jahren immer wieder aufgegriffen. Zunächst stellten berühmte Informatiker wie Dijkstra und Hill fest, dass das Ziel nicht erreichbar oder nicht wünschenswert wäre [Dij63, Dij64, Hil72]. Aber es gab auch Forscher wie Sammet und Miller, die eine gegensätzliche Meinung vertraten. Sie beschrieben das Ziel als sehr erstrebenswert, stellten aber auch fest, dass es mit den damaligen Mitteln nicht erreichbar war [Sam66, Mil81].

Das wohl erste große System ist NLC, der Natural Language Computer, von Ballard und Bierman aus dem Jahr 1979 [BB79]. NLC ist eine Rechenmaschine, die Berechnungen auf Matrizen ausführen und diese als Tabellen interpretieren kann. Es zeigt das

Berechnungsergebnis direkt an, damit der Benutzer das Ergebnis seiner Befehle beurteilen kann und verarbeitet Befehle wie „add x1 to x2“. Befehle können auch in Schleifen durchgeführt werden, indem zunächst der Befehl genannt wird und anschließend nach um Wiederholung gebeten wird , z.B. mit „repeat the last command 5 times“ oder „repeat for all entries in row 3“ [BB80]. Die Sprachanalyse wird in NLC syntaktisch mithilfe von manuell erstellten Mustern durchgeführt. NLC versucht Referenzen aufzulösen; für die Auflösung von Fachbegriffen kommt ein domänenspezifisches Wörterbuch zum Einsatz. Bezogen auf die eingeschränkte Domäne (und das damit eingeschränkte Vokabular) funktioniert NLC zufriedenstellend [BBS83]. Probanden konnten Testaufgaben in einer Evaluation zu über 90% lösen, ohne über Programmierkenntnisse zu verfügen. Ballard und Bierman diskutieren auch, welche Entwicklungen sie in Zukunft erwarteten: Programmieren in natürlicher Sprache sollte die Eingabesprache nicht einschränken – dafür erwarten sie jedoch domänenspezifische Systeme.

AppleScript ist eine Skrip- oder Programmiersprache, die möglichst natürlichsprachlich aussehen soll und großteils auf HyperTalk zurückgeht [Goo98]. Dazu werden die Schlüsselwörter und die Syntax so gewählt, dass das Programm sehr gut lesbar wird; Anweisungen in AppleScript lesen sich fast wie richtige Sätze. Bemerkenswert ist, dass AppleScript eine lange Zeit multilingual war, d.h. dass die Schlüsselwörter in verschiedene Sprachen übersetzbar waren; mittlerweile unterstützt AppleScript aber nur noch englische Eingaben. Obwohl diese Programmiersprache sehr natürlich aussieht, bleibt sie doch eine Programmiersprache mit expliziten Strukturen (Schleifen, if/then/else-Verzweigungen) und unterstützt das Behandeln von Ereignissen (wie dem Öffnen oder Schließen eines Programms). AppleScript ist nur durch Skripte erweiterbar, nicht durch Klassen. Es dient somit nur der Interaktion mit bereitgestellten Objekten, z.B. eine Applikation, ein Dokument oder Seiten daraus sowie Peripheriegeräten des Rechners.

Price et al. stellten 2000 mit NaturalJava ein System vor, das die natürliche Sprache mit dem Programmieren stärker verbinden möchte [PRZH00]. So bietet es eine natürlichsprachliche Oberfläche um Java-Programme zu schreiben, umfasst jedoch keine Interpretationskomponente. Vielmehr erwartet es vom Benutzer, dass er konkrete Programmierschritte diktiert. Möchte er bspw. eine for-Schleife in einer Methode verwenden, muss er dies konkret auch so fomulieren: „Create a for loop that iterates from 0 to 9.“ Ebenso muss man zu erstellende Klassen konkret benennen und erhält keine Unterstützung vom System. Somit erzeugt NaturalJava keine Programme aus Geschichten, Anforderungen oder Spezifikationen, sondern überführt „vorgelesenen“ Quelltext wieder in korrekte Java-Syntax. Welche Java-Konstrukte oder Methoden verwendet werden, bleibt dem Be-

nutzer überlassen. NaturalJava dient lediglich dem Erstellen von Quelltext, Navigieren in, oder Bearbeiten von, existierendem Quelltext wird nicht unterstützt.

KlarDeutsch ist eine domänenspezifische deutsche Programmiersprache. Es ist eine sehr einfach gehaltene Sprache im Stile von Pascal und wird für Anlagensteuerung durch Ingenieure genutzt. KlarDeutsch ist ein Vertreter einer kleinen Nische, stellt aber eine tatsächlich eingesetzte Sprache dar, die nicht im Rahmen eines Forschungsprojekts entstand, sondern in der Industrie. sEnglish wird ebenfalls kommerziell eingesetzt und ermöglicht es, Programme für MATLAB in einer englischen, domänenspezifischen Sprache zu verfassen.[1] Als vorteilhaft wird angeführt, dass komplexe MATLAB-Programme so verfasst werden können, dass sie ohne weitere Schulung oder Einarbeitung verstanden und abgeändert werden können. Die Texte in sEnglish lesen sich jedoch sehr technisch und hölzern und aufrgund der Domäne enthält ein sEnglish-Text Variablennamen und -deklarationen.

Inform7 ist eine Programmierumgebung, in der sich Programme für Textadventures (auch interaktive Geschichten oder *interactive fiction* genannt) schreiben und ausführen lassen [Nel05]. Bei Textadventures handelt es sich um Computerspiele, die in einer Textkonsole gespielt werden; die Spielsteuerung findet über vordefinierte Kommandos (wie z.B. „go South“) statt. Derartige Spiele sind aus sog. Räumen (eher Szenen) aufgebaut, in denen Objekte und Charaktere vorhanden sind, mit denen der Spieler interagieren kann. Die Beschreibung der Spiele erfolgt in Inform7 natürlichsprachlich, wobei der Entwickler Räume und Objekte definiert, ihre (räumliche) Beziehung festlegt, deren Beschreibung verfasst und die Spiellogik anhand vorgegebener Aktionen und Strukturen vorgibt. Durch das Spielgenre und den Funktionsumfang von Inform7 lassen sich die Textadventures fast wie eine normale Geschichte auf Englisch aufschreiben. Dabei sind die Domäne und die unterstützte Syntax jedoch so stark eingeschränkt, dass man von einer domänenspezifischen Programmiersprache sprechend muss, nicht von natürlicher Programmierung.

Benutzer von Inform7 beklagten sich laut Lieberman und Ahmad darüber, dass beim Erstellen von Textadventures alle Details angegeben werden müssen, auch wenn diese Details mit Allgemeinwissen als „logisch“ angesehen werden. Lieberman und Ahmad entwickelten in der Folge, auch basierend auf den Erkenntnissen aus Metafor, das System MOOIDE (gesprochen „moody“) [LA10]. (Details zu Metafor finden sich weiter unten.) MOOIDE ähnelt in vielen Bereichen Inform7, jedoch unterstützt es eine deutlich komplexere Eingabesprache und bindet wie Metafor Allgemeinwissen aus Open Mind Common Sense [SLM+02] ein. In der Evaluation zeigte sich, dass MOOIDE von Programmier-Laien verwendet werden kann und dass die Einbindung von Allgemeinwissen funktioniert; allerdings treffen 40 % der Probanden auf Probleme mit dem Englisch-Zerteiler.

[1] http://www.sysbrain.com/system_english/, zuletzt besucht am 11.05.2016.

Le et al. stellen 2013 SmartSynth vor, das aus Beschreibungen Automatisierungsskripte für Smartphones ableitet [LGS13]. Hierzu annotieren Sie die APIs der Smartphones mit Zusatzinformationen und kennzeichnen so Ereignisse (z.B. Nachricht eingetroffen), Prädikate (z.B. gibt es ungelesene Nachrichten?) und Aktionen (z.B. Telefon ausschalten). Anschließend bilden sie die natürlichsprachliche Eingabe auf diese API-Komponenten ab. Neben der computerlinguistischen Verarbeitung der Eingabe analysiert SmartSynth den Datenfluss im generierten Programmcode und ermittelt z.B. bei Abbildungsschwierigkeiten, welche API-Bestandteile in Lücken eingefüllt werden könnten. Benutzer können mit SmartSynth Aktionsfolgen beschreiben, die ausgeführt werden, wenn bestimmte Ereignisse eintreten (z.B. „Wenn eine SMS eintrifft während das Telefon mit dem Auto verbunden ist, lese die SMS vor und beantworte sie mit dem Text ‚Ich fahre gerade'"). Können Teile der Benutzereingabe nicht auf Funktionalitäten abgebildet werden, wird der Benutzer um weitere Erklärung gebeten. Wie in der vorliegenden Arbeit verwendet SmartSynth die Werkzeuge des Pakets CoreNLP aus Stanford; für die Auflösung von Synonymen kommen jedoch nur Wortlisten zum Einsatz. SmartSynth erreicht eine Genauigkeit von 90% bei den veröffentlichten Fallstudien.

Macho von Cozzie et al. erzeugt aus Quelltextschnipseln und dem Benutzerhandbuch (der *man page*) ein ausführbares Programm. Zudem verwendet es Paare aus Eingabe und zugehöriger und Ausgabe, um den erzeugten Quelltext zu verbessern [CFK11]. In ihrer Veröffentlichung beschreiben sie, dass Macho erfolgreich Werkzeuge aus dem Linux-Programmbibliothek coreutils (z.B. pwd, cat usw.) generieren kann. Macho generiert ein Programm auf eine sehr kurze Beschreibung hin (meist nur ein Satz, manchmal zwei), weswegen dieser Ansatz wohl weniger Sprachverständnis enthält und mehr dem Programmieren durch Beispiele zugerechnet werden kann. Macho sollte in der Folge weitergetrieben werden, jedoch berichten die Autoren in einem (unfertigen) Papier, dass die Entwicklung von Macho 2 aufgrund nicht näher genannter Schwierigkeiten eingestellt wurde [CK12].

Auch Manshadi et al. kombinieren das Programmieren in natürlicher Sprache mit einer weiteren Technik, dem Programmieren durch Beispiele [MGA13]. Sie zielen darauf ab, dass bei Tabellenkalkulationen häufig Textwerte nach vorgegebenen Schemata abgeändert werden müssen. Sie zeigen, wie eine Manipulation einer Zeichenkette aus einer relativ ungenauen Beschreibung in natürlicher Sprache und einer kleinen Menge an Beispieldaten (Ein- und Ausgabepaare) synthetisiert werden kann. Hierzu lernt ihr Programm aus den Beispieldaten ein Maximum-Entropie-Modell. Die natürlichsprachliche Beschreibung liefert für dieses Modell zusätzliche Daten,um seine Genuigkeit zu erhö-

hen. Durch diesen Ansatz unterscheidet sich die Sprachverarbeitung in ihrem Ansatz stark von der vorliegenden Arbeit.

Gulvani und Marron stellen 2014 eine Entwicklung für die Tabellenkalkulation Excel vor, mit der komplexe Anfragen in Formeln übersetzt werden können [GM14]. Die Anfrage eines Benutzers wird dabei in ein Programm in einer domänenspezifischen Sprache überführt, das für eine vorliegende Tabelle die Anfrage beantwortet. Der Ansatz nutzt dabei aus, dass es sich um eine Tabellenkalkulation handelt und berücksichtigt den konkreten Inhalt der aktuellen Tabelle; während der Abbildung wird bspw. auf Tabellenüberschriften und Werte in den Zellen zugegriffen. Dadurch erzeugt das Verfahren für 94 % der 3570 getesteteten englischen Anfragen eine korrekte Lösung. Durch die enge Kopplung an die Domäne, nicht zuletzt auch der Anfragesprache, kann das Verfahren jedoch nicht (einfach) auf andere Anwendungen übertragen werden.

Desai et al. stellen 2015 ein Verfahren vor, mit dem natürlichsprachliche Eingaben (einzelne Sätze) in eine domänenspezifische Sprache übersetzt werden können [DGH+15]. Die Zielgruppe sind Entwickler, die eine Steuerung in einer domänenspezifischen Sprache natürlichsprachlich erschließen möchten. Das Verfahren benötigt zum Training eine Grammatik sowie eine Menge von Eingabe-/Ausgabe-Paaren aus englischen Sätzen und Programmen in der domänenspezifischen Sprache. Aus diesen wird eine Abbildung von englischen Begriffen auf die Terminalsymbole der Grammatik abgeleitet; zudem wird mit den Eingabe-/Ausgabe-Paaren ein Bewertungsschema für die erzeugten Programme trainiert. Im Betrieb erzeugt das Verfahren zu jeder Eingabe eine sortierte Liste an möglichen Programmen; es kann entweder das höchstbewertete Programm verwendet oder eine Top-N-Liste zur Auswahl angeboten werden. Das Verfahren arbeitet ausschließlich statistisch und führt keine eigene linguistische Analyse durch; teilweise basieren die Statistiken auf Syntaxbäumen, die mit den Stanford'schen Werkzeugen erzeugt werden.

Thummalapenta et al. zielen nicht darauf ab, Programme zu synthetisieren, sondern Tests [TSSC12, TDS+13]. In ihren Publikationen von 2012 und 2013 beschrieben sie, wie sie ausgehend von natürlichsprachlichen Test-Beschreibungen automatisch ablaufende Tests für webbasierte Systeme (z.B. Fehlerdatenbanken) ableiten. Besonders an ihrem Verfahren ist, dass es das zu testende System während der Testgenerierung anspricht, d.h. alle übersetzten Testschritte werden direkt auf dem System ausgeführt. So wird sichergestellt, dass der erzeugte Test die Anwendung richtig anspricht: Falls ein Test erzeugt wird, so „passt“ er auch zur Anwendungsmechanik, d.h. der Test läuft einen tatsächlich erreichbaren Kontrollfluss in der Anwendung ab; somit ist der Test grundsätzlich benutzbar. Das Verfahren versucht dabei aber nicht, die Test-Beschreibung zu verstehen, sondern verlässt sich darauf, dass die Tester die nötigen Schritte immer auf die gleiche Weise aufschrei-

ben. Sie merken an, dass die untersuchten Test-Beschreibungen häufig sehr ähnliche Formulierungen verwenden und ein stark eingeschränktes Vokabular nutzen [TSSC12]. Eine ähnliche Beobachtung wie Thummalapenta et al. machen Landhäußer und Genaid bei der Analyse von User-Stories aus einem agilen Projekt [LG12] (Details zu dieser Arbeit finden sich in Abschnitt 4.3.3). Die vorliegende Arbeit untersucht nicht, inwiefern Tests aus natürlichsprachlichen Beschreibungen mit NLCI in automatisch ablaufende Tests übersetzt werden können. Denkbar ist jedoch, dass das zu testende System und der Testtreiber für NLCI modelliert werden und so auch Programme erzeugt werden können, die das zu testende System ansprechen und prüfen, ob die erwarteten Antworten geliefert werden.

Die beschriebenen Programmiersysteme und Ansätze sind meist spezielle Umsetzungen und auf die jeweilige Domäne zugeschnitten oder enthalten nur einen sehr geringen Anteil an Computerlinguistik. Dadurch wird letztendlich eher eine „Natürlichmachung“ von synthetischen Programmiersprachen erreicht, aber von natürlichsprachlichem Programmieren kann man noch nicht sprechen. Daneben wurde in der Vergangenheit auch daran geforscht, wie Programmier-Laien Arbeitsabläufe oder Algorithmen beschreiben. Dabei entstanden flexiblere Systeme, die nur Teilaspekte der Programmierung abdecken (z.B. nur Programmgerüste erstellen) oder anderweitig auf das Einsatzszenario zugeschnitten sind.

Miller untersuchte 1981 empirisch wie Menschen andere Menschen „programmieren“ [Mil81]. Hierzu ließ er 14 College-Studenten Arbeitsabläufe so beschreiben, dass sie von anderen Menschen ausgeführt werden können. Die Arbeitsabläufe waren dabei so gewählt, dass sie üblicherweise mit oder von einem Rechner durchgeführt werden könnten. Die entstandenen Texte wurden als natürlichsprachliches Äquivalent zu einem kurzen Programm angesehen, das ein Entwickler *ad hoc* geschrieben habe. Bei der Analyse der Texte fand Miller heraus, dass die natürlichsprachlich formulierten „Algorithmen“ eher als (lineare) Anweisungsabfolgen formuliert werden, als mithilfe von (verschachtelten) Kontrollstrukturen.[2] Die entstandenen Texte waren darüber hinaus mit einem kleinen Vokabular geschrieben, obwohl die Probanden keine Anweisungen bekommen hatten, ihre Sprache einzuschränken; dies sieht Miller als Hinweis dafür, dass man das Vokabular beim Programmieren einschränken könne, ohne die Benutzer zu sehr einzuschränken. Er beobachtet aber auch, dass der Satzbau und die verwendeten (Kor-)Referenzen im Text die damaligen computerlinguistischen Werkzeuge überfordern würden; er schließt daraus, dass die beim Programmieren zulässige Syntax eingeschränkt werden müsse.

[2] Die gleiche Beobachtung kann man anhand der Texte im NLCI-Korpus machen – allerdings begünstigt unsere Aufgabenstellung derartige Formulierungen. Nur wer den Ablauf wie ein Programm(-ierer) (und damit unnatürlich) beschreibt, formuliert mithilfe von expliziten Kontrollstrukturen.

Pane und Myers fassen in Abschnitt 5-6 in ihrem technischen Bericht [PM96] die Forschungsergebnisse der folgenden beiden Jahrzehnte zusammen. Studien verschiedener Autoren zeigten, dass Programmier-Laien häufig ähnliche Beschreibungen für Standardaufgaben und Programmierkonstrukte wie Schleifen formulieren. Wenn die Probanden die Beschreibung für einen anderen Menschen verfassen, setzen sie mehr Allgemeinwissen voraus, als einem Rechner zur Verfügung steht. Diese Voraussetzungen kann man aber senken, wenn man im Szenario den Empfänger der Arbeitsanweisung als unwissend beschreibt.

In den Jahren 2001 und 2002 publizierten Pane und Myers die Ergebnisse von weiteren Studien, die der von Miller ähneln. Mit ihnen wollten sie untersuchen, wie man zukünftige Programmiersprachen möglichst anwenderorientiert gestalten kann. In Referenz [PRM01] untersuchten sie dazu, wie Kinder die Funktionsweise und andere Aspekte des Computerspiels Pac-Man beschreiben. Den Probanden wurden dazu Bildschirmfotos und Videosequenzen des Spiels gezeigt. Pane und Myers berichten, dass die Probanden überwiegend Aktionen oder Spielregeln beschreiben. Viele Probanden wählten dazu nicht die Perspektive eines Programmierers, sondern beschrieben aus der eines Spielers. Außerdem verlassen sich die Probanden darauf, dass der Leser implizite Referenzen auflösen kann. Zum Beispiel wenden sie Operationen direkt auf eine Menge von Objekten an, anstelle über die Menge zu iterieren und die Objekte einzeln anzusprechen [PMM02].[3]

Lieberman und Liu beschreiben die Herausforderungen, die an ein Programmiersystem gestellt werden, das mit natürlicher Sprache programmiert wird [LL06]. Ausgehend und inspiriert von den Studien von Pane und Myers beschreiben sie dabei nicht nur die Probleme (wie bspw. wechselnde Perspektiven oder Mehrdeutigkeiten), sondern skizzieren auch eine mögliche Lösung dafür: Sie schlagen vor, für die Textanalyse die mittlerweile verfügbaren Werkzeuge der Computerlinguistik zu verwenden. Für das Auflösen von Mehrdeutigkeiten und ähnlichen Problemen skizzieren sie einen Dialog zwischen Programmiersystem und Programmierer, bei dem der Programmierer *erklärt*, was passieren soll und das Programmiersystem Rückfragen bei Problemen stellt. Sie glauben, dass für Programmieren in natürlicher Sprache kein vollständiges Sprachverständnis notwendig ist, da der Eingabetext (d.h. das Programm) im Rahmen einer Anwendung interpretiert werden kann. Sie gehen also implizit davon aus, dass die Programmierdomäne für das System vorgegeben ist und „nur noch“ eine Abbildung des Textes auf diese Domäne nötig ist. Sie vermuten, dass dies leichter zu erreichen ist, als allgemeines Textverständnis. Insbesondere beschreiben sie die Mächtigkeit der natürlichen Sprache durch Mehrdeutigkeiten und Ausdrucksstärke nicht als Herausforderung, sondern als Vorteil [LL04]. 2005 beschrei-

[3] Diese Eigenheit lässt sich auch in den Texten des NLCI-Korpus wieder finden.

ben sie Metafor, einen Ansatz, um die Semantik eines natürlichsprachlichen Programmes zu erfassen [LL05b, LL05c]. Ausgehend von einer natürlichsprachlichen Beschreibung erzeugt Metafor ein in sich abgeschlossenes Python-Programm, das die beschriebenen Konzepte enthält. Die Methoden der erzeugten Klassen sind hier jedoch lediglich leere Rümpfe, die danach vom Entwickler gefüllt werden müssen. Um die Semantik der Beschreibung zu verarbeiten, wird Open Mind Common Sense [SLM$^+$02] verwendet; die konkrete Semantik der Begriffe spielt jedoch keine Rolle. NLCI muss im Vergleich dazu erkennen, welcher Teil der Beschreibung sich auf welches API-Element bezieht, um den gewünschten Quelltext erzeugen zu können. Metafor kann in der beschriebenen Version noch keine Aktionen auswählen und beschäftigt sich nicht mit Kontrollstrukturen oder Ähnlichem.

Mihalcea, Liu und Lieberman beschreiben 2006, wie man Aktionen und Kommentare in natürlichsprachlichen Programmen ermitteln kann [MLL06]. Ähnlich wie bei NLCI analysieren sie den Eingabetext syntaktisch und fassen im Satz Subjekte als Agenten (Akteure), deren Fähigkeiten (also die Verben) als Methoden und Objekte als Dinge oder Orte auf. Ihre Analyse verwendet Schlüsselwörter, um Wiederholungen zu erkennen, und berücksichtigt dabei auch Wörter, die Gruppen von Objekten bezeichnen; Verzweigungen des Kontrollflusses mit *if/else* ergeben sich aufgrund von Konditionalsätzen oder Modalwörtern. Mithilfe der in diesem Artikel veröffentlichten Analysen ließe sich Metafor erweitern, um konkrete Implementierungen für die erzeugten Methodenrümpfe aus dem Eingabetext abzuleiten.

Vadas und Curran beschreiben ein System, das Ähnlichkeiten mit Metafor und NaturalJava aufweist [VC05]. Sie zeigen, dass man lauffähige Programme aus natürlicher Sprache erzeugen kann. Hierzu verarbeiten sie den Eingabetext mit einem Zerteiler auf der Basis von kombinatorischen Kategorialgrammatiken (engl. *combinatory categorical grammars* (CCG)). Das Verfahren ist ähnlich aufgebaut wie NLCI: Zunächst erfolgt eine Vorverarbeitung mit einem Parser, dann folgen die semantische Aufbereitung und die Identifikation des Verbes, das die Funktionalität ausdrückt. Das identifizierte Verb und seine Argumente werden dann mit einer Muster-Tabelle abgeglichen; sind sie darin verzeichnet, wird mithilfe einer Schablone Quelltext für Python erzeugt. Die Autoren sammelten Beschreibungstexte von 12 Probanden und räumen am Ende des Artikels ein, dass die meisten Texte mit dem beschriebenen Verfahren nicht übersetzt werden können.

Little und Miller beschäftigen sich mit der Analyse von Schlüsselwörtern, mit deren Hilfe aus natürlichsprachlichen Anweisungen Skripte für ausgewählte Programme erzeugt werden können [LM06]. Die Eingabe muss grammatikalisch nicht korrekt sein, wichtig ist die korrekte Verwendung der Schlüsselwörter. Damit Parameter für Aktionen

korrekt aufgelöst werden können, muss sich der Benutzer an Konventionen und unter Umständen an eine bestimmte Syntax halten (z.B. Argumente in Anführungszeichen setzen). Synonyme dürfen verwendet werden und werden vom System erkannt. Umgesetzt wurde das Verfahren bereits für den Internet Explorer und Microsoft Word. In einer Benutzerstudie konnten sie zeigen, dass Programmier-Laien an sie gestellte Aufgaben mit dem System zu 90% umsetzen können.

Chong und Pucella beschreiben in ihrem Bericht [CP04] ein Verfahren, das es erleichtern soll, einfache sprachliche Schnittstellen für aktionsgetriebene Anwendungen (oder APIs) zu erstellen. Dazu entwerfen sie eine Architektur, welche die computerlinguistische Analyse mithilfe von CCGs anwendungsunabhängig bereitstellt. Der Ansatz ähnelt der Architektur von NLCI insofern, als der Sprachanalyse ein anwendungsspezifisches Lexikon bereitgestellt werden muss (ähnlich der NLCI-Ontologie). Dieses Lexikon enthält jedoch lediglich Begriffe aus der Anwendungdomäne und keine Informationen dazu, wie die Anwendung angesprochen werden kann. Die Anbindung an die eigentliche Anwendung (entsprechend der Quelltexterzeugung bei NLCI) ist bei Chong und Pucella ein anwendungsspezifischer Interpretierer, der die Ergebnisse des Parsers in Aufrufe an die API übersetzt.

Knöll und Menzini beschreiben Pegasus, den wohl ambitioniertesten Ansatz der jüngeren Vergangenheit [KM06]. Pegasus überführt Begriffe des Eingabetextes in eine Ideensprache, in der Konzepte unabhängig von der konkreten natürlichen Sprache beschrieben werden können. Die Begriffe in der Ideensprache werden wie ein Netzwerk miteinander verwoben und beschreiben Konzepte ähnlich wie eine Ontologie. Pegasus unterstützt eine kleine Teilmenge der deutschen und englischen Sprache, die in einer kurzen EBNF angegeben werden kann. Die Autoren beschreiben aber weder, wie Pegasus den Eingabetext in die Ideensprache überführt, noch wie daraus Quelltext erzeugt werden kann; unklar bleibt auch, ob Pegasus Kontrollstrukturen und zeitliche Bezüge auflösen kann. Pegasus wurde nicht fertig gestellt und eine jüngere Arbeit der Autoren wendet sich der Aufgabe zu, die Natürlichkeit bestehender Programmiersprachen zu erhöhen, anstatt direkt in natürlicher Sprache zu Programmieren [KGM11].

4.2 Sprachverarbeitung in der Softwaretechnik

Verarbeitung geschriebener Sprache wird in der Softwaretechnik nur zögerlich eingesetzt, obwohl die computerlinguistischen Werkzeuge heute sehr gut sind. Für einige Aufgaben

im Softwareentwicklungsprozess gibt es jedoch Werkzeuge, die ähnliche Probleme angehen, wie die vorliegende Arbeit. Dieser Abschnitt gibt einen kurzen Überblick über die Anwendung von NLP-Werkzeugen in der Softwaretechnik.

Gelhausen beschreibt in seiner Dissertation [Gel10] ein Verfahren, das natürlichsprachliche Texte in UML-Diagramme umwandelt. Gelhausen schließt sich der These Chomskys an, dass syntaxbasierte Analysen nicht dazu verwendet werden können, die Semantik eines Textes zu erfassen. Sein Verfahren basiert darauf, dass die Semantik vom Benutzer explizit im Text annotiert wird. Die entwickelte Annotationssprache kennzeichnet die Semantik mithilfe von thematischen Rollen wie sie bei Fillmore eingeführt wurden [Fil69], jedoch verwendet sie auf das Problem zugeschnittene thematische Rollen. Gelhausens Verfahren überführt den annotierten Text in einen Graph, dessen Knoten die Wörter darstellen und die Kanten die thematischen Rollen kodieren. Mithilfe verschiedener Graphtransformationen wird dieser Text-Graph dann in einen UML-Graph überführt; dabei werden basierend auf den thematischen Rollen aus den Wörtern Klassen, Relationen oder Methoden erzeugt. Das von ihm entwickelte Verfahren erzeugt so vollständige Domänenmodelle in Form von UML-Klassendiagrammen, die jedoch vor dem Einsatz in einem Softwareprojekt von einem Analysten verfeinert (d.h. meist ausgedünnt) werden müssen. Die Darstellung der semantischen Informationen bei Gelhausen ähnelt den Annotationen, die bei NLCI verwendet werden. Jedoch werden bei Gelhausen die semantischen Informationen annotiert, bei NLCI alle Zwischenergebnisse.

Körner und Landhäußer stellen ein Verfahren zur automatischen semantischen Erschließung eines Textes vor [KL10]. Es ermittelt die von Gelhausens Ansatz benötigten thematischen Rollen, indem der Text zunächst syntaktisch, dann mithilfe der Ontologie Cyc [Len95] analysiert wird. Stellt das Verfahren fest, dass eine Entscheidung nicht automatisch getroffen werden kann, so wird der Benutzer um eine Entscheidung gebeten. Auch wenn durch das Verfahren nicht der gesamte Rollensatz von Gelhausen erkannt werden kann, so zeigt es doch, dass über syntaktische Analysen, Heuristiken und Hintergrundwissen in Form von Ontologien die Semantik zu einem gewissen Grad erschlossen werden kann.

Basierend auf den Erkenntnissen von Gelhausen stellt sich Körner in seiner Dissertation [Kör14] die Frage, welche Teile des Softwareentwicklungsprozesses (teil-)automatisiert werden können. Hierzu entwickelte er verschiedene interaktive Werkzeuge, die im Prozess zwischen den Anforderungen und den Domänenmodellen angreifen. So inspiziert RESI natürlichsprachliche Spezifikationen, ermittelt linguistische Schwachstellen und schlägt teilweise Verbesserungen vor. Hierzu greift es auf NLP-Werkzeuge, WordNet und Ontologien zurück, die Weltwissen enthalten. In jüngeren Publikationen wird

gezeigt, dass das entwickelte Verfahren nicht auf Software-Spezifikationen beschränkt ist [KLT14] und mit Echtwelt-Spezifikationen umgehen kann [LKK^{+}15]. Ebenso zeigt Körner, dass seine Verfahren (ebenso wie NLCI) nicht für die konkrete Zieldomäne trainiert werden müssen, um akzeptable bis sehr gute Ergebnisse zu erzielen. Teilweise könnten seine automatischen Inspektionen in NLCI genutzt werden, um ein interaktives Programmiersystem zu entwickeln. So könnten Probleme im natürlichsprachlichen Eingabetext aufgedeckt und der Benutzer direkt um eine Korrektur gebeten werden. Für die vorliegende Arbeit liegen derartige Korrekturdialoge jedoch nicht im Fokus, weswegen diese Möglichkeiten im NLCI-Prototyp nicht genutzt werden.

Keszöcze und Kollegen stellen 2013 das Lips-System vor, das einen natürlichsprachlichen Ansatz für die Modellgetriebene Architektur (MDA) bietet [KSKD13]. Sie extrahieren mit einem Eclipse-Einschub aus natürlichsprachlichen Beschreibungen UML-Klassendiagramme, die anschließend in einem MDA-Prozess verwendet werden. Im Gegensatz zur Vorliegenden Arbeit wird bei Lips die natürlichsprachliche Beschreibung nicht auf einen vorgebenen Funktionsumfang einer API abgebildet, sondern es wird durch den Text ein neues System definiert und von Lips im Hintergrund erzeugt. Treten Konzepte (oder Synonyme) im Text mehrfach auf, müssen sie auf die bereits erzeugten Systemkomponenten abgebildet werden. Kann Lips ein Konzept nicht im System ermitteln, wird mittels WordNet versucht, Synonyme und Hyperonyme aufzulösen. Kann keine Komponente mit Sicherheit bestimmt werden, bittet Lips den Benutzer um Hilfe. Treten Mehrdeutigkeiten oder andere linguistische Probleme auf, greift Lips ebenfalls auf den Dialog mit dem Benutzer zurück [SWD12]. Das von Keszöcze und Soeken et al. vorgestellte Werkzeug zur Modellierung von Systemen kann auch dazu genutzt werden, Akzeptanztests abzuleiten. Hierzu wird bei der satzweisen Modellierung des Systems gleichzeitig ein entsprechender Testschritt in Cucumber[4] formuliert. Wurden alle Anforderungen in dieser Art erfasst, erhält der Entwickler nicht nur das Modell (und damit auch automatisch ein Klassen- und Methodengerüst), sondern auch die zugehörigen Akzeptanztests.

[4]Cucumber gibt vor, dass Tests so formuliert werden, dass ausgehend von einem Systemzustand eine Aktionsfolge (*wenn*) Schritt für Schritt beschrieben wird und am Ende eine Folge von überprüfbaren Resultaten steht (*dann*). Die Aktionen sollen dabei aus der Perspektive des Benutzers geschrieben werden, der die in der User-Story beschriebene Funktionalität benutzt. Zum Beispiel „*Wenn* ich das Programm starte und auf die Schaltfläche ‚Datei laden‘ klicke, *dann* öffnet sich ein Datei-laden-Dialog und der Dialog zeigt am Anfang das Benutzerverzeichnis des aktuellen Benutzers.“ Im Beispiel gibt es also zwei Aktionen (Programm starten sowie Schaltfläche anklicken) und zwei prüfbare Konsequenzen (Dialog öffnet sich sowie Startverzeichnis des Dialogs). Tests, die in Cucumber geschrieben sind, lassen sich automatisch in ein Methodengerüst übersetzen, das vom Testentwickler nur noch ausgefüllt werden muss. Werkzeuge zum Übersetzen von Cucumber in Quelltextgerüste existieren für viele Programmiersprachen.

Da das System zeitgleich mit den Tests definiert wird, ist eine separate Abbildung von Textelement auf Modell- oder API-Element nicht nötig: Die Tests passen automatisch zum System.

Insgesamt kann man zusammenfassen, dass es verschiedene Ansätze gibt, den Softwareentwicklungsprozess mit NLP-Werkzeugen zu verbessern. Für eine umfangreiche Aufarbeitung der automatischen Modellierung und computerlinguistischer Analyse von Anforderungsdokumenten sei auf die Dissertationen von Gelhausen [Gel10] bzw. Körner [Kör14] verwiesen.

4.3 Verwandte Arbeiten zu den Textanalysen

Die einzelnen Analyseschritte von NLCI lösen Teilaufgaben auf dem Weg vom Text hin zum ausführbaren Quelltext. Die folgenden Unterabschnitte beschreiben Arbeiten, mit denen die Analysestufen verwandt sind.

4.3.1 Datenbankabfragen mit natürlicher Sprache

Da NLCI die Textelemente der Eingabetexte auf eine Ontologie abbilden muss, ähnelt der erste Schritt Ontologie- und Datenbankabfragen mit natürlicher Sprache. In diesem Forschungsgebiet wird seit den 1970er Jahren aktiv geforscht, eine dieser frühen Arbeiten ist die von Waltz [Wal75]. Bezogen auf die Domäne (Datenbanktabellen oder Ontologien) und die Ausgabe (z.B. SQL- oder SPARQL-Abfragen) sind diese Ansätze jedoch eingeschränkt. Die Konferenz, die sich diesem Forschungsgebiet verschrieben hat, ist die International Conference on Applications of Natural Language to Information Systems (NLDB), die 2015 zum 20. Mal statt fand. Die Konferenzreihe betrachtet ein großes Themenspektrum von der Textanalyse für den Ontologieaufbau bis hin zur Informationsbeschaffung (engl. *information retrieval*). Eine gute Einführung in das Feld geben Androutsopoulos et al. in Referenz [ART95], ein jüngerer Überblick wird in den Referenzen [MBMLM13, PGA+13] gegeben.

Ein Werkzeug, das natürlichsprachliche Anfragen an Ontologien in die Anfragesprache SPARQL übersetzt, ist PANTO [WXZY07]. Ähnlich wie NLCI verwendet es das Paket CoreNLP aus Stanford, um Syntaxbäume aus den Anfragen zu erzeugen. Aus diesen Bäumen extrahiert PANTO dann Konstituenten, um eine interne Repräsentation aufzubauen, die aus Anfragetripeln (engl. *query triples*) besteht. Anschließend werden diese Anfragetripel auf Strukturen der Ontologie abgebildet; ist dieser Schritt erfolgreich, kann PANTO eine SPARQL-Anfrage ableiten.

Ein ähnliches Werkzeug zum Abfragen von Ontologien ist FREyA [DAC10]. Es benutzt ebenfalls die Stanford'schen Syntaxbäume, um aus den Benutzeranfragen SPARQL-Anfragen zu erzeugen. Das Besondere an FREyA ist die Abbildung der Textelemente auf die Ontologie-Konzepte: Zunächst werden mögliche Kandidaten über eine Textähnlichkeit bestimmt; schlägt die automatische Abbildung fehl, wird der Benutzer mit einer Auswahlliste konfrontiert. Aus der Auswahl des Benutzers lernt FREyA (halbüberwacht) und soll so ständig besser werden. Wenn es bei NCLI verschiedene Kandidaten für die Abbildung auf die API gibt, so werden alle Alternativen im Text annotiert, unter der Annahme, dass spätere Analysen eine Entscheidung treffen können. Die Evaluation von NLCI zeigt, dass das Herauszögern der Entscheidung hilfreich ist: Insbesondere für Methodenaufrufe gibt es zunächst viele Kandidaten, die jedoch aufgrund des (Text-)Kontextes ausgeschlossen werden können.

Ein ähnliches Ziel verfolgen AquaLog und sein Nachfolger PowerAqua [LFMS12]. PowerAqua stellt dem Benutzer eine Schnittstelle zur Verfügung, mit der gleichzeitig verschiedene, heterogene Ontologien abgefragt werden können. Ebenso wie PANTO überführt es die Anfrage des Benutzers in Anfragetripel. Anschließend bestimmen verschiedene Module, welche der angebundenen Ontologien die Anfrage beantworten können und ermitteln die Ergebnisse. In einem abschließenden Schritt werden die Ergebnisse bewertet und falls nötig zusammengefasst, um die endgültige Antwort zu erzeugen.

Betrachtet man anstelle von Ontologien das semantische Internet, so eröffnet sich ein großer Informationsschatz – jedoch ist es sehr schwer, ein Abfragesystem für das semantische Netz zu entwerfen. Cimiano und Unger fordern, dass Frage-/Antwortsysteme (QA-Systeme) die Lücke zwischen den Benutzern und den Daten schließen müssen, sodass die Benutzer nicht wissen müssen, wo im Netz die Antwort steht [UC11]. Sie entwerfen das Werkzeug Pythia, das ein domänenspezifisches Sprachmodell dazu verwendet, Begriffe aus der Anfrage des Nutzers in die Terminologie der Domäne zu übersetzen. Dieser Ansatz funktioniert sehr gut in kleinen, eingeschränkten Domänen, die Autoren weisen jedoch darauf hin, dass der Aufwand erheblich größer wird, wenn die Domäne größer ist.

4.3.2 Linguistische Analyse von zeitlichen Abläufen in Texten

Um die gewünschte Reihenfolge der Aktionen in einem Eingabetext und die zu seiner Umsetzung benötigten Kontrollstrukturen zu ermitteln, müssen die zeitlichen Bezüge zwischen den Aktionen betrachtet werden. Im folgenden Abschnitt wird ein kurzer Überblick über Arbeiten gegeben, die zeitliche Bezüge zwischen Aktionen oder Ereignissen herstellen.

Im Rahmen des internationalen Workshops zu semantischer Evaluation (International Workshop on Semantic Evaluation (SemEval)) werden im Vorfeld des jeweiligen Workshops Herausforderungen als Wettbewerbe formuliert. In den Jahren 2007, 2010, 2013 und 2015 gab es Aufgaben, die unserer in gewisser Weise ähneln, da Ereignisse oder zeitliche Bezüge zwischen ihnen bestimmt werden sollen. Die Aufgabe *TempEval-2* aus dem Jahr 2010 beschäftigte sich konkret mit der Herausforderung, zeitliche Beziehungen zwischen Ereignissen herzustellen. Häufig geht es hierbei um Nachrichtenbeiträge oder andere uniforme Dokumentensammlungen, deren Ereignisse auf einer Zeitachse dargestellt werden sollen. In jüngeren Jahren entwickelte sich die Aufgabenstellung jedoch vom reinen Erkennen und Einordnen weg. Im Fokus standen 2015 die dokumentenübergreifende Analyse, domänenspezifische Aufgaben (z.B. für klinische Texte) und Analysen für QA-Systeme. Die Aufgaben unterscheiden sich aufgrund der Eingabetexte stark von dem was NLCI bewältigen muss: NLCI erhält immer nur ein Eingabedokument und muss für die Aktionen darin eine Ordnung herstellen; in der Regel umfassen die SemEval-Aufgabe jedoch viele Texte, die sich inhaltlich überlappen können.

Pustejovsky et al. beschreiben, wie man exakte Zeitpunkte in Texten ermitteln kann [PKLS05]. Sie analysieren Zeitungsartikel mit dem Ziel, ein QA-System dazu zu befähigen, Anfragen wie „What happened in the automotive industry last week?“ zu beantworten. Die Analyse berücksichtigt dabei zeitliche Ausdrücke wie „last week“ und löst sie im Kontext des jeweiligen Zeitungsartikels (z.B. Veröffentlichungsdatum, Kontext, Verweise auf andere Ereignisse usw.) auf. Sie evaluieren das Verfahren mit dem TIMEBANK-Korpus [PHS+03], dessen Texte mit TimeML, einer Auszeichnungsprache für zeitliche Bezüge und Informationen [PIS+05], annotiert ist. Mit TimeML kann man die Reihenfolge von Ereignissen annotieren und exakte Zeitpunkte angeben. Bei den Texten im NLCI-Korpus gibt es keine „globale Zeitrechnung“ und alle Texte sind unabhängig voneinander; konkrete Zeitbegriffe finden sich in den Texten ebenfalls nicht.

Schilder beschäftigt sich mit juristischen bzw. offiziellen Dokumenten, um Ereignisse zeitlich einzugrenzen [Sch07]. So ermittelt sein Verfahren beispielsweise aus Texten konkrete Zeitpunkte für Ereignisse wie „entered the USA on Dec. 31, 2005“; ebenso können Einschränkungen ermittelt werden, wie beispielsweise bei „entered the USA before Dec. 31, 2005“. Aus dem zweiten Beispiel würde extrahiert werden, dass das Ereignis „entering“ vor dem 31. Dezember 2005 stattgefunden haben muss, auch wenn (aus diesem Textfragment) kein konkreter Zeitpunkt dafür ermittelt werden kann. Wie bei Pustejovsky et al. geht es darum, Ereignisse auf der globalen Zeitachse zu verankern.

Ohlbach stellt ein System vor, dass textuelle Zeitausdrücke interpretieren kann, das System for Computational Treatment of Temporal Notions (CTTN) [Ohl07]. Es interpre-

tiert Begriffe wie „noon“, „tomorrow“, aber auch Wiederholungen wie „every Tuesday“. CTTN ist mit einem Standardvokabular ausgestattet, das mit benutzerdefinierten Begriffen erweitert werden kann. Wird ein Begriff von CTTN interpretiert, erhält man einen Zeitpunkt, eine Zeitspanne oder ganze Zeitachsen. Ebenso wie bei den ersten beiden Vertretern finden sich in den Texten des NLCI-Korpus keine derartigen Ausdrücke.

Mani et al. verfolgen einen statistischen Ansatz, um zeitliche Beziehungen zwischen Ereignissen zu ermitteln [MVW$^+$06]. Das Verfahren erkennt ob ein Ereignis (direkt oder überhaupt) vor einem anderen stattfindet oder ob zwei Ereignisse zeitlich überlappen ablaufen; außerdem wird annotiert, wenn ein Ereignis während eines anderen beginnt *und* endet. Zwei weitere Beziehungen markieren es, wenn ein Ereignis ein anderes beginnt (also auslöst) oder beendet. Wie bei den Arbeiten in den vorigen Abschnitten werden die Ergebnisse mit TimeML im Text annotiert.

Chambers et al. erzeugen, ebenfalls mit einem statistischen Lernverfahren, Annotationen für dieselben Beziehungen wie Mani et al. Vorausgesetzt wird, dass die Ereignisse als solche im Text gekennzeichnet sind und miteinander in Beziehungen stehen; das Verfahren ermittelt dann, welche konkreten Beziehungen annotiert werden sollen [CWJ07]. Neben zeitlichen Attributen wird das Verfahren mit Wortart-Markierungen, Lemmata und Synsets gefüttert und betrachtet Hilfswörter und Modalwörter.

Kolya et al. setzen *conditional random fields* (CRF) dazu ein, die zeitliche Abfolge von Ereignissen zu ermitteln [KEB10]. Ein CRF ist ein Vorhersagemodell, das ähnlich arbeitet wie ein Hidden-Markov-Modell, im Gegensatz dazu jedoch auf die gesamte Eingabe zugreifen kann. Wie bei den übrigen statistischen Verfahren wird hierfür ein vollständig markierter Trainingskorpus benötigt

Berglund et al. untersuchen Zeugenberichte von Autounfällen und ermitteln die zeitliche Abfolge der Aktionen mit Entscheidungsbäumen [BJN06]. Interessant ist dieser domänenspezifischen Ansatz, da das Ergebnis an eine Simulationssoftware weitergegeben wird, die eine dreidimensionale Simulation des Unfalls erzeugt; die Analysen wurden jedoch für schwedische Texte entwickelt.

Lapata und Lascarides analysieren Beziehungen von Ereignissen innerhalb eines Satzes, z.B. zwischen einem Haupt- und einem Nebensatz. Dabei berücksichtigen sie – ähnlich wie auch NLCI – Schlüsselwörter wie „before“ und „after“. Sie führen jedoch keine Mustersuche mit den Schlüsselwörtern durch, sondern verfolgen einen statistischen Ansatz [LL05a]. Es werden hierbei jedoch nur Beziehungen innerhalb eines Satzes aufgelöst und Abfolgen aus mehreren Sätzen nicht betrachtet.

Nemec untersucht tschechische Texte und schlägt ein regelbasiertes System vor, um die Beziehungen zwischen Ereignissen zu bestimmen [Nem07]. Der vorgeschlagene Al-

gorithmus betrachtet lediglich syntaktische Merkmale und entscheided so ausschließlich auf der Grundlage des grammatikalischen Aufbaus eines Satzes. Im Zentrum der Analyse stehen finite Verben, da sie in entweder in der Vergangenheit, der Gegenwart oder der Zukunft stehen. Eine Betrachtung auf dieser Granularität ist jedoch beim Programmieren in natürlicher Sprache nicht ausreichend: Man muss die Aktionen in eine Reihenfolge bringen. Es reicht nicht aus, drei Gruppen zu bilden.

Diese und ähnliche Arbeiten verfolgen meist das Ziel, Ereignisse, Nachrichten oder Ähnliches auf der globalen Zeitachse zu verankern. NLCI muss jedoch eine lokale Ordnung der Aktionen herstellen. Zeitliche Ausdrücke oder Formulierungen wie in Nachrichtentexten kommen in Aktionsbeschreibungen praktisch nicht vor. Ebenso wenig verfügen die Aktionsbeschreibungen über keinen zeitlichen Kontext (wie das Datum einer Nachrichtenmeldung). Neben der Übersetzung von textuellen Begriffen in Zeitpunkte oder Ähnliches existiert Forschung zum sogenannten *temporal reasoning*; einen Überblick über das Forschungsgebiet geben Sanampudi und Kumari in Referenz [SK10]. Zum Beispiel beschreiben Russel et al. den *event calculus* [RN09, Kapitel 12.3]. Sie beschreiben, wie Ereignisse und ihre relative zeitliche Abfolge als Prädikate kodiert werden können. Die Autoren beschreiben jedoch nicht, wie man von einem Text (automatisch) zu einer Eingabe für das System kommt. Das Hauptaugenmerk bei solchen Arbeiten liegt auf der Interpretation oder automatischen Deduktion. Die Analysen von NLCI – sowohl die für die zeitliche Ordnung, als auch die für die Kontrollstrukturen – könnten Eingabedaten für ein solches System liefern.

4.3.3 Aufbau von API-Modellen

Eine zentrale Rolle innerhalb von NLCI nimmt die Modellierung der Domäne oder API in Form einer Ontologie ein. In der Literatur finden sich zahlreiche Ansätze, Ontologien aus bestehenden Programmen, Dokumentationen oder anderen Software-Artefakten zu erstellen und darüber andere Software-Artefakte (z.B. Tests, die Dokumentation oder Anforderungen) mit der API zu verknüpfen. Nachfolgend werden einige Ansätze vorgestellt, die dem in Abschnitt 6.2.2 vorgestellten Ansatz ähnlich sind oder deren Erkenntnisse genutzt wurden.

Cimiano et al. beschreiben eine allgemeine Vorgehensweise zum halbautomatischen Aufbau von Ontologien für beliebige Domänen [CMSV09]. Das beschriebene Vorgehen soll für Ontologie-Ingenieure eine Richtschnur darstellen, die sie durch die Phasen der Ontologieerstellung leitet: Die Einarbeitung in die Domäne, die Vorbereitung der Daten, die Modellierung und zuletzt die Evaluation und Verwendung des entstandenen

Ontologie-Generators. Außerdem beschreiben sie, wie man bestehende Wissensquellen (das sind meistens Texte) in eine Ontologie überführt und welche Werkzeuge verfügbar sind. Der Aufbau der NLCI-Ontologie unterscheidet sich in diesem konkreten Punkt von den vorgestellten Techniken, da keine (Text-)Dokumente für die Ontologie-Extraktion vorbereitet werden müssen, sondern eine API also in der Regel Quelltext oder eine Bibliothek.

Simperl et al. beschreiben ein ähnliches Vorgehen für den Ontologie-Aufbau im Allgemeinen wie Cimiano et al. [STV08]. Sie beschreiben, dass die folgenden Phasen durchlaufen werden müssen, um den Ontologie-Ingenieur bestmöglich mit Werkzeugen unterstützen zu können: Durchführbarkeitsstudie, Anforderungsanalyse, Auswahl der Informationsquellen, Populationsmethoden sowie der Werkzeuge, Vorbereitung sowie Ausführung der Population und abschließend die Bewertung sowie Inbetriebnahme der entstandenen Ontologie. Simperl et al. beschreiben detailliert, welche Aktivitäten in den einzelnen Phasen durchgeführt werden sollten und vor welche Entscheidungen man höchstwahrscheinlich gestellt wird. Da NLCI die Struktur der Ontologie vorgibt und die Wissensquelle offensichtlich nur die anzusprechende API sein kann, nimmt NLCI dem Ontologie-Ingenieur die meisten Entscheidungen ab bzw. gibt Vorgehensweisen vor.

Yang et al. definieren eine Ontologie für Klassen, Relationen, Funktionen und Instanzen eines existierenden Softwaresystems [YCO99]. Die Ontologie stellt für sie den Startpunkt für das Re-Engineering dar, da sie in der Ontologie Wissen über die Geschäftsprozesse modellieren, die mit der Software durchgeführt werden. Hierzu verknüpfen sie die Konzepte in der Ontologie mit Quelltextelementen aus der Software. Wenn auch der Anwendungsfall ein völlig anderer ist, ähneln sich die Strukturen ihrer Ontologie und der NLCI-Ontologie.

Zhang et al. verwenden auch eine Ontologie, um Stellen im Quelltext mit der dazugehörigen Dokumentation zu verknüpfen [ZWRH06]. Auch sie möchten damit den Softwareentwicklungs-Prozess verbessern, indem sie Nachvollziehbarkeit (engl. *traceability*) herstellen. Sie erzeugen dazu zunächst von Quelltext und Dokumentation zwei unabhängige Ontologien und verknüpfen diese anschließend miteinander. Die Ontologie für die Quelltextelemente wurde so entworfen, dass sie die wichtigsten Konzepte von objektorientiertem Quelltext aufnehmen kann. NLCI verwendet einen ähnlichen Ansatz, jedoch wird bei NLCI aus dem Eingabetext keine Ontologie erzeugt, sondern dieser direkt mit der NLCI-Ontologie verknüpft, welche die Quelltextelemente enthält.

Sabou erzeugt eine Ontologie aus der Dokumentation von Web-Services, um Anfragesysteme zu verbessern [Sab04]. Das Verfahren bereitet die Dokumentation auf und ermittelt Verb-Nomen-Paare, die dann einem Analysten vorgelegt werden. Er muss entschei-

den, welche Paare relevant sind und für den Ontologie-Aufbau verwendet werden sollen. Sowohl der Anwendungsfall als auch die Quelle für den Ontologie-Aufbau unterscheiden sich stark von der Aufgabe von NLCI. Sabou beschreibt jedoch auch, wie eine erzeugte Ontologie evaluiert werden kann und beschreibt Ausbeute, Präzision und die Abdeckung gegenüber einem manuell erstellten Goldstandard im Kontext von Ontologien.

Ratiu et al. stellen ein Verfahren vor, mit dem man verschiedene APIs aus derselben Domäne in einer Ontologie modellieren kann [RFJ08]. Die Autoren stellen fest, dass wichtige Konzepte aus der Domäne in allen bzw. vielen APIs vertreten sind, alle APIs jedoch noch weitere, weniger wichtige Konzepte enthalten. Durch das Zusammenfassen der verschiedenen Implementierungen fallen die Kernkonzepte aus den verschiedenen APIs zusammen und die weniger wichtigen ergeben ein Rauschen. Um dieses Rauschen zu unterdrücken, werden die APIs zunächst in Graphen überführt und miteinander verglichen. Dabei werden die Strukturen ermittelt, die ähnlich in verschiedenen Graphen existieren, und dadurch als relevant eingestuft. NLCI hat derzeit zum Ziel, eine API sprachlich zu erschließen; es ist aber nicht ausgeschlossen, das Verfahren von Ratiu et al. zu adaptieren, um verschiedene, ähnliche APIs zusammenzufassen und gemeinsam anzusprechen oder durch die Identifikation der Kernkonzepte den angebotenen Funktionsumfang einzuschränken. Mit letzterem Ansatz könnte unter Umständen die Präzision der Abbildung von Eingabetext auf Ontologie verbessert werden, wenn es in einer API viele ähnlich benannte Konzepte gibt, von denen nicht alle relevant sind.

Landhäußer und Genaid versuchen das Implementieren von Akzeptanztests in agilen Projekten zu erleichtern [LG12, Gen12]. In agilen Projekten werden Anforderungen häufig als User-Story formuliert; ob die User-Story erfolgreich umgesetzt wurde, wird vom Kunden (im Idealfall jemand aus der Fachabteilung) geprüft. Hierzu kommen Akzeptanztests zum Einsatz, die häufig in (kontrollierter) natürlicher Sprache formuliert werden, z.B. mithilfe von Cucumber. Hierzu erstellen sie eine Ontologie von Test- und Produktivquelltext und verknüpfen bereits abgeschlossene (d.h. erfolgreich mit einem Akzeptanztest getestete) User-Stories mit dem Test-Quelltext; anschließend wird für ungetestete User-Stories ermittelt, welche Test-Methoden vermutlich benötigt werden, um den Akzeptanztest zu implementieren. Zwar wird bei dieser Arbeit auch Text mit einer API-Ontologie verknüpft wie bei NLCI, jedoch liegt die initiale Verknüpfung bereits in Form von abgeschlossenen Tests vor.

4.4 Zusammenfassung

Zusammenfassend kann man sagen, dass *natürliche* Programmiersprachen vorwiegend Programmiersprachen mit natürlichsprachlichen (meist englischen) Schlüsselwörtern sind und nur eine Brücke darstellen, die vielleicht von Programmier-Anfängern beim Lernen benutzt wird. Insgesamt gibt es nur wenige Beiträge, die tatsächlich natürlichsprachliches Programmieren vorantreiben. Die konzeptionell umfangreichste verwandte Arbeit, Pegasus, ist unvollendet und wurde scheinbar zugunsten von Arbeiten am naturalistischen Programmieren aufgegeben; ob diese Arbeit später vollendet wird, bleibt unklar.

Sprachanalysen, die sich mit Teilaspekten des natürlichsprachlichen Programmierens beschäftigen, gibt es in größerer Zahl. Insbesondere die Arbeiten zur Analysen von zeitlichen Abhängigkeiten kommen unserem Vorhaben entgegen, wenn sie auch für einen anderen Anwendungsfall entwickelt wurden. Schneidet man ein Verfahren auf einen speziellen, eng begrenzten Anwendungsfall zu, funktioniert es meist sehr gut. Daraus ergibt sich jedoch nur eine Lösung für den Einzelfall. Die vorliegende Arbeit strebt ein Verfahren an, dass über Einzelfälle hinaus verwendet werden kann und einen geringen Anpassungsaufwand beim Wechsel der anzusprechenden Anwendung besitzt.

Kapitel 5

Die NLCI-Architektur im Detail

„The details are not the details. They make the design.“
Charles Eames

NLCI soll es Laien ermöglichen ihre Geräte mit natürlicher Sprache zu Programmieren. Englisch (oder eine andere natürliche Sprache) ersetzt dann die Programmiersprachen – dabei erwarten wir keine komplexe Implementierungen von Algorithmen sondern eher skriptartige Abläufe. Benutzer werden Begriffe aus der Domäne verwenden, um ihre Programme zu formulieren. NLCI muss diese domänenspezifischen Begriffe in von der Maschine unterstützte Befehle übersetzen. Hierzu werden verschiedene Analysekomponenten verwendet, die dedizierte Aufgaben haben; am Ende der Verarbeitungskette steht ein Ausgabemodul, das die gewonnenen Erkenntnisse auswertet und Quelltext erzeugt. Die Analysekomponenten von NLCI verarbeiten allesamt geschriebenen Text.

Dieses Kapitel stellt die entworfenen Analysen detailliert vor und erklärt, wie die in Kapitel 3 gesteckten Ziele erreicht werden.

5.1 Umsetzung der NLCI-Architektur

Die in Kapitel 3 eingeführte Architektur wurde dadurch realisiert, dass die einzelnen Analysekomponenten als Einschübe für GoldenGATE implementiert wurden. Einschübe können bestehende Annotationen lesen, verändern und neue hinzufügen. Sie können zudem zu Fließbändern zusammengesetzt werden, mit denen man einen Text nacheinander durch mehrere Einschübe bearbeiten lassen kann. Im Gegensatz zu anderen NLP-Rahmenarchitekturen erlaubt GoldenGATE hierbei nach jeder Fließbandstufe in den Prozess einzugreifen und die Annotationen gegebenenfalls zu bearbeiten.

Die Analysekomponenten wurden zwar als Einschübe für GoldenGATE implementiert, sind jedoch auch ohne es lauffähig. So können sie vollständig isoliert verwendet oder eigenständig ohne die grafische Benutzeroberfläche ausgeführt werden. Da in beiden Spielarten das (annotierte) Dokument die Schnittstelle zwischen den einzelnen Stufen bildet, ergibt sich eine Trennung der einzelnen Implementierungen auf eine natürliche Art und Weise: Der annotierte Text fungiert als zentraler Datenspeicher.

Durch die klare Trennung der einzelnen Analysen können sie unabhängig voneinander (weiter-)entwickelt werden; theoretisch können sich unterschiedliche Entwickler diese Aufgabe teilen. Baut eine Stufe auf den Ergebnissen einer anderen Stufe auf, so profitiert sie automatisch von einer Verbesserung ihres Vorgängers – ohne, dass sie selbst noch einmal angepasst werden muss.

Ein Hauptanliegen der NLCI-Architektur ist, dass die NLP-Stufen nicht mit Wissen bezüglich der (Text-)Domäne ausgestattet werden. Dieser Wunsch kann natürlich nicht immer erfüllt werden, da beispielsweise die Erkennung von verwendeten Klassen einer API notwendigerweise die API-Klassen kennen muss. Jedoch ist das Domänenwissen für NLCI in einer Ontologie abgelegt, wodurch die Analysen zwar auf Domänenwissen zugreifen können, jedoch das Domänenwissen ausgetauscht werden kann, ohne die Analysen selbst anzupassen. Allen Analysen stehen folglich dieselben Informationen über die Zieldomäne zur Verfügung und das Wissen über die Domäne wird nicht mit dem Sprachwissen vermischt. Diese Trennung von Verarbeitungs- und Domänenwissen erlaubt es außerdem, dass Sprachexperten mit der Entwicklung (und ggf. Implementierung) der Sprachanalysen betraut werden können oder umgekehrt Domänenexperten mit der Domänenakquise.

Die Domänenakquise wird damit zu einer zusätzlichen expliziten Aufgabe im Entwicklungsprozess. Daher soll die Akquise einfach und schnell möglich sein. Um das zu gewährleisten, gibt NLCI eine Ontologiestruktur vor, die domänenunabhängig ist und nur noch befüllt werden muss. Sie enthält die Informationen über die Ziel-API wie bspw. die verfügbaren Datentypen. Die Struktur ist dabei so einfach, dass die Ontologie für kleine APIs manuell befüllt werden kann.

Das folgende Unterkapitel beschreibt detailliert den Aufbau der Domänenontologie. Darauf folgt eine Beschreibung der Sprachanalysen, die in NLCI integriert wurden.

5.2 Die Domänenontologie – das Herz der NLCI-Architektur

NLCI benötigt (spätestens zur Quelltexterzeugung) Informationen über die konkrete Implementierung der anzusprechenden API. Dazu nutzt NLCI die strukturelle Ähnlichkeit zwischen Ontologien und objektorientierten Modellen: Wir modellieren den verfügbaren Funktionsumfang der Ziel-API als Ontologie und stellen diese dem NLCI-Prozess zur Verfügung. So ergibt sich ein eine passende und kompakte Darstellung der Zieldomäne. Die konzeptionelle Ähnlichkeit der Darstellungen macht es besonders elegant, eine objektorientierte API in einer Ontologie abzubilden. NLCI verwendet hierbei eine Ontologie im OWL-Format.

Die Ontologie wurde so gestaltet, dass sie objektorientierte Strukturen aufnehmen kann und richtet sich nicht nach dem semantischen Inhalt einer zu modellierenden API. Um die API sprachlich beschreiben zu können, fügen wir Zusatzinformationen zu den Ontologiekonzepten hinzu. Theoretisch kann so auch eine mehrsprachige Modellierung einer API erfolgen, jedoch beschränkt sich NLCI bisher auf Englisch.

NLCI stellt damit eine wiederverwendbare Ontologieschablone zur Verfügung, die mit der zu verwendenden API nur noch befüllt werden muss. Danach werden die Bezeichner bereinigt und ggf. in einzelne Wörter aufgetrennt (bspw. nach Aufbereitung aufgrund der Binnenmajuskelschreibweise). Darauf aufbauend werden die Bezeichner der Konzepte mithilfe von WordNet um Synonyme erweitert und somit das API-Modell sprachlich erweitert. Die folgenden Abschnitte zeigen, wie genau die Ontologie aufgebaut ist, wie die linguistischen Informationen modelliert werden und wie die Aufbereitung erfolgt.

5.2.1 Die Struktur der NLCI-Ontologie

Die Struktur der NLCI-Ontologie orientiert sich an der allgemeinen Struktur objektorientierter Enwürfe. Sie ähnelt der von Yang et al. 1999 vorgestellten Ontologie sowie der, die Zhang et al. verwenden, um die Nachvollziehbarkeit (engl. traceability) zwischen Quelltext und Dokumentation wieder herzustellen [YCO99, ZWRH06].

Tabelle 5.1a zeigt den Aufbau der Ontologie. Die Relationen, die in Tabelle 5.1b aufgeführt sind, werden zur Verknüpfung der Konzepte miteinander verwendet. Auf der obersten Ebene der Konzepthierarchie steht *Thing*; es stellt die Oberklasse aller Ontologiekonzepte dar und ist somit die Wurzel der Hierarchie. Darunter spannt sich die Taxonomie der Konzepte auf, die für die Darstellung einer objektorientierten API benötigt werden. Unter dem Konzept *Class* befinden sich *Object* und *Component*. Das erste bezeichnet Klassen,

die als Container auftreten können, das zweite bezeichnet die darin enthaltenen Bestandteile. Unterhalb von *Method* befinden sich *PredefinedMethod* sowie *UserDefinedMethod*, welche die Methoden enthalten. Die Instanzen des Konzepts *Parameter* umfassen die Parameter von Methoden und *DataType* enthält die programmiersprachenspezifischen (primitiven) Datentypen. Zuletzt kennzeichnet *Origin* die Herkunft eines Elements; soll bspw. eine Java-API modelliert werden, wird in einem *Origin*-Element eine JAR-Datei referenziert und mit den darin enthaltenen Klassen und Methoden verknüpft.

API-Elemente werden als Individuen der entsprechenden Konzepte dargestellt. Methoden, die spezifisch von einer Klasse implementiert werden, werden als *UserDefinedMethod* erfasst und über die *hasMethod*-Relation mit der entsprechenden Klasse verknüpft. Standardmethoden, die von der jeweiligen Programmiersprache bereitgestellt werden, werden als *PredefinedMethod* erfasst. Sie müssen jedoch nicht mit jedem *Class*-Individuum verknüpft werden. Inferenzmaschinen für Ontologien können auch Vererbungsbeziehungen auswerten, weswegen eine einmalige Registrierung derartiger Methoden ausreicht; die Verfügbarkeit dieser Methoden wird dann zur Laufzeit von der Inferenzmaschine eingeblendet.

Beispiel

Javas `StringBuilder`-Klasse wäre in der Ontologie ein Individuum des *Class*-Konzeptes. Seine Methode `append(s: String)` wäre ein Individuum des *Method*-Konzeptes und mittels einer *hasMethod*-Relation mit dem `StringBuilder`-Individuum verbunden. Beide Individuen hätten zudem eine *fromOrigin*-Relation, die beide auf das *Origin*-Individuum verweisen, das für die JAR-Datei der Java-Standardbibliothek steht. Die Methode `toString()` von Javas `Object` würde man nur ein Mal registrieren und dann mit einer Regel mit allen Elementen verknüpfen.

Bei OWL-Ontologien können an Instanzen und Konzepte beliebige Kommentare zur Dokumentation angehängt werden. Diese Dokumentationsfunktion nutzen wir zum Speichern der natürlich-sprachlichen Bezeichner der Klassen und Methoden usw.

Die Ontologiekonzepte sind als disjunkte Klassen definiert. Ontologien können zwar auch überlappende Klassen darstellen, jedoch ergibt dies im objektorientierten Umfeld keinen Sinn. Klassen im Sinne der Ontologie sind *Class*, *Method* und so weiter; ein Element einer API kann aber nur eine Klasse oder eine Methode sein. Mehrfachvererbung kann ebenfalls nicht dazu führen, dass ein Individuum verschiedenen Klassen gleichzeitig angehört.

Tabelle 5.1: Die Struktur der NLCI-Ontologie.

Konzeptname	Beschreibung & Beispiel
Thing	Das allgemeinste Konzept der Ontologie.
Class	Eine instantiierbare Klasse.
Object	Eine *Class*, die andere *Components* enthalten kann; z.B. ein Mensch.
Component	Eine *Class*, das Teil eines *Objects* ist; z.B. der Arm eines Menschen.
Method	Ausführbare Methode.
PredefinedMethod	Von der Programmiersprache definierte Methode; z.B. `toString()`.
UserDefinedMethod	Von der API definierte Methode; z.B. `lift(arm: Class)`.
Parameter	Methodenparameter.
DataType	Datentypen der Programmiersprache.
Origin	Quelle eines *Things*; z.B. eine JAR-Datei.

(a) Die Klassen der NLCI-Ontologie

Relation	Quelle	Ziel
consistsOf	*Object*	*Container*
extends	*Class*	*Class*
fromOrigin	*Thing*	*Origin*
hasMethod	*Class*	*Method*
hasMethodParameter	*Method*	*Parameter*
ofType	*Parameter*	*DataType*
partOf	*Component*	*Object*

(b) Die Objekt-Relationen der NLCI-Ontologie

5.2.2 Befüllen der Ontologiestruktur

Die Domänenontologie kann von Hand mit den Elementen einer API befüllt werden (z.B. mit Protégé). Die linguistischen Analysen von NLCI erwarten, dass die Ontologie über sprechende Namen verfügt: Klassen müssen (zusammengesetzte) Substantive als Namen haben und Methoden Verben oder Verbalphrasen.

Um große APIs in die Ontologiedarstellung zu überführen, ist es ratsam, ein Hilfsprogramm zu implementieren. Ein Ontologie-Generator sollte alle verfügbaren Informationen in die Ontologie einfließen lassen. Wenn man beispielsweise Java-APIs modelliert, sollte man auf die Kommentare im JavaDoc zurückgreifen und die Quelltextkommentare ebenfalls auswerten. Wenn die API sprechende Namen verwendet, kann der Generator die Bezeichner der API übernehmen. Dies ist oft der Fall, da die Namenskonventionen unabhängig von unserem speziellen Einsatzzweck als sinnvoll erachtet und in der Ausbildung von Informatikern und Entwicklern gelehrt werden (siehe bspw. Referenzen [BD04], [Mar09] oder [Bal11]). Die Literatur empfiehlt, die Begriffe aus dem Problembereich auch beim Entwurf der Lösung zu verwenden, um so sprechende Namen zu erhalten (siehe bspw. Referenz [Abb83] oder Kapitel 4 und 5 in Referenz [Kri94]).

Ist die API selbst nicht sprechend benannt, können sprechende Namen in der Ontologie hinterlegt werden; so kann man APIs sprachlich erschließen ohne sie zu verändern. (Alternativ könnte man eine sprechend benannte Fassade für die anzusteuernde API entwerfen und implementieren.) In Kapitel 6 beschreiben wir den Ontologieaufbau für openHAB sowie für Alice. Der Ontologie-Generator für openHAB überführt eine Konfigurationsdatei für openHAB in die Ontologie und nimmt dabei schematische Ersetzungen vor, um eine sprechende Benennung in der Ontologie zu erreichen. Der Ontologie-Generator für Alice-Modelle übernimmt die Bezeichner unverändert aus der Alice-API.

5.2.3 Vorverarbeitung und Aufbereitung der API-Bestandteile

Wurde eine API von einem Generator aus einer API erzeugt, enthält sie zunächst nur Elemente mit ihren Bezeichnern sowie die Verweise auf die Quelldateien, in denen die Elemente gespeichert sind.

Um diese Bezeichner mit NLP-Werkzeugen verarbeiten zu können, müssen sie aus Wörtern der Eingabesprache bestehen. Ist die API sprechend benannt, so können die Bezeichner nach einer einfachen Vorverarbeitung automatisch zu sprachlichen Bezeichnern transformiert werden.

Wenn die Bezeichner der Konzepte aus mehreren Wörtern bestehen, dann müssen sie auf die Weitergabe an die NLP-Komponenten vorbereitet werden: Bezeichner in Pro-

Tabelle 5.2: Ergebnis-Beispiele für die einfache Wortauftrenn-Heurisik.

Eingabe	Ausgabe
Husky	Husky
SitStandWalk	Sit Stand Walk
body-front	body front
drive_right	drive right
footl	foot l
Spaceman2	Spaceman 2
uglyfish	uglyfish

grammiersprachen enthalten in der Regel keine Leerzeichen und zur Trennung der einzelnen Bestandteile eines Bezeichners werden Konventionen verwendet, bspw. die Binnenmajuskelschreibweise (engl. *CamelCase*). NLP-Werkzeuge erwarten hingegen einen Satz, eine Wortfolge oder ein einzelnes Wort als Eingabe. Damit die Bezeichner dieser Anforderung gerecht werden, werden sie von NLCI mehrstufig normalisiert: Zunächst werden sie aufgrund gängiger Konventionen (Unterstriche, Bindestriche, CamelCase, vor und nach Zahlen) getrennt. Bezeichner, die ausschließlich aus Kleinbuchstaben bestehen, werden hier noch nicht aufgetrennt. Tabelle 5.2 zeigt einige Bezeichner beispielhaft in ihrer ursprünglichen Schreibweise und nach dem Auftrennen. Der Bezeichner `uglyfish` wurde nicht aufgetrennt, obwohl er augenscheinlich aus mehreren Wörtern besteht.

Diese einfache Heuristik ist für unsere Zwecke meist ausreichend, kann jedoch verbessert werden. Besteht ein Bezeichner aus mehreren Wörtern und ist komplett klein bzw. groß geschrieben, so kann in der späteren Verarbeitung höchstwahrscheinlich keine Verknüpfung zwischen einem englischen Wort und dem Bezeichner hergestellt werden. Daher könnte man den Benutzer auffordern, für alle derartigen Bezeichner jeweils eine sprechende, konventionskonforme Variante anzugeben, damit die Verarbeitung besser funktioniert. Da der nächste Verarbeitungsschritt die Bezeichner mit Synonymen aus WordNet anreichern soll, wird stattdessen versucht, den Bezeichner buchstabenweise aufzutrennen und für die dadurch erhaltenen Kandidaten einen Eintrag in WordNet zu finden.

Zuerst werden gängige Suffixe (z.B. L und R für „left“ bzw. „right“) gesucht und diese abgespalten. Sind keine derartigen Suffixe vorhanden, wird buchstabenweise gespalten und die entstehenden Paare in WordNet gesucht. Finden sich beide Teile in WordNet, wird an der entsprechenden Stelle getrennt und der gespaltene Name gespeichert. Findet sich kein Paar, dessen Teile in WordNet gefunden werden können, begnügen wir uns

Tabelle 5.3: Mit der WordNet-Heuristik aufgetrennte Bezeichner.

Eingabe	Eigenheit	Ausgabe
wheelr	„l“ oder „r“ angehängt	wheel r
uglyfish	keine Trennzeichen	ugly fish
IEEERemainder	Einzelwort von hinten	IEEE Remainder
handletopr	Einzelwort von vorne	handle topr

mit einer Trennung, bei der wenigstens ein Teil gefunden werden kann. Einzelbuchstaben werden hierbei nicht betrachtet, da WordNet über Einträge für Buchstaben verfügt: z.B. Transliterationen bspw. für griechische Buchstaben (*nu*) ggf. sogar im Plural (*nus*) oder für Abkürzungen (*g* für Gramm). Da WordNet im nächsten Schritt zur Anreicherung der Ontologie mit Synonymen verwendet wird, garantiert das Vorgehen, dass bei einer erfolgten Auftrennung immer Synonyme zugeordnet werden können. Tabelle 5.3 zeigt Beispiele für Bezeichner, die mit der vorgestellten Heuristik aufgetrennt wurden.

Das auf WordNet basierte Verfahren teilt einen Bezeichner in zwei Teile – an sich wünschenswert wäre ein Verfahren, das Bezeichner ggf. an mehr als einer Stelle auftrennt. Ein weitaus ausgefeilterer Trenner für Bezeichner ist der von Enslen und Kollegen in Referenz [EHPVS09] beschriebene. Er erreicht insgesamt eine Genauigkeit von 97% – allerdings bei den Bezeichnern, die nicht mit Binnenmajuskeln geschrieben wurden, lediglich von 22%.

Technisches Detail

Die aufgetrennte Variante des Bezeichners ersetzt *nicht* den ursprünglichen Namen des Ontologie-Elements, sondern sie wird als Zusatzinformation gespeichert. Hierzu werden Kommentare zu den Ontologie-Individuen hinzugefügt.

OWL erlaubt es, zu beliebigen Elementen einen Kommentar hinzuzufügen. Dieser Kommentar kann mit einem Qualifizierer versehen werden, der eigentlich die Sprache des Kommentars kennzeichnet. Der Qualifizierer ist jedoch frei wählbar und so können beliebige textuelle Zusatzinformationen zu den Ontologie-Individuen hinzugefügt werden.

Durch diese Vorgehensweise wird die Ontologie nicht durch zusätzliche Individuen aufgebläht und alle Zusatzinformationen werden direkt beim betroffenen Individuum gespeichert. Zukünftig anfallende Zusatzinformationen können ebenfalls so

gespeichert werden, ohne die bestehenden Werkzeuge oder die Struktur der Ontologie anzupassen.

5.2.4 Anreicherung der API mit Synonymen

Wie zu Beginn dieses Abschnittes bereits erläutert wurde, enthält die NLCI-Ontologie natürlichsprachliche, sprechende Bezeichner für die Elemente der API. Ein Bezeichner pro API-Element ist jedoch für eine „natürliche" Anbindung nicht ausreichend, da die Benutzer nicht gezwungen werden sollen, sich an diese Bezeichnung zu halten. Diese Anforderung ergibt sich aus der Tatsache, dass wir täglich dazu angeleitet werden, gerade nicht immer denselben Begriff für etwas zu benutzen. Dies finden wir in der Schulausbildung, bei der Schülern ein „monotoner Schreibstil" abgewöhnt werden soll; ebenso werden Begriffswiederholungen in den Nachrichten[1] sehr oft vermieden. Um zu erreichen, dass sich Benutzer nicht an ein vorher festgelegtes, enges Vokabular halten müssen, verknüpfen wir die Ontologieelemente mit WordNet-Synsets [Wei12].

Die Verknüpfung der API mit WordNet wird einmalig vor der Benutzung der Ontologie durchgeführt und ist vollautomatisch.[2] Die Zusatzinformationen werden zu den Ontologieelementen gespeichert, ähnlich wie die aufgetrennten Bezeichner, und können später von allen (linguistischen) Analysen verwendet werden. Dadurch, dass die Verknüpfung als Vorbereitungsschritt durchgeführt wird, wird sie von der linguistischen Verarbeitung der Eingabetexte getrennt. Es ist auch möglich, manuell weitere Bezeichnungen in die Ontologie einzusetzen, falls die Verknüpfung mit WordNet alleine nicht ausreichend ist.

Möchte man Bezeichner einer API mit synonymen Begriffen anreichern, trifft man auf das Problem, dass Synonyme (nicht nur in WordNet) für einzelne Bedeutungen von Wörtern definiert sind. Bezeichner sind jedoch meist Wortgruppen und bestehen so aus mehreren Wörtern (z.B. `liftArm()` als Methodenname oder `BigWhoop` als Klassenname). Aus diesem Grund bestimmt NLCI den Kopf des Elementnamens und ermittelt dessen Synonyme.

In der Linguistik bestimmt man den Kopf einer Phrase meist mit grammatikalischen Regeln, z.B. verwendet man das Subjekt als Kopf und alle Attributierungen als Modifikatoren. Da Klassen- und Methodennamen keine Sätze sind, kann NLCI auf derartige

[1] Man denke zum Beispiel an die Berichterstattung über Michael Schuhmacher, den Kerpener, Wahlschweizer, Rekord-Formel-1-Weltmeister, ...

[2] Alternativ könnte man Techniken des Ontology-Matchings/-Mappings verwenden [Noy09]. Hierzu müsste man aus dem „Anfragetext" eine Ontologie erstellen und diese mit der NLCI-Ontologie verknüpfen. Bestehende Ansätze, wie bspw. in den Referenzen [GSY09, GSY04], ziehen hierbei auch Synonyme usw. in Betracht. Der Aufwand für die Abbildung einer Ontologie auf eine andere ist aber hoch und der Nutzen für NLCI ungewiss.

Tabelle 5.4: Beispiele für die Auflösung des Phrasenkopfes. Die Wortarten der Bezeichner-Bestandteile sind in Klammern angegeben.

Konzept	zugeh. Wortart	Beispiel-Individuum	Kopf
Class	Substantiv	ugly/JJ fish/NN	fish
Class	Substantiv	left/JJ upper/JJ arm/NN	arm
Class	Substantiv	peanut/NN butter/NN cookies/NNS	cookies
Method	Verb	open/VBZ fridge/NN	open
Method	Verb	turn/VBZ head/NN right/NN	turn

Heuristiken nicht zurückgreifen. Um eine einfache und möglichst effiziente Heuristik abzuleiten, verlässt sich die Vorverarbeitung darauf, dass die Namenskonventionen für objektorientierte Systeme befolgt wurden (bspw. die aus Referenz [BD04]): Sie legen fest, dass Klassennamen aus Adjektiven und Nomen bestehen und Methodennamen mit einem Verb beginnen. Die Vorverarbeitung ermittelt folglich die Wortarten der Wörter aus denen die Bezeichner bestehen und nimmt an, dass bei Klassen das (ggf. zusammengesetzte) Nomen und bei Methoden das Verb der Phrasenkopf ist; die übrigen Wörter werden als Modifikatoren betrachtet. Tabelle 5.4 zeigt einige Beispiele. Links in der Tabelle steht das Ontologie-Konzept, und die erwartete Wortart des Kopfes; dann folgen ein Beispiel-Individuum sowie der ermittelte Kopf.

Sind für einen Bezeichner mehrere Kandidaten der entsprechenden Wortart vorhanden, muss ausgewählt werden. Mehrere Substantive bilden in germanischen Sprachen zusammengesetzte Substantive (im Englischen *compound nouns*) so, dass der sinngebende Teil rechts steht (Rechtsköpfigkeitsprinzip, oder „right-hand head“-Regel [Pla03]). Daher sieht NLCI bei zusammengesetzten Substantiven (engl. *compound nouns*) das Wort ganz rechts als Kopf an. Gibt es kein Substantiv, betrachtet NCLI die Verben der Wortgruppe; gibt es laut Wortartmarkierung auch keine Verben, so bleibt NLCI nichts anderes übrig, als die Wortartmarkierung zu ignorieren; NLCI verwendet dann das erste Wort der Wortgruppe als Kopf. Besteht ein Bezeichner aus nur einem Wort, so wird es in jedem Fall verwendet, um Synonyme in WordNet zu ermitteln.

Wurde ein Wort als Kopf ausgewählt, wird in WordNet nach passenden Synonymen gesucht. Da WordNet Wortarten kennt, wird die Suche entsprechend des Elementtyps eingeschränkt, d.h. für Klassen wird nur nach Substantiven gesucht und für Methoden nur nach Verben. Da im Englischen oftmals Substantiv- und Verbform zusammenfallen und so bspw. für Substantive auch synonyme Verben ermittelt werden würden, wird die

Suche und damit die Ergebnismenge eingeschränkt. Wird mit der gewünschten Wortart kein Synset gefunden, so wird die jeweils andere Wortart zur erneuten Suche verwendet; wird auch hier kein Treffer erzielt, wird die Einschränkung aufgehoben und im gesamten WordNet gesucht. Sollte auch dann kein Treffer erzielt worden sein, wird der Kopf verworfen und ggf. mit einem anderen Bestandteil des Bezeichners die Suche erneut begonnen.

Die ermittelte Synonymmenge kann unter Umständen groß werden und falsch-positive Treffer enthalten. Das liegt daran, dass bis zu dieser Stelle lediglich eine lexikalische Suche erfolgt ist; WordNet-Synsets beziehen sich aber nicht auf ein Wort, sondern auf eine Wort-Bedeutung. Daher ermittelt die Synonymsuche alle WordNet-Synsets aller Bedeutungen eines Wortes und fügt somit ein semantisches Rauschen zur Ontologie hinzu. Um dieses Rauschen so gering wie möglich zu halten, wird die Synset-Menge vor dem Speichern ausgedünnt. Hierzu wird ausgenutzt, dass sowohl die NLCI-Ontologie als auch WordNet eine hierarchische Struktur haben: NLCI bestimmt für ein Ontologie-Individuum die übergeordneten Objekte und vergleicht diese mit den Holonym-/Meronym-Begriffen in WordNet. Können parallele Beziehungen in WordNet und der NLCI-Ontologie ermittelt werden, wird die betroffene Bedeutung als treffend angesehen und gespeichert; die übrigen, nicht treffenden Bedeutungen (und somit die zugehörigen WordNet-Synsets) werden nicht zur NLCI-Ontologie hinzugefügt. Bleibt die Holonym- oder die Meronym-Abfrage erfolglos, wird analog mit den Hyperonym bzw. Hyponymen verfahren.

Beispiel

In diesem Beispiel betrachten wir das Class-Individuum `leftArm`. In der Ontologie gehöre es zum Class-Individuum `body`. `body` habe zudem das folgende Synset:

- *(S_{body})* body, organic structure, physical structure: *the entire structure of an organism*

NLCI ermittelt im ersten Schritt `arm` als sinngebendes Wort und dazu die folgenden möglichen WordNet-Synsets:

- *(S_1)* arm: *a human limb*
- *(S_2)* arm, weapon, weapon system: *any instrument or intrumentality used in fighting or hunting*

Dann werden die zugehörigen Holonyme mit WordNet bestimmt. Für S_1 werden zwei Holonyme gefunden, für S_2 eins:

- *($H_1(S_1)$)* body, organic structure, physical structure: *the entire structure of an organism*
- *($H_2(S_1)$)* homo, man, human being, human: –

- $(H_1(S_2))$ weaponry, arms, implements of war, weapons system, munition: *weapons considered collectively*

Nun vergleicht NLCI diese Begriffe mit dem WordNet-Synset des übergeordneten Ontologie-Individuum `body` von `arm`. Da $H_1(S_1)$ und S_{body} übereinstimmen, wird S_1 als WordNet-Synset zu `arm` gespeichert und S_2 verworfen.

Technisches Detail

In der Ontologie wird keine Synonymliste gespeichert, sondern ein Verweis auf das ermittelte WordNet-Synset. Möchte eine Verarbeitungsstufe mit den ermittelten Synonymen arbeiten, muss eine Verbindung zu WordNet hergestellt werden. Nur mithilfe von WordNet lassen sich die dort verfügbaren Informationen (wie Ober- und Unterbegriffe) sinnvoll auswerten. Diese Informationen in die NLCI-Ontologie einzupflegen und die benötigten Algorithmen in NLCI zu implementieren ist nicht wünschenswert.

5.3 NLP-Analysen

Die einzelnen Analysen wurden als GoldenGATE-Einschübe entworfen und haben zur Ausführungszeit Zugriff auf den Text und alle bereits vorgenommenen Annotationen. Da die Analysen aufeinander aufbauen, müssen Sie in der Reihenfolge angewendet werden, in der sie im Folgenden beschrieben werden. Derzeit gibt es keine technischen Vorkehrungen, die eine richtige Reihenfolge sicherstellen; die Reihenfolge der Analysestufen ist fest in der Implementierung verankert. Die folgenden Unterkapitel beschreiben die einzelnen Analysen und gehen auf die Ergebnisse sowie auf die Voraussetzungen ein.

5.3.1 Verknüpfung von Text und API

Der erste Verarbeitungsschritt ermittelt einzelne Akteure, deren Aktionen und dazu ggf. Objekte; diese entsprechen Klassen der API, deren Methoden sowie Parametern. In dieser Analyse wird angenommen, dass die Subjekte von Aktiv-Sätzen die Akteure und die Prädikate die Aktionen sind [Wei14, LWT]. Die grammatikalischen, direkten und indirekten Objekte werden als Parameter interpretiert (z.B. „Somebody opens the door“ → `Somebody.open(door: Class)`).

Abbildung 5.1 zeigt den Ablauf dieses Verarbeitungsschritts im Detail; die einzelnen Schritte werden in den folgenden Unterabschnitten besprochen. Zunächst werden die Sätze in prädikatähnliche Ausdrücke überführt und anschließend deren Bestandteile auf die

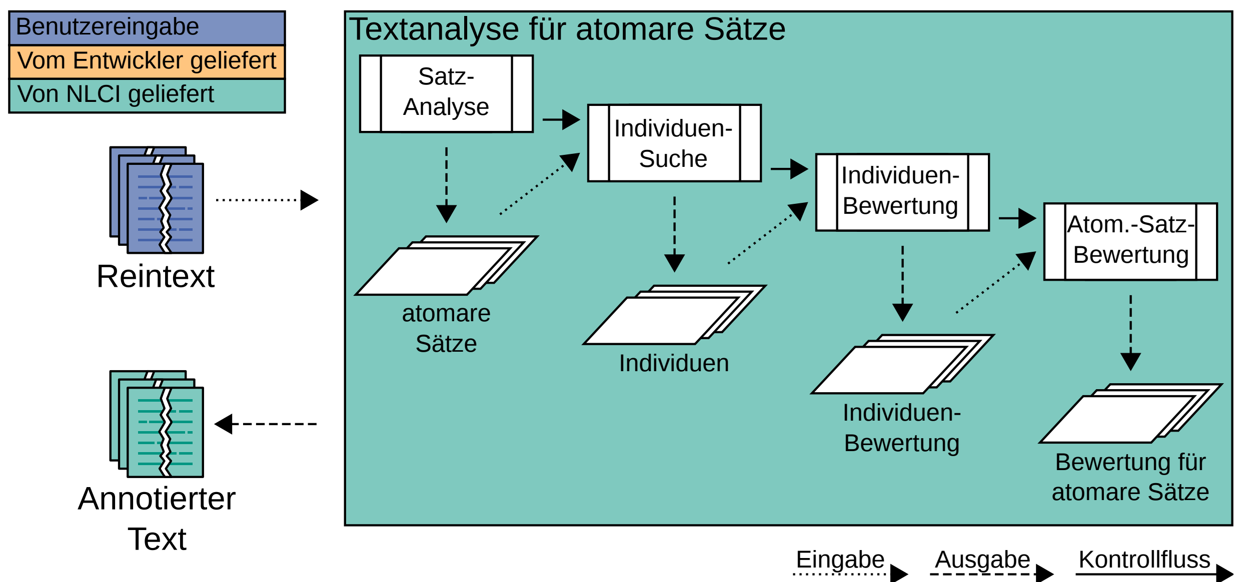


Abbildung 5.1: Schematischer Ablauf der API-Verknüpfungsstufe.

API abgebildet. Treten hierbei Mehrdeutigkeiten auf, werden alle mögliche Varianten ermittelt und bewertet; nachfolgende Schritte müssen dann auswählen, welche Interpretation gewählt werden soll.

5.3.1.1 Atomare Sätze

Um die Eingabe möglichst stark zu vereinfachen und nur die relevanten Informationen zu behalten, werden die Sätze zunächst in sogenannte atomare Sätze übersetzt. Ein atomarer Satz ähnelt einem logischen Prädikat und erfasst eine Aktion mit allen beteiligten Elementen. Ein atomarer Satz besteht aus den atomaren Konstituenten *actor*, *action* und *object*. Dargestellt werden – je nach Satzumfang – die Elemente als *action(actor)*, *action(actor, object)*, oder *action(object)*. Dadurch abstrahieren atomare Sätze von aktiver und passiver Formulierung. Ebenso abstrahieren sie von Konjunktionen und Aufzählungen, indem ein Satz in mehrere atomare Sätze umgewandelt wird.

Um einen Satz in einen oder mehrere atomare Sätze zu überführen, nutzen wir den Abhängigkeitsgraph des Eingabesatzes. Abbildung 5.2 zeigt einen Abhängigkeitsgraph für einen Beispielsatz. Die Umwandlung beginnt bei der Wurzel des Graphs `turns`. `turns` ist das erste Prädikat des Satzes und seine direkten Nachbarn `John` und `doorknob` sind Subjekt bzw. Objekt des Satzes. Sie sind mit Kanten des Typs `nsubj` bzw. `dobj` mit dem Prädikat verbunden. Diese drei Knoten bilden den ersten atomaren Satz *turns(John, doorknob)*. Der dritte Nachbarknoten von `turns` ist mit einer Konjunktion (`conj_and`-Kante) verbunden; dies zeigt an, dass er das nächste Prädikat ist und ein neuer atomarer Satz

Tabelle 5.5: Beispiele für atomare Sätze. Passivsätze, Konjunktionen und Aufzählungen werden aufgelöst.

Satz	# Atom. Sätze	Atom. Sätze
The janitor opens the door.	1	*open(janitor, door)*
The door is opened by the janitor.	1	*open(janitor, door)*
The janitor opens the door and the window.	2	*open(janitor, door)* *open(janitor, window)*
John opens the window and the janitor the door.	2	*open(John, window)* *open(janitor, door)*

begonnen werden muss. Das zweite Prädikat wird umgesetzt als *opens(John, door)* und erhält denselben Akteur wie der erste Satz: `John` ist auch das Subjekt des zweiten Satzteils, was man an den zwei eingehenden `nsubj`-Kanten erkennen kann.

Nach der Überführung werden die atomaren Konstituenten (und der atomare Satz) im Text annotiert, damit sie für weitere Analysen genutzt werden können. Diese Überführung wird für jedes Prädikat wiederholt; hierbei werden nicht nur Aktiv-Sätze behandelt, sondern auch Sätze

- im Passiv,
- mit Befehlen im Imperativ,
- mit direkter Rede,
- mit Modifikatoren für Subjekte und Objekte (z.B. Appositionen, Adjektive und Relativsätze)
- mit Gerundiv-Formen und
- mit sogenannten *full-infinitives*, die auch als *to-infinitives* bekannt sind.

Tabelle 5.5 zeigt einige Beispiele und deren Umsetzung in atomare Sätze.

Beispiel

Gegeben sei der Eingabesatz „John repairs the door, which is broken, while the table is cleaned by the old janitor.“ Er verwendet zugleich Aktiv, Passiv, einen Relativsatz und Modifikatoren.

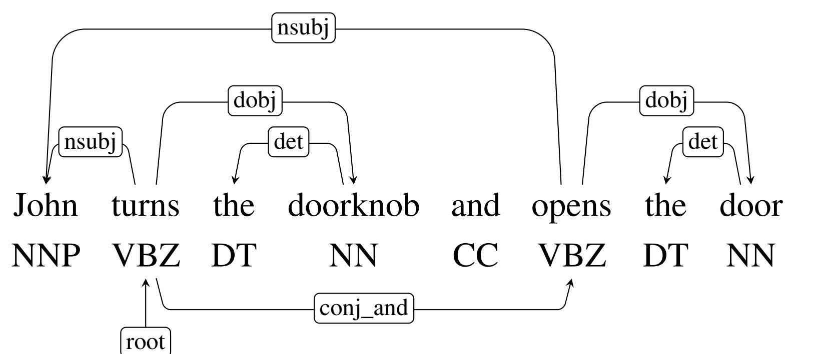


Abbildung 5.2: Abhängigkeitsgraph für den Satz „John turns the doorknob and opens the door.“ In der Zeile unter den Wörtern wurden die Wordarten notiert.

Durch die Behandlung der oben aufgeführten sprachlichen Variationen können auch Sätze in atomare Sätze überführt werden, die die Phänomene mischen. Für dieses Beispiel werden die folgenden beiden atomaren Sätze identifiziert (Phrasenköpfe sind mit ˆ gekennzeichnet):

- *repair(John, broken ˆdoor)*
- *clean(old ˆjanitor, table)*

5.3.1.2 Verknüpfung atomarer Konstituenten mit der API

Im nächsten Schritt werden die atomaren Konstituenten mit den Objekten und Methoden der API verknüpft: NLCI verknüpft Aktionen mit API-Methoden und Akteure sowie (syntaktische) Objekte mit Klassen der API. Die API-Elemente werden in der NLCI-Ontologie gesucht; findet NLCI einen Treffer, dann annotiert es eine Verknüpfung zum gefunden Ontologie-Individuum im Text. Bei der späteren Quelltexterzeugung sind dann alle nötigen Informationen ohne Weiteres erreichbar. Um die gesuchten API-Elemente in der Ontologie zu ermitteln, wird die lexikalische Nähe der Elemente aus dem Text zu den API-Elementen betrachtet. Dazu verwendet NLCI eine angepasste Technik aus der Informationsbeschaffung.

Im Allgemeinen ist die Verknüpfung für eine atomare Konstituente nicht eindeutig. Da NLCI in dieser Verarbeitungsstufe nicht auf Kontextinformationen zurückgreift, annotiert es eine atomare Konstituente gegebenenfalls mehrfach: Für jede mögliche API-Verknüpfung wird eine Annotation erzeugt. Zunächst werden so viele Anküpfungspunkte wie möglich identifiziert und bewertet; die Bewertung wird ebenfalls annotiert und trägt später zur Entscheidungsfindung bei. Alternativ könnte man die Kontextbetrachtung in

die Verknüpfungssuche integrieren, jedoch würde man dann unterschiedliche Analysen vermengen und dadurch die Wartbarkeit verringern.

Im ersten Schritt wird die Ontologie nach allen Wörtern durchsucht, die in der betrachteten atomaren Konstituente enthalten sind; hierbei werden der Kopf der atomaren Konstituente und ihre Modifikatoren betrachtet. Die Ontologie liefert als Antwort alle Individuen, die mindestens eins der angefragten Wörter enthält. Eine derartige Suche (im Gegensatz zu einer Suche, die ausschließlich die Köpfe betrachtet) ist nötig, da wir nicht davon ausgehen können, dass Benutzer die richtigen Begriffe verwenden. Die Modifikatoren sind manchmal der einzige Weg, das gewünschte API-Element zu ermitteln. Um die Ausbeute weiter zu erhöhen, greift die Suche auch auf die ermittelten Synonyme der API-Elemente und die Stammformen der Anfragewörter zu. Um ein möglichst präzises Suchergebnis zu erhalten, schränkt NLCI die Suche hierbei entsprechend des Konstituententyps ein: Suchen nach Aktionen liefern dementsprechend Methoden und Suchen nach Akteuren oder Objekten liefern Klassen. Trotz dieser Einschränkung können die Suchen anhängig von Suchanfrage und API-Umfang eine große Anzahl an Kandidaten liefern, die gemäß Gleichung 5.1 sortiert werden. Sie weist einem Kandidaten r für eine Anfrage q eine Bewertung $B(r,q)$ zwischen 0 und 1 zu.

$$B(r,q) = R(r,q) \times C(r,q) \times s \times u^n - o \tag{5.1}$$

NLCI verwendet diese Bewertungsfunktion anstelle eines anderen Ähnlichkeitsmaßes, um die Charakteristika von APIs und der Ontologie auszunutzen.

Der erste Faktor von $B(r,q)$ ist die Repräsentativität $R(r,q)$; sie beschreibt, wie gut der Name des Kandidaten zur Anfrage passt. R entspricht dem Verhältnis aus der Anzahl an überlappenden Zeichen im Kandiatennamen zur Zeichenfolge der Anfrage:

$$R(r,q) = \left(\frac{c_{\text{found}}(r)}{c_{\text{total}}(q)}\right) \tag{5.2}$$

Der zweite Faktor von $B(r,q)$ ist die Abdeckung $C(r,q)$; sie beschreibt, wie gut die Anfrage den Namen des Kandidaten abdeckt. Sie ist definiert als Quotient aus gefundenen Zeichen der Anfrage und allen Zeichen im Namen des Kandidaten:

$$C(r,q) = \left(\frac{c_{\text{used}}(q)}{c_{\text{total}}(r)}\right) \tag{5.3}$$

Zusammen beschreiben Repräsentativität und Abdeckung, wie gut die Anfrage (atomare Konstituente) und Suchergebnis (Individuum) zusammenpassen. NLCI erlaubt auch teilweise Übereinstimmungen in den Namen um Schwächen der API-Namen auszugleichen. Ein Vergleich nach Elementen würde z.B. für die Anfrage [old, janitor] und einen

besten API-Treffer `oldJan` nur das Wort „old“ als Treffer zählen; das beschriebene Verfahren betrachtet dabei auch den überlappenden Anfang von „Jan“ und „janitor“.

Die übrigen Terme von Gleichung 5.1 nehmen API-Charakteristika in die Bewertung mit auf. Die Werte der Faktoren s, u und o sind Konfigurationsparameter, deren Werte während der Entwicklung manuell ermittelt wurden. Dazu wurde die Bewertung von Suchergebnissen mithilfe von Trainingsanfragen aus dem NLCI-Korpus kalibriert. Die nachfolgend angegebenen Werte haben sich im Allgemeinen als gut erwiesen; soll NLCI in der Praxis eingesetzt werden, sollten für den konkreten Einsatzzweck optimale Werte anhand eines Benchmarks mit einem Optimierungsverfahren bestimmt werden (z.B. einer Suche mit dem Hillclimbing-Verfahren Nelder-Mead).

Da Synonyme zusätzliches Rauschen zu den Suchergebnissen hinzufügen, verwenden wir die Konstante s als Strafterm wenn wir einen Treffer nur aufgrund des Synonymeintrags gefunden haben. Bei deinem Treffer im Kandidatennamen setzen wir s auf 1; bei Treffern mit Synonymen wird s auf eine Konstante zwischen 0 und 1 gesetzt, um die Bewertung zu verringern. In der Standardkonfiguration setzen wir s auf $0,4$.

Der Faktor u^n bewertet die Container/Komponenten-Beziehung in der Ontologie. u ist ein Diskontierungsfaktor, der ebenfalls auf $0,4$ gesetzt ist, um die Bewertung von Elementen zu verringern, die im Text ohne ihre Elternelemente genannt wurden; n ist die Anzahl der fehlenden Komponenten in der Anfrage. Der Beispielsatz „The janitor raises his left arm“ wird von NLCI zum atomaren Satz *raise(ˆjanitor arm left)* umgewandelt. Der beste Kandidat für [janitor, arm, left] könnte das Individuum `OldJanitor` $\rightarrow$ `UpperBody` $\rightarrow$ `LeftArm` sein. Da die Anfrage nur das erste und letzte Element der Hierarchie (be-) trifft, werden nur diese für die Berechnung der Bewertung herangezogen, was zunächst zu einem besseren Wert führt. Dann wird u^n bestimmt und die Bewertung entsprechend durch den Diskontierungsfaktor verringert, da die innere Komponente `UpperBody` fehlt; in unserem Beispiel um $u^n = 0,4^1 = 0,4$.

Die Unterscheidung der Ontologiestruktur zwischen eigenständigen Klassen und Komponenten können wir ebenfalls in der Bewertung einfließen lassen. Wir bevorzugen eigenständige Klassen gegenüber Komponenten und setzen daher den Faktor o für Klassen auf 0 und auf $0,2$ für Komponenten.[3]

Nachdem alle Kandidaten bewertet wurden, annotiert NLCI alle Treffer im Text, deren Bewertung $B(r,q)$ einen konfigurierbaren Schwellwert überschreitet. Kleinere Werte als $0,1$ sind als Schwellwert nicht sinnvoll, da sonst sehr viele irrelevante Treffer im Text annotiert werden.

[3] Unsere Erfahrungen in der Evaluation stützen die Annahme, dass Benutzer Klassen bevorzugt verwenden und Komponenten seltener benutzt werden.

Tabelle 5.6: Beispiel-Ontologie (Auszug). Alle gezeigten Methoden sind mit den Klassen `Caretaker` und `OldJanitor` verbunden. Synonyme sind in eckigen Klammern angegeben.

Konzept	Individuum
Objekt	`Caretaker` [janitor, groundkeeper]
Objekt	`OldJanitor`
Objekt	`OldTimer`
Objekt	`Table`
Komponente	`OldJan`→`ArmL`
Benutzerdefinierte Methode	`clean(Class)`
Benutzerdefinierte Methode	`turnTo(Class)`
Benutzerdefinierte Methode	`turn()`
...	...

Im nächsten Schritt werden die gefundenen Kandidaten zu atomaren Sätzen zusammengefasst und für diese eine kombinierte Bewertung S bestimmt. Für jeden atomaren Satz bestimmt NLCI die (API-seitig zugelassenen) Kombinationen aus Akteur, Aktion und Objekt. Für jeden Akteur bestimmt NLCI die Schnittmenge aus den für ihn verfügbaren Methoden in der Ontologie und den ermittelten Methodenkandidaten und erzeugt so Tupel *(class, method)*. Dann ermittelt NLCI alle Objektkandidaten, die für die jeweilige Methode als Argument geeignet sind. Die Bewertung S des atomaren Satzes $\mathfrak{A}$ und den jeweiligen Kandidaten ist $\mathfrak{R} = \{r_{actor}, r_{action}, r_{\text{object}}\}$ und wird aus den Bewertungen $B(r,q)$ der Kandidaten wie folgt berechnet:

$$S(\mathfrak{A},\mathfrak{R}) = \frac{\sum B(r,q)}{|\mathfrak{A}|} \tag{5.4}$$

Wenn keine Objekte als Argument für die Methode ermittelt werden konnten, dann ist $B_{\text{object}} = 0$. Wenn eine Methode kein Argument benötigt, dann ist $|\mathfrak{A}| = 2$ andernfalls ist $|\mathfrak{A} = 3|$. Nach diesem Schritt werden alle gültigen Kombinationen (also alle möglichen interpretationen des atomaren Satzes bei einer gegebenen NLCI-Ontologie) zusammen mit den Bewertungen im Text annotiert.

Beispiel

In diesem Abschnitt wird der oben beschriebene Such- und Bewertungsalgorithmus anhand eines Beispiels erläutert. Wir verwenden hierzu die Beispiel-Ontologie in Tabelle 5.6 und betrachten die dort gezeigten vier Objekte, eine Komponente und drei Methoden. Wir gehen davon aus, dass alle drei Methoden mit den Klassen „Caretaker“ und „OldJanitor“ verbunden sind. „Caretaker“ hat die Synonyme „janitor“ und „groundkeeper“. In der Ontologie sind noch weitere Elemente verzeichnet, wie z.B. das Objekt `OldJan`, dessen linker Arm in Tabelle 5.6 zu sehen ist. Dieses Beispiel beschäftigt sich jedoch nur mit dem gezeigten Auszug aus der Ontologie. (Verarbeitet NLCI einen Eingabetext, wird immer die gesamte Ontologie betrachtet.)

Bestimmen des atomaren Satzes

Betrachten wir nun den Eingabesatz „The old janitor cleans the table.“ Als erstes leitet NLCI den atomaren Satz $\mathfrak{A}$ =*clean(old ˆjanitor, table)* ab; der Kopf der Konstituente ist mit einem ˆ gekennzeichnet. Dann ermittelt es die Ontologieindividuen für die atomaren Konstituenten.

Um Objekt-Individuen für den Akteur zu ermitteln, stellt NLCI an die Ontologie die separaten Suchanfragen für *janitor* und *old*. Die Ontologie liefert `OldJanitor` und `OldTimer` als Antwort auf die Anfrage *old*. `OldJan`→`ArmL` ist auch Teil des Suchergebnisses, da der Anfang der Hierarchiekette zur Suchanfrage passt. Als letztes Suchergebnis wird `Caretaker` geliefert, da *janitor* ein Synonym von `Caretaker` ist.

Bewerten der Kandidaten

Als nächstes muss NLCI die Kandidaten entsprechend Gleichung 5.1 bewerten. In Tabelle 5.7 sind die einzelnen Bestandteile der Kandidatenbewertung detailliert aufgeführt. In der ersten Spalte steht das ermittelte Ontologie-Individuum, in der zweiten das Synonym, sofern eines für die Suche verwendet wurde. Die folgenden Spalten enthalten die Komponenten der Gleichung 5.1; in der letzten Spalte steht das Bewertungsergebnis.

Die Länge der Suchanfrage *old ˆjanitor* ist 10, weswegen die Anzahl der übereinstimmenden Zeichen in R durch 10 geteilt wird. Die Repräsentativität von `Caretaker` wird für sein Synonym „janitor“ bestimmt, da es für die Ermittlung in der Ontologie verwendet wurde. Aus demselben Grund ist $s = 0,4$ für `Caretaker` und 1 für die anderen Individuen.

C hängt von der Länge der Namen der Individuen ab. Da nur die erste Hierarchiestufe von `OldJan→ArmL` gefunden wurde, wird die Länge von `OldJan` verwendet um C zu berechnen. Außerdem ist $u^n = 0,4^1 = 0,4$ und da der Arm nur eine Komponente ist, wird $o = 0,2$ gesetzt. Zuletzt ist $u^n = 1$ und $o = 0$ für die ersten drei Kandidaten, da es sich bei allen um eigenständige Klassen handelt.

Für unsere Beispielanfrage ist `OldJanitor` der erwartete Treffer. `Caretaker` hat nur eine geringe Bewertung, da Synonyme als wertvolle aber rauschbehaftete Teilergebnisse betrachtet werden. Das Individuum `OldJan→ArmL` erhält eine Bewertung unterhalb unserer Schwelle von $0,1$ und wird daher aus der Ergebnisliste getilgt.

Bewerten des atomaren Satzes

NLCI ermittelt `cleanUp(class)` für *clean* sowie `Table` für *table* mit Bewertungen von $0,714$ bzw. $1,0$. Die Bewertung der Kombination für den atomaren Satz $\mathfrak{A}$ und die drei atomaren Konstituenten wird folgendermaßen berechnet:

$$S(\mathfrak{A}, \{\texttt{OldJanitor,cleanUp(class),Table}\}) = \frac{1,0 + 0,714 + 1,0}{3} = 0,905$$

S für den atomaren Satz mit `Caretakter` ist $0,665$ und für den atomaren Satz mit `OldTimer` ist $S = 0,608$; alle drei Kombinationen werden zusammen mit den jeweiligen Bewertungen im Text annotiert.

Hätte NLCI kein Objekt gefunden, wäre der atomare Satz $\mathfrak{A}'$ =*clean(Janitor)*; dann würde die Bewertung der Kombination niedriger ausfallen, da `cleanUp(class)` ein Argument erwartet:

$$C(\mathfrak{A}', \{\texttt{OldJanitor, cleanUp(class)}\}) = \frac{1,0 + 0,714}{3} = 0,571$$

Dasselbe gilt, wenn eine Methode ein Argument erwartet, aber keines mit dem erwarteten Typ gefunden werden kann oder wenn die Methode kein Objekt erwartet, im atomaren Satz aber eines angegeben ist.

5.3.1.3 Ermitteln von Methodenparametern

Ausgehend von den atomaren Sätzen müssen nun die Methodenparameter im Text bestimmt werden. Zwar enthalten atomare Sätze bereits Methodenparameter, diese können jedoch nur korrekt sein, wenn es sich dabei um Objekte handelt, die Teil der API sind. Methodenparameter können jedoch auch Zeichenketten, Zahlen und Werte aus Aufzäh-

Tabelle 5.7: Die ermittelten Ontologie-Individuen r und ihre Bewertung B für die Anfrage *old ˆjanitor* und die Ontologie aus Tabelle 5.6.

r	Syn_{ret}	R	C	s	u^n	o	B
`OldJanitor`	-	$10/10$	$10/10$	1	1	0	1
`Caretaker`	janitor	$7/10$	$7/7$	$0,4$	1	0	$0,28$
`OldTimer`	-	$3/10$	$3/8$	1	1	0	$0,11$
`OldJan`→`ArmL`	-	$6/10$	$6/6$	1	$0,4$	$0,2$	$0,04$

lungen sein. Oft sind die konkreten Werte nicht so verwendbar, wie sie im Text genannt sind, sondern müssen in den jeweiligen Typ umgewandelt werden.

Um die Argumente im Text zu bestimmen, führt NLCI eine weitere syntaktische Analyse durch [Sek14]: Ausgehend von den Aktionen der atomaren Sätze werden die Verbalphrasen des Syntaxbaumes auf Methodenparameter untersucht.

Im Englischen können Prädikate bzw. Verben mit drei verschiedene Arten von Objekten verknüpft werden: direkten, indirekten und präpositionalen Objekten. Direkte Objekte werden von der beschriebenen Aktion bearbeitet; ein indirektes Objekt ist ein Empfänger der Aktion; die Rolle des präpositionalen Objekts hängt von der Präposition ab. Abhängig vom Valenzrahmen des Verbs können die Objekte obligatorisch oder optional sein. Zudem gibt es „freie“ Objekte, wie Temporaladverben, die z.B. die Dauer oder Häufigkeit einer Aktion angeben. Alle diese Objektarten sind Argumentkandidaten für die zu generierenden Methoden, insbesondere können ganze Phrasen die Rollen eines Objekts einnehmen und nicht nur einzelne Wörter.

Beispiel

Gegeben sei der Eingabesatz „Sophie walks five meters.“

Im Syntaxbaum wird dieser Satz auf der obersten Ebene als Aussagesatz dargestellt, bestehend aus einer Nominalphrase („Sophie“) und einer Verbalphrase („walks five meters“). Im vorigen Analyseschritt wurde der atomare Satz *walk(Sophie, five ˆmeters)* aufgebaut und walk mit einer Methode `walk(int : meters)` verknüpft – allerdings ohne ein passendes Argument, da zu *five meters* keine Ontologieindividuen ermittelt werden können. Nun muss eine Ganzzahl als Argument für den Parameter `meter` identifiziert werden. Dazu verknüpft NLCI die Aktion *walk* mit der Verbalphrase im Syntaxbaum. In der Verbalphrase findet sich jedoch keine Ganzzahl: „five meters“ muss entsprechend bearbeitet werden.

Tabelle 5.8: Ableitungen, die für die Argumentermittlung verwendet werden (Auszug).

Kategorie	Syntaxregel	Beispiel
Richtungswert	ADVP → RB	Sophie turns right.
Zahl	NP → CD NNS	Sophie walks five meters.
Zeichenkette	–	Sophie says „Hello World!“.
	NP → NN	Nach der Vorverarbeitung: Sophie says something.
Klasse	NP → NP PP	Sophie takes an apple from the table.

NLCI enthält eine Syntaxregel, die „five“ als Argument vom Typ Zahl identifiziert. Die Regel NP → CD NNS besagt, dass eine Nominalphrase aus einer Kardinalzahl (Wortart CD) und einem Nomen im Plural (Wortart NNS) besteht. Das mit CD markierte Wort wird als Argument verwendet und vorher in die Zahl 5 umgewandelt. Das Nomen „meters“ wird als möglicher Parametername zwischengespeichert.

Unter der Annahme, dass es keine Klassen gibt, die dieselben Namen haben wie primitive Typen, wird in den Verbalphrasen nacheinander nach den folgenden Argumenttypen gesucht: Zunächst werden Richtungsangaben identifiziert, danach numerische Werte, dann Zeichenketten; als letztes werden Klassen berücksichtigt, was der Object-Annotation der atomaren Sätze entspricht. Für jede dieser Kategorie gibt es Syntaxregeln, nach denen die Verbalphrase untersucht wird; Tabelle 5.8 zeigt einen Auszug aus dem Regelkatalog, eine vollständige Liste der Regeln ist in Anhang E angegeben.

Richtungsangaben NLCI lässt Richtungsangaben in einem eng begrenzten Wertebereich zu: `LEFT`, `RIGHT`, `UP`, `DOWN`, `FORWARD` sowie `BACKWARD`.

Richtungsangebende Wörter werden von NLCI bei der Verarbeitung in diese kanonischen Werte übersetzt. Abhängig von ihrer Stellung im Satz und dessen Struktur erhalten die Wörter verschiedene Wortartmarkierungen, die von NLCI berücksichtigt werden müssen. Bei der Erkennung werden zusätzlich die adverbialen Synonyme aus WordNet berücksichtigt.

Numerische Werte Numerische Werte können als Zahlen (markiert mit der Wortart CD) oder als Adverbialphrase (z.B. „twice“, markiert mit der Wortart RB) in Eingabetexten vorkommen. Häufig werden Formulierungen benutzt, die eine Zahl und etwas Gezähltes enthalten, z.B. „two apples“ oder „two meters“; somit entsteht häufig die in der

Tabelle angegebene Ableitung NP $\rightarrow$ CD NNS. Das Substantiv, das mit NNS gekennzeichnet wird, ist zudem ein Kandidat für einen Parameternamen, ebenso wie etwaige Angaben von Einheiten.

Die ermittelten Zahlwörter werden anschließend mit einem Werkzeug aus CoreNLP in Zahlen übersetzt; CoreNLP wandelt sowohl Ziffern und Zahlen, als auch Ausdrücke wie „quarter“ und „dozen“ in Ziffernfolgen um.

Zeichenketten Zeichenketten kommen in den untersuchten Texten ausschließlich in wörtlicher Rede vor und werden daher von Anführungszeichen umschlossen. Diese Kennzeichnung von Zeichenketten (auch wenn sie nur aus einem Wort bestehen) kann zur Erkennung verwendet werden.

Wenn Zeichenketten bei der Vorverarbeitung nicht explizit behandelt werden, werden sie vom Zerteiler ebenso zerteilt, wie die anderen Sätze der Eingabe. Dann lässt sich eine Zeichenkette nur noch anhand der Anführungszeichen erkennen. Zeichenketten, die nicht gesondert gekennzeichnet sind, werden von NLCI nicht betrachtet.

Sollen Zeichenketten mit derselben Mechanik erkannt werden, wie die übrigen Methodenargumente, muss der Eingabetext entsprechend vorbereitet werden: Die in Anführungszeichen stehenden Zeichenketten müssen durch einen Platzhalter (z.B. „something“) ersetzt werden, damit im Syntaxbaum nur ein Blatt für die Zeichenkette entsteht. Dieses Blatt kann entsprechend der Regel in Tabelle 5.8 erkannt werden; alternativ kann der Platzhalter erkannt und so direkt eine Zeichenkette als Argument identifiziert werden. Als alternative Vorbereitung könnte man die Zeichenkette aufgrund der Anführungszeichen im Syntaxbaum erkennen und die Knoten zusammenfassen.

In der aktuellen Implementierung belässt NLCI den Eingabetext und den Syntaxbaum unverändert und nutzt die Anführungszeichen zur Erkennung von Zeichenketten als Methodenargumente.

Objekte Zuletzt werden Objekte als Argumente gesucht. Die Suche entspricht an dieser Stelle der Objektsuche bei den atomaren Sätzen im vorangegangenen Abschnitt.

Eine Abweichung von der Objektsuche bei den atomaren Sätzen ergibt sich, wenn Richtungsangaben als Modifikatoren verwendet werden, z.B. bei „lift the right arm“. Die Objektsuche ermittelt für derartige Ausdrücke gezielt die Teilobjekte, in diesem Fall den rechten Arm, und vermerkt diese als mögliche Argumente. Da der benötigte Methodenparameter von der API abhängt (bspw. könnte es eine Methode `liftArm(side: Direction)` oder `lift(arm: Object)` geben), notiert NLCI zusätzlich die Seitenangabe „right“ als mögliches Argument.

Die oben erklärten Analysen für die verschiedenen Parametertypen werden zunächst auf der mit der Aktion verknüpften Verbalphrase durchgeführt. Dabei wird die Ableitungskette der Verbalphrase im Syntaxbaum mit den Parametertyp-spezifischen Ableitungsregeln verglichen. Gibt es für den aktuellen Teilbaum keine passende Ableitungsregel, so wird die Analyse für alle direkten Kinder der Teilbaumwurzel erneut durchgeführt. So werden größere Phrasen so weit vereinfacht (bzw. Teilphrasen ignoriert), bis entweder eine Analyse einen Argumentkandidaten liefert oder keine Kinder mehr vorhanden sind.

Zuletzt werden die verfügbaren Argumente mit den möglichen Methodenaufrufen (d.h. dem *action*-Teil des atomaren Satzes) verknüpft. Hierzu wird zunächst der Typ des Parameters mit dem Typ des Argument-Kandidaten verglichen. Benötigt eine Methode zwei Parameter, die denselben Typ haben, werden die vorher notierten Kandidaten für Parameternamen ausgewertet, um eine Zuweisung der Argumente zu ermöglichen.

Kann ein Parameter nicht mit einem Argument belegt werden, so ist die Methode nicht aufrufbar und wird entsprechend entfernt. Parameterlose Methoden sind immer aufrufbar.

Beispiel

Gegeben sei die API-Methode `give(object: Class, to: Class)` mit zwei Parametern desselben Typs. Außerdem sei ein Eingabesatz mit der Verbalphrase „gives the orange juice to Sophie“ zur Analyse eingegeben.

Wenn die Objektsuche für die beiden Nominalphrasen „orange juice“ und „Sophie“ die typgleichen Objekte `orangeJuice` und `Sophie` liefert, bleibt zunächst unklar, welches Argument für welchen Parameter gewählt werden soll.

Bei der Objektsuche wurde für das Argument „Sophie“ der mögliche Parametername „to“ aus der Präposition bestimmt. Daraus lässt sich ableiten, dass `Sophie` als zweites Argument der Methode verwendet werden soll. Da danach nur noch ein Parameter ungebunden ist, kann die Auflösung vollständig vorgenommen werden. Es wird somit der Methodenaufruf `give(orangeJuice, Sophie)` ermittelt.

5.3.1.4 Auswahl der auszuführenden Methoden

NLCI ermittelt in den oben beschriebenen Schritten unter Umständen mehrere verschiedene Kandidaten für Akteure und Methoden zu einem atomaren Satz. Im letzten Schritt wird daher ausgewählt, welche Akteure und welche Methoden verwendet werden sollen. Die Auswahl nimmt dazu an, dass sich synonyme Begriffe in einem Eingabetext immer auf dasselbe Objekt beziehen. Handelt der Eingabetext von zwei Entitäten, die ein gemeinsames Synonym haben und verwendet der Eingabetext das Synonym in ver-

schiedenen Sätzen für *beide* Entitäten, so ist nicht definiert, auf welche Entität NLCI das Synonym bezieht.

Zunächst wird für jeden möglichen Akteur bestimmt, wie viele ausführbare Methodenkandidaten er im gesamten Eingabetext besitzt. Gewählt wird der beste Akteur, d.h. der Akteur, der die meisten Methoden ausführen kann. Gibt es für einen atomaren Satz mehrere beste Akteure, werden die Bewertungen der Methoden herangezogen, die bei der Verknüpfung mit der API bestimmt wurden. Wird ein Akteur-Kandidat auf diese Weise eliminiert (d.h. nicht ausgewählt), dann kann er bei den Methodenaufrufen nicht mehr als Argument verwendet werden. Aus diesem Grund muss nach der Auswahl eines Akteurs (bzw. nach dem Ausschließen der übrigen) für alle betroffenen Kandidaten für Methodenaufrufe geprüft werden, ob der eliminierte Akteur-Kandidat durch einen anderen ersetzt werden kann (bspw. den nächstbesten aus der Objektsuche).

Nach der Auswahl der Methodenaufrufe werden die Ergebnisse im Text annotiert. Notiert werden der Akteur, die Methode und ggf. die Argumente mit zugehörigem Typ und Parameternamen.

5.3.2 Korrektur der Reihenfolge

Nachdem die Aktionen und Methodenargumente zugeordnet wurden, könnte man – rein inhaltlich – den entsprechenden Quelltext erzeugen. Jedoch beschreiben Menschen, wie oben erwähnt, selten in korrekter chronologischer Reihenfolge. Überträgt man die Aktionen in der Reihenfolge in den Quelltext, in der sie im Text genannt werden, erhält man unter Umständen nicht die gewünschte Aktionsfolge.

Weichen Autoren von einer chronologischen Beschreibung ab, verwenden Sie oft Temporalausdrücke wie „before“, „after“ usw. So zeigen Sie an, wie die genannten Aktionen miteinander zusammenhängen. Man kann auch sagen: Temporalausdrücke steuern die Chronologie eines Textes.

NLCI reiht die Aktionen auf einer Zeitachse auf, bevor der Quelltext generiert wird [LHT14]. Geht man hierbei naiv vor, so werden die Aktionen in der Reihenfolge auf der Zeitachse aufgetragen, in der sie im Text genannt wurden. Möchte man die tatsächlich beschriebene Aktionssequenz erhalten, so muss man die Temporalausdrücke berücksichtigen und die Aktionen auf der Zeitachse entsprechend einordnen bzw. umsortieren. Die folgenden Abschnitte beschreiben, was Temporalausdrücke sind, wie man sie und die zu ihnen gehörenden Aktionen ermitteln kann und wie letztlich die gewünschte Aktionssequenz hergestellt werden kann.

Tabelle 5.9: Signalwörter für Temporalausdrücke

Temporaladverben	Temporale Präpositionen
after, afterward(s), afterwhile, antecendently, before, beforehand, concurrently, eventually, finally, first (of all), firstly, henceforth, henceforward, hereafter, hereupon, initially, last, later (on), latterly, precedently, simultaneously, subsequently, synchronously, then, thenceforth, thenceforward, thereafter, thereon, thereupon, ultimately, when, whereupon, while	after, as, at, before, by, in, prior to, subsequent

5.3.2.1 Temporalausdrücke

Temporalausdrücke (engl. *temporal expressions*) sind Ausdrücke oder Konstruktionen der Sprache, die zeitliche Zusammenhänge beschreiben. Die englische Sprache kennt drei verschiedene Typen von Temporalausdrücken: Tempus (die Zeitform der Verben), Temporaladverben und temporale Präpositionen. Für die Bestimmung der chronologisch korrekten Aktionssequenz betrachtet NLCI Muster von Verben und Temporalausdrücken im Text.

Die Tempora, in denen die Verben eines Textes stehen, reichen alleine nicht aus, um eine Reihenfolge zu bestimmen. Sie erzeugen, für sich alleine genommen, keine strikte zeitliche Reihenfolge zwischen allen Aktionen. Wenn die Zeit wechselt, in der die Verben stehen, können zeitliche Relationen aufgebaut werden. Allerdings sind diese Relationen selten ausreichend um eine vollständige Sortierung zu erzeugen. Zudem werden Ablaufbeschreibungen (auch im NLCI-Korpus) überwiegend im Präsens formuliert. Daher verwendet NLCI keine Muster basierend auf den Tempora der Verben.

Die anderen beiden Typen eignen sich besser zur Herstellung der zeitlichen Reihenfolge: Temporaladverben (z.B. *afterwards*, *yesterday* und *now*) sowie temporale Präpositionen (z.B. *ago* und *before*) drücken zeitliche Abhängigkeiten zwischen Aktionen aus. Die Muster, die NLCI betrachtet, um die Temporalausdrücke im Text zu ermitteln, verwenden Signalwörter. Sie sind in Tabelle 5.9 aufgeführt und sind Englischwörterbüchern entnommen. Wörter wie *now*, *yesterday* oder *ago* wurden ausgelassen. Sie sind ebenfalls Temporaladverbien oder temporale Präpositionen und fallen damit eigentlich auch in die Kategorie der Temporalausdrücke. Sie sind aber für die zeitliche Sortierung von Ablaufbeschreibungen nicht relevant bzw. hilfreich.

Temporalausdrücke, die auf Gleichzeitigkeit hindeuten, werden hier nicht betrachtet. Eine Auflösung von parallelen Aktionen erfolgt mithilfe von Kontrollstrukturen (siehe hierzu Abschnitt 5.3.3).

5.3.2.2 Muster mit Temporalausdrücken

Die Signalwörter aus Tabelle 5.9 werden dazu verwendet, sprachliche Muster zu formulieren, aus denen wiederum zeitliche Abhängigkeiten geschlossen werden können. Um die richtige Reihenfolge auf der Zeitachse herzustellen, verwenden wir Operatoren, welche die Zeitachse entsprechend der jeweiligen Muster anpassen. Um die zeitlichen Abhängigkeiten zu erfassen, genügen drei Operatoren: $after(a,b)$, $before(a,b)$ und $at(n,a)$. Die ersten beiden Operatoren haben Aktionen als Parameter; der erste Parameter ist der *Anker*, der zweite die *verschobene Aktion*. $after(a,b)$ setzt Aktion b direkt hinter Aktion a; analog setzt $before(a,b)$ Aktion b direkt vor Aktion a. α kennzeichnet als leere Aktion den Anfang der Aktionsfolge, ω entsprechend das Ende. Der dritte Operator, $at(n,a)$, verschiebt die Aktion a an die (numerische) Stelle n auf die Zeitachse; alle folgenden Aktionen wandern entsprechend auf der Zeitachse nach rechts. NLCI betrachtet über 20 verschiedene Muster (zuzüglich ähnliche Muster, die durch Synonyme entstehen); einen Auszug der Muster und Operatoren findet sich in Tabelle 5.10.

Beispiel

$after(a,b)$ ist nicht dasselbe, wie $before(b,a)$. Operatoren bewegen die verschobene Aktion auf der Zeitachse, während der Anker fest an seiner Stelle verbleibt.

Betrachtet man die Zeitachse $[\alpha,a,b,c,\omega]$, so erzeugt die Ausführung des Operators $after(a,c)$ die Zeitachse $[\alpha,a,c,b,\omega]$, während $before(c,a)$ das Ergebnis $[\alpha,b,a,c,\omega]$ erzeugt.

Der Temporalausdruck bestimmt den Operator, seine Position im Satz bestimmt, welche Aktion der Anker ist und welche die zu verschiebende Aktion. Betrachtet man „before a, do b“, dann ist a der Anker und b die zu verschiebende Aktion: $before(a,b)$. Steht der Temporalausdruck zwischen den Aktionen, dann ist der Effekt umgekehrt: „Do a before b“ wird als $before(b,a)$ interpretiert. Befindet sich ein Temporalausdruck in einer Subphrase, so wird die Analyse auf der Subphrase durchgeführt. Ein Temporalausdruck muss sich nicht auf eine konkrete Aktion beziehen, sondern es kann auch eine Referenz angegeben werden. Häufig wird z.B. mit „before *that*“ die Aktion im direkt voranstehenden Satz als Anker verwendet.

Tabelle 5.10: Temporale Muster und Operatoren.

Temporales Muster	Operator
At the end, do a.	$before(\omega, a)$
In the beginning, do a.	$after(\alpha, a)$
As n^{th} do a.	$at(n, a)$
After a, do b.	$after(a, b)$
Before a, do b.	$before(a, b)$
Do a. *Before that*, do b.	$before(a, b)$
Before a *started*, do b.	$before(a, b)$
Do a. *Then* do b.	$after(a, b)$
When a, do b.	$at(a, b)$

...

Beispiel

Da die Analyse der Temporalausdrücke Satz für Satz durchgeführt wird, werden die Beschreibungen nicht als Bedingungen aufgefasst: Wird eine Aktion b in einem Satz ans Ende verschoben und danach beschrieben, dass nach b eine andere Aktion c stattfindet, dann ist c die letzte Aktion auf der Zeitachse und nicht b:

Gegeben sei die folgende Aktionsbeschreibung: „At the end, do b. After that, do c. Do a." Daraus ergibt sich die Zeitachse in der textuellen Reihenfolge $[\alpha, b, c, a, \omega]$. Aus den ersten beiden Sätzen werden die Operatoren $before(\omega, b)$ und $after(b, c)$ gewonnen und nacheinander auf die Zeitachse angewandt. So ergibt sich zuerst das Zwischenergebnis $[\alpha, c, a, b, \omega]$ und dann das Endergebnis $[\alpha, a, b, c, \omega]$.

5.3.2.3 Mustersuche und Korrektur der Reihenfolge

Das Vorgehen wird in Abbildung 5.3 skizziert. Auch diese Verarbeitungsstufe verwendet die Wortartmarkierungen, den Syntaxbaum und den Abhängigkeitsgraphen. Zuerst wird eine naive Zeitachse initialisiert, indem alle Aktionen in der textuellen Reihenfolge auf der Zeitachse aufgetragen werden. Dann wird der Text Satz für Satz betrachtet, um Temporalausdrücke zu identifizieren und die für sie ermittelten Operatoren anzuwenden. NLCI muss dazu nach Schlüsselphrasen suchen, die die temporalen Muster identifizieren und prüfen, ob das Muster in einer Subphrase liegt. Dann werden Anker und zu verschiebende Aktion identifiziert und am Ende die Zeitachse entsprechend angepasst. Die folgenden Absätze betrachten die einzelnen Schritte der Verarbeitung detailliert.

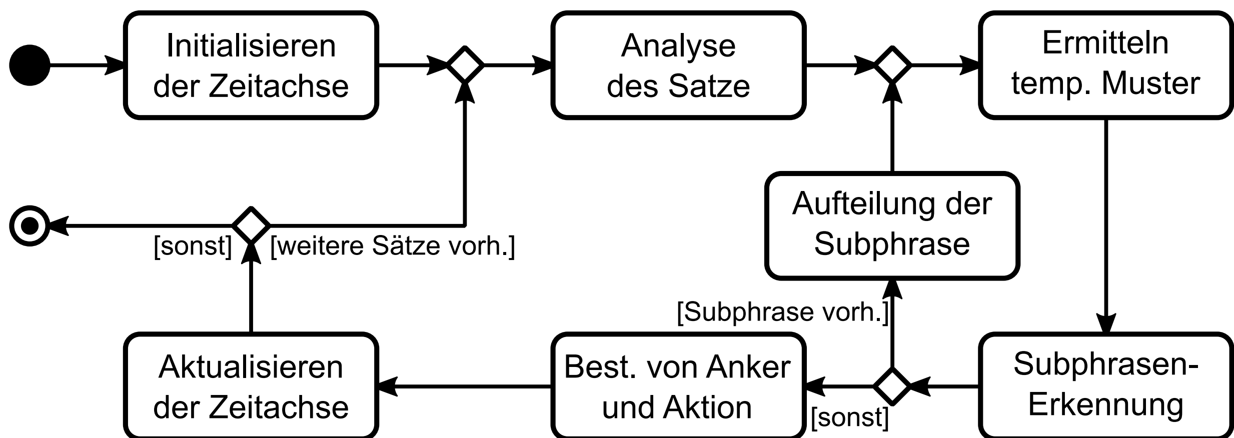


Abbildung 5.3: Schematischer Überblick über die Zeitachsenanalyse.

Mustererkennung NLCI sucht entsprechend der Temporalausdrücke nach Schlüsselphrasen im Satz. Dann wird die Position des Temporalausdrucks bezüglich der Aktionen im Satz bestimmt: Wie oben beschrieben, ist es wichtig, an welcher Stelle der Temporalausdruck steht, vor zwei Aktionen oder zwischen zwei Aktionen: zum Beispiel „*before a*, do *b*“ und „do *a before b*“. Die Positionierung des Temporalausdrucks beeinflusst, welche der Aktionen als Anker verwendet und welche Aktion positioniert werden muss.

Abspalten von Subphrasen NLCI verwendet den Syntaxbaum des Satzes, um zu bestimmen, ob der Temporalausdruck in einer untergeordneten Phase liegt. Ist dies der Fall können Teile des Satzes von der Analyse ausgenommen werden: Der Satz wird an der Phrasengrenze aufgespalten und die entstehenden Teile separat betrachtet. Enthält ein Teil keinen Temporalausdruck, so wird er ignoriert; enthalten beide Teile einen Temporalausdruck, werden sie getrennt voneinander analysiert.

Bestimmung des Ankers und der zu verschiebenden Aktion Der Anker des Operators kann auf drei verschiedene Arten angegeben werden:

1. Als absolute Position mit Ausdrücken wie „at the end“ oder relativ mit Formulierungen wie „before *that*“.
2. Der Anker kann eine Aktion im Satz sein, wie *b* in „Do *a* after *b*“.
3. Der Anker kann eine Aktion in einem vorangegangenen Satz sein, wie *a* in „Do *a*. … After *a*, do *b*“.

The cheerleader cheers (a). The penguin flaps (b) its wings twice. The cheerleader turns (c) to face the penguin. The penguin turns (d) its head right. The penguin flaps (e) its wings once. The penguin glides (f) away. Then the cheerleader says (g): “Where are you going?”.

(a) Chronologische Beschreibung.

Before the cheerleader turns (c) to face the penguin, the penguin flaps (b) its wings twice. But at the beginning of the scene, the cheerleader cheers (a). After the cheerleader has turned (→c) to face the penguin, the penguin turns (d) its head right. At the end, the cheerleader says (g): “Where are you going?”. But before that, the penguin flaps (e) its wings once and then glides (f) away.

(b) Nicht-chronologische Beschreibung.

Abbildung 5.4: Gegenüberstellung von chronologischer und nicht-chronologischer Beschreibung.

Die ersten beiden Fälle können gänzlich mit den Informationen abgedeckt werden, die mit der Mustersuche ermittelt wurden. Der dritte Fall ist komplizierter; hier muss zuerst die referenzierte Aktion ermittelt werden. Die konkrete Formulierung einer solchen Referenz kann sehr unterschiedlich ausfallen und es gibt keine computerlinguistsiche Bibliothek, die eine Art Korreferenzanalyse für Aktionen durchführt. NLCI betrachtet daher nur die Referenzierungen, die die Aktion wortwörtlich wiederholen. Eine derartige Referenz findet sich im dritten Satz des Beispiels in Abbildung 5.4: „After the cheerleader has turned to face the penguin...“ In diesem Satz ist „turn“ eine Wiederholung der Aktion im vorangegangenen Satz (in der Abbildung ist dies mit ($\rightarrow c$) gekennzeichnet).

Anpassen der Zeitachse Die Operatoren sind für jedes Muster erfasst; entsprechend beschreibt Tabelle 5.10, wo die zu verschiebende Aktion positioniert werden soll. Wird in einem Satz kein Temporalausdruck erkannt, so wird keine Veränderung an der Zeitachse vorgenommen.

Sind am Ende alle Sätze in dieser Art abgearbeitet, so steht auf der Zeitachse die erkannte chronologische Reihenfolge der Aktionen im Text. Die Aktionen werden entsprechend der Zeitachse mit einer Zeit-Annotation versehen und die Korrektur der Reihenfolge ist abgeschlossen.

Beispiel

Dieses Beispiel verdeutlicht das Verfahren anhand der Texte in Abbildung 5.4. Sie beschreiben dieselbe Aktionsfolge; der erste Text geht hierbei streng chronologisch vor, wohingegen der zweite Text dieselben Aktionen nicht-chronologisch beschreibt. Beim ersten Text sind keine Umordnungen nötig, da die gewünschte Reihenfolge der Aktionen mit der textuellen Reihenfolge übereinstimmen. Es ergibt sich – bereits mit der naiven Herangehensweise – eine korrekte Zeitachse wie in Abbildung 5.5a.

Vorbereitung und Aktionen a, ..., d

NLCI betrachtet zuerst die textuelle Reihenfolge der Aktionen und erzeugt so aus dem zweiten Text die initiale Zeitachse $[\alpha, c, b, a, d, g, e, f, \omega]$. So entsteht eine Zeitachse, die alle Aktionen der Beschreibung enthält, deren Reihenfolge jedoch noch korrigiert werden muss. Anschließend wird der Text Satz für Satz analysiert. Begonnen wird mit dem ersten Satz; für das Beispiel nehmen wir aber an, dass die ersten drei Sätze bereits vollständig analysiert wurden und daher die Aktionen $a, \dots, d$ korrekt auf der Zeitachse positioniert wurden: $[\alpha, a, b, c, d, g, e, f, \omega]$.

Satz 4, Aktion (g)

Die Analyse untersucht dann den Satz „At the end, the cheerleader says (g): ‚Where are you going?'". NLCI sucht zunächst nach den temporalen Mustern und identifiziert „at the end", was der einzige Temporalausdruck in diesem Satz ist. Außerdem ermittelt NLCI, dass dieser Temporalausdruck am Satzanfang steht und der Satz nur die Aktion *says (g)* enthält.

Als nächstes prüft NLCI, ob der Temporalausdruck in einer untergeordneten Phrase liegt (was hier nicht der Fall ist). Dann werden Anker und zu verschiebende Aktion bestimmt: Der Anker des Ausdrucks „at the end" ist ω, da es sich immer auf die letzte Aktion auf der Zeitachse bezieht. Da *says (g)* die einzige Aktion im Satz ist, wird es als zu verschiebende Aktion ausgewählt. Aus diesen Informationen leitet NLCI den Operator $before(\omega, g)$ ab und wendet ihn auf die Zeitachse an.

Als Ergebnis erhalten wir die Zeitachse $[\alpha, a, b, c, d, e, f, g, \omega]$.

Satz 5, Aktionen (e) und (f)

Als nächstes wird der Satz „But before that, the penguin flaps (e) its wings once and then glides (f) away" betrachtet. Wieder ermittelt NLCI die temporalen Muster und

identifiziert zwei unterschiedliche Temporalausdrücke: „before that“ und „then“. Der Ausdruck „before that“ steht am Anfang des Hauptsatzes und der Ausdruck „then“ steht am Anfang der untergeordneten Phrase. NLCI teilt den Satz in zwei Teile an der Konjunktion („and“) auf und verarbeitet die beiden getrennt voneinander.

Der erste Teil „But before that, the penguin flaps its wings once“ enthält nur die Aktion *flaps (e)*; NLCI wählt sie als die zu verschiebende Aktion aus, die von „before that“ verschoben wird. „Before that“ wird von NLCI immer auf die direkt vorangegangene Aktion bezogen; entsprechend wählt NLCI die Aktion *says (g)* im vierten Satz als Anker. Als Operator wird somit auf $before(g,e)$ geschlossen. Der zweite Teil des Satzes, „then glides away“, enthält die Aktion *glides (f)*; da es keine Alternativen gibt, ist dies wieder die zu verschiebende Aktion des gesuchten Operators. „Then“ bezieht sich ebenso wie „before that“ auf die direkt vorangegangene Aktion; in diesem Fall bezieht es sich auf die Aktion *flaps (e)* des ersten Satzteils. „Then“ fügt die zu verschiebende Aktion direkt hinter dem Anker ein: $after(e, f)$.

Zusammengefasst entsteht durch die Verschiebungen die zeitliche Abfolge $[e, f, g]$. Diese Abfolge steht genau so bereits auf der Zeitachse, weswegen keine Anpassungen mehr nötig sind.

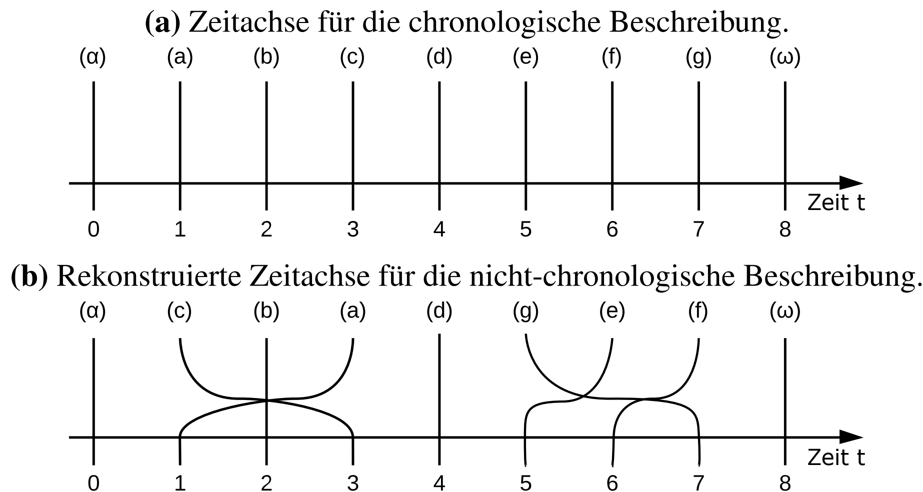
Ergebnis

Nachdem der letzte Satz verarbeitet wurde, erhält man die Zeitachse, wie er in Abbildung 5.5b dargestellt ist; er zählt die Aktionen in derselben Reihenfolge auf, wie die Zeitachse des chronologisch geschriebenen Beispieltextes.

5.3.3 Ermitteln von Kontrollstrukturen

Ebenso wie bei der Korrektur der Reihenfolge ermittelt NLCI auch die Kontrollstrukturen Satz für Satz. Die Analyse der Kontrollstrukturen ist jedoch umfangreicher und nutzt außer Signalwörtern auch die Abhängigkeitsgraphen der betrachteten Sätze und die Wortarten der enthaltenen Wörter [LH15].

Die Analysen, um die Kontrollstrukturen zu erkennen, sind zwar spezifisch für die jeweilige Kontrollstruktur, jedoch ist die Vorgehensweise für alle gleich: Der Abhängigkeitsgraph wird ausgehend von Schlüsselphrasen nach Aktionen und deren (zeitlichen) Beziehungen durchsucht. Eine Konfiguration steuert, welche Kanten im Graph besucht werden, welche Knoten betrachtet werden und wann die Suche abgebrochen wird. Diese Konfiguration bezeichnen wir als Profil der Kontrollstruktur (*trace profile*). Nachdem alle Sätze verarbeitet wurden, sorgt ein Nachbearbeitungsschritt dafür, dass die Kontrollstruk-



(a) Zeitachse für die chronologische Beschreibung.

(b) Rekonstruierte Zeitachse für die nicht-chronologische Beschreibung.

Abbildung 5.5: Gegenüberstellung der Zeitachsen für die chronologische und nicht-chronologische Beschreibungen aus Abbildung 5.4.

Tabelle 5.11: Ein (gekürztes) Beispiel für das Profil für gleichzeitige Aktionen (*do together*).

Schlüsselphrasen	at the same time
Abhängigkeiten	conj, prep
Wortarten	VBG, VBZ
Stoppwörter	times

turen korrekt geschachtelt werden und unnötige Kontrollstrukturen entfernt werden. So werden bspw. sequentielle Blöcke entfernt, die nur eine Anweisung erhalten; die Anweisung selbst bleibt hiervon unberührt. Am Ende werden die Ergebnisse im Text annotiert.

Ein Profil enthält die Abhängigkeitstypen, die betrachtet werden sollen, eine Liste an Wortarten, die zur Aktionsidentifikation genutzt werden, sowie Schlüsselphrasen, die die Kontrollstruktur kennzeichnen. NLCI sieht in der Standardkonfiguration Verben als Aktion an. Die Liste von Stoppwörtern enthält alle Wörter, die nicht als Aktionen betrachtet werden sollen, ungeachtet ihrer Wortart. Die Stoppwortliste umfasst in der Regel die Objekte einer Domäne (bzw. die Klassennamen einer API) und ist demnach in allen Profilen gleich. Tabelle 5.11 zeigt eine verkürzte Fassung des Profils für die Erkennung gleichzeitig ablaufender Aktionen.

Setzt man einen Wortartmarkierer oder Zerteiler ein, der keine perfekten Ergebnisse liefert, so erwächst aus dieser Konfiguration eine Tuningmöglichkeit: Je nachdem, wie viele (oder wenige) Abhängigkeiten, Schlüsselphrasen, und Stoppwörter man in der Kon-

figuration erfasst, kann man einige (aber nicht alle) Fehler des Markierers oder Zerteilers ausgleichen. Das kann insbesondere dann hilfreich sein, wenn der Wortartmarkierer häufig Verben als Nomen markiert oder umgekehrt. Treten in einer Domäne Fehler auf, so fallen Sie häufig auf dieselben domänenspezifischen Wörter (wenn bspw. die Domäne Wörter oder Formulierungen verwendet, auf denen der Markierer nicht trainiert wurde, oder mit denen er nicht umgehen kann). Die Evaluation der Kontrollstrukturerkennung betrachtet in Abschnitt 6.2.6 aus diesen Überlegungen heraus verschiedene Szenarien: Zunächst wird eine Standardkonfiguration verwendet, die linguistischen Überlegungen folgt. Eine weitere Konfiguration behandelt einige häufig auftretende Zerteilungsfehler. Erwartungsgemäß funktioniert die angepasste Konfiguration besser als die Standardkonfiguration, wenn es Zerteilungsfehler gibt. Umgekehrt sind die Ergebnisse bei einer perfekten Eingabe besser, wenn man die Standardkonfiguration verwendet.

Der nachfolgende Unterabschnitt erklärt, wie der Abhängigkeitsgraph durchlaufen wird und verdeutlicht das Vorgehen anhand eines ausführlichen Beispiels. Der zweite Unterabschnitt beschreibt die Kontrollstrukturen, die NLCI zur Zeit erkennen kann.

5.3.3.1 Kontrollstruktur-unabhängige Verarbeitung des Abhängigkeitsgraphen

NLCI verwendet den Abhängigkeitsgraph (siehe Abschnitt 2.1), um die Aktionen zu ermitteln, die zu einer Kontrollstruktur gehören. Das Durchlaufen des Graphen funktioniert für alle Kontrollstrukturen gleich: NLCI beginnt an der Schlüsselphrase und führt eine eingeschränkte Tiefensuche durch, um die zugehörigen Aktionen zu ermitteln. Kanten werden bei der Tiefensuche nur ein mal besucht; Kanten werden nur besucht, wenn sie im Profil der gerade betrachteten Kontrollstruktur stehen. Die Richtung der Kanten wird bei der Suche ignoriert.

Ebenso wie bei der Korrektur der Reihenfolge werden Aktionen anhand der Wortart erkannt. Die Standardkonfiguration listet daher alle Wortartmarkierungen für Verben auf, um Aktionen unabhängig von der konkreten Verbform (z.B. Tempus) zu erkennen. Ermittelt NLCI auf diese weise eine Aktion, wird die Aktion gespeichert und die Suche ausgehend von ihrem Knoten neu gestartet. Die betrachteten Kanten variieren abhängig von der betrachteten Kontrollstruktur. Daher fasst die Konfiguration für die Kontrollstrukturerkennung mehrere Profile zusammen: Eine für jede Kontrollstruktur. Ähnliche Kontrollstrukturen wie *do together* und *do in order* teilen sich ein Profil; der genaue Typ (also *together* vs. *in order*) wird in einem späteren Schritt bestimmt.

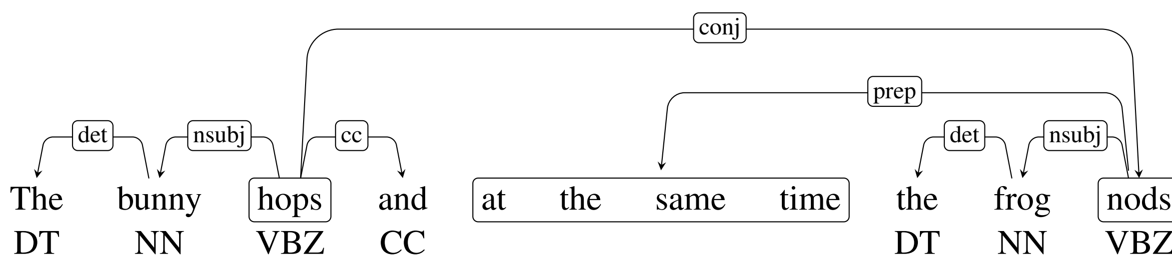


Abbildung 5.6: Ein Abhängigkeitsgraph für den Satz „The bunny hops and at the same time the frog nods.“ Die Wortarten sind unterhalb der Wörter angegeben.

Beispiel

Dieses Beispiel zeigt, wie NLCI Gleichzeitigkeit (d.h. die Kontrollstruktur *do together*) erkennt und die betroffenen Aktionen ermittelt. Das Beispiel verwendet das verkürzte Profil aus Tabelle 5.11. Abbildung 5.6 zeigt den Abhängigkeitsgraph, der betrachtet wird.

Die Graphverarbeitung beginnt bei der Schlüsselphrase, die im Profil definiert ist. In Abbildung 5.6 ist das „at the same time“. Ausgehend von diesem Knoten betrachtet NLCI alle Kanten und prüft, ob ihr Typ im Profil angegeben ist. Die einzige Kante ausgehend von der Schlüsselphrase hat den Typ `prep` und dieser ist im Profil vermerkt. Am anderen Ende der Kante befindet sich der Knoten „nods“ mit der Wortart VBZ; VBZ kennzeichnet ein Verb in der dritten Person Singular im Präsens und wird als Aktion angesehen. Ausgehend von „nods“ werden die noch nicht besuchten Kanten betrachtet; da `prep` bereits besucht wurde und `nsubj` nicht im Profil verzeichnet ist, ist `conj` die einzige unbesuchte Kante. Am anderen Ende der Kante befindet sich der Knoten „hops“, der ebenfalls mit der Wortart VBZ gekennzeichnet ist und daher als Aktion angesehen wird. Da „hops“ keine Kanten besitzt, die noch nicht besucht wurden und im Profil stehen, wird zu „nods“ zurückgesprungen. Danach gibt es keine erreichbaren, unbesuchten Kanten mehr; daher bricht die Suche an dieser Stelle ab. Folglich wurden zwei Aktionen als zur Kontrollstruktur zugehörig ermittelt; im Dokument wird annotiert, dass diese beiden Aktionen zeitgleich auftreten.

5.3.3.2 Betrachtete Kontrollstrukturen

NLCI unterstützt die gängigen Kontrollstrukturen und fügt diesen einfache Strukturen für Parallelität hinzu: *do together* umschließt einen Block von Aktionen, die parallel ausgeführt werden. *for all together* führt einen Aktionsblock für oder mit allen Objekten aus

einer Objektmenge parallel aus. Wie die anderen Kontrollstrukturen umfassen sie eine oder mehrere Aktionen; wie bei der Programmierung üblich, lassen sich die Strukturen verschachteln. Die Auswahl der betrachteten Kontrollstrukturen orientiert sich an den Kontrollstrukturen, die Alice anbietet. Zum einen ist der Funktionsumfang der Kontrollstrukturerkennung damit ausreichend für die Fallstudie. Zum anderen handelt es sich bei Alice um eine Lernumgebung für objektorientierte Programmierung und so kann man davon ausgehen, dass die angebotenen Kontrollstrukturen einerseits den gängigen entsprechen und andererseits ausreichend für Programmiereinsteiger sind. Die folgenden Absätze beschreiben die untersuchten Kontrollstrukturen und erklären, wie NLCI die Kontrollstrukturen und ihren Inhalt erkennt.

do together* und *do in order Kontrollstrukturen mit *do* beschreiben die Aktionsfolge innerhalb eines einzelnen Satzes. Man kann zwischen *do together* und *do in order* unterscheiden; die erste drückt aus, dass es sich um zeitgleich, die zweite, dass es sich um strikt sequentiell ablaufende Aktion handelt. Um hiervon betroffene Aktionen zu ermitteln, sucht NLCI nach Aktionen, die mit Wörtern wie „simultaneously“ oder Ausdrücken wie „first … then“ verbunden sind.

while Die Kontrollstruktur *while* drückt aus, dass die Aktionen innerhalb des Rumpfes von *while* wiederholt werden sollen, bis eine Abbruchbedingung erreicht wird. NLCI ermittelt hierzu zunächst die Aktionen, die wiederholt werden sollen und anschließend die Abbruchbedingung.

for all in order* und *for all together Die Aktionen, die zur Kontrollstruktur *for all* gehören, werden auf oder mit einer Menge von Objekten ausgeführt. Hierzu muss NLCI nicht nur die Aktionen ermitteln, sondern auch die Menge der Objekte. Naheliegend ist es, eine Aufzählung des Benutzers auszuwerten und die Aktionen auf bzw. mit allen aufgezählten Objekten auszuführen. Benutzer neigen allerdings dazu, mit (benannten) Mengen zu hantieren, und verlangen vom System, dass es diese Mengen selbstständig auflöst.

NLCI verwendet die Ontologie, um Gruppen von Objekten zu erkennen (z.B. werden bei „all animals“ alle Objekte im Text gesucht, die der Oberklasse „animal“ angehören). Das funktioniert, weil NLCI in diesem Schritt auf die Informationen zugreifen kann, die in Form der atomaren Sätze vorliegt; in der vorbereitenden Analyse wurde dann eine Aktion erkannt, jedoch ohne konkreten Akteur.

Gibt der Benutzer eine Gruppe und eine Ausnahme an, dann werden die genannten Objekte aus der Menge entfernt (z.B. „all animals but not the rabbit“). Erkennt NLCI, dass eine Kontrollstruktur mit *for all* anlegen soll, jedoch keine Objektmenge, dann wird der Benutzer gefragt, welche der verfügbaren Objekte in die Gruppe aufgenommen werden sollen.

Je nach Formulierung wird die Aktionsfolge entweder vollständig nacheinander auf den einzelnen Objekten ausgeführt (das entspricht der Variante *for all in order*), oder auf allen Objekten gleichzeitig (das entspricht der Variante *for all together*).

loop Die letzte – und vermutlich einfachste – unterstützte Kontrollstruktur ist die Schleife (*loop*). Eine Schleife führt eine Aktionsfolge mehrfach aus, wobei die Anzahl der Wiederholungen fest angegeben ist. Um Schleifen zu erkennen, sucht NLCI im Satz nach Kardinalzahlen, die mit „times“ verbunden sind (z.B. „five times“); außerdem berücksichtigt NLCI Adverbien wie „twice“. Anschließend wird die Aktion bestimmt, die im Abhängigkeitsgraphen mit der wiederholenden Phrase verbunden ist. Aktionsfolgen werden durch Aufzählungen und durch Begriffe wie „simultaneously“ gebildet.

if/else Ein Verzweigen des Kontrollflusses, wie es beim Programmieren mit *if/else* erreicht wird, wird derzeit von NLCI nicht unterstützt. Bedingte Anweisungen waren im Anwendungsgebiet von NLCI noch nicht notwendig, können jedoch ergänzt werden. Im Syntaxbaum des Eingabesatzes können Bedingungen dadurch erkannt werden, dass sie in untergeordneten Phrasen (Konditionalsätzen) stehen, die mit Schlüsselwörtern wie „if“, „when“ und so weiter beginnen. Die bedingte Aktion ist mit dem Verb der Bedingung im Abhängigkeitsgraphen mit einer *advcl* Kante verbunden (diese Kante kennzeichnet einen Adverbialsatz). Die Bedingung selbst ist mittels einer *mark*-Kante mit dem Schlüsselwort verbunden.[4] Eine Erweiterung der Kontrollstrukturenerkennung ist somit verhältnismäßig einfach zu erreichen.

5.3.4 Weitere domänenspezifische Analysen & Zusammenfassung

Die hier vorgestellte Auswahl an Textanalysen erhebt keinen Anspruch auf Vollständigkeit. Die Evaluation von NLCI hat jedoch gezeigt, dass für Texte der angestrebten Komplexität keine weiteren allgemeinen Analysen nötig sind. Je nach Domäne und Anwendungsszenario ist es jedoch denkbar (sogar: wünschenswert), dass weitere Analysen speziell für den jeweiligen Anwendungsfall hinzugenommen werden: Wenn die Domäne die

[4] Eine entsprechende Analyse beschreiben Mihalcea et al. in Referenz [MLL06] sowie Körner und Landhäußer in Referenz [KL10].

Sprache oder den Funktionsumfang einschränkt, dann sollte diese Einschränkung ausgenutzt werden. Die Architektur von NLCI ist so gestaltet, dass dies einfach möglich ist und weitere Analysen in den Prozess eingegliedert werden können.

Die Fallstudien im folgenden Kapitel beleuchten, wie gut NLCI in verschiedenen Szenarien eingesetzt werden kann. Dabei werden keine domänenspezifischen Analysen eingesetzt, um die Leistungsfähigkeit vergleichbar zu betrachten. Der Aufwand zur Vorbereitung von NLCI auf einen Einsatzzweck beschränkt sich daher auf die Modellierung der jeweiligen API in der NLCI-Ontologie und das Entwickeln eines Code-Erzeugers.

Nach Abschluss der oben vorgestellten Analysen ist das Eingabedokument hinreichend weit angereichert, sodass es der nächsten Verarbeitungsstufe zugeführt werden kann: Der Programmsynthese.

5.4 Programmsynthese

Die Programmsynthese kann nun alle gesammelten Informationen nutzen, um das Ergebnisskript zu erzeugen. Der Code-Erzeuger ist hierbei nicht abhängig von der gewählten Domäne/API, sondern nur von der eingesetzten Programmiersprache: Die Information, welche Klassen und Methoden konkret aufgerufen werden sollen, liefert der annotierte Text in Verbindung mit der NLCI-Ontologie unabhängig von der gewählten Domäne. In der Ontologie steht auch, wo die Quelldateien der API zu finden sind (z.B. in welcher JAR-Datei sich eine Java-Klasse befindet). Entwickler müssen somit für jede Programmiersprache nur einen Code-Erzeuger bereitstellen. Damit der Erzeuger nicht auf die Eigenheiten der NLP-Annotationen eingehen muss, wird der annotierte Text in eine Art abstrakten Syntaxbaum überführt und dieser dem Erzeuger zur Verfügung gestellt.

Der Code-Erzeuger kann auch auf Eigenheiten der eingesetzten Programmiersprache eingehen oder Optimierungen enthalten: Ob Schleifen ausgerollt werden oder nicht, kann der Erzeuger bestimmen; existiert in der Zielsprache das Konzept der Schleife nicht, kann der Erzeuger die ausgerollte Variante in das Ergebnisskript schreiben. Von derartigen Optimierungen oder Eigenheiten ist die Textanalyse vollständig entkoppelt – genau so wie die Code-Erzeugung von linguistischen Eigenheiten nichts wissen muss.

Für die Fallstudie mit openHAB wurde kein Code-Erzeuger entworfen, da man für ein derartiges System keinen Quelltext erzeugen möchte, den der Benutzer dann noch in openHAB laden muss. Vielmehr möchte man ein interaktives System direkt ansprechen, und die einzelnen Befehle direkt umgesetzt sehen. Die Fallstudie zu openHAB wird in Abschnitt 6.1 detailliert vorgestellt.

Für die Fallstudie zur Animationsgenerierung mit Alice wurde ein zweistufiger Code-Erzeuger entwickelt. Zuerst wird aus den ausgewerteten Annotationen ein Code-Modell erzeugt, das dem Aufbau von Alice-Programmen genügt. Anschließend wird dieses Modell einem Generator übergeben, der die benötigten Quelldateien zusammenfasst und den Quelltext erzeugt. Diese Zweiteilung bietet sich an, da Alice dem Alice-Benutzer zwar einen Java-artigen Quelltext zeigt, intern aber mit einem eigenen Datenformat arbeitet. Nur den Quelltext zu erzeugen, ist für Alice nicht ausreichend, da die erzeugte Datei neben dem Programmtext auch alle (Binär-)Daten zu den 3D-Modellen enthalten muss. Der Generator erzeugt folglich keinen (vom Menschen) lesbaren Quelltext, sondern das von Alice geforderte Dateiformat. Öffnet man die erzeugte Datei in Alice, kann man den Quelltext lesen. Details zur Programmsynthese für Alice finden sich in Abschnitt 6.2.7.

5.5 Zusammenfassung

Dieses Kapitel stellte das Konzept von NLCI sowie den Aufbau der Forschungsarbeit detailliert dar. Den Kern von NLCI bildet eine Ontologie, welche die Ziel-API bzw. Domäne beschreibt und den Verarbeitungsstufen zur Verfügung stellt. Die NLCI-Ontologie beschreibt objektorientierte APIs und enthält alle Informationen, die man benötigt, um eine API anzusprechen. Darüber hinaus enthält die Ontologie linguistische Informationen wie beispielsweise Verknüpfungen zu Synsets von WordNet.

Die vorgestellten Textanalysen sind API- bzw. domänenunabhängig. Der erste Schritt der Textanalyse verknüpft den Eingabetext mit der Domänenontologie und stellt so einen Bezug zwischen Wörtern und Programmierelementen (d.h. Klassen und Methoden) her. Die folgenden Textanalysen erfassen linguistische Strukturen, die von Menschen dazu benutzt werden, Aktionen und Handlungsfolgen zu beschreiben. Die Architektur ist offen gestaltet, sodass für spezielle Anwendungsfälle weitere Analysestufen hinzugefügt werden können. Diese können bspw. über konkretes Domänenwissen verfügen und so die Textanalyse verbessern. Aus den linguistischen Analysen wird dann ein domänenabhängiger, linguistischer AST erzeugt. Den Abschluss bildet ein Code-Erzeuger, welcher abhängig von der gewählten Programmiersprache ist; er kann unabhängig von linguistischen Analysen entworfen werden und benötigt keine Anpassungen, wenn die Domänenontologie oder die NLP-Analysen ausgetauscht werden.

Im folgenden Kapitel werden zwei Fallstudien vorgestellt und Einsetzbarkeit und Leistungsfähigkeit von NLCI betrachtet.

Kapitel 6

Fallstudien und Evaluation

„Building the right product requires systematically and relentlessly testing that vision to discover which elements of it are brilliant, and which are crazy.“

Eric Ries, 2014

Dieses Kapitel stellt die Fallstudien vor, in denen NLCI auf seine Tauglichkeit für die natürlich-sprachliche Programmierung untersucht wurde. Betrachtet wurden zwei stark unterschiedliche Systeme: openHAB und Alice.

Das erste betrachtete, einfachere System ist openHAB [ope]. Es ist eine quelloffene Steuerungssoftware für intelligente Häuser. Es ermöglicht die zentrale Steuerung von Licht, Rollos und ähnlichen Elementen in einem Haus. openHAB verfügt nur über einen eingeschränkten Funktionsumfang und die zugehörige NLCI-Ontologie ist typischerweise klein, da sie einen konkreten Haushalt modelliert und die Ontologie nur die Elemente dieses Haushalts enthalten muss; die Steuerelemente stellen nur wenige Funktionen bereit und haben eine geringe funktionale Überlappung. Der Haushalt, der in der Fallstudie betrachtet wird, umfasst 114 Klassen und lediglich neun Methoden. Die sprachliche Erschließung einer Haussteuerung ist dennoch ein interessanter Einsatzzweck, dessen Umsetzung verhältnismäßig einfach erscheint. Für die Fallstudie wurde das Beispielszenario des openHAB-Projektes verwendet, das die verschiedenen Steuerelemente demonstriert.

Das zweite betrachtete System ist Alice [Con97]. Alice ist eine interaktive Programmierumgebung, in der vorgefertigte 3D-Objekte dazu verwendet werden können, Animationen und Spiele zu programmieren. Es wurde entwickelt, um eine grundlegende Einführung in die objektorientierte Programmierung zu geben [CAB+00]. Daher enthält es Objekte, Methoden, Variablen und Kontrollstrukturen. In der Fallstudie werden 3D-Animationen aus gegebenen Drehbüchern erzeugt. Die Fallstudie mit Alice ist umfang-

reicher als die mit openHAB, da Alice über viele verschiedene Modelle verfügt und der zur Verfügung stehende Funktionsumfang weitaus größer ist. In der Fallstudie wird eine NLCI-Ontologie mit 914 Klassen und 393 Methoden verwendet.

Die nachfolgenden Unterkapitel beschreiben detailliert, wie NLCI für den jeweiligen Einsatzzweck verwendet wurde und welche Schritte hierfür nötig waren. NLCI wurde nicht auf die beiden Systeme angepasst und es wurden keine domänenspezifischen Analysen verwendet; als Vorbereitung notwendig war somit ausschließlich die Modellierung der API in der NLCI-Ontologie. Beschrieben werden daher der teil-manuelle Ontologie-Aufbau für openHAB sowie der vollständig automatische für Alice. Dann zeigen Fallstudien, wie gut die vorgestellten Analysen funktionieren und wie gut das Endergebnis letztendlich ist.

6.1 Steuerung eines intelligenten Hauses mit openHAB

openHAB ist eine Steuerungssoftware für intelligente Häuser und wird als quelloffenes Projekt vertrieben. Eine vollständige Beschreibung des Funktionsumfangs findet sich auf der Webseite des Projekts[1]; dort ist folgendes zu lesen:

> „openHAB is a software for integrating different home automation systems and technologies into one single solution that allows over-arching automation rules and that offers uniform user interfaces. This means openHAB [. . .] has a powerful rule engine to fulfill all your automation needs [. . .] and is easily extensible to integrate with new systems and devices [. . .].“

openHAB vereint verschiedene Komponenten der Heimautomatisierung wie bspw. Heizungen, Lichter und normale Schalter in einer zentralen Steuerung. Benutzer können Regeln definieren, die einzelne Aktionen (z.B. ein bestimmtes Licht einschalten) oder ganze Abfolgen von Aktionen auslösen (z.B. zuerst das Licht im Wohnzimmer dimmen, danach die Stereoanlage und den Fernseher einschalten). Auslöser für Regeln können das Drücken eines Knopfes, ein Kalendereintrag oder Sensordaten sein (z.B. das Fallen der Außentemperatur unter einen Schwellwert). openHAB verfügt über einen Dienstgeber für eine grafische Benutzerschnittstelle und kann über Smartphones gesteuert werden; eine sprachliche Steuerung existierte zum Zeitpunkt der Fallstudie nicht. NLCI soll dazu genutzt werden, niedergeschriebene Anweisungen in entsprechende API-Aufrufe umzusetzen. Da hierbei keine Kontrollstrukturen benötigt werden und die Befehle chronologisch

[1] http://www.openhab.org/features/introduction.html; zuletzt besucht am 08.02.2016.

korrekt aufgeschrieben sind, wird auf die Evaluation der Korrektur der Reihenfolge und der Erkennung der Kontrollstrukturen verzichtet. Diese Fallstudie verdeutlicht daher vor allem, wie gering der Aufwand zur Erstellung der NLCI-Ontologie ist.

6.1.1 Aufbau der Ontologie

openHAB stellt neun verschiedene Komponententypen zur Verfügung, die teilweise aktiv (bspw. Lichtschalter), teilweise passiv (bspw. Temperatursensoren) sind. Für einen Haushalt müssen die tatsächlich verfügbaren Komponenten in einer Konfiguration benannt werden; die darin aufgeführten Komponenten verfügen über die in openHAB vordefinierten Funktionen. Die konfigurierten Komponenten können dann zu Gruppen zusammengefasst werden, um bspw. alle Lichter im Keller mit einem einzelnen Kommando ansprechen zu können. Die Namen der Komponenten und Gruppen sind frei wählbar.

In dieser Fallstudie gehen wir davon aus, dass openHAB bereits für einen Haushalt konfiguriert ist und nur noch eine sprachliche Ansteuerung ermöglicht werden soll. Daher erfassen wir zunächst alle Komponententypen von Hand in der Ontologie; Grundlage hierfür bildet das Anwender-Handbuch von openHAB. Aufgrund der eingeschränkten Funktionalitäten der Komponententypen umfasst die Ontologie nur neun Methoden. Dieser manuelle Teil der Vorbereitung ist openHAB-spezifisch, jedoch unabhängig von einer konkreten openHAB-Konfiguration. Soll eine andere openHAB-Konfiguration verwendet werden, kann die hier erstellte Ontologie-Vorlage mit einer anderen Konfiguration gefüllt werden.

Die Fallstudie nutzt die Konfiguration eines Haushalts, die als Beispiel auf der Projektseite von openHAB bereitgestellt wurde. Der Haushalt umfasst zwei Stockwerke, neun Zimmer und insgesamt 92 Komponenten, darunter Heizungen, eine Stereoanlage und Gruppen von Lichtern. Die Komponenten sind sprechend benannt und darüber hinaus zu 22 Gruppen zusammengefasst, welche hauptsächlich die Stockwerke und Räume repräsentieren. Die Konfiguration ist in Abschnitt A.1 als Referenz abgedruckt.

Um aus der Konfiguration für openHAB eine NLCI-Ontologie zu erstellen, wurde ein Ontologie-Generator entwickelt. Er liest eine openHAB-Konfiguration ein und fügt die enthaltenen Komponenten in die vorbereitete NLCI-Ontologie ein. Die Entwicklung des Generators war einfach möglich, da die Konfiguration als Textdatei vorliegt und nahm weniger als einen halben Tag in Anspruch. Der Generator führt vor dem Einfügen der Elemente in die Ontologie einige Bereinigungen an den Elementnamen durch, die sich aus Konventionen ergeben.

Tabelle 6.1: Übersicht über die Eingabetexte für die openHAB-Fallstudie.

# Texte	4
# Befehlssätze	15
# Wörter (gesamt)	149
# API-Elemente zu identifizieren	39
# API-Aufrufe zu identifizieren	20

Beispiel

In der openHAB-Konfiguration sind die Elemente nach Stockwerken geordnet angegeben. Ebenso lässt sich aus dem Namen eines Elements erschließen, wo es sich befindet. So tragen alle Elemente als Präfix ihren Typ im Namen, anschließend folgt eine Kennzeichnung für das Stockwerk, zuletzt steht der eigentliche Name des Elements. Der Generator übersetzt bspw. die Kennzeichnung „GF“ zu „ground floor“ oder „FF“ zu „first floor“. So wird zu „Light_FF_Bed_Ceiling“ der sprechende Name „Light first floor bedroom ceiling“ in der Ontologie gespeichert.

Die erzeugte Ontologie wurde dann automatisch auf die Verwendung mit NLCI vorbereitet, d.h. die Bezeichner aufgetrennt und mit WordNet verknüpft. Allen Klassen wurden dabei Synonyme zugewiesen, die in den Eingabetexten verwendet werden können.

An der generierten Ontologie wurden für die Fallstudie keine Veränderungen bzw. Verbesserungen vorgenommen. Da die Ontologie den gesamten zu steuernden Haushalt abbildet, können die Eingabetexte direkt mit den Befehlen beginnen und müssen die Konfiguration nicht erst aufbauen.

6.1.2 Auswertung

Zur Evaluation verwendeten wir vier Eingabetexte mit insgesamt 15 Befehlssätzen, welche die Aktionsteile der Regeln aus dem Beispielszenario von openHAB widerspiegeln. Tabelle 6.1 gibt einen Überblick über die verwendeten Texte. Der Vollständigkeit halber sind die Texte in Abschnitt A.2 abgedruckt. Die Eingabesätze sind nicht beschreibend, sondern enthalten Anweisungen an ein (anonymes) System; NLCI setzt hierbei automatisch den einzigen möglichen Akteur der API (openHAB selbst) als Standardakteur ein. Beispiele für Befehlssätze sind „Turn the light on over the table in the kitchen.“ und „Turn on the heaters in the living room and increase the volume of the radio.“

Wir vergleichen die erwarteten API-Aufrufe mit den tatsächlichen Ergebnissen von NLCI mehrstufig: Zunächst prüfen wir für jedes erwartete API-Element, ob das entsprechende Text-Element eine Annotierung von NLCI erhalten hat oder nicht. Dann ermitteln wir Präzision, Ausbeute und das F_1-Maß für die Aktionen und Objekte (da es nur einen impliziten Akteur gibt, wird dieser immer korrekt zugewiesen). Anschließend ermitteln wir Präzision, Ausbeute und das F_1-Maß für die atomaren Sätze, bestehend aus Akteur, Aktion und Objekt, das heißt für die möglichen API-Aufrufe. Da NLCI nicht nur ein Ergebnis annotiert, sondern eine sortierte Liste von möglichen Ergebnissen liefert, werden die Metriken für die TOP-N der Ergebnisse ermittelt (mit $N \in \{1,2,3,5,10\}$). Hierdurch lässt sich abschätzen, ob die im Anschluss durchzuführende Methodenauswahl überhaupt funktionieren kann – nur wenn das richtige Element in der Ergebnismenge vorhanden ist, kann es ausgewählt werden.

Präzision, Ausbeute und das F_1-Maß (engl. *precision*, *recall* und F_1 *score*) sind Metriken, die häufig in der Informationsbeschaffung benutzt werden. Sie werden dort klassischerweise verwendet, um die Güte von Suchergebnissen zu bewerten: Ausgehend von einer Grundgesamtheit an Dokumenten wird beurteilt, wie viele relevante Dokumente für eine Suchanfrage zurückgeliefert werden und wie viele irrelevante Dokumente sich im Ergebnis befinden. Die Präzision berechnet sich als Quotient aus relevanten gelieferten Dokumenten im Verhältnis zu allen gelieferten Dokumenten; d.h. echt-positive dividiert durch echt-positive und falsch-positive Treffer: $P = p_t/(p_t + p_f)$. Die Ausbeute bewertet, wie viele der relevanten Dokumente tatsächlich geliefert wurden; d.h. echt-positive geteilt durch echt-positive und falsch-negative Treffer: $R = p_t/(p_t + n_f)$. Wünschenswert ist es, eine Präzision und Ausbeute von jeweils 1 zu erhalten. Bei Suchverfahren muss man die beiden Metriken jedoch oft gegeneinander abwägen: Es ist sehr einfach, eine Ausbeute von 1 zu erreichen: man liefert einfach den gesamten Dokumentbestand zurück. Dann geht jedoch die Präzision gegen 0. Umgekehrt kann man ein Dokument nur dann als Treffer zurück liefern, wenn man sich sehr sicher ist; dann wird typischerweise die Präzision sehr gut, die Ausbeute im Gegenzug stark abfallen. Das F_1-Maß trägt dieser gegenseitigen Abhängigkeit Rechnung und ist definiert als das harmonische Mittel von Präzision und Ausbeute: $F_1 = 2 \times (P \times R)/(P + R)$. Es wird oft verwendet, um eine einzelne Maßzahl für die Güte eines Verfahrens zu erhalten. Der Wertebereich aller drei Metriken ist von 0 bis 1, wobei 1 das beste Ergebnis ist; häufig werden daher die Ergebnisse in % angegeben.

Wir passen diese Metriken wie folgt an unseren Einsatzzweck an: Präzision ist der Quotient aus der Anzahl der korrekten Annotationen dividiert durch alle Annotationen. Ausbeute ist der Quotient aus der Anzahl der korrekten Annotationen dividiert durch die Anzahl der erwarteten Annotationen. In unserem Fall gibt es pro Textelement nur eine

Tabelle 6.2: Evaluationsergebnisse der openHAB-Fallstudie. (Alle Angaben in %.)

	TOP-1	TOP-2	TOP-3	TOP-5	TOP-10
Präzision	63,2	43,1	33,7	29,5	22,8
Ausbeute	61,5	79,5	79,5	87,2	87,2
F_1-Maß	62,3	55,9	47,3	44,2	36,1

(a) Die Ergebnisse für die Zuordnung der API-Elemente zu atomaren Konstituenten.

	TOP-1	TOP-2	TOP-3	TOP-5	TOP-10
Präzision	73,7	37,8	28,3	17,4	14,9
Ausbeute	70,0	70,0	75,0	75,0	75,0
F_1-Maß	71,8	49,1	41,1	28,3	24,8

(b) Die Ergebnisse für die Zuordnung der API-Aufrufe zu atomaren Sätzen.

korrekte Annotation, weswegen die Präzision beim TOP-N-Vergleich mit steigendem N sinken muss, sobald die gewünschten Annotationen unter den TOP-N sind.

6.1.3 Bewertung der Ergebnisse

Tabelle 6.2 fasst die Ergebnisse zusammen: Tabelle 6.2a zeigt die Ergebnisse für die Zuordnung von API-Elementen zu den atomaren Konstituenten (Akteure, Aktionen und Objekte). Diese Zuordnung bildet die Grundlage für die danach folgende Verknüpfung von atomaren Sätzen mit API-Aufrufen.

Sechs der 39 Individuen sind weder in der TOP-10, noch in der restlichen Ergebnisliste; NLCI kann diese nicht in der API ermitteln. Drei der Fehler erwachsen aus Fehlern des verwendeten Zerteilers: Wenn beispielsweise ein Verb, das eine Aktion beschreibt, fälschlicherweise als Substantiv gekennzeichnet wurde, dann sucht NLCI nicht nach der zugehörigen Methode[2]. Bei den verbleibenden drei fehlenden Individuen handelt es sich um Gruppen, die nicht in der Ontologie gefunden werden können, da sie nicht über sprechende Namen verfügen. Dieses Problem sollte in der openHAB-Konfiguration behoben

[2]NLCI geht davon aus, dass Aktionen mit Verben beschrieben werden, ein Nominalstil wird nicht akzeptiert. Möchte man den Nominalstil verbieten oder weitergehend analysieren, eignen sich Körners Analysen aus seiner Dissertation [Kör14] bzw. die aus Referenz [LKK+15].

Am Lehrstuhl wird zudem an anderen Verfahren geforscht, die weniger syntaktische Informationen verwenden und dadurch an dieser Stelle robuster sind. Ein mögliches Verfahren verwendet einen Graph als interne Darstellung und bettet die Textanalysen als Agenten ein, die den Graph lesen und bearbeiten können [WT15]. Hier könnte ein Agent (Teil-)Sätze ohne Aktionen bearbeiten und Fehler der Wortartmarkierung ausgleichen.

werden (was im Sinne einer sprechend benannten API wäre). Alternativ könnten Bezeichner in einem Vorbereitungsschritt beim Erzeugen der Ontologie bereinigt werden. Die Namensgebung dieser Gruppen folgt einem Schema, das bei der Implementierung des Ontologie-Generators zunächst nicht erkannt wurde; dennoch könnte es aufgelöst werden. Zuletzt könnten sprechende Bezeichner manuell angegeben werden.

Die Metriken für die Zuordnung der API-Elemente zu den atomaren Konstituenten sind jedoch nur mittelbare Indikatoren: Eine Präzision und eine Ausbeute von 100% auf Akteuren und Objekten ist z.B. nicht hilfreich, wenn die Erkennung der Aktionen *nicht* funktioniert; dann können im nächsten Schritt keine API-Aufrufe erzeugt werden. Tabelle 6.2b zeigt die Ergebnisse für die Verknüpfung der atomaren Sätze mit den möglichen API-Aufrufen.

Die Abbildungen der Eingabetexte auf die Elemente der openHAB-API bzw. der Elemente im Beispiel-Haushalt verlief in der Fallstudie überwiegend erfolgreich: Die 15 Befehlssätze enthalten 20 API-Aufrufe, von denen 14 korrekt identifiziert wurden und an erster Position im Ergebnis stehen. Die Ausbeute der TOP-1-Betrachtung beträgt 70%. Nur wenige Fehler ergeben sich aus Fehlern des verwendeten Zerteilers. Dieses Ergebnis ist leider noch nicht gut genug, um in einer Anwendung eingesetzt zu werden. Bedenkt man jedoch, dass dieses Ergebnis ohne Anpassungen auf den Einsatzzweck openHAB erreicht wurde und die Hälfte der nicht gefundenen API-Elemente auf schlechte Benennungen in der verwendeten openHAB-Konfiguration zurückführen ist, ist das Ergebnis positiv zu bewerten.

Die Auswertung dieser Fallstudie zeigt, dass eine NLCI-Ontologie für ein System verhältnismäßig einfach erzeugt werden kann. Liegt die API maschinenlesbar vor (wovon im Allgemeinen ausgegangen werden kann), so lässt sich die NLCI-Ontologie automatisch erzeugen. Sind die Bezeichner in der API nicht sprechend benannt, kann dies in der API selbst gelöst werden, mit einer Fassade vor der API oder in der NLCI-Ontologie, indem zusätzlich sprechende Namen für die API-Bezeichner notiert werden. Die folgende Fallstudie muss nun zeigen, wie sich der Aufwand und die Güte der Ergebnisse bei einem größeren und komplexeren System verhält.

6.2 3D-Animationen mit Alice

Alice ist eine Programmiersprache mit zugehöriger integrierter Entwicklungsumgebung, die für Programmieranfänger gedacht ist, denen das objektorientierte Programmiermodell beigebracht werden soll [Con97, CAB+00]. Alice wird an der Carnegie Mellon Univer-

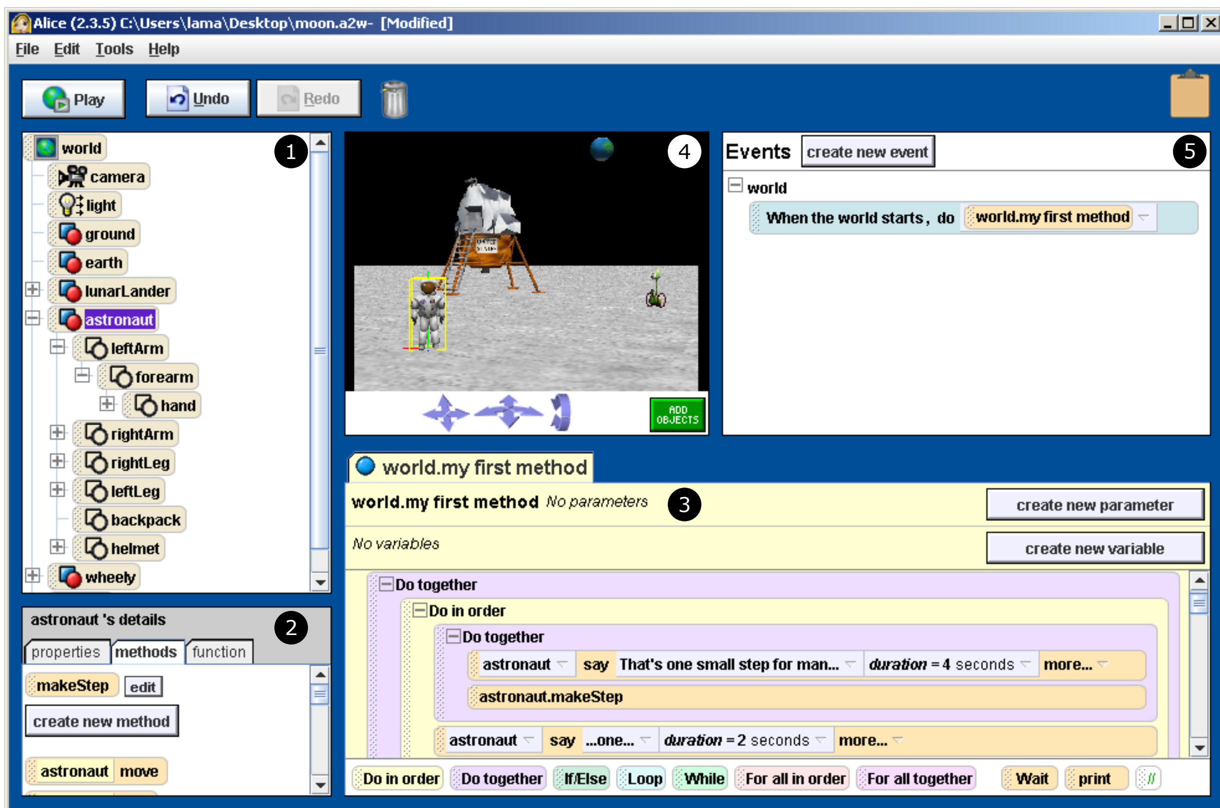


Abbildung 6.1: Bildschirmfoto der Alice-Benutzeroberfläche.

sität entwickelt und steht kostenfrei zum Herunterladen[3] bereit; die von uns eingesetzte Version 2.x ist zudem als Quelltext verfügbar.

In Alice können Animationen mit und ohne Interaktion programmiert werden; für die Interaktion mit einem Spieler stehen einfache Steuerungsmechanismen für Tastatur und Maus zur Verfügung. In unserem Projekt dient Alice als Zielplattform für natürlichsprachlich beschriebene Animationen – insofern beschränkt sich unser Anwendungsfall auf nicht-interaktive Programme.

6.2.1 Die Programmierumgebung Alice

Abbildung 6.1 zeigt die grafische Benutzeroberfläche von Alice. In der Entwicklungsumgebung sind alle Elemente zugreifbar, die für die Programmierung benötigt werden:

1. Objekt-Übersicht: Hier werden alle in der Alice-Welt verwendeten Objekte aufgeführt. Ebenso kann man hier auf ihre Bestandteile zugreifen. In der Abbildung ist der Astronaut ausgewählt und seine Bestandteile aufgefächert.

[3]http://www.alice.org, zuletzt besucht am 31.01.2016.

2. Eigenschaften, Methoden (Handlungen) und Funktionen (das sind (Abfrage-)Methoden mit Rückgabewerten) des aktuell ausgewählten Objekts: Die Objekt-Eigenschaften umfassen Farbe, Opazität sowie Textur und sind für unseren Ansatz weniger relevant. Eigenschaften können vor und zur Laufzeit über entsprechende Zugriffsmethoden verändert werden. Methoden und Funktionen werden mit der Maus von hier in den Skript-Editor gezogen.

3. Skript-Editor: Hier kodiert der Benutzer die Handlung. Es können die Methoden der verwendeten Modelle sowie Kontrollstrukturen verwendet werden. Die verfügbaren Kontrollstrukturen sind unterhalb des Skripts aufgeführt.

4. Vorschaufenster: In diesem Fenster kann man den initialen Aufbau der Szenerie betrachten, die Kameraposition verändern und neue Objekte hinzufügen. Objekte können beliebig in der Welt positioniert und in ihrer Größe skaliert werden. Klickt man auf die Schaltfläche „Play“, so wird das Programm übersetzt und die erzeugte Animation abgespielt.

5. Ereigniseditor: Hier lassen sich Methodenaufrufe mit Ereignissen verknüpfen. In Abbildung 6.1 sieht man, dass zu Beginn der Welt eine Methode *my first method* aufgerufen wird. Diese entspricht der *main*-Methode eines Java-Programms.

Wie in der Objektübersicht zu sehen ist, sind die Elemente innerhalb einer Alice-Welt hierarchisch aufgebaut. In der Welt sind zunächst die Objekte verankert, die für die Darstellung der Animation benötigt werden (Kamera und Licht). Dann folgen die verwendeten Objekte, die in Alice (3D-)Modelle genannt werden; sie sind ebenfalls hierarchisch aufgebaut: Der abgebildete Astronaut „besteht“ aus einem Helm, einem Rucksack, Armen und Beinen, die wiederum aus weiteren Teil-Objekten aufgebaut sein können. Auf der untersten Ebene stehen geometrische Körper bzw. Polygongitter mit Texturen.

Die verfügbaren Modelle können als Klassen einer Klassenbibliothek (vergleichbar mit der Java-Standardbibliothek) angesehen werden. Alle Modelle verfügen über 22 Standardmethoden (wie bspw. bewegen, drehen, und so weiter), die von Alice zur Verfügung gestellt werden. Im Gegensatz zu den Klassen der Java-Standardbiblitohekt sind die Modelle jedoch durch Methoden und Funktionen erweiterbar: Benutzer können neue Methoden aus bestehenden zusammensetzen und Funktionen definieren.

Das zentrale Element in Alice ist die sogenannte Alice-Welt. Sie stellt ein Programm dar, das ausgeführt werden kann; sie entspricht in etwa einer JAR-Datei in Java. Benutzerdefinierte Methoden und Funktionen werden zusammen mit den verwendeten Modellen

direkt in der jeweiligen Alice-Welt-Datei gespeichert. Andere Welten (d.h. andere Programme) werden so nicht beeinflusst. Eine Welt ist somit immer eigenständig und benötigt keine weiteren Dateien, um auf einer anderen Alice-Installation geladen zu werden (auch wenn dort die verwendeten Modelle nicht vorhanden sein sollten). Es ist nicht vorgesehen, eigene Implementierungen aus einer fremden Welt zu laden oder in die Modelle der Klassenbibliothek zu integrieren.

Im folgenden Abschnitt bezeichnet „Alice-Klassenbibliothek" die Gesamtheit der Modelle, die bei Alice mitgeliefert werden; in der Regel enthalten sie keine Methoden außer den Standardmethoden. Modelle, die um Methoden erweitert und in einer Welt-Datei gespeichert wurden, werden als Welt-Modelle bezeichnet; bei einem Welt-Modell handelt es sich in der Regel um ein Duplikat eines Modells aus der Klassenbibliothek.

Neue Modelle können nicht innerhalb der Alice-Entwicklungsumgebung erstellt werden, aber es ist möglich, neue Modelle über sogenannte Charakter-Dateien zu laden. Diese Charakter-Dateien enthalten die von den Modellen benötigten Grafiken bzw. Texturen sowie die benutzerdefinierten Methoden.

Neben dem Erstellen von neuen Methoden und dem (sequenziellen) Aufrufen von Methoden umfasst Alice alle gängigen Kontrollstrukturen, die aus der prozeduralen Programmierung bekannt sind: Neben einfachen *for*- und *while*-Schleifen gibt es *foreach*-Schleifen und die Möglichkeit Methodenaufrufe zu parallelisieren (*do together* und *for all together*). Zusätzlich gibt es die Möglichkeit eine *if-else*-Verzweigung zu verwenden; diese ist jedoch bei Drehbüchern nicht zu erwarten, da sie einen alternativlosen Handlungsstrang beschreiben.

Da Programmieranfänger die Zielgruppe von Alice sind, werden überwiegend sprechende englische Namen für Modelle und deren Methoden bzw. Funktionen verwendet; sind Namen aus mehreren Wörtern zusammengesetzt, wird zumeist die Binnenmajuskelschreibweise (engl. *CamelCase*) verwendet. Diese Namenskonvention wird weitestgehend befolgt – sowohl bei den bei Alice mitgelieferten Modellen und Beispielen, als auch bei den meisten benutzerdefinierten Modellen, die wir bei unserer Recherche gefunden haben. Dieser Umstand kommt unserem Vorhaben entgegen, natürliche Sprache auf Alice-Elemente abzubilden.

In der Fallstudie verwenden wir alle Alice-Modelle, die uns zur Verfügung stehen und erzeugen daraus eine große NLCI-Ontologie. Als Eingabetexte kommen die Texte des NLCI-Korpus zum Einsatz (siehe Abschnitt 3.3.1). Das folgende Unterkapitel beschreibt den automatischen Aufbau der NLCI-Ontologie, da dieser spezifisch für Alice entwickelt und in Abschnitt 5.2 nicht beschrieben wurde. Danach folgen die Evaluierungen der in Abschnitt 5.3 beschriebenen NLP-Analysen .

6.2.2 Aufbau der Ontologie

Um eine möglichst aussagekräftige Evaluation durchführen zu können, wurden alle verfügbaren Modelle verwendet; d.h die Modelle aus der Alice-Bibliothek und die Modelle aus den Animationen Dritter, die wir bei unserer Recherche gefunden haben.

Für die Evaluation stehen 897 Quelldateien zur Verfügung, die 914 3D-Modelle mit insgesamt 371 Methoden umfassen; die 3D-Modelle sind aus insgesamt 7692 Bausteinen aufgebaut. Hinzu kommen die 22 Methoden, die in der Rahmenarchitektur definiert sind und bei allen Modellen und Komponenten verfügbar sind. Diese Menge an 3D-Modellen und Methoden manuell in Ontologie-Instanzen zu überführen, ist kaum möglich. Die Modelle und deren Methoden verfügen überwiegend über sprechende Namen, die sich an gängige Namenskonventionen halten. Diese Tatsache sorgt dafür, dass die automatisch aufgebaute Ontologie nicht manuell aufbereitet werden muss. Der hier beschriebene Ontologie-Erzeuger funktioniert vollautomatisch und das Ergebnis wurde für die Evaluation nicht nachbearbeitet.

Die folgenden Abschnitte beschreiben den Aufbau der Ontologie und betrachten, ob die erzeugte Ontologie die Klassenbibliothek von Alice richtig abbildet.

6.2.2.1 Der Ontologie-Erzeuger

Der Ontologie-Erzeuger liest gegebene Charakter- und Welt-Dateien aus und speichert die enthaltenen 3D-Modelle als Instanzen von *Object* und deren Methoden als Instanzen von *UserDefinedMethod* in der Ontologie; die Bausteine, aus denen die Modelle aufgebaut sind, werden als *Component* modelliert. Die Charakter- oder Welt-Dateien werden dabei als *Origin* erfasst und mit allen aus ihnen extrahierten Elementen verknüpft.

Um den Generator zu entwerfen, wurde die Alice-Architektur analysiert und die Modelltypen auf die Konzepte der NLCI-Ontologie abgebildet [Pet12]. Alice unterscheidet intern zwischen Methoden und Funktionen. Methoden haben keinen Rückgabewert und werden bspw. für Bewegungsabläufe und Handlungen verwendet. Funktionen haben einen Rückgabewert und werden dazu verwendet, den Zustand eines Modells abzufragen, so wie bspw. den Abstand eines Modells zu einem anderen.

Diese Unterscheidung gibt es in der Ontologievorlage von NLCI an sich nicht, jedoch kann die Unterscheidung weiterhin beibehalten werden, indem zur vorgegebenen Klasse für Methoden zwei Unterklassen angelegt werden. Da die Ober-/Unterklassenbeziehung von der Inferenzmaschine ausgewertet wird, gehören alle Instanzen dieser Alice-spezifischen Unterklassen automatisch auch zur von NLCI festgelegten Klasse für Methoden. Ebenso wird bei anderen Eigenheiten von Alice verfahren. Die Methoden, die die Alice-

Umgebung bereitstellt, werden zuerst zur Ontologie hinzugefügt und mithilfe von Regeln mit allen Objekten verknüpft.

Der Generator beginnt seine Suche nach Modellen und deren Elementen (Methoden, Funktionen etc.) ausgehend von einer Datei: einer Charakter- oder einer Welt-Datei. Dann wird der hierarchische Aufbau ausgenutzt, um die enthaltenen Elemente nacheinander zu betrachten. Hierzu wurde eine Zuständigkeitskette entworfen, deren erstes Glied das aktuelle Element übergeben bekommt. Für jeden Elementtyp wurde eine Verarbeitungseinheit implementiert, die das aktuelle Element betrachtet und ggf. in der Ontologie registriert. Kann eine Verarbeitungseinheit das aktuelle Element nicht verarbeiten, so gibt sie es an die nächste Einheit in der Kette weiter. Am Ende der Kette steht eine Verarbeitungseinheit, die alle Elementtypen entgegen nimmt und protokolliert, dass das Element nicht in der Ontologie registriert wurde. So wurde sichergestellt, dass keine Elementtypen übersehen wurden.

Die Verarbeitungseinheiten übernehmen die Namen der Modelle und ihrer Methoden ohne Änderung in die Ontologie. Teilmodelle wie bspw. der Kopf einer Figur werden mit einem vollqualifizierten Namen in die Ontologie eingefügt, da es in der Ontologie keine zwei Elemente mit demselben Namen geben darf. Fügte man Köpfe verschiedener Modelle mit demselben Namen `head` in die Ontologie ein, so wäre diese für Inferenzmaschinen unbrauchbar, da in einer Ontologie keine zwei Elemente mit demselben Namen vorhanden sein dürfen.

Da die benutzerdefinierten Methoden nicht in der Alice-Klassenbibliothek abgelegt werden, sondern nur in den Welten, müssen die Welt-Dateien ebenfalls analysiert und in der Ontologie registriert werden. Durch diese Registrierung treten naturgemäß Duplikate in der Ontologie auf, die aufgelöst werden müssen. Hierzu fasst der Generator gleiche Modelle zusammen und speichert die Methoden, die aus unterschiedlichen Quellen stammen, gemeinsam ab. Da NLCI die Herkunft der Elemente in der Ontologie speichert (siehe Abschnitt 5.2.1), können alle Methoden, die zu einem Modell definiert wurden, in der Ontologie erfasst und später verwendet werden. (Wird am Ende des NLCI-Prozzesses eine Welt-Datei erzeugt, müssen ggf. Methoden aus unterschiedlichen Quell-Dateien zusammengefasst und in der erzeugten Welt-Datei hinterlegt werden.)

Beispiel

In der Klassenbibliothek befindet sich das Modell eines Astronauten, das lediglich über die Standardmethoden verfügt.

Ein Benutzer verwendet nun dieses Modell in einer Animation und legt eine neue Methode `makeSmallStep()` an. Die Animation speichert er als Welt W_1 ab. In einer

weiteren Animation verwendet er ebenfalls den Astronauten und legt eine Methode `makeGiantLeap()` an. Diese Animation speichert er als Welt W_2 ab.

Erzeugt man nun aus der Klassenbibliothek und den beiden Welten eine Ontologie, so wird zuerst der Astronaut in der Ontologie abgelegt (und per Regelanwendung mit den Standardmethoden verknüpft). Entsprechend wird als Quelle für das Modell der Pfad zur Klassenbibliothek eingetragen.

Nach der Klassenbibliothek werden die Welten in die Ontologie hinzugefügt. Für die darin enthaltenen Astronauten werden jedoch *keine* neue Instanzen in der Ontologie angelegt. Die beiden Methoden `makeSmallStep()` und `makeGiantLeap()` werden mit dem bestehenden Astronauten verknüpft. Beide Methoden enthalten eine Quellenangabe, die den Pfad zu Welt W_1 bzw. W_2 angibt.

Ob Modelle beim Generieren der Ontologie zusammengefasst werden können, wird anhand ihres Namens und ihrer Struktur entschieden. Damit zwei Modelle zusammengefasst werden, müssen ihre Namen übereinstimmen, wobei angehängte Zahlen nicht berücksichtigt werden. Anschließend wird für alle Teilmodelle (Arme, Beine usw.) überprüft, ob sie genau so im anderen Modell auch vorhanden sind. Wird auf diese Weise festgestellt, dass zwei Modelle identisch sind, dann wird nur ein Modell in der Ontologie angelegt und die in beiden verfügbaren Methoden darin vereint. Sind die Modelle unterschiedlich, so werden sie getrennt voneinander in der Ontologie abgelegt. Diese Überprüfung wird jedes Mal durchgeführt, wenn ein Modell der Ontologie hinzugefügt werden soll. Die Idee hinter dieser Verschmelzung ist, dass bei normaler Benutzung von Alice höchstens der Name des Modells verändert wird; wird ein Modell mehrfach in einer Welt verwendet, so fügt Alice automatisch Zahlen als Suffixe zum Namen hinzu. Verändert ein Benutzer die Bestandteile des Modells, so können wir nicht mehr davon ausgehen, dass es sich um das gleiche Modell handelt. Dann darf nicht mehr zusammengefasst werden.

Beispiel

Ein Benutzer verwendet in einer Welt drei mal das Modell `Astronaut`, das in seiner ursprünglichen Form unter anderem aus einem `Helmet` und dem Rucksack `Backpack` besteht. Alice bezeichnet die Modelle in der Welt automatisch als `Astronaut` bis `Astronaut3`.

Der Benutzer entscheidet sich dazu, den dritten Astronauten zu verändern und benennt dabei den Helm in `Head` um.

> Wird diese Welt in eine Ontologie überführt, so werden `Astronaut` und `Astronaut2` zusammengefasst. Der `Astronaut` in der Ontologie enthält alle Methoden, welche die beiden Modelle in der Welt hatten. Da `Astronaut3` die Gleichheit aufgrund des neuen Kopfes verletzt, wird es als eine separate Instanz in der Ontologie angelegt.

Modelle nur aufgrund ihres Namens und ihrer Bestandteile zu vergleichen, ist nur bedingt korrekt: Die Heuristik des Generators geht davon aus, dass es keine zwei unterschiedliche Modelle mit demselben Namen und demselben Aufbau gibt und dem Benutzer lediglich die Werkzeuge der Alice-Entwicklungsumgebung zur Verfügung stehen. Diese Vorgehensweise führt derzeit nicht zu Fehlern, da in Alice Benutzer die Modelle nur sehr eingeschränkt verändern können, z.B. in der Farbgebung.

6.2.2.2 Bewertung der erzeugten Ontologie

Im Rahmen einer Studienarbeit wurde die Klassenbibliothek von Alice dahingehend untersucht, ob sich die Modelle an die Namenskonventionen halten, die dem beschriebenen Vorgehen zugrunde liegen [Ama15]. Die ursprüngliche Klassenbibliothek von Alice verstößt bei den Modellen, die in unseren Evaluationsbeispielen verwendet werden, nicht gegen die getroffenen Annahmen. Betrachtet man die gesamte ursprüngliche Klassenbibliothek, so haben 59 (7%) der Modelle keinen eindeutigen Namen und es existieren Modelle, die mehrfach unter verschiedenen Namen abgelegt sind. Innerhalb von Alice ist das nicht problematisch, da die Modelle in einer Hierarchie dargestellt werden und daher der Name nicht das einzige Erkennungsmerkmal ist. Im Rahmen der Programmierung mit NLCI würde dieser Umstand jedoch zu Fehlern führen, weswegen die Klassenbibliothek bereinigt wurde. Dabei wurden Rechtschreibfehler in Modellnamen behoben und durchgesetzt, dass es keine zwei verschiedene Modelle mit demselben Namen gibt. Für die Evaluation sind diese Anpassungen unerheblich, da die verwendeten Modelle nicht von den Korrekturen betroffen waren. Daher wurde die unbereinigte Variante verwendet.

Beim Einlesen der Klassenbibliothek und der verfügbaren Welten werden keine Elemente übersehen, d.h. es werden keine Modelle oder Modellelemente als nicht verarbeitet protokolliert. Eine so entstandene Ontologie wird automatisch vor und nach der Verknüpfung mit WordNet von einer Inferenzmaschine auf Konsistenz geprüft; nur wenn sie konsistent ist, kann sie im NLCI-Prozess verwendet werden.

Die Zusammenfassung von gleichen Modellen wurde getestet: Extrahiert man ein Modell mehrfach in eine Ontologie, so ändert sie sich beim zweiten Durchlauf nicht; enthält eins der Duplikate Methoden, die die übrigen nicht enthalten, dann werden diese Me-

Tabelle 6.3: Übersicht über die Eingabetexte für die Alice-Fallstudie zur Verknüpfung der API-Elemente.

# Texte	50
# Beschreibungssätze	703
# Wörter (gesamt)	6764
# API-Elemente zu identifizieren	1517
# API-Aufrufe zu identifizieren	570

thoden erwartungsgemäß in der Ontologie registriert. Umgekehrt wurde überprüft, ob identisch benannte aber strukturell unterschiedliche Modelle zusammengefasst werden. Auch hier konnten keine Fehler festgestellt werden.

Die in Abschnitt 5.2.4 vorgestellte Anreicherung der Ontologie mit Synonymen aus WordNet funktioniert zufriedenstellend: Für 909 der 914 Modelle und alle Methoden wurde ein oder mehrere Synonyme ermittelt. Diese Beobachtung zeigt jedoch nur, dass die Bezeichner in der Alice-Bibliothek englische Wörter sind, die in WordNet verzeichnet sind. Man kann hieraus nicht ablesen, ob tatsächlich die richtigen Synsets verknüpft wurden. Dass dieses semantische Rauschen nicht zu Problemen führt, zeigen die Auswertungen in Abschnitt 6.2.4.

6.2.3 API-Verknüpfung

Die folgenden Abschnitte beschreiben die Evaluation der API-Verknüpfung. Zunächst wird beschrieben, welche Texte zur Evaluation verwendet wurden und wie diese vorbereitet werden mussten. Der darauf folgende Abschnitt präsentiert und diskutiert die erzielten Ergebnisse.

6.2.3.1 Vorbereitung der Evaluation

Zur Evaluation der API-Verknüpfung werden 50 Texte aus dem Alice Korpus verwendet. Teile des Korpus wurden ausgeschlossen, z.B. die Texte, die lediglich den Aufbau einer Szenerie beschreiben. Tabelle 6.3 fasst die verwendeten Texte zusammen.

Die Musterlösungen der zugehörigen Animationen eignen sich leider nicht als Goldstandard für die Evaluierung: Autoren beschreiben nicht immer exakt die jeweilige Animation und lassen unter Umständen Aktionen aus. Außerdem gibt es zwei weitere Fehlerquellen, die eine Abweichung von der Musterlösung rechtfertigen: Redundanz und

fehlende Informationen. Im Folgenden bezeichnen wir Beschreibungen als redundant, wenn Sie für die Erzeugung der Animation nicht nötig wären, da sie bereits in anderen Aktionen enthalten sind. Informationen fehlen immer dann, wenn (auch ein Mensch) einen Methodenaufruf nicht aus dem Eingabetext ableiten könnte, ohne eine Rückfrage zu stellen, Annahmen zu treffen oder die Informationen aufzufüllen.

Aus diesen Gründen wurden zu den verwendeten Texten auf Basis der Musterlösungen ermittelt, welche Modelle und Methoden zu den jeweiligen Textelementen annotiert werden sollten. So entstand für jeden Text eine spezifische Musterlösung, mit der das Ergebnis von NLCI verglichen werden kann.

Beispiel

Gegeben ist eine Szenerie, in der ein Mann `m` und eine Frau `f` mit 10 Metern Abstand voneinander stehen und in die Kamera blicken. Die Animation zeigt dann, wie sich die Frau zum Mann dreht und mit der linken Hand winkt. Daraufhin dreht sich der Mann zur Frau hin und geht dann zu ihr.

Die API stellt dazu die Methode `turnTo(Object o)` zum Drehen zu einem Objekt `o`, die Methode `wave(Hand h)` zum Winken mit einer Hand `h` sowie die Methode `goTo(Object z)` um zu einem Ziel `z` zu laufen zur Verfügung Die Methode `goTo(...)` sei dabei in der API so implementiert, dass zunächst `turnTo(...)` aufgerufen und anschließend die Laufbewegung ausgeführt wird; so stellt die API sicher, dass sich ein Modell zuerst zu seinem Ziel hin dreht und dann beginnt loszulaufen.

Redundanz

Beschreibt der Proband nun, dass 1) sich die Frau zum Mann dreht, 2) mit der linken Hand winkt, 3) sich dann der Mann zur Frau dreht und 4) zu ihr hin läuft, so werden vier Methodenaufrufe generiert: `f.turnTo(m)`, `f.wave(leftHand)`, `m.turnTo(f)` sowie `m.goTo(f)`.

Die Animation könnte jedoch – aufgrund der Implementierung der API – aus den Methodenaufrufen `f.turnTo(m)`, `f.wave(leftHand)`, sowie `m.goTo(f)` vollständig erzeugt werden; der Methodenaufruf `m.turnTo(f)` ist redundant und könnte in der tatsächlichen Implementierung ausgelassen worden sein.

Da dem Probanden die tatsächliche Implementierung unbekannt ist, ist bei der Bewertung der Ergebnisse unerheblich, was in der tatsächlichen Implementierung steht. Redundanz ist sowohl für die Ausführung des generierten Programms als auch für die Evaluation somit unschädlich. Maßgeblich ist, ob NLCI zur Beschreibung des Probanden einen passenden API-Aufruf ermitteln kann.

Fehlende Informationen

Beschreibt ein Proband hingegen, dass 1) sich die Frau zum Mann dreht, 2) winkt, 3) sich dann der Mann zur Frau dreht und 4) zu ihr hin läuft, so können nur drei Methodenaufrufe generiert werden: `f.turnTo(m)`, `m.turnTo(f)` sowie `m.goTo(f)`. Der gemäß der tatsächlichen Implementierung erwartete Methodenaufruf `f.wave(leftHand)` kann aus der Beschreibung nicht generiert werden: der Proband hat nicht angegeben, mit welcher Hand gewunken wird. Bei der Evaluation wird daher zwar erwartet, dass NLCI die Methode `wave(...)` mit dem Verb „winken“ verknüpft, später bei der Methodenauswahl jedoch keinen Aufruf erzeugt.

NLCI könnte an dieser Stelle mögliche passende Methodenargumente nur raten und gerade in diesem Beispiel gibt es zwei sinnvolle gleichberechtigte Argumentkandidaten, die linke und die rechte Hand. Wir haben uns bei der Entwicklung dazu entschieden, *keine* passenden Argumente zu suchen, da bei Alice nur der Parameteryp `Modell` angegeben werden kann und somit *alle* Modelle in der Animation Argumentkandidaten wären.

Eine ideale API sollte eine überladene Methode `wave()` ohne Parameter anbieten oder einen Standardwert für das Argument angeben. Ist eine Überladung verfügbar, für die alle benötigten Argumente in der Beschreibung ermittelt werden können, so wählt NLCI in einem nachgelagerten Schritt automatisch diese Methode aus; sind mehrere Überladungen aufrufbar wählt NLCI die Methode, die die meisten Argumente hat.

Zur Auflösung der Korreferenzketten war die Korreferenzauflösung der CoreNLP-Bib-liothek aus Stanford vorgesehen. Im Verlauf der Entwicklung von NLCI zeigte sich jedoch, dass CoreNLP die Korreferenzen im NLCI-Korpus nicht gut auflösen kann. Besonders der Umstand, dass im Korpus häufig sächliche Akteure (z.B. ein „alien“, ein „bunny“ oder ein „penguin“) verwendet werden, macht die Auflösung von Personalpronomen schwierig, da eine Prüfung des Geschlechts wenig zur Auflösung beiträgt. Darüber hinaus verknüpft die Korreferenzauflösung keine Aktionen miteinander, wenn diese im Text wiederholt werden. Da dies eine Schwäche des vorgelagerten Zerteilers ist und nicht primär von NLCI, wurden die Pronomen in den Eingabetexten manuell aufgelöst.

6.2.3.2 Bewertung der Ergebnisse

Die Bewertung der Ergebnisse erfolgte wie bei der Auswertung der Ergebnisse zu openHAB in Abschnitt 6.1.2; die Kennzahlen Präzision, Ausbeute sowie das F_1-Maß wurden

Tabelle 6.4: Evaluationsergebnisse der Alice-Fallstudie zur Verknüpfung der API-Elemente: Korreferenzen wurden manuell aufgelöst. Vier Individuen sind nicht in der TOP-10; es wurden keine weiteren API-Aufrufe erzeugt. (Alle Angaben in %.)

	TOP-1	TOP-2	TOP-3	TOP-5	TOP-10
Präzision	70,6	36,3	24,6	14,8	7,5
Ausbeute	68,7	76,0	80,0	82,3	83,9
F_1-Maß	69,7	49,2	37,6	25,2	13,7

(a) Die Ergebnisse für die Zuordnung der API-Elemente zu atomaren Konstituenten.

	TOP-1	TOP-2	TOP-3	TOP-5	TOP-10
Präzision	78,1	39,2	26,1	15,7	7,9
Ausbeute	67,0	68,1	68,2	69,5	70,0
F_1-Maß	72,1	49,7	37,8	25,7	14,2

(b) Die Ergebnisse für die Zuordnung der API-Aufrufe zu atomaren Sätzen.

wie dort beschrieben bestimmt. Tabelle 6.4 fasst die Ergebnisse für die aufbereiteten Eingabetexte zusammen. Tabelle 6.4a zeigt die Ergebnisse für von API-Elementen zu den atomaren Konstituenten (Akteure, Aktionen und Objekte); in Tabelle 6.4b sind die Ergebnisse der Verknüpfung der atomaren Sätze mit den möglichen API-Aufrufen aufgeführt. Die Ausbeute ist für die API-Aufrufe in der TOP-10 geringer als bei der Verknüpfung der Ontologie-Elemente. Das ist ein erwartetes Ergebnis, da in der Ontologie viel mehr scheinbar passende Individuen vorhanden sind, als danach bei der Verknüpfung in atomare Sätze zusammengefasst werden können. Der starke Abfall der Präzision und der flache Anstieg der Ausbeute sind ebenso erwartete bzw. erwünschte Ergebnisse: Ist ein korrektes Ergebnis in der Top-N vorhanden, fällt die Präzision der Top-N+1 zwangsläufig (es sei denn, es gibt nur N Ergebnisse) und die Ausbeute steigt nicht weiter an.

Zusammengefasst ermittelt und bewertet NLCI die gewünschten API-Aufrufe so, dass sich für die am besten bewerteten Ergebnisse (TOP-1) eine Ausbeute von 67% und eine Präzision von 78% ergibt. Dieses Ergebnis ist leider noch nicht gut genug, um in einer Anwendung eingesetzt zu werden; andererseits ist diese Abbildung von Text auf eine API ohne jeden Eingriff und ohne spezielles Training vorgenommen worden. Die API-Verknüpfung ermittelt zudem nur Individuen, d.h. Objekte, als mögliche Methodenargumente; ist ein Argument jedoch ein primitives Element oder eine Zeichenkette, so wird es hier noch nicht verknüpft und der zugehörige Methodenaufruf ist nicht erzeugbar. Hinzu kommt, dass die TOP-1-Betrachtung der Präzision vernachlässigt, dass die

API-Verknüpfung jeweils nur einen atomaren Satz betrachtet und nicht das Dokument als Ganzes. Die nachfolgende Analyse zur Auswahl der aufzurufenden Methoden betrachtet hingegen einen atomaren Satz bzw. einen Methodenaufruf im Kontext des gesamten Dokuments. Hierdurch sollten sich bessere Werte für Präzision und Ausbeute erzielen lassen. Das Ergebnis dieser zusätzlichen Analyse wird im folgenden Abschnitt ausgewertet.

Zwei weitere Punkte beeinträchtigen die Leistungsfähigkeit von NLCI: Der Umfang der Ontologie bzw. der bereitgestellten API ist mit 914 Klassen um ein Vielfaches größer, als die tatsächlich verwende Menge an Klassen, was zu einem Rauschen in den Suchergebnissen führen kann. Zudem hält sich die API zwar weitgehend an Namenskonventionen, jedoch könnte die Benennung von Methoden (und Klassen) noch verbessert werden. Eine bessere Bezeichnung der API-Elemente in der API und eine Einschränkung der in der Ontologie abgebildeten Klassen sollte die Ausbeute bzw. die Präzision steigern.

6.2.4 Auswahl der auszuführenden Methoden

Dieser Abschnitt betrachtet, wie sehr die oben beschriebene Auswahl der möglichen Methodenaufrufe eingegrenzt werden und ggf. verfeinert werden kann. Diese Auswertung beleuchtet, welche Ergebnisse NLCI erzielen kann, wenn die API-Verknüpfung mit der Bestimmung der aufzurufenden Methode und der erweiterten Analyse der Methodenparameter (siehe Abschnitt 5.3.1.3 sowie Abschnitt 5.3.1.4) kombiniert wird.

Für diesen Evaluationsschritt wurden insgesamt 61 Texte zu sieben Animationen betrachtet. Zunächst wird beschrieben, wie die Evaluation durchgeführt wurde, anschließend werden die erzielten Ergebnisse diskutiert. Da die Auswahl der Methoden – ebenso wie die Verknüpfung mit der API – stark von der verwendeten NLCI-Ontologie abhängt, wurden zwei Durchläufe durchgeführt. Der erste Durchlauf verwendet eine vollständige Ontologie, wie sie auch für die übrigen Evaluationen verwendet wurde. Der zweite Durchlauf verwendet eine Ontologie, der ausschließlich die Animationen im NLCI-Korpus zugrunde liegen. Die beiden Ontologien unterscheiden sich erheblich was den Umfang angeht (914 ggü. 60 Modelle, jeweils zzgl. Teilmodelle, Methoden usw.) und so liegt der Schluss nahe, dass NLCI mit der wesentlich kleineren Ontologie besser umgehen kann, da sie potentiell weniger ähnliche Modelle und Methoden enthält. So müsste es wesentlich einfacher sein, ein „passendes“ Ontologie-Element zu identifizieren bzw. es sollten sich weniger anscheinend passende Kandidaten ermitteln lassen. In der Folge erwarten wir eine steigende Präzision. Für die Ausbeute erwarten wir keine Verbesserung, da eine Einschränkung des Suchraums nicht dazu führt, dass der verwendete Suchalgo-

rithmus ein Element findet, dass er im großen Suchraum „übersehen" hat. Die Bewertung eines Kandidaten ist zudem von anderen Elementen in der Ontologie unabhängig.

6.2.4.1 Vorbereitung der Evaluation

Wie bei der vorigen Analyse wurden die Eingabetexte entsprechend vorbereitet und die Pronomina in den Eingabetexten aufgelöst. Anschließend wurden die Texte mit NLCI analysiert und geprüft, welche Methodenaufrufe erzeugt werden konnten. Wie in Abschnitt 6.2.3.1 bereits festgestellt wurde, beschreiben die Probanden nicht exakt die Animation, die ihnen gezeigt wurde. Daher müssen die erzeugten Methodenaufrufe einzeln darauf geprüft werden, ob sie die jeweilige Beschreibung umsetzen. Hierzu wurden die Eingabetexte Aktion für Aktion mit dem generierten Quelltext verglichen; traten hierbei Unklarheiten (bspw. zur Ursache eines Fehlers) auf, wurden die Annotationen im Eingabetext betrachtet.

Die Auswertung verwendet die Metriken Präzision und Ausbeute. Gezählt wurden dazu die Methodenaufrufe, die für eine gegebene Aktionsbeschreibung erwartet werden. Dabei wurde zusätzlich betrachtet, wie viele Argumente die jeweils erwartete Methode benötigt. Beschrieben die Probanden eine Aktion nicht oder mit weniger als den benötigten Argumenten, so wurde die Methode als nicht generierbar gewertet und entsprechend auch nicht erwartet, dass NLCI einen Methodenaufruf generiert; erzeugt NLCI aus einer derartigen Beschreibung dennoch einen (dann falschen) Methodenaufruf, so wurde dieser als falsch gewertet. Beschrieb ein Proband hingegen eine Aktion, die in der API verfügbar ist, aber in der Animation nicht verwendet wurde, so wurde ermittelt, ob NLCI für die beschriebene Aktion einen korrekten Aufruf generiert hat oder nicht.

Die Auswertung dieser Ergebnisse ist nicht mehr wie in Abschnitt 6.2.3 als TOP-N-Betrachtung möglich, da pro atomarem Satz nur noch eine Methode aus der Ontologie ausgewählt und somit ein Methodenaufruf erzeugt wird. Daher ergeben sich aus dieser Auswertung die Präzision und die Ausbeute sowie das zugehörige F_1-Maß; zusätzlich zur aggregierten Auswertung können wir jedoch die Maße getrennt nach der Anzahl an Methodenparametern ermitteln.

6.2.4.2 Bewertung der Ergebnisse mit vollständiger Ontologie

Die Auswertung der Ergebnisse zeigt, dass NLCI die Methodenaufrufe überwiegend korrekt erkennen kann und dabei wenige falsche Methodenaufrufe generiert: Von 625 Methodenaufrufen ermittelte NLCI 447 korrekt und nur 70 Aufrufe falsch, was einer Präzision von $86,5\%$ und einer Ausbeute von $71,5\%$ entspricht ($F_1 = 78,3\%$).

Tabelle 6.5: Evaluationsmetriken für die Methodenauswahl. Alle Angaben in %.

Metrik	0 Parameter	1 Parameter	2 Parameter	Gesamt
Präzision	85,7	86,1	100,0	86,5
Ausbeute	74,5	70,9	54,1	71,5
F_1-Maß	79,7	77,8	70,2	78,3

Tabelle 6.5 fasst die Ergebnisse zusammen; eine ausführliche Übersicht findet sich in Tabelle 6.6. Auffällig ist, dass die Präzision mit steigender Parameterzahl steigt, die Ausbeute hingegen fällt. Dies ist erwartungsgemäß, da es viel mehr Methoden ohne Parameter gibt, als Methoden mit zwei Parametern. Wird eine Methode mit zwei Parametern beschrieben, so ist dies meist eindeutig, was zur perfekten Präzision von 100% führt. Umgekehrt sind Beschreibungen für zwei Argumente immer länger und/oder komplizierter, als Beschreibungen ohne Argumente. Die Beschreibungen von Methodenaufrufen mit mehr Parametern können daher seltener erfolgreich analysiert werden können, was zur geringen Ausbeute von $54,1\%$ führt.

Die Analyse der Zwischenergebnisse, d.h. der annotierten Texte, ergab, dass viele der fehlenden und falsch generierten Methodenaufrufe direkt auf das Versagen des Zerteilers zurückzuführen sind: Verhältnismäßig häufig werden Verben nicht als Verb, sondern als Substantiv markiert, was eine weitere Verarbeitung für NLCI unmöglich macht. Markiert der Zerteiler ein anderes Wort im Satz als Verb, wird in der Folge nach einer anderen Aktion gesucht, als eigentlich gemeint war. Dies tritt sowohl bei einfachen Sätzen auf, als auch bei komplizierten Beschreibungen von Methodenaufrufen mit mehreren Parametern.

Beispiel

Das Problem der fehlerhaften Zerteilung tritt relativ häufig bei Sätzen auf, die der „einfachen“ Subjekt-Prädikat-Objekt-Struktur folgen und deren Prädikat dieselbe Form hat, wie ein englisches Substantiv: Sätze, die mit „The woman waves …“ beginnen, werden vom Zerteiler fast nie korrekt als „The/DT woman/nn waves/VBZ …“ markiert, sondern als „The/DT woman/NN waves/NNS …“, das Verb „waves“ also als Substantiv im Plural anstatt als Verb im Präsens. Im zugehörigen Syntaxbaum wird aus diesen drei Wörtern dann eine Nominalphrase, die meist alleine steht, weil kein Wort folgt, das als Verb interpretiert werden kann. Der Syntaxbaum ist dann unsinnig und nicht mehr zur Verarbeitung zu gebrauchen. Da die Abhängigkeitsgraphen aus Syntaxbäumen abgeleitet werden, können diese ebenfalls nicht mehr erfolgreich zur Analyse verwendet werden.

Tabelle 6.6: Detaillierte Auswertung der Methodenauswahl durch NLCI. Die Tabelle gibt an, wie viele Methoden mit 0, 1 bzw. 2 Parametern korrekt ermittelt wurden (✓), nicht ermittelt werden konnten (!) und falsch ermittelt wurden (×).

		0 Parameter				1 Parameter				2 Parameter			
Animation	Texte	Σ	✓	!	×	Σ	✓	!	×	Σ	✓	!	×
Alice	4	15	14	1	0	52	47	5	0	0	0	0	0
Cheerleader	11	74	32	42	16	36	17	19	3	0	0	0	0
Cowboy	8	42	30	12	2	7	6	1	0	2	2	0	0
Moon	3	9	9	0	0	32	25	7	9	2	1	1	0
Rabbit	22	103	96	7	13	144	103	41	12	2	0	2	0
Wizard	7	37	29	8	4	7	2	5	5	0	0	0	0
Woman	6	2	0	2	0	28	17	11	6	31	17	14	0
Gesamt	61	282	210	72	35	306	217	89	35	37	20	17	0

NLCI enthält zwar eine Komponente, die derartige Sätze als fehlerhaft markiert, wenn sie kein Verb enthalten, jedoch hat NLCI keine Handhabe, mit der Analyse eines solchen Satzes fortzufahren oder den Benutzer zu einer anderen Formulierung zu zwingen [Bes14]. Ist der Beschreibungssatz länger, so nimmt die Wahrscheinlichkeit zu, dass der Zerteiler ein anderes Wort als Prädikat auswählt. Dann versagt die Fehlererkennung und der resultierende Syntaxbaum wird als korrekt angesehen. Meist scheitert die Erkennung einer Methode dann an der Unsinnigkeit der Interpretation: Im obigen Beispiel würde nach einem Objekt gesucht werden, das „woman“ und „waves“ im Namen hat.

Die sinkende Präzision der generierten Methodenaufrufe mit weniger als zwei Parametern lässt sich dadurch erklären, dass diese weniger eindeutig sind. Es gibt mehr Methoden ohne Parameter als mit Parametern und da diese immer aufrufbar sind, kann NLCI einen Methodenaufruf immer generieren, wenn es in der Ontologie eine Methode findet, die keinen Parameter hat.

Vergleicht man die Ergebnisse in Tabelle 6.4 mit denen in Tabelle 6.5, so kann man einen Anstieg der Ausbeute und der Präzision beobachten: In der TOP-1-Auswertung zur Verknüpfung atomarer Sätze mit API-Aufrufen erreicht NLCI eine Präzision von 78,1%, mit der Methodenauswahl kann dieser Wert auf 86,5% gesteigert werden. Die Ausbeute bei der TOP-10-Auswertung ist ebenfalls geringer, jedoch ist die Differenz deutlich kleiner (70,0% statt 71,5%). Diese geringe Steigerung lässt sich dadurch begründen, dass die Methodenauswahl nicht auf die TOP-N angewiesen ist, sondern die gesamte Ergebnislis-

te zur Verfügung hat. Dennoch handelt es sich nur um eine Auswahl; was in der TOP-∞ nicht enthalten ist, kann auch nicht ausgewählt werden.

6.2.4.3 Bewertung der Ergebnisse mit eingeschränkter Ontologie

Unsere Erwartung, dass sich die Präzision von NLCI bei gleichbleibender Ausbeute durch das gezielte Zuschneiden der Ontologie steigern lässt, hat sich nicht erfüllt. Die Ergebnisse, die NLCI mit der eingeschränkten Ontologie erzielen kann, sind mit denen identisch, die im vorangegangenen Abschnitt diskutiert wurden, weswegen sie hier nicht noch einmal abgedruckt werden.

Bezogen auf die Präzision von NLCI kann dieses Ergebnis als positiv interpretiert werden, da die Präzision augenscheinlich nicht von der Größe der verwendeten Ontologie abhängt. Die Einschränkung, welche Methode im aktuellen Kontext (bspw. verfügbare Objekte, Argumente usw.) überhaupt ausführbar sind, führt in der Regel zur richtigen Auswahl; insbesondere wird aus der größeren Auswahl an Klassenkandidaten aufgrund der verwendeten Methoden der jeweils passende ausgewählt und kein „beinahe“ passender.

6.2.5 Korrektur der Reihenfolge

Die folgenden Abschnitte betrachten, wie erfolgreich die Korrektur der Reihenfolge ist, die in Abschnitt 5.3.2 vorgestellt wurde. Zunächst wird beschrieben, welche Texte zur Evaluation verwendet wurden und wie diese vorbereitet werden mussten. Der zweite Abschnitt präsentiert und diskutiert die Ergebnisse.

6.2.5.1 Vorbereitung der Evaluation

Die Textmenge für die Evaluation der zeitlichen Analyse ist nur eine Teilmenge des Korpus. Ein Großteil der Texte im Korpus verwenden keine Temporalausdrücke (TA) und schildern die Handlung sequentiell korrekt. In der Folge werden sie bei dieser Evaluation nicht betrachtet. Die Texte enthalten keine TA, da die Probanden bei der Texterfassung dazu angehalten wurden, keine zeitlichen Bezüge zu verwenden und die Animation in der Reihenfolge zu beschreiben, wie sie im Video zu sehen ist. Die vier verwendeten Texte der *Rabbit*-Animation enthalten nur wenige TA, die zudem keine Verschiebung von Aktionen nach sich ziehen. Die jeweils zehn Texte zu den beiden anderen Animationen, *Cheerleader* und *Dragon*, verwenden TA sehr häufig, da sie ausdrücklich für diese

Evaluation erstellt wurden. Die Probanden wurden daher ermuntert, TA zu verwenden. Insgesamt wurden für diese Evaluation 24 Texte herangezogen.

Um die Auswertung vornehmen zu können, erstellten wir Zeitachsen als Musterlösungen für jeden einzelnen Text. Wie in Abschnitt 6.2.3.1 bereits festgestellt wurde, beschreiben die Probanden nicht exakt die Animation, die ihnen gezeigt wurde. Zu den oben erwähnten fehlenden Aktionen kommt bei der Bewertung der zeitlichen Reihenfolge hinzu, dass einige Texte aufgrund der (absichtlich eingefügten) TA zwar die richtigen Aktionen beschreiben, diese jedoch in einer anderen Reihenfolge als in der Animation. Daher musste für jeden Text eine individuelle Musterlösung erstellt werden, die sich ausschließlich am jeweiligen Eingabetext orientiert. Diese Musterlösungen wurde mit den Ergebnissen von NLCI verglichen.

Zu dieser Auswertung des Analyseergebnisses wurden zudem etwaige Fehlerursachen untersucht und in zwei Kategorien eingeteilt: Die Auswertung der Programmausführung erlaubt eine Aussage darüber, ob ein TA von NLCI betrachtet wurde oder nicht. Die von NLCI betrachteten TA lassen sich weiter aufteilen in korrekt interpretierte und falsch interpretierte.

Wird ein TA nicht betrachtet (oder falsch aufgelöst), kann es passieren, dass viele oder gar alle Aktionen an der falschen Stelle auf der Zeitachse stehen: Aus diesem Grund ist die absolute Zahl falsch positionierter Aktionen keine hilfreiche Angabe zur Evaluation. Stattdessen bestimmen wir eine Editierdistanz: die Distanz ist die Anzahl der notwendigen Umordnungen, um eine fehlerhafte Zeitachse vollständig zu korrigieren.

Beispiel

Gegeben sei ein Text mit vier Aktionen a_i mit $i \in 0,\dots,3$. Im Text werden die Aktionen in der Reihenfolge der i genannt. Die korrekten Zeitachse sei $[a_3, a_0, a_1, a_2]$.

Wenn NLCI die (falsche) Zeitachse $[a_0, a_1, a_2, a_3]$ als Ergebnis liefert, so sind alle Aktionen an der falschen Stelle auf der Zeitachse; die absolute Fehlerzahl wäre dementsprechend vier. Verschiebt man die letztgenannte Aktion a_3 an den Anfang, so ergibt sich die gewünschte Zeitachse. Damit ergibt sich eine Editierdistanz von eins.

6.2.5.2 Bewertung der Ergebnisse

Tabelle 6.7 fasst die Ergebnisse der Auswertung zusammen: Nach der Animation und der Anzahl der verwendeten Texte enthält die Tabelle die Anzahl der enthaltenen Temporalausdrücke ($\sum$). Die folgenden Spalten enthalten die Anzahlen der erkannten TA ($\checkmark$), der fehlerhaft interpretierten TA ($\times$), der nicht ausgewerteten TA (!) und die Summe der Edi-

Tabelle 6.7: Evaluationsergebnisse für die Korrektur der Reihenfolge.

Animation	Texte	$\sum$TA	✓	×	!	↔
Cheerleader	10	81	69	5	7	6
Dragon	10	69	61	1	7	1
Rabbit	4	16	15	1	0	1
Gesamt	24	166	145	7	14	8
			87%	4%	8%	

tierdistanzen (↔). Insgesamt interpretiert NLCI 4% der TA falsch und wertet 8% der TA nicht aus.

Wie in der Tabelle zu sehen ist, machen die 21 Verarbeitungsfehler (sieben Fehlinterpretationen und 14 nicht betrachtete TA) insgesamt nur acht Umordnungen nötig. Das liegt daran, dass NLCI zunächst eine naive Zeitachse aufbaut, welche die Aktionen in der textuellen Reihenfolge enthält. Wenn ein TA (wie bspw. ein „then“ am Anfang eines Satzes) keine Auswirkung auf die Reihenfolge hat, so entsteht kein Fehler, wenn der TA nicht ausgewertet wird. Im Folgenden werden die Fehler detaillierter untersucht.

Drei der sieben Fehler erwachsen aus einer fehlerhaften Ausgabe des verwendeten Zerteilers; wenn der Zerteiler in diesen Fällen ein korrektes Ergebnis liefern würde, würde NLCI die korrekte Interpretation annotieren. Ein weiterer Fehler ist eine Anforderung, die NLCI nicht erfüllt: Ein Autor beschreibt, dass eine Aktion vor *und* nach einer anderen Aktion ausgeführt werden soll, d.h. wiederholt werden muss. NLCI betrachtet bei der zeitlichen Analyse jedoch lediglich Umordnungen auf der Zeitachse. Um die Beschreibung des Autors umzusetzen, müsste NLCI aufgrund des TA eine Aktion zwei mal auf der Zeitachse auftragen. Eine derartige Auflösung ist derzeit im Funktionsumfang von NLCI nicht enthalten. Die verbleibenden drei Fehler sind Interpretationsfehler, die aufgrund der verwendeten Regeln entstehen.

Die nicht betrachteten TA entfallen ebenfalls auf unterschiedliche Fehlerursachen: Ein TA wurde nicht betrachtet, bei dem eine Aktion zwischen zwei anderen eingefügt werden sollte. Hierbei handelte es sich jedoch nicht um eine Folge von Aktionen, sondern um eine Wiederholung: „Do a three times. After the second time, do b.“ NLCI betrachtet die dreifache Wiederholung von „do a“ als einzelne Aktion auf der Zeitachse. Um die Beschreibung des Autors umzusetzen, müsste NLCI diese Aktion aufbrechen und mehrfach auf der Zeitachse eintragen.

Die verbleibenden 13 TA, die von NLCI nicht betrachtet wurden, stammen aus Texten *eines* Probanden. Der Proband verstand es als Herausforderung, möglichst verschachtelte

Texte mit möglichst vielen TA zu verfassen. Dabei verwies er in TA sehr häufig auf andere Aktionen; da NLCI nur eine sehr eingeschränkte Heuristik zur Auflösung von Referenzen enthält, können viele der TA nicht aufgelöst werden. So lange der Proband auf eine Aktion mit genau derselben Wortfolge verwies, konnte der TA aufgelöst werden; paraphrasierte er die andere Aktion, dann scheitert NLCI. Interessanterweise trat dieses Problem häufig dann auf, wenn der Proband einen Satz mit „after" begann und sich dann auf die Aktion im direkt vorangegangenen Satz bezog. Da NLCI zunächst eine Zeitachse erzeugt, welche die Aktionen in der textuellen Reihenfolge enthält, führt es nicht zu einer fehlerhaften Zeitachse, wenn ein derartiger TA nicht aufgelöst werden kann.

Während der Auswertung stießen wir auf drei TA, die bei der Entwicklung übersehen wurden; sie wurden in die Liste der Schlüsselwörter aufgenommen und die Auswertung erneut durchgeführt. Durch diese TA wurden ursprünglich drei Formulierungen übersehen, die jedoch nur zu einer Umordnung geführt hatten; insofern wurde das Ergebnis durch die Aufnahme der Schlüsselwörter nicht stark beeinflusst.

NLCI kann in der aktuellen Fassung nicht immer die korrekte Reihenfolge herstellen. Insgesamt werden jedoch nur 8% der TA nicht aufgelöst und 145 der 152 erfassten TA werden korrekt aufgelöst. Viele der nicht aufgelösten TA führen nicht zu Fehlern, sodass NLCI 18 der 24 Zeitachsen vollständig korrekt rekonstruieren kann. Die fehlerhaften Zeitachsen können mit maximal zwei, meistens mit nur einer Umordnung vollständig korrigiert werden. Das größte Problem für NLCI stellt beim Auflösen der TA die fehlenden Korreferenzinformationen für Aktionen dar; wenn diese Informationen (derzeit manuell) im Text annotiert werden, kann NLCI diese TA korrekt auflösen. Eine automatische Korreferenzanalyse erscheint möglich und wird in Abschnitt 7.2 diskutiert.

6.2.6 Ermitteln von Kontrollstrukturen

Die folgenden Abschnitte betrachten die Erkennung von Kontrollstrukturen, wie sie in Abschnitt 5.3.3 vorgestellt wurde. Zunächst wird beschrieben, welche Texte zur Evaluation verwendet wurden und wie diese vorbereitet werden mussten. Der zweite Abschnitt präsentiert und diskutiert die Ergebnisse.

6.2.6.1 Vorbereitung der Evaluation

Für die Evaluation der Kontrollstrukturerkennung werden 52 Eingabetexte verwendet. Tabelle 6.8 gibt eine Übersicht über die sechs zugrundeliegenden Animationen. Sie benötigen eine überschaubare Anzahl von Objekten und Methodenaufrufen. Es werden unterschiedlich viele Kontrollstrukturen benötigt, um den Ablauf zu beschreiben. In Klammern

ist angegeben, wie viele verschiedene Kontrollstrukturen benötigt werden. In der letzten Spalte ist die Anzahl an Eingabetexten angegeben, die für die Evaluation verwendet wurden. Die Animationen *Beach*, *Farm* und *Moon* verwenden viele Kontrollstrukturen, die anderen Animationen eher wenige; *Rabbit* ist die einfachste Animation und kommt gänzlich ohne Kontrollstrukturen aus.

Wie bei den oben beschriebenen Evaluationen mussten die Musterlösungen für jeden einzelnen Eingabetext angelegt werden. Grundlage für die Musterlösungen war die Implementierung der entsprechenden Animation. Die Auswertung, welche Kontrollstrukturen korrekt identifiziert werden, wurde automatisiert.

Die Evaluation betrachtet drei verschiedene Szenarien (siehe Abschnitt 5.3.3). Der erste Durchlauf betrachtet eine realistische, fehlerbehaftete Eingabe und verwendet die linguistisch begründete Standardkonfiguration. Hierbei wurden die Ergebnisse der Vorverarbeitung so verwendet, wie sie von den NLP-Werkzeugen bereitgestellt werden. Daher ist dieser Durchlauf das kritischste Szenario der Evaluation.

Der zweite Durchlauf untersucht, ob die angepasste lockerere Konfiguration Fehler der NLP-Vorverarbeitung ausgleichen kann. Während der Entwicklung der Kontrollstrukturenerkennung fiel auf, dass viele Fehler aufgrund fehlerhafter Zwischenergebnisse entstehen, die von NLP-Werkzeugen stammen. Daher lässt die verwendete Konfiguration bspw. zu, dass Nomen und Nominalphrasen als Aktionen angesehen werden. Diese Anpassung trägt dem Umstand Rechnung, dass die Wortartmarkierung und der Zerteiler häufig Verben als Substantive klassifiziert. In der Folge sollte die Anzahl der nicht gefundenen Aktionen sinken; die Ausbeute der Analyse sollte steigen. Im Gegenzug werden unter Umständen Wörter als Aktionen betrachtet, die richtigerweise als Nomen klassifiziert wurden; daher sollte die Präzision entsprechend sinken. Dieses Szenario kann als realistisch betrachtet werden, da in der Realität Fehler in der Vorverarbeitung auftreten können und die verwendete Konfiguration diese bestmöglich auszugleichen versucht.

Die ersten beiden Szenarien erlauben eine Einschätzung, wie gut die Kontrollstrukturenerkennung mit den derzeit vorhandenen Werkzeugen funktioniert. Das dritte Szenario untersucht hingegen, wie gut die Ergebnisse werden können, wenn man von einer idealen Welt ausgeht, in der es keine Fehler in der Vorverarbeitung gibt. Für dieses Szenario wurden die Ergebnisse der Vorverarbeitung der Eingabetexte manuell aufbereitet und alle Fehler behoben. Die Bereinigung beschränkt sich auf die Texte der Animationen *Beach* und *Farm*, da sie die meisten Kontrollstrukturen aufweisen und die Fehlerkorrektur sehr aufwändig ist. Da es sich um insgesamt 28 der 52 Texte handelt, ist dennoch eine aussagekräftige Untersuchung möglich. Der dritte Durchlauf verwendet die Standardkonfiguration.

Tabelle 6.8: Eine Übersicht über die Animationen, die für die Evaluation der Kontrollstrukturerkennung verwendet wurden. Die Zahl in Klammern gibt an, wie viele verschiedene der sechs betrachteten Kontrollstrukturen verwendet wurden.

Animation	#Objekte	#Methoden	#Kontrollstrukturen	#Texte
Alice	3	23	7 (3)	4
Beachday	9	11	10 (5)	14
Cheerleader	2	11	1 (1)	3
Farm	9	15	10 (6)	14
Moon	3	25	10 (2)	3
Rabbit	3	15	0 (0)	14

6.2.6.2 Bewertung der Ergebnisse

Tabelle 6.9 fasst die Ergebnisse der drei Durchläufe zusammen. In der ersten Spalte ist angegeben, um welche Animation es sich handelt. Die Spalte $\sum$ gibt an, wie viele Annotationen erwartet werden; je nach Text ist diese Anzahl jedoch unterschiedlich, da die Texte nicht exakt die Animationen wiedergeben. Normalerweise wird für jedes Element in einer Kontrollstruktur eine Annotation benötigt. Die Spalte $\checkmark$ gibt an, wie viele Annotationen korrekt angelegt wurden; $\times$ gibt an, wie viele falsche Annotationen angelegt wurden. ! gibt an, wie viele Annotationen fehlen und $\leftrightarrow$ gibt an, wie viele Annotationen falsch platziert wurden. Fehlerhaft platzierte Annotationen kennzeichnen zwar die erwartete Kontrollstruktur, jedoch ist die Reihenfolge der Annotationen von Bedeutung, da sie bei der Quelltexterzeugung ausgewertet wird.

Beispiel

Gegeben sei ein Text, in dem drei Aktionen wiederholt ausgeführt werden. Dann benötigt jede Aktion eine Schleifen-Annotation, die angibt zu welcher Schleife die Aktion gehört.

Wenn NLCI die Schleife erkennt, jedoch nur zwei Aktionen kennzeichnet, ergeben sich folgende Anzahlen in der Tabelle: Insgesamt werden drei Annotationen benötigt ($\sum$); zwei Aktionen wurden korrekt markiert ($\checkmark$), eine wurde ausgelassen (!). Die Anzahl der Schleifendurchläufe wird als Attribut der Schleifen-Annotation gespeichert und benötigt daher keine weitere Annotation.

Betrachten wir zunächst die strenge, realistische Auswertung mit der Standardkonfiguration. Der erste Block von Tabelle 6.9 fasst die Ergebnisse zusammen. 82% der erwar-

Tabelle 6.9: Evaluationsergebnisse für die Erkennung der Kontrollstrukturen.

Animation	∑	✓	×	!	↔
Alice	43	36	0	7	0
Beachday	221	151	26	66	4
Cheerleader	26	11	4	15	0
Farm	308	287	6	20	1
Moon	47	27	19	19	1
Rabbit	150	137	22	13	0
Gesamt	795	649	77	140	6
		82%	10%	18%	1%

(a) Ergebnisse bei fehlerbehafteter Eingabe und Verwendung der Standardkonfiguration.

Animation	∑	✓	×	!	↔
Alice	43	34	4	9	0
Beachday	221	195	10	17	9
Cheerleader	26	16	4	10	0
Farm	308	295	9	12	1
Moon	47	27	24	19	1
Rabbit	150	138	27	12	0
Gesamt	795	705	78	79	11
		89%	10%	10%	1%

(b) Ergebnisse bei fehlerbehafteter Eingabe und Verwendung der angepassten Konfiguration.

Animation	∑	✓	×	!	↔
Beachday	221	215	2	6	0
Farm	308	298	3	10	0
Gesamt	529	513	5	16	0
		97%	1%	3%	0%

(c) Ergebnisse bei bereinigter Eingabe und Verwendung der Standardkonfiguration.

teten Annotationen werden von NLCI korrekt eingefügt. Lediglich 1% der Annotationen steht an der falschen Stelle. Interessant ist das Verhältnis von falschen zu fehlenden Annotationen: NLCI macht mit 10% verhältnismäßig wenige Fehler, jedoch führen die fehlerhaften Eingabedaten dazu, dass 18% der erwarteten Annotationen fehlen. Das liegt hauptsächlich daran, dass der Wortartmarkierer bzw. der Zerteiler in den Eingabetexten Verben häufig als Nomen klassifiziert.

Die angepasste Konfiguration gleicht einige dieser Fehler aus. Entgegen der Vermutung von oben nimmt die Anzahl der falschen Annotationen lediglich um eine Annotation zu. Die Anzahl der fehlenden Annotationen kann hingegen auf 10% gesenkt werden: Diese Variante erreicht insgesamt eine deutlich geringere Fehlerrate und erzeugt 89% der erwarteten Annotationen korrekt.

Im idealen Fall ohne Eingabefehler erzielt NLCI ein beinahe perfektes Ergebnis: Nur 3% der Annotationen fehlen und nur fünf von 518 Annotationen sind falsch. Die Ergebnisse der *Beach*-Animation konnten deutlich verbessert werden, da in den zugehörigen Texten viele Zerteilungsfehler aufgetreten sind. Nachdem die Eingabefehler behoben wurden, sind 97% der Annotationen korrekt und NLCI nimmt nur zwei falsche Annotationen vor. Insbesondere die Anzahl der fehlenden Annotation ist sehr stark auf nur sechs gefallen. Die unbearbeiteten Texte zur *Farm*-Animation enthalten deutlich weniger Fehler, weswegen die entsprechenden Ergebnisse ohnehin schon vergleichsweise gut waren.

Die letzte Auswertung unterstreicht, dass das Verfahren zu Kontrollstrukturerkennung sehr gut funktioniert, wenn die Vorverarbeitung korrekt verläuft. Verläuft sie weniger gut, können einige Fehler über eine angepasste Konfiguration ausgeglichen werden.

6.2.7 Programmsynthese

Eine Alice-Welt-Datei ist eine vollständige Serialisierung aller in der Welt verwendeten Elemente; das bedeutet, dass ein Code-Generator für Alice sämtliche Modelle und deren Methoden (die über verschiedene Quelldateien verteilt sein können) einsammeln und zusammen mit dem benötigten Quelltext abspeichern muss. Zur Vereinfachung erzeugt der Code-Generator hierbei eine Datei, die für ein Modell alle Methoden usw. enthält, die in der Ontologie verzeichnet sind, unabhängig von ihrer Verwendung im Eingabetext.

Der Alice-Welt-Generator erhält von NLCI das aus dem annotierten Text erzeugte AST-Modell zu einer Beschreibung und überführt dieses in ein Alice-Pendant. Hierbei werden alle erkannten Methoden und Kontrollstrukturen direkt übernommen und entsprechend der erkannten Reihenfolge in das Alice-Code-Modell überführt. Anschließend wird das Alice-Code-Modell nur noch serialisiert.

Der Code-Generator für Alice wurde im Rahmen der Evaluation der Methodenauswahl ebenfalls mit überprüft. Der Generator überführte dabei alle betrachteten Eingabetexte in Alice-Welt-Dateien, die sich in Alice laden ließen und die den annotierten Quelltext widerspiegeln.

Bei der Generierung der Welt-Dateien traten nur zwei Fehler auf, die jedoch nicht die Erzeugung der betroffenen Welt-Dateien verhinderten: Beide Male trat ein Fehler beim Einsammeln der benötigten Modelle auf, der augenscheinlich durch eine Charakter-Datei ausgelöst wurde, die sich mit dem ZIP-Algorithmus nicht fehlerfrei entpacken ließ. Beim Entpacken konnten einige Dateien nicht extrahiert werden, die scheinbar die Funktion der Modelle nicht beeinträchtigt.

Eine Eigenheit von Alice wurde im Code-Generator nicht betrachtet: Da in Alice-Welten alle Objekte global zugreifbar sind, kann in Methoden eines Objektes direkt auf andere Objekte in der Welt zugegriffen werden (ohne diese als Argumente an die Methode zu übergeben). Enthält ein Modell eine derartige Methode und das behandelte andere Objekt ist nicht Teil der Welt, so wird es nicht in die Welt importiert. Gemessen an der Beschreibung des Probanden ist dies auch richtig – wird das andere Objekt *nie* benötigt, so muss es auch nicht in die generierte Welt aufgenommen werden. Lädt man eine Welt in Alice, die einen solchen Fall enthält, so meldet Alice beim Laden eine Referenz, die nicht aufgelöst werden kann und löscht den Zugriff auf das referenzierte Objekt nach Rückfrage. Dieser Eingriff wird in der Bewertung nicht als schädlich angesehen, da die Welt vorher wie nachher die Modelle und Methoden enthält, die NLCI erkannt hat. Ebenso lässt sich die Welt problemlos ausführen, sobald die Referenz von Alice entfernt wurde. In einem Szenario, in dem Methoden auf globale Objekte zugreifen müssen und diese Objekte Teil des Generats sein müssen, müsste die Benutzt-Relation beim Erstellen der NLCI-Ontologie erfasst und bei der Code-Erzeugung ausgewertet werden.

6.3 Zusammenfassung

Dieses Kapitel zeigte anhand zweier Fallstudien, dass NLCI in der Praxis eingesetzt werden kann. Die benötigte NLCI-Ontologie ist einfach zu erstellen und kann für kleine Systeme sogar manuell aufgebaut werden. Die Vorverarbeitung der API ist hilfreich und scheint dem Zweck innerhalb von NLCI gerecht zu werden.

Die Fallstudie mit openHAB zeigte, dass NLCI erfolgreich in einem Szenario eingesetzt werden kann, in dem ein Benutzer kurze Befehle an ein System gibt. Die Erkennung der in openHAB verwendeten Elemente erfolgte zufriedenstellend, ebenfalls ihre Kombination zu Methodenaufrufen.

Die zweite Fallstudie zu Alice zeigte, dass NLCI auch dazu verwendet werden kann, kurze skriptartige Programme als Aktionsfolgen zu beschreiben. Hierbei können die Eingaben der Benutzer nicht nur auf einfache Methodenaufrufe abgebildet werden, sondern auch auf Kontrollstrukturen. Beschreibt ein Benutzer eine Aktionsfolge nicht chronologisch korrekt, kann die beabsichtigte Reihenfolge der Aktionen in den überwiegenden Fällen rekonstruiert werden.

Die Code-Erzeugung, die am Ende des NLCI-Prozesses steht, kann komfortabel durchgeführt werden. Der Code-Generator für Alice-Welt-Dateien benötigt keine Kenntnis über die Art und Struktur der Annotationen im Eingabetext, da er von NLCI eine AST-ähnliche Darstellung erhält, die sich einfach in einen AST für die Zielplattform überführen lässt.

Die Evaluationen haben aber auch gezeigt, dass NLCI von der Qualität der computerlinguistischen Vorverarbeitung der Eingabetexte abhängt. Da die Analysen auf den Syntaxbäumen oder den Abhängigkeitsgraphen basieren, sind sie empfindlich gegenüber falschen Ergebnissen. Insbesondere die Auswertung der Methodenauswahl hat jedoch gezeigt, dass die NLCI-Ontologie den Analysen so viele Informationen über die Ziel-API geben kann, dass diese verhältnismäßig wenige Fehler produzieren. Die meisten Fehler sind nicht erkannte Aktionen oder Reihenfolgeverschiebungen; bei der Kontrollstrukturerkennung verhält es sich ebenso.

Bei der Interpretation der Ergebnisse muss berücksichtigt werden, dass die computerlinguistischen Werkzeuge nicht auf den Einsatzzweck vorbereitet wurden. Insofern sind die Ergebnisse als untere Schranke zu betrachten. Insbesondere das Fehlen von Aktionen aufgrund einer mangelhaften Verb-Erkennung scheint sehr einfach durch ein Training des Zerteilers lösbar zu sein.

Kapitel 7

Fazit und Ausblick

„The basic concept of allowing a person to communicate with a computer in his natural language will surely take many many years, and may exceed the lifetime of some of us. This does not mean that it is not a goal worth striving for.“

Jean E. Sammet, 1966

In dieser Arbeit wurde eine Architektur zur Erzeugung von Programmen durch die Analyse natürlichsprachlicher Texte vorgestellt. Die Architektur entkoppelt zunächst das Wissen über die anzusteuernde API, d.h. das Domänenwissen, von der Textanalyse und sieht vor, dass das Domänenwissen in Form einer Ontologie zur Verfügung gestellt wird. So kann erreicht werden, dass die Textanalysen wiederverwendbar sind, auch wenn sich die Ziel-API oder die Ziel-Programmiersprache ändert.

Eine weitere Eigenschaft der vorgestellten Architektur ist es, dass die Textanalysen durch eine Fließbandarchitektur ebenfalls voneinander entkoppelt sind. So müssen die computerlinguistischen Vorverarbeitungen durch Wortartmarkierer, Zerteiler oder ähnliches nur ein mal durchgeführt werden. Die Ergebnisse der (Vor-)Verarbeitung stehen dann allen folgenden Analysestufen zur Verfügung. Eine Stufe kann die Ergebnisse einer vorangehenden Stufe verwenden, interpretieren und sogar korrigieren oder erweitern. Die vorgestellten Analysen sind zwar als domänenunabhängige Module entworfen worden, jedoch ist es möglich, domänenspezifische Analysen in das Fließband einzuhängen, um so die Leistungsfähigkeit des Systems zu verbessern.

Die prototypische Implementierung der Architektur, der Natural Language Command Interpreter (NLCI), gibt eine Struktur für die benötigte Ontologie vor, die auf objektorientierte Systeme zugeschnitten ist und einfach gefüllt und erweitert werden kann. NLCI führt zudem die folgenden Analysen durch:

1. Vorverarbeitung der Ontologie. Die dem System zur Verfügung gestellte Ontologie wird auf den Einsatz vorbereitet und sprachlich angereichert. So werden bspw. Bezeichner aufgetrennt, wenn sie aus mehreren Wörtern bestehen und Synonyme aus WordNet hinzugefügt.

2. Vorverarbeitung des Eingabetextes mit computerlinguistischen Standardwerkzeugen. Eingesetzt werden der Wortartmarkierer und der Zerteiler aus dem Paket CoreNLP der Computerlinguisten aus Stanford.

3. Verknüpfung des Textes mit der API. Zu den Textelementen (z.B. Subjekte, Objekte, Prädikate usw.) werden entsprechende API-Elemente ermittelt und mithilfe einer Bewertungsfunktion sortiert.

4. Auswahl von Methodenargumenten und aufzurufender Methode. Basierend auf der zweiten Analyse wählt NLCI in diesem Schritt Argumente für Methodenaufrufe in den Sätzen aus und bestimmt, welche Methodenaufrufe im Kontext des gesamten Textes aufgerufen werden sollen.

5. Korrektur der Reihenfolge. Ist die Beschreibung einer Aktionsfolge nicht chronologisch, so ermittelt NLCI die gewünschte Reihenfolge aus dem Text.

6. Erkennung von Kontrollstrukturen. Beschreibungen von Aktionsfolgen enthalten selten explizite Kontrollstrukturen wie Schleifen oder Ähnliches. NLCI leitet daher Kontrollstrukturen aus den sprachlichen Entsprechungen ab (z.B. Schleifen aus Aktionen, die auf einer Menge von Objekten ausgeführt werden).

Die Fallstudien mit NLCI haben gezeigt, dass die vorgeschlagene Architektur und die entworfenen Analysen dazu geeignet sind, aus natürlichsprachlichen Beschreibungen oder Befehlen Skripte bzw. API-Aufrufe zu erzeugen. Die Auswertung der Ergebnisse hat aber auch gezeigt, wo die größte Schwäche von NLCI liegt: Fehler, die von den computerlinguistischen Analysen in der Vorverarbeitung gemacht werden, können nur manchmal entdeckt und nicht ausgeglichen werden.

Die Vorverarbeitungsstufen wurden im Rahmen der Fallstudie nicht trainiert oder anderweitig auf die Eingabetexte zugeschnitten, sodass bei der Bewertung davon ausgegangen werden kann, dass die Ergebnisse eine untere Leistungsschranke darstellen. Die Probleme, die häufig auftraten sind allesamt einfacherer Natur, die sich entweder mit weiteren Vorbereitungsstufen oder dem Training der verwendeten Programme beheben lassen sollten. Kann die Performanz der Vorverarbeitung gesteigert werden, profitieren

automatisch alle nachfolgenden Stufen von der Verbesserung, ohne dass diese angepasst werden müssen.

Der folgende Abschnitt diskutiert ob die in Abschnitt 1.5 gesteckten Ziele erreicht und ob Belege für die aufgestellten Thesen gefunden werden konnten. Der zweite Abschnitt beschreibt mögliche zukünftige Forschungsthemen, die sich aus den Ergebnissen der vorliegenden Arbeit ergeben.

7.1 Die Thesen dieser Arbeit

Die vorliegende Arbeit suchte anhand der Fallstudien mit NLCI nach Belegen für die in Abschnitt 1.5 formulierten Thesen.

These 1: Die vorgestellte Architektur kann genutzt werden, um natürlichsprachlich ausgedrückte, imperative Programme in Quelltext zu übersetzen. Die beiden Fallstudien belegen, dass NLCI sowohl für openHAB als auch für Alice in der Lage ist, die natürlichsprachlichen Beschreibungen in Quelltext zu übersetzen. Auch wenn die Programmsynthese nicht perfekt funktioniert, so erzeugt NLCI selten falsche Methodenaufrufe sondern scheitert aufgrund der Vorverarbeitung an der Erkennung der Aktionsbeschreibung.

These 2: Es ist nicht notwendig, die zulässige Eingabesprache stark einzuschränken; die Ergebnisse heutiger NLP-Analysen sind als Grundlage gut genug. NLCI kann Imperative und beschreibende Sätze (Aussagesätze) verarbeiten, solange diese richtig von den verwendeten NLP-Werkzeugen verarbeitet werden. Dabei kommt es NLCI nicht darauf an, ob die Sätze im Aktiv oder Passiv geschrieben sind und ob sie Nebensätze enthalten. Da viele Analysen auf den typisierten Abhängigkeitsgraphen basieren, ist die Reihenfolge der Wörter im Satz ebenfalls unerheblich (bspw. kann vom Standardmuster „Subjekt, Prädikat, Objekt“ abgewichen werden).

Zur Evaluation von NLCI wurden Eingabetexte verwendet, die von Probanden geschrieben wurden. Den Probanden wurden nicht vorgeschrieben, wie sie formulieren sollten, dennoch wählten sie verhältnismäßig einfache Formulierungen. Dieser Umstand liegt zum Teil an der gewählten Aufgabenstellung: Dadurch, dass wir den Probanden Animationen zeigten, deren Aktionsfolgen sie beschreiben sollten, schränkten wir die Freiheit der Probanden durchaus ein. Eine derartige Einschränkung ist jedoch nicht schädlich, da jedes System, das von einem Endanwender mit NLCI programmiert werden soll bzw.

kann, einen eingeschränkten Funktionsumfang hat. Der überwiegende Teil der Eingabesätze kann mit den verfügbaren NLP-Analysen korrekt verarbeitet werden; treten Fehler auf, ließen sie sich häufig auf mangelndes Training der verwendeten NLP-Analysen zurückführen, nicht auf systematische oder konzeptionelle Unmöglichkeit.

These 3: Das in der Architektur vorgesehene API-Modell für das Domänenwissen kann ausgetauscht und so die Domäne gewechselt werden. Die Fallstudien zeigen, dass die entworfenen Analysen mit Beschreibungen für beide Szenarien umgehen können. Die nötigen Ontologien konnten ohne Anpassungen an der Vorlage von NLCI erstellt werden.

Waren Zusatzinformationen vorhanden, konnte die Vorlage so erweitert werden, dass die Analysestufen davon nichts merkten: Zum Beispiel unterteilt Alice Methoden in *Methoden* ohne Rückgabewert und *Funktionen* mit Rückgabewert. Diese Unterscheidung wurde in der Ontologie für Alice als Unterklassen der vorgesehenen Methodenklasse modelliert. Da die verwendete Inferenzmaschine die Vererbungsbeziehungen auflöst, fällt diese Verfeinerung bei den NLP-Analysen (z.B. der API-Verknüpfung) nicht auf.

Die NLCI-Ontologie wurde in beiden Fällen von einem eigens entwickelten Ontologie-Generator aufgebaut, für dessen Implementierung die jeweilige Ziel-API analysiert werden musste. Im Fall von openHAB war dies sehr einfach möglich, da die API nur wenige Methoden ausführen kann; diese wurden manuell erfasst. Die verfügbaren Klassen konnten aus der openHAB-Konfigurationsdatei eingelesen werden, was einfach von statten ging. Der Ontologie-Generator für Alice ist etwas umfangreicher, da die komplizierten Dateiformate von Alice analysiert werden mussten. Ausgehend von dieser Analyse war es ebenfalls einfach, den Generator zu implementieren.

These 4: Nicht-Sequenzialität der Beschreibung schadet nicht. Die Evaluation der Reihenfolgenkorrektur hat gezeigt, dass chronologisch nicht in der richtigen Reihenfolge genannte Aktionen erkannt werden können, wenn die Beschreibung Temporalausdrücke enthält. NLCI ist häufig in der Lage, die gewünschte Reihenfolge zu rekonstruieren und so eine korrekte Aktionsfolge zu erstellen.

These 5: Kontrollstrukturen können aus natürlichsprachlichen Beschreibungen synthetisiert werden. Die Evaluation der Kontrollstrukturanalyse hat gezeigt, dass man den Benutzern eines natürlichsprachlichen Programmiersystems nicht vorschreiben muss, wie einfache Kontrollstrukturen diktiert werden müssen. Wie einige verwandte Arbeiten

nahelegten, lassen sich viele Kontrollstrukturen aus natürlichsprachlichen Formulierungen ableiten und entsprechend generieren.

NLCI erkennt einfache Wiederholungen (*loop*), *while*-Schleifen und kann *foreach*-Schleifen erkennen (solange die Gruppe der Objete erkannt werden kann, über die iteriert wird). NLCI erkennt nicht nur diese Schleifen, sondern auch einfache Arten von Parallelität: Aktionen können als gleichzeitig ablaufend erkannt werden (*do together*) und NLCI erkennt eine parallele Variante der *foreach*-Schleife.

Bedingte Anweisungen können von NLCI noch nicht umgesetzt werden. Eine mögliche Textanalyse für Bedingungen wurde in der vorliegenden Arbeit sowie in verwandten Arbeiten skizziert, jedoch nicht in NLCI implementiert.

Programmieren in natürlicher Sprache Dem übergeordneten Ziel, Benutzern zu ermöglichen in natürlicher Sprache zu programmieren, ist NLCI ein großes Stück näher gekommen. Für das betrachtete Einsatzszenario zeigen die obigen Belege für die Thesen, dass das angestrebte Ziel in den betrachteten Kontexten erreicht werden konnte. Die Evaluation des Prototyps anhand zweier unterschiedlicher Systeme (openHAB und Alice) hat gezeigt, dass NLCI flexibel genug ist, um in diesen stark unterschiedlichen Einsatz-Szenarien verwendet zu werden.

Das Ziel einer einfachen Domänenakquise konnte erreicht werden: Die NLCI-Ontologie für openHAB und Alice können fast vollständig automatisch erzeugt werden; Ontologien für kleine Systeme können sogar von Hand erstellt werden.

Betrachtet man das Ziel, Methodenaufrufe aus Texteingaben zu generieren, so wurde auch dieses erreicht: NLCI ermittelt in vielen Fällen die gewünschten Methodenaufrufe ($F_1 = 78,3\%$). Die aufwändigere Korrektur der Reihenfolge funktioniert ebenfalls zufriedenstellend, 86% der Temporalausdrücke können korrekt ausgewertet werden und ein großer Teil der übrigen führt nicht zu Fehlern bei der Feststellung der gewünschten Reihenfolge. Die Synthese der betrachteten Kontrollstrukturen rundet die Fähigkeiten von NLCI ab, die in 89% der Fälle funktioniert; geht man von einer korrekten Vorverarbeitung aus, kann die Erkennungsrate sogar auf 97% gesteigert werden.

7.2 Zukünftige Arbeiten

Trotz – oder aufgrund – der erfreulichen Ergebnisse von NLCI in dieser Arbeit ergeben sich weitere Forschungsfragen, die untersucht werden müssen. Die folgenden Abschnitte skizzieren mögliche nächste Schritte auf dem Weg zu einem noch besseren, interaktiven natürlichsprachlichen Programmiersystem.

Korreferenzauflösung für Aktionen Die Arbeiten zur Korrektur der Reihenfolge und der Erkennung der Kontrollstrukturen haben gezeigt, dass es kein (gut funktionierendes) Verfahren gibt, das Korreferenzketten für Aktionen aufbauen kann. Ansätze aus dem Bereich der Auflösung von Personalpronomina könnten weiterentwickelt werden, um tatsächlich identische (nicht gleiche) Aktionen in einer Aktionsfolge zu ermitteln [Nik15]. Wird mehrfach beschrieben, dass eine Aktion (vom selben Objekt) ausgeführt wird und werden diese Aktionen in der Beschreibung mit Temporalausdrücken verknüpft, so ist es sehr wahrscheinlich, dass es sich nicht um Wiederholungen sondern um Referenzen handelt. Bei dieser Auflösung müssen auch Paraphrasierungen oder synonyme Begriffe betrachtet werden. Eine Wissensdatenbank wie die NLCI-Ontologie sollte hierbei unterstützend wirken können, da sie hilft, verschiedene Aktionswörter auf denselben Methodenaufruf abzubilden.

Modellieren von Vor- und Nachbedingungen Die NLCI-Ontologie modelliert derzeit weder Vor- oder Nachbedingungen noch Invarianten. Wären diese Informationen verfügbar, könnten sie für die Sicherstellung der korrekten Reihenfolge genutzt werden. Zudem könnte aus einer unterbrochenen Kette von Vor- und Nachbedingungen zu einer Aktionsfolge geschlossen werden, dass eine Aktion fehlt und an welcher Stelle. Ob dieses zusätzliche Wissen in der Praxis verfügbar und nutzbar ist, muss untersucht werden.

Analyse gesprochener Sprache NLCI beschränkt sich auf die Analyse geschriebener Sprache. Das Ermöglicht den Einsatz computerlinguistischer Analysen, wie bspw. einem Zerteiler. Erhält ein Prozess als Eingabe jedoch keinen Text, sondern nur noch eine (von einem automatischen Spracherkenner (ASR) erzeugte) Folge von Wörtern, so können Zerteiler selten einen korrekten Syntaxbaum ableiten.

Betrachtet man einen von einem ASR erzeugten Wortstrom, so scheinen sich aktive Ontologien sehr gut für die Befehlserkennung zu eignen [Bel14]. Aktive Ontologien sind jedoch dafür entwickelt worden, (kurze) einzelne Befehle entgegenzunehmen und direkt umzusetzen; zudem müssen aktive Ontologien derzeit für jeden Anwendungsfall inkl. der Analysen manuell erstellt werden. Ob sie sich eignen, um einen skriptartigen Ablauf zu programmieren ist derzeit unbekannt.

Eine andere Herangehensweise an die Herausforderungen gesprochener Sprache ist die, die Weigelt und Tichy verfolgen [WT15]. Sie beschreiben ein System, das ähnlich wie NLCI mit dem Domänenwissen konfiguriert wird. Es ist jedoch nicht als Fließband aufgebaut, sondern als Agentensystem, wobei die Agenten den Verarbeitungsstufen von NLCI entsprechen. Erste Ergebnisse zeigen hier, dass Wortartmarkierer auf Wortfolgen

einigermaßen gute Ergebnisse erzielen können [Koc15] und auch die Erkennung von Aktionen in Transkriptionen gesprochener Sprache möglich ist.

Anwendungsgebiete Anwendungsgebiete für NLCI ergeben sich in vielen Bereichen. Mit den Analysen könnten Arbeitsanweisungen für (Haushalts-)Roboter formuliert werden und die Analyse paralleler Aktionsfolgen könnten für die Modellierung von Arbeitsabläufen genutzt werden. Zudem ist es denkbar, dass alle Programme, die derzeit über eine grafische Oberfläche gesteuert werden, sprachliche Befehle empfangen.

Eine weitere aussichtsreiche Anwendung ist das Erzeugen von Testfällen durch Endanwender bzw. Mitarbeiter von Fachabteilungen. Bereits heute können technisch wenig bis nicht versierte Anwender, die jedoch über domänenspezifisches Fachwissen verfügen, Testfälle in kontrollierten natürlichen Sprachen formulieren. Techniken, die NLCI einsetzt, oder NLCI als Ganzes könnten dazu verwendet werden, einige oder alle kontrollierte Aspekte dieser Sprachen aufzuweichen oder gänzlich natürlich zu machen. Eine Abbildung eines Testtreibers und einer Anwendung in der NLCI-Ontologie erscheint umsetzbar; damit ist die Erzeugung von Testfällen auf Anwendungsebene „nur noch“ eine Formulierung von Aktionsfolgen.

Anhang A

Details zur openHAB-Fallstudie

A.1 Konfiguration des Demo-Haushalts

```
Group All
Group gGF       (All)
Group gFF       (All)
Group gC        (All)
Group Outdoor   (All)
Group Shutters  (All)
Group Weather   (All)
Group Status    (All)

Group GF_Living     "Living Room"   <video>     (gGF)
Group GF_Kitchen    "Kitchen"       <kitchen>   (gGF)
Group GF_Toilet     "Toilet"        <bath>      (gGF)
Group GF_Corridor   "Corridor"      <corridor>  (gGF)

Group FF_Bath       "Bathroom"      <bath>      (gFF)
Group FF_Office     "Office"        <office>    (gFF)
Group FF_Child      "Child's Room"  <boy1>      (gFF)
Group FF_Bed        "Bedroom"       <bedroom>   (gFF)
Group FF_Corridor   "Corridor"      <corridor>  (gFF)

/* active groups */
Group:Switch:OR(ON, OFF)        Lights      "All Lights [(%d)]"
                  (All)
Group:Switch:OR(ON, OFF)        Heating     "No. of Active Heatings [(%d)]"     <
    heating>       (All)
Group:Number:AVG                Temperature "Avg. Room Temperature [%.1f C]"    <
    temperature>   (Status)
Group:Contact:OR(OPEN, CLOSED)  Windows     "Open windows [(%d)]"               <
    contact>       (All)

/* Lights */
Dimmer Light_GF_Living_Table        "Table"         (GF_Living, Lights)
Switch Light_GF_Corridor_Ceiling    "Ceiling"       (GF_Corridor, Lights)
Switch Light_GF_Kitchen_Ceiling     "Ceiling"       (GF_Kitchen, Lights)
Switch Light_GF_Kitchen_Table       "Table"         (GF_Kitchen, Lights)
Switch Light_GF_Corridor_Wardrobe   "Wardrobe"      (GF_Corridor, Lights)
```

```
Switch Light_GF_Toilet_Ceiling      "Ceiling"       (GF_Toilet, Lights)
Switch Light_GF_Toilet_Mirror       "Mirror"        (GF_Toilet, Lights)

Switch Light_FF_Bath_Ceiling        "Ceiling"       (FF_Bath, Lights)
Switch Light_FF_Bath_Mirror         "Mirror"        (FF_Bath, Lights)
Switch Light_FF_Corridor_Ceiling    "Corridor"      (FF_Corridor, Lights)
Switch Light_FF_Office_Ceiling      "Ceiling"       (FF_Office, Lights)
Switch Light_FF_Child_Ceiling       "Ceiling"       (FF_Child, Lights)
Switch Light_FF_Bed_Ceiling         "Ceiling"       (FF_Bed, Lights)

Switch Light_C_Corridor_Ceiling     "Ceiling"       (gC, Lights)
Switch Light_C_Staircase            "Staircase"     (gC, Lights)
Switch Light_C_Washing_Ceiling      "Washing"       (gC, Lights)
Switch Light_C_Workshop             "Workshop"      (gC, Lights)

Switch Light_Outdoor_Garage         "Garage"        (Outdoor, Lights)
Switch Light_Outdoor_Terrace        "Terrace"       (Outdoor, Lights)
Switch Light_Outdoor_Frontdoor      "Frontdoor"     (Outdoor, Lights)

/* Heating */
Switch Heating_GF_Corridor  "GF Corridor"   <heating>   (GF_Corridor, Heating)
Switch Heating_GF_Toilet    "Toilet"        <heating>   (GF_Toilet, Heating)
Switch Heating_GF_Living    "Livingroom"    <heating>   (GF_Living, Heating)
Switch Heating_GF_Kitchen   "Kitchen"       <heating>   (GF_Kitchen, Heating)

Switch Heating_FF_Bath      "Bath"          <heating>   (FF_Bath, Heating)
Switch Heating_FF_Office    "Office"        <heating>   (FF_Office, Heating)
Switch Heating_FF_Child     "Child's Room"  <heating>   (FF_Child, Heating)
Switch Heating_FF_Bed       "Bedroom"       <heating>   (FF_Bed, Heating)

/* Rollershutters */
Switch Shutter_all (Shutters)

Rollershutter Shutter_GF_Toilet         "Toilet"        (GF_Toilet, Shutters)
Rollershutter Shutter_GF_Kitchen        "Kitchen"       (GF_Kitchen, Shutters)
Rollershutter Shutter_GF_Living         "Livingroom"    (GF_Living, Shutters)

Rollershutter Shutter_FF_Bed            "Bedroom"       (FF_Bed, Shutters)
Rollershutter Shutter_FF_Bath           "Bath"          (FF_Bath, Shutters)
Rollershutter Shutter_FF_Office_Window  "Office Window" (FF_Office, Shutters)
Rollershutter Shutter_FF_Office_Door    "Office Door"   (FF_Office, Shutters)

/* Indoor Temperatures */
Number Temperature_GF_Corridor  "Temperature [%.1f C]"  <temperature>   (
    Temperature, GF_Corridor)
Number Temperature_GF_Toilet    "Temperature [%.1f C]"  <temperature>   (
    Temperature, GF_Toilet)
Number Temperature_GF_Living    "Temperature [%.1f C]"  <temperature>   (
    Temperature, GF_Living)
Number Temperature_GF_Kitchen   "Temperature [%.1f C]"  <temperature>   (
    Temperature, GF_Kitchen)
Number Temperature_FF_Bath      "Temperature [%.1f C]"  <temperature>   (
    Temperature, FF_Bath)
Number Temperature_FF_Office    "Temperature [%.1f C]"  <temperature>   (
```

```
    Temperature, FF_Office)
Number Temperature_FF_Child      "Temperature [%.1f C]"  <temperature>   (
    Temperature, FF_Child)
Number Temperature_FF_Bed        "Temperature [%.1f C]"  <temperature>   (
    Temperature, FF_Bed)

/* Windows */
Contact Window_GF_Frontdoor      "Frontdoor [MAP(en.map):%s]"        (GF_Corridor,
     Windows)
Contact Window_GF_Kitchen        "Kitchen [MAP(en.map):%s]"          (GF_Kitchen,
    Windows)
Contact Window_GF_Living         "Terrace door [MAP(en.map):%s]"     (GF_Living,
    Windows)
Contact Window_GF_Toilet         "Toilet [MAP(en.map):%s]"           (GF_Toilet,
    Windows)

Contact Window_FF_Bath           "Bath [MAP(en.map):%s]"             (FF_Bath,
    Windows)
Contact Window_FF_Bed            "Bedroom [MAP(en.map):%s]"          (FF_Bed,
    Windows)
Contact Window_FF_Office_Window "Office Window [MAP(en.map):%s]"    (FF_Office,
    Windows)
Contact Window_FF_Office_Door    "Balcony Door [MAP(en.map):%s]"     (FF_Office,
    Windows)

Contact Garage_Door              "Garage Door [MAP(en.map):%s]"      (Outdoor,
    Windows)

Group Weather_Chart                                                  (Weather)
Number Weather_Temperature       "Outside Temperature [%.1f C]"  <temperature> (
    Weather_Chart) { http="<[http://weather.yahooapis.com/forecastrss?w=638242&u=
    c:60000:XSLT(yahoo_weather_temperature.xsl)]" }
Number Weather_Humidity          "Outside Humidity [%.1f %%]"    <temperature> (
    Weather) { http="<[http://weather.yahooapis.com/forecastrss?w=638242&u=c
    :60000:XSLT(yahoo_weather_humidity.xsl)]" }
Number Weather_Humidex           "Humidex [SCALE(humidex.scale):%s]"            (
    Weather)
Number Weather_Temp_Max          "Todays Maximum [%.1f C]"   <temperature> (
    Weather_Chart)
Number Weather_Temp_Min          "Todays Minimum [%.1f C]"   <temperature> (
    Weather_Chart)
Number Weather_Chart_Period      "Chart Period"
DateTime Weather_LastUpdate      "Last Update [%1$ta %1$tR]" <clock>

/* NTP binding demo item */
DateTime        Date             "Date [%1$tA, %1$td.%1$tm.%1$tY]"   <calendar>  {
     ntp="Europe/Berlin:de_DE" }

/* Demo items */
Switch DemoSwitch                "Switch"
Dimmer DimmedLight               "Dimmer [%d %%]"        <slider>
Color  RGBLight                  "RGB Light"             <slider>
Rollershutter DemoShutter        "Roller Shutter"
Dimmer DemoBlinds                "Blinds [%d %%]"        <rollershutter>
```

```
Number Scene_General            "Scene"                 <sofa>
Number Radio_Station            "Radio"                 <network>
Dimmer Volume                   "Volume [%.1f %%]"
Number Temperature_Setpoint     "Temperature [%.1f C]" <temperature>

String UnknownDevices    "Unknown Devices in Range: [%s]" { bluetooth="?" }
Number NoOfPairedDevices "Paired Devices in Range: [%d]"  { bluetooth="!" }
```

A.2 Eingabetexte der Fallstudie

Text 1

Turn on the light over the table in the kitchen. Move down the toilet shutter and turn off the light in the office.

Text 2

First move up the shutter in the bedroom. After turning on the light in the bathroom, please turn on the light in the kitchen and the bureau light.

Text 3

Increase the light in the kitchen and turn of the heating in the office. Move up the roller-shutter in the living room after turning off the heating in the bath. Stop the living room rollershutter and open the kitchen rollershutter.

Text 4

Turn off all lights on the first floor. Dim all lights on the ground floor to 60 percent. Set the RGB light to red. Turn off the heaters in all rooms. Turn on the heater in the ground floor toilet. Turn on the heater in the living room. Set the radio to station number 1. Increase the volume of the radio.

Anhang B

Fragebogen

B.1 Statistische Informationen

Zu den Beschreibungen wurden zudem statistische Informationen abgefragt, um die Ergebnisse qualitativ beurteilen zu können. Hierbei wurden folgende Merkmale erfasst:

- Name
- Alter
- Geschlecht
- Programmierkenntnisse (objektorientiert, prozedural, funktional, keine, sonstiges)

B.2 Basisaktionen

Die Basisaktionen, die bei allen Objekten zur Verfügung stehen, sind: move, turn, roll, resize, say, think, play sound, move to, move toward, move away from, orient to, turn to face, point at, set point of view, set pose, stand up, move at speed, turn at speed, roll at speed, constrain to face, constrain to point at. In den nachfolgenden Übersichtstabellen sind die Basisaktionen zur Kürzung nicht angegeben.

B.3 Aufgabenbeschreibung

In diesem Fragebogen geht es darum, eine kurze Filmsequenz in natürlicher Sprache zu beschreiben.

Es soll ein System entwickelt werden, das Anweisungen in natürlicher Sprache entgegen nimmt und diese dann in Quellcode übersetzt, der vom Computer verstanden wird. Dies soll dazu dienen, Menschen mit geringen oder fehlenden Programmierkenntnissen die Möglichkeit zu bieten, selbst ein Programm zu entwickeln.

Für die Umsetzung dieses Projekts muss in einem ersten Schritt herausgefunden werden, wie Menschen solche Anweisungen an einen Computer formulieren würden. Zu diesem Zweck wurden mithilfe des Programms Alice zwei Filmsequenzen entwickelt, von denen eine im Laufe dieses Fragebogens von Ihnen in natürlicher Sprache beschrieben werden soll.

Lesen Sie sich bitte die Anleitung zunächst vollständig durch, bevor Sie die Fragen beantworten.

Ein Beispiel

Starten Sie nun die Filmsequenz. Sehen Sie sich diese bis zum Ende an und lesen Sie anschließend die folgende Beschreibung, die Ihnen später bei der Anfertigung Ihrer eigenen Beschreibung als Richtlinie dienen soll:

> The ground is covered with grass. In the background there is a sunflower on the far right facing southwest. In the foreground there is a monkey in the middle facing southwest. To the monkey's right, there is a penguin facing south. To the monkey's left, there is a remote. Left of the remote, there is a light bulb. Behind the light bulb, to the right, there is a duck prince facing southeast. Behind the monkey, to the right, there is a bucket.
>
> The duck prince turns to face the monkey. The duck prince commands. The monkey sighs. The monkey turns to face the remote. The monkey jumps. The monkey presses a button. A very short time passes. The penguin turns to face the bucket. The penguin glides. The duck prince quickly turns to face the penguin. The duck prince scolds. A very short time passes. The monkey laughs. The monkey presses a button. The penguin jumps once.

Starten Sie nun die Filmsequenz. Sehen Sie sich diese bis zum Ende an und beschreiben Sie die Filmsequenz anschließend bitte so genau wie möglich. Stellen Sie sich dabei vor, Sie wollten dem Computer Anweisungen geben, was er zu tun hat. Versuchen Sie bei der Beschreibung, die Reihenfolge der Handlung einzuhalten. Sollten Sie die Filmsequenz mehrmals betrachten oder pausieren wollen, so ist dies kein Problem.

Die Beschreibung soll bevorzugt in englischer Sprache verfasst werden, sollten Sie sich dazu nicht in der Lage fühlen, dürfen Sie auf Deutsch ausweichen.

Die folgenden Objekte werden in der Filmsequenz vorkommen und sollen in Ihrer Beschreibung mit dem angegebenen Namen angesprochen werden. Es ist weiterhin zu beachten, dass die Objekte neben Basisfähigkeiten über bestimte spezielle Fähigkeiten verfügen – diese sind bei der jeweiligen Filmsequenz tabellarisch angegeben.

Sollte es Ihnen nicht möglich sein, mit den vorgegebenen Aktionen eine im Film gesehene Handlung zu beschreiben, ist es vorzuziehen, diese mit anderen Worten zu beschreiben, als sie ganz wegzulassen.

B.4 Animationen

Je nach dem welche Animation beschrieben werden sollte, wurde eine Tabelle mit den verfügbaren Objekten und Aktionen dem Fragebogen beigefügt. Nachstehend finden sich die Tabellen zu den Animationen des Korpus.

Alice

Objekt	Aktionen
Alice (AliceLidell)	shake head, go to (target),
Kaninchen (Bunny)	hop, hop to (target), jump into hole
Katze (ChechireCat)	appear, disappear
Loch im Boden (Hole)	

Beachday

Objekt	Aktionen
Frosch (Frog)	tap foot, croak, nod head, jump
Känguru (Kangaroo)	nod, clap, wag tail
Kaninchen (Bunny)	jump
Mädchen (Girl)	wave

Cheerleader

Objekt	Aktionen
Cheerleader (Cheerleader)	cheer, move to the middle, jump
Pinguin (Penguin)	flap wings, turn head right, glide away
Wasserfall (Waterfall)	

Cowboy

Objekt	Aktionen
Cowboy (Cowboy)	clapHands, kick, nod
Kamel (Camel)	nod

Dragon

Objekt	Aktionen
Drache (Dragon)	fly in the air, fly away
Frau (Woman)	
Hund (Dog)	
Windmühle (Windmill)	

Farm

Objekt	Aktionen
Boden (Ground)	shake
Farm (FarmHouse)	
Feld (Cornfield)	
Husky (Husky)	roll, wag tail, backflip, sit stand (lässt ihn hinsetzen wenn er steht und umgekehrt)
Kuh (Cow)	swish tail
Mann (OldMan)	clap, say, hands up, hands down
Pferd (Horse)	say
Scheune (Barn)	
Zaun (Fence)	

Monkey

Objekt	Aktionen
Affe (Monkey)	jump, press button, sigh, laugh
Eimer (Bucket)	
Entenprinz (DuckPrince)	scold, knock over, command
Glühbirne (LightBulb)	
Pinguin (Penguin)	jump, glide

Moon

Objekt	Aktionen
Alien (Alien)	nod, shoot at (target)
Astronaut (Astronaut)	make step
Feuer (Fire)	appear, spin like crazy
Mondkapsel (LunarLander)	

Penguin

Objekt	Aktionen
Affe (Monkey)	jump, press button, sigh, laugh
Eimer (Bucket)	
Entenprinz (DuckPrince)	scold, knock over, command
Glühbirne (LightBulb)	
Pinguin (Penguin)	jump, glide

Rabbit

Objekt	Aktionen
Briefkasten (Mailbox)	open door
Broccoli (Broccoli)	
Frosch (Frog)	hop, croak
Kaninchen (Bunny)	eat, hop, open mailbox, tap foot
Palme (PalmTree)	

Szenerie: Mummy

- Apfelbaum (AppleTree)
- Astronaut (Astronaut)
- Cowboy (Cowboy)
- Esel (Donkey)
- Haus (House)
- Mumie (Mummy)

Szenerie: Saloon

- Affe (Monkey)
- Pferd (Horse)
- Saloon (Saloon)
- Scheune (Barn)

Wizard

Objekt	Aktionen
Altertümliche Frau (FirstCenturyWoman)	appear
Entenprinz (DuckPrince)	move scepter
Feuer (Fire)	appear, disappear
Zauberer (Wizard)	move wand, disappear

Woman

Objekt	Aktionen
Blauer Junge (BlueBoy)	raise hand (hand, number of times), bring (object, to), hold hand (other hand, with which hand)
Frau (Woman)	wave hand (hand, number of times)
Känguru (Kangaroo)	bring (object, to), hold hand (other hand, with which hand)

Anhang C

Alice-Korpus

	Texte		Sätze		Wörter		
Animation	∑T	ϟ	∑S	S/T	∑W	W/T	W/S
Alice	4	0	82	20,5	652	163	8,08
Beachday	15	1	263	17,53	2 551	170,07	9,96
Cheerleader	12	1	154	12,83	1 428	119	9,59
Cowboy	12	4	132	11	1 541	128,42	11,31
Dragon	12	0	160	13,33	1 671	139,25	10,66
Farm	17	1	387	22,76	3 649	214,65	9,67
Monkey	1	0	22	22	162	162	7,36
Moon	4	1	73	18,25	752	188	10,71
Penguin	1	0	23	23	157	157	6,83
Rabbit	23	1	409	17,78	3 389	147,35	8,76
Szenerie: Mummy	15	0	143	9,53	1 366	91,07	10,08
Szenerie: Saloon	15	0	98	6,53	903	60,2	10,46
Wizard	11	4	142	12,91	1 074	97,64	8,5
Woman	6	0	122	20,33	1 143	190,5	9,27
Gesamt	148	13	2 210	14,93	20 438	138,09	9

Legende: Anzahl Texte ∑T, davon von der Auswertung ausgeschlossen ϟ. Anzahl Sätze ∑S, durchschnittliche Anzahl Sätze pro Text S/T. Anzahl Wörter ∑W, durchschnittliche Anzahl Wörter pro Text W/T, durchschnittliche Anzahl Wörter pro Text W/S.

Derzeit enthält das Korpus insgesamt 14 Welten und insgesamt 148 Texte. Damit hat es einen Umfang von insgesamt 20 438 Wörtern in 2 210 Sätzen.

Die Texte sind hier in der korrigierten Fassung abgedruckt. Die Originaldokumente wurden gesichert und stehen digital zur Verfügung. Die jeweils aktuelle Fassung des Korpus findet sich unter https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Corpus.

C.1 Animationen und Beispieltexte

Im Folgenden sind die einzelnen Animationen des Korpus aufgeführt. Zu jeder Animation wird eine Abbildung der Anfangsszenerie sowie ein Text als Beispiel abgedruckt. Der vollständige Korpus kann hier aufgrund seiner Größe nicht vollständig abgedruckt werden.

Alice

Text	Sätze	Wörter	Vokabulargröße
01	23	213	100
02	15	105	65
03	27	164	63
04	17	170	80
4	82	652	172

Alice 01

The ground is green, covered with grass. The sky is blue. In the foreground, on the far left, there is a girl with blond hair that looks like Alice from the fairy tale. Alice looks to the southwest. In the foreground, on the left, there is a black hole. Behind the hole, a bit to the left, there is a white bunny. The bunny looks to the southwest, too. In the background, a little right, there is a big old tree. The Bunny says, 'Hello'. Alice answers to the bunny. Alice says, 'uuhm'. The bunny jumps three times and then says, 'Come over here'. Alice shakes her head. The bunny says, 'Come on, What time is it'. Alice goes to the bunny. While she goes, she says, 'Ohh' and after a little while, 'Okay'. A cat with a broad grin appears on the tree. The cat turns its head to the bunny and says, 'grin'. Alice and the bunny turn their heads to the cat at the same time. The bunny turns its whole body to face the black hole. While it turns, it says, 'Ahhhhhhh'. The bunny jumps 3 times to the hole and then jumps into the hole. While the bunny jumps, Alice turns to the bunny and says, 'Hey, Wait'.

Beachday

Text	Sätze	Wörter	Vokabulargröße
00	18	185	73
01	21	160	57
02	23	220	80
03	18	181	68
04	20	187	72
05	20	195	65
06	15	183	79
07	9	67	45
08	24	217	66
09	26	210	83
10	22	209	95
11	0	0	0
12	10	160	70
13	20	180	72
14	17	197	86
15	263	2551	267

Beachday 00

The ground is covered with sand. The sky is blue. In the background there is a BeachTerrain and in the center of the background there is a Lighthouse. In front of the BeachTerrain on the right hand side, there is a BeachHouse facing southwest. On the left side of the screen, in front of the BeachTerrain are two PalmTrees. In front of the BeachHouse there is Girl facing southwest. In the foreground in the left side, there is a Bunny facing east. In front of the Bunny there is a Frog facing southeast. In the foreground on the right side, there is a Kangaroo facing the Frog.

The Girl moves to the center of the screen. While the Girl is moving, the Frog taps its foot twice. The Girl waves before the Kangaroo simultaneously wags its tail and claps. All animals turn to face the Girl. The Frog and the Bunny jump and at the same time the Kangaroo nods. All animals consecutively move towards the Girl. The Girl turns to face the Lighthouse. Everyone turns to face the Lighthouse and moves towards the Lighthouse.

Cheerleader

Text	Sätze	Wörter	Vokabulargröße
00	16	108	54
01	10	104	60
02	15	115	61
03	9	86	51
04	19	222	90
05	12	127	72
06	12	117	63
07	14	100	59
08	9	130	70
09	9	94	54
10	11	118	68
11	18	107	42
12	154	1428	229

Cheerleader 00

The ground is covered with grass. The sky is blue. In the background you can see a waterfall. In front of the waterfall on the right hand side there is a penguin facing toward the camera. Next to the penguin on the left hand side a cheerleader is looking to the waterfall. The cheerleader cheers. The penguin flaps his wings twice. The cheerleader turns to face the penguin. The penguin turns its head right. The penguin flaps its wings once. The penguin glides away. The cheerleader says, 'Where are you going.' The cheerleader moves to the middle. The cheerleader jumps twice. The cheerleader cheers. The cheerleader jumps once.

Cowboy

Text	Sätze	Wörter	Vokabulargröße
00	8	78	48
01	12	82	45
02	11	124	65
03	9	78	48
04	16	121	52
05	11	91	47
06	9	122	63
07	7	84	53
08	16	319	156
09	10	141	78
10	16	239	123
11	7	62	37
12	132	1541	334

Cowboy 00

The ground is covered with sand. In the background there is a pyramid. In the left front corner there is a cowboy facing east. On the right side a bit further backwards there is a camel facing southwest. The camel nods twice and at the same time the cowboy walks two meters. Then the cowboy turns right by a quarter revolution. The cowboy jumps and claps his hands three times. The cowboy and the camel nod four times.

Dragon

Text	Sätze	Wörter	Vokabulargröße
00	15	127	53
01	9	83	43
02	16	165	68
03	10	126	58
04	15	171	79
05	12	149	72
06	12	157	68
07	11	116	59
08	13	155	72
09	13	131	63
10	12	132	71
11	21	159	52
12	159	1671	217

Dragon 00

The ground is covered with grass. In the foreground you can see on the right hand side a woman looking into the camera and on the left hand side a dog facing the woman. In the background on the left side there is a windmill. On the right side in the background stands a dragon looking towards the dog. The dragon turns to face the windmill. The dog turns to face the dragon. The dragon says: 'What a beautiful windmill'. The dog turns to face the windmill. The dragon turns to face the camera. The dragon flies in the air. The woman jumps. The woman moves towards the windmill. The woman turns to face the dragon. The woman says: 'You are scaring me'. The dragon flies away.

Farm

Text	Sätze	Wörter	Vokabulargröße
00	12	121	75
01	21	227	105
02	29	259	102
03	29	226	91
04	26	228	95
05	25	236	87
06	17	222	112
07	21	128	69
08	29	209	84
09	38	280	125
10	0	0	0
11	23	287	132
12	21	249	104
13	21	194	86
14	24	300	129
15	22	228	96
16	29	255	92
17	387	3649	432

Farm 00

Insert scene description here.

The husky rolls and then the OldMan claps. While the Husky wags its tail for four times, the OldMan first says ’Good boy’ and then says ’Now do a backflip’. The OldMan raises and lowers his hands while the Husky performs a backflip. The Cow swishes its tail two times and at the same time the Horse simultaneously says ’pffff’ and turns to face the Husky. While the Husky is sitting for 3 seconds, the OldMan turns left for 1 revolution. The Ground shakes. The OldMan says ’WAAA’ and turns left, raises his hands and moves out of the scene. The Ground shakes again. All animals turn to face the OldMan. Then all animals consecutively move forward.

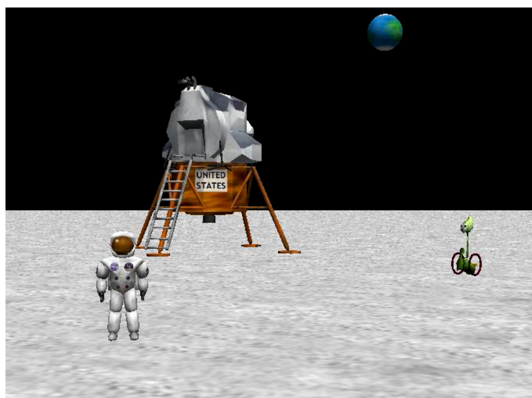
Monkey

Text	Sätze	Wörter	Vokabulargröße
01	22	162	60
1	22	162	60

Monkey 01

The ground is covered with grass. In the background there is a sunflower on the far right facing southwest. In the foreground there is a monkey in the middle facing southwest. To the monkey's right there is a penguin facing south. To the monkey's left there is a remote. Left of the remote there is a light bulb. Behind the light bulb, to the right, there is a duck prince facing southeast. Behind the monkey, to the right, there is a bucket. The duck prince turns to face the monkey. The duck prince commands. The monkey turns to face the remote. The monkey jumps. The monkey presses a button. The monkey sighs while the penguin jumps at the same time. The penguin turns to face the bucket. The penguin glides. The duck prince quickly turns to face the penguin. The duck prince scolds. The penguin turns to face the duck prince. The penguin glides. The monkey laughs. The monkey presses a button.

Moon



Text	Sätze	Wörter	Vokabulargröße
01	28	234	108
02	12	118	70
03	19	208	104
04	14	193	89
4	73	753	216

Moon 01

The ground is stony in a light gray. The sky is black. In the foreground there is a spaceman. The spaceman looks south. Behind the spaceman, in the background, there is a lunar module. In the middle of the scene, on the far left, there is a green alien on wheels. In the sky, up on the right, there is a planet looking like the earth. The spaceman makes a step forward. While he makes the step, he says, 'That is one small step for man'. While he does this, the alien moves a few meters forward and turns a bit left. At the same time the alien turns his head to face the spaceman. The spaceman says, 'one'. The spaceman says, 'giant leap for'. At the same time he turns his head to face the alien. The spaceman says, 'Heee'. The alien turns to face the spaceman. The alien says, 'grumpfl'. At the same time, it nods. The spaceman turns to face the alien. The spaceman says, 'I come in Peace'. At the same time he makes a step. The alien lifts its arm, like it wants to shoot at the astronaut. A sound like 'woosh' appears. Suddenly, there is a big fire around the spaceman. The alien turns a half revolution. The alien moves out of the scene. Someone in the lunar module says, 'Houston'. Then it says, 'we have a problem'.

Penguin

Text	Sätze	Wörter	Vokabulargröße
00	23	157	61
1	23	157	61

Penguin 00

The ground is covered with grass. In the background there is a sunflower on the far right facing southwest. In the foreground there is a monkey in the middle facing southwest. To the monkey's right, there is a penguin facing south. To the monkey's left, there is a remote. Left of the remote, there is a light bulb. Behind the light bulb, to the right, there is a duck prince facing southeast. Behind the monkey, to the right, there is a bucket. The duck prince turns to face the monkey. The duck prince commands. The monkey sighs. The monkey turns to face the remote. The monkey jumps. The monkey presses a button.

A very short time passes. The penguin turns to face the bucket. The penguin glides. The duck prince quickly turns to face the penguin. The duck prince scolds. A very short time passes. The monkey laughs. The monkey presses a button. The penguin jumps once.

Rabbit

Text	Sätze	Wörter	Vokabulargröße
01	20	169	57
02	9	106	47
03	19	191	75
04	11	109	48
05	16	123	43
06	12	129	59
07	14	222	93
08	23	159	62
09	19	152	56
10	11	105	54
11	12	109	54
12	22	148	57
13	12	130	55
14	13	152	73
15	21	154	60
16	21	138	50
17	23	148	52
18	24	155	61
19	24	163	57
20	16	144	62
21	19	137	53
22	26	180	69
23	22	166	56
23	409	3389	293

Rabbit 01

The ground is covered with grass, the sky is blue. In the background on the left hand side there is a palm tree. In the foreground on the left hand side there is a closed mailbox facing southeast. Right to the mailbox there is a frog facing east. In the foreground on the

right hand side there is a bunny facing southwest. In front of the bunny there is a broccoli. The bunny turns to face the broccoli. The bunny hops three times to the broccoli. The bunny eats the broccoli. The bunny turns to face the frog. The bunny taps its foot twice. The frog croaks. The frog turns to face northeast. The frog hops three times to northeast. The bunny turns to face the mailbox. The bunny hops three times to the mailbox. The bunny opens the mailbox. The bunny looks into the mailbox and at the same time the frog turns to face the bunny. The frog hops two times to the bunny. The frog croaks.

Szenerie: Mummy

Text	Sätze	Wörter	Vokabulargröße
00	6	72	26
01	6	69	23
02	7	74	30
03	7	91	43
04	11	94	41
05	14	101	33
06	8	67	30
07	11	96	26
08	9	86	38
09	13	108	32
10	5	69	33
11	14	104	27
12	7	85	43
13	10	100	39
14	15	150	46
15	143	1366	134

Szenerie: Mummy 00

In the background to the left there is a house facing southwest. Next to the house to the right there is a mummy facing the tree. To the right in the background there is a tree. In the foreground to the left there is a cowboy facing the astronaut. In the foreground to the right there is an astronaut facing the house. Behind the astronaut there is a donkey facing the cowboy.

Szenerie: Saloon

Text	Sätze	Wörter	Vokabulargröße
00	8	50	18
01	5	58	20
02	5	53	27
03	6	68	31
04	10	78	38
05	8	61	23
06	8	52	26
07	10	80	23
08	7	53	23
09	8	64	23
10	3	46	22
11	8	64	19
12	4	39	22
13	5	54	24
14	3	83	31
15	98	903	85

Szenerie: Saloon 00

A saloon is in the background left. A barn is in the background right. A horse is in the foreground left. A monkey is in the foreground right. The saloon is facing south. The barn is facing southwest. The horse is facing the monkey. The monkey is facing the horse.

Wizard

Text	Sätze	Wörter	Vokabulargröße
01	27	150	68
02	15	86	42
03	11	79	45
04	16	139	63
05	11	96	54
06	17	112	50
07	9	97	63
08	11	99	49
09	14	86	54
10	6	24	24
11	5	105	83
11	142	1073	263

Wizard 01

There are four objects. One object is in the background. The object in the background is a happy tree. The happy tree faces front. There is a wizard called Bob in the middle. Bob is in the foreground. To the left of Bob is Christine. Christine is a firstCenturyWoman1. Bob faces Christine. Christine faces front. Christine is invisible. There is a fire animation in front of Christine. The fire animation is invisible. In the foreground to the right, there is nothing. Daryl is a duck prince. Daryl and Bob are spell casters. Daryl is off the screen to the right. Daryl faces Bob.

Bob moves his wand. The fire animation appears. Christine appears and at the same time the fire animation disappears. Christine turns her head at Bob. Little time passes. Daryl moves to the foreground, to the right. Daryl moves his scepter. While Daryl moves his scepter, Bob vanishes.

Woman

Text	Sätze	Wörter	Vokabulargröße
01	25	212	56
02	25	303	97
03	16	169	59
04	22	183	74
05	13	99	46
06	18	170	61
6	119	1136	170

Woman 01

The sky is blue. The ground is covered with sand. On the right side in the background there is a factory. On the left side in the background there is a farm house. On the left side in the foreground there is a kangaroo. On the right side in the foreground stands a blue boy facing south. On the left side of the blue boy stands a woman facing south.

The blue boy raises his left hand once. The blue boy turns to face the woman. The woman turns to face the blue boy. The woman waves her left hand once. The blue boy raises his right hand twice. The woman waves her right hand twice. The blue boy moves to the woman. The blue boy moves 0.6 meter to the right. The woman and the blue boy turn to face the factory. The blue boy holds her right hand with his left hand. The blue boy brings the woman to the factory. The woman moves to the kangaroo. The woman and the kangaroo turn to face the farm house. The woman moves 0.2 meter to the left. The woman moves 0.3 meter forward. The kangaroo holds her right hand with its left hand. The kangaroo brings the woman to the factory.

C.2 Selbsteinschätzung der Probanden

Frage	kein	wenig	mittel	sehr gut	keine Angabe
Englischkenntnisse	2	4	21	26	13
Programmiererfahrung	14	7	12	20	13

C.3 Angaben zur Muttersprache der Probanden

Sprache	Anzahl
Deutsch	38
Englisch	3
sonstige	3
Unbekannt	22

Anhang D

Annotationen von NLCI

D.1 NLP-Vorverarbeitung

POS Umschließt ein Wort in einem Eingabesatz und markiert dessen Wortart.
Attribute:

- `value`: (Zeichenkette); Wortart gemäß PENN-Markierungssatz.

SyntaxTreeNode Geschachtelte Annotation, die den Syntaxbaum abbildet.
Attribute:

- `sentenceID`: (int); dokumentweite Satz-Identifikation.
- `label`: (Zeichenkette); Markierung der entsprechenden Konstituente.

Coreference Speichert das Ergebnis der Korreferenzanalyse; umschließt Wörter (meist Nomen oder Pronomen). Über eine gemeinsame `chainID` wird eine Kette von Nennungen aufgebaut, die sich auf dieselbe Entität beziehen.
Attribute:

- `chainID`: (int); dokumentweite Identifikation der Korreferenzkette.
- `mentionID`: (int); dokumentweite Identifikation der Nennung.
- `representativeMention`: (Zeichenkette); Text des Anfangs der Korreferenzkette (idealerweise der Name der Entität).
- `representativeMentionID`: (int); ID des Anfangs der Korreferenzkette.
- `sentenceID`: (int); dokumentweite Satz-Identifikation.

Dependency Speichert das Ergebnis der Abhängigkeitsanalyse; jedes Wort des Textes wird von einer `Dependency`-Annotation umschlossen, auch wenn das Wort an keiner Abhängigkeit teilnimmt.
Attribute:

- `index`: (int); dokumentweite Wort-Identifikation.
- `sentence`: (int); dokumentweite Satz-Identifikation.
- `typedDependencies`: (Zeichenkette); Liste von typisierten Abhängigkeiten.
- `typedDependenciesCollapsed`: (Zeichenkette); Liste von typisierten Abhängigkeiten.
- `typedDependenciesCCprocessed`: (Zeichenkette); Liste von typisierten Abhängigkeiten.

Die Liste der typisierten Abhängigkeiten wird als kommagetrennte Liste folgendermaßen dargestellt: `reln~id`, wobei `reln` der Name der Abhängigkeit ist und `id` der `index` des *dependent*. Das annotierte Wort ist der *governor*.

D.2 Allgemeine Annotationen von NLCI

sentence Umschließt einen Satz der Eingabe (inkl. aller Nebensätze usw.).
Attribute:

- `type`: `ActionDescription` (Aktionssatz), `EventDescription` (Ereignissatz), `ErrorDescription` (Erkennung fehlgeschlagen), `SetupDescription` (Aufbausatz).
- `<asId>~<actorId>~<actionId>~<objectId>`: score (float); in diesen Attributen werden die Bewertungen der ermittelten atomaren Sätze gespeichert. Die IDs sind satzlokal und numerisch.

Die `<objectId>` in der `score`-Annotation wird auch dazu benutzt, die folgenden Zustände anzuzeigen. Negative Angaben an dieser Stelle haben die folgenden Bedeutungen:

-1 Die Methode erfordert keine Parameter und der atomare Satz enthält kein Objekt.
-2 Die Methode erfordert keine Parameter, der atomare Satz enthält aber ein Objekt.
-3 Die Methode besitzt einen Parameter, der kein Objekt ist, der atomare Satz enthält aber ein Objekt.
-4 Die Methode besitzt einen Parameter, der kein Objekt ist, und im atomaren Satz kann kein Objekt gefunden werden.

atomicSentence Umschließt eine atomare Konstituente (Akteur, Objekt und Objekt).
Attribute:

- `id`: (int); ID des zugehörigen Atomaren Satzes (in einem Eingabesatz können mehrere atomare Sätze vorhanden sein). `type`: `actor`, `action`, `object`, `misc`; kann der Satz nicht analysiert werden, werden die erkannten Elemente als `misc` gekennzeichnet.

actor, action, object, misc Umschließt eine atomare Konstituente und kennzeichnet ihre Funktion. Werden für eine Konstituente mehrere Treffer in der NLCI-Ontologie ermittelt, erhält die Konstituente mehrere solche Annotationen. Kann die Funktion nicht ermittelt werden, wird eine `misc`-Annotation verwendet, um die Verweise auf die Ontologie-Individuen zu annotieren.
Attribute:

- `ind_id`: (int); Identifikation der atomaren Konstituente (wird im score-Attribut der `sentence`-Annotation zur Referenzierung verwendet).
- `ontologyIndiv`: (Zeichenkette); Name eines Individuums aus der NLCI-Ontologie.
- `score`: (float); Bewertung des Ontologie-Individuums für die atomare Konstituente.

MethodActor Umschließt eine atomare Konstituente vom Typ `actor`, wenn die Methodenauswahl erfolgreich war.
Attribute:

- `id`: (int); Identifikation des zugehörigen Methodenaufrufs.
- `name`: (Zeichenkette); Name eines Individuums aus der NLCI-Ontologie.

Method Umschließt eine atomare Konstituente vom Typ `action`, wenn die Methodenauswahl erfolgreich war.
Attribute:

- `id`: (int); fortlaufende ID für Methodenaufrufe.
- `name`: (Zeichenkette); Name des Methodenindividuums in der Ontologie.
- `score`: (double); Bewertung des Methodenaufrufs entsprechend der Bewertung des Atomaren Satzes.

MethodArgument Umschließt eine atomare Konstituente vom Typ `object` oder einen anderen Methodenparameter (Zeichenketten o.Ä.) im Eingabesatz.
Attribute:

- `id`: (int); ID des zugehörigen Methodenaufrufs.
- `paramterName`: (Zeichenkette); Name des Parameters.
- `parameterType`: (Zeichenkette); Typ des Parameters.
- `value`: (Zeichenkette); Argumentwert oder Name eines Individuums aus der NLCI-Ontologie.

D.3 Ergebnisse der Reihenfolgeanalyse

PointOfTime Umschließt eine atomare Konstituente vom Typ `action` nach der Reihenfolgeanalyse.
Attribute:

- `position`: (int); Position der Aktion auf der Zeitachse.

D.4 Ergebnisse der Kontrollstrukturenanalyse

Alle Annotationen der Kontrollstrukturenanalyse umschließen eine atomare Konstituente vom Typ `action` und verfügen über ein Attribut `id`, das konkrete Kontrollstrukturen zusammenfasst (bspw. zwei Annotationen, die zum selben `DO_IN_ORDER`-Block gehören).

DO_IN_ORDER Annotation für die sequenzielle Ausführung der enthaltenen Aktionen.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.

DO_TOGETHER Annotation für die parallele Ausführung der enthaltenen Aktionen.

- `id`: (int); Identifikation der Kontrollstruktur.

LOOP Annotation für die Aktionen einer einfachen Schleife.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.
- `n`: (int); Anzahl der Schleifenwiederholungen.

WHILE Annotation für die Aktionen, die innerhalb des `WHILE`-Schleifenrumpfs ausgeführt werden sollen.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.

WHILE_CONDITION Annotation für die Bedingung der `WHILE`-Schleife. Soll eine Funktion mit einem Boole'schen Rückgabewert sein.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.

FOR_ALL_IN_ORDER Annotation für die sequenzielle Ausführung der Aktionen mit den referenzierten Gruppenobjekten.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.
- `entityListRef`: (int); Referenz auf eine Individuenliste (s.u.).

FOR_ALL_TOGETHER Annotation für die parallele Ausführung der Aktionen mit den referenzierten Gruppenobjekten.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.
- `entityListRef`: (int); Referenz auf eine Individuenliste (s.u.).

ENTITY_LIST Annotation, die eine Liste von Akteuren enthält, die in einer `FOR_ALL`-Annotation referenziert wird.
Attribute:

- `id`: (int); Identifikation der Kontrollstruktur.
- `ontIndivs`: (Zeichenkette); eine Liste mit Referenzen auf Akteur-Individuen in der NLCI-Ontologie.

Anhang E

Syntaxregeln für die Bestimmung von Methodenparametern

NLCI analysiert die Verbalphrasen einer Aktionsbeschreibung, um Methodenparameter zu identifizieren. Dazu wird in den Syntaxbäumen nach Teilbäumen der Verbalphrase gesucht, die den folgenden Mustern entsprechen.

E.1 Richtungsangaben

Mögliche Synonyme für Richtungsangaben:

Richtungsangabe	Mögliche Synonyme
forward	forward, forwards, frontward, frontwards, forrad, forrard, forth, onward, ahead, onwards, forrader, fore, forwardly
backward	backward, backwards, back, rearward, rearwards
up	up, upwards, upward, upwardly
down	down, downwards, downward, downwardly
left/right side	(the)? right side , (the)? right hand side , (the)? left side , (the)? left hand side

Regeln für die Richtungserkennung:

NP → (DT)? JJ (NOUN)? NOUN
NP → (DT)? NOUN
NOUN , JJ
ADVP → RB , PRT → RP , ADVP → NOUN , ADJP → JJ , RB

E.2 Numerische Werte

Regel	Beispiel
ADVP → RB	once, twice, thrice
(NP|QP|NP-TMP) → (DT)? (NOUN)* CD (NOUN)+	„jump 2 times“, „walk 2 meters“
NP → (DT)? (JJ|NOUN) NOUN	„a half revolution“
NP → NP (PP → of CD NOUN)	„at a speed of 2 m/s“

Anhang F

Wortartmarkierungen nach PENN

Der Penn-Markierungssatz nach Marcus et al. [MSM93] mit den Markierungen CC bis WRB sowie den zwölf Sonderkennzeichnungen für Zeichen und Symbole: In Referenz [San95] finden sich ausführliche Erklärungen sowie Beispiele zu den einzelnen Wortarten.

Tag	Bedeutung	Tag	Bedeutung
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

Tag	Bedeutung	Tag	Bedeutung
PRP	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

Anhang G

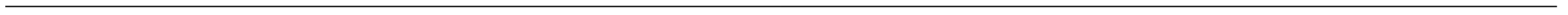
Typisierte Abhängigkeiten nach Stanford

Die nachstehende Übersicht entstammt Referenz [dM15]. Dort finden sich ausführliche Erklärungen zu den einzelnen Abhängigkeiten sowie Beispiele.

Die typisierten Abhängigkeiten sind hierarchisch aufgebaut. Die allgemeinste Abhängigkeit ist *dependent (dep)*; sie wird benutzt, wenn keine präzisere Abhängigkeit existiert oder ermittelt werden kann.

Abhängigkeit	Beschreibung
root	root
dep	dependent
aux	auxiliary
auxpass	passive auxiliary
cop	copula
arg	argument
agent	agent
comp	complement
acomp	adjectival complement
ccomp	clausal complement with internal subject
xcomp	clausal complement with external subject
obj	object
dobj	direct object
iobj	indirect object
pobj	object of preposition
subj	subject
nsubj	nominal subject

Abhängigkeit	Beschreibung
nsubjpass	passive nominal subject
csubj	clausal subject
csubjpass	passive clausal subject
cc	coordination
conj	conjunct
expl	expletive (expletive „there“)
mod	modifier
amod	adjectival modifier
appos	appositional modifier
advcl	adverbial clause modifier
det	determiner
predet	predeterminer
preconj	preconjunct
vmod	reduced, non-finite verbal modifier
mwe	multi-word expression modifier
mark	marker (word introducing an advcl or ccomp
advmod	adverbial modifier
neg	negation modifier
rcmod	relative clause modifier
quantmod	quantifier modifier
nn	noun compound modifier
npadvmod	noun phrase adverbial modifier
tmod	temporal modifier
num	numeric modifier
number	element of compound number
prep	prepositional modifier
poss	possession modifier
possessive	possessive modifier ('s)
prt	phrasal verb particle
parataxis	parataxis
goeswith	goes with
punct	punctuation
ref	referent
sdep	semantic dependent
xsubj	controlling subject



Abbildungsverzeichnis

1.1 Die Verarbeitung soll für den Benutzer transparent geschehen: Er interagiert textuell in natürlicher Sprache mit der Maschine. 5
1.2 Vieles ist in der Architektur vorgegeben. Entwickler steuern neben der API nur die API-Beschreibung und ggf. den Quelltext-Erzeuger bei. . . . 6
1.3 Der interne Aufbau der NLCI-Architektur. 7

2.1 Der Syntaxbaum des Satzes „My dog also likes eating sausage.“ 14
2.2 Ein Syntaxbaum für den Satz „Sophie saw a man with a telescope.“ . . . 16
2.3 Ein Syntaxbaum für den Satz „Sophie saw a man with a telescope.“ . . . 16
2.4 Ein Abhängigkeitsgraph für den Satz „Sophie saw a man with the telescope“. Die Wortarten sind unterhalb der Wörter angegeben. Der Abhängigkeitsgraph zeigt die Variante aus Abbildung 2.2. 17
2.5 Beispiel für eine Ontologie. Konzepte sind in einer Festbreitenschrift abgebildet, Instanzen in einer Kursivschrift. 23
2.6 Beispiel für eine einfache Ontologie geöffnet im Editor Protégé. 25

3.1 Der allgemeine Ansatz im Vergleich zur NLCI-Systematik. 28
3.2 Eine detailliertere Übersicht über NLCI mit den bereitgestellten Einschüben. 30
3.3 Empirisch erhobene Texte steuern den Forschungsprozess. 35
3.4 Erstellung des NLCI-Korpus. Von Probanden geschriebene Texte können zur Evaluation herangezogen werden. 36

5.1 Schematischer Ablauf der API-Verknüpfungsstufe. 79
5.2 Abhängigkeitsgraph für den Satz „John turns the doorknob and opens the door.“ In der Zeile unter den Wörtern wurden die Wordarten notiert. . . . 81
5.3 Schematischer Überblick über die Zeitachsenanalyse. 95
5.4 Gegenüberstellung von chronologischer und nicht-chronologischer Beschreibung. 96

5.5 Gegenüberstellung der Zeitachsen für die chronologische und nicht-chronologische Beschreibungen aus Abbildung 5.4. 99
5.6 Ein Abhängigkeitsgraph für den Satz „The bunny hops and at the same time the frog nods.“ Die Wortarten sind unterhalb der Wörter angegeben. 101

6.1 Bildschirmfoto der Alice-Benutzeroberfläche. 114

Tabellenverzeichnis

2.1 Auszug aus dem Markierungssatz der PENN-Tree-Bank. 12

3.1 Selbsteinschätzung der 66 Autoren des NLCI-Korpus 37
3.2 Eine Übersicht über die Animationen im NLCI-Korpus. 39
3.3 Eine Übersicht über die Texte im NLCI-Korpus. 42

5.1 Die Struktur der NLCI-Ontologie. 71
5.2 Ergebnis-Beispiele für die einfache Wortauftrenn-Heurisik. 73
5.3 Mit der WordNet-Heuristik aufgetrennte Bezeichner. 74
5.4 Beispiele für die Auflösung des Phrasenkopfes. 76
5.5 Beispiele für atomare Sätze. 80
5.6 Beispiel-Ontologie (Auszug). 84
5.7 Die ermittelten Ontologie-Individuen r und ihre Bewertung B für die Anfrage *old ˆjanitor* und die Ontologie aus Tabelle 5.6. 87
5.8 Ableitungen, die für die Argumentermittlung verwendet werden (Auszug). 88
5.9 Signalwörter für Temporalausdrücke 92
5.10 Temporale Muster und Operatoren. 94
5.11 Ein (gekürztes) Beispiel für das Profil für gleichzeitige Aktionen (*do together*). 99

6.1 Übersicht über die Eingabetexte für die openHAB-Fallstudie. 110
6.2 Evaluationsergebnisse der openHAB-Fallstudie. 112
6.3 Übersicht über die Eingabetexte für die Alice-Fallstudie zur Verknüpfung der API-Elemente. 121
6.4 Evaluationsergebnisse der Alice-Fallstudie zur Verknüpfung der API-Elemente. 124
6.5 Evaluationsmetriken für die Methodenauswahl. 127
6.6 Detaillierte Auswertung der Methodenauswahl durch NLCI. 128
6.7 Evaluationsergebnisse für die Korrektur der Reihenfolge. 131

6.8 Übersicht über die Animationen für die Evaluation der Kontrollstruktur-erkennung. 134
6.9 Evaluationsergebnisse für die Erkennung der Kontrollstrukturen. 135

Literaturverzeichnis

[Abb83] ABBOTT, Russell J.: Program Design by Informal English Descriptions. In: *Commun. ACM* 26 (1983), November, Nr. 11, S. 882–894. http://dx.doi.org/10.1145/182.358441. – DOI 10.1145/182.358441. – ISSN 0001–0782 (zitiert auf Seite 72).

[Ama15] AMANN, Sonja: *Programmieren in natürlicher Sprache: Neustrukturierung der Alice-Bibliothek und Analyse der Namenskonventionen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Studienarbeit, Oktober 2015 (zitiert auf Seite 120).

[ART95] ANDROUTSOPOULOS, I. ; RITCHIE, G.d. ; THANISCH, P.: Natural language interfaces to databases – an introduction. In: *Natural Language Engineering* 1 (1995), März, Nr. 01, S. 29–81. http://dx.doi.org/10.1017/S135132490000005X. – DOI 10.1017/S135132490000005X. – ISSN 1469–8110 (zitiert auf Seite 58).

[Bal11] BALZERT, Heide: *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. 2. Aufl. 2004. Heidelberg : Spektrum Akademischer Verlag, 2011. – ISBN 978–3–8274–2903–2 (zitiert auf Seite 72).

[BB79] BALLARD, Bruce W. ; BIERMANN, Alan W.: Programming in Natural Language: NLC as a Prototype. In: *Proceedings of the 1979 annual conference*. New York, NY, USA : ACM, 1979 (ACM '79). – ISBN 0–89791–008–7, S. 228–237 (zitiert auf Seite 47).

[BB80] BIERMANN, Alan W. ; BALLARD, Bruce W.: Toward Natural Language Computation. In: *Comput. Linguist.* 6 (1980), April, Nr. 2, S. 71–86. – ISSN 0891–2017 (zitiert auf Seite 48).

[BBS83] BIERMANN, Alan W. ; BALLARD, Bruce W. ; SIGMON, Anne H.: An Experimental Study of Natural Language Programming. In: *International Journal of Man-Machine Studies* 18 (1983), Nr. 1, S. 71–87. http:

//dx.doi.org/10.1016/S0020-7373(83)80005-4. – DOI 10.1016/S0020–7373(83)80005–4. – ISSN 0020–7373 (zitiert auf Seite 48).

[BD04] BRÜGGE, Bernd ; DUTOIT, Allen H.: *Objektorientierte Softwaretechnik - mit UML, Entwurfsmustern und Java*. Pearson Studium, 2004. – ISBN 978–3–8273–7082–2 (zitiert auf den Seiten 72 und 76).

[Bel14] BELLEGARDA, Jerome R.: Spoken Language Understanding for Natural Interaction: The Siri Experience. In: MARIANI, Joseph (Hrsg.) ; ROSSET, Sophie (Hrsg.) ; GARNIER-RIZET, Martine (Hrsg.) ; DEVILLERS, Laurence (Hrsg.): *Natural Interaction with Robots, Knowbots and Smartphones*. Springer New York, 2014. – ISBN 978–1–4614–8279–6, S. 3–14 (zitiert auf den Seiten 3 und 144).

[Bes14] BEST, Jonathan Kim R.: *Programmieren in natürlicher Sprache: Der Eingabeassistent Input & Verify AliceNLP*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, März 2014 (zitiert auf den Seiten 43, 44 und 128).

[BJN06] BERGLUND, Anders ; JOHANSSON, Richard ; NUGUES, Pierre: A Machine Learning Approach to Extract Temporal Information from Texts in Swedish and Generate Animated 3D Scenes. In: *Proceedings of EACL-2006, 11th Conference of the European Chapter of the Association for Computational Linguistics*. Trento, Italy : Association for Computational Linguistics, 2006, S. 385–392 (zitiert auf Seite 61).

[Boo94] BOOCH, Grady: *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company, 1994. – ISBN 978–0–8053–5340–2 (zitiert auf Seite 22).

[BP12] BOND, Francis ; PAIK, Kyonghee: A Survey of Wordnets and thei Licenses. In: *Proceedings of the 6th Global WordNet Conference (GWC 2012)*. Matsue, 2012, S. 64–71 (zitiert auf Seite 22).

[Bur07] BURNARD, Lou: *Reference Guide for the British National Corpus (XML Edition)*. Oxford : Research Technologies Service at Oxford University Computing Services, 2007 (zitiert auf Seite 34).

[BVH+04] BECHHOFER, Sean ; VAN HARMELEN, Frank ; HENDLER, Jim ; HORROCKS, Ian ; MCGUINNESS, Deborah L. ; PATEL-SCHNEIDER, Peter F. ;

STEIN, Lynn A. ; DEAN, Mike (Hrsg.) ; SCHREIBER, Guus (Hrsg.): OWL Web Ontology Language Reference / W3C. 2004. – W3C Recommendation (zitiert auf Seite 24).

[CAB+00] CONWAY, Matthew ; AUDIA, Steve ; BURNETTE, Tommy ; COSGROVE, Dennis ; CHRISTIANSEN, Kevin: Alice: Lessons Learned from Building a 3D System for Novices. In: *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*. The Hague, The Netherlands : ACM Press, 2000. – ISBN 1–58113–216–6, S. 486–493 (zitiert auf den Seiten 10, 107 und 113).

[CEE+10] CARSTENSEN, Kai-Uwe (Hrsg.) ; EBERT, Christian (Hrsg.) ; EBERT, Cornelia (Hrsg.) ; JEKAT, Susanne J. (Hrsg.) ; KLABUNDE, Ralf (Hrsg.) ; LANGER, Hagen (Hrsg.): *Computerlinguistik und Sprachtechnologie – Eine Einführung*. Spektrum Akademischer Verlag, 2010. – ISBN 978–3–8274–2023–7 (Print), 978–3–8274–2224–8 (Online) (zitiert auf Seite 34).

[Cer13] CERF, Vinton G.: Software at Scale. In: *Commun. ACM* 56 (2013), Dezember, Nr. 12, S. 7–7. http://dx.doi.org/10.1145/2534706.2534708. – DOI 10.1145/2534706.2534708. – ISSN 0001–0782 (zitiert auf Seite 2).

[CFK11] COZZIE, Anthony ; FINNICUM, Murph ; KING, Samuel T.: Macho: Programming with Man Pages. In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. Berkeley, CA, USA : USENIX Association, 2011 (HotOS'13), S. 7–7 (zitiert auf Seite 50).

[Cho95] CHOMSKY, Noam: *The Minimalist Program*. Cambridge : MIT Press, 1995 (zitiert auf Seite 14).

[CK12] COZZIE, Anthony ; KING, Samuel: Macho: Writing Programs with Natural Language and Examples. 2012. – Forschungsbericht (zitiert auf Seite 50).

[CMSV09] CIMIANO, Philipp ; MÄDCHE, Alexander ; STAAB, Steffen ; VÖLKER, Johanna: Ontology Learning. Version: 2009. http://dx.doi.org/10.1007/978-3-540-92673-3. In: STAAB, Steffen (Hrsg.) ; STUDER, Rudi (Hrsg.): *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009 (International Handbooks on Information Systems). – DOI 10.1007/978–3–540–92673–3. – ISBN 978–3–540–70999–2 978–3–540–92673–3, S. 245–267 (zitiert auf Seite 62).

[Con97] CONWAY, Matthew J.: *Alice: Easy-to-Learn 3D Scripting for Novices*, Faculty of the School of Engineering and Applied Science, University of Virginia, Dissertation, Dezember 1997 (zitiert auf den Seiten 10, 107 und 113).

[CP04] CHONG, Stephen ; PUCELLA, Riccardo: A Framework for Creating Natural Language User Interfaces for Action-Based Applications. (2004), Dezember (zitiert auf Seite 55).

[CWJ07] CHAMBERS, Nathanael ; WANG, Shan ; JURAFSKY, Dan: Classifying Temporal Relations Between Events. In: *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. Prague, Czech Republic : Association for Computational Linguistics, 2007 (ACL '07), S. 173–176 (zitiert auf Seite 61).

[DAC10] DAMLJANOVIC, Danica ; AGATONOVIC, Milan ; CUNNINGHAM, Hamish: Natural Language Interfaces to Ontologies: Combining Syntactic Analysis and Ontology-Based Lookup through the User Interaction. In: AROYO, Lora (Hrsg.) ; ANTONIOU, Grigoris (Hrsg.) ; HYVÖNEN, Eero (Hrsg.) ; TEN TEIJE, Annette (Hrsg.) ; STUCKENSCHMIDT, Heiner (Hrsg.) ; CABRAL, Liliana (Hrsg.) ; TUDORACHE, Tania (Hrsg.): *The Semantic Web: Research and Applications* Bd. 6088. Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–13485–2, S. 106–120 (zitiert auf Seite 59).

[DGH+15] DESAI, Aditya ; GULWANI, Sumit ; HINGORANI, Vineet ; JAIN, Nidhi ; KARKARE, Amey ; MARRON, Mark ; R, Sailesh ; ROY, Subhajit: Program Synthesis using Natural Language. In: *arXiv:1509.00413 [cs]* (2015), September (zitiert auf Seite 51).

[Dij63] DIJKSTRA, Edsger W.: On the design of machine independent programming languages. In: *Annual Review in Automatic Programming* 3 (1963), S. 27–42. http://dx.doi.org/10.1016/S0066-4138(63)80003-8. – DOI 10.1016/S0066–4138(63)80003–8. – ISSN 0066–4138 (zitiert auf Seite 47).

[Dij64] DIJKSTRA, Edsger W.: Some Comments on the Aims of MIRFAC. In: *Commun. ACM* 7 (1964), März, Nr. 3, S. 190–. http://dx.doi.org/10.1145/

363958.364002. – DOI 10.1145/363958.364002. – ISSN 0001–0782 (zitiert auf Seite 47).

[dM08] DE MARNEFFE, Marie-Catherine ; MANNING, Christopher D.: The Stanford Typed Dependencies Representation. In: *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2008 (CrossParser '08). – ISBN 978–1–905593–50–7, S. 1–8 (zitiert auf Seite 17).

[dM15] DE MARNEFFE, Marie-Catherine ; MANNING, Christopher D.: Stanford Typed Dependencies Manual. 2015. – Forschungsbericht (zitiert auf den Seiten 18 und 183).

[ECD+04] ETZIONI, Oren ; CAFARELLA, Michael ; DOWNEY, Doug ; KOK, Stanley ; POPESCU, Ana-Maria ; SHAKED, Tal ; SODERLAND, Stephen ; WELD, Daniel S. ; YATES, Alexander: Web-scale Information Extraction in Knowitall: (Preliminary Results). In: *Proceedings of the 13th International Conference on World Wide Web*. New York, NY, USA : ACM, 2004 (WWW '04). – ISBN 1–58113–844–X, S. 100–110 (zitiert auf Seite 22).

[EHPVS09] ENSLEN, E. ; HILL, E. ; POLLOCK, L. ; VIJAY-SHANKER, K.: Mining Source Code to Automatically Split Identifiers for Software Analysis. In: *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009, S. 71–80 (zitiert auf Seite 74).

[FBCC+10] FERRUCCI, David ; BROWN, Eric ; CHU-CARROLL, Jennifer ; FAN, James ; GONDEK, David ; KALYANPUR, Aditya A. ; LALLY, Adam ; MURDOCK, J. W. ; NYBERG, Eric ; PRAGER, John ; SCHLAEFER, Nico ; WELTY, Chris: Building Watson: An Overview of the DeepQA Project. In: *AI Magazine* 31 (2010), Nr. 3, S. 59–79. http://dx.doi.org/10.1609/aimag.v31i3.2303. – DOI 10.1609/aimag.v31i3.2303. – ISSN 0738–4602 (zitiert auf Seite 31).

[Fel98] FELLBAUM, Christiane (Hrsg.): *WordNet: An Electronic Lexical Database*. Cambridge, Mass. [u.a.] : MIT Press, 1998 (Language, speech and communication). – ISBN 0–262–06197–X (zitiert auf Seite 20).

[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 (zitiert auf Seite 56).

[FK79] FRANCIS, W. N. ; KUCERA, Henry: *Brown Corpus Manual*. Providence, Rhode Island : Department of Linguistics, Brown University, 1979 (zitiert auf Seite 34).

[Gel10] GELHAUSEN, Tom: *Modellextraktion aus natürlichen Sprachen : Eine Methode zur systematischen Erstellung von Domänenmodellen*. Karlsruhe, KIT Scientific Publishing, Dissertation, April 2010 (zitiert auf den Seiten 56 und 58).

[Gen12] GENAID, Adrian: *Automatisierte Qualitätssicherung von User Stories und Assistenz bei der Entwicklung funktionaler Tests*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, Juni 2012 (zitiert auf Seite 64).

[GM14] GULWANI, Sumit ; MARRON, Mark: NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In: *SIGMOD Record*. Snowbird, USA : ACM, Juli 2014. – ISBN 978–1–4503–2376–5 (zitiert auf Seite 51).

[Goo98] GOODMAN, Danny: *Danny Goodman's AppleScript Handbook*. 2nd. iUniverse.com, Incorporated, 1998. – ISBN 0–9665514–1–9 (zitiert auf Seite 48).

[GOS09] GUARINO, Nicola ; OBERLE, Daniel ; STAAB, Steffen: What Is an Ontology? In: STAAB, Steffen (Hrsg.) ; RUDI STUDER, Dr. (Hrsg.): *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009 (International Handbooks on Information Systems). – ISBN 978–3–540–92673–3, S. 1–17 (zitiert auf Seite 22).

[Gru93] GRUBER, Thomas R.: A translation approach to portable ontology specifications. In: *Knowledge Acquisition* 5 (1993), Nr. 2, S. 199 – 220. http://dx.doi.org/10.1006/knac.1993.1008. – DOI 10.1006/knac.1993.1008. – ISSN 1042–8143 (zitiert auf Seite 22).

[GSY04] GIUNCHIGLIA, Fausto ; SHVAIKO, Pavel ; YATSKEVICH, Mikalai: S-Match: an Algorithm and an Implementation of Semantic Matching. In:

BUSSLER, Christoph (Hrsg.) ; DAVIES, John (Hrsg.) ; FENSEL, Dieter (Hrsg.) ; STUDER, Rudi (Hrsg.): *The Semantic Web: Research and Applications* Bd. 3053. Springer Berlin / Heidelberg, 2004. – ISBN 978–3–540–21999–6, S. 61–75 (zitiert auf Seite 75).

[GSY09] GIUNCHIGLIA, Fausto ; SHVAIKO, Pavel ; YATSKEVICH, Mikalai: Semantic Matching. In: LING, Liu (Hrsg.) ; TAMER, Özsu M. (Hrsg.): *Encyclopedia of Database Systems*. Springer US, 2009. – ISBN 978–0–387–35544–3, S. 2561–2566 (zitiert auf Seite 75).

[Ham12] HAMPEL, Sina: *Programmieren in natürlicher Sprache: Aufbau des Alice-Korpus*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelorarbeit, November 2012 (zitiert auf Seite 34).

[HF97] HAMP, Birgit ; FELDWEG, Helmut: GermaNet - a Lexical-Semantic Net for German. In: *Proceedings of the ACL workshop Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications*. Madrid, Spain, 1997 (zitiert auf Seite 22).

[HH10] HENRICH, Verena ; HINRICHS, Erhard: GernEdiT - The GermaNet Editing Tool. In: *Proceedings of the Seventh Conference on International Language Resources and Evaluation (LREC 2010)*. Valetta, Malta, 2010, S. 2228–2235 (zitiert auf Seite 22).

[Hil72] HILL, I. D.: Wouldn't it be nice if we could write computer programs in ordinary English - or would it? In: *Computer Bulletin* 16 (1972), Nr. 6, S. 306–312. – publisher = {British Computer Society} (zitiert auf Seite 47).

[Hor08] HORROCKS, Ian: Ontologies and the Semantic Web. In: *Communications of the ACM* 51 (2008), Dezember, Nr. 12, S. 58–67. http://dx.doi.org/10.1145/1409360.1409377. – DOI 10.1145/1409360.1409377. – ISSN 0001–0782 (zitiert auf Seite 25).

[Int14] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *The World in 2014: ICT Facts and Figures*. 2014 (zitiert auf Seite 2).

[JAGL86] JOHANSSON, Stig ; ATWELL, Eric ; GARSIDE, Roger ; LEECH, Geoffrey: *The Tagged LOB Corpus: Users' Manual*. Bergen : Norwegian Computing Centre for the Humanities, 1986 (zitiert auf Seite 34).

[JM09] JURAFSKY, Daniel ; MARTIN, James H.: *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. 2. ed., [Pearson International Edition]. Englewood Cliffs, NJ : Prentice Hall, Pearson Education International, 2009 (Prentice Hall series in artificial intelligence). – ISBN 0–13–504196–1 978–0–13–504196–3 (zitiert auf den Seiten 4 und 12).

[KEB10] KOLYA, Anup K. ; EKBAL, Asif ; BANDYOPADHYAY, Sivaji: JU_CSE_TEMP: A First Step Towards Evaluating Events, Time Expressions and Temporal Relations. In: *Proceedings of the 5th International Workshop on Semantic Evaluation*. Los Angeles, California : Association for Computational Linguistics, 2010 (SemEval '10), S. 345–350 (zitiert auf Seite 61).

[KFNM04] KNUBLAUCH, Holger ; FERGERSON, Ray W. ; NOY, Natalya F. ; MUSEN, Mark A.: The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In: MCILRAITH, Sheila A. (Hrsg.) ; PLEXOUSAKIS, Dimitris (Hrsg.) ; VAN HARMELEN, Frank (Hrsg.): *The Semantic Web – ISWC 2004* Bd. 3298. Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–23798–3, S. 229–243 (zitiert auf Seite 24).

[KGM11] KNÖLL, Roman ; GASIUNAS, Vaidas ; MEZINI, Mira: Naturalistic Types. In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA : ACM, 2011 (Onward! 2011). – ISBN 978–1–4503–0941–7, S. 33–48 (zitiert auf Seite 55).

[Kie16] KIESEL, Viktor: *Optimieren von Wortartmarkiererergebnissen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelorarbeit, April 2016 (zitiert auf Seite 13).

[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), S. 92–99 (zitiert auf den Seiten 56 und 103).

[KLT14] KÖRNER, Sven J. ; LANDHÄUSSER, Mathias ; TICHY, Walter F.: Transferring Research Into the Real World: How to Improve RE with AI in the Automotive Industry. In: *1st International Workshop on Artificial Intelligence for Requirements Engineering*, 2014 (zitiert auf Seite 57).

[KM06] KNÖLL, Roman ; MEZINI, Mira: Pegasus: First Steps Toward a Naturalistic Programming Language. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. Portland, Oregon, USA : ACM, 2006 (OOPSLA '06). – ISBN 1–59593–491–X, S. 542–559 (zitiert auf Seite 55).

[Koc15] KOCYBIK, Markus: *Projektion von gesprochener Sprache auf eine Handlungsrepräsentation*, Karlsruher Institut für Technologie (KIT) - IPD Tichy, Bachelorarbeit, Juli 2015 (zitiert auf Seite 145).

[Kör14] KÖRNER, Sven J.: *RECAA – Werkzeugunterstützung in der Anforderungserhebung*, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie (KIT), Dissertation, Februar 2014 (zitiert auf den Seiten 56, 58 und 112).

[Kri94] KRISTEN, Gerald: *Object orientation - the KISS method: from information architecture to information system.* Addison-Wesley, 1994. – ISBN 978–0–201–42299–3 (zitiert auf Seite 72).

[KSKD13] KESZÖCZE, Oliver ; SOEKEN, Mathias ; KUKSA, Eugen ; DRECHSLER, Rolf: Lips: An IDE for model driven engineering based on natural language processing. In: *2013 1st International Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE).* San Francisco, CA, USA, Mai 2013, S. 31–38 (zitiert auf Seite 57).

[LA10] LIEBERMAN, Henry ; AHMAD, Moin: Knowing What You're Talking About: Natural Language Programming of a Multi-Player Online Game. In: CYPHER, Allen (Hrsg.) ; DONTCHEVA, Mira (Hrsg.) ; LAU, Tessa (Hrsg.) ; NICHOLS, Jeffrey (Hrsg.): *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010. – ISBN 978–0–12–381541–5, S. 331–343 (zitiert auf Seite 49).

[Len95] LENAT, Douglas B.: CYC: A Large-scale Investment in Knowledge Infrastructure. In: *Commun. ACM* 38 (1995), November, Nr. 11, S. 33–38. http:

//dx.doi.org/10.1145/219717.219745. – DOI 10.1145/219717.219745. – ISSN 0001–0782 (zitiert auf den Seiten 22 und 56).

[LFMS12] LOPEZ, Vanessa ; FERNÁNDEZ, Miriam ; MOTTA, Enrico ; STIELER, Nico: PowerAqua: Supporting Users in Querying and Exploring the Semantic Web. In: *Semantic Web* 3 (2012), Januar, Nr. 3, S. 249–265. http://dx.doi.org/10.3233/SW-2011-0030. – DOI 10.3233/SW–2011–0030 (zitiert auf Seite 59).

[LG12] LANDHÄUSSER, Mathias ; GENAID, Adrian: Connecting User Stories and Code for Test Development. In: *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. Piscataway, NJ, USA : IEEE Press, 2012 (RSSE '12). – ISBN 978–1–4673–1759–7, S. 33–37 (zitiert auf den Seiten 52 und 64).

[LGS13] LE, Vu ; GULWANI, Sumit ; SU, Zhendong: Smartsynth: Synthesizing Smartphone Automation Scripts from Natural Language. In: *MobSys'13* Bd. 2, 2013, S. 5 (zitiert auf Seite 50).

[LH15] LANDHÄUSSER, Mathias ; HUG, Ronny: Text Understanding for Programming in Natural Language: Control Structures. In: *Proceedings of the 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2015 (zitiert auf Seite 98).

[LHT14] LANDHÄUSSER, Mathias ; HEY, Tobias ; TICHY, Walter F.: Deriving Timelines From Texts. In: *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2014. – ISBN 978–1–4503–2846–3, S. 45–51 (zitiert auf Seite 91).

[Lie06] LIEBIG, Thorsten: Reasoning with OWL – System Support and Insights / Universität Ulm. 2006 (2006-04). – Ulmer Informatik-Berichte (zitiert auf Seite 23).

[LKK+15] LANDHÄUSSER, Mathias ; KÖRNER, Sven J. ; KEIM, Jan ; TICHY, Walter F. ; KRISCH, Jennifer: DeNom: A Tool to Find Problematic Nominalizations using NLP. In: *2nd International Workshop on Artificial Intelligence for Requirements Engineering*. Ottawa, Canada, August 2015 (zitiert auf den Seiten 57 und 112).

[LL04] LIU, Hugio ; LIEBERMAN, Henry: Toward a Programmatic Semantics of Natural Language. In: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, 2004, S. 281 –282 (zitiert auf Seite 53).

[LL05a] LAPATA, Mirella ; LASCARIDES, Alex: Inferring Sentence-internal Temporal Relations. In: *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, 2005, S. 153–160 (zitiert auf Seite 61).

[LL05b] LIU, Hugo ; LIEBERMAN, Henry: Metafor: Visualizing Stories as Code. In: *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces*. San Diego, California, USA : ACM, 2005. – ISBN 1–58113–894–6, S. 305–307 (zitiert auf Seite 54).

[LL05c] LIU, Hugo ; LIEBERMAN, Henry: Programmatic Semantics for Natural Language Interfaces. In: *CHI '05 extended abstracts on Human factors in computing systems*. Portland, OR, USA : ACM, 2005 (CHI '05). – ISBN 1–59593–002–7, S. 1597–1600 (zitiert auf Seite 54).

[LL06] LIEBERMAN, Henry ; LIU, Hugo: Feasibility Studies for Programming in Natural Language. In: LIEBERMAN, Henry (Hrsg.) ; PATERNÒ, Fabio (Hrsg.) ; WULF, Volker (Hrsg.): *End User Development*. Springer Netherlands, Januar 2006 (Human-Computer Interaction Series 9). – ISBN 978–1–4020–4220–1 978–1–4020–5386–3, S. 459–473 (zitiert auf den Seiten 8 und 53).

[LM06] LITTLE, Greg ; MILLER, Robert C.: Translating keyword commands into executable code. In: *Proceedings of the 19th annual ACM symposium on User interface software and technology*. Montreux, Switzerland : ACM, 2006 (UIST '06). – ISBN 1–59593–313–1, S. 135–144 (zitiert auf Seite 54).

[LWT] LANDHÄUSSER, Mathias ; WEIGELT, Sebastian ; TICHY, Walter F.: NLCI: A Natural Language Command Interpreter. In: *Automated Software Engineering*. – ISSN 0828–8910, 1573–7535. – Akzeptiert für die „Special Issue: AI and SE Synergies“ (zitiert auf Seite 78).

[Mar09] MARTIN, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ : Prentice Hall, 2009 (Robert C. Martin Series). – ISBN 978–0–13–235088–4 (zitiert auf Seite 72).

[MBMLM13] MARTINEZ-BARCO, Patricio ; MÉTAIS, Elisabeth ; LLOPIS, Fernando ; MOREDA, Paloma: Editorial: An Overview of the Applications of Natural Language to Information Systems. In: *Data Knowl. Eng.* 88 (2013), November, S. 109–112. http://dx.doi.org/10.1016/j.datak.2013.08.001. – DOI 10.1016/j.datak.2013.08.001. – ISSN 0169–023X (zitiert auf Seite 58).

[McE05] MCENERY, Tony: Corpus Linguistics. In: MITKOV, Ruslan (Hrsg.): *The Oxford Handbook of Computational Linguistics*. Oxford : Oxford University Press, 2005. – ISBN 978–0–19–927634–9, S. 448–463 (zitiert auf Seite 34).

[MGA13] MANSHADI, Mehdi H. ; GILDEA, Daniel ; ALLEN, James F.: Integrating Programming by Example and Natural Language Programming, 2013 (zitiert auf Seite 50).

[Mil81] MILLER, L.A.: Natural language programming: Styles, strategies, and contrasts. In: *IBM Systems Journal* 20 (1981), Nr. 2, S. 184–215. http://dx.doi.org/10.1147/sj.202.0184. – DOI 10.1147/sj.202.0184. – ISSN 0018–8670 (zitiert auf den Seiten 47 und 52).

[Mil95] MILLER, George A.: WordNet: A Lexical Database for English. In: *Commun. ACM* 38 (1995), November, Nr. 11, S. 39–41. http://dx.doi.org/10.1145/219717.219748. – DOI 10.1145/219717.219748. – ISSN 0001–0782 (zitiert auf Seite 20).

[Mil11] MILLER, James E. ; LODGE, Ken (Hrsg.): *A Critical Introduction to Syntax*. Bloomsbury, 2011 (Continuum Critical Introductions to Linguistics). – ISBN 978–0–8264–9703–1 (zitiert auf Seite 14).

[MLL06] MIHALCEA, Rada ; LIU, Hugo ; LIEBERMAN, Henry: NLP (Natural Language Processing) for NLP (Natural Language Programming). In: GELBUKH, Alexander (Hrsg.): *Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing 2006)* Bd. 3878. Mexico City, Mexico : Springer Berlin / Heidelberg,

Februar 2006 (Lecture Notes in Computer Science), S. 319–330 (zitiert auf den Seiten 54 und 103).

[MSB+14] MANNING, Christopher D. ; SURDEANU, Mihai ; BAUER, John ; FINKEL, Jenny ; BETHARD, Steven J. ; MCCLOSKY, David: The Stanford CoreNLP Natural Language Processing Toolkit. In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland : Association for Computational Linguistics, Juni 2014, S. 55–60 (zitiert auf den Seiten 13 und 18).

[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. – ISSN 0891–2017 (zitiert auf den Seiten 13, 34 und 181).

[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*. Sydney, Australia : Association for Computational Linguistics, 2006, S. 753–760 (zitiert auf Seite 61).

[Nel05] NELSON, Graham: *Natural Language, Semantic Analysis and Interactive Fiction*. Juni 2005 (zitiert auf Seite 49).

[Nem07] NEMEC, P.: Automatic Analysis of Temporal Relations within a Discourse. In: *14th International Symposium on Temporal Representation and Reasoning*, 2007, S. 117–128 (zitiert auf Seite 61).

[Nik15] NIKOLOV, Pavel: *Programmieren in natürlicher Sprache: Auflösen von Referenzen in englischen Texten*, Karlsruher Institut für Technologie (KIT) - IPD Tichy, Diplomarbeit, Juni 2015 (zitiert auf Seite 144).

[Noy09] NOY, Natalya F.: Ontology Mapping. In: STAAB, Steffen (Hrsg.) ; STUDER, Rudi (Hrsg.): *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009 (International Handbooks on Information Systems). – ISBN 978–3–540–92673–3, S. 573–590 (zitiert auf Seite 75).

[Oba13] *President Obama asks America to learn computer science*. Dezember 2013 (zitiert auf Seite 2).

[Oba16] *FACT SHEET: President Obama Announces Computer Science For All Initiative*. https://www.whitehouse.gov/the-press-office/2016/01/30/fact-sheet-president-obama-announces-computer-science-all-initiative-0, Januar 2016 (zitiert auf Seite 2).

[Ohl07] OHLBACH, Hans J.: Computational Treatment of Temporal Notions: The CTTN–System. In: SCHILDER, Frank (Hrsg.) ; KATZ, Graham (Hrsg.) ; PUSTEJOVSKY, James (Hrsg.): *Annotating, Extracting and Reasoning about Time and Events*. Springer Berlin / Heidelberg, Januar 2007 (Lecture Notes in Computer Science 4795). – ISBN 978–3–540–75988–1 978–3–540–75989–8, S. 72–87 (zitiert auf Seite 60).

[ope] OPENHAB UG (HAFTUNGSBESCHRÄNKT): *openhab – empowering the smart home*. http://www.openhab.org/, (zitiert auf den Seiten 9 und 107).

[Pet12] PETERS, Oleg: *Programmieren in natürlicher Sprache: Extraktion einer Alice-API-Ontologie*, Karlsruher Institut für Technologie (KIT) - IPD Tichy, Bachelorarbeit, September 2012 (zitiert auf Seite 117).

[PGA+13] PAZOS R., Rodolfo A. ; GONZÁLEZ B., Juan J. ; AGUIRRE L., Marco A. ; MARTINEZ F., José A. ; FRAIRE H., Héctor J.: Natural Language Interfaces to Databases: An Analysis of the State of the Art. In: CASTILLO, Oscar (Hrsg.) ; MELIN, Patricia (Hrsg.) ; KACPRZYK, Janusz (Hrsg.): *Recent Advances on Hybrid Intelligent Systems*. Springer Berlin Heidelberg, 2013 (Studies in Computational Intelligence 451). – ISBN 978–3–642–33020–9 978–3–642–33021–6, S. 463–480 (zitiert auf Seite 58).

[PHS+03] PUSTEJOVSKY, James ; HANKS, Patrick ; SAURÍ, Roser ; SEE, Andrew ; GAIZAUSKAS, Robert ; SETZER, Andrea ; RADEV, Dragomir ; SUNDHEIM, Beth ; DAY, David ; FERRO, Lisa ; LAZO, Marcia: The TIMEBANK Corpus. In: *Corpus linguistics* Bd. 2003, 2003, S. 40 (zitiert auf Seite 60).

[PIS+05] PUSTEJOVSKY, James ; INGRIA, Robert ; SAURÍ, Roser ; CASTAO, José ; LITTMAN, Jessica ; GAIZAUSKAS, Rob ; SETZER, Andrea ; KATZ, Graham ; MANI, Inderjeet: The Specification Language TimeML. In: *The language of time: A reader* (2005), S. 545–557 (zitiert auf Seite 60).

[PKLS05] PUSTEJOVSKY, James ; KNIPPEN, Robert ; LITTMAN, Jessica ; SAURÍ, Roser: Temporal and Event Information in Natural Language Text. In:

Language Resources and Evaluation 39 (2005), Mai, Nr. 2-3, S. 123–164. http://dx.doi.org/10.1007/s10579-005-7882-7. – DOI 10.1007/s10579–005–7882–7. – ISSN 1574–020X, 1572–0218 (zitiert auf Seite 60).

[Pla03] PLAG, Ingo: *Word-Formation in English*. Cambridge; New York : Cambridge University Press, 2003 (Cambridge Textbooks in Linguistics). – ISBN 978–0–521–52563–3 (zitiert auf Seite 76).

[PM96] PANE, John F. ; MYERS, Brad A.: Usability Issues in the Design of Novice Programming Systems / School of Computer Science, Carnegie Mellon University. Pittsburgh, Pennsylvania, August 1996 (CMU-CS-96-132). – Technical Report (zitiert auf Seite 53).

[PMM02] PANE, John F. ; MYERS, Brad A. ; MILLER, L.B.: Using HCI techniques to design a more usable programming system. In: *IEEE 2002 Symposia on Human Centric Computing Languages and Environments, 2002. Proceedings*, 2002, S. 198–206 (zitiert auf Seite 53).

[PRM01] PANE, John F. ; RATANAMAHATANA, Chotirat ; MYERS, Brad A.: Studying the Language and Structure in Non-programmers' Solutions to Programming Problems. In: *International Journal of Human-Computer Studies* 54 (2001), Nr. 2, S. 237–264. http://dx.doi.org/10.1006/ijhc.2000.0410. – DOI 10.1006/ijhc.2000.0410. – ISSN 1071–5819 (zitiert auf Seite 53).

[PRZH00] PRICE, David ; RILOFFF, Ellen ; ZACHARY, Joseph ; HARVEY, Brandon: NaturalJava: A Natural Language Interface for Programming in Java. In: *Proceedings of the 5th international conference on Intelligent user interfaces*. New Orleans, Louisiana, USA : ACM, 2000 (IUI '00). – ISBN 1–58113–134–8, S. 207–211 (zitiert auf Seite 48).

[RFJ08] RATIU, D. ; FEILKAS, M. ; JURJENS, J.: Extracting Domain Ontologies from Domain Specific APIs. In: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, 2008, S. 203 –212 (zitiert auf Seite 64).

[RN09] RUSSELL, Stuart ; NORVIG, Peter: *Artificial Intelligence: A Modern Approach*. 3 edition. Upper Saddle River, NJ : Prentice Hall, 2009. – ISBN 978–0–13–604259–4 (zitiert auf Seite 62).

[Sab04] SABOU, Marta: Extracting Ontologies from Software Documentation: A Semi-automatic Method and its Evaluation. In: *Workshop on Ontology Learning and Population (ECAI-OLP)*. Valencia, Spain, August 2004 (zitiert auf Seite 63).

[Sam66] SAMMET, Jean E.: The Use of English as a Programming Language. In: *Commun. ACM* 9 (1966), März, Nr. 3, S. 228–230. http://dx.doi.org/10.1145/365230.365274. – DOI 10.1145/365230.365274. – ISSN 0001–0782 (zitiert auf Seite 47).

[San95] SANTORINI, Beatrice: Part-of-speeach Tagging Guidlines for the Penn Treebank Project / Department of Computer and Information Science, University of Pennsylvania. 1995 (MS-CIS-90–47 (3rd Revision, 2nd printing)). – Technical Report (zitiert auf den Seiten 12 und 181).

[Sau11] SAUTTER, Guido: *Efficient Conversion of Scientific Legacy Documents into Semantic Web Resources using biosystematics as a working example /*. Karlsruhe, Karlsruher Institut für Technologie (KIT) – IPD Böhm, Dissertation, Februar 2011 (zitiert auf Seite 19).

[SBA07] SAUTTER, Guido ; BÖHM, Klemens ; AGOSTI, Donat: Semi-Automated XML Markup of Biosystematic Legacy Literature with the GoldenGate Editor. In: *Proceedings of the Pacific Symposium on Biocomputing (PSB'07)* Bd. 12. Wailea, HI, USA, 2007, S. 391–402 (zitiert auf Seite 18).

[SBMN13] SOCHER, Richard ; BAUER, John ; MANNING, Christopher D. ; NG, Andrew Y.: Parsing With Compositional Vector Grammars. In: *ACL*. 2013 (zitiert auf Seite 16).

[Sch07] SCHILDER, Frank: Event Extraction and Temporal Reasoning in Legal Documents. In: SCHILDER, Frank (Hrsg.) ; KATZ, Graham (Hrsg.) ; PUSTEJOVSKY, James (Hrsg.): *Annotating, Extracting and Reasoning about Time and Events*. Springer Berlin / Heidelberg, Januar 2007 (Lecture Notes in Computer Science 4795). – ISBN 978–3–540–75988–1 978–3–540–75989–8, S. 59–71 (zitiert auf Seite 60).

[Sek14] SEKER, Ali: *Programmieren in natürlicher Sprache: Erfassung von Methodenargumenten aus natürlicher Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, 2014 (zitiert auf Seite 87).

[SK10] SANAMPUDI, Suresh K. ; KUMARI, G. V.: Temporal Reasoning in Natural Language Processing: A Survey. In: *International Journal of Computer Applications* 1 (2010), Februar, Nr. 4, S. 68–72. http://dx.doi.org/10.5120/100-209. – DOI 10.5120/100–209 (zitiert auf Seite 62).

[SKW07] SUCHANEK, Fabian M. ; KASNECI, Gjergji ; WEIKUM, Gerhard: Yago: A Core of Semantic Knowledge. In: *Proceedings of the 16th International Conference on World Wide Web*. New York, NY, USA : ACM, 2007 (WWW '07). – ISBN 978–1–59593–654–7, S. 697–706 (zitiert auf Seite 22).

[SLM+02] SINGH, Push ; LIN, Thomas ; MUELLER, Erik T. ; LIM, Grace ; PERKINS, Travell ; ZHU, Wan L.: Open Mind Common Sense: Knowledge Acquisition from the General Public. In: MEERSMAN, Robert (Hrsg.) ; TARI, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*. Springer Berlin Heidelberg, Oktober 2002 (Lecture Notes in Computer Science 2519). – ISBN 978–3–540–00106–5 978–3–540–36124–4, S. 1223–1237 (zitiert auf den Seiten 49 und 54).

[Stu09] STUCKENSCHMIDT, Heiner: *Ontologien : Konzepte, Technologien und Anwendungen*. Berlin : Springer, 2009 (Informatik im Fokus). – ISBN 978–3–540–79330–4 (zitiert auf den Seiten 24 und 25).

[STV08] SIMPERL, Elena ; TEMPICH, Christoph ; VRANDECIC, Denny: A methodology for Ontology Learning. In: BUITELAAR, Paul (Hrsg.) ; CIMIANO, Philipp (Hrsg.): *Proceedings of the 2008 conference on Ontology Learning and Population: Bridging the Gap between Text and Knowledge* Bd. 167. Amsterdam : IOS Press, Juni 2008 (Frontiers in Artificial Intelligence and Applications). – ISBN 978–1–58603–818–2, S. 225–249 (zitiert auf Seite 63).

[SWD12] SOEKEN, Mathias ; WILLE, Robert ; DRECHSLER, Rolf: Assisted Behavior Driven Development Using Natural Language Processing. In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*. Berlin, Heidelberg : Springer-Verlag, 2012 (TOOLS'12). – ISBN 978–3–642–30560–3, S. 269–287 (zitiert auf Seite 57).

[TDS+13] THUMMALAPENTA, S. ; DEVAKI, P. ; SINHA, S. ; CHANDRA, S. ; GNANASUNDARAM, S. ; NAGARAJ, D.D. ; KUMAR, S. ; KUMAR, S.: Efficient and Change-resilient Test Automation: An Industrial Case Study. In: *2013 35th International Conference on Software Engineering (ICSE)*, 2013, S. 1002–1011 (zitiert auf Seite 51).

[TKMS03] TOUTANOVA, Kristina ; KLEIN, Dan ; MANNING, Christopher D. ; SINGER, Yoram: Feature-rich Part-of-Speech Tagging with a Cyclic Dependency Network. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology* Bd. 1. Edmonton, Canada : Association for Computational Linguistics, 2003 (NAACL '03), S. 173–180 (zitiert auf Seite 13).

[TM00] TOUTANOVA, Kristina ; MANNING, Christopher D.: Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In: *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics* Bd. 13. Hong Kong : Association for Computational Linguistics, 2000 (EMNLP '00), S. 63–70 (zitiert auf Seite 13).

[TSSC12] THUMMALAPENTA, S. ; SINHA, S. ; SINGHANIA, N. ; CHANDRA, Satish: Automating Test Automation. In: *2012 34th International Conference on Software Engineering (ICSE)*, 2012, S. 881–891 (zitiert auf den Seiten 51 und 52).

[UC11] UNGER, Christina ; CIMIANO, Philipp: Pythia: Compositional Meaning Construction for Ontology-Based Question Answering on the Semantic Web. In: *Natural Language Processing and Information Systems - 16th International Conference on Applications of Natural Language to Information Systems, NLDB 2011, Alicante, Spain, June 28-30, 2011. Proceedings*, 2011, S. 153–160 (zitiert auf Seite 59).

[VC05] VADAS, David ; CURRAN, James R.: Programming With Unrestricted Natural Language. In: *Proceedings of the Australasian Language Technology Workshop*. Sydney, Australia, Dezember 2005, S. 191–199 (zitiert auf Seite 54).

[Wal75] WALTZ, David: Natural Language Access to a Large Data Base: An Engineering Approach. In: *Proceedings of the 4th International Joint Conference on Artificial Intelligence* Bd. 1. Tblisi, USSR : Morgan Kaufmann Publishers Inc., 1975 (IJCAI'75), S. 868–872 (zitiert auf Seite 58).

[Wei12] WEIGELT, Sebastian: *Programmieren in natürlicher Sprache: Aufbau einer Alice-Ontologie – Korpus-Ontologie-Assoziation*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Studienarbeit, 2012 (zitiert auf Seite 75).

[Wei14] WEIGELT, Sebastian: *Programmieren in natürlicher Sprache: Entitäten-Erkennung und -Verknüpfung innerhalb domänenfremder Modelle mittels einer punktsystembasierten Abbildung*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, 2014 (zitiert auf Seite 78).

[Wel09] WELLER, Kathrin: Ontologien: Stand und Entwicklung der Semantik für WorldWideWeb. In: *LIBREAS. Library Ideas* 5 (2009), Nr. 2 (15) (zitiert auf Seite 22).

[WT15] WEIGELT, S. ; TICHY, W.F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)* Bd. 2, 2015, S. 819–820 (zitiert auf den Seiten 112 und 144).

[WXZY07] WANG, Chong ; XIONG, Miao ; ZHOU, Qi ; YU, Yong: PANTO: A Portable Natural Language Interface to Ontologies. In: FRANCONI, Enrico (Hrsg.) ; KIFER, Michael (Hrsg.) ; MAY, Wolfgang (Hrsg.): *The Semantic Web: Research and Applications* Bd. 4519. Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–72666–1, S. 473–487 (zitiert auf Seite 58).

[YCO99] YANG, Hongji ; CUI, Zhan ; O'BRIEN, P.: Extracting Ontologies from Legacy Systems for Understanding and Re-engineering. In: *Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International*, 1999, S. 21 –26 (zitiert auf den Seiten 63 und 69).

[ZWRH06] ZHANG, Yonggang ; WITTE, Rene ; RILLING, Juergen ; HAARSLEV, Volker: An Ontology-based Approach for Traceability Recovery. In: *3rd*

International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), 2006, S. 36–43 (zitiert auf den Seiten 63 und 69).