Karlsruhe Institute of Technology

Zielsystemunabhängige Quelltextsynthese aus natürlicher Sprache

Masterarbeit
von

Viktor Kiesel

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

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl.-Inform. Sebastian Weigelt

Bearbeitungszeit: 23.05.2019 – 10.12.2019

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 10.12.2019

(**Viktor Kiesel**)

Publikationsgenehmigung

Melder der Publikation

Hildegard Sauer

Institut für Programmstrukturen und Datenorganisation (IPD)
Lehrstuhl für Programmiersysteme
Leiter Prof. Dr. Walter F. Tichy

+49 721 608-43934
hildegard.sauer@kit.edu

Erklärung des Verfassers

Ich räume dem Karlsruher Institut für Technologie (KIT) dauerhaft ein einfaches Nutzungsrecht für die Bereitstellung einer elektronischen Fassung meiner Publikation auf dem zentralen Dokumentenserver des KIT ein.

Ich bin Inhaber aller Rechte an dem Werk; Ansprüche Dritter sind davon nicht berührt.

Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

Art der Abschlussarbeit:	Masterarbeit
Titel:	Zielsystemunabhängige Quelltextsynthese aus natürlicher Sprache
Datum:	10.12.2019
Name:	Viktor Kiesel

Karlsruhe, 10.12.2019

(Viktor Kiesel)

Kurzfassung

Die Steuerung von Maschinen, wie Robotern und intelligenter Assistenten, durch Sprache gewinnt im täglichen Leben immer stärker an Bedeutung. In Zukunft werden Nutzer erwarten, dass Maschinen komplexe Befehlssequenzen verstehen, die zum Beispiel Kontrollstrukturen enthalten. In dieser Arbeit wurde daher das Thema der zielsystemunabhängigen Quelltextsynthese aus natürlicher Sprache untersucht. Dabei wurden Werkzeuge entwickelt, die aus aufbereiteten Sprachinformationen, wie beispielsweise extrahierten Kontrollstrukturen, einen abstrakten Syntaxbaum synthetisieren. Dazu wurden Strukturen des Syntaxbaumes durch Mustererkenner extrahiert und anschließend verbunden. Der Syntaxbaum wurde durch Besucher und Schablonen-Systeme in Quelltext übersetzt. Mit diesem Vorgehen konnte Quelltext für einen Küchenroboter, einen Legoroboter und ein System zum Erstellen von Aktivitätsdiagrammen erzeugt werden. Die entwickelten Werkzeuge sind in der Lage Quelltext in Java, Python, C und PlantUML zu erzeugen. Bei einer Online-Studie akzeptierten Probanden in einem Mehrheitsentscheid 47,68% der Aktivitätsdiagramme als korrekt. Vollständig korrekter Quelltext wurde bei einer manuellen Evaluation in 32,91% der Fälle erzeugt. Die Ergebnisse zeigen, dass Quelltext mit Kontrollstrukturen aus natürlicher Sprache synthetisiert werden kann.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Zielsetzung . . . 2
- 1.2 Anwendungsbeispiele . . . 2
- 1.3 Struktur der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Übersetzerbau . . . 5
  - 2.1.1 Grundbegriffe . . . 5
  - 2.1.2 Aufbau eines Kompilierer . . . 7
- 2.2 Computerlinguistik . . . 8
  - 2.2.1 Syntax, Semantik und Pragmatik . . . 8
  - 2.2.2 Computerlinguistik-Fließbänder . . . 9
  - 2.2.3 Maschinelles Lernen . . . 11
  - 2.2.4 Ontologie . . . 12
- 2.3 Schablonen-Systeme . . . 12
- 2.4 Gütekriterien . . . 13

3 PARSE-Projekt **17**
- 3.1 Grundsätzliche Entwurfsprinzipien . . . 17
- 3.2 Architektur . . . 18
  - 3.2.1 Vorverarbeitung . . . 18
  - 3.2.2 Agenten . . . 19
    - 3.2.2.1 Aktionserkennung . . . 19
    - 3.2.2.2 Synthese von Methodendefinitionen . . . 20
    - 3.2.2.3 Kontextanalyse . . . 20
    - 3.2.2.4 Koreferenzauflösung . . . 20
    - 3.2.2.5 Aktionsidentitätsauflösung . . . 20
    - 3.2.2.6 Kontrollstrukturen . . . 21
    - 3.2.2.7 Domänenzuordnung . . . 21
    - 3.2.2.8 Disfluenzenerkennung . . . 21
    - 3.2.2.9 Mehrdeutigkeitsauflösung . . . 22
    - 3.2.2.10 Dialogagent . . . 22
  - 3.2.3 Ontologie . . . 22
  - 3.2.4 Nachverarbeitung . . . 22
- 3.3 Zielsysteme . . . 23
- 3.4 Einordnung dieser Arbeit ins Projekt . . . 23

4 Verwandte Arbeiten **25**
- 4.1 Quelltextsynthese . . . 25
  - 4.1.1 Ansätze basierend auf Sprachverarbeitungsfließbändern . . . 26
  - 4.1.2 Ansätze basierend auf neuronalen Netzen . . . 27
  - 4.1.3 Robotersteuerung . . . 28

4.1.4 Abstrakter Syntaxbaum . . . 29
4.2 Programmtransformation . . . 30
4.3 Zusammenfassung . . . 31

5 Analyse und Entwurf **33**
5.1 Problemstellung und Teilprobleme . . . 34
5.2 Voranalyse . . . 34
5.3 Ontologie . . . 35
5.4 Synthese des abstrakten Syntaxbaumes . . . 35
5.4.1 Mustererkennung . . . 37
5.4.2 Strukturen des abstrakten Syntaxbaums . . . 39
5.4.2.1 Aktionen und Bedingungen . . . 42
5.4.2.2 Sequenz . . . 44
5.4.2.3 Kontrollstrukturen . . . 45
5.4.2.4 Parallelität . . . 47
5.4.3 Verbindung der Strukturen . . . 48
5.4.3.1 Aufbau der Pfade . . . 49
5.4.3.2 Pfadverschmelzung . . . 50
5.4.3.3 Schwierigkeiten und Ähnlichkeitsfunktion . . . 51
5.4.4 Folgefehler . . . 52
5.5 Quelltexterzeugung . . . 53
5.5.1 Aufbereitung des abstrakten Syntaxbaumes . . . 54
5.5.2 Zielsystemunabhängige Quelltextsynthese . . . 54
5.6 Maschinelles Lernen . . . 55
5.7 Zusammenfassung . . . 56

6 Implementierung **57**
6.1 Projektorganisation . . . 57
6.2 Synthese des abstrakten Syntaxbaums . . . 58
6.2.1 Ablauf . . . 58
6.2.2 Mustererkennung . . . 60
6.2.2.1 Wurzel des Syntaxbaums . . . 60
6.2.2.2 Blätter des Syntaxbaums . . . 60
6.2.2.3 Verzweigungen . . . 62
6.2.2.4 Schleifen . . . 63
6.2.2.5 Parallelität . . . 64
6.2.3 Verbindung der Strukturen . . . 64
6.2.4 Ausführungsreihenfolge von Sequenzen . . . 65
6.2.5 Analyse des Syntaxbaumes . . . 66
6.3 Nachverarbeitungsfließband . . . 67
6.3.1 ASTTreeBuilder . . . 67
6.3.2 CodeGen . . . 68
6.3.3 CodeInjector . . . 72
6.3.4 ParseGraphDrawer . . . 73

7 Evaluation **77**
7.1 Qualität vorhandener PARSE-Agenten . . . 77
7.2 Hypothesen . . . 78
7.3 Synthese des abstrakten Syntaxbaumes . . . 79
7.3.1 Automatische Evaluation des Syntaxbaumes . . . 79
7.3.2 Darstellung der Syntaxbäume . . . 80
7.3.3 Aufbau der Studie . . . 80

7.3.4 Ergebnisse 82
7.3.4.1 Fehleranalyse durch Wahlverfahren 83
7.3.4.2 Fehleranalyse durch Verschmelzung der Fehlerfälle 85
7.3.5 Probleme im Aufbau der Studie 87
7.3.6 Evaluation der Methoden und Parameter 88
7.3.7 Fehlerbeschreibung 90
7.3.7.1 Syntaktische Fehler 90
7.3.7.2 Semantische Fehler 91
7.4 Syntaxbäume 93
7.5 Zielsystemunabhängigkeit 94
7.6 Zusammenfassung und Fazit 96

8 Zusammenfassung und Ausblick **105**

Literaturverzeichnis **109**

Anhang **113**
A Kurzbeschreibung bei *Prolific* 113
B Einführung der Online-Studie 113
B.1 Sentence 114
B.2 Activities 114
B.3 Conditions, Branches and Loops 115
B.4 Concurrency 116
B.5 No errors and other errors 116

Abbildungsverzeichnis

2.1 Ein gerichteter Graph mit Übergangswahrscheinlichkeiten als Kantengewichte. 6
2.2 Syntaxbaum mit zugehöriger kontextfreier Grammatik 7
2.3 Fließbandarchitektur eines Kompilierers 7
2.4 Das Standard-Modell des Sprachverarbeitungsfließbandes 9
2.5 Ausschnitt einer Ontologie 12

3.1 PARSE-Architektur 19

5.1 Syntaxbaumerzeugung 37
5.2 (a) Aktions- und (b) Bedingungsstruktur als Symbol dargestellt 43
5.3 Sequenzstruktur 44
5.4 Verzweigungsstruktur und Symbol 45
5.5 „While“-Schleifenstruktur und Symbol 45
5.6 „Do-While“-Schleifenstruktur und Symbol 46
5.7 „For“-Schleifenstruktur und Symbol 46
5.8 Parallelblock 47
5.9 Beispiel zum Aufbau eines Pfades des Syntaxbaumes über Musterbereiche . 49
5.10 Beispiel zur Verschmelzung der Pfade von der Wurzel aus 50
5.11 Beispiel für fehlerhafte Musterbereiche über mehrere Ebenen 51
5.12 Beispiel für fehlerhafte Musterbereiche innerhalb einer Ebene. 52
5.13 Nachverarbeitungsfließband zur Quelltexterzeugung 53

6.1 Aufbau der Maven-Projektstruktur 58
6.2 *IASTPattern*-Schnittstelle zur Implementierung der Mustererkenner. 59
6.3 *IASTParentConfidenceCalculator*-Schnittstelle zur Implementierung der Strukturverbindungsalgorithmen. 60
6.4 Grundsätzlicher Aufbau einer Struktur des Syntaxbaumes im PARSE-Graphen 61
6.5 Wurzel-Struktur des PARSE-Syntaxbaumes 61
6.6 (a) Methoden- und (b) Textstruktur im PARSE-Syntaxgraphen 62
6.7 Verzweigungsstruktur im PARSE-Syntaxbaum 62
6.8 While-Schleifenstruktur im PARSE-Syntaxbaum 63
6.9 For-Schleifenstruktur im PARSE-Syntaxbaum 64
6.10 Parallelblock im PARSE-Syntaxbaum 64
6.11 Beispiel zur Berechnung der Konfidenzmatrix mit fehlerhafter Struktur . . . 66
6.12 Die abstrakte Klasse `ASTNode` stellt die Grundlage für alle Knoten dar. . . . 68
6.13 Vererbungshierarchie der *IVisitor*-Schnittstelle zur Implementierung der Besucher für die Übersetzung des Syntaxbaums in Quelltext. 70
6.14 Beispiel für ein mit `PlantUMLCodeVisitor` erzeugtes Aktivitätsdiagramm . 73

7.1 Beispiel eines durch PlantUML erzeugten Aktivitätsdiagramm, welches auf natürlicher Sprache basiert 93
7.2 Beispiel eines durch PlantUML erzeugten Aktivitätsdiagramm mit Methoden 94

B.1 Example of an activity diagram 114
B.2 Activity Diagram with Branch and For-Loop 115
B.3 Activity Diagram with Concurrency 116

Tabellenverzeichnis

2.1 Die Konfusionsmatrix 13

6.1 Aufzählung der verschiedenen Knoten aus denen der Syntaxbaum aufgebaut wird und ihrer Funktion. 69
6.2 Übersetzung des Komparators zu String-Vergleichszeichen. 69

7.1 Inter-Annotator-Übereinstimmung für Ergebnisse der Studie 82
7.2 Ergebnisse der Online-Studie für Fehlerfälle für unterschiedliche Wahlstrategien 83
7.3 Ergebnisse der Online-Studie für Aktivitätsdiagramme für unterschiedliche Wahlstrategien 83
7.4 Kategorisierung sonstiger Fehler in der Online-Studie 84
7.5 Ergebnisse der Online-Studie für Fehlerkategorien nach Verschmelzung für unterschiedliche Wahlstrategien 87
7.6 Ergebnisse der Online-Studie für Aktivitätsdiagramme nach Verschmelzung der Fehlerfälle für unterschiedliche Wahlstrategien. 87
7.7 Fehlerzählung und Gütekriterien bezüglich Methoden, Parameter und vollständiger Aufrufe. 89
7.8 Kontrollstrukturfehler 89
7.9 Quelltextzeilen für implementierte Besucher-Klasse 95
7.10 Differenz zwischen Online-Studie und Methodenevaluation. 96

Quelltextverzeichnis

6.1 CStyleVisitor-Methode für Verzweigungs-Struktur 71
6.2 Pseudo-Quelltext durch PseudoCodeVisitor 71
6.3 Beispiel für PlantUML-Quelltext, welcher durch PlantUMLCodeVisitor erzeugt wird. 72
6.4 Implementierung der Parallelität in JavaCodeVisitor 74
6.5 Erzeugter Java-Quelltext für das Zielsystem ARMAR-III 74
6.6 Implementierung der For-Schleife in PythonCodeGen 74
6.7 Implementierung der Übersetzung der Parallelität in CCodeVisitor 75
6.8 Beispiel für erzeugten parallelen C-Quelltext 75
6.9 Legoroboter-Schablone für Java-Quelltext 76

7.1 Beispiel für erzeugten Java-Quelltext 98
7.2 Beispiel für erzeugten C-Quelltext 99
7.3 Beispiel für erzeugten Java-Quelltext für einen Legoroboter 100
7.4 Beispiel für erzeugten Python-Quelltext 101
7.5 Beispiel für erzeugten PlantUML-Quelltext mit Methoden 102
7.6 Beispiel für erzeugten PlantUML-Quelltext mit Text 103

1 Einleitung

In der heutigen Gesellschaft haben Computer einen festen Platz eingenommen und es ist zu erwarten, dass ihre Bedeutung weiterhin zunimmt. Obwohl dadurch die Nachfrage nach Software-Entwicklern ständig steigt, besitzen nur wenige Menschen tatsächlich die nötigen Kenntnisse, um Software zu entwickeln. Das Entwickeln von Software ist aber ohne diese Kenntnisse nicht möglich. Selbst einfache Programme benötigen Wissen über die Syntax der Programmiersprache und ein Verständnis des logischen Ablaufs eines Programmes. Im Gegensatz dazu ist jeder Mensch in der Lage mit eigenen Worten Anweisungen zu erteilen. Wäre man in der Lage die Lücke zwischen formaler Programmiersprache und natürlicher Sprache zu überbrücken, so könnte jeder Mensch Programme erstellen.

Das Thema dieser Arbeit ist daher die „zielsystemunabhängige Quelltextsynthese aus natürlicher Sprache“. Dadurch könnte beispielsweise die Steuerung von Gerätschaften stark vereinfacht werden. Historisch gesehen wandelte die Kommunikation zwischen Mensch und Maschine sich im Laufe der Zeit. Frühe Rechner wurden noch mit Lochkarten betrieben. Die Entwicklung zu Kommandozeilen und grafischen Benutzeroberflächen vereinfachte die Bedienbarkeit der Geräte. Sie konnten nun auch von Laien bedient werden. Heutzutage gewinnt die Sprachsteuerung immer stärker an Bedeutung. Intelligente Assistenten wie *Siri* und *Amazon Alexa* nehmen eine immer größere Rolle im Alltag vieler Menschen ein. Sie helfen bei einfachen Aufgaben, wie dem Vorlesen neuer Nachrichten oder der Suche nach einem Restaurant. Diese Assistenten werden zwar über sprachliche Kommandos gesteuert, ihre Möglichkeiten sind aber durch fest einprogrammierte Anweisungen begrenzt. Komplexe Befehlsfolgen können nicht erkannt und daher auch nicht ausgeführt werden. In naher Zukunft ist anzunehmen, dass die natürliche Sprache als Schnittstelle zwischen Mensch und Maschine weiter an Bedeutung gewinnt. Von Rechnersystemen wie Robotern wird zukünftig erwartet, dass sie komplexe Befehlsfolgen verstehen. Sie sollten daher in der Lage sein Anweisungen zu interpretieren und selbstständig durchzuführen. Die Rechnersysteme sollen sich an den Menschen anpassen, Anweisungen in natürlicher Sprache verstehen und daraus ausführbaren Quelltext generieren.

In der Forschung findet die Erzeugung von Quelltext aus natürlicher Sprache daher einige Beachtung. Ein Überblick über die verschiedene Ansätze gibt die Arbeit von Pulido-Prieto und Juárez-Martínez [PPJM]. Dabei wird festgestellt, dass die meisten Ansätze die Problemstellung einschränken. Die Einschränkungen finden sich dabei bei der Programmiersprache, der Domäne und auch Sprache. Trotz verschiedener Ansätze existiert noch keine allgemeingültige, domänenunabhängige Lösung zur Übersetzung von natürlicher Sprache in Quelltext.

1.1 Zielsetzung

Ziel dieser Masterarbeit ist die Entwicklung eines Werkzeugs, welches aus natürlicher Sprache zielsystemunabhängig Quelltext synthetisiert. Der erzeugte Quelltext soll lauffähig, korrekt und vollständig sein. Als Eingabe dienen lange und komplexe Befehlsbeschreibungen von Laien in natürlicher Sprache. In der Vorverarbeitung wurden die gesprochenen Sätze durch Sprachverarbeitungswerkzeuge semantisch analysiert. Beispiele für die vorhandenen semantischen Informationen sind beschriebene Aktionen, ihre Ausführungsreihenfolge und den Programmablauf steuernde Kontrollstrukturen. Die Informationen liegen in Form eines Graphen vor. Die Herausforderung liegt nun darin, die gesammelten Informationen des Graphen in Quelltext zu übersetzen. Dabei ist es unter anderem nötig sicherzustellen, dass die Variablenbenutzung konsistent ist und unvollständige Kontrollstrukturen vervollständigt werden. Da weder alle zukünftigen Anwendungen noch Schnittstellen bekannt sind, ist die softwaretechnische Erweiterbarkeit auf neue Eingabeinformationen und andere Zielsysteme von besonderer Bedeutung. Dazu wird als Zwischenrepräsentation der Ablauf des Programms in Form eines abstrakten Syntaxbaum zusammengesetzt. Dieser Syntaxbaum kann anschließend in kompilierbaren, lauffähigen und semantisch korrekten Quelltext übertragen werden. Eine vorgegebene, aber austauschbare, Anwendung soll den Quelltext ausführen. Die Anwendung soll hierbei durch ihre entsprechende Schnittstelle angesprochen werden. Wissen über die Schnittstelle und Einsatzumgebung werden in einer Wissensrepräsentation gespeichert.

1.2 Anwendungsbeispiele

Der Einsatz eines Haushaltsroboters wäre beispielhaft für die Anwendung der Sprachsteuerung. Dieser Roboter soll zum Lüften das Fenster öffnen. Da es aber sein kann, dass die Heizung noch an ist, soll diese gegebenenfalls ausgemacht werden. Der Nutzer gibt dem Roboter die Anweisung „*Robo open the window but before that turn down the heating if it's on*“[1]. Der Roboter muss diese Anweisung nun verstehen und die entsprechenden Handlungen durchführen. Um dies zu erreichen, muss der Roboter semantisch äquivalenten Quelltext erzeugen, welchen er ausführen kann. Da dem Roboter bereits einige Grundbefehle einprogrammiert wurden, kann dies ein mögliches Ergebnis sein:

```
if(robot.getACState().equals(ACState.ON))
    robot.deactivate(ac);
robot.open(window);
```

In einem weiteren Beispiel will der Nutzer dem Roboter „lüften“ als „Grundbefehl“ beibringen. Der Nutzer wiederholt die Anweisung und fügt an, dass der Roboter sich diese Abfolge als „*to ventilate*“ merken soll. Hier muss der Roboter ebenfalls Quellcode erzeugen, diesen aber als neuen Grundbefehl seiner Schnittstelle hinzufügen. Der Nutzer sollte nun in der Lage sein, dem Roboter nur den Befehl zum Lüften zu geben.

1.3 Struktur der Arbeit

Zuerst werden in Kapitel 2, die zum Verständnis dieser Arbeit benötigten Grundlagen vermittelt. Da diese Arbeit auf Wissen aus der Sprachverarbeitung und dem Übersetzerbau basiert, teilt sich das Kapitel in diese beiden Abschnitte auf. Zusätzlich werden auch Schablonen-Systeme und Gütekriterien erläutert.

Auf die Grundlagen folgt in Kapitel 3 eine nähere Betrachtung des PARSE-Projektes. Hierbei wird ein detailierter Überblick über die Architektur, die Entwurfsprinzipien und

[1] Satzzeichen bei gesprochener Sprache nicht enthalten

Zielsysteme gegeben. Am Ende findet eine Einsortierung dieser Arbeit ins PARSE-Projekt statt.

Der Stand der Forschung zur Problemstellung wird in Kapitel 4 vorgestellt. Wie auch die Grundlagen, teilt sich das Kapitel in zwei Bereiche auf: Den Großteil des Kapitels nimmt die Computerlinguistik ein. Die Arbeiten mit ähnlichen Konzepten und Themen, werden in Abschnitte gebündelt. Abseits der Computerlinguistik werden Arbeiten aus dem Bereich der Programmtransformation vorgestellt. Hierbei liegt der Fokus auf der Übersetzung eines abstrakten Syntaxbaumes in Quelltext.

In Kapitel 5 wird die Problemstellung direkt behandelt. Sie wird zu Beginn des Kapitels beschrieben und in ihre Teilprobleme zerlegt. Zu Beginn des Kapitels wird die Problemstellung beschrieben und aufgezeigt, wie diese in Teilprobleme zerlegt werden kann. Im Weiteren wird ein Überblick über den verwendeten Ansatz gegeben. Dabei wird gezeigt wie die Teilprobleme interagieren und genutzt werden können, um die Problemstellung zu lösen. Eine detaillierte Analyse der Teilprobleme folgt in den nächsten Abschnitten. Zeitgleich werden für die Teilprobleme softwaretechnische Verfahren entworfen, die den Lösungsansatz umsetzen.

Die Implementierung des im vorherigem Kapitel entworfenen Lösungsansatz wird in Kapitel 6 beschrieben. Hierbei werden auch die dabei entstehenden Herausforderungen aufgezeigt und ihre Lösung erläutert. Ähnlich der Analyse teilt sich das Kapitel in zwei Teile auf. Im ersten Teil wird Implementierung für die Synthese des abstrakten Syntaxbaum beschrieben. Die Implementierung, um den aufgebauten Syntaxbaum in Quelltext zu übersetzen, wird im zweiten Teil behandelt.

In Kapitel 7 wird die Qualität des Werkzeuges und der Lösung evaluiert. Die Erzeugung des Syntaxbaumes wird durch eine *Online*-Studie und manuelle Evaluation bewertet. Anschließend wird die Verwendung von abstrakten Syntaxbäumen als Zwischenrepräsentation und der Einfluss des Zielsystems betrachtet.

Zum Schluss werden in Kapitel 8 die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf weiterführende Forschungen gegeben.

2 Grundlagen

Ein starkes Argument für die Vermarktung von intelligenten Assistenten wie *Siri* und *Alexa* ist ihre einfache Benutzbarkeit. Dabei dient idealerweise natürliche Sprache als Mensch-Maschine-Schnittstelle. Computer können aber nur konkrete Befehlsabfolgen in einer Maschinensprache verarbeiten. Daher ist es notwendig eine Verbindung zwischen der gesprochenen informalen Sprache des Anwenders und der verarbeitenden formalen Sprache des Computers aufzubauen. Das Thema „Quelltextsynthese aus natürlicher Sprache" ist deshalb in der Forschung von hoher Bedeutung. Während es in der Computerlinguistik meist als Problem des Sprachverständnisses betrachtet wird, kann es aber auch als Problem des Übersetzerbaus gesehen werden.

Zum Verständnis dieser Arbeit wird daher in diesem Kapitel ein Überblick über beide Bereiche gegeben. Die Grundlagen für die Quellcodeübersetzung werden in Abschnitt 2.1 behandelt. Abschnitt 2.2 erläutert Grundbegriffe und Konzepte aus der Computerlinguistik, welche zum Verständnis des in Kapitel 3 erklärten Projektes benötigt werden. Darauffolgend werden in Abschnitt 2.3 Schablonen-Systeme erläutert. Den Schluss bildet Abschnitt 2.4 mit einer Beschreibung häufig verwendeter Gütekriterien.

2.1 Übersetzerbau

Das Thema „Programmierung in natürlicher Sprache" kann als Problem des Übersetzerbaus betrachtet werden. Die Übersetzung natürlicher Sprache in Quelltext könnte daher als Kompilierung verstanden werden. Die natürliche Sprache nimmt bei dieser Sicht die Rolle der „höheren Programmiersprache" ein. Als Zielsprache kann jede Programmiersprache dienen.

In diesem Abschnitt sollen daher einige Grundlagen der Übersetzerbaus wiederholt werden. Das Buch „Compiler: Prinzipien, Techniken und Werkzeuge" von Ullman et al. [ULSA] dient als Quelle für diesen Abschnitt.

2.1.1 Grundbegriffe

Als **Graph** wird in der Informatik eine Datenstruktur bezeichnet, die aus Knoten und Kanten besteht. Sie repräsentiert eine Menge von Objekten und die Beziehungen zwischen diesen Objekten. Die Knoten des Graphen stellen hierbei die Objekte mit ihren Eigenschaften dar. Die Beziehungen werden durch die Kanten repräsentiert und können gerichtet und gewichtet sein. Gerichtete Kanten beschreiben eine Beziehung zwischen zwei

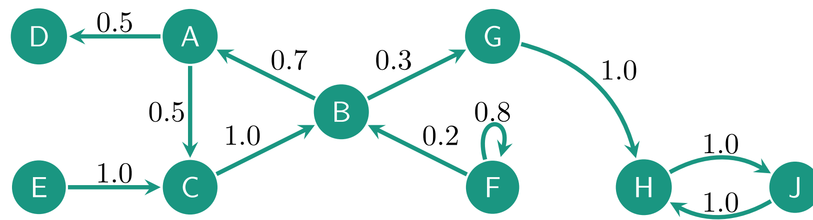


Abbildung 2.1: Ein gerichteter Graph mit Übergangswahrscheinlichkeiten als Kantengewichte.

Objekten, die nur in eine Richtung existiert. Sollen Kanten sich unterscheiden, so können sie Eigenschaften besitzen. Beispiele hierfür wären Pfadnutzungskosten oder Beziehungsbezeichnungen. Man bezeichnet sie dann als gewichtet. Besitzt ein Graph einen Zyklus, so existiert ein Pfad von einem Knoten, der zum anfänglichen Knoten zurückführt. Ein Graph ist zusammenhängend, falls er ohne die Entfernung von Kanten nicht teilbar ist. In Abbildung 2.1 ist ein zusammenhängender Graph beispielhaft dargestellt. Dieser Graph beschreibt ein statistisches Modell möglicher Zustandsänderungen eines Systems. Die Zustände werden durch Knoten repräsentiert. Der Graph ist gerichtet und die Kanten besitzen Übergangswahrscheinlichkeiten als Gewichte. Die Übergangwahrscheinlichkeiten beschreiben die Wahrscheinlichkeit einer spezifischen Zustandsänderung. Damit ist es möglich, den neuen Zustand zu schätzen oder die Wahrscheinlichkeit für eine Sequenz von Zustandsänderungen zu berechnen. Daneben besitzt der Graph in $\{H \rightarrow J \rightarrow H\}$, $\{F \rightarrow F\}$ und $\{C \rightarrow B \rightarrow A \rightarrow C\}$ Zyklen. Der Zyklus in $\{F \rightarrow F\}$ ist eine reflexive Kante. Graphen bilden in der Informatik eine Grundlage für viele weiterführende Strukturen wie Automaten, Markow-Modelle, Netzwerke und Bäume.

Bäume sind zusammenhängende, zyklenfreie Graphen. Sie bestehen aus einem Wurzelknoten und mehreren Blatt- und inneren Knoten. Die Knoten sind durch Kanten hierarchisch in mehrere Ebenen unterteilt. Auf oberster Ebene befindet sich ein einzelner Wurzelknoten. Von der Wurzel sind alle anderen Knoten erreichbar. Alle Knoten einer Ebene besitzen den gleichen Abstand zum Wurzelknoten. Die Elternknoten sind mit ihren Kindknoten durch Kanten verbunden. Kindknoten befinden sich auf der darunterliegenden Ebene ihres Elters. Blattknoten sind Knoten ohne Kindknoten und in ihrer Hierarchie auf der untersten Ebene. Als **Wald** bezeichnet man einen Graphen, der aus mehreren, voneinander getrennten Bäumen besteht.

Eine **formale Grammatik** besteht aus Symbolen und Ableitungsregeln. Sie definiert dadurch eine **formale Sprache**. Formale Sprachen haben eine eindeutige Syntax und Semantik, die eine Ausführung auf einem Rechner ermöglicht. Beispiele für formale Sprachen sind Java, Python und XML. Die Symbole teilen sich in *Nichtterminal-*, *Terminal-*, und *Startsymbol* ein. Nichtterminalsymbole werden über definierte Regeln auf eine Sequenz von Symbolen abgeleitet. Die ursprünglichen Symbole werden als *Kopf* oder *linke Seite* bezeichnet. Andere Symbole können im Kopf als Voraussetzung die Ableitung durch Regeln beschränken. Dadurch nimmt der Kontext, indem ein Nichtterminalsymbol steht, Einfluss auf die anwendbaren Ableitungsregeln. Die Symbole, durch die der Kopf ersetzt wird, nennt man *Rumpf* oder *rechte Seite*. Terminalsymbole können nicht weiter abgeleitet werden. Sie sind daher mit den Token identisch. Das Startsymbol ist ein besonderes Nichtterminalsymbol. Es ist das ursprüngliche Symbol aus welchem die Sprache abgeleitet wird. Die **kontextfreie Grammatik** ist eine Sonderform der formalen Grammatik. Hierbei besteht der Kopf aus genau einem einzigen Nichtterminalsymbol. Dadurch gibt es zur Anwendung einer Regel keine weiteren Voraussetzungen. Das heißt alle Ableitungsregeln sind unabhängig vom Kontext des Nichtterminalsymbols. Der Rumpf darf weiterhin frei aus mehreren Nichtterminal- und Terminalsymbolen bestehen.

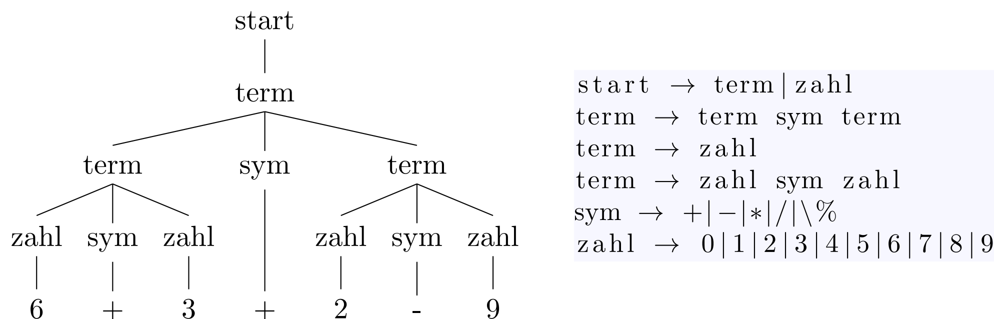


Abbildung 2.2: Syntaxbaum mit zugehöriger kontextfreier Grammatik

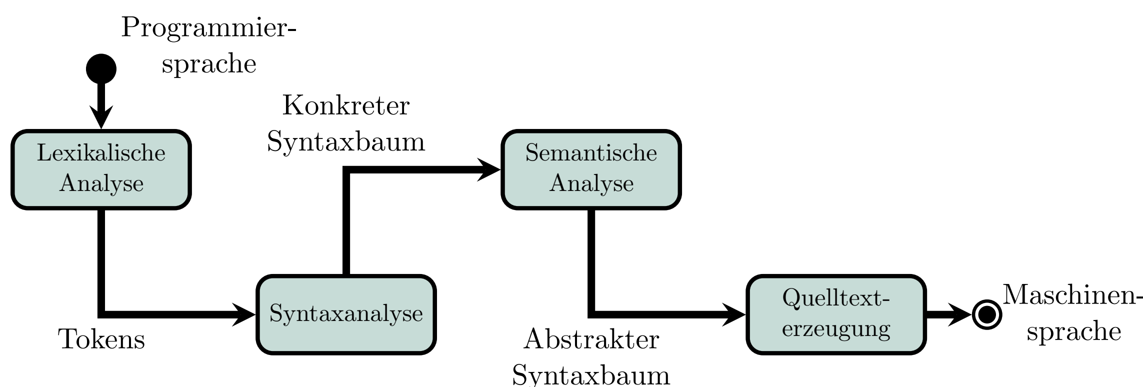


Abbildung 2.3: Fließbandarchitektur eines Kompilierers: Die höhere Programmiersprache wird in mehreren Stufen in Maschinensprache übersetzt.

Syntaxbäume sind eine spezielle Form von Bäumen. Sie dienen dazu, die grammatikalische Struktur von formalen Worten darzustellen. In den Blättern stehen die Terminalsymbole, d.h. die Token der Quelltextanweisung. Die Wurzel ist das Startsymbol der Grammatik. Die Nichtterminalsymbole werden durch innere Knoten repräsentiert. Ableitungen anhand Regeln einer Grammatik entsprechen den Kanten zwischen Knoten. Der Kopf entspricht hierbei dem Elternknoten. Alle Kindknoten, von links nach rechts gelesen, symbolisieren den Rumpf der Ableitungsregel. Bei Syntaxbäumen wird zwischen **abstrakten** und **konkreten Syntaxbäumen** unterschieden. Konkrete Syntaxbäume stellen die direkte Ableitungsstruktur der Tokens aus einem Startsymbol dar. Abstrakte Syntaxbäume haben hingegen schon semantische Informationen eingebaut. So kann beispielsweise eine nötige implizite Transformation des Variablentyps oder die Abbildung mehrerer Token auf bestimmte Befehle schon vorgenommen worden sein. In Abbildung 2.2 ist ein Syntaxbaum, inklusive der kontextfreien Grammatik, dargestellt.

2.1.2 Aufbau eines Kompilierer

Der Aufbau eines Kompilierers besteht in der Regel aus einer Fließbandarchitektur, welche in Abbildung 2.3 dargestellt ist. Eine höhere Programmiersprache wird bei der **lexikalischen Analyse** portioniert und in eine Sequenz von Tokens umgewandelt. Variablen werden durch spezielle Token ersetzt, die auf den Variablennamen in einer **Symboltabelle** verweisen. In der Symboltabelle werden Bezeichner für Variablen, Namensräume und weitere Informationen gesammelt. Darüber wird die Konsistenz dieser in allen Stufen des Fließbandes sichergestellt. Aus den Tokens wird bei der **Syntaxanalyse** ein **konkreter Syntaxbaum** aufgebaut. Hierbei wird mit Hilfe der kontextfreien Grammatik die Syntax der Tokens abgeleitet. Der konkrete Syntaxbaum ist eine direkte Repräsentation der

Tokens. In der **semantischen Analyse** wird darauf folgend, der konkrete Syntaxbaum in einen **abstrakten Syntaxbaum** umgewandelt. Beim abstrakten Syntaxbaum werden die Tokens in Befehle abstrahiert. Hierbei werden die Bezeichner der Symboltabelle mit Typinformationen versehen. Zeitgleich findet auch eine Typüberprüfung statt. Anhand des abstrakten Syntaxbaums kann nun der maschinenlesbare Quelltext generiert werden. Dies geschieht meist iterativ über mehrere Zwischenrepräsentation. Je nach Kompilierer können hierbei auch Quelltextoptimierungen vorgenommen werden.

2.2 Computerlinguistik

Für eine Vielzahl an Anwendungen aus verschiedenen Bereichen gewinnt die Verarbeitung von natürlicher Sprache immer mehr an Bedeutung. Unter **natürlicher Sprache** versteht man Sprache wie sie im Alltag geschrieben und gesprochen wird. Beispiele für natürliche Sprachen wären deutsch, englisch und französisch. Bei natürlichen Sprachen erlaubt die Grammatik und Wortwahl Mehrdeutigkeiten, die von Menschen anhand des Kontext implizit aufgelöst werden. Bei der Übersetzung natürlicher Sprachen in formale Sprachen stellen viele Eigenschaften natürlicher Sprachen, wie beispielsweise implizites Wissen und mehrdeutige Wörter, Herausforderungen dar. Auch unterscheidet sich die verwendete Sprache je nach Sprecher und Anwendungsbereich. So ist zum Beispiel der Wortschatz eines Kindes stark beschränkt. Dahingegen formuliert ein erfahrener Informatiker Anweisungen an einen Roboter, die unterbewusst formaler Sprache ähneln. Weiterhin ist es möglich, dass Sprecher Begriffe verwenden, die in ihrem Fachbereich gebräuchlich sind. Ein Werkzeug, welches in dieser **Domäne** eingesetzt wird, muss Kontextwissen über den Bereich besitzen. Das Wissen über die Domäne kann zum Beispiel durch eine Ontologie (siehe Abschnitt 2.2.4) repräsentiert werden. Bei der Sprachverarbeitung erleichtert dieser Kontext die Auflösung von mehrdeutigen Wörtern.

In diesem Unterkapitel werden zuerst in Abschnitt 2.2.1 einige Grundbegriffe und Konzepte aus dem Bereich der Verarbeitung von natürlicher Sprache vorgestellt. In Abschnitt 2.2.2 wird der Grundaufbau der Programme als Fließband erklärt. Da viele Werkzeuge der Computerlinguistik auf Klassifikatoren basieren, wird in Abschnitt 2.2.3 das Thema maschinelles Lernen erläutert. Eine bewährte Wissensrepräsentation in der Sprachverarbeitung wird in Abschnitt 2.2.4 beschrieben. Als Grundlage für dieses Unterkapitel dient das Buch von Jurafsky [Jur].

2.2.1 Syntax, Semantik und Pragmatik

Das Ziel der Computerlinguistik ist die Gewinnung von Informationen über Syntax, Semantik und Pragmatik von Texten in natürlicher Sprache.

Als **Syntax** bezeichnet man den korrekten Satzbau einer Sprache. Der Aufbau von Sätzen in natürlicher und formale Sprache ist durch Regeln formuliert. Diese Regeln sind über die Grammatik einer Sprache definiert. Die Reihenfolge der grammatikalischen Ableitungen zur Erzeugung eines korrekten Satzes bilden eine Baumstruktur. Dieser Baum stellt die Syntax einer Sprache dar und wird deshalb als **Syntaxbaum** bezeichnet. Die Syntax der natürlichen Sprache ähnelt der formalen Syntax. Während die Grammatiken formaler Sprache sich vollständig beschreiben lassen, sind Grammatiken natürlicher Sprachen dafür zu komplex. Ein weitere Herausforderung sind mehrdeutige Wörter. Dies führt dazu, dass es je nach Bedeutung der Wörter mehrere mögliche Syntaxbäume geben kann. Das Erkennen der korrekten Syntax eines Satzes ist bei natürlicher Sprache daher selten sicher möglich. Ein **Zerteiler** (engl. parser) konstruiert aus Sätzen und Satzteilen über eine Grammatik einen Syntaxbaum.

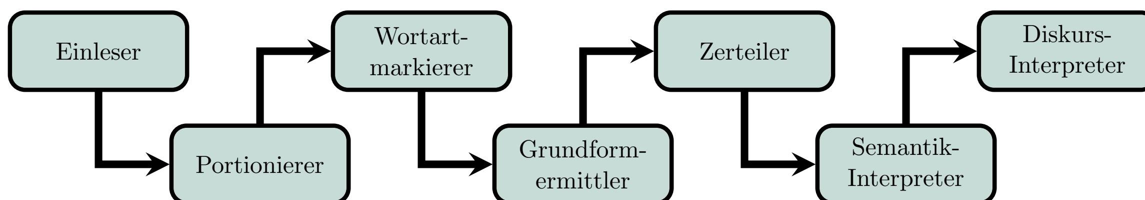


Abbildung 2.4: Das Standard-Modell des Sprachverarbeitungsfließbandes

Unter **Semantik** versteht man die wörtliche Bedeutung der Worte und Sätze. Das gleiche Wort kann je nach Syntax und Kontext eine unterschiedliche Bedeutung haben. Das korrekte Interpretieren der Semantik ist zur Ausführung einer Anweisung unabkömmlich. So müssen für die vollständige Beschreibung einer Aktion auch **semantische Rollen** erkannt werden. Semantische Rollen sind die Akteure und Objekte einer Aktion.

Von der Semantik wird die **Pragmatik** unterschieden. Darunter versteht man eine kontextabhängige Bedeutung der Worte und Sätze. Beispiele hierfür sind ironische und sarkastische Aussagen und solche, die von „Geheimwissen" abhängen. So ist zum Beispiel beim Satz *„the eagle has landed"* die Landung eines Adlers als semantische Bedeutung direkt ersichtlich. Unter Kenntnis der Redewendung kann hingegen noch die pragmatische Bedeutung der Missionserfüllung erkannt werden.

2.2.2 Computerlinguistik-Fließbänder

Unabhängig von Domäne und konkretem Anwendungszweck basiert die Software-Architektur einer Sprachverarbeitungsanwendung in der Regel auf einer Fließbandarchitektur. Die einzelnen Stufen des Fließbandes bauen dabei meist aufeinander auf. So dienen die Ergebnisse von früheren Stufen als Eingabe für darauf folgende Stufen. Mit jeder Stufe wird daher ein tieferes Verständnis des ursprünglichen Textes herausgearbeitet. Zur Verdeutlichung wird in diesem Abschnitt die in Abbildung 2.4 dargestellte, sprachverarbeitende Fließbandarchitektur mit ihren Stufen beschrieben. Als Grundlage für diesen Abschnitt dient neben dem Buch von Jurafsky die Beschreibung der Fließbandarchitektur im **N**atural **L**anguage **T**ool**k**it [NLTK] [LB].

Als erstes müssen die zu verarbeitenden Texte eingelesen werden. Liegen die Texte nicht in schriftlicher Form vor, müssen **Spracherkenner** eingesetzt werden. Spracherkenner wandeln gesprochene Sprache in geschriebenen Text um. Dabei werden mehrere Laute erkannt und auf das wahrscheinlichste Wort abgebildet. Dies kann durch *verdeckte Markowmodelle* (engl. Hidden Markov Model) oder neuronale Netze stattfinden. Satzzeichen werden dabei üblicherweise nicht erkannt.

Nach der Einlesung liegen die Text als Fließtext vor. Ein solcher Text wird durch **Portionierer** in einzelne **Token** portioniert. Diese bilden die kleinste Sinneinheit, die von anderen Werkzeugen weiterverarbeitet wird. In der Sprachverarbeitung entsprechen Tokens meist einzelnen Worten. Eine Herausforderung bilden hierbei Sinneinheiten wie *„New York"* oder *„Dr. Acula"*, die aus mehreren Wörtern bestehen. Sie sollten durch ein einzelnes Token repräsentiert werden.

Als Grundlage für die Ermittelung der Syntax wird den Wörtern ihre Wortart, d.h. grammatikalische Klassen wie Nomen, Verben oder Adjektive zugewiesen. Dies bezeichnet man als **Wortartmarkierung** (engl. Part-of-Speech tagging). Die Klassifizierung wird anhand eines Modells durchgeführt, welches durch maschinelle Lernverfahren trainiert wurde. Als Trainingskorpus wird meist der „*Wall Street Journal*"-Korpus [MMS] verwendet. Wortartmarkierer beinhalten oft eigene Portionierer.

Im nächsten Schritt, reduziert ein **Grundformermittler** Wörter in ihre Grundform. Bei Verben wird die Zeitform entfernt und nur der Wortstamm belassen. Die Pluralform bei Nomen wird hingegen auf die Singularform abgebildet. Man unterscheidet zwischen zwei Arten von Grundformermittlern. Bei der **Lemmatisierung** (engl. lemmatisation) wird das Wort über Regeln tatsächlich auf die grammatikalisch korrekte Grundform reduziert. So wäre das Lemma von *„features"* der korrekte Singular *„feature"*. Wird hingegen das Wort nur zurechtgeschnitten, so bezeichnet man dies als **Stammformreduktion** (engl. stemming). Hierbei würde der Plural *„features"* beispielsweise in die Form *„featur"* reduziert werden. Da unterschiedliche Wörter auf die gleiche Stammform geschnitten werden, können Informationen verloren gehen. Ein Beispiel hierfür wären die Wörter *„operate"*,*„operation"* und *„opera"* welche zu *„oper"* reduziert werden.

Anhand der markierten Wortarten baut ein Zerteiler einen Syntaxbaum auf. Nach der syntaktischen Analyse folgt die **semantische Analyse**. Hierbei wird die Bedeutung der einzelnen Wörter interpretiert. Einfluss auf die Bedeutung haben die lexikalischen Definitionen des Wortes, die syntaktischen Strukturen und der umgebende Kontext, d.h. die Bedeutung anderer Worte und Sätze. Ein einfacher Ansatz für die semantische Analyse ist das Zuweisen der Bedeutung nach Häufigkeit. Hierbei wird abhängig von der Wortart eines Wortes, die am häufigsten verwendete Bedeutung des Wortes zugewiesen. Als Teil der semantischen Analyse werden auch beispielsweise semantische Rollen zugewiesen und Synonyme erkannt.

Auf der semantischen Bedeutung der einzelnen Worte aufbauend, folgt die Interpretation des ganzen Textes. Hier werden Zusammenhänge zwischen den Worten und Sätzen gefunden und die Bedeutung von mehreren Sätzen festgestellt. Eine Gruppe von Sätzen, die einen gemeinsamen kohärenten Kontext bilden, bezeichnet man als **Diskurs**. Dieses vollständige Sprachverständnis ist äußerst komplex und im Allgemeinen nur schwer lösbar. Es existieren aber Lösungsansätze für Teilprobleme des Sprachverständnisses. Die **Koreferenzauflösung** wäre ein Beispiel für ein Verfahren welches der Diskurs-Interpretation zugeschrieben wird. Oft werden in natürlicher Sprache die selben Entitäten mit unterschiedlichen Begriffen angesprochen. Anstatt wiederholt die volle Bezeichnung der Entität zu verwenden, kann sie durch Worte wie *„she/this/that"* referenziert werden. Auch ist es möglich sich auf Entitäten durch ihre übergeordnete Konzepte zu beziehen. Bei der Koreferenzauflösung werden diese Referenzen ihren Entitäten zugewiesen.

Eine Herausforderung, die in jeder Stufe des Fließbandes auftreten kann, ist die **Mehrdeutigkeit** von Semantik, Pragmatik und Syntax. Da sprachverarbeitende Werkzeuge in der Regel auf statistischen Modellen basieren, führen sie in fast jedem Werkzeug zu Ungenauigkeiten. Als Beispiel einer semantischen Mehrdeutigkeit dient das Wort *„Bank"*. Es kann als *„Sitzbank"* oder *„Geldinstitut"* verstanden werden. Es ist daher nötig den Kontext des Wortes über bekannte Definitionen aufzulösen. Dazu wird die Definition des unbekannten Wortes mit den Definitionen der Wörter seiner Umgebung abgeglichen. Dem Wort wird die Bedeutung zugewiesen, die am meisten Überlappung mit seinem Kontext hat [Les]. Ein Korpus hierfür kann mit Hilfe von Wikipedia erstellt werden [Mih]. Auch die Syntax eines Satz kann von Mehrdeutigkeiten betroffen sein. Ein Beispiel hierfür wäre der Satz *„the girl saw a man with a telescope"*. Hier besteht eine syntaktische Mehrdeutigkeit, die sich auch auf die Semantik auswirkt. So kann zum Einen *„man with a telescope"* eine syntaktische Einheit bilden. Dann bedeutet der Satz, dass ein Mädchen einen Mann sieht, der ein Teleskop besitzt. Andererseits besteht auch die Möglichkeit, dass das Mädchen einen Mann durch ein Teleskop sieht. Dies würde einen anderen Syntaxbaum implizieren. Als dritte Möglichkeit wäre noch die Interpretation möglich, bei der das Mädchen einem Mann mit einem Teleskop „sägt". Hierbei wäre der Syntaxbaum identisch mit dem zweiten Fall, aber die Semantik wäre anders. Um diese Mehrdeutigkeit aufzulösen, müsste der Satz vollständig durch Diskurs-Interpreter verstanden werden. Dies wäre aber ohne Informationen zum

weiteren Kontext nicht möglich.

2.2.3 Maschinelles Lernen

Die meisten Werkzeuge der Sprachverarbeitung basieren auf Methoden des maschinellen Lernens. Das Wissen über Grundlagen dieses Bereichs vereinfacht deshalb das Verständnis verwandter Arbeiten. Daher soll hier eine kurze Einführung gegeben werden. Für diesen Abschnitt dient das Buch „Data Mining“ von Witten et al. [WFHP] als Grundlage.

Verfahren aus dem Bereich des maschinellen Lernens sind Verfahren zur **Klassifikation**. Hierbei soll eine **Instanz** mit bestimmten Merkmalen, anhand dieser Merkmale einer Klasse zugewiesen werden. Die Klassifikation teilt sich daher in zwei Strukturen auf. Die erste Struktur ist der eigentliche **Klassifikationsalgorithmus**. Der Algorithmus kann auf unterschiedlichen Ansätzen beruhen und ist universell einsetzbar. So gibt es beispielsweise Verfahren, die auf mathematische Funktionen, Regeln oder neuronalen Netzen basieren. Um den Algorithmus auf einer bestimmten Domäne einzusetzen, muss vorher ein **Modell** trainiert werden. Hierbei werden anhand eines Trainingskorpus entsprechende Kriterien gelernt. Durch diese Kriterien lassen sich Instanzen klassifizieren. Daneben existieren Verfahren, bei denen das Modell während der Klassifikation modifizierbar ist. Sie können neue Instanzen dazu nutzen, um ihr Modell zu verfeinern.

Grundsätzlich gibt es zwei Arten des maschinellen Lernens: Beim **überwachten Lernen** werden neben den Eingabedaten auch Musterlösungen bereitgestellt. Hierbei wird das Modell auf die Einteilung der Instanzen in die vorgegebenen Klassen trainiert. Die zweite Art ist das **unüberwachte Lernen** . Hierbei wird auf Trainingsdaten ohne Lösungen trainiert. Das trainierte Modell leitet anhand dem Datensatz selbstständig Klassen (engl. Cluster) ab. Durch diese Verfahren können Strukturen des Datensatzes sichtbar gemacht werden.

Beispiel 2.2.1: Einfache Klassifikation durch Regeln

Für einen Wetterdatensatz sollen abhängig von `Temperatur` und `Niederschlagsmenge`, die Tage bestimmt werden, an denen es geschneit hat.
Beim überwachten Verfahren werden für die Trainingsdaten Musterlösungen bereit gestellt. Das trainierte Modell besteht dadurch aus einer einfach Regel:
`Temperatur(Niedrig)` & `Niederschlagsmenge(Hoch)` → `Schneewetter`
Bei Anwendung eines unüberwachten Verfahren hingegen werden zwei Klassen entdeckt: `feucht-kalte` und `trocken-heiße` Tage. Dabei ist nicht garantiert, dass mit der `feucht-kalte` Klasse tatsächlich Schneewetter gemeint ist.

Als besonderen Teil der maschinellen Lernverfahren wären die **neuronalen Netze** zu nennen. Die Grundidee der neuronalen Netze ist die Simulation der Funktionsweise eines Gehirns durch Rechnersysteme. Dies wird durch ein Netzwerk aus mehreren Schichten von *Perzeptrons*, die als Neuronen bezeichnet werden, erreicht. Neuronen sind *lineare Klassifikatoren*, die aus einer Menge von gewichteten Eingabewerte einen Ausgabewert erzeugen. Sie stellen damit eine einzelne mathematische Funktion dar. In einem neuronalen Netz sind diese Neuronen in mehrere Schichten eingeteilt. Jede folgende Schicht nutzt die Ausgabewerte der vorherigen Schicht als Eingabe. Dadurch können komplexe mathematischen Funktionen definiert werden. Die Struktur und Gewichte dieser Netze werden durch maschinelle Lernverfahren gefunden. Durch leistungsstärkere Computerhardware erlebten neuronale Netze seit 2009 durch das *Tiefenlernen* [HDY+] eine Renaissance. Oft verwendete Ansätze in der Computerlinguistik sind **rekurrente neuronale Netze** [FGS] und die **Enkodierer-Dekodierer-Architektur** (engl. encoder-decoder architecture) [WSC+].

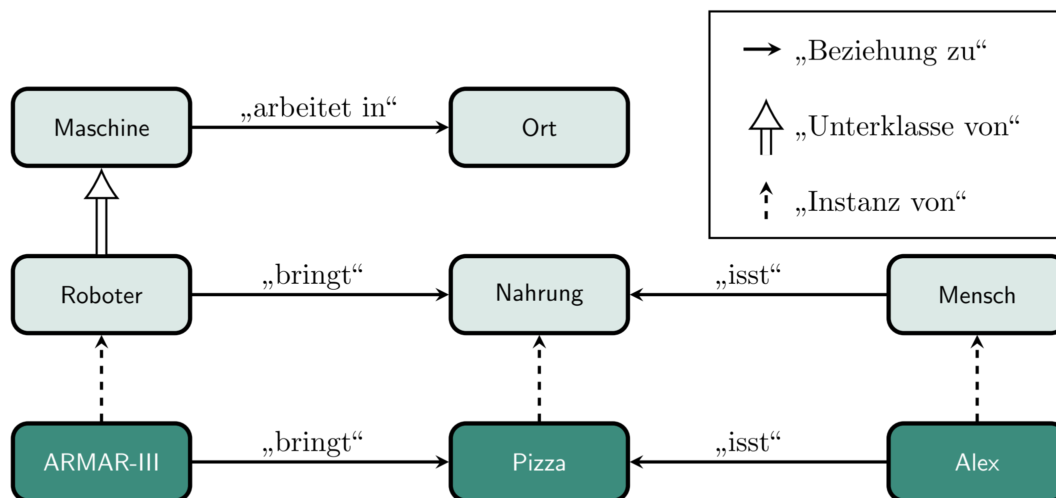


Abbildung 2.5: Ausschnitt einer Ontologie in der die Konzepte Mensch und Roboter zum Konzept Nahrung in Beziehung gesetzt werden.

Rekurrente neuronale Netze zeichnen sich dadurch aus, dass Neuronen späterer Schichten auch Auswirkung auf Neuronen früherer Schichten haben. Die Enkodierer-Dekodierer-Architektur ist eine zweistufige Fließbandarchitektur für neuronale Netze. In der ersten Stufe wird die Eingabe durch ein neuronales Netze in eine Zwischenrepräsentation „verschlüsselt“. Diese Zwischenrepräsentation wird in der zweiten Stufe durch ein weiteres neuronales Netz in die gewünschte Ausgabe „entschlüsselt“.

2.2.4 Ontologie

Nach Gruber [Gru] dienen Ontologien dazu Computersystemen implizites Wissen über Zusammenhänge (engl. relations) zwischen Konzepten (engl. concepts) bzw. Dingen explizit zur Verfügung zu stellen. Sie sind damit formale Wissensrepräsentationen. Ein Roboter kann ohne die explizite Information, dass die Kühlschranktür geschlossen werden muss, nicht darauf schließen, dass ein offener Kühlschrank Energie verschwendet. Spricht ein Mensch davon, dass etwas aus dem Kühlschrank geholt werden soll, so wird *implizit* erwartet, dass der Kühlschrank zuerst geöffnet und anschließend auch wieder geschlossen wird. Dieses implizite Wissen wird von einem Menschen also nicht explizit erwähnt, muss aber trotzdem bekannt sein. Unter implizites Wissen können auch Informationen fallen, die sich nur schwer oder gar nicht formulieren lassen. Ein Beispiel hierfür wäre der aufrechte Gang ohne das Gleichgewicht zu verlieren. Dieses Wissen lässt sich daher auch nicht in einer Ontologie darstellen.

In Abbildung 2.5 ist ein Ausschnitt aus einer Ontologie dargestellt. Hier wird explizit die semantische Relation zwischen den Konzepten „Roboter“,„Mensch“ und „Nahrung“ spezifiziert. Der Roboter ist hierbei als „Maschine“ zu verstehen. Daher besitzt ein Roboter ebenfalls die Beziehung „arbeiten in“ zum Konzept „Ort“. Als Roboter ist es seine Aufgabe Nahrung „zu bringen“. Der Mensch hingegen „isst“ die Nahrung. In der Ontologie werden auch die konkreten Instanzen „Armar-III“,„Alex“ und „Pizza“ ihrem Konzept zugeordnet.

2.3 Schablonen-Systeme

Oft ist es nötig bei bestimmten Dateien oder Texten nur einzelne Teile zu ändern, während der Rest identisch bleibt. Dies kann leicht mit Schablonen-Systemen durchgeführt werden. Die Architektur eines Schablonen-System (engl. template engine) besteht aus drei Teilen.

Tabelle 2.1: Die Konfusionsmatrix

	Instanz positiv	Instanz negativ
Klassifikation positiv	richtig positiv	falsch positiv
Klassifikation negativ	falsch negativ	richtig negativ

Den ersten Teil bildet die namensgebenden Schablone. Sie besteht aus einer Datei mit reservierten Platzhaltern, welche durch verschiedene Datenwerte ausgefüllt werden können. Die möglichen Werte bilden den zweiten Teil des Schablonen-Systems. Hierbei ist es möglich, dass die Werte einen bestimmten Datentyp besitzen müssen oder von Bedingungen eingeschränkt werden. Wird das Schablonen-System als Teil einer Anwendung verwendet, so können auch Attribute von Instanzen als Werte verwendet werden. Der zentrale Teil ist das System selbst, welches dafür zuständig ist aus der Schablone und den Werten die gewünschte Ausgabe zu erstellen. Dazu muss das System zuerst die Schablone einlesen und interpretieren. Danach können die gewünschten Werte beispielsweise von Instanzen bezogen und auf ihre Korrektheit überprüft werden. Zum Zusammenführen von Schablone und Datenwerten, werden die Werte in die Platzhalter der Schablone injiziert. Das so erzeugte Ergebnis kann nun ausgegeben werden. Schablonen-Systeme werden im Bereich der Modellgetriebenen Softwareentwicklung zur Erzeugung von Quelltext aus Modellen wie beispielsweise Klassendiagrammen eingesetzt.

2.4 Gütekriterien

Eine Evaluation anhand Metriken erlaubt einen Vergleich und die qualitative Einordnung unterschiedlicher Ansätze. Hierfür existieren eine Vielzahl an Metriken, die es erlauben verschiedene Aspekte der Ansätze zu beurteilen. Die für diese Arbeit relevanten Gütekriterien werden in diesem Kapitel kurz vorgestellt.

Als erstes sollen die verschiedenen möglichen Ergebnis- und Fehlerfälle bei binären Entscheidungen beschrieben werden. Bei binären Entscheidungen muss eine Instanz einer von zwei Klassen zugewiesen werden. In Tabelle 2.1 ist die Konfusionsmatrix zu sehen. In den Spalten stehen die korrekten Eigenschaften der getesteten Instanzen. Die Zeilen hingegen stellen das Ergebnis der Klassifikation dar. Wurde eine positive Instanz positiv klassifiziert, so ist der Fall **richtig positiv** (engl. true positive). Bei einer negativen Instanz, die negativ klassifiziert wurde, ist der Fall **richtig negativ** (engl. true negative). Wird eine positive Instanz fehlerhaft als negative Instanz klassifiziert, so bezeichnet man das als **falsch negativ** (engl. false negative). Als **falsch positiv** (engl. false positive) bezeichnet man den entgegengesetzten Fall, bei dem eine negative Instanz positiv klassifiziert wird. Fälle, die mehr als zwei Klassen haben, lassen sich auf eine binäre Entscheidung reduzieren. Dazu wird die Klassifikation auf Gleichheit überprüft. Fälle bei denen sich Instanz und Klassifikation unterscheiden, werden als „Falsch“ klassifiziert. Die Fälle mit gleicher Instanz und Klassifikation als „Richtig“. Im Idealfall ist der gesamte Datensatz ebenfalls „Richtig“

Die erste Metrik ist die **Präzision** (engl. precision). Sie bewertet wie präzise die Klassifizierung ist. Hierbei ist die Anzahl an gefundener Instanzen nebensächlich. Das heißt es wird der Anteil an richtig positiver Instanzen im Vergleich zu allen positiven Instanzen bestimmt.

$$\text{Präzision} = \frac{\text{richtig positiv}}{\text{richtig positiv} + \text{falsch positiv}} \tag{2.1}$$

Die gegenteilige Metrik ist die **Ausbeute** (engl. recall). Sie bewertet den Anteil gefundener

richtig positiver Instanzen gegenüber der Gesamtmenge an gesuchten Instanzen.

$$\text{Ausbeute} = \frac{\text{richtig positiv}}{\text{richtig positiv} + \text{falsch negativ}} \tag{2.2}$$

Weder Präzision noch Ausbeute erfassen alle möglichen Ergebnistypen der Konfusionsmatrix. Dies führt dazu, dass jeweils Teile des Ergebnisses nicht beachtet werden. Die **Genauigkeit** (engl. accuracy) ist eine Metrik die alle Ergebnistypen verwendet. Dadurch ist es möglich Klassifkatoren beider Zielanforderungen in Bezug zu setzen. Die Genauigkeit beschreibt den Anteil richtig klassifizierter Instanzen an der Gesamtmenge.

$$\text{Genauigkeit} = \frac{\text{richtig positiv} + \text{richtig negativ}}{\text{richtig positiv} + \text{richtig negativ} + \text{falsch positiv} + \text{falsch negativ}} \tag{2.3}$$

Datensätze, die stark unausgeglichen sind, d.h. fast alle Instanzen sind vom gleichen Typ, reduzieren die Aussagekraft der Genauigkeit. Werden hier alle Instanzen mit der häufigsten Klasse klassifiziert, so ist die Genauigkeit hoch, obwohl der Klassifikator im Allgemeinen eine niedere Qualität aufweist.

Mit dem F-Maß (engl. F-score) gibt es eine Alternative zur Genauigkeit, die Ausbeute und Präzision kombiniert. Im Allgemeinem Fall kann durch die Wahl der Konstante β die Gewichtung der Ausbeute gegenüber der Präzision bestimmt werden. Zum Beispiel bewertet ein hoher β-Wert Klassifikatoren besser, die eine hohe Ausbeute fokussieren.

$$\mathrm{F}_\beta = (1 + \beta^2) \cdot \frac{\text{Präzision} \cdot \text{Ausbeute}}{(\beta^2 \cdot \text{Präzision}) + \text{Ausbeute}} \tag{2.4}$$

Am häufigsten wird das F_1-Maß verwendet. Beide Qualitätsmetriken werden hierbei identisch gewichtet. Dadurch stellt es das harmonische Mittel beider Metriken dar.

$$\mathrm{F}_1 = 2 \cdot \frac{\text{Präzision} \cdot \text{Ausbeute}}{\text{Präzision} + \text{Ausbeute}} \tag{2.5}$$

Beispiel 2.4.1: Präzision, Ausbeute, Genauigkeit und F_1-Maß

Ein Klassifkator sollte in einem Datensatz von Nachrichten unerwünschte Werbung erkennen. Der Datensatz und die Klassifikation sind hier grafisch dargestellt. Dabei stellt ein roter Kreis die unerwünschte Werbung dar. Grüne Kreise sind hingegen normale Nachrichten. Die durch den Klassifkator zugewiesene Klasse ist durch die rechteckige Umrandung dargestellt.

Durch die Zählung aller Fälle ergibt sich folgende Konfusionsmatrix:

	Instanz Werbung	Instanz Normal
Klassifikation Werbung	4	1
Klassifikation Normal	2	3

Anhand der Konfusionsmatrix lassen sich Präzision, Ausbeute, Genauigkeit und das F_1-Maß berechnen:

Präzision	**Ausbeute**	**Genauigkeit**	**F_1-Maß**
80,00%	66,67%	70,00%	72,72%

Die vorherigen Metriken dienen zur Bewertung binärer Klassifikatoren. Sie können daher nur zwischen „Richtig“ und „Falsch“ unterscheiden. Bei der Sprachverarbeitung und Codeerzeugung kann es aber vorkommen, dass dies nicht ausreichend ist. Ein Beispiel hierfür wäre die maschinelle Übersetzung von natürlicher Sprache in eine Fremdsprache. Die Musterlösung würde hier von menschlichen Übersetzern erstellt werden. Hierbei können Übersetzer verschiedene Musterlösungen für die gleichen Sätze erstellen. Ein direkter Vergleich zwischen maschineller Übersetzung und den Musterlösungen ist schwierig. So können verschiedene Teile der Übersetzung mit der Musterlösung unterschiedlicher Übersetzer übereinstimmen. Auch stellen Synonyme, die in keiner der Musterlösungen erhalten sind, ein Problem dar. Es ist daher nötig die Ähnlichkeit der maschinellen Übersetzung festzustellen, anstatt ihrer Übereinstimmung mit der Musterlösung. Die **BLEU**-Metrik (engl. **bil**ingual **e**valuation **u**nderstudy) [PRWZ] erlaubt das Feststellen der Ähnlichkeit zwischen einem Kandidaten und mehreren Musterlösungen. Der BLEU-Wert eines Textes entspricht dem Durchschnitt der BLEU-Werte der einzelnen Sätze des Textes. Der BLEU-Wert eines Satzes berechnet sich durch einen Vergleich des Kandidaten mit den Musterlösungen. Da in der Regel die Berechnung auf einzelnen Worten nicht optimal ist, werden N-Gramme gebildet. N-Gramme sind zusammenhängende Wortsequenzen eines Satzes der Länge N. Am häufigsten wird BLEU-4 verwendet, welches auf 4-Grammen basiert. Für jedes einzigartige N-Gramm wird mit Gleichung 2.6 zuerst die maximale Häufigkeit des N-Gramms in den einzelnen Musterlösungen bestimmt. Die Summe der maximierten Häufigkeiten wird bei Gleichung 2.7 im Anschluss durch die Anzahl der N-Gramme des Kandidaten gemittelt.

$$\text{Maximale Häufigkeit(N-Gramm)} = \max_{\forall \text{Musterlösungen}} \big(\text{Anzahl(N-Gramm in Musterlösung)}\big) \tag{2.6}$$

$$\text{BLEU-N} = \frac{\sum_{\text{N-Gramm} \in \text{Kandidaten}} \big(\text{Maximale Häufigkeit(N-Gramm)}\big)}{\text{Gesamtanzahl an N-Grammen im Kandidaten}} \tag{2.7}$$

Beispiel 2.4.2: BLEU-2

Kandidat	the	man	brings	me	the	soda
Musterlösung 1	the	robot	brings	me	the	limo
Musterlösung 2	the	android	gets	me	the	soda

Zuerst müssen für den Kandidaten die Bigramme des Satzes gebildet werden. Für die Bigramme wird nun die Häufigkeit in den beiden Musterlösungen festgestellt:

Bigramm	the man	man brings	brings me	me the	the soda
Musterlösung 1	0	0	1	1	0
Musterlösung 2	0	0	0	1	1
Maximum	0	0	1	1	1

Für jedes Bigramm wird daraus die maximale Häufigkeit summiert und durch die Anzahl an Bigrammen im Kandidaten geteilt. Somit steht eine BLEU-2-Ähnlichkeit zwischen Kandidaten und Musterlösungen von $\frac{3}{5}$ fest.

$$\text{BLEU-2} = \frac{1+1+1}{5} = \frac{3}{5}$$

3 PARSE-Projekt

Am IPD Tichy des **K**arlsruher **I**nstitut für **T**echnologie (KIT) wird die PARSE-Rahmenarchitektur (**P**rogramming **AR**chitecturea for **S**poken **E**xplanations) entwickelt [WT]. Ziel des Projektes ist es ein domänenunabhängiges System zu entwickeln, welches in der Lage ist natürliche Spracheingaben eines Anwenders zu verstehen und in Programmcode zu übersetzen. Die allgemeine Anwendungsunabhängikeit ist hierbei ein von Anfang an definiertes Ziel von PARSE. Die Freiheit bei der Wahl der Domäne garantiert, dass PARSE an verschiedenen Orten zum Einsatz kommen kann. Dies erfordert auch eine Unabhängigkeit vom ausführenden Zielsystem und auch der Eingabe. So soll es möglich sein, dass PARSE dazu dienen kann einen Küchenroboter mit Gebärdensprache zu steuern oder über Spracheingabe einen intelligenten Assistenten zu programmieren. Von besonderem Interesse ist hierbei, dass PARSE längere und komplexere Befehlsfolgen verarbeiten kann.

Da diese Arbeit als Bestandteil von PARSE eingebunden ist, beschäftigt sich dieses Kapitel ausführlich mit dem Projekt. In Abschnitt 3.1 werden die vier Designprinzipien von PARSE erläutert. Abschnitt 3.2 beschreibt die Architektur des Systems und die Verarbeitung von Nutzereingaben. Dabei werden die verschiedenen Agenten, die ein Teil von PARSE sind und den zentralen Graph bearbeiten, kurz beschrieben. In Abschnitt 3.3 werden mögliche Zielsysteme und der für die Evaluation verwendete Sprachkorpus vorgestellt. Zum Schluss findet in Abschnitt 3.4 eine Einordnung dieser Arbeit in das PARSE-System statt.

3.1 Grundsätzliche Entwurfsprinzipien

Um das Ziel der allgemein anwendbaren Rahmenarchitektur zu erreichen, basiert der Entwurf von PARSE auf vier Grundprinzipien.

Agentenbasierter Entwurf

Das Grundprinzip des agentenbasierten Entwurfs führt zu einer unabhängigen und kontinuierlichen Entwicklung der einzelnen sprachverarbeitenden Agenten. Es ist daher möglich sie unabhängig voneinander zu modifizieren und evaluieren. Die Agenten können dadurch wissensbasierte, probabilistische oder regelbasierte Ansätze verfolgen. Die restlichen Implementierungen abseits der Agenten, sind ebenfalls modular aufgebaut.

Evaluationsgetriebene Entwicklung

Eine evaluationsgetriebene Entwicklung der Rahmenarchitektur bedingt eine konstante Evaluation des ganzen Systems und der einzelnen Module. Damit wird sichergestellt, dass

neue Erweiterungen regelmäßig auf ihre Qualität getestet werden und so Probleme frühzeitig erkannt werden. Hierfür wurde ein Korpus mit realistischen Anwendungsfällen erzeugt.

Wissensbasierter Entwurf

Mit einem wissensbasierten Entwurf kann ein tieferes Verständnis des gesprochenen Textes erreicht werden. Die Anbindung an Wissensrepräsentationen wie Ontologien erlaubt PARSE Mehrdeutigkeiten aufzulösen und Wissen über das Zielsystem und die Umgebung mitzuverarbeiten.

Domänenunabhängiges Sprachverständnis

Da PARSE als Rahmenarchitektur in verschiedenen Domänen Anwendung finden soll, müssen die Agenten und Module von der Repräsentation der Domäne getrennt sein. Soll PARSE an eine andere Domäne angepasst werden, so soll nur die Wissenrepräsentation gewechselt und die Agenten unmodifiziert wiederverwendet werden. Um diesen Wechsel möglichst einfach zu halten, wird Domänenwissen nur über eine fest definierte Schnittstelle abgerufen.

3.2 Architektur

Die in Abbildung 3.1 dargestellte PARSE-Architektur lässt sich in drei Hauptschritte separieren. Zu Beginn wird in einem Vorverarbeitungsfließband die Spracheingabe durch einen Spracherkenner (engl. Automatic Speech Recognizer [ASR]) eingelesen und durch seichte Sprachverarbeitungswerkzeuge (engl. shallow natural language processing) der Anfangsgraph aufgebaut. Hier werden beispielsweise Worte mit ihrer Wortart (engl. Part-of-Speech) oder Grundform (engl. Lemma) markiert. Der Entwurf erlaubt einen Austausch der Vorverarbeitung, um alternative Eingabemöglichkeiten, wie Gesten und schriftliche Texte, nutzen zu können.

Der Hauptteil der Architektur basiert auf Agenten. Dabei dient ein zentraler Graph als Repräsentation der Spracheingabe und auch als Speicherstruktur für die Ergebnisse der verschiedenen Agenten. Jeder Agent hat eine spezifische Aufgabe, die dazu dient weitere semantische Informationen anhand des Graphen zu gewinnen. Die Agenten arbeiten parallel und unabhängig voneinander, können aber die zurückgeschriebenen Ergebnisse der anderen Agenten nutzen. Der Graph wird durch diesen Vorgang abhängig inkrementell transformiert. Diese Kapselung der Agenten und Module ermöglicht es einzelne Teile leicht unabhängig von einander zu erweitern, testen und zu evaluieren.

In einem Nachverarbeitungsfließband soll zum Schluss der Verarbeitung der zentrale Graph zu lauffähigen Quellcode unabhängig vom Zielsystem und Programmiersprache übersetzt werden. Dabei soll es möglich sein, dass dem Zielsystem neue Methoden beigebracht werden oder komplexe Anweisungen als Befehlsfolge zur Ausführung bereit gestellt werden.

Um Informationen über die Domäne des Systems zu erhalten, existiert abseits der Hauptverarbeitung auch eine Anbindung an entsprechende themenspezifische Wissensrepräsentation (Ontologien). Diese sind zur leichteren Verwendung separiert und werden für die Verarbeitung sofern nötig verschmolzen.

3.2.1 Vorverarbeitung

Die Vorverabeitung dient dazu die Spracheingabe einzulesen und einen anfänglichen Graphen zu erzeugen. Die Architektur ist ein Fließband. Zu Beginn der Vorverarbeitung

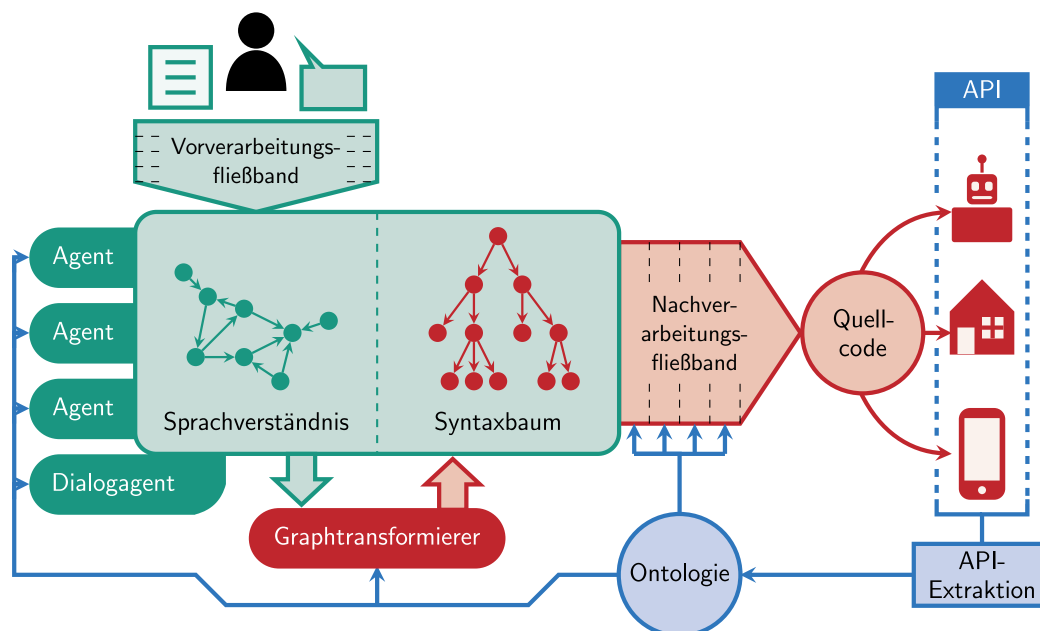


Abbildung 3.1: Dargestellt ist die PARSE-Architektur. Grün stellt in dieser Grafik die Teile der Architektur dar, die in anderen Arbeiten bereits bearbeitet wurden. Blau ist Umgebung, d.h. Wissensrepräsentation und Schnittsstelle zum Zielsystem. Der Teil, der in dieser Arbeit bearbeitet wird, ist rot hervorgehoben.

müssen die aufgenommenen Laute zur Weiterverarbeitung in eine textuelle Form überführt werden. Dies geschieht durch Spracherkenner (engl. automatic speech recognition [ASR]) [Pas]. Anhand von statistischen Modellen und maschinellen Lernverfahren werden Lautfolgen auf Worte abgebildet werden. Der Spracherkenner übergibt eine Liste von Tokens. Da dem gesprochenen Text wichtige Erkennungsmittel wie Satzzeichen fehlen, kann nur eine seichte Sprachverarbeitung (engl. shallow natural language processing) durchgeführt werden [Koc]. Darunter fallen Schritte wie die Wortartmarkierung und die Phrasenerkennung (engl. chunking). Um Entitäten identifizieren zu können, findet eine Eigennamenerkennung (engl. named-entity recognition) statt. Dies wird in Orts-, Personen-, Organisationsnamen oder auch sonstigen Namen (engl. miscellanous names) eingeteilt. Im Anschluss werden die Prädikate mit ihrer semantischen Rolle markiert. In der Vorverarbeitung wird als letzter Schritt die Liste von annotierten Tokens in einen Graphen überführt [Schb]. Hierbei werden die Tokens auf Knoten abgebildet. Die Kanten des Graphen stellen die Beziehungen zwischen den Tokens dar.

3.2.2 Agenten

Die PARSE-Rahmenarchitektur basiert auf Agenten, welche einen gemeinsamen Graphen bearbeiten. Die Agenten sind von einander unabhängig, was es erlaubt für verschiedene Agenten wissensbasierte und propabilistische Ansätze zu nutzen. Weiterhin ermöglicht es auch die Agenten separat zu entwickeln, testen und evaluieren. Die Agenten arbeiten nur auf dem Graphen und kommunizieren nicht direkt miteinander. In diesem Abschnitt werden die verschiedenen Agenten mit ihrer Funktion kurz beschrieben.

3.2.2.1 Aktionserkennung

Handlungen, die von einer Entität ausgeführt werden sollen, bezeichnet man als Aktionen. Sie werden von einem Agenten [Ou] detektiert, markiert und ihre sequentielle Abarbeitung

dient als Grundlage für den geplanten Ablauf. Der Agent ist auch dafür zuständig der Aktion ihre Akteure und Objekte zuzuweisen.

3.2.2.2 Synthese von Methodendefinitionen

Ein wichtiger Agent [Ste] synthetisiert Methodensignaturen aus dem natürlich sprachlichen Text. Dieser Agent unterscheidet auch ob es sich um die gegebenen Befehle um Befehlssequenzen oder Lehrsequenzen handelt. Lehrsequenzen bestimmen die Definition einer neuen Methode. Befehlssequenzen bilden hingegen einzelne Akteure, Aktionen und Objekte auf entsprechende Methoden und Parameter der Schnittstelle ab. Dies wird mit neuronalen Netzen erreicht.

3.2.2.3 Kontextanalyse

Die Kontextanalyse dient dazu den Graphen, um acht Arten von Kontext zu erweitern. Ein Agent [Hey] findet hierbei Entitäten und markiert sie mit ihrer Eigenschaften(bsw. `Tasse(rot/blau/grün)`), Zuständen(bsw. `Tasse(gefüllt/leer)`) und bildet sie auf ihre Konzepte und übergeordneten Konzepte ab. Konzepte können in verschiedenen Beziehungen zu einander stehen. Zusätzlich werden Aktionen mit den beteiligten Entitäten versehen. Zum Schluss werden die Entitäten in eine Ortsbeziehung, d.h. ihre räumliche Position zueinander, gebracht.

3.2.2.4 Koreferenzauflösung

Da in Parse komplexe Sätze und Befehlsfolgen verarbeitet werden sollen, werden Entitäten nicht immer mit ihrem vollständigen Namen aufgeführt. Ein Agent [Hey] muss daher die Koreferenzen von solchen Worten auflösen. In PARSE verwendet der Agent zur Auflösung von Korefenzen die Ergebnisse der Kontexnanalyse um unwahrscheinliche Entitäten auszusieben.

Beispiel 3.2.1: Koreferenzauflösung

„*Robo, you(Robo) need to grab the red bottle and then you(Robo) bring it(red bottle) to me(Speaker).*“

In der Beschreibung werden sind drei unterschiedliche Entitäten enthalten. Die grün markierten Entitäten und Referenzen beziehen sich auf den Roboter und die rot hinterlegten auf die rote Flasche. Zum Schluss wird der Sprecher in blau als implizite Entität erkannt. Dieser wird im Satz nicht explizit genannt.

3.2.2.5 Aktionsidentitätsauflösung

Ähnlich der Koreferenzauflösung bei Entitäten, ist es nötig zu erkennen, wann sich eine gesprochene Aktion auf eine bereits vorher genannte Aktion bezieht. Dies vermeidet Probleme bei nicht-chronologischer Beschreibung der Befehle. Zwar existiert hierfür bereits eine Arbeit [Bit], aber noch kein fertiger Agent. Aus diesem Grund gibt es auch noch keine Auflösung der temporalen Reihenfolge der Befehle.

Beispiel 3.2.2: Aktionsidentitätauflösung

„*(1) Open the door and then (3) come back. But after you've (1) unlocked the door (2) do...*“

Der Satz enthält drei Aktionen, die in nicht sequentieller Reihenfolge gegeben werden. Dabei stellen die beiden rot markierten Teilsätze die selbe Aktionen dar. Sie soll daher nur einmal ausgeführt werden. Zeitgleich soll die blau hinterlegte Aktion vor der grün hervorgehobenen Aktion ausgeführt werden.

3.2.2.6 Kontrollstrukturen

Kontrollstrukturen sind für die Regelung des Programmablaufs essentiell. In PARSE gibt es mehrere Agenten [WHS], die für Erkennung und Analyse von Kontrollstrukturen zuständig sind. Der Agent für die Bedingungsanalyse erkennt Wenn-Dann-Sonst-Sätze und teilt diese in ihre drei Komponenten auf. Zusätzlich gibt es weitere Agenten für die Erkennung von Schleifen und Parallelität. Schleifen werden je nach Typ in eine Kopf- und Rumpfkomponente getrennt. Bei der Parallelität werden die Aktionen, die gleichzeitig auszuführen sind extrahiert. Zur Erkennung von Kontrollstrukturen wird ein regelbasiertes Verfahren verwendet.

Beispiel 3.2.3: Kontrollstrukturen

„*if the dishwasher is full please start it otherwise bring me coffee*“

Im Satz ist eine Verzweigung dargestellt. In grün ist die Bedingung markiert. Wird die Bedingung zu wahr ausgewertet, so soll die rote Aktion ausgeführt werden. Sonst wird die blau hinterlegte Aktion ausgeführt.

3.2.2.7 Domänenzuordnung

PARSE soll in unterschiedlichen Domänen eingesetzt werden können. Dazu werden für alle Domänen Repräsentationen angelegt und zur Laufzeit entschieden, welche Repräsentation benötigt wird. Die Zuordnung der Eingabe des Nutzers auf eine Repräsentation geschieht durch die Extraktion und Zuordnung der Themen der Eingabe auf eine Repräsentation [Kei].

3.2.2.8 Disfluenzenerkennung

Beim Erteilen von Befehlen durch Menschen kann es zu Disfluenzen kommen. Disfluenzen sind beispielsweise Verzögerungswörter oder die Benennung der falschen Entität, welche noch im selben Satz korrigiert wird. Bei der Verarbeitung der Befehle müssen diese Disfluenzen ignoriert werden. Ein Agent [Hoc] annotiert die fehlerhaften Teile, um so einen korrekten Satz ohne Disfluenzen zu erhalten.

Beispiel 3.2.4: Disfluenzen

„*Hey armar i uh want you to uhm take the clean dish i mean dirty dishes and uhm put them in the the dishclea dishwasher*“

Der Satz enthält mehrere Disfluenzen. Verzögerungen des Sprechers sind rot markiert. Blau hinterlegt sind Entitäten, die der Sprecher falsch benannt und anschließend korrigiert hat. Der von Disfluenzen bereinigte Satz ist grün hervorgehoben.

3.2.2.9 Mehrdeutigkeitsauflösung

Die Auflösung von Mehrdeutigkeiten ist notwendig, um korrekte Ergebnisse zu erreichen. In PARSE existieren hierfür zwei Agenten: Der erste Agent verwendet WordNet zur Auflösung von Mehrdeutigkeiten. Zu besseren Ergebnissen hingegen führt ein auf Wikipedia basierender Agent [Kei]. Da dieser Agent aber nicht in Echtzeit ausführbar ist, wird der WordNet basierte Agent verwendet. Die Ergebnisse des WordNet-Agenten werden zusätzlich durch andere Agenten besser verarbeitet.

3.2.2.10 Dialogagent

Zur Fehlererkennung wurde ein Dialogagent [Scha] eingebaut. Dieser untersucht den Graphen auf bestimmte Muster, welche auf Fehlerklassen hinweisen, und stößt einen Dialog mit dem Nutzer an. Der Nutzer gibt Antworten auf die ihm gestellten Fragen und der Graph wird daraufhin angepasst.

3.2.3 Ontologie

Um die Domänen und Schnittstelle des Zielsystems darzustellen, wird eine Ontologie als Wissenrepräsentation verwendet [Kei]. Dabei lassen sich die Konzepte der Ontologie in zwei Gruppen trennen: Der erste Teil beschreibt die Domäne des verwendeten Zielsystems und Entitäten mit denen das Zielsystem interagieren kann. Die Konzepte des Zielsystems stellen die zweite Gruppe der Ontologie dar und modellieren die Fähigkeiten der Schnittstelle. Damit eine Ontologie in PARSE verwendet werden kann, muss ihre Struktur PARSE bekannt gemacht werden. Dies geschieht durch eine Konfigurationsdatei. Als Konzepte gibt es in PARSE Informationen über das Zielsystem, die vom System ausführbaren Methoden und Parameter der Methoden. Hierbei werden auch konkrete Entitäten und Zustände offengelegt, die in der Domäne des Zielsystems verwendet werden. Als letzte Entitäten werden primitive Datentypen als Konzepte modelliert. Die Konzepte werden mit Beziehungen verbunden. Neben den Beziehungen zwischen den Konzepten, werden die von Entitäten annehmbaren Zustände und durch Methoden ausgelöste Zustandsübergänge beschrieben.

3.2.4 Nachverarbeitung

Sobald die Agenten des Hauptteils ihre Aufgabe erfüllt haben, kann die Nachverarbeitung beginnen. Der fertige Graph wird von einer Fließbandarchitektur übernommen und schrittweise in Quelltext übersetzt. Da PARSE eine Rahmenarchitektur darstellt, ist es wichtig, dass eine Erweiterbarkeit in andere Zielsysteme, Programmiersprachen und Umgebungen gegeben ist. Zur Darstellung der Schnittstelle mit dem ausführenden System wird eine Wissensrepräsentation verwendet.

3.3 Zielsysteme

Das Ziel des PARSE-Projekts ist es Programmierung in natürlicher Sprache unabhängig des Zielsystems zur Verfügung zu stellen. Dies erfordert, dass die Erweiterbarkeit bei der Übersetzung in Quellcode beachtet wird. In diesem Abschnitt sollen drei Zielsysteme kurz beschrieben werden, für die Quelltext erzeugt wird.

ARMAR-III [ARA+] ist ein humanoider Küchenroboter, welcher am KIT entwickelt wurde. In der Bachelorarbeit von Günes [Gü] wurde für PARSE ein Korpus angelegt, bestehend aus Befehlsfolgen für ARMAR-III. Diese gesprochenen Befehlsfolgen bestehen aus mehreren Sätzen und beschreiben komplexe Szenarien. Sie wurden in einer Studie aufgenommen und anschließend transkribiert. So enthalten die Texte Kontrollstrukturen wie Verzweigungen, Wiederholungen und Nebenläufigkeiten. Da der Korpus auf gesprochener natürlicher Sprache basiert enthält er keine Satzzeichen. Weiterhin existieren Eigenheiten der gesprochenen Sprache, die bei einem schriftlichen Text fehlen. Beispiele hierfür sind Disfluenzen, vom Spracherkenner falsch erkannte Worte und grammatikalische Fehler. Die verwendete Sprache ist Englisch. Der Korpus besteht aus 238 Texten die sich auf 13 Szenarien verteilen.

Als zweites Zielsystem dient ein Legoroboter. Am IPD Tichy wird der Legoroboter dazu verwendet Studenten Konzepte der Sprachverarbeitung praktisch näherzubringen. Dabei erstellen die Studenten Sprachverarbeitungswerkzeuge, Ontologien und Korpora für den Legoroboter. Durch einen Austausch der Ontologie kann PARSE zur Erzeugung von Quelltext für den Legoroboter verwendet werden.

Die Synthese von UML-Aktivitätsdiagrammen mit der Auszeichnungssprache *PlantUML*[1] stellt das dritte Zielsystem dar. Die PlantUML-Syntax für Aktivitätsdiagramme entspricht einer prozeduralen Programmiersprache. Dadurch lassen sich Anweisungen in natürlicher Sprache mit PARSE in PlantUML-Quelltext übersetzen. Eine interessante Eigenschaft von PlantUML ist, dass natürliche Sprache nicht in Methoden übersetzt werden muss. Das erlaubt eine Erzeugung von PlantUML-Quelltext unabhängig einer konkreten Ontologie.

3.4 Einordnung dieser Arbeit ins Projekt

In diesem Kapitel wurde ein Überblick über das PARSE-Projekt gegeben. Da diese Arbeit in PARSE integriert wird, stehen die Ergebnisse der Agenten zur Erfüllung der Aufgabe zur Verfügung. Dadurch besteht die Möglichkeit mit komplexen Strukturen, die in vorherigen Arbeiten erstellt wurden, zu arbeiten. Das Ziel dieser Arbeit ist die Entwicklung eines Werkzeugs, dass den vollständig aufgebauten Graphen in lauffähigen Code übersetzt. Daraus ist ersichtlich, dass die Arbeit größtenteils dem Nachverarbeitungsschritt zuzuordnen ist. Darauf aufbauend, erlaubt die Verwendung von PARSE eine tiefere Analyse der Problemstellung und stellt auch eine größere Breite an Möglichkeiten zur Lösungsfindung zur Verfügung. Weiterhin wird durch PARSE die Evaluation der Arbeit vereinfacht, da es nicht nötig ist, einen eigenen Korpus zu erstellen.

[1] `https://plantuml.com/de/`, zuletzt besucht am 09.12.2019

4 Verwandte Arbeiten

In diesem Kapitel soll ein genereller Überblick über den aktuellen Stand der Forschung im Bereich „Programmierung durch natürliche Sprache“ gegeben werden. Der Fokus richtet sich hierbei auf die für diese Arbeit relevanten Ansätze. Dieses Kapitel teilt sich dabei in zwei Teile. In Abschnitt 4.1 werden Arbeiten zur Quelltextsynthese behandelt. Zur Einleitung werden daher in Abschnitt 4.1.1 einige regel-basierte Ansätze vorgestellt, die natürliche Sprache in Quelltext übersetzen. Im Gegensatz dazu, führte das gestiegene Interesse an neuronalen Netzen in der Forschung dazu, dass diese auch in der Sprachverarbeitung zum Einsatz kommen. Arbeiten, die auf neuronalen Netzen basieren, werden daher in Abschnitt 4.1.2 betrachtet. Auf die neuronalen Netze folgt eine gezielte Annäherung an das Thema dieser Arbeit. Da die Kommunikation mit humanoiden Robotern über natürliche Sprache naheliegt, findet auf dieser Domäne einige Forschung statt. Daher wird der Fokus in Abschnitt 4.1.3 auf die Steuerung von Robotern mit natürlicher Sprache gelegt. Danach werden in Abschnitt 4.1.4 Arbeiten vorgestellt, die einen abstrakten Syntaxbaum als Zwischenrepräsentation verwenden. Der zweite Teil dieses Kapitels betrachtet in Abschnitt 4.2 Forschungsarbeiten aus der Programmtransformation. Das Hauptaugenmerk wird dabei auf die Übersetzung von Syntaxbäumen in Quelltext gerichtet. Abschließend wird in Abschnitt 4.3 der aktuellen Stand zusammengefasst.

4.1 Quelltextsynthese

Die Sprachverarbeitung beschäftigt sich mit einer Vielzahl von Themen im Bezug zu natürlicher Sprache. In diesen Forschungsbereich fallen Themen wie das Klassifizieren von grammatikalischen Strukturen, das Verständnis von Sprache und die Synthese von natürlichen Texten. Das Thema „Programmierung in natürlicher Sprache“ kombiniert dabei viele Themen der Sprachverarbeitung. So muss beispielsweise die Beschreibung eines erst Programms verstanden werden, um daraus Quelltext erzeugen zu können.

Ein allgemeiner Überblick über die verschiedene Ansätze zu „Programmieren in natürlicher Sprache“ gibt die Arbeit von Pulido-Prieto und Juárez-Martínez [PPJM]. Neben der Zusammenfassung verschiedener Methoden, werden auch vier Herausforderungen identifiziert. Als erstes werden Mehrdeutigkeiten bei indirekten Referenzen genannt. Während bei Programmiersprachen eine Entität jedes mal explizit genannt wird, findet dies bei natürlicher Sprache meist indirekt statt. Als nächste Herausforderungen werden das Fehlen von Iterationen und Verzweigungen in natürlicher Sprache aufgelistet. Beide Kontrollstrukturen sind von enormer Bedeutung bei formaler Programmierung. Bei natürlicher Sprache

wird implizit erwartet, dass der Geltungsbereich der Kontrollstruktur verstanden wird. Als letzte Herausforderung wird die Abstraktion bei natürlicher Sprache genannt. Entitäten können zum einen abstrakte Konstrukte in einer Hierarchie sein und zum anderen durch andere Wörter modifiziert werden. Weiterhin kommen sie zum Schluss, dass viele Ansätze das Ziel einschränken um zu einer teilweisen Lösung zu kommen.

4.1.1 Ansätze basierend auf Sprachverarbeitungsfließbändern

Die Verarbeitung von Sprache findet in der Regel über eine Fließbandarchitektur statt. Viele ältere Arbeiten nutzen diese Architektur. Der Einsatz eines Fließbandes ist aber auch in modernen Zeiten die gängigste Methode.

Eine frühe Arbeit aus dem Jahr 2000, die das Thema Programmierung in natürlicher Sprache betrachtet, ist *NaturalJava* [PRZH]. *NaturalJava* ist eine Benutzerschnittstelle zur Programmierung mit Java. Es dient dazu einem erfahrenen Programmierer zu ermöglichen in natürlicher Sprache zu programmieren. Ziel ist es die Syntax der Programmiersprache durch natürliche Spracheingaben zu abstrahieren. *NaturalJava* besteht aus den drei Teilsystemen `Sundance`, `PRISM` und `TreeFace`. `Sundance` extrahiert Kasusrahmen [Fil] aus englischen Eingabesätzen. Dabei erzwingen bestimme Wörter entsprechenden Kasusrahmen. Diese Kasusrahmen werden von `PRISM` mit Hilfe von Entscheidungsbäumen interpretiert. Sie werden dazu verwendet Quelltext über einzelne, formale Befehle zu modifizieren oder zu erweitern. `TreeFace` wird von `PRISM` als Schnittstelle zwischen Quelltext und *NaturalJava* verwendet. Es stellt das Java-Programm als abstrakten Syntaxbaum dar und erlaubt so die Navigation innerhalb des Quelltextes. Beschränkungen von *NaturalJava* liegen darin, dass es eine Benutzerschnittstelle ist und so nur Modifikationen des Quelltextes mit einzelnen Befehlen durchführt. Es ist so nicht möglich Befehlsbeschreibungen ohne Bezug zum Quelltext zu übersetzen. Ein erfahrener Programmierer ist zur Verwendung notwendig. Dazu verwendet es feste Begriffe, um Aktionen durchzuführen. Dies führt dazu, dass der Einsatzbereich stark eingeschränkt ist. Zuletzt fehlen auch Komfortfunktionen moderner Entwicklungsumgebungen. So muss man beispielsweise bei einer Namensänderung jede Variable einzeln korrigieren. Zur Evaluation haben zwei Programmierer eine Vergleichsmethode mit *NaturalJava* implementiert. Die Aufgabe war es durch eigene Beschreibungen in natürlicher Sprache zum gleichen Quelltext zu kommen. Dies wurde erreicht.

SmartSynth [LGS] ist eine weitere Arbeit, die sich mit dem der Erzeugung von Quelltext aus natürlicher Sprache beschäftigt. Ziel von *SmartSynth* ist es Anwendern zu ermöglichen ihre Mobiltelefone mit natürlicher Sprache zu programmieren. *SmartSynth* verwendet dafür eine eigene domänenspezifische Programmiersprache für Mobiltelefone. Anders als bei *NaturalJava* ist die Eingabe aber grundsätzlich frei. Das heißt, der Anwender ist nicht gezwungen feste Begriffe zu verwenden. Der in der Arbeit verwendete Ansatz nutzt Methoden aus der Sprachverarbeitung, um Komponenten der Schnittstelle und ihre Relation zueinander zu extrahieren. Zuerst werden die Komponenten identifiziert. Dazu werden die Sätze in Teilsätze zerlegt und diese anhand ihrer Merkmale bewertet. Beispiele für Merkmale die Auswirkungen auf die Bewertung haben wären die Ähnlichkeit des Teilsatzes zur Komponente, passende reguläre Ausdrücke und die Länge der Teilsätze. Für die Erkennung der Beziehungen zwischen den Komponenten wird ein regel-basierter Ansatz angewendet. Die Regeln wurden von Hand erstellt. Zusätzlich werden Ansätze aus dem Bereich der Programmsynthese angewendet, um fehlende Strukturen abzuleiten. Hierbei wird beispielsweise für Parameter, denen ein passendes Argument fehlt, ein solches gesucht. Die Suche funktioniert über die Bewertung aller möglichen Beziehungen zwischen den Komponenten. Diese Komponenten und Strukturen, die den Datenfluss beschreiben, werden dann in domänenspezifischen Quelltext übersetzt. Die Codeerzeugung ähnelt dabei einem umgekehren Zerteiler (engl. reverse parser). *SmartSynth* ist auch in der Lage

zur Fehlerbehebung mit dem Anwender einen Dialog anzustoßen. Da es auf Konfidenzen basiert, ist es in der Lage dem Nutzer mehrere Quelltexte vorzustellen, sodass dieser das korrekte Skript auswählen kann. *SmartSynth* wurde auf 640 Texten aus 50 Szenarien evaluiert. Werden Methoden aus der Programmsynthese eingesetzt, so ist es in der Lage eine Korrektheit von 90% zu erreichen. Werden ausschließlich Methoden aus der Verarbeitung natürlicher Sprache eingesetzt, wird die Korrektheit hingegen auf 58,7% reduziert. Probleme existieren bei der Synchronisation zwischen verschiedenen Anwendungen und bei Kontrollstrukturen. Letzteres führt dazu, dass die Komplexität der Eingabe aber einfach gehalten werden muss. Beim Erstellen des Korpus fanden Le et al. heraus, dass ca. 10% der von Anwender gefragten Skripte deshalb nicht synthetisierbar sind.

*NLCI (**N**atural **L**anguage **C**ommand **I**nterpreter)* von Landhäußer et al. [LWT] wurde ebenfalls am IPD Tichy entwickelt. Das Ziel von *NLCI* ist es Entwicklern zu ermöglichen schnell und einfach ihre Anwendungen, um natürlichsprachliche Schnittstellen zu erweitern. *NLCI* ist in der Lage Sätze in natürlicher Sprache auf Methodenaufrufe einer Anwendungsschnittstelle abzubilden. Eine Ontologie dient hierbei als Kernkomponente, die die Verbindung zur Schnittstelle der Anwendung darstellt. Bei einer Anwendungsschnittstelle, bei der die Methoden ausreichend deskriptiv benannt sind, lässt sich die Ontologie automatisch erstellen. Dazu werden auch die Kommentare und JavaDocs durchsucht. Durch einen Austausch der Ontologie ist es möglich *NLCI* auf andere Domänen anzupassen. Die Architektur von *NLCI* besteht aus einem sprachverarbeitenden Fließband. Um natürliche Sprache in Quelltext zu übersetzten, wird die Eingabe in atomare Sätze getrennt. Atomare Sätze stellen hierbei Aktionen da, welche auf die entsprechenden Methoden der Schnittstelle abgebildet werden können. Sie bestehen aus einer Aktion mit allen nötigen Informationen wie Akteur und Objekt, um diese Aktion auszuführen. Jeden Satz wird mit verschiedenen Informationen angereichert, woraus die aufzurufenden Methoden, Klassen und Parameter extrahiert werden. Die Abbildung auf die Schnittstelle findet durch Konfidenzen statt. Hierbei fließt die Übereinstimmung des Namens, die Hierarchie des Konzepts und auch Synonyme ein. *NLCI* ist in der Lage die korrekte Ausführungsreihenfolge und Kontrollstrukturen durch Signalwörter zu erkennen. Für die Erzeugung des Quelltextes baut *NLCI* einen abstrakten Syntaxbaum des Programablaufs auf. Die Ausführungsreihenfolge und Kontrollstrukturen werden in diesem Syntaxbaum explizit dargestellt. *NLCI* maximiert bei der Erzeugung des Baumes alle Konfidenzen gemeinsam. Eine automatische Korrektur, beispielsweise von fehlenden Befehlsteilen, findet nicht statt. Die Evaluation fand auf zwei Korpora statt. Bei der Haussteuerung *OpenHAB* erreichte *NLCI* eine Genauigkeit von 73,7% und eine Trefferquote von 70%. Der Korpus bestand aber nur aus 15 Befehlssätzen verteilt auf 5 Texte. Das Programm *Alice*, dass dazu dient 3D-Szenen, ähnlich einem Theater, zu beschreiben, hat mit 703 Befehlssätzen in 50 Texten den größeren Korpus. Dort erreichte *NLCI* eine Präzision von 78% und eine Ausbeute von 67%.

4.1.2 Ansätze basierend auf neuronalen Netzen

Mit der Entwicklung leistungsstarker paralleler Rechner stieg das Interesse der Forschung an neuronalen Netze. Neuronale Netze fanden so auch Einzug in die Computerlinguistik. Ein Überblick über die Einsatzmöglichkeiten bei der Sprachverarbeitung wird in der Arbeit von Otter et al. [OMK] gegeben. In diesem Abschnitt liegt der Fokus auf der Übersetzung von natürlicher Sprache in Quelltext.

Ein erstes Beispiel für Ansätze, die auf neuronalen Netzen basieren, wäre die Fallstudie von Mou et al. [MML+]. Als Methode verwenden die Autoren ein rekurrentes neuronales Netz, welches einzelne Buchstaben einliest und buchstabenweise Quelltext ausgibt. Ihr einfacher Ansatz scheitert aber daran korrekten, lauffähigen Quelltext zu erzeugen. Von 280 generierten Buchstaben waren vier fehlerhaft, die dadurch zu verschiedenen Fehlern führten. Falsche Variablennamen, falscher Vergleichsoperator und fehlerhafter Rückgabetyp waren

beispielhafte Fehler. Sie schließen daraus aber, dass eine grundsätzliche Übersetzung von natürlicher Sprache in Quelltext in Zukunft möglich zu sein scheint.

Einen ähnlichen Ansatz verwenden später Lin et al. für *Tellina* [LWP+]. Wie auch bei Mou et al. wird ein rekurrentes neuronales Netz verwendet. Anders als der einfache Ansatz wird das neuronale Netz mit verschiedenen Techniken verbessert. Die Architektur von Tellina basiert auf einem dreistufigen Fließband. Als erster Schritt findet eine Entitätserkennung auf dem natürlichsprachigen Text statt. Hierbei wird von domänenspezifischen Konstanten, wie beispielsweise Dateinamen und Zeiten, abstrahiert. Dadurch wird die Eingabe in eine allgemeine Form gebracht, die einer Textschablone entspricht. Die Daten werden für spätere Schritte als Argumente aufbewahrt. Darauf folgend übersetzt im zweiten Schritt ein rekurrentes neuronales Netz die Textschablone in eine gewichtete Liste von Programmschablonen. Im dritten Schritt werden, über eine k-Nächste-Nachbarn-Suche, die gespeicherten Daten als Argumente in die Programmschablonen eingesetzt. Damit wird der Quelltext erzeugt.

Für die Evaluation dienten über 5000 Beispielbefehle eines Kommandozeileninterpreters. *Tellina* fand in 30% der Fälle den vollständig korrekten Befehl. Dieser war zusätzlich innerhalb der besten drei Ergebnisse in 36% der Testfälle zu finden. Zusätzlich wurde nur die Befehlsstruktur, d.h. Befehle ohne Argumente, betrachtet. *Tellina* gibt hier die korrekte Struktur in 69,4% der Fälle aus. In 80% ist sie unter den besten drei Ergebnissen zu finden. Zwar waren die Ergebnisse die *Tellina* ausgab selten korrekt, aber die Verwendung von *Tellina* half dem Anwender den korrekten Befehl 22% schneller zu finden ($p < 0{,}1$). *Tellina* hat Schwierigkeiten bei Befehlen die Kontrollstrukturen enthalten.

4.1.3 Robotersteuerung

Menschenähnliche Roboter wie *ARMAR-III* werden in Zukunft viele Aspekte des täglichen Lebens beeinflussen. Es ist zu erwarten, dass die Bedienung durch natürliche Sprache geschieht. Daher liegt ein Fokus der Forschung darin, Roboter Befehlsfolgen in natürlicher Sprache in Handlungsabfolgen zu übersetzen. Die Handlungsabfolgen werden durch Quelltext repräsentiert.

Die Arbeit von Lincoln und Veres [LV] behandelt die Programmierung komplexer BDI-Agenten [**B**elief,**D**esire and **I**ntention] durch natürliche Sprache. BDI-Agenten bestehen aus Wissen über ihre Umwelt, Zielen, die sie verfolgen und möglichen Handlungsabsichten. In der Arbeit wird eine Entwicklungsumgebung zur Programmierung komplexer Systeme von BDI-Agenten in natürlicher Sprache vorgestellt. Als Eingabesprache dient „*sEnglish*“, welches zusätzlich in einer Ontologie die Datenstrukturen und Konzepte des BDI-Agenten enthält. Sie wird in die domänenspezifische Sprache „BDI Jason“ übersetzt. Dazu werden die Sätze iterativ in bekannte, spezifischere Teilsätze zerlegt. Die niedrigste Stufe hat triviale Sätze, die aus Programmcode bestehen. Dieser Ansatz benötigt Verständnis bei der Entwicklung von BDI-Agenten. Ähnlich zu *NaturalJava* dient die Eingabe in natürlicher Sprache dazu von der Syntax zu abstrahieren.

Eine weitere Arbeit zur Steuerung von Robotern ist von Khayallah et al. [KTF]. Dabei wird einem simulierten Roboter durch Sprachbefehle Anweisungen gegeben. Die Anweisungen sind von „moderater Komplexität“ und können auch Fragen sein. Die Architektur besteht aus einem vierstufigen Fließband. Zuerst zerteilt ein Analysierer die Spracheingabe semantisch und übergibt es an einen Spezialisierer. Der Analysierer nutzt dafür eine Ontologie und *„Embodied Construction Grammars“* [FDB] Der Spezialisierer extrahiert die relevanten Informationen zum Ausführen der Anweisung. Als nächster Schritt werden die Informationen von einem Problemlöser mit Hilfe einer Ontologie verarbeitet. Hier erzeugt der Problemlöser konkrete Anweisungen die der simulierte Roboter ausführen soll und aktualisiert die Ontologie. Der Problemlöser bestimmt zuerst den Befehlstyp. Die

Teilprobleme der Anweisung werden je nach Typ mit Hilfe eines Modells der Welt gelöst. Die Lösung dieser Teilprobleme wird dann in Steuerbefehle für den simulierten Roboter übersetzt. Ein Vorteil ist, dass bei Wechsel der Eingabesprache nur der Spezialisierer ausgewechselt werden muss. Eine genannte Einschränkung ist die Skalierbarkeit. Für das Verständnis von mehr Texten, werden größere Grammatiken benötigt.

Matuszek et al. [MHZF] nutzen maschinelle Lernverfahren, um ein Roboterkontrollsystem zu befehligen. Ein Zerteiler wird hier auf Paaren von Anweisungen in englischer Sprache und dazugehörigen Befehlen in der Kontrollsprache gelernt. Die Grundlage für den Zerteiler bildet der *„Unification Based Learner“* [KZGS], welcher um Wortsequenzen und semantische Strukturen als Eingabe erweitert wurde. Bei unbekannten Worten findet eine Synonymersetzung statt. Weiterhin wurden Zahlen durch Variablen ersetzt. Ziel ist es, dass der Zerteiler anhand der Beispiele die grundlegenden Beziehungen zwischen englischen Anweisungen und Kontrollsprache versteht, anstatt nur Anweisungen in natürlicher Sprache auf entsprechende Befehle in Kontrollsprache abzubilden. Als Anwendungsdomäne dient die Pfadbeschreibung eines Roboters durch Sprachbefehle in natürlicher, englischer Sprache. Hierbei wird dem Roboter in einer simulierten Stadt ein Weg beschrieben. Diese Wegbeschreibung soll er selbstständig interpretieren und dann folgen. Übersetzt wird in die domänenspezifische Roboterkontrollsprache *RCL*, die der Programmiersprache *LISP* ähnelt. Sie besitzt Kontrollstrukturen und Konzepte höherer Ordnung. Eine Simulation in der Domäne der Routenplanung dient als Grundlage für die Evaluation. Hierbei werden zwei Korpora verwendet. Der erste Korpus enthält 189 einfache Pfadbeschreibungen. Der Roboter erreicht hier das Ziel in 66% der Fälle. Beim zweiten Korpus mit 418 komplexeren Beschreibungen und Bedingungen, sinken die korrekt verstandenen Pfadbeschreibungen auf 49%. Neben der Simulation findet auch eine Kreuzevaluation statt, um den Zerteiler direkt zu evaluieren. Hierbei erreicht der Zerteiler, bei exakter Übersetzung eine Präzision von 71%. Es wird mehrfach erwähnt, dass keine Fehlerkorrektur stattfindet.

In der Arbeit von Lauria et al. [LBK+] werden einem Roboter durch sprachliche Erklärungen neue Fähigkeiten beigebracht. Als Domäne dient auch hier die Pfadbeschreibung eines Roboters zu einem bestimmten Ort. Um das Gespräch zwischen System und Nutzer modellieren, wird eine semantische Zwischenrepräsentation in Form von *„**D**iscourse **R**epresentation“* **S**tructures [DSR] verwendet. *DRS* bestehen aus einer Menge von Entitäten und einer Menge von Einschränkungen der Entitäten. In der Arbeit wird die Struktur um atomare und zusammengesetzte Aktionen erweitert. Über Regeln werden aus den *DRS* werden ausführbare Primitive extrahiert. Die Regeln werden durch die *„Procedure Specification Language“* definiert. Eine Sequenz von Primitiven stellt die Handlungsanweisungen des Roboter dar. Die Reihenfolge der Sequenz ist bereits in den *DRS* enthalten. Wiederholungen werden durch spezielle Prozeduren (`goUntilSeeColor(Forward, Red)`) implementiert. Wird eine Regel in der *DRS* gefunden, so wird sie direkt in Quelltext übersetzt. Dieser Ansatz erlaubt die Wiederverwendung von Prozeduren, die dem Roboter vorher beigebracht wurden. Bei der Wegweisung wären dies beigebrachte Orte.

4.1.4 Abstrakter Syntaxbaum

In dieser Arbeit soll ein abstrakter Syntaxbaum als Zwischenrepräsentation verwendet werden. Diese Abstraktion vom Quelltext ermöglicht eine Trennung von Programmablauf und konkreter Programmiersprache. Die Übersetzung von Syntaxbaum zu Programmiersprache findet in einem separaten Schritt statt. In diesem Abschnitt sollen daher Arbeiten betrachtet werden, die einen abstrakten Syntaxbaum als Zwischenrepräsentation nutzen.

Yin und Neubig [YN] kombinieren eine „probabilistische Grammatik“ mit einen rekurrenten neuronalen Netz, um Anweisungen in natürlicher Sprache in abstrakte Syntaxbäume zu übersetzen. Das neuronale Netz basiert auf einer Enkodierer-Dekodierer-Architektur.

Die probabilistische Grammatik dient dazu, die die Syntax einer Programmiersprache zu beschreiben. Der Syntaxbaum wird schrittweise durch ein neuronales Netz aufgebaut, welches sequentiell jeweils zwischen zwei Aktionen wählt. Die erste Aktion ist die Anwendung von Ableitungsregeln der Grammatik auf den Baum. Die zweite Aktion ist das Zuweisen von Daten, wie Bezeichnern und Konstanten, zu den Blättern des Syntaxbaums. Damit soll das neuronale Netz von der syntaktischen Struktur des Syntaxbaums entkoppelt werden. Der so erzeugte Syntaxbaum entspricht einem durch standardmäßigen Zerteiler der Programmiersprache erstellten Syntaxbaum. Dadurch kann er leicht in Quelltext übersetzt werden. Evaluiert wurde auf drei Datensätzen. Um die Korrektheit festzustellen, wird die „BLEU-4“-Metrik [PRWZ] verwendet. Der erste Datensatz besteht aus Handlungsanweisungen auf Spielkarten des Computerspiels *„Hearthstone“*. Dieser Korpus besteht aus einer Repräsentation der Karte in *JSON* und der entsprechenden Übersetzung in der Programmiersprache *Python*. Hierbei wurde eine Ähnlichkeit zwischen Musterlösungen und erzeugten Quelltext von 75,8% erreicht. Beim Datensatz *„Django“* konnte eine Ähnlichkeit von 84,5% erreicht werden. Dieser Datensatz beschreibt Anwendungsbeispiele für eine Internetseiten-Rahmenarchitektur mit Quelltext und Beschreibung in natürlicher Sprache. Der dritte Datensatz *„IFTTT“* dient dazu die Domänenspezifität zu überprüfen. Er besteht aus einfachen Beispieln, implementiert in der domänenspezifischen Programmiersprache der Anwendungsautomatisierung IFTTT. Dabei wird eine Ähnlichkeit des gesamten Syntaxbaumes von 82,0% erreicht.

Die Arbeit von Rabinovich et al. [RSK] nutzt *„abstrakte Syntaxnetze“ (engl. abstract syntax networks)* zur Quelltexterzeugung und semantischen Zerteilung aus natürlicher Sprache. Bei der semantischen Zerteilung werden Anfragen in natürlicher Sprache auf eine formale Repräsentation der Bedeutung abgebildet. In der Arbeit werden abstrakte Syntaxbäume als Zwischenrepräsentation verwendet. Abstrakte Syntaxnetze basieren auf rekurrenten neuronalen Netzen mit Enkodierer-Dekodierer-Architektur. In der Arbeit wurde das Dekodierer-Modell um modulare Untermodule erweitert. Jedes dieser Untermodule ist dafür zuständig bestimmte Strukturen des abstrakten Syntaxbaumes zu erzeugen. So baut beispielsweise ein Untermodul die Struktur für Konstruktoren auf und ein anderes Untermodul weist einzelnen Datentypen primitive Werte zu. Der Syntaxbaum wird von der Wurzel aus aufgebaut und die Untermodule wählen Kindermodule rekursiv aus. Dadurch spiegelt die Wahl der Module den Syntaxbaum wieder. Die Evaluation findet auf zwei Arten statt. Zuerst wird die semantische Zerteilung auf drei Korpora evaluiert. Die abstrakten Syntaxnetze erreichen hierbei eine Genauigkeit von 85,9% auf *ATIS*, 87,1% auf *GEO* und 92,9% auf *JOBS*. Für die Erzeugung von Quelltext werden Spielkarten mit Anweisungen verschiedener Komplexität des Computerspiels *„Hearthstone“* als Korpus verwendet. Bei der exakten Übersetzung wird eine Genauigkeit von 22,7% erreicht. Mit der *„BLEU-4“*-Metrik hingegen kann eine Ähnlichkeit von 79,2% erreicht werden. Die Knoten des Syntaxbaum werden noch auf Ähnlichkeit überprüft. Hierbei wird die Schnittmenge zwischen dem erzeugten Syntaxbaum und dem Goldstandard berechnet. Der Wert für das F_1-Maß beträgt hierbei 75,6%.

4.2 Programmtransformation

Neben der Computerlinguistik greift diese Masterarbeit auch Themen aus der Programmtransformation auf. Das Übersetzen (engl. pretty printing) eines abstrakten Syntaxbaums in Quelltext findet häufig bei der *„Transpilierung“* Anwendung. Als Transpilierung ist die Transformation von Quelltext einer Programmiersprache in eine andere zu verstehen. Zwar ist die Aufgabenstellung dieser Masterarbeit die „Programmierung in natürlicher Sprache“, die Eingabe wird aber durch *PARSE* (siehe Kapitel 3) vorverarbeitet. Aus der Eingabe in natürlicher Sprache wird daher ein Modell in Form eines Graphen erzeugt. Dieses Modell

stellt eine Abstraktion von der natürlichen Sprache dar. Daher ist es überlegenswert Ansätze aus der Forschung des Bereichs der Programmtransformation in Betracht zu ziehen.

Bei der Arbeit von Hasu [Has] beschäftigt sich damit XML-Quelltext aus einem formalen Modell zu erzeugen und verwendet für die Quelltextsynthese einen Syntaxbaum als Zwischenrepräsentation. Dabei wird „konstruktives Übersetzen“ (engl. constructive pretty printing) als Konzept vorgestellt. Beim konstruktiven Übersetzen wird iterativ ein Modell aufgebaut, welches daraufhin übersetzt wird. Das Konzept besteht aus drei Teilen. Der Nutzer baut ein Modell auf, welches die gewünschte Struktur widerspiegelt. Eine Grammatik enthält Ableitungsregeln, welche das Modell in einen Syntaxbaum übersetzen. Der Syntaxbaum wird über ein Besucher-Muster abgelaufen und ausgeschrieben. Dies Unterscheidet sich zu Schablonen-Systemen (engl. template engines), bei welchen eine Vorlage durch Argumente ausgefüllt wird.

Jonge [Jon] beschäftigt sich mit Quelltextübersetzungen zum Zwecke einer Anpassung der Software bei gleichbleibender Funktionalität. Beispiele hierfür wären die Formatierung oder Erzeugung von Quelltext in unterschiedlichen Sprachen. Die Übersetzung ist in zwei Phasen und Module separiert und erlaubt so eine leichtere Erweiterbarkeit der Eingabe und Ausgabe. Die erste Phase ist die Erweiterung der abstrakten Repräsentation, um Informationen zur Formatierung. Das heißt der abstrakte Syntaxbaum wird zu einem Gestaltungsbaum (engl. format tree) transformiert. Dieser Gestaltungsbaum ist in der Auszeichnungssprache *BOX* [BV] definiert. BOX ist eine Auszeichnungssprache (engl. markup language), die die Formatierung als Anordnung von BOX-Ausdrücken darstellt. Diese Ausdrücke können in der zweiten Phase in Text übersetzt werden. Dazu werden ebenfalls Besucher mit Texterzeugungsregeln verwendet. Weiterhin werden weiterführende Techniken bei der Übersetzung formuliert. Als erste von vier Techniken wird die anpassbare Übersetzung beschrieben. Hierbei soll bei der Übersetzung eine spezifische Formatierung eingehalten werden. In der Arbeit werden dafür Übersetzungsregeln vorgestellt, die spezifizieren wie Konstrukte in Quelltext übersetzt werden. Sie sind dabei je nach Programmiersprache und Anwender anpassbar. Als nächstes wird konservatives Übersetzen genannt, wobei die Gestaltung des ursprünglichen Quelltextes erhalten bleibt. Die dritte Technik sind die modularen Übersetzungstabellen. Sie haben zum Ziel die Wiederverwendung von Regeln zu erleichtern. Eine Tabelle besteht dabei aus Übersetzungsregeln mit dazugehöriger Priorität. Es ist nun möglich mehrere Tabellen gemeinsam zu verwenden. Regeln mit niedrigerer Priorität werden aber von denen mit höherer Priorität überschrieben. Daher ist es beispielsweise möglich allgemeine Regeln für eine Programmiersprache zu erstellen und diese mit spezifischen Regeln zur Formatierung zu spezifizieren. Als letzte Technik wird eine gemeinsame Methode zum Übersetzen von konkreten und abstrakten Syntaxbäumen vorgestellt. Diese Techniken werden in Form der *„Generic Pretty Printers“*-Rahmenarchitektur implementiert.

4.3 Zusammenfassung

In diesem Kapitel wurde ein Überblick über den Stand der Forschung im Bereich der „Programmierung in natürlicher Sprache“ gegeben. Im Groben lässt sich die Art der Ansätze in zwei Gruppen teilen. Die traditionelle Fließbandarchitektur der Sprachverarbeitung scheint sich grundsätzlich bewährt zu haben. Die Fließbänder waren in der Lage lauffähigen Quelltext zu erzeugen, selbst wenn dieser semantisch nicht korrekt war. Bei der Erzeugung von semantisch korrektem Quelltext liegen die traditionellen Ansätze gegenüber neuronalen Netzen noch vorne. Bei den neuronalen Netzen hingegen ist ersichtlich, dass neuronale Netze alleine keine Lösung bringen. Eine direkte Übersetzung von Sprache zu Quelltext führt zu fehlerhaftem Quelltext. Daher separieren moderne Ansätze die Syntax von der Semantik und nutzen Grammatiken, um syntaktisch korrekten Quelltext über neuronale

Netze zu erzeugen. Da dieser Bereich der Forschung aber erst spät fokussiert wurde, sind hier noch starke Verbesserungen zu erwarten. Die Erstellung ausführlicher Korpora sollte hier auch große Fortschritte bringen.

Im Bereich der sprachverarbeitenden Quelltextsynthese schränken die meisten Ansätze die Zielsetzung ein. Bei der Eingabesprache werden oft bestimmte Satzstrukturen oder Begriffe vorgegeben. Damit wird der Mensch dazu gezwungen, seine natürliche Sprache zu formalisieren. Weiterhin gibt es auch starke Unterschiede bei der Komplexität der Eingabesätze. Viele Methoden verwenden nur einzelne Sätze, die einem konkreten Befehl ähneln. Alle Arbeiten schränken die Domäne ein. Dadurch wird der Raum der möglichen Lösungen verringert und die Genauigkeit erhöht. Viele Methoden erlauben es aber die Domäne, beispielsweise durch Wechsel der Ontologie, auszutauschen. Damit wird zumindest eine teilweise Unabhängigkeit von der Domäne erreicht. Da es selbst Menschen schwer fällt, den Kontext eines Gesprächs ohne Rückfragen immer korrekt festzustellen, ist diese Einschränkung vermutlich nicht aufzulösen. Als letztes wird die Ausgabe eingeschränkt. Viele Ansätze erwarten, dass der Anwender, durch Wahl des korrekten Quelltextes, aushilft. Sie präsentieren daher eine Reihe von wahrscheinlich Antworten, wobei die wahrscheinlichste selten die Korrekte ist. Aber auch bei der Synthese des Quelltextes gibt es Probleme. Kontrollstrukturen wie Bedingungen und Wiederholungen fehlen oftmals. Generell scheint die Ausgabe sich eher darauf zu beschränken einzelne Befehle zu erzeugen, anstatt längere zusammenhängende Skripte zu erzeugen. Trotz verschiedener Ansätze existiert noch keine allgemeingültige, domänenunabhängige Lösung zur Übersetzung von natürlicher Sprache in Quelltext.

Abseits der Übersetzung der natürlichen Sprache in Quelltext wird gezeigt, dass einige Ansätze fehlerhaften Quelltext über Methoden der Programmsynthese und -Analyse korrigieren. Damit wird eine höhere Korrektheit erreicht. Der Einsatz solcher Methoden erscheint daher vorteilhaft.

Im Bereich der Programmtransformation scheinen bereits einige Methoden zu existieren, mit denen sich ein erzeugter abstrakter Syntaxbaum in Quelltext übersetzen lässt. Dies findet in der Regel durch das Besucher-Muster statt. Es ist daher sinnvoll einen abstrakten Syntaxbaum als Zwischenrepräsentation zu verwenden. Der Fokus der Forschung in diesem Bereich liegt dabei auf der Erleichterung der softwaretechnischen Erweiterbarkeit der Methoden.

5 Analyse und Entwurf

In Kapitel 4 wurden verschiedene Arbeiten zum Thema „Programmieren in natürlicher Sprache“ vorgestellt. Hierbei wurde abschließend in Abschnitt 4.3 festgestellt, dass eine allgemeingültige Lösung zur Programmierung in natürlicher Sprache nicht existiert. Viele der vorgestellten Ansätze schränken den möglichen Lösungsraum durch Vorgaben in Wortwahl, Anwendungsumgebung, Komplexität und Ausgabe ein. Erst durch diese Einschränkungen werden akzeptable Lösungen erreicht. Dies steht aber der Allgemeingültigkeit der Lösung entgegen.

In dieser Arbeit soll eine „Zielsystemunabhängige Quelltextsynthese aus natürlicher Sprache“ erreicht werden. Dazu wird die in Kapitel 3 beschriebene Rahmenarchitektur PARSE verwendet. Diese stellt eine Vielzahl an sprachverarbeitenden Werkzeugen zur Verfügung, die eine Eingabe in natürlicher, englischer Sprache auswerten. Als zentrale Datenstruktur in PARSE dient ein Graph der sämtliche Informationen speichert. Die Architektur von PARSE besteht aus drei Phasen: Ein Vorverarbeitungsfließband bereitet die natürlichsprachliche Eingabe auf und erzeugt den zentralen Graphen. In der Hauptverarbeitung wird dieser Graph durch Agenten analysiert und um deren Ergebnisse erweitert. Der Graph wird nach Abschluss der Hauptverarbeitung im Nachverarbeitungsfließband zu Quelltext übersetzt. PARSE extrahiert somit bereits ausreichend Informationen zu Sprachanalyse, Anwendungskontext und Programmierstrukturen aus der Eingabe. Der Fokus dieser Arbeit liegt deshalb auf der Erzeugung von Quelltext aus den PARSE zur Verfügung gestellten Informationen.

Dieses Kapitel behandelt eine Analyse der Problemstellung und den Entwurf von Ansätzen und Verfahren, die dafür eine Lösung darstellen. Zu Beginn des Kapitels findet in Abschnitt 5.2 eine Voranalyse der Problemstellung statt. Hierfür wurden zwei Werkzeuge erstellt, die kurz vorgestellt werden. Die Teilprobleme aus denen sich die Problemstellung bildet, werden in den folgenden Abschnitten genauer betrachtet. So wird in Abschnitt 5.3 die Repräsentation der Anwendung und Umgebung durch eine Ontologie kurz erläutert. In Abschnitt 5.4 wird die Übersetzung der Eingabe in eine geeignete Zwischenrepräsentation erklärt. Die darauf folgende zielsystemunabhängige Synthese des Quelltextes aus der Zwischenrepräsentation wird in Abschnitt 5.5 beschrieben. Hierbei werden jeweils mehrere Verfahren entworfen und im Detail vorgestellt. In Abschnitt 5.6 wird diskutiert, weshalb maschinelle Lernverfahren als alternative Lösung nicht geeignet sind.

5.1 Problemstellung und Teilprobleme

Die Aufgabenstellung der „Zielsystemunabhängigen Quelltextsynthese aus natürlicher Sprache“ lässt sich in folgende Teilprobleme zerlegen: Zuerst muss für das Zielsystem eine Ontologie erstellt werden. Sie enthält Informationen über die Umgebung und Schnittstelle des Systems. Darauf folgt das Abbilden der Befehlsbeschreibungen auf entsprechende Methoden der Schnittstelle über die Ontologie. Die Hauptaufgabe besteht darin den PARSE-Graphen auf Quelltext zu übersetzen. Über einen abstrakten Syntaxbaum als Zwischenrepräsentation kann die Aufgabe in zwei Teilschritte zerlegt werden. Im ersten Schritt wird der abstrakte Syntaxbaum aus dem PARSE-Graphen aufgebaut. Anschließend übersetzt man den erstellten Syntaxbaum in Quelltext. Dieser Quelltext muss zum Schluss dem Zielsystem zur Weiterverarbeitung übergeben werden. Das System kann den Quelltext dann entweder ausführen oder als neue Methode implementieren.

Eine softwaretechnische Herausforderung an den Entwurf ist die Variabilität der Eingabe. Der in PARSE erstellte Graph kann nachträglich, durch die Implementierung weiterer Agenten, um neue Informationen ergänzt werden. Da das in dieser Arbeit entwickelte Werkzeug Teil der Nachverarbeitung ist, ist es wichtig diese neuen Informationen leicht einpflegen zu können. Beim Entwurf sollte außerdem beachtet werden, dass sich Zielsysteme in bestimmten Eigenschaften unterscheiden können. Eine solche Eigenschaft wäre zum Beispiel die Programmiersprache.

5.2 Voranalyse

Um die Problemstellung überblicken zu können, wurde zu Beginn eine Voranalyse durchgeführt. Die Voranalyse hatte mehrere Ziele. Die übersichtliche Darstellung des PARSE-Graphen war das erste Ziel. Hierfür wurde ein Agent erstellt, der den Graphen in einer graphischen Form ausgibt. Dies wurde durch die Verwendung von Latex und der TikZ-Graphenbibliothek erreicht. Durch die graphische Darstellung war es möglich den Umfang der bereitgestellten Informationen und auch Muster leicht zu erkennen.

Durch einen in der Voranalyse entwickelten Prototypen konnte der Ablauf zur Erzeugung von Quelltext aus dem PARSE-Graphen erkundet werden. Dabei wurden einige Herausforderungen mit möglichen Lösungen festgestellt. Das direkte Übersetzen von PARSE-Graph in Quelltext führt dazu, dass viele Schritte für verschiedene Zielsysteme wiederholt werden müssen. Daher sollte mindestens eine Trennung zwischen der Aufbereitung des PARSE-Graphen und der Erzeugung von Quelltext stattfinden. So werden die einzelne Aufbereitungsschritte wiederverwendbar und die Quelltexterzeugung vom PARSE-Graphen separiert. Daher ist eine wichtige Herausforderung der Arbeit eine geeignete Zwischenrepräsentation zwischen Graph und Quelltext zu finden. Der abstrakte Syntaxbaum wurde als zweckmäßige Form bestimmt. Syntaxbäume stellen eine Ebene der Abstraktion zu Quelltext dar und können direkt im PARSE-Graphen iterativ aufgebaut werden. Damit wird die Aufgabe der Quelltextübersetzung in zwei Schritte zerteilt. Der abstrakte Syntaxbaum soll im ersten Schritt aus dem PARSE-Graphen zusammengesetzt werden. Der Graph ist bereits in vorherigen Arbeiten durch Sprachverarbeitungswerkzeuge mit vielen Informationen angereichert worden. Eine Mustererkennung über die auf dem Eingabegraphen enthaltenen Informationen könnte dazu verwendet werden, um einen Syntaxbaum aufzubauen. Beim Aufbau des Syntaxbaums sollten verschiedene Punkte beachtet werden. Zum einen muss die semantische Korrektheit, d.h. der erzeugte Quelltext entspricht der Befehlsbeschreibung, garantiert werden. Zum anderen ist es nötig sicherzustellen, dass der Syntaxbaum in lauffähigen Quelltext übersetzt werden kann. Der abstrakte Syntaxbaum sollte daher für die Übersetzung vollständig sein. Das Modul zur Quelltexterzeugung sollte nicht mehr auf den Graphen zugreifen müssen. Für die Quelltexterzeugung wurden Ansätze aus der

Programmtransformation und des Übersetzerbaus betrachtet. Hierbei wird ebenfalls als Zwischenrepräsentation ein abstrakter Syntaxbaum verwendet. Dieser Baum wird in der Regel anschließend mithilfe des Besucher-Musters abgelaufen und in Quelltext übersetzt. Der erzeugte Quelltext kann dann über ein Schablonen-System an das Zielsystem übergeben werden. Dieser Ansatz wurde auch im Prototypen erfolgreich verwendet.

5.3 Ontologie

Das Zielsystem wird über eine vorher definierte Schnittstelle angesprochen. Eine Ontologie repräsentiert in PARSE die Schnittstelle und Umgebung des Zielsystems. Über Konzepte der Ontologie ist es möglich Akteuren und Aktionen ihre entsprechenden Entitäten und Methoden der Schnittstelle zuzuweisen. Durch einen Austausch der Ontologie lässt sich PASRSE leicht um weitere Zielsysteme erweitern. Eine Möglichkeit Ontologien für Zielsysteme zu erstellen, ist es sie vollständig von Hand zu modellieren. Da dies bereits bei Systemen mittlerer Größe bereits mit hohem Aufwand verbunden ist, bietet sich auch ein halb-automatisches Verfahren an [LWT]. Dabei wird zuerst die Umgebung des Zielsystems manuell in die Ontologie implementiert. Die Schnittstelle des Zielsystems kann dann automatisch extrahiert und in die Ontologie eingefügt werden. Dazu wird die Schnittstelle mit einem Zerteiler zerlegt und die Ontologie um gefundene Methoden und Klassen erweitert. Passende Synonyme werden automatisch aus WordNet extrahiert. In dieser Arbeit wird das Teilproblem der Erzeugung der Ontologie umgangen, da die betrachteten Zielsysteme bereits vollständig erstellte Ontologien besitzen.

Das zweite Teilproblem bezüglich der Ontologie ist die Abbildung, der in der Eingabe erkannten Aktionen, auf Methoden der Schnittstelle. Das Zuweisen passender Methoden zu Aktionsbeschreibungen wird in der Masterarbeit von Steurer [Ste] gelöst. Teilsätze sind daher bereits im PARSE-Graphen mit systemunabhängigen Methodensignaturen annotiert. Diese Methodensignaturen enthalten alle Informationen, die nötig sind, um die echte Schnittstelle des Zielsystems anzusprechen. Sie müssen aber noch in die entsprechende Form für einen korrekten Zugriff auf die Schnittstelle transferiert werden.

5.4 Synthese des abstrakten Syntaxbaumes

In diesem Abschnitt soll das Vorgehen zum Erzeugen des Syntaxbaumes vorgestellt werden. Zuerst soll aber in Definition 5.1 der Begriff der *Struktur* eingeführt werden.

Es gibt drei grundsätzliche Ansätze zur Erzeugung des abstrakten Syntaxbaums: Das erste Verfahren wäre ein „Top-Down"-Ansatz und ähnelt einer Suche. Hierbei wird der Syntaxbaum vom Wurzelknoten aus hierarchisch bis zu den Blättern aufgebaut. Dabei werden für jeden Knoten, der kein Blatt darstellt, die passenden Strukturen anhand Muster im Eingabegraphen gesucht, übersetzt und mit dem Knoten verbunden. Dies wird rekursiv wiederholt, bis der gesamte Syntaxbaum aufgebaut wurde. Da der Baum von der Wurzel bis zu den Blättern aufgebaut wird, entsteht ein einziger Syntaxbaum. Wird das Verfahren ordnungsgemäß beendet, so ist die Lauffähigkeit des Syntaxbaums garantiert. Bei der Anbindung einer neuen Struktur wird sichergestellt, dass diese Struktur passt. So wird bei der Anbindung der Strukturen direkt eine implizite statische Analyse der Korrektheit der Verbindung durchgeführt. Zeitgleich ist es aber möglich, dass ein lauffähiger, aber unvollständiger Syntaxbaum erzeugt wird. Wird beispielsweise ein Blatt verfrüht angebunden, findet keine weitere Suche in die Richtung statt. Damit könnten Teile der Anweisungsfolgen verloren gehen. Dies wäre aber wohl durch eine Abdeckungsheuristik feststellbar und im Dialog überprüfbar, sofern die Fehlerstelle identifizierbar ist. Problematisch ist auch die zeitgleiche Verbindung während der Mustersuche nach Strukturen. Werden neue Mustererkenner implementiert, so müssen alle anderen Mustererkenner an sie angepasst werden. Dies erschwert die Erweiterbarkeit.

Definition 5.1: Strukturen und Teilstrukturen

Strukturen sind wiederkehrende Grundbausteine aus welchen ein abstrakter Syntaxbaum aufgebaut ist. Sie lassen sich in äquivalente Quelltextbausteine übersetzen und erweitern damit die Komplexität eines Programmes. Komplexe Programmabläufe sind bereits mit wenigen Strukturen darstellbar, da sie sich rekursiv verwenden lassen. Kontrollstrukturen sind ein Beispiel für Strukturen.

Teilstrukturen

Die **Teilstruktur** ist eine spezielle Form der Struktur. Komplexere Strukturen unterteilen sich in mehrere, einfachere Teilstrukturen. Teilstrukturen existieren daher nur als Teil übergeordneter Strukturen. Teilstrukturen erlauben eine separate Betrachtung verschiedener Aspekte der Struktur und somit auch eine saubere Separierung der Kinder.

Basisstruktur

Die **Basisstruktur** ist eine besondere Form der Teilstruktur. Sie ist das verbindende Element zwischen den Teilstrukturen. Die Basisstruktur dient somit auch als Verbindung zwischen den Teilstrukturen und der Elterstruktur. Weiterhin speichert die Basisstruktur Informationen über die gesamte Struktur.

Das zweite Verfahren wäre ein „Bottom-Up“-Ansatz und besteht aus zwei Phasen. In der ersten Phase, sollen im Eingabegraphen Muster gesucht werden und in Strukturen des Syntaxbaums übersetzt werden. Die Verbindung der Strukturen zu einem vollständigen abstrakten Syntaxbaum würde in der zweiten Phase stattfinden. Durch die Trennung der beiden Phasen wird die Erweiterbarkeit erleichtert. Die Extraktion der Strukturen aus dem Eingabegraphen ist unabhängig von der Verbindung der Strukturen. Dadurch wäre garantiert, dass alle auffindbaren Strukturen des Eingabegraphen auch gefunden werden. Beim Hinzufügen neuer Mustererkenner kann sich auf das Finden der Strukturen konzentriert werden. Da der Aufbau des Syntaxbaums durch die Verbindung der Strukturen ebenfalls separat ist, kann das Problem der Verbindung isoliert betrachtet werden. Dies erlaubt auch eine einfache Implementierung verschiedener Verfahren zur Verbindung der Strukturen und somit alternative Lösungen für die gleichen Strukturen. So kann auch entschieden werden, ob eine Analyse des Syntaxbaums während der Verbindungsphase oder in einer dritten Phase stattfindet. Ein Nachteil hingegen wäre, dass die Lauffähigkeit des Quelltextes nicht garantiert ist. Werden nicht alle Strukturen verbunden, so ist dies ein starker Hinweis darauf, dass die aktuelle Lösung fehlerhaft ist. Da der gesamte Graph untersucht wird, existieren Teillösungen, die einen Dialog mit dem Nutzer erleichtern können. So lassen sich Lücken zwischen Teillösungen leichter erkennen. Über einen Dialog könnte man den Nutzer bitten, zu spezifizieren, wie die einzelnen Teillösungen zusammenzufügen sind.

Der dritte Ansatz basiert darauf, dass Eingabe und Ausgabe beides Graphen sind. Die Idee wäre es den Eingabegraphen über Graphersetzungsregeln[1] in einen Syntaxbaum zu transformieren. Durch diese Regeln würden Strukturen aus dem Graphen extrahiert und verbunden werden. Theoretisch wäre es möglich die anderen beiden Ansätze durch Regeln zu modellieren. Dies könnte es erlauben je nach Situation den geeignetsten Ansatz zu wählen. Dieser Ansatz hat aber den Nachteil, dass er einen Graphen als Eingabe benötigt und daher weniger transferierbar wäre. Weiterhin ist es unwahrscheinlich, dass eine direkte Übersetzung vom Graphen in den Syntaxbaum möglich ist. Daher würde dieser Ansatz einen großen Regelsatz benötigen, um alle Zwischenschritte zu modellieren. Die vollständi-

[1] Verallgemeinerung der Ableitungsregeln für formale Sprachen

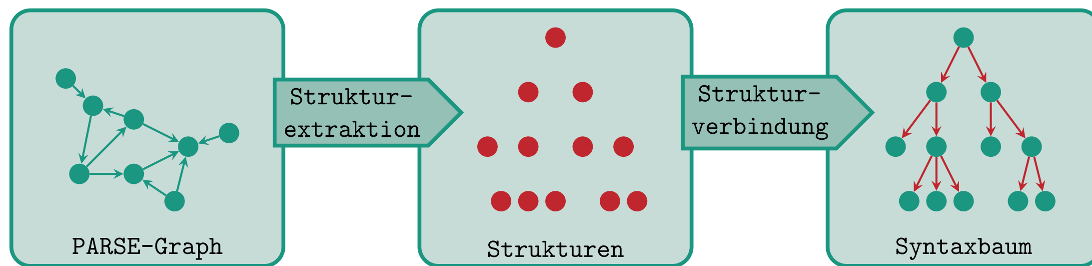


Abbildung 5.1: Syntaxbaumerzeugung

ge Synthese des abstrakten Syntaxbaum durch Graphersetzungsregeln würde aus diesem Grund zu hohen Aufwand führen. Durch Graphersetzungsregeln könnte aber der Eingabegraph vorverarbeitet werden, um so den Graph für die anderen Ansätze zu vereinfachen.

Das in dieser Arbeit zum Aufbau des Syntaxbaumes ausgewählte Verfahren entspricht dem beschriebenen „Bottom-up“-Verfahren. Es besteht aus zwei Verarbeitungsschritten und ist in Abbildung 5.1 dargestellt. Gewählt wurde es, da bekannt ist, dass in Zukunft PARSE um neue Agenten erweitert wird. Die Ergebnisse der neuen Agenten können sich auf alle Aspekte der Syntaxbaumsynthese auswirken. Daher spielt die Erweiterbarkeit des Ansatzes eine große Rolle. Durch die isolierten Verarbeitungsschritte ist eine einfache Erweiterung der Werkzeuge möglich. So werden im ersten Schritt Mustererkenner verwendet, um Strukturen aus dem PARSE-Graphen zu extrahieren. Die gefundenen Strukturen werden in einem zweiten Schritt dann verbunden, um einen abstrakten Syntaxbaum aufzubauen. Während beim beschriebenen „Bottom-up“-Ansatz möglichst viele Muster erkannt und in Strukturen übersetzt werden sollen, wird bei diesem Verfahren die Anzahl an Mustererkenner minimiert. Nur die zum Aufbau des abstrakten Syntaxbaums benötigten Mustererkenner werden verwendet. Daher liegt der Fokus des Verfahrens auf der Verbindung der erkannten Strukturen. Die Verbindung soll möglichst unabhängig von den zu verbindenden Strukturen sein. Der so erzeugte Syntaxbaum wird dann bei der Quelltexterzeugung weiterverarbeitet. Zum Schluss ist eine weitere Eigenschaft des Ansatzes vorteilhaft. Wird angenommen, dass die Ausführung der vom Nutzer gegebenen Anweisungen notwendig ist, so ist ein syntaktisch fehlerhafter Syntaxbaum eine gute Möglichkeit Fehler zu erkennen. Der Nachteil dieses Ansatzes ermöglicht dadurch eine vereinfachte Fehleranalyse.

5.4.1 Mustererkennung

Unabhängig vom gewählten Ansatzes müssen aus dem PARSE-Graphen Strukturen des Syntaxbaumes extrahiert werden. Die Strukturen können mit Verfahren aus der *Mustererkennung* (siehe Definition 5.2) gefunden werden. Dabei kann zwischen verschiedenen Arten von *Mustererkennern* unterschieden werden. So gibt es Mustererkenner, die unabhängig von anderen *Mustern* auf dem Eingabegraphen arbeiten können. Daneben gibt es Mustererkenner, die Ergebnisse aus anderen Mustererkennern benötigen. Es ist daher denkbar, dass eine komplexe Hierarchie von Mustererkennern besteht, die jeweils aufeinander aufbauen. Dies könnte durch eine baumähnliche Struktur repräsentiert werden. In den Blättern des Baumes wären die Mustererkenner, die unabhängig von anderen Erkennern arbeiten können. Die Abhängigkeit würde durch die Elterbeziehung repräsentiert. Mehrere Eltern können das gleiche Kind besitzen und damit vom selben Mustererkenner abhängen. Mit einer einzelnen Ausführung des Kindes werden daher die benötigten Strukturen für alle Eltern erzeugt. Wurden alle Kinder ausgeführt, können die Eltern rekursiv aufsteigend ausgeführt werden. Es besteht die Möglichkeit, dass mehrere Hierarchien bestehen, die voneinander unabhängig sind und damit einen Hierarchiewald beschreiben. Eine gemeinsame Wurzel wäre nicht nötig, könnte aber die Handhabung erleichtern. So wäre mit

Definition 5.2: Mustererkennung

Die **Mustererkennung** (engl. pattern recognition) ist eine musterbasierte Suche auf einem Datensatz.

Muster

Ein **Muster** (engl. pattern) ist eine wiederkehrende Regelmäßigkeit in einem Datensatz, wie beispielsweise dem PARSE-Graphen. Ein Muster kann beispielsweise eine Struktur des Syntaxbaumes, Attribute von Strukturen oder andere Änderungen des Parse-Graphen impliziert.

Mustererkenner

Ein **Mustererkenner** (engl. pattern recognizer) ist ein Werkzeug, welches in einem gegebenen Datensatz ein spezifisches Muster erkennt. Der Mustererkenner führt für jedes erkannte Muster eine bestimmte Aktion durch. In dieser Arbeit werden Mustererkenner genutzt, um Strukturen des Syntaxbaums zu erzeugen.

Musterbereich

Der **Musterbereich** ist die Menge der Token des Eingabesatzes, deren Merkmale ein bestimmtes Muster implizieren.
Da eine Struktur aus mehreren *Teilstrukturen* bestehen kann, können diese Teilstrukturen ebenfalls eigene Musterbereiche besitzen. Diese Musterbereiche der Teilstrukturen werden **Teilmusterbereiche** genannt. Die Teilmusterbereiche sind immer eine Untermenge ihres übergeordneten Musterbereichs.

Beispiel 5.4.1: Musterbereiche einer Verzweigungsstruktur

„*If it's raining* *please close the window* *else open the door*"

Da die Verzweigungsstruktur aus drei Teilstrukturen besteht, implizieren die Teilsätze jeweils eine Teilstruktur. Sie bilden damit drei disjunkte Teilmusterbereiche. In Grün ist der Teilmusterbereich der Bedingung markiert. Rot und Blau sind jeweils die Teilsmusterbereiche der Dann- und Sonst-Teilstruktur. Der Musterbereich der Verzweigungstruktur ist der gesamte orange hinterlegte Satz.

der Ausführung der gemeinsamen Wurzel garantiert, dass alle Mustererkenner ausgeführt wurden.

In der Hauptverarbeitung von PARSE extrahieren Agenten unabhängig voneinander Informationen. Der PARSE-Graph wird von den Agenten um die extrahierten Informationen erweitert. Diese Extraktion von Informationen stellt eine Abstraktion von der Ebene der natürlichen Sprache dar. Für die Mustersuche können die extrahierten Informationen als Grundlage verwendet werden. Da die Agenten unabhängig voneinander arbeiten, sind auch ihre Ergebnisse separiert. Die Mustererkenner können dadurch auf den Ergebnissen ebenfalls unabhängig voneinander arbeiten. Werden die Agenten als Mustererkenner interpretiert, so stellen sie die niedrigste Stufe der Hierarchie dar. Nur Mustererkenner, die explizit das Ziel haben den bereits erstellten Syntaxbaum zu bearbeiten, müssen auf das Ergebnis anderer Erkenner warten. Dies erlaubt die Höhe der Hierarchie der Mustererkenner zu minimieren.

5.4.2 Strukturen des abstrakten Syntaxbaums

Im Allgemeinen nutzen Übersetzer grammatikalische Regeln, um aus den Ausdrücken des Quelltextes den entsprechenden abstrakten Syntaxbaum aufzubauen. Moderne Sprachen wie Java, C++ und Python basieren auf einer Vielzahl unterschiedlicher Regeln. Viele dieser Regeln werden nur für spezielle Ausdrücke benötigt. Eine solche Komplexität ist bei der Programmierung in natürlicher Sprache nicht sinnvoll. Diese findet in der Regel durch Anwender statt, welche kaum Programmierkenntnisse besitzen. Aus natürlicher Sprache erzeugter Quelltext ist daher simpler als durch erfahrene Programmierer geschriebener Quelltext. Dem generierten Quelltext entsprechende abstrakte Syntaxbäume, lassen sich deswegen durch einen verringerten Regelsatz beschreiben.

Die Reduzierung des Regelsatzes führt dazu, dass *Strukturen* (siehe Definition 5.1) der Syntaxbäume nur auf eine reduzierte Menge von Ausdrücken abgebildet werden müssen. Es ist dadurch möglich im PARSE-Graphen nur nach den Mustern zu suchen, die sich auf diese Strukturen abbilden lassen. Da sich die Strukturen rekursiv verwenden lassen, sind sie ausreichend, um komplexe Programmabläufe darzustellen. Zum Aufbau des abstrakten Syntaxbaumes sind daher nur die in Definition 5.3 aufgezählten Strukturen nötig.

Definition 5.3: Syntaxbaum-Strukturen

Strukturen aus denen ein abstrakter Syntaxbaum aufgebaut werden kann:

Aktionen

Aktionen sind Ausdrücke, die von der Schnittstelle bereitgestellten Methoden entsprechen. Neue Methoden sollen gelernt und dann verwendet werden können.

Bedingungen

Bedingungen sind Ausdrücke, welche sich zu einem booleschen Ausdruck auswerten. In PARSE werden sie ebenfalls durch Methoden repräsentiert.

Sequenz

Eine **Sequenz** ist die Zuordnung einer Menge von Ausdrücken in eine feste Reihenfolge. Sie entspricht damit einem Anweisungsblock und bildet die Grundlage des Programmablaufes.

Kontrollstrukturen

Kontrollstrukturen steuern den Programmablauf:

- **Schleifen** erlauben die wiederholte Ausführung der selben Sequenzen.
- **Verzweigungen** erlauben die bedingte Ausführung unterschiedlicher Sequenzen.

Parallelität

Parallelität ist die zeitgleiche Ausführung mehrerer Sequenzen. In dieser Arbeit soll nur eine einfache Form der Parallelität betrachtet werden. Abseits des synchronisierten Starts und Endes findet keine weitere Synchronisierung oder Kommunikation zwischen den Prozessen statt.

Hierbei sei auch erwähnt, dass die in Definition 5.1 beschriebenen *Teilstrukturen* eine Vereinfachung gegenüber abstrakten Syntaxbäumen komplexer Programmiersprachen darstellen. Deren Teilstrukturen sind meist generischer und ermöglichen auch alternative Strukturen. Da in dieser Arbeit die Menge an verwendeten Strukturen reduziert wird, ist es

möglich auch die Menge der möglichen Teilstrukturen zu vereinfachen. Aus diesem Grund beschränkt sich die Menge der Teilstrukturen nur auf Bedingungen und Sequenzen. Weiterhin sind die Teilstrukturen einer Struktur immer identisch. Kontrollstrukturen sind die einzigen Strukturen, die aus Teilstrukturen bestehen.

Definition 5.4: Strukturmengen

Strukturmengen sind Mengen von Strukturen, aus denen sich ein abstrakter Syntaxbaum aufbauen lässt. Die Strukturmenge hat Einfluss auf die Komplexität des Syntaxbaumes und damit auf den äquivalenten Quelltext.

Quelltext-Strukturmenge

Die **Quelltext-Strukturmenge** beschreibt die Menge an Strukturen, aus denen ein abstrakter Syntaxbaum bestehen muss, um ihn in äquivalenten Quelltext bestimmter Komplexität übersetzen zu können. Sie ist damit eine untere Schranke für die Komplexität des Syntaxbaumes.

Korpus-Strukturmenge

Die **Korpus-Strukturmenge** entspricht den Strukturen, die im natürlich sprachlichen Korpus beschrieben werden. Ein Werkzeug übersetzt sie in äquivalente Strukturen des abstrakten Syntaxbaum. Diese Menge stellt so die höchstmögliche Komplexität dar, die sich durch diesen Korpus erreichen lässt.

Werkzeug-Strukturmenge

Die **Werkzeug-Strukturmenge** entspricht den Strukturen, die von einem Werkzeug verarbeitet werden können. Es filtert die Korpus-Strukturmenge und stellt die obere Schranke für die Komplexität des Syntaxbaums dar.

In Definition 5.4 wird der Begriff der *Strukturmenge* eingeführt. Die Betrachtung der Menge an Syntaxbaum-Strukturen, die zur Quelltextsynthese nötig sind, führt zur *Quelltext-Strukturmenge.* Sie beschreibt die Menge an Strukturen, die nötig sind um Quelltext entsprechender Komplexität zu generieren. Hierbei sollte beachtet werden, dass zur Beschreibung unterschiedlich komplexer Programme unterschiedliche Strukturmengen nötig sind. In den Definitionen 5.5, 5.6, 5.7, 5.8 und 5.9 werden die verschiedenen Strukturmengen erläutert. In dieser Arbeit soll mindestens die Komplexität der *gesteuerten Anweisungssequenz* erreicht werden. Eine Erweiterung dieser um einfache *Parallelität* soll ebenfalls analysiert werden.

Definition 5.5: Einzelanweisungen

Die Menge der **Einzelanweisungen** bestehen nur aus der Aktions-Struktur. Dies ermöglicht nur die Ausführung eines einzelnen Befehls und bildet den Kern aller weiteren Mengen. Sie entspricht damit einfachsten Befehlsinterpretierer.

Beispiel 5.4.1:

- „Close the window.“
- „Bring me the orange juice.“
- „Walk forward until you see the color red.“

```
close(window)
bring(SPEAKER, orangeJuice)
walkForwardUntilColor(Color.RED)
```

Definition 5.6: Anweisungssequenzen

Die Menge der **Anweisungssequenzen** entsteht bei einer Erweiterung von Einzelanweisungen um Sequenz-Strukturen. Mit dieser Menge ist es möglich Anwendungen mit sequentiellem Programmablauf darzustellen.

Beispiel 5.4.2:

- „Close the window and then clean the table.“

```
close(window)
clean(table)
```

- „First close the fridge. Then grab the juice and bring it to me. After that clean the table.“

```
close(fridge)
grab(juice)
bring(juice,SPEAKER)
clean(table)
```

- „Walk forward until you see red then sing.“

```
walkForwardUntilColor(Color.RED)
sing()
```

Definition 5.7: Bedingte Einzelanweisung

Die Menge der **bedingten Einzelanweisungen** entsteht durch eine Verknüpfung der Einzelanweisungen mit Kontrollstrukturen und Bedingungen. In dieser Menge finden sich beispielsweise einzelne Methoden, die durch eine Bedingung vor der Ausführung geschützt sind. Auch sind hier Methoden zu finden, die wiederholt ausgeführt werden, bis eine Abbruchbedingung zutrifft.

Beispiel 5.4.3:

- „If it rains, close the window.“

```
if(weather.isRaining()):
    close(window)
```

- „Knock on the door twice.“

```
for i in range(2):
    hit(door)
```

- „Secure the ladder while I'm on it.“

```
while (SPEAKER.isOn(ladder)):
    grab(ladder)
```

PARSE nutzt Agenten, um Strukturen zu finden und den PARSE-Graphen zu erweitern. Die Menge dieser Strukturen stellt die *Werkzeug-Strukturmenge* für PARSE dar und beschreibt alle Strukturen, die durch PARSE erkannt werden können. Strukturen der *Korpus-Strukturmenge*, die nicht in der Werkzeug-Strukturmenge sind, werden vom Werkzeug nicht verarbeitet. Die Schnittmenge zwischen Werkzeug- und *Quelltext-Strukturmenge* definiert die Menge an Strukturen, aus denen der abstrakte Syntaxbaum aufgebaut wird. Die beiden Strukturmengen stellen dadurch die obere Schranke für die Komplexität des Quelltextes dar. Um die benötigte Komplexität der Quelltext-Strukturmenge zu erreichen, muss sie eine Teilmenge der Werkzeug-Strukturmenge sein. Entspricht die Werkzeug-Strukturmenge nicht mindestens der Quelltext-Strukturmenge, so ist es nötig PARSE, um Agenten zu erweitern. Aus Kapitel 3 ist ersichtlich, dass die Werkzeug-Strukturmenge von PARSE die benötigte Komplexität der „gesteuerten Anweisungssequenz“ erreicht. Daher ist es ausreichend, die Quelltext-Strukturmenge zu verwenden.

Definition 5.8: Gesteuerte Anweisungssequenz

Die **gesteuerte Anweisungssequenz** entsteht durch die Vereinigung der Strukturen der Anweisungssequenzen mit denen der bedingten Einzelanweisung. Diese Menge ermöglicht den Aufbau komplexer Programmabläufe mit der Steuerung von Sequenzen durch Kontrollstrukturen. Sie entspricht damit der imperativen Programmierung.

Beispiel 5.4.4:

- „Go to the kitchen. If it rains, close the window, else clean the table.“

```
moveTo(kitchen)
if(weather.isRaining()):
    close(window)
else:
    clean(table)
```

- „If it rains play the jazz radio channel. Else if it snows then make me hot chocolate and cookies.“

```
if(weather.isRaining()):
    playRadio(jazz)
else:
    if(weather.isSnowing()):
        make(hotChocolate)
        bake(cookies)
```

- „Swirl the dough 20 times and put it on the table. While the cookies are in the oven, check the cookies and if they're done, take them out of the oven.“

```
for i in range(20):
    swirl(dough)
putOn(dough, table)
while(oven.contains(cookies)):
    if(cookies.areFinished()):
        takeOut(Cookies)
```

5.4.2.1 Aktionen und Bedingungen

Die Schnittstelle einer Anwendung stellt eine Menge von Methoden zur Verfügung. In PARSE wird die Schnittstelle der programmierbaren Anwendung durch eine Ontologie repräsentiert. In dieser Ontologie werden die Methoden der Schnittstelle offengelegt. Zwischen den Methoden der Schnittstelle und der Aktions-Struktur des Syntaxbaumes besteht ein direkter Zusammenhang. Im abstrakten Syntaxbaum sollen Aktions-Strukturen auf die Methoden der Schnittstelle abgebildet werden. Die Aktions-Strukturen bestehen aus einem Objekt, dem Namen der aufzurufenden Methode und einer variablen Anzahl an Parametern. Es ist möglich die Parameter, Namen und Objekte als eigene Teilstrukturen zu entwerfen. Da diese aber mit Hilfe der Ontologie erzeugt werden, reicht es aus sie als Attribute der Aktions-Struktur zu speichern. Da die Aktions-Struktur keine Kinder besitzt, stellt sie immer die Blätter des abstrakten Syntaxbaums dar.

Im PARSE-Graphen wurden die Informationen zur Methodensignatur aus Teilsätzen bereits in einer vorherigen Arbeit extrahiert. Dadurch fungiert die Mustersuche eher als Übertragung der gefundenen Methodensignatur in den Syntaxbaum. Die Methodensignatur ist im PARSE-Graphen Teil einer Baumstruktur. Ein Knoten vom Typ `commandMapper` dient als Wurzelknoten und umrahmt den gesamten Text. Er zeigt zusätzlich an, ob die Befehlsbeschreibung direkt ausgeführt werden soll oder eine Lehrsequenz darstellt. Bei einer Lehrsequenz soll die Anwendung um eine weitere Methode erweitert werden. Da es so nötig ist, Informationen über die neue Methode zur Verfügung zu stellen, teilt sich der Baum in zwei Bereiche auf. Die Deklaration markiert den Teilsatz, welcher der Lehrsequenz ihren Namen und Argumente gibt. Die Beschreibung sammelt die einzelnen Methodensignaturen, die der Befehlsbeschreibung entsprechen. Jeder Teilsatz erhält eine Nummer und einen Knoten vom Typ `functionCall`. Die Kinder dieses Knotens sind mögliche Metho-

Definition 5.9: Parallelität

Paralleliät entsteht bei einer Erweiterung der gesteuerten Anweisungssequenz um Paralleliätsstrukturen. Dadurch ist es möglich die Komplexität weiter zu erhöhen und nebenläufige Programme zu beschreiben.

Beispiel 5.4.5:

- „Sing and dance at the same time."
- „While bringing me juice from the kitchen, grab me also a piece of cake."
- „Play rock music while you cut and mix the salat."

```
parallel:
    section:
        sing()
    section:
        dance()
```

```
parallel:
    section:
        bring(SPEAKER,juice)
    section:
        bring(SPEAKER,cake)
```

```
parallel:
    section:
        playRadio(rock)
    section:
        cut(salad)
        mix(salad)
```

a) `Objekt.methode(Parameter)`

b) `Objekt.bedingung(Parameter)`

Abbildung 5.2: (a) Aktions- und (b) Bedingungsstruktur als Symbol dargestellt

densignaturen für diesen Teilsatz. Sie bestehen aus ihrer Bewertung und einem Verweis auf den Namen der Methode der Ontologie (`onto#grab`). Als Kinder besitzen sie die Parameter, mit denen die Methode aufgerufen werden soll. Die Parameter bestehen aus der Entität (`onto#cup`) und dem entsprechenden Argument (`onto#grab.what`) der Methode. Bei der Transferierung in die Aktions-Struktur muss die Entität als Parameter aufgelöst und eingesetzt werden.

Als Bedingung wird eine Methode bezeichnet, die einen booleschen Wert zurückliefert. Erst durch die Kombination mit Kontrollstrukturen gewinnen sie an Nutzen. Sie ermöglichen die Steuerung des Programmablaufes. In der Ontologie werden mögliche Bedingungsabfragen der Schnittstelle als Methoden zur Verfügung gestellt. Wird der Raum der möglichen Bedingungsabfragen auf die durch die Schnittstelle zur Verfügung gestellten Methoden eingeschränkt, so lassen sich Bedingungen als Methoden betrachten. Diese Gleichsetzung erlaubt die Betrachtung von Bedingungen als Teilmenge der Methoden. Sie können damit ebenfalls über die Methodensignaturen im PARSE-Graphen gefunden und in Aktions-Strukturen überführt werden. Um bei einer späteren Analyse fehlerhafte Verbindungen finden zu können, wäre es aber weiterhin notwendig Bedingungen von Methoden unterscheiden zu können. Die Identifizierung aufgrund sprachlicher Eigenheiten erweist sich als schwierig. Die Vermutung, dass Bedingungen in Form einer Frage gegeben werden, ist lässt sich nicht bestätigen. So kann ein Nutzer Sätze wie „*Can you bring me the medicine?*" for-

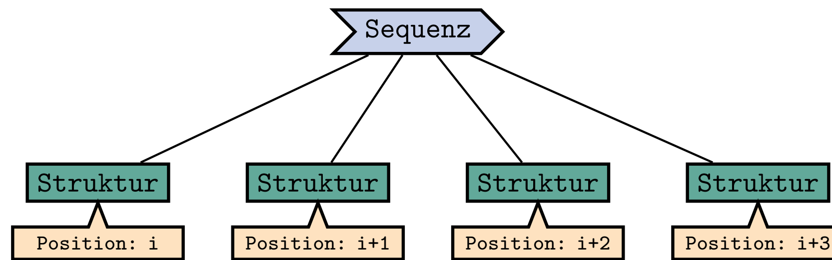


Abbildung 5.3: Sequenzstruktur

mulieren, um einem Roboter den Befehl zu geben ihm Medizin zu bringen. Sollte dies als Bedingung erkannt werden, so kann dies zu einer unzureichenden Ausführung führen. Eine alternative Möglichkeit hierzu wäre die Untersuchung der Methodensignatur. Anders als Methoden, müssen Bedingungen einen booleschen Wert als Rückgabetyp besitzen. Damit ist eine Identifizierung mit Hilfe der Ontologie möglich. Problematisch sind hierbei aber Methoden der Schnittstelle, die als Nebenfunktion einen booleschen Wert zurückgeben. Ein Beispiel hierfür wäre eine Methode, die dazu dient den Kühlschrank zu öffnen, aber bei bereits offenem Kühlschrank den Wert `false` zurückgibt. Solche Methoden dienen in der Regel dazu dem Programmierer über die Ausführung der Methode zu informieren. Da Bedingungen aber eine Teilmenge der Methoden sind, ist eine Ausführung, ohne Beachtung des Rückgabetyps, solcher Methoden nicht schädlich. Sie können daher wie Methoden verwendet werden. Das gegenteilige Problem, dass Methoden ohne Rückgabetyp als Bedingungen verwendet werden, lässt sich leicht erkennen und vermeiden. In Abbildung 5.2 werden Aktionen und Bedingungen symbolisch dargestellt.

5.4.2.2 Sequenz

Um Einzelanweisungen zu Anweisungssequenzen zu erweitern, ist die Verwendung von Sequenz-Strukturen notwendig. Diese bringen eine unsortierte Menge von Aktionen in eine explizite Ausführungsreihenfolge. Da die Sequenz, abseits der Wurzel, nur als Teilstruktur von Kontrollstrukturen und Parallelblöcken vorkommt, ist eine explizite Suche nach Sequenz-Strukturen unnötig. Die Sequenz-Struktur wird als Teilstruktur automatisch erzeugt, sobald die übergeordnete Elterstruktur gefunden wurde. Aktuell enthält PARSE keine semantischen Informationen zur korrekten Ausführungsreihenfolge der Aktionen. Aber jedes Token der Befehlsbeschreibung wird mit einem Positions-Attribut nach gesprochener Reihenfolge versehen. Die Ausführungsreihenfolge der Anweisungen wird in der Regel vom Nutzer sequentiell beschrieben. Ausnahmen bilden hierbei Sätze, die eine temporale Reihenfolge definieren, wie beispielsweise „*open the window but before that bring juice*“. Solche Strukturen müssen erkannt und bei der Erstellung der Ordnung beachtet werden. In PARSE soll dies in Zukunft durch einen eigenen Agenten stattfinden. Es soll daher angenommen werden, dass der Nutzer die Anweisungen immer sequentiell gibt. Eine Struktur kann daher die eigene Position anhand der Position der Token des *Musterbereichs* (siehe Definition 5.2) der Struktur bestimmen. Dies kann beispielsweise durch die Mittelung der Token-Positionen des Musterbereichs geschehen. Die sequentielle Ausführungsreihenfolge kann durch eine Sortierung der Kinder der Sequenz anhand der Position wiederhergestellt werden. In Zukunft kann ein neuer Agent die Tokens mit der korrekten Ausführungsreihenfolge annotieren. Die Strukturen können ohne Änderung des Verfahrens das neue Attribut statt der Position verwenden. So kann die Sequenz leicht auf die korrekte Ausführungsreihenfolge umgestellt werden. In Abbildung 5.3 ist die Sequenz symbolisch dargestellt.

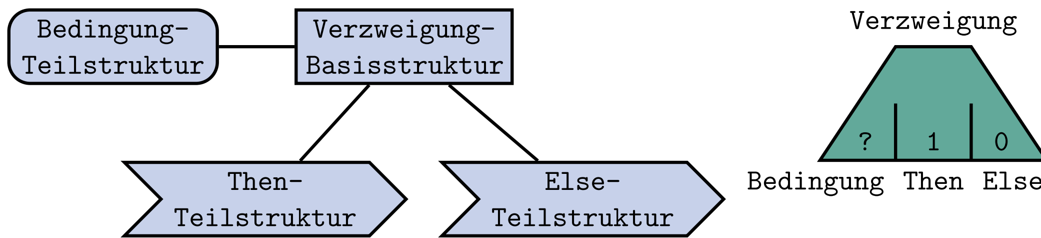


Abbildung 5.4: Verzweigungsstruktur und Symbol

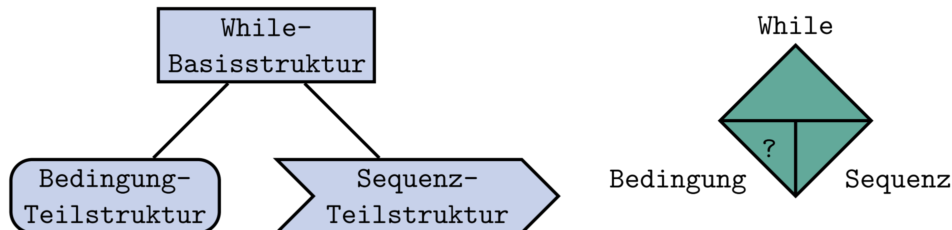


Abbildung 5.5: „While“-Schleifenstruktur und Symbol

5.4.2.3 Kontrollstrukturen

Die Strukturmenge der Anweisungssequenz erlaubt bereits die Erzeugung von einfachen Programmen. Diese sind aber in ihrer Komplexität stark eingeschränkt, da ohne weitere Kontrollstrukturen der Ablauf des Programms nicht steuerbar ist. Die wichtigsten Kontrollstrukturen zur Steuerung eines komplexen Programmablaufs sind Schleifen und Verzweigungen.

Die erste Kontrollstruktur ist die Verzweigung. Hierbei entscheidet die Auswertung der Bedingung im Kopf darüber, welche mögliche Rumpf-Sequenz ausgeführt wird. Der Kopf besteht aus einer Bedingung, welche über eine durch die Schnittstelle bereitgestellte Methode abgefragt wird. Der Rumpf der Verzweigung teilt sich hingegen in zwei mögliche Sequenzen. Die „Then“- und „Else“-Sequenzen können aus mehreren beliebigen Strukturen bestehen. Je nach Auswertung der Bedingung wird eine der beiden Sequenzen ausgeführt. Während die „Then“-Sequenz immer existieren muss, kann die „Else“-Sequenz weggelassen werden. Die Else-Sequenz wird in dieser Arbeit ebenfalls immer erstellt, da eine leere Sequenz äquivalent zu einer nicht vorhandenen ist. Dies erlaubt eine Vereinheitlichung beim Bearbeiten der in Abbildung 5.4 symbolisch dargestellten Verzweigungsstruktur. Es ist möglich die „Else“-Sequenz in einem späteren Schritt zur Optimierung wegzulassen. In PARSE existiert ein Agent, welcher Teilsätze mit unabhängig oder ihrer Zugehörigkeit zur Verzweigung entsprechend markiert. Dies steht im `commandType`-Attribut. Daneben werden im `conditionNumber`-Attribut die Verzweigungen durchnummeriert. Damit ist es möglich zusammengehörige Teilstrukturen zu erkennen. Aber Attribute eines Tokens können nicht gleichzeitig mehrfach belegt werden. Dadurch ist der PARSE-Agent zur Erkennung von Verzweigungen nicht in der Lage verschachtelte Verzweigungen zu erkennen. Ist ein Teilsatz mit `INDP` markiert, so ist er nicht Teil einer Verzweigung und kann ignoriert werden. Für jede der drei Teilstrukturen gibt es eine eigene Markierung. Sie werden entsprechend mit `IF`, `THEN` und `ELSE` annotiert. Die markierten Tokens stellen den Musterbereich dieser Teilstrukturen dar. Der Musterbereich der gesamten Struktur ist die Vereinigung seiner Teilstrukturen.

Als zweite Kontrollstruktur dienen Schleifen dazu eine Befehlssequenz zu wiederholen und lassen sich in zwei Kategorien einteilen: Als erste Art bestehen „While“-Schleifen aus einer Bedingung im Kopf und der Befehlssequenz im Rumpf. Bei negativem Auswertungsergebnis der Bedingung wird die Schleifenwiederholung abgebrochen. Der zu wiederholende

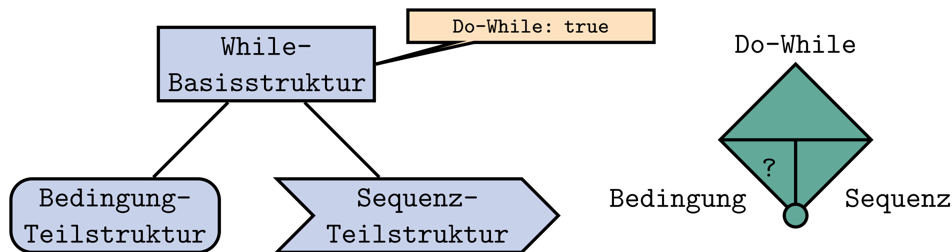


Abbildung 5.6: „Do-While“-Schleifenstruktur und Symbol

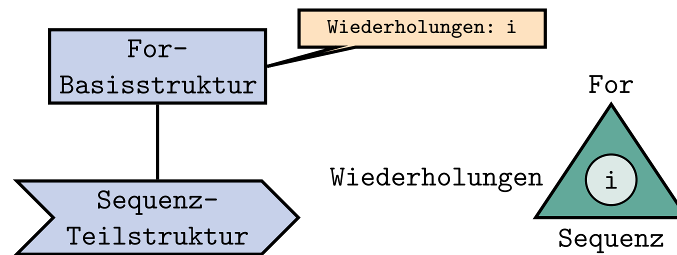


Abbildung 5.7: „For“-Schleifenstruktur und Symbol

Schleifenrumpf kann hingegen aus mehreren beliebigen Strukturen bestehen. Diese Schleifenart ist in Abbildung 5.5 dargestellt. Eine Spezialform dieser Schleifenart ist die in Abbildung 5.6 dargestellte „Do-While“-Schleife. Sie ist von der Form identisch zur normalen „While“-Schleife, wird aber immer mindestens einmal ausgeführt. Die Abbruchbedingung steht daher am Ende des Rumpfes. Die „For“-Schleife als zweite Art ist komplexer aufgebaut, da ihr Kopf aus drei Teilen besteht. So wird neben der Abbruchbedingung im Kopf ebenfalls eine Variable definiert, welche beim jedem Durchlauf des Rumpfes iteriert wird. Der Schleifenrumpf hingegen gleicht dem der „While“-Schleife. In Abbildung 5.7 ist die „For“-Schleife symbolisch dargestellt. Im PARSE-Projekt erkennt ein Agent diese Schleifen und erstellt einen „Loop“-Knoten im Graphen aus den Informationen der Schleifenbeschreibung. Dieser Knoten kann zur Erzeugung der Schleifen-Struktur verwendet werden.

Im „Loop“-Knoten verweist unabhängig vom Schleifentyp das `dependentPhrase`-Attribut auf den Teil der Beschreibung, die den Rumpf des Schleifenblocks bildet. Es ist als eine Liste von Tokens gespeichert und entspricht dem Musterbereich der Teilstruktur der Rumpf-Sequenz. Zusätzlich gibt es noch eine `dependentLoopAction`-Verbindung vom „Loop“-Knoten zu den Token der Sequenz. Im `keyphrase`-Attribut steht das Wort welches den Knotentyp impliziert. Je nach gefundenem Wort, wird ein anderer `keyphraseType` verwendet. Eine Verbindung vom Typ `loopKeyPhrase` zeigt auch direkt auf das entsprechende `keyphrase`-Token. Dazu gibt es das `condition`-Attribut, welches abhängig vom Schleifentyp verwendet werden. Die Art der Schleife ist nicht explizit Teil des Knotens. Stattdessen gibt das `keyphraseType`-Attribut einen starken Hinweis auf die Schleifenart. „While“- und „Do-While“-Schleifen besitzen `Opening`, `Ending` oder `Wrapping` als `keyphraseType`. Hierbei steht bei `Opening` die `keyphrase` am Anfang der Schleifenbeschreibung und impliziert eine „While“-Schleife. Äquivalent ist dies auch für `Ending` und `Do-While`-Schleifen der Fall. Hierbei steht die `keyphrase` am Ende der Schleifenbeschreibung. `Wrapping` bedeutet, dass es den Typ `Ending` und `Opening` annehmen kann. Es ist damit mehrdeutig. Diese Mehrdeutigkeit kann aufgelöst werden, indem eine der beiden Möglichkeiten festgeschrieben wird. Da bei der Erstellung des Agenten angenommen wurde, dass `Opening` öfters vorkommt, soll `Wrapping` ebenso verwendet werden. Das `condition`-Attribut verweist auf den Teil der Schleifenbeschreibung, welcher die Bedingung deklariert. Es stellt damit den Musterbereich der Bedingungs-Teilstruktur der Schleifenstruktur dar. Wegen der struktu-

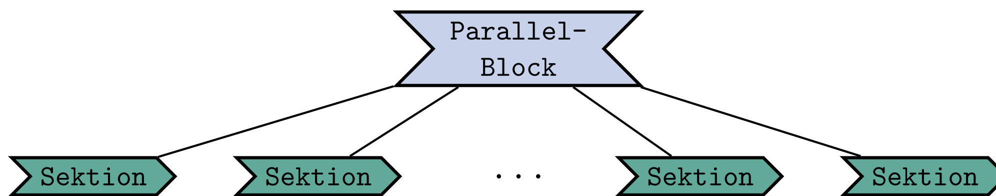


Abbildung 5.8: Parallelblock

rellen Ähnlichkeit zwischen „While"- und „Do-While"-Schleife, kann die selbe Struktur für beide verwendet werden. Es ist ausreichend, ein Attribut zu definieren, welches die Unterscheidung erlaubt. Die „For"-Schleife hingegen lässt sich durch den Typ `For` erkennen. Das `keyphrase`-Attribut ist hierbei ein Zählwort wie „*twice*" oder „*five times*". Da „For"-Schleifen als Zählschleifen benutzt werden, ist die Anzahl an Wiederholungen ausreichend, um den Schleifenkopf zu konstruieren. Deshalb bleibt das `condition`-Attribut ungenutzt. Aus dem `keyphrase`-Attribut muss aber die Wiederholungsanzahl für die „For"-Schleife extrahiert werden. In dieser Arbeit wird hierfür eine einfache Tabelle genutzt, die Wörter wie „*twice*" auf ihre entsprechenden Werte abbildet.

5.4.2.4 Parallelität

In diesem Abschnitt wird als letzte Struktur die Erweiterung der gesteuerten Anweisungssequenz um Parallelität untersucht. Besitzt eine Anwendung die Möglichkeit dazu, so können dadurch mehrere Sequenzen zeitgleich ausgeführt werden. Die Parallelität kann daher als besondere Form der Kontrollstruktur gesehen werden. Anstatt den Pfad des Programmablaufs zu steuern, teilt eine Parallelitätsstruktur diesen Pfad in mehrere Pfade auf. Soll beispielsweise ein Roboter in der Lage sein gleichzeitig zu singen und tanzen, so müssen beide Befehle zeitgleich ausgeführt werden. Die korrekte Integration der Parallelität erzeugt viele Herausforderungen. Als erstes muss eine passende Repräsentation der Parallelität im abstrakten Syntaxbaum gefunden werden. Der Beginn der parallelen Ausführung kann durch eine einzelne Parallel-Struktur dargestellt werden, deren Kindstrukturen bei der Übersetzung parallel ausführbaren Quelltext implizieren. Die Herausforderung besteht darin, dass die Kindstrukturen der Parallel-Struktur nicht nur aus einer einzelnen Struktur bestehen, sondern einen neuen Syntaxbaum implizieren können. Am Ende der Ausführung muss die parallele Ausführung für alle gestarteten Kinder wieder beendet werden. Eine große Herausforderung wäre das Einsammeln der Ergebnisse der Kinder. Hierfür wäre es beispielsweise nötig gemeinsame Variablen zu deklarieren. Wegen der hohen Komplexität soll dies aber in dieser Arbeit nicht beachtet werden. Es werden daher nur Probleme betrachtet, bei denen die Kinder voneinander unabhängig sind.

Eine Möglichkeit Parallelität im abstrakten Syntaxbaum zu repräsentieren, wäre die Nutzung eines „Parallelblocks". Ein solcher ist in Abbildung 5.8 symbolisch dargestellt. Die Kindstrukturen dieses Parallelblockes würden jeweils einen parallelen Abschnitt als Teilbaum darstellen. Hierbei hilft, dass die Wurzel eines Syntaxbaumes immer als Sequenz-Struktur repräsentiert werden kann. Dies lässt sich durch zwei Fälle zeigen: Sollen die Kindstruktur der Wurzel eine Sequenz bilden, so ist es nötig, dass die Wurzel eine Sequenz-Struktur ist. Besitzt die Wurzel hingegen nur eine Kindstruktur, so lässt sich die Wurzel ebenfalls als Sequenz-Struktur darstellen. Der Grund hierfür ist, dass eine Sequenz mit nur einer Kindstruktur, bei der Ausführung äquivalent zu ihrer Kindstruktur ist. Dadurch lässt sich jeder Syntaxbaum so erweitern, dass die Wurzel aus einer Sequenz-Struktur besteht. Dies erlaubt es Syntaxbäume als Sequenz-Teilbäume des Parallelblockes einzubinden. Hierbei sollte eine Barriere den Parallelblock zur Synchronisierung abschließen. Eine Herausforderung stellen die unterschiedlichen Implementierungen in verschiedenen Zielsystemen dar. Bereits die Wahl der Programmiersprache führt zur Notwendigkeit die

parallele Konstruktion anzupassen. So bietet OpenMP für C++ eine angenehme Möglichkeit Parallelität leicht durch parallele Abschnitte (engl. parallel sections) zu implementieren. Diese haben auch eine implizite Barriere an ihrem Ende. Parallele Abschnitte fehlen hingegen bei Java, weshalb man auf eigene Implementierungen zurückgreifen muss. Aber auch Zielsysteme mit der gleichen Programmiersprache können sich in der konkreten Implementierung unterscheiden. So kann beispielsweise ein System die nebenläufige Ausführung des Programms erkennen, während ein anderes System spezielle Anweisungen über die Schnittstelle erwartet. Damit wäre die Parallelität, die einzige Struktur, deren Quelltext-Äquivalent von mehr als nur der Programmiersprache abhängt. Um die Zielsystemunabhängigkeit der Übersetzung zu erreichen, muss sie also gesondert betrachtet werden. Dies

Ein Agent in PARSE erkennt Parallelität und erstellt einen Knoten vom Typ `concurrentAction`. Der Knoten besitzt ein `keyphrase`-Attribut für das Signalwort und ein `type`-Attribut für den Typ. Weiterhin gibt es noch eine `concurrentActionKeyPhrase`-Verbindung, welche auf das Token des Signalwortes zeigt. Zur Übersetzung in einen Parallelblock werden diese nicht benötigt. Das dritte Attribut ist `dependentPhrases` und ist eine Liste der Tokens, die im Rumpf der Parallelität sind. Zusätzlich gibt es eine `dependentconcurrentAction`-Verbindung, die vom Knoten auf das Token der Aktionen verweist. Anhand der referenzierten Aktionen kann der Musterbereich für die Sequenz-Teilstrukturen und der Parallelblock bestimmt werden.

5.4.3 Verbindung der Strukturen

Um von den gefundenen Strukturen zum endgültigen abstrakten Syntaxbaum zu kommen, müssen die Strukturen verbunden werden. Dabei ist darauf zu achten, dass der erzeugte Baum vollständig ist und die Verbindungen konsistent sind. Dies bedeutet, dass beispielsweise an Stelle einer nötigen Bedingung keine Schleife angebunden werden darf. Auch muss darauf geachtet werden, dass die Strukturen einer Ebene in sich konsistent sind. Dies ist wichtig, um die Bedeutung der Beschreibung zu erhalten. Hierbei muss die Reihenfolge beachtet werden, in der beispielsweise Befehle ausgeführt werden. Um die Verbindungen aufzubauen, gibt es mehrere Möglichkeiten. Beim ersten Ansatz werden zu einer passenden Struktur alle möglichen Kindstrukturen gesucht. Der Syntaxbaum wird so rekursiv von der Wurzel aus in Richtung der Blättern aufgebaut. Dies hat den Vorteil, dass der entstehende Syntaxbaum immer verbunden ist. Erst wenn alle Strukturen verbunden sind, ist der Syntaxbaum vollständig. Bleiben am Ende unverbundene Strukturen übrig, so weißt dies auf Fehler hin. Als Nachteil ist zu nennen, dass Strukturen wie Sequenzen, durch Mehrdeutigkeiten leicht zu Fehlern führen können. Im Gegensatz dazu wäre es auch möglich den Baum von den Blättern aus aufzubauen. Hierzu würde jede Struktur ihre Vorgänger suchen und sich an diese als Kind anbinden. Bei dieser Ansatz würden also mehrere Teilbäume gleichzeitig aufgebaut und mit jedem Schritt weiter miteinander verbunden. Mit dem Anschluss der Teilbäume an die Wurzel als letzten Schritt entsteht der vollständige Syntaxbaum. Da die Strukturen jeweils nur einen Vorgänger haben können, kann es leichter sein, bei Mehrdeutigkeiten eine Entscheidung zu fällen. Weiterhin lassen sich die Probleme dieses Ansatzes leichter in Teilprobleme zerlegen.

Auf dem zweiten Ansatz basierend wird ein Verfahren entworfen, welches in zwei Phasen einen abstrakten Syntaxbaum aus den gefundenen Strukturen aufbaut: In der ersten Phase werden die Strukturen zu separaten Pfaden verbunden. Um die Strukturen zu finden, die einen Pfad bilden, wird der Musterbereich der Strukturen betrachtet. In der zweiten Phase werden die gefundenen Pfade zu einem Syntaxbaum verschmolzen. Da die Pfade sich in ihren Strukturen überschneiden, können sie leicht zusammengefügt werden. Dadurch entsteht der vollständig aufgebaute abstrakte Syntaxbaum.

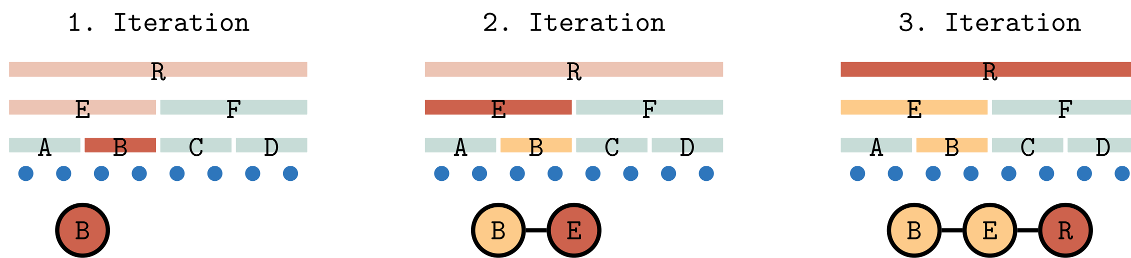


Abbildung 5.9: Beispiel zum Aufbau eines Pfades des Syntaxbaumes über Musterbereiche

5.4.3.1 Aufbau der Pfade

Als erstes soll der Aufbau eines einzelnen Pfades betrachtet werden. Um einen vollständig Pfad aufzubauen, müssen zuerst die Strukturen des Pfades gefunden werden. Dies kann durch einen Vergleich der Musterbereiche der Strukturen erreicht werden. Überschneiden sich die Musterbereiche zweier Strukturen, so ist ein Zusammenhang zwischen diesen anzunehmen. Dieser Zusammenhang ist in der Regel eine hierarchische Ordnung. Enthält eine Ordnung alle Strukturen eines Pfades von der Wurzel bis zum Blatt, so entspricht sie dem Pfad. Die hierarchische Ordnung kann durch einen Vergleich der Musterbereiche gefunden werden. Hierzu soll zum besseren Verständnis zuerst in Definition 5.10 die *Größe von Strukturen* eingeführt werden.

Definition 5.10: Größe von Strukturen

Die Größe von Strukturen bringt eine Menge von Strukturen anhand ihrer Musterbereiche in eine Ordnung.

Überdeckende Struktur

Überdeckt der Musterbereich einer Struktur den Musterbereich einer anderen Struktur vollständig, so wird die Struktur als **größer** oder **überdeckenede Struktur** bezeichnet.

Überdeckte Struktur

Die Struktur deren Musterbereich vollkommen überdeckt ist, wird entsprechend als **kleiner** und **überdeckte Struktur** bezeichnet.

Umschlossene Struktur

Eine Struktur ist von zwei Strukturen **umschlossen**, wenn sie kleiner als die überdeckende Struktur ist und gleichzeitig größer als die überdeckte Struktur.

Überdeckt eine Struktur eine andere Struktur, so ist die größere Struktur Vorfahre der überdeckten Struktur. Entsprechend ist die überdeckte Struktur der Nachfolger der größeren Struktur. Gibt es keine andere Struktur, die von den beiden Strukturen umschlossen ist, so besteht zwischen den beiden Strukturen eine Elter-Kind-Beziehung. Oft ist es sogar so, dass eine Teilstruktur eine andere Struktur überdeckt. In diesem Fall ist die Teilstruktur der Vorfahre der Struktur. Entspricht der Musterbereich der Teilstruktur dem der anderen Struktur, so besteht eine Elter-Kind-Beziehung zwischen der größeren Teilstruktur und der überdeckten Struktur.

Wird eine Menge von Strukturen betrachtet, deren *Kern* (siehe Definition 5.11) existiert, so kann diese Menge in eine lineare hierarchische Ordnung gebracht werden. Anhand dem Beispiel in Abbildung 5.9 soll dies verdeutlicht werden. Die blauen Punkte stellen hierbei

Definition 5.11: Kern und Kernstruktur

Der **Kern** ist der Schnitt der Musterbereiche einer Menge von Strukturen. Ist der Kern nicht leer, so besteht zwischen den Strukturen ein Zusammenhang.

Kernstruktur

Als **Kernstruktur** wird die Struktur bezeichnet, deren Musterbereich dem Kern am ähnlichsten ist. Die Ähnlichkeit der Kernstruktur zum Kern wird über eine **Ähnlichkeitsfunktion** bestimmt. Sie ist definiert als Abbildung zwischen Kern und Kernstruktur auf einen Wert.

Minimalkern

Ein **Minimalkern** ist ein Kern, für den es keine Menge an Strukturen gibt, die einen anderen Kern implizieren, der Teil des Minimalkerns ist.

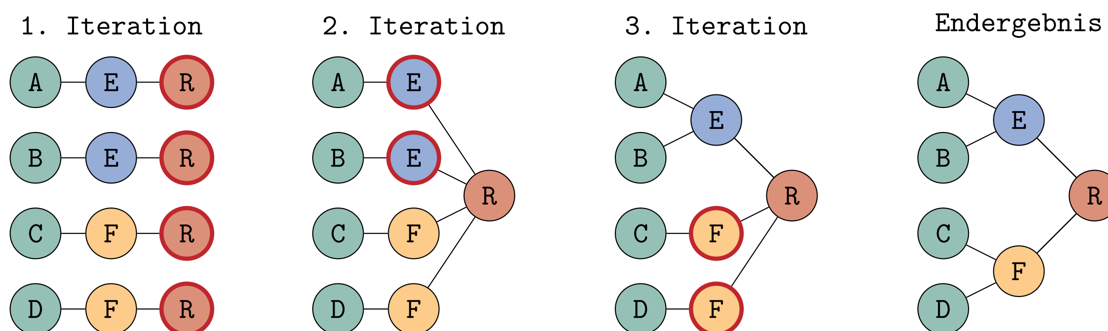


Abbildung 5.10: Beispiel zur Verschmelzung der Pfade von der Wurzel aus

symbolisch Wörter dar. Der aktuell im Aufbau befindliche Pfad steht darunter. Die Linien über den Wörtern markieren die Musterbereiche der zugehörigen Struktur. In rot sind alle Musterbereiche markiert, die den Minimalkern `B` beinhalten. Die anderen sind grün dargestellt und werden nicht betrachtet. Die Kernstruktur (1. Iteration, dunkelrot) der Menge bildet hierbei die niedrigste Ebene der Ordnung. Wird die Kernstruktur aus der Menge entfernt (2. Iteration, orange), lässt sich ein neuer Kern (2. Iteration, dunkelrot) bilden. Dieser neue Kern ist größer als der ursprüngliche Kern, aber der nächst kleinste Kern der ursprünglichen Menge. Er bildet daher die neue Kernstruktur. Die vorherige Kernstruktur ist ein Kind der neuen Kernstruktur, da es keine umschlossene Struktur zwischen der vorherigen Kernstruktur und der neuen Kernstruktur geben kann. Ein Sonderfall entsteht, wenn die Kernstruktur der Menge eine Teilstruktur ist. Die zur Teilstruktur gehörige Basisstruktur ist automatisch Elter und wird die nächste Kernstruktur. Dieser Vorgang lässt sich iterativ wiederholen, bis die gesamte Menge verbunden ist (3. Iteration). Soll ein Pfad aufgebaut werden, so muss die Ordnung alle Strukturen des Pfades von der Wurzel bis zum Blatt enthalten. Das Blatt ist dabei immer die Minimalkernstruktur. Die Wurzel (`R`) des Syntaxbaumes bildet immer den Abschluss des Verfahrens. Dadurch ist garantiert, dass alle gefundenen Pfade miteinander zu einem Syntaxbaum verbunden werden können. Ein ähnliches Verfahren wäre anwendbar, um den Pfad von der Wurzel aus aufzubauen.

5.4.3.2 Pfadverschmelzung

Das Vorgehen kann erweitert werden, um den gesamten Syntaxbaum aufzubauen. Um die anfänglichen Pfade aus den Strukturen aufzubauen, müssen zuerst die Blätter der Pfade identifiziert werden. Dazu werden alle Minimalkerne der Eingabe gebildet, da die Kernstrukturen der Minimalkerne den Blättern entsprechen. Um die Strukturen des Pfades zu

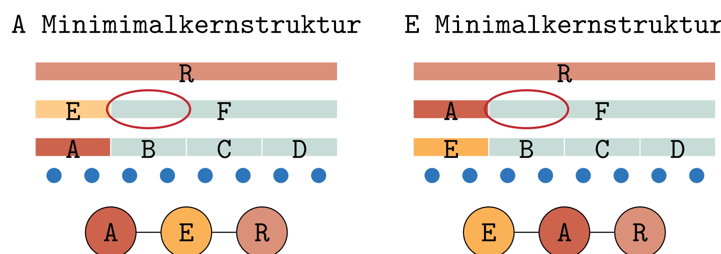


Abbildung 5.11: Beispiel für fehlerhafte Musterbereiche (rot umrandet) über mehrere Ebenen. Die Musterbereiche der Strukturen `A` und `E` sind identisch. Dadurch existieren für den Minimalkern mehrere Minimalkernstruktruen und somit mehrere mögliche Pfade. Diese Mehrdeutigkeit lässt sich im Allgemeinen nicht auflösen.

finden, werden die Musterbereiche der Strukturen mit dem Minimalkern verglichen. Alle Strukturen deren Musterbereich den Minimalkern überdeckt, sind Teil des Pfades. Die gesammelten Strukturen eines Pfades können mit dem in Abschnitt 5.4.3.1 beschriebenen Verfahren zu Pfaden verbunden werden. Dadurch wurden alle möglichen Pfade von der Wurzel zu den Blättern des Syntaxbaumes gefunden. In Abbildung 5.10 ist die Verschmelzung der Pfade zu einem Baum beispielhaft dargestellt. Die grün markierten Minimalkernstrukturen entsprechen hierbei den Blättern des abstrakten Syntaxbaumes. Da sie Minimalkernstrukturen entsprechen, sind sie einzigartig. Im ersten Schritt sind die separaten Pfade dargestellt. Durch eine Verschmelzung der Pfade kann der Baum aufgebaut werden. Um zu vermeiden, dass mehrdeutige Teilbäume enstehen sollte die Verschmelzung bei der Wurzel anfangen. Die rot markierten Wurzelstrukturen `R` sind immer Teil aller Pfade. Sie werden als erstes vereint. Im zweiten Schritt werden die blauen Strukturen `E` verschmolzen. Dies wird in den nächsten Schritten iteriert bis im letzten Schritt keine mehrfachen Strukturen existieren. Der Syntaxbaum ist nun vollständig aufgebaut.

5.4.3.3 Schwierigkeiten und Ähnlichkeitsfunktion

Im idealen Fall sind die Kerne der Kernstrukturen (echte) Teilmengen der Kerne ihrer Vorfahren. Der ideale Fall erlaubt eine weitere Vereinfachung. So ließe sich die Ordnung auch durch eine Sortierung anhand der Größe der Strukturen herstellen. Hierbei sollte aber beachtet werden, dass Teilstrukturen von ihrer Basisstruktur nicht getrennt werden. Die kleinsten Strukturen bilden die Blätter, während die größte Struktur der Wurzel entspricht.

Eine Schwierigkeit ist, dass selbst wenn alles korrekt erkannt wurde es zu Mehrdeutigkeiten kommen kann. Sind die Musterbereiche zweier Strukturen wie in Abbildung 5.11 identisch und ebenfalls Kern, so entstehen Mehrdeutigkeiten. In der Praxis ist aber eine dieser Strukturen eine Teilstruktur, weshalb sie sich als Kernstruktur ausschließen lässt. Dieser Problemfall sollte aber trotzdem bei der Erweiterung um weitere Strukturen beachtet werden. Werden die Strukturen fehlerhaft oder ungenau erkannt, so hat dies auch Auswirkungen auf die Musterbereiche. So können sich beispielsweise Musterbereiche überschneiden, die im idealen Fall disjunkt sind, oder disjunkte Musterbereiche verschoben sein. Der verschobene Problemfall ist in Abbildung 5.12 dargestellt. Je nach Implementierung kann dies unterschiedliche Auswirkungen haben. Alternativ kann es auch sein, dass der Minimalkern, je nach Wahl der Struktur um den fehlerhaften Bereich, verkleinert wird. So können an der gleichen Stelle mehrere Strukturen als Kernstruktur in Frage kommen. Bei großen Fehlern, kann eine eigentlich größere Struktur von einer eigentlich kleineren überdeckt werden. Dadurch kann es passieren, dass die Reihenfolge der Strukturen im Pfad vertauscht wird. Ein weiteres Problem kann bei der Bestimmung des Kerns entstehen. Ist der Musterbereich eines Vorfahren der idealen Kernstruktur kleiner als der ideale Kern, so

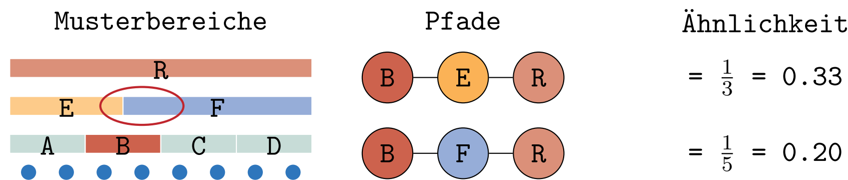


Abbildung 5.12: Beispiel für fehlerhafte Musterbereiche (rot umrandet) innerhalb einer Ebene. Dadurch existieren für die Minimalkernstruktur `B` mehrere Elterstruktruen und somit mehrere mögliche Pfade. Die Verwendung einer Ähnlichkeitsfunktion wie Gleichung 5.1 führt zu einer höheren Bewertung von `E` gegenüber `F`.

wird ein kleinerer Kern gefunden. Dieser fehlerhafte Kern erschwert die Bestimmung der korrekten Kernstruktur, da der Kern nur eine Teilmenge des Musterbereichs der idealen Kernstruktur ist. Es kann daher sein, dass die Struktur, mit dem fehlerhaften Musterbereich dem Kern am ähnlichsten ist. Sie würde damit an die falsche Stelle eingeordnet werden. Grundsätzlich entsteht je nach Wahl der Kernstruktur bei Mehrdeutigkeiten ein anderer Pfad.

Da die Wahl der Kernstruktur Auswirkungen auf das Ergebnis hat, ist es wichtig eine treffende Ähnlichkeitsfunktion zu definieren. Die Ähnlichkeitsfunktion ist abhängig von dem Kern und dem Musterbereich der möglichen Kernstruktur. Eine Möglichkeit die Mehrdeutigkeit in Abbildung 5.12 aufzulösen, wäre daher die verwendung einer Ähnlichkeitsfunktion. Ein einfaches Beispiel für eine solche Ähnlichkeitsfunktion wäre die Abdeckungsrate des Musterbereichs der Struktur durch den Kern:

$$A(\text{Struktur}, \text{Kern}) = \frac{\text{Anzahl Tokens im Kern}}{\text{Anzahl Tokens des Musterbereichs der Struktur}} \tag{5.1}$$

Da Kern und Musterbereich sich auf die gleichen Satzteile beziehen, können die Wörter als Symbole interpretiert werden. Eine weitere Möglichkeit wäre daher die Verwendung von Stringmetriken, die die Distanz zwischen zwei Sequenzen von Symbolen bestimmen. Beispiele hierfür wären die Levenshtein- [Lev] oder Hamming-Distanz.

5.4.4 Folgefehler

Die Synthese des abstrakten Syntaxbaumes baut auf den Ergebnissen vorheriger Arbeiten auf. Dies führt dazu, dass Folgefehler großen Einfluss auf die Korrektheit des abstrakten Syntaxbaumes haben. Wird beispielsweise im Satz „*if the fridge is open grab some vodka*“ die Wortart von „*grab*“ als Substantiv statt Verb falsch markiert, führt dies dazu, dass die Vorverarbeitung die Teilsätze nicht korrekt teilt. Diese fehlende Teilung führt dann dazu, dass der Agent, der die Bedingungen erkennt, dem gesamten Satz die gleiche Zuweisung gibt. Beim Aufbau des abstrakten Syntaxbaumes fehlen nun notwendige Informationen, um die Verzweigung vollständig aufzubauen. Fehler dieser Art können viele unterschiedliche Ursprünge haben. Beim Aufbau ist ihre Korrektur in der Regel nicht möglich. Werden aber Sätze als fehlerhaft identifiziert, so können sie für einen Dialog markiert werden. Der Dialogagent kann daraufhin den Nutzer bitten eine Korrektur vorzunehmen. Damit der Dialogagent auf Informationen des abstrakten Syntaxbaum zugreifen kann, wird der Syntaxbaum im PARSE-Graphen aufgebaut. Nachdem der Syntaxbaum aufgebaut wurde, kann eine statische Analyse des Baumes stattfinden. Da der Aufwand zur Implementierung von Analyseverfahren aus dem Übersetzerbau den Rahmen dieser Arbeit übersteigt, soll nur eine oberflächliche Analyse stattfinden. Bei der oberflächlichen Analyse überprüfen die Mustererkenner, ob der Baum eine Reihe von Bedingungen, die überprüfen ob Strukturen

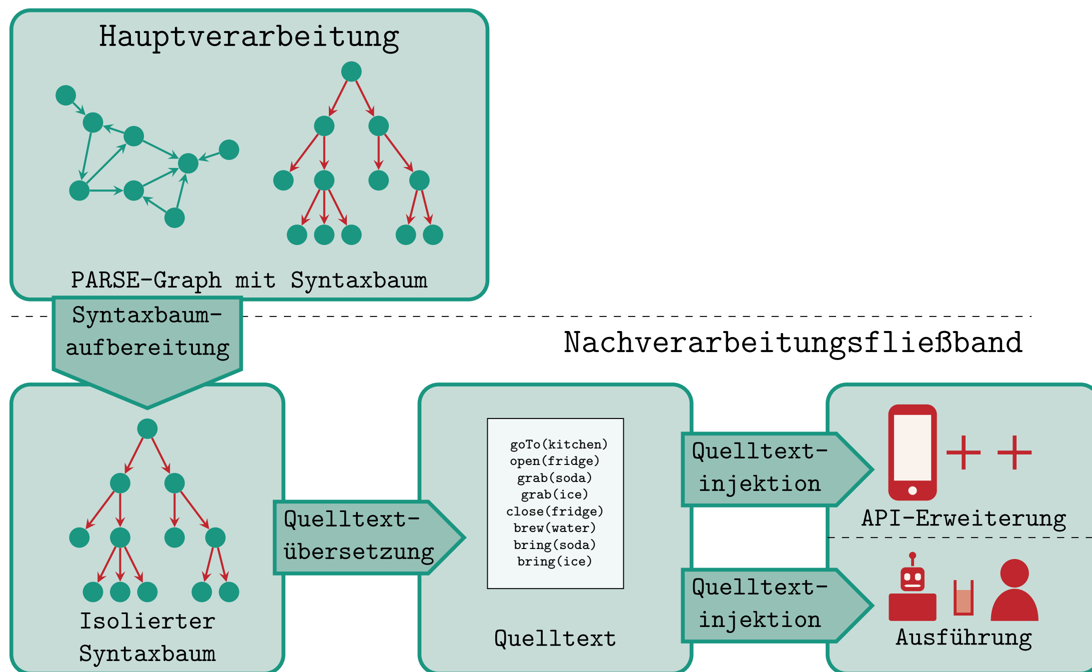


Abbildung 5.13: Nachverarbeitungsfließband zur Quelltexterzeugung

oder Verbindungen fehlen, erfüllt. Für jede Bedingung, die der Syntaxbaum nicht erfüllt, wird der Graph um einen Fehlerknoten erweitert.

5.5 Quelltexterzeugung

Durch die Verwendung des abstrakten Syntaxbaumes als Zwischenrepräsentation, muss das Quelltexterzeugungsmodul nur noch auf diesen zugreifen. Es kann somit vollständig wie in Abbildung 5.13 dem Nachverarbeitungsfließband zugeschrieben werden. Der erzeugte abstrakte Syntaxbaum wurde während der Hauptverarbeitung in den PARSE-Graphen geschrieben. Da ein Zugriff auf den PARSE-Graphen im Nachverarbeitungsfließband nicht mehr möglich ist, muss der Syntaxbaum zuerst aus dem PARSE-Graphen extrahiert werden. Der isolierte Syntaxbaum kann dann in Quelltext übersetzt werden. Bei der Übersetzung des Syntaxbaumes in Quelltext gibt es bereits verschiedene Ansätze aus dem Bereich der Programmtransformation. Ein Anwendungsbereich dabei ist die Transpilierung, d.h. die Übersetzung von Quelltext einer Programmiersprache in den Quelltext einer anderen. Hierbei wird ebenfalls als Zwischenrepräsentation eine Form von Syntaxbaum verwendet, welcher in der Regel über das Besucher-Muster abgelaufen wird und in Quelltext übersetzt (engl. pretty printing) wird. Zum Schluss soll der erzeugte Quelltext abhängig seiner Bestimmung ins Zielsystem injiziert werden. Hierfür eignet sich die Verwendung eines Schablonen-Systems (engl. template system).

Eine Herausforderung ist es sicherzustellen, dass möglichst viele Zielsysteme angesprochen werden können. Diese Zielsysteme unterscheiden sich in der Domäne, der Schnittstelle und auch der Programmiersprache. Die Domäne und Schnittstelle des Systems wird über eine Wissensrepräsentation, d.h. Ontologie beschrieben. Dies Ontologie wurde in Abschnitt 3.2.3 beschrieben. Eine weitere Herausforderung ist daher, das Werkzeug unabhängig der konkreten Wissensrepräsentation zu entwerfen. Um sicherzustellen, dass die Anbindung an diese verschiedenen Systeme möglich ist, soll auf verschiedenen Systemen, die sich in Einsatzbereich und Programmiersprache unterscheiden, evaluiert werden.

5.5.1 Aufbereitung des abstrakten Syntaxbaumes

Da der PARSE-Graph nicht als Syntaxbaum entworfen wurde, müssen zur Weiterverarbeitung die Syntaxbaum-Strukturen des PARSE-Graphen zu einem vollwertigen abstrakten Syntaxbaum extrahiert werden. Dazu werden die Strukturen und Verbindungen des Syntaxbaum im PARSE-Graphen in äquivalente Strukturen übersetzt. Es bestünde hierbei die Möglichkeit eine letzte Auflösung von Mehrdeutigkeiten vorzunehmen. Wurden die möglichen Verbindungen zwischen den Strukturen im PARSE-Graphen mit Konfidenzen annotiert, so lassen sich aus den gleichen Strukturen unterschiedliche Bäume durch alternative Verbindungen aufbauen. Unterschiedliche Verbindungen der Muster haben Auswirkungen auf die Struktur des Baumes. Durch die Konfidenzen der Verbindungen, ist es möglich eine Konfidenz für den gesamten Baum zu berechnen. Hierbei sollte auch darauf geachtet werden, dass die Syntaxbäume kohärent sind. Eine statische Analyse auf den so erzeugten Syntaxbäumen erlaubt fehlerhafte Verbindungen und Bäume direkt auszuschließen. Das Ergebnis ist ein einziger abstrakter Syntaxbaum, dessen Kompilierbarkeit nach der Übersetzung in Quelltext gesichert ist. Auf der anderen Seite dürften fehlende Strukturen und fehlerhafte Verbindungen die häufigsten Fehlerfälle sein. Zwar können fehlende Strukturen leicht festgestellt werden. Problematisch dabei ist aber, dass fehlende Strukturen keine Alternative besitzen. Daher wäre eine Fehlerkorrektur durch das Suchen nach einer Alternative bei fehlenden Strukturen nicht möglich. Bei fehlerhaften Verbindungen könnte hingegen eine Suche nach Alternativen zu einer Verbesserung führen. Der Aufwand für eine solche Analyse würde aber den Rahmen dieser Arbeit übersteigen. Aus diesem Grund wurde der Fokus auf der Erzeugung eines einzelnen abstrakten Syntaxbaumes gelegt. Der erzeugte Syntaxbaum soll daher nur durch Bedingungen überprüft werden.

5.5.2 Zielsystemunabhängige Quelltextsynthese

Die Übersetzung eines abstrakten Syntaxbaumes in Quelltext erfolgt durch einen Besucher. Dieser durchläuft den Syntaxbaum rekursiv und übersetzt die Knoten des Baumes in Quelltext. Dieses Verfahren wird beim Übersetzerbau in der Regel eingesetzt, um eine höhere Programmiersprache über Syntaxbäume in eine maschinennahe Sprache zu übersetzen. Auch im Bereich der Programmtransformation wird das gleiche Verfahren verwendet. Hierbei wird beispielsweise der ursprüngliche Quelltext in Form eines Syntaxbaumes bearbeitet. Anschließend wird dieser bearbeitete Syntaxbaum dann durch Besucher in die ursprüngliche oder eine andere Programmiersprache übertragen. Sind also ausreichend Informationen im Syntaxbaum vorhanden, so kann dieser leicht in den Quelltext einer Programmiersprache übersetzt werden. Das Verfahren ist also unabhängig von der Programmiersprache verwendbar. Sind die Sprachen strukturell ähnlich, so können durch die Implementierung neuer Besucher Programmiersprachen leicht hinzugefügt werden. Damit wird die Unabhängigkeit des Verfahrens von der Programmiersprache erreicht.

Das Übersetzen der Parallel-Struktur benötigt weitere Beachtung, da sich die konkrete Implementierung selbst bei gleicher Programmiersprache unterscheiden kann. Die einfachste Art Sonderfälle der Parallelität zu integrieren, wäre die Verwendung weitere Besucher. Die Besucher würden die Besucher der entsprechenden Programmiersprache um die konkrete Art der Parallelität erweitern. Bei der Übersetzung der anderen Strukturen hingegen würde keine Änderung vorgenommen.

Bei der Injektion des erzeugten Quelltextabschnittes muss unterschieden werden, ob die Anweisungen eine Befehls- oder Lehrsequenz darstellen. Je nachdem unterscheidet sich die Form und auch das Injektionsziel. Bei einer Lehrsequenz muss die Schnittstelle um den Quelltextabschnitt mit einer passenden Methodensignatur erweitert werden. Die so erlernte neue Fähigkeit muss danach ebenfalls in der Ontologie repräsentiert werden. Bei einer Befehlssequenz hingegen muss der Quelltext nur der Anwendung zur Ausführung

übergeben werden. Abhängig vom Zielsystem können sich hier aber die genauen Spezifikationen unterscheiden. Mit der Verwendung eines Schablonen-Systems kann der Quelltext elegant injiziert werden. Hierbei können auch notwendigen Spezifikationen der Anwendung über die Schablone dargestellt werden. Der generierte Quelltext wird in dieser Phase nicht mehr verändert. Es ist daher sinnvoll die Synthese des Quelltextes von der Injektion ins Zielsystem zu trennen. Dies würde eine Wiederverwendbarkeit des Besuchers einer Programmiersprache erlauben. Andere Zielsysteme könnten durch einen Austausch der Schablonen angesprochen werden.

5.6 Maschinelles Lernen

Neben einem Musterbasiertem Ansatz wäre, wie aus Kapitel 4 ersichtlich, ein Ansatz über maschinelles Lernen eine Möglichkeit die Problemstellung zu lösen. Während als Eingabe weiterhin der PARSE-Graph dient, kann der Klassifikator auf verschiedene Ziellösungen trainiert werden. Die erste Möglichkeit wäre die direkte Übersetzung vom Eingabegraph in Quelltext. Dies zeigte sich aber bereits in Abschnitt 4.1.2 meist als wenig erfolgreich an. So gab es Probleme die gewünschte Bedeutung der Texte in Quelltext zu übersetzen. Aber auch der generierte Quelltext war meist inkonsistent und fehlerhaft. Beispielhaft hierfür waren unterschiedliche Variablennamen und fehlerhafte Methodenaufrufe. Das direkte Übersetzen durch Klassifikatoren führte daher selten zu einem korrekten Ergebnis. Eine weitere Möglichkeit basiert darauf, dass Quelltext und Syntaxbäume sich leicht in einander übersetzen lassen. Daher kann, statt der direkten Übersetzung, auch ein Klassifikator darauf trainiert werden, den Eingabegraphen in einen abstrakten Syntaxbaum zu übersetzen. Der so erzeugte Syntaxbaum kann dann mit einem Standardverfahren in Quelltext übersetzt werden. Da beides als Graph repräsentiert wird, könnte dies zu besseren Ergebnissen führen, als die direkte Übersetzung in Quelltext.

Abseits der direkten Übersetzung kann der musterbasierte Ansatz mit Klassifikatoren kombiniert werden. Der Klassifikator würde so nur einzelne Muster erkennen und aus ihnen Teilbäume erzeugen. Die Mustererkenner des in Abschnitt 5.4.1 beschriebenen Ansatz könnten so durch Klassifikatoren ersetzt werden, ohne den Entwurf zu ändern. Dieser hybride Ansatz würde auch erlauben bei bestimmten Strukturen einfache Mustererkenner zu verwenden statt Klassifikatoren. Ein weiter hybrider Ansatz wäre die Verbindung der gefundenen Strukturen durch Klassifikatoren.

Jeder Ansatz der auf maschinellem Lernen beruht, benötigt aber große Mengen an Trainingsdaten und Lösungen. Für PARSE existiert aber nur ein Korpus mit Szenariobeschreibungen für einen Küchenroboter. In diesem Korpus gibt es weder einen Goldstandard für Zwischenschritte noch für das Endergebnis. Daher ist das maschinelle Lernen als Ansatz zur Lösung des Problems mit dem PARSE-Korpus ungeeignet. Bei alternativen Korpora mit entsprechenden Goldlösungen, besteht das Problem, dass ihre Texte nicht auf PARSE ausgelegt ist. Da die Goldlösung meist als Quelltext vorhanden ist, müsste die vorher in einen äquivalenten Syntaxbaum übersetzt werden. Dazu gibt es bei diesen Korpora keinen Goldstandard für den PARSE-Graphen. Man müsste also diesen entweder aufwendig von Hand erzeugen, oder durch PARSE erzeugte Fehler übernehmen. Es ist weiterhin fraglich ob der maschinelle Ansatz einem musterbasierten Ansatz überlegen ist. Im PARSE-Graphen wurden bereits viele Informationen der Szenariobeschreibung extrahiert. Die Erkennung und Übersetzung der Muster in Strukturen des Syntaxbaums ist daher meist offensichtlich. Das Verbinden der Strukturen lässt sich mit dem in Abschnitt 5.4.3 vorgestellten Verfahren ebenfalls gut durchführen. Der größte Vorteil der Verwendung von Klassifikatoren wäre eine implizite Fehlerkorrektur durch Betrachtung eines größeren Raumes möglicher Muster. Dabei können auch Spezialfälle gefunden werden, die durch eine menschliche Analyse nicht ersichtlich sind. Dies kann aber auch durch eine gezielte Korrektur des Syntaxbaumes in

späteren Schritten erreicht werden. Erst wenn statische Analyseverfahren nicht ausreichen, wäre wieder eine nähere Betrachtung von maschinellen Lernverfahren zur Fehlerkorrektur anzuraten. Daher wird in dieser Arbeit der Ansatz des maschinellen Lernens nicht weiter betrachtet.

5.7 Zusammenfassung

In diesem Kapitel wurde die Problemstellung der „Zielsystemunabhängigen Quelltextsynthese aus natürlicher Sprache“ analysiert, in ihre Teilprobleme zerlegt und eine Lösung entworfen. Als wichtigste Teilprobleme wurden die Synthese eines abstrakten Syntaxbaumes und die Übersetzung dieses Baumes in Quelltext identifiziert. Es wurden Verfahren entworfen, die über Mustererkenner Strukturen des Syntaxbaumes erzeugen und durch darauf folgendes iteratives Verknüpfen der Minimalkernstrukturen den vollständigen Syntaxbaum aufbauen. Dazu wurde eine Menge von Syntaxbaum-Strukturen ausgewählt und beschrieben. So konnte eine Komplexität des erzeugten Quelltextes erreicht werden, die der imperativen Programmierung gleicht. Auch ein Ansatz für einfache Parallelität wurde aufgezeigt. Bei der Erzeugung des Quelltextes wurde auf die Unabhängigkeit des Zielsystems geachtet. Dazu wird der erzeugte abstrakte Syntaxbaum zuerst durch einen Besucher in Quelltext übersetzt und darauf mit einem „Schablonen“-Systems ins Zielssytem integriert. Diese Separierung erlaubt die Wiederverwendung von Teilen unabhängig des Zielsystems und ermöglicht eine einfache Erweiterbarkeit. Schwierigkeiten der vorgestellten Lösung beruhen im Allgemeinen auf Mehrdeutigkeiten und Folgefehlern, da sie den Abschluss der PARSE-Verarbeitung darstellt.

6 Implementierung

In diesem Kapitel wird die in Kapitel 5 beschriebene Lösung zum Problem der „Programmierung in natürlicher Sprache“ implementiert. Die Werkzeuge basieren auf der Kapitel 3 beschriebene PARSE-Rahmenarchitektur und werden in diese eingebaut.

Zuerst wird in Abschnitt 6.1 die Organisation der Projekts beschrieben. Die in dieser Arbeit entworfenen Werkzeuge stellen den Abschluss der Verarbeitung von PARSE dar. Sie lassen sich daher in zwei Bereiche einsortieren. Die Synthese des abstrakten Syntaxbaumes in Abschnitt 6.2 ist der Hauptverarbeitung von PARSE zuzuschreiben. Sobald der abstrakte Syntaxbaum synthetisiert wurde, ist die Hauptverarbeitung abgeschlossen. Die Nachverarbeitung in Abschnitt 6.3 beschreibt die Übersetzung des Syntaxbaumes in Quelltext. Die dafür zuständigen Werkzeuge sind die Stufen des Nachverarbeitungsfließbandes. Hier wird der abstrakte Syntaxbaum aus dem PARSE-Graphen isoliert, über Besucher in Quelltext übersetzt und anschließend durch Schablonen-Systeme zur Ausführung injiziert.

6.1 Projektorganisation

Für die Organisation der Implementierung des Projektes wurde Maven verwendet. Der Aufbau der Projektstruktur ist in Abbildung 6.1 dargestellt. Die oberste Ebene stellt das Projekt `S4C` dar. Der Name S4C steht für „*syntax tree based **s**ynthesis of **s**ystem independent **s**ource **c**ode*“ [Syntaxbaumbasierte Synthese von Systemunabhängigem Quelltext]. Es fasst alle implementierten Module und Werkzeuge zusammen. Das Projekt besteht aus drei Hauptmodulen: Das Modul `AST-API` wird von allen anderen Modulen eingebunden. Es enthält die benötigten Schnittstellen und Konstanten. Die Schnittstellen erlauben die Erweiterbarkeit und Austauschbarkeit der Werkzeuge. Die Konstanten sind Bezeichner für Attribute, Knoten- und Kantentypen. Auch befindet sich hier eine eigene Implementierung eines abstrakten Syntaxbaumes und Hilfsklassen. Die beiden Untermodule `ASTSynth` und `PostPL` beinhalten die Werkzeuge. Das Hauptmodul `ASTSynth` ist dafür zuständig den abstrakten Syntaxbaum innerhalb des PARSE-Graphen aufzubauen. Der Grundaufbau wird in `ASTSynth-Core` implementiert. Die Implementierungen der Mustererkenner befinden sich in dem Untermodul `ASTPatterns`. Sie untersuchen den PARSE-Graphen und erzeugen die in Abschnitt 5.4.2 beschriebenen Strukturen des Syntaxbaumes (siehe Definition 5.1). Die Verbindungen zwischen den Strukturen werden über Module in `ASTParentConfidenceCalculators` aufgebaut. Beide Module werden in `ASTSynth-Core` verwendet. Eine detailliertere Betrachtung der Synthese des Syntaxbaumes findet in Abschnitt 6.2 statt. Das in Abschnitt 6.3 erklärte dritte Hauptmodul `PostPL` entspricht dem

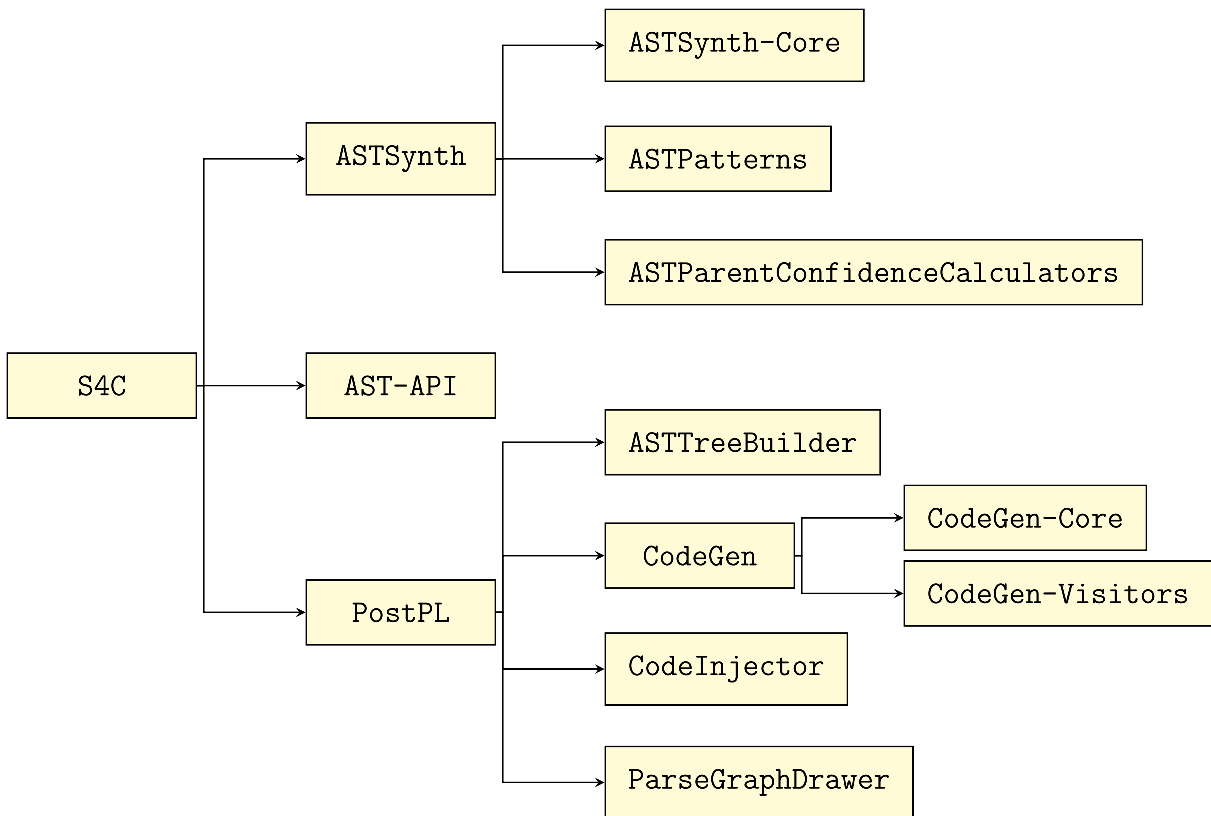


Abbildung 6.1: Aufbau der Maven-Projektstruktur

Nachverarbeitungsfließband. Die einzelnen Stufen des Fließbandes sind die Untermodule `ASTTreeBuilder`, `CodeGen` und `CodeInjector`. Zuerst isoliert `ASTTreeBuilder` den abstrakten Syntaxbaum aus dem PARSE-Graphen. Dieser wird dann in der Stufe `CodeGen` über Besucher in Quelltext übersetzt. Dieses Modul besteht aus zwei Untermodulen: In `CodeGen-Core` wird der Syntaxbaum über Besucher aus `CodeGen-Visitors` abgelaufen. Der erzeugte Quelltext wird in `CodeInjector` in eine entsprechende Datei injiziert. Die Stufe `ParseGraphDrawer` ist unabhängig von den anderen Stufen und erlaubt eine grafische Darstellung des PARSE-Graphen.

6.2 Synthese des abstrakten Syntaxbaums

Die Synthese des abstrakten Syntaxbaumes findet in der Hauptverarbeitung von PARSE statt. Hierzu wurde der Agent `ASTSynth` implementiert. `ASTSynth` erweitert den PARSE-Graphen um Knoten und Kanten, die einen abstrakten Syntaxbaum darstellen. Dies wird über Heuristiken und Muster erreicht.

6.2.1 Ablauf

Der Agent `ASTSynth` synthetisiert den Syntaxbaum in drei Schritten. Im ersten Schritt findet eine Mustersuche statt, bei der die in Abschnitt 5.4.2 beschriebenen Strukturen des Syntaxbaumes extrahiert werden. Jede Struktur wird hierbei von einem eigenen Mustererkenner gefunden und in den PARSE-Graphen geschrieben. Die Modularität wird durch die in Abbildung 6.2 dargestellte Schnittstelle *`IASTPattern`* und die `ServiceLoader`-Architektur von Java erreicht. Die Schnittstelle *`IASTPattern`* besitzt vier Methoden: Die erste Methode `getBaseType()` gibt den Bezeichner der Basisstruktur, d.h. den Knotentypen, zurück. Der Knotentyp dieser Basisstruktur und weiterer Strukturen muss vor der Verwendung im PARSE-Graphen deklariert werden. Mit `init(...)` wird der Mustererkenner

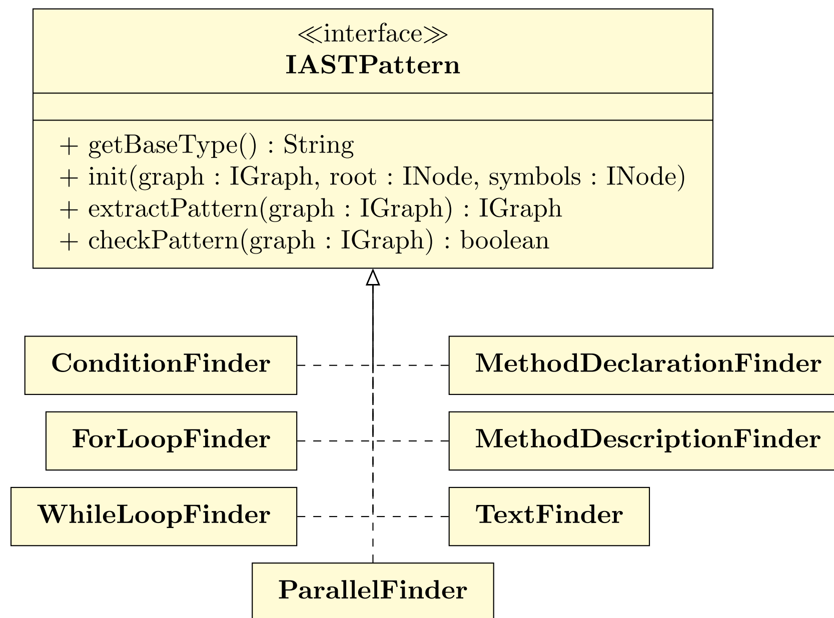


Abbildung 6.2: *IASTPattern*-Schnittstelle zur Implementierung der Mustererkenner.

initialisiert, d.h die Knotentypen des PARSE-Graphen deklariert. Nach der Initialisierung ruft `ASTSynth` die Methode `extractPattern(...)` sequentiell für jeden Mustererkenner auf. Der aufgerufene Mustererkenner analysiert den PARSE-Graphen und erzeugt die gefundenen Strukturen. Nachdem der Syntaxbaum vollständig synthetisiert wurde, überprüft der Mustererkenner mit `checkPattern(...)` den Baum auf Fehler. Für jeden gefundenen Fehler wird ein Fehlerknoten im PARSE-Graphen erstellt.

Mit der Ausführung der Mustererkenner wurden alle Strukturen aus dem PARSE-Graphen extrahiert. Die Strukturen sind aber noch nicht verbunden. Dies geschieht, indem für alle Strukturen der direkte Vorgänger gefunden wird. Von besonderer Bedeutung ist hierbei die *IASTParentConfidenceCalculator*-Schnittstelle, welche in Abbildung 6.3 abgebildet ist. Darüber wird ein Algorithmus implementiert, welcher für jede Struktur des Baumes mögliche Vorgängerknoten sammelt und bewertet. Die Tabelle die für eine Struktur mögliche Eltern mit ihrer Bewertung enthält, wird als Konfidenzmatrix bezeichnet. Die Initialisierung des Algorithmus findet in der Methode `init()` statt. Die Methode `findPossibleParentAndCalculateConfidence()` gibt für eine Struktur die berechnete Konfidenzmatrix zurück. Anhand der Bewertung in der Konfidenzmatrix wird für jede Struktur in `ASTSynth` die am besten bewertete Struktur als Elter ausgewählt und beide Strukturen verbunden. Da die Wurzel alle Strukturen des Baumes verbindet, ist sie nicht Teil der Konfidenzmatrix. Die Wurzel wird nur dann ausgewählt, wenn sich kein anderer Vorgänger finden lässt. Das stellt sicher, dass der Baum immer alle gefunden Strukturen enthält. Wurden alle Strukturen bis zur Wurzel verbunden, so ist diese Phase abgeschlossen.

Bei der Verbindung der Strukturen zu einem Syntaxbaum wurde keine Ausführungsreihenfolge festgelegt. Das bedeutet, dass bei der späteren Übersetzung des Baumes die Reihenfolge der Anweisungen nicht definiert ist. Dies führt bei Befehlsbeschreibungen, die sequentielle Instruktionen beschreiben zu Fehlern. Daher müssen die Strukturen, nachdem der Syntaxbaum vollständig aufgebaut wurde, in die korrekte Ausführungsreihenfolge gebracht werden. Dazu wurde den Blättern des Syntaxbaumes eine Instruktionsnummer ihres Musterbereichs (siehe Definition 5.2) zugewiesen. Strukturen können rekursiv ihre Position in der Ausführungsreihenfolge feststellen, indem sie die Position ihrer Kinder mitteln. Nachdem alle Strukturen ihre Position in der Ausführungsreihenfolge bestimmt haben, ist der Verarbeitung des Syntaxbaum in `ASTSynth` abgeschlossen. Der abstrakte Syntaxbaum

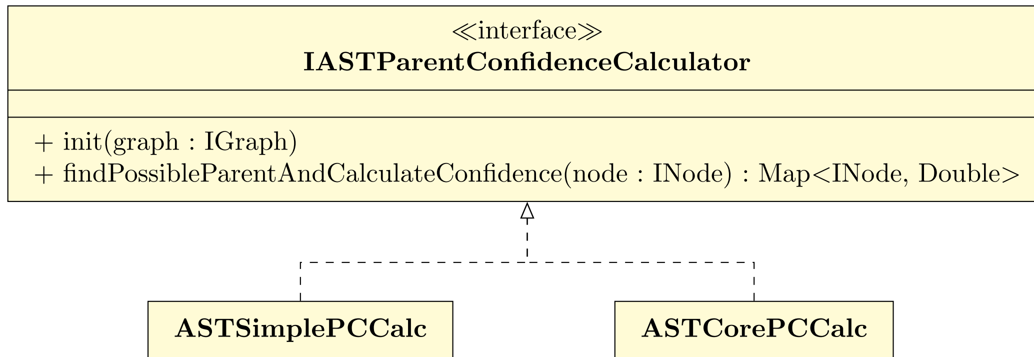


Abbildung 6.3: *IASTParentConfidenceCalculator*-Schnittstelle zur Implementierung der Strukturverbindungsalgorithmen.

ist nun vollständig im PARSE-Graphen aufgebaut und kann dem Nachverarbeitungsfließband übergeben werden.

6.2.2 Mustererkennung

Der Syntaxbaum wird im PARSE-Graphen synthetisiert. Die Strukturen des Syntaxbaums werden im PARSE-Graphen durch Knoten und Kanten repräsentiert. Am Beispiel der in Abbildung 6.4 dargestellten Schleife, soll der grundsätzliche Aufbau der extrahierten Strukturen beschrieben werden. Eine Struktur besteht aus aus einer Basisstruktur und mehreren Teilstrukturen. Bei der Schleife entsprechen die Teilstrukturen dem Kopf und Rumpf der Schleife. Im PARSE-Syntaxbaum wird für die Basisstruktur und jede Teilstruktur ein eigener Knoten mit entsprechendem Knotentyp erzeugt. Zusätzlich enthält jeder Knoten ein Attribut `position`, welches der Position der Struktur in der Ausführungsreihenfolge entspricht. Durch eine Sortierung der Strukturen anhand des `position`-Attribut lassen sich die Strukturen in die Ausführungsreihenfolge ordnen. Die Teilstrukturen werden durch die Basisstruktur mit Kanten vom Typ `AST-CIntern` gebündelt. Der Vorgänger der Struktur ist über die `AST-Connection`-Kante der Basisstruktur erreichbar. Die Teilstrukturen werden entsprechend über die `AST-Connection`-Kante mit der Basisstruktur ihres Nachfolgers verbunden. Jede Teilstruktur besitzt einen Satz von Verbindungen vom Typ `AST-Pointer`, die auf die Token des Musterbereichs (siehe Definition 5.2) verweisen. Die Musterbereiche der Basisstruktur werden durch die Teilstrukturen impliziert und sind nicht explizit modelliert.

6.2.2.1 Wurzel des Syntaxbaums

Die Wurzel des abstrakten Syntaxbaum besteht aus einem einzigen Knoten vom Typ `AST-Root` und ist in Abbildung 6.5 dargestellt. Anders als die restlichen Strukturen wird die Wurzel direkt von `ASTSynth` in den PARSE-Graphen eingetragen. So erspart man sich den Aufbau einer Hierarchie von Mustererkennern, da die Wurzel als einzige Struktur von anderen Mustererkennern bearbeitet wird (siehe Abschnitt 5.4.1). Das Attribut `newMethod` der Wurzel wird vom Mustererkenner `MethodDeclarationFinder` beschrieben. `MethodDeclarationFinder` extrahiert bei einer Lehrsequenz, die von `MethodSynthesizer` (siehe Abschnitt 3.2.2.2) erzeugte Methodensignatur. Dazu erstellt `MethodSynthesizer` einen Baum mit `commandMapper` als Wurzel. Die Methodensignatur befindet sich auf der `declaration`-Seite des `commandMapper`-Baumes. Bei einer Befehlssequenz bleibt das Attribut unbeschrieben. Von der Wurzel lassen sich alle anderen Strukturen erreichen.

6.2.2.2 Blätter des Syntaxbaums

Die Blätter des PARSE-Syntaxbaumes entsprechen Aktionsstrukturen, d.h. sie spiegeln Aktionen im Syntaxbaum wieder. Als Blätter gibt es die in Abbildung 6.6 dargestellten

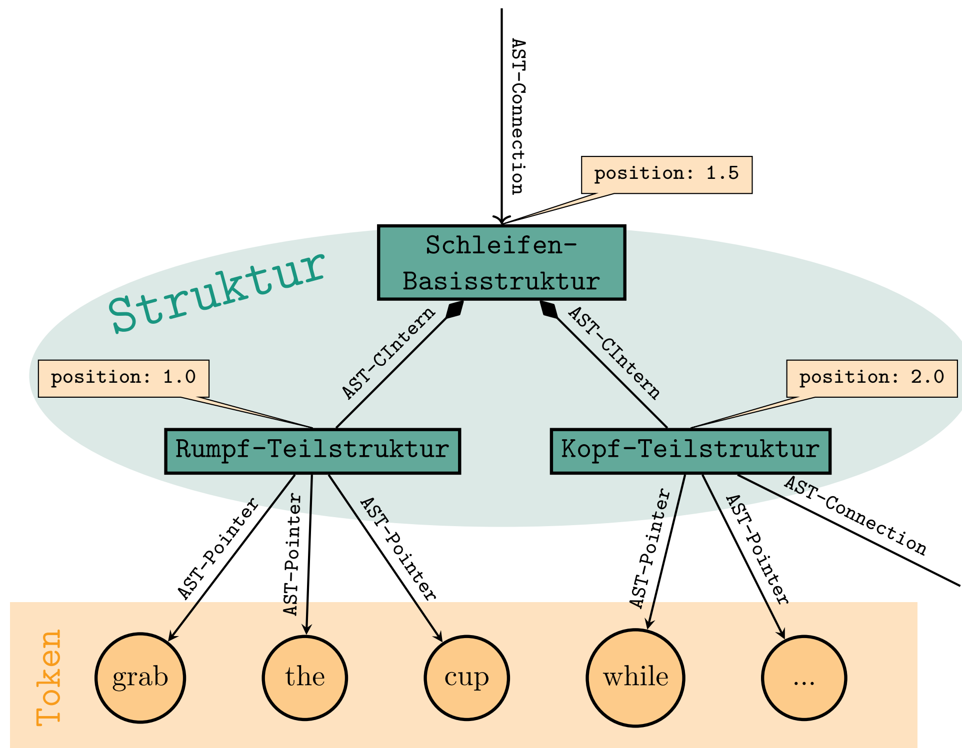


Abbildung 6.4: Grundsätzlicher Aufbau einer Struktur des Syntaxbaumes im PARSE-Graphen

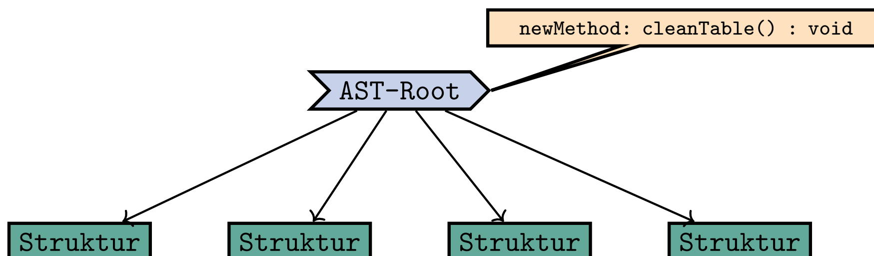


Abbildung 6.5: Wurzel-Struktur des PARSE-Syntaxbaumes

Strukturen. Während `AST-Method` für Methoden basierten Quelltext verwendet wird, ist `AST-Text` dazu gedacht textbasiert Pseudo-Quelltext zu erzeugen. Als Blätter besitzen beide weder Teil- noch Kindstrukturen und bestehen somit aus einem einzigen Knoten. `AST-Text` und `AST-Method` sind äquivalent und übernehmen beide die Instuktionsnummer ihrer Instruktion als `position`-Attribut. Sie lassen sich deshalb austauschen. Eine Kombination würde aber dazu führen, dass `AST-Text` und `AST-Method` beide die Blätter des Syntaxbaums darstellen. Sie würden sich also gegenseitig blockieren.

Zur Repräsentation von Aktionen in Form einer Methode wird der Knoten vom Typ `AST-Method` genutzt. Das Modul `MethodFinder` ist dafür zuständig die vom `MethodSynthesizer` gefunden Methoden in `AST-Method` zu übersetzen. `MethodSynthesizer` erstellt einen Baum mit `commandMapper` als Wurzel. Der `commandMapper`-Baum wird vom Mustererkenner auf der `description`-Seite abgelaufen und die Methode extrahiert. Das Attribut `method` dient zum Speichern der Methodensignatur der extrahierten Methode. `MethodSynthesizer` findet für jede Aktion einer Befehlsbeschreibung in der Regel mehrere alternative Methoden und bewertet diese. Für die am besten bewertete Methode einer Aktion wird ein `AST-Method`-Knoten erzeugt. Als Musterbereich werden die vom `MethodSynthesizer` fest-

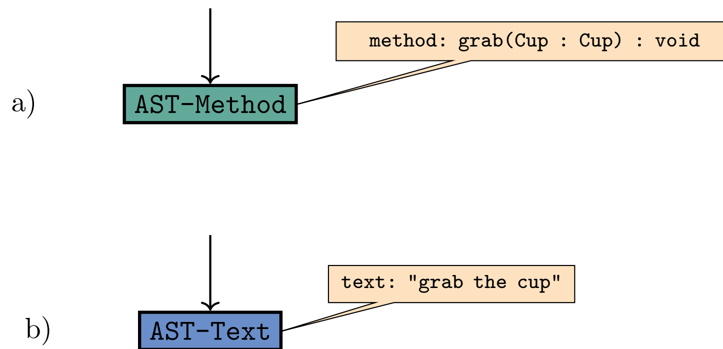


Abbildung 6.6: (a) Methoden- und (b) Textstruktur im PARSE-Syntaxgraphen

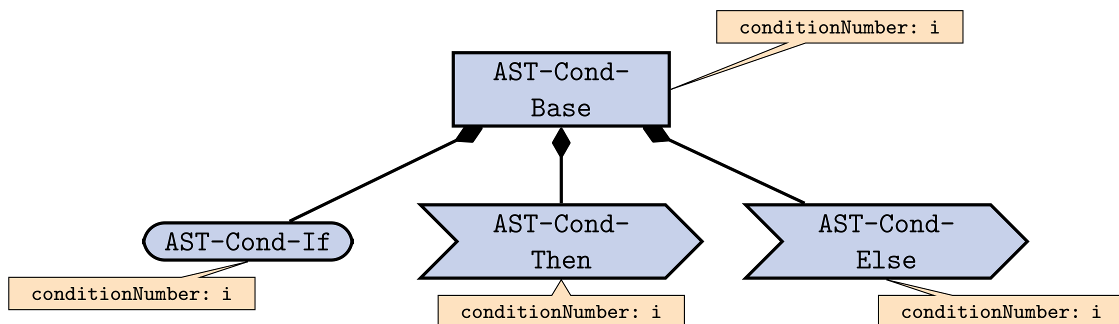


Abbildung 6.7: Verzweigungsstruktur im PARSE-Syntaxbaum

gelegten Token übernommen. Da `MethodSynthesizer` die Parameter der Methode bereits korrekt anordnet, wird diese Anordnung übernommen.

Da `MethodSynthesizer` anfangs nicht zur Verfügung stand, wurde der Mustererkenner `TextFinder` implementiert. `TextFinder` erzeugt `AST-Text`-Knoten, die äquivalent zu `AST-Method` sind, aber auf natürlicher Sprache basieren. `AST-Text` speichert im `text`-Attribut Text der Eingabe, welcher sich aus Token der gleichen Instruktionsnummer ergibt. Der Musterbereich entspricht daher der Instruktion. Durch `AST-Text` lässt sich textbasiert Pseudo-Quelltext erzeugen.

6.2.2.3 Verzweigungen

Zur Steuerung des Programmes werden Kontrollstrukturen (siehe Abschnitt 5.4.2.3) benötigt. Eine Verzweigung des Programmablaufs durch Wenn-Dann-Sonst-Sätze wird im Syntaxbaum durch die Verzweigungsstruktur repräsentiert. Der Mustererkenner `ConditionFinder` extrahiert Verzweigungen anhand der `conditionNumber` des Token-Knoten. Token mit der gleichen `conditionNumber` implizieren die gleiche Verzweigung. Da Token so nur einer `conditionNumber` zugewiesen werden können, sind verschachtelte Verzweigungen nicht möglich. Für jede Verzweigung werden zuerst die vier in Abbildung 6.7 dargestellten Knoten erstellt, welche zur eindeutigen Identifizierung mit der `conditionNumber` markiert werden. Der Knoten vom Typ `AST-Cond-Base` bildet hierbei die Basisstruktur. Als Attribut besitzt jeder Knoten der Verzweigung eine `conditionNumber`, welche es erlaubt die Verzweigung zu identifizieren. Die Teilstrukturen der Verzweigung sind durch die Kante `AST-CIntern` mit der Basisstruktur verbunden. Die Token-Knoten die Teil einer Verzweigung sind, besitzen ein `commandType`-Attribut, welches den Teil der Verzweigung angibt (`IF`, `THEN` und `ELSE`). Anhand des `commandType`-Attribut der Token wird den Teilstrukturen der entsprechende Musterbereich zugewiesen. Die Teilstruktur mit Knotentyp `AST-Cond-If` entspricht der Bedingung und der Musterbereich verweist auf die Bedingung

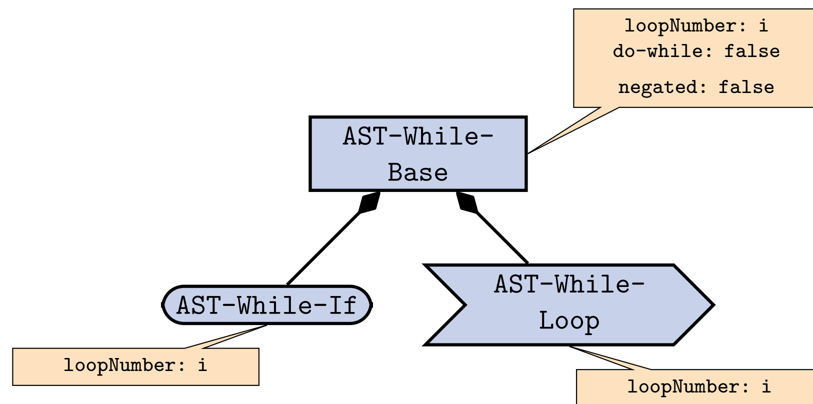


Abbildung 6.8: While-Schleifenstruktur im PARSE-Syntaxbaum

beschreibenden Token. Der Knoten vom Typ `AST-Cond-Then` markiert die Dann-Sequenz und der `AST-Cond-Else`-Knoten die Sonst-Sequenz. Die Kinder beider Sequenzen werden durch das Attribut `position` in ihre Ausführungsreihenfolge gebracht. Der `AST-Cond-Else`-Knoten ist zwar nicht immer nötig, wird aber automatisch immer erzeugt. Dies führt dazu, dass dieser Knoten teilweise besonders behandelt werden muss.

6.2.2.4 Schleifen

Als weitere Kontrollstrukturen besitzen Schleifen ebenfalls eigene Knoten und unabhängig von der Schleifenart einige Gemeinsamkeiten. So befinden sich beide Mustererkenner zum Erkennen von Schleifen im Modul `LoopFinder`. `WhileLoopFinder` ist ein Mustererkenner für While-Schleifen und Do-While-Schleifen. For-Schleifen werden von `ForLoopFinder` extrahiert. Beide Klassen erkennen die ihnen zugehörigen Schleifentypen und erstellen entsprechende Basis- und Teilstrukturen als Knoten im PARSE-Graphen. Jeder der Knoten besitzt ein `loopNumber`-Attribut, um die Schleife zu identifizieren. Zusätzlich wird von der Basisstruktur auf den von PARSE erzeugten `loop`-Knoten mit einer `AST-Base-Pointer`-Kante verwiesen. Dabei werden auch die relevanten Attribute des `loop`-Knotens übernommen.

Die in Abbildung 6.8 dargestellte While-Schleife besteht aus drei Knoten. `AST-While-Base` ist die Basisstruktur der Schleife. Die Teilstrukturen teilen sich in die Knotentypen `AST-While-If` und `AST-While-Loop` auf. `AST-While-If` bildet hier mit der Bedingung den Kopf der Schleife und übernimmt die `loopCondition`-Kanten des `loop`-Knoten als Musterbereich. Der Rumpf wird durch die Sequenz `AST-While-Loop` dargestellt und der Musterbereich entspricht den von `dependentLoopAction` referenzierten Token. Um zwischen der Do-While- und der While-Schleife zu unterscheiden, besitzt die Basisstruktur das Attribut `do-while`. Zusätzlich gibt es das Attribut `negated`, welches die Bedingung der Schleife negiert. Dies erlaubt Wiederholungsbedingungen wie „*repeat this while the dishwasher is empty*“ von Abbruchbedingungen wie „*repeat this until the dishwasher is empty*“ zu unterscheiden. Zur Erkennung einer Abbruchbedingung dient das Signalwort „*until*“. Die Teilstrukturen sind durch `AST-CIntern` mit der Basisstruktur verbunden.

Das Modul `LoopFinder` enthält einen zweiten Mustererkenner `ForLoopFinder`, der vom Aufbau `WhileLoopFinder` ähnelt und für For-Schleifen zuständig ist. Die For-Schleife besteht anders als die While-Schleife nur aus zwei Knoten und ist in Abbildung 6.9 dargestellt. Wie beim `WhileLoopFinder` werden die Token der von `dependentLoopAction` als Musterbereich der Teilstruktur `AST-For-Loop` übernommen. Der Unterschied besteht darin, dass die Anzahl an Iterationen als Zählwort, d.h. Wörter wie „*twice*“ und „*four times*“, im Loop-Knoten übernommen ist. Das Zählwort wird in eine Zahl übersetzt und als `iterations`-Attribut in den Knoten `AST-For-Base` geschrieben. Dafür wird eine Tabelle genutzt,

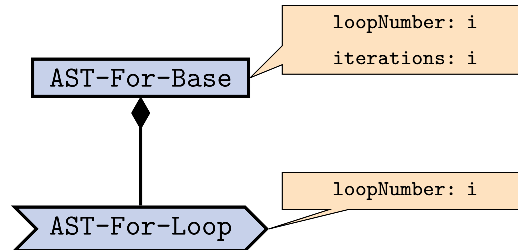


Abbildung 6.9: For-Schleifenstruktur im PARSE-Syntaxbaum

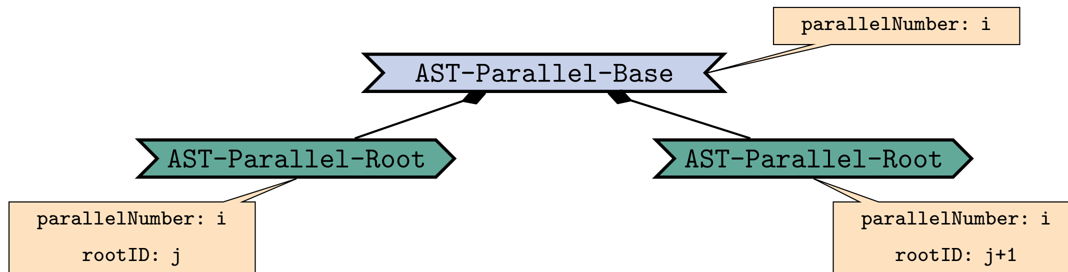


Abbildung 6.10: Parallelblock im PARSE-Syntaxbaum

die Zählworte wie „*twice*“ oder „*four times*“ in die Anzahl Iterationen übersetzt. Dadurch ist kein Knoten für die Bedingungs-Teilstruktur notwendig. Um sicherzustellen, dass es bei der Übersetzung von verschachtelten For-Schleife in Quelltext identische Iterationsvariablen nicht vorkommen, wird jede For-Schleife nummeriert. Der Name der Iterationsvariabel enthält die Nummer der Schleife (`loopNumber`-Attribut), um sicherzustellen, dass jede For-Schleife eine einzigartige Iterationsvariable besitzt. Der Rumpf der For-Schleife wird mit den Knotentyp `AST-For-Loop` als Sequenz dargestellt.

6.2.2.5 Parallelität

Als Repräsentation des Parallelismus im PARSE-Syntaxbaum wurde der Parallelblock in Abschnitt 5.4.2.4 definiert. Im Mustererkenner `ParallelFinder` wird der in Abbildung 6.10 dargestellte Parallelblock als Knoten vom Typ `AST-Parallel-Base` erzeugt. Dazu wird die Existenz des Knotens `concurrentAction` verwendet. Die Kante `dependentconcurrentAction` von `concurrentAction` verweist auf die parallelen Sektionen, d.h. die zeitgleich auszuführenden Instruktionen. Für jede parallele Sektion wird ein Knoten vom Typ `AST-Parallel-Root`, d.h. Sequenz-Teilstrukturen, erzeugt. Die Parallelstruktur kann mehrere Sequenz-Teilstrukturen besitzen, obwohl `concurrentAction` nur zwei parallele Sektionen besitzt. Dies erlaubt die zeitgleiche Ausführung einer beliebigen Anzahl von Befehlssequenzen, falls der Agent zum Erkennen von Parallelität entsprechend erweitert wird. Weiterhin werden in der Kante `dependentconcurrentAction` die gefundenen parallelen Sektionen nummeriert. Die Sequenz-Teilstrukturen übernehmen diese Nummerierung im `rootID`-Attribut, um zwischen den parallelen Sektionen unterscheiden zu können.

6.2.3 Verbindung der Strukturen

Nachdem alle Strukturen erzeugt wurden, wird ein abstrakter Syntaxbaum durch Verbindung der Strukturen aufgebaut. Dies geschieht ebenfalls in `ASTSynth`. Um die Erweiterbarkeit zu gewährleisten, können über die Schnittstelle *`IASTParentConfidenceCalculator`* Strukturverbindungsalgorithmen implementiert werden. In dieser Arbeit wurden zwei Algorithmen implementiert: `ASTSimplePCCalc` ist ein einfaches Verfahren welches für den Prototyp verwendet wurde. Hierbei werden zuerst für jede Teil- und Basisstruktur andere

Strukturen gesammelt, welche auf Teile des gleichen Musterbereichs (siehe Definition 5.2) verweisen. Die gefundenen Strukturen dürfen dabei weder ein anderes Blatt, noch die selbe Struktur sein. Als Vorgänger wurde die Struktur gewählt, die die Überlappung beider Musterbereich am stärksten abdeckt (siehe Gleichung 5.1).

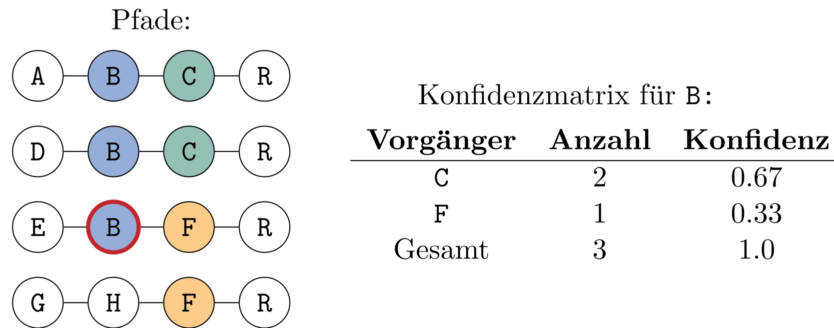
Eine vereinfachte Form des in Abschnitt 5.4.3 beschriebenen Verfahren wurde in `ASTCorePCCalc` implementiert. Diese Vereinfachung ist möglich, weil die Syntaxbäume in der Regel eine geringe Tiefe aufweisen. Hierbei werden bei der Initialisierung für alle Methoden- und Bedingungsstrukturen, d.h. Blätter Pfade erstellt und gespeichert. `AST-Text` oder `AST-Method` sind immer Blätter des Syntaxbaumes und ihr Musterbereich entspricht einer einzelnen Instruktion. Der Musterbereich der Wurzel `AST-Root` umschließt alle Instruktionen und damit die gesamte Eingabe. Damit sind die beiden Extrema der Musterbereiche aller Pfade bekannt. Die Position anderer Strukturen im Pfad lässt sich durch die Ähnlichkeit ihrer Musterbereiche zu einem der Extrema berechnen. Je ähnlicher der Musterbereich zu dem Musterbereich der Blattstruktur ist, desto näher am Blatt ist seine Struktur. Da die Wurzel für alle Strukturen identisch ist, ist die Berechnung der Ähnlichkeit zum Blatt ausreichend. Nur für Strukturen, deren Musterbereich sich mit dem Musterbereich des Blattes überschneidet, wird angenommen, dass sie Teil des Pfades sind. So soll vermieden werden, die Ähnlichkeit für Strukturen zu berechnen, die nicht Teil des Pfades sind. Zur Berechnung der Ähnlichkeit wird die Levenshtein-Distanz [Lev] verwendet. Die Levenshtein-Distanz berechnet in der Regel die Ähnlichkeit zwischen zwei Strings. Dazu wird die minimale Anzahl an Buchstaben gezählt, die eingefügt, gelöscht oder ersetzt werden müssen, um die beiden Strings ineinander umzuwandeln. Wird ein Token des Musterbereichs als Symbol, d.h. Buchstabe interpretiert, so kann die Levenshtein-Distanz dazu verwendet werden die Ähnlichkeit zwischen zwei Musterbereichen zu berechnen. Es wird also gezählt, wie viele Token man einfügen, löschen oder ersetzen müsste, um den gleichen Musterbereich zu erhalten. Ist die Levenshtein-Distanz aller Strukturen des Pfades zum Blatt bekannt, so kann der Pfad durch eine Sortierung anhand der Levenshtein-Distanz gefunden werden. Hierbei wird von Teilstrukturen mit gleicher Basisstruktur, nur die zum Blatt ähnlichste Struktur gewählt. Die restlichen Teilstrukturen werden aus dem Pfad entfernt. Dadurch wird verhindert, dass beispielsweise Kopf und Rump der gleichen Struktur Teil des selben Pfads sind.

Wurden alle Pfade gefunden, so lässt sie die Konfidenzmatrix für eine Struktur berechnen. In Abbildung 6.11 ist dies beispielhaft dargestellt. Dort soll die Konfidenzmatrix für die Struktur `B` berechnet werden. Dafür werden zuerst für alle Pfade die möglichen direkten Vorgänger (grün und orange) von `B` bestimmt. Je häufiger eine Elter-Kind-Beziehung in den Pfaden existiert, desto sicherer ist die Existenz dieser Verbindung. Die Konfidenz für die Verbindung zum Vorgänger einer Struktur berechnet sich deshalb durch die gemittelte Häufigkeit der Eltern-Kind-Beziehung zwischen beiden Strukturen:

$$\text{Konfidenz(K,V)} = \frac{\sum \big(\text{V ist direkter Vorgänger von K in Pfad}\big)}{\sum \big(\text{Pfad enthält K}\big)} \tag{6.1}$$

6.2.4 Ausführungsreihenfolge von Sequenzen

Mit der Extraktion und Verbindung aller Strukturen ist ein abstrakter Syntaxbaum aufgebaut. Beim Übersetzen dieses Baumes in Quelltext ist aber die Ausführungsreihenfolge der Methoden in Sequenzen nicht definiert. Deshalb muss noch allen Knoten eine Ausführungsposition im Verhältnis zu ihren Geschwistern zugewiesen werden. Strukturen übernehmen daher die Instruktionsnummer Token ihrer Musterbereiche, um eine Ausführungsreihenfolge zu etablieren. Die Instruktionsnummern der Token bestimmen also die Ausführungsreihenfolge, d.h. die Position der Strukturen innerhalb der Sequenz. Dies erlaubt eine



Vorgänger	Anzahl	Konfidenz
C	2	0.67
F	1	0.33
Gesamt	3	1.0

Abbildung 6.11: Beispiel zur Berechnung der Konfidenzmatrix mit fehlerhafter Struktur (rot umrandet). Die restlichen Konfidenzmatrizen sind eindeutig.

Festlegung der Ausführungsreihenfolge unabhängig des aufgebauten Syntaxbaumes. Jeder Knoten besitzt zur Speicherung seiner Position in der Ausführungsreihenfolge ein Attribut `position`. Der Wert des Attributs spiegelt die Reihenfolge wieder. Werden zwei Strukturen bezüglich dem Attribut `position` verglichen, so soll die Struktur mit dem niedrigeren `position`-Wert zuerst ausgeführt werden. Dadurch lässt sich die Ausführungsreihenfolge zwischen mehreren Strukturen durch eine Sortierung anhand des `position`-Attribut finden. Bei der Festlegung der Position ist zu beachten, dass die Positionwerte übernommen werden. Das heißt, dass für alle Strukturen das selbe Verfahren verwendet wird. Ein Problem bei der Festlegung der Ausführungsreihenfolge sind fehlerhafte Musterbereiche. Überlappende Musterbereiche können dazu führen, dass zwei Strukturen den selben Positionswert übernehmen. Zur Festlegung der Position wurden zwei Funktionen implementiert. Die erste Funktion wurde im Prototypen verwendet und übernahm einfach die niedrigste Position der Kinder. Bei der Analyse wurde festgestellt, dass dies unzureichend ist wie am Beispiel 6.2.1 zusehen ist. Daher wurde der Durchschnitt der Positionen der Kinder als zweite Funktion implementiert. Statt der Instruktionsnummern des Musterbereichs wäre es möglich die Position der Token des Musterbereichs zu verwenden. Dies kann bei einer fehlerhaften Zerteilung eine bessere Bestimmung des `position`-Wertes erlauben, da die Verschiebung um ein Token geringere Auswirkung hat.

Da die Aktionsstruktur keine Kinder besitzt, wird ein alternatives Vorgehen zur Bestimmung der Position verwendet. So legt der Agent `MethodSynthesizer` bereits eine Ausführungsreihenfolge für Methoden fest. Knoten vom Typ `AST-Method` übernehmen diese Reihenfolge als Position. Die restlichen Strukturen nutzen rekursiv ihre Kinder, um ihre eigene Position zu bestimmen. So werden alle Strukturen des Syntaxbaumes in eine Ausführungsreihenfolge gebracht. Die Ausführungsreihenfolge ist so aber auch abhängig von aufgebauten Baum. Werden die Strukturen fehlerhaft verbunden, so hat dies auch Einfluss auf die Ausführungsreihenfolge. Vorteilhaft ist aber, dass Mehrdeutigkeiten beim Musterbereich nur noch in den Blättern auftreten können. Wurde Position der Blätter korrekt annotiert, so entstehen keine Fehler bei der Bestimmung der Ausführungsreihenfolge.

6.2.5 Analyse des Syntaxbaumes

In *`IASTPattern`* wurde die Methode `checkPattern()` eingebaut, die es erlaubt Strukturen auf grundsätzliche Korrektheit durch ihre Mustererkenner überprüfen zu lassen. Hier wird nur überprüft, dass verpflichtenden Strukturverbindungen, wie beispielsweise Aktionen als Kinder von Bedingungen, gesetzt sind. Ist dies nicht der Fall, so wird ein Fehlerknoten erstellt, der mit `AST-Error-Pointer` auf den fehlerhaften Knoten verweist. Der Fehlerknoten kann zwei Gefahrenstufen haben: Ein fehlerhafter Syntaxbaum wird mit der Stufe „Fehler“ annotiert. Dies bedeutet, dass eine Weiterverarbeitung des Baumes garantiert zu einem Fehler führt. Ein Beispiel hierfür ist die Voraussetzung, dass an

Beispiel 6.2.1: Fehlerhafte Ausführungsreihenfolge

Wahlverfahren	A	B	C
Minimum	1	1	3
Maximum	1	3	3
Mittelung	1	2	3

In diesem Beispiel führt eine Überlappung der Musterbereiche zu einer fehlerhaften Ausführungsreihenfolge. Die Strukturen `A`, `B` und `C` sind Aktionsstrukturen und daher Blätter des Syntaxbaumes. Der rot hervorgehobene Musterbereich der Struktur `B` ist fehlerhaft und zu groß. Er überschneidet sich damit mit den Musterbereichen der Struktur `A` und `C`. Die Verfahren „Minimum" und „Maximum" führen daher dazu, dass jeweils zwei Strukturen die gleiche Instruktionsnummer übernehmen. Sie haben damit die gleiche Position. Die korrekte Ausführungsreihenfolge lässt sich dadurch nicht feststellen. Die Wahl des kleinsten oder größten Instruktionsnummer eines Musterbereichs für die Struktur ist also ungeeignet. Eine einfache Möglichkeit dieses Problem zu vermeiden, ist die Zuweisung des Mittels aller Werte der Instruktionsnummern des Musterbereichs der Struktur als Position. Fehlerhafte Musterbereiche sind in der Regel zu groß. Um eine Zahl möglichst weit von den fehlerhaften Rändern des Musterbereichs zu übernehmen, sollte ein möglichst mittiger Positionswert gewählt werden. Dies kann durch die Mittelung oder auch den Median der möglichen Werte der Instruktionsnummern erreicht werden. Die Mittelung hat auch den Vorteil, dass eine reelle Zahl bestimmt wird. Dies erlaubt auch kleine Differenzen festzustellen.

die Bedingungs-Teilstruktur einer While-Schleife oder Verzweigung eine Methode anliegen muss. Die Fehlerstufe „Warnung" deutet hingegen auf einen problematischen Bereich hin. Dies kann dazu führen, dass der Baum zwar syntaktisch korrekt ist, aber von seiner Bedeutung fehlerhaft. Eine solche Warnung wird beispielsweise bei leeren Schleifenrümpfen oder Dann-Teilstrukturen von Verzweigungen gegeben. Neben der Gefahrenstufe besitzt der Knoten auch einen Fehlertyp und textuelle Fehlermeldung als Attribute. Der Dialogagent kann die Fehlerknoten aufgreifen, um eine Fehlerkorrektur durch den Nutzer zu veranlassen.

6.3 Nachverarbeitungsfließband

Die Übersetzung des abstrakten Syntaxbaumes in Quelltext erfolgt in der Nachverarbeitung. Diese ist als Fließband aufgebaut und besteht aus mehreren Stufen. Zuerst wird der synthetisierte Syntaxbaum in `ASTTreeBuilder` aus dem PARSE-Graphen extrahiert. Der extrahierte Baum wird danach in `CodeGen` durch Besucher in Quelltext übersetzt. Der erzeugte Quelltext wird in `CodeInjector` zum Schluss in eine Schablone injiziert.

Für die Nachverarbeitung dient `ASTPostPipelineData` als gemeinsame Datenstruktur. Sie speichert den in der Hauptverarbeitung entstandenen PARSE-Graphen, den extrahierten Syntaxbaum, den erzeugten Quelltext und bei einer Lehrsequenz auch die synthetisierte Methodensignatur.

6.3.1 ASTTreeBuilder

Der `ASTTreeBuilder` extrahiert den abstrakten Syntaxbaum aus dem PARSE-Graphen. Dazu durchläuft er den Syntaxbaum von der Wurzel aus und übersetzt jede Struktur des

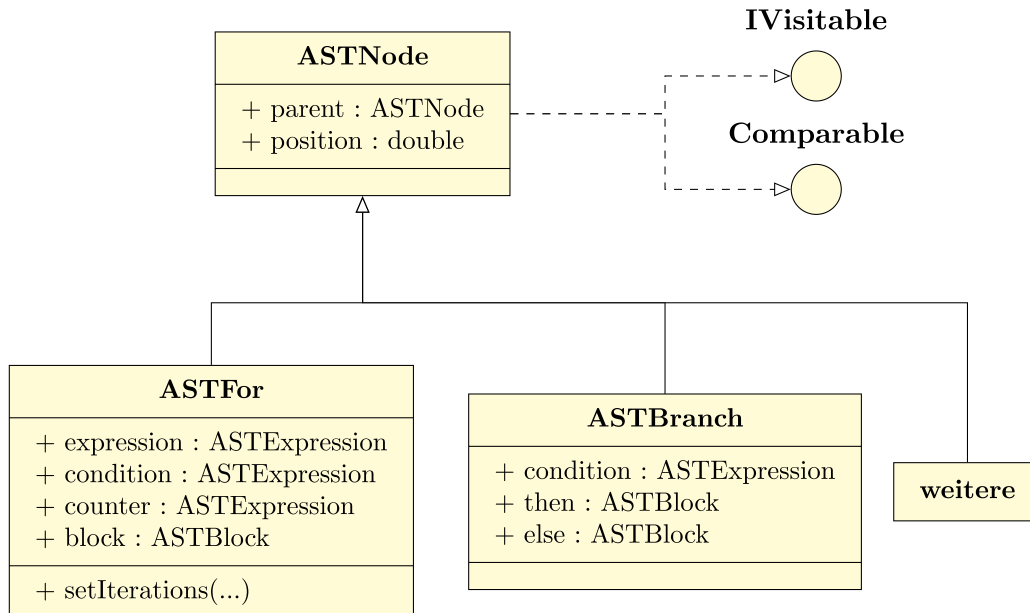


Abbildung 6.12: Die abstrakte Klasse `ASTNode` stellt die Grundlage für alle Knoten dar.

PARSE-Syntaxbaums in Knoten eines in Java implementierten abstrakten Syntaxbaumes. Da nicht mehr auf den PARSE-Graphen zugegriffen werden muss, erleichtert dies das Verarbeiten des Syntaxbaumes. Die in Abbildung 6.12 dargestellte Klasse `ASTNode` dient dabei als Grundlage für die Knoten des abstrakten Syntaxbaum. Sie implementiert die Schnittstelle *`IVisitable`* für die in Abschnitt 6.3.2 verwendeten Besucher. Das `parent`-Attribut referenziert den Elterknoten und erlaubt so den Baum aufzubauen. Das Attribut `position` spiegelt die Ausführungsreihenfolge einer Sequenz wieder. Dazu wird die Schnittstelle *`Comparable`* implementiert, die es erlaubt die Knoten nach ihrer Position zu sortieren. Alle anderen Knoten erben von `ASTNode`. Es gibt eine Vielzahl von Knoten die jeweils eine Struktur oder Teilstruktur implementieren. Sie sind in Tabelle 6.1 aufgezählt.

6.3.2 CodeGen

Das Modul `CodeGen` ist für die Übersetzung des abstrakten Syntaxbaum in Quelltext zuständig. Es besteht aus zwei Untermodulen: Die implementierten Besucher befinden sich im Untermodul `CodeGen-Visitors`. Durch die Verwendung von Besuchern ist es leicht möglich den selben Syntaxbaum in unterschiedliche Programmiersprachen zu übersetzen. In Abbildung 6.13 ist die Schnittstelle *`IVisitor`* mit ihren konkreten Implementierungen dargestellt. Die Schnittstelle besitzt eine Methode *`getID()`* zum Abrufen einer ID. Dies erlaubt den zu verwendenden Besucher in der Konfigurationsdatei einzustellen. Die Übersetzung der Strukturen des Syntaxbaums in Quelltext geschieht über die *`toCode(...)`*-Methoden. Für jede Struktur des Syntaxbaumes besitzt der Besucher eine entsprechende *`toCode(...)`*-Methode. Die in Quelltextausschnitt 6.1 dargestellte Methode übersetzt eine Verzweigungstruktur in C-Quelltext. Als Beispiel hierfür dient die in Quelltextausschnitt 6.1 dargestellte Methode zur Übersetzung einer Verzweigungs-Struktur des Baumes in C-Quelltext. Wird die Methode aufgerufen, so wird aus der Struktur Quelltext erzeugt, der die Struktur widerspiegelt. Dabei werden rekursiv alle Nachfolger der Struktur besucht und so für den Teilbaum Quelltext erzeugt. Um den gesamten Syntaxbaum zu übersetzten, reicht es daher dem Besucher die Wurzel-Struktur zu übergeben. Weiterhin steht zur Synthese neuer Methoden aus einem Syntaxbaum und der Methodensignatur die Methode *`toNewMethod(...)`* zur Verfügung. Um die Implementierung neuer Besucher zu vereinfachen, wurde die abstrakte Klasse *`AbstractVisitor`* geschrieben. Die Klasse enthält Hilfsmetho-

Tabelle 6.1: Aufzählung der verschiedenen Knoten aus denen der Syntaxbaum aufgebaut wird und ihrer Funktion.

Klassenbezeichnung	**Funktion**
`ASTRoot:`	Wurzel-Struktur
`ASTExpression:`	Generischer Ausdruck
`ASTDeclaration:`	Variablen-Deklaration
`ASTMethodCall:`	Methodenaufruf
`ASTConditionCall:`	Bedingungsaufruf
`ASTComparator:`	Vergleichsaufruf
`ASTBlock:`	Sequenz-Struktur
`ASTBranch:`	Verzweigungs-Struktur
`ASTWhile:`	While-Schleifen-Struktur
`ASTFor:`	For-Schleifen-Struktur
`ASTParallel:`	For-Schleifen-Struktur
`ASTText:`	Textzeile

Tabelle 6.2: Übersetzung des Komparators zu String-Vergleichszeichen.

Komparator	**Vergleichszeichen**
`EQ`	`==`
`GE`	`>=`
`GT`	`>`
`LE`	`<=`
`LT`	`<`
`NE`	`!=`

den für die Einrückung, und implementiert *`toCode(...)`*-Methoden für `ASTExpression`, `ASTBlock` und `ASTRoot`. Die Hilfsmethode `toComparatorSign(...)` übersetzt den Enumerator für Komparatoren nach Tabelle 6.2. Sprachen mit ähnlicher Syntax können zur Übersetzung weitere Methoden in abstrakten Besuchern bündeln. Für Java und C existiert deshalb der abstrakte Besucher `CStyleVisitor`. Die Methode *`generateHead(...)`* wird in der Implementierung von `toNewMethod(...)` verwendet, um die Methodensignatur in Quelltext zu übersetzen.

Das zweite Untermodul `CodeGen-Core` ist die Stufe des Nachverarbeitungsfließbandes. Hier werden die einzeln Besucher-Module über die `ServiceLoader`-Architektur von Java geladen. Dann wird als erstes anhand der Existenz der Methodensignatur bestimmt ob eine Befehls- oder Lehrsequenz übersetzt wird. Bei einer Befehlssequenz wird der *`toCode(...)`*-Methode des Besucher nur die Wurzel des Syntaxbaum übergeben. Der Besucher durchläuft der Baum rekursiv bis zu den Blättern und gibt einen String mit dem erzeugten Quelltext zurück. Zur Übersetzung einer Lehrsequenz wird stattdessen die Besucher-Methode *`toNewMethod(...)`* aufgerufen. Hierbei wird dem Besucher zusätzlich zur Wurzel des Syntaxbaum auch die Methodensignatur übergeben. Der Besucher übersetzt beides in Quelltext einer neuen Methode. Am Ende wird der synthetisierte Quelltext `ASTPostPipelineData` zur Weiterverarbeitung übergeben.

Das `MethodSynthesizer`-Modul stand zur Zeit der Implementierung des Prototypen noch nicht zur Verfügung. Daher wurde für den Prototyp der `PseudoCodeVisitor`-Visitor implementiert. Dieser übersetzt den abstrakten Syntaxbaum in Pseudo-Quelltext, welcher noch Text in natürlicher Sprache enthält. In Quelltextausschnitt 6.2 ist ein Beispiel in Pseudo-Quelltext für den Satz „*if the dishwasher is full then close it and press the red button twice*“ dargestellt.

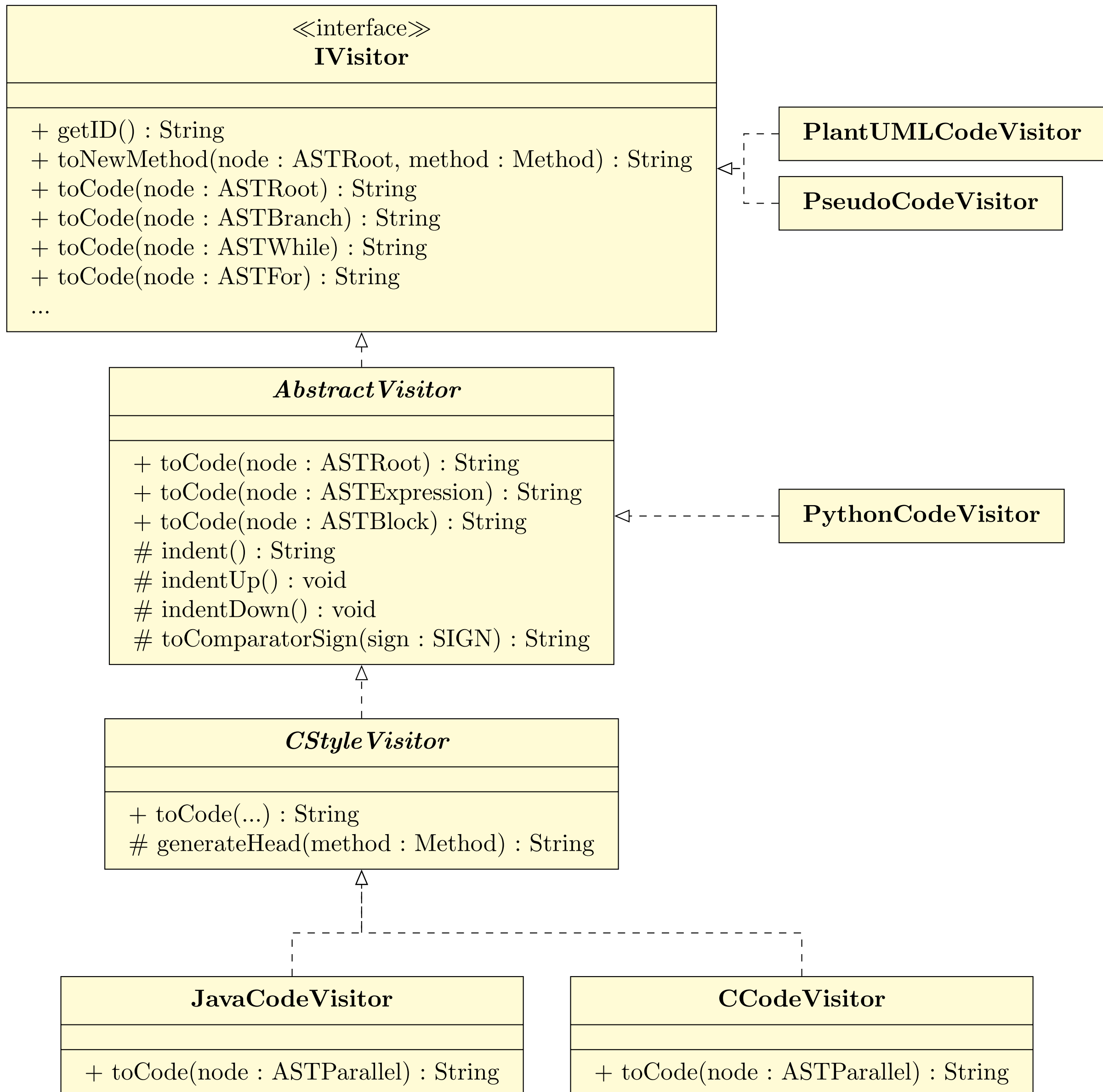


Abbildung 6.13: Vererbungshierarchie der `IVisitor`-Schnittstelle zur Implementierung der Besucher für die Übersetzung des Syntaxbaums in Quelltext.

Die Synthese von Aktivitätsdiagrammen mit PlantUML wurde in Abschnitt 3.3 als Zielsystem definiert. Da der PlantUML-Quelltext für Aktivitätsdiagramme einer prozeduralen Programmiersprache entspricht, kann der Syntaxbaum in PlantUML-Quelltext übersetzt werden. Daher wurde mit `PlantUMLCodeVisitor` ein Besucher für PlantUML implementiert. In Quelltextausschnitt 6.3 ist erzeugter PlantUML-Quelltext dargestellt. Zur Erzeugung des Aktivitätsdiagramm (siehe Abbildung 6.14) muss der Quelltext anschließend kompiliert werden.

Eine Möglichkeit eine Vielzahl an Programmiersprachen zu implementieren, wäre die Verwendung einer Sprache wie *Haxe* [1]. Haxe ist als plattformübergreifende Programmiersprache entworfen. Dadurch lässt sie sich in eine Vielzahl von häufig genutzten Sprachen wie Java, Python und C++ übersetzen. Da der Haxe-Kompilierer keine Anwendungsschnittstelle zur Übersetzung einzelner Methoden besitzt, erschwert dies das Übersetzen. Dadurch muss immer ein minimales, korrektes Haxe-Programm erzeugt werden. Das heißt, dass die

[1] `https://haxe.org/`, zuletzt besucht am 09.12.2019

```
public String toCode(ASTBranch astBranch) {
    //Conditions
    String code = "if␣(";
    for (ASTNode c : astBranch.getConditions().getExpressions())
        code += c.visit(this) + "␣&&␣";
    if (!astBranch.getConditions().getExpressions().isEmpty())
        code = code.substring(0, code.length() - 4);
    code += ")\n";

    //Then-Block
    code += "{\n" + astBranch.getThenBlock().visit(this) + "}";

    //Else-Block
    code += "␣else␣{" + astBranch.getElseBlock().visit(this) + "}";

    return code;
}
```

Quelltextausschnitt 6.1: `CStyleVisitor`-Methode zur Übersetzung einer Verzweigungs-Struktur in Quelltext (ohne Einrückung des generierten Quelltextes)

```
if (>:if the dishwasher is full)
then {
>:then close it(dishwasher)
for(int iter0=0;iter0 LT 2;>:iter0++) {
>:and press the red button
}
} else {}
```

Quelltextausschnitt 6.2: Beispiel für durch `PseudoCodeVisitor` im Prototyp erzeugten Pseudo-Quelltext.

Schablonen für jedes Zielsystem in Haxe geschrieben werden müssen. Stattdessen wurden direkt Besucher für die Programmiersprachen erzeugt.

Die Zielsysteme ARMAR-III und der Legoroboter verwenden beide Java als Programmiersprache. Daher wurde mit `JavaCodeVisitor` ein Besucher für Java geschrieben. Dazu mussten nur die `toCode(...)`-Methoden des Besuchers implementiert werden. Die Implementierung der Methode zur Übersetzung der Parallelität (siehe Quelltextausschnitt 6.4) stellte hierbei eine Herausforderung dar. Da Java keinen parallelen Sektionen ähnlich OpenMP kennt, wurde die Parallelität durch den expliziten Aufruf der `start()`-Methode der Klasse `Thread` modelliert. Um Aktionen parallel auszuführen, werden sie in anonyme Klassen eingebaut und mit `start()` gestartet. Die implizite Barriere wird explizit durch `join()` modelliert. Java-Quelltext der durch `JavaCodeVisitor` für das Zielsystem ARMAR-III erzeugt wurde ist in Quelltextausschnitt 6.5 dargestellt.

Bei der Implementierung einer Sprache bei der die Einrückung Auswirkung auf die Ausführung hat, muss die Formatierung beachtet werden. Der Besucher `PythonCodeVisitor` für die Programmiersprache Python ist hierfür ein Beispiel. Für die Einrückung wurde die abstrakte Klasse `AbstractVisitor` geschrieben, von der `PythonCodeVisitor` erbt. `AbstractVisitor` enthält eine zusätzliche Variabel, die die Einrückungsebene darstellt. Durch Erhöhen oder Erniedrigen der Variable kann der Einrückungsabstand verändert werden. Eine Hilfsmethode `indent()` gibt eine der Einrückungsebene entsprechende Anzahl an Tabulatoren als String zurück. Soll eine Zeile eingerückt werden, so wird der

```
@startuml

#HotPink:if the dishwasher is full then close it and
press the red button twice;
detach

start

'Generated Code:
if (:if the dishwasher is full;) then (True)
:then close it(dishwasher);
while ( for 2 times )
:and press the red button;
endwhile


else (False)
endif



stop

'


@enduml
```

Quelltextausschnitt 6.3: Beispiel für PlantUML-Quelltext, welcher durch `PlantUMLCodeVisitor` erzeugt wird.

Tabulator-String an den Anfang der Zeile gesetzt. Zusätzlich benötigte bei der Erstellung des Besuchers für Python die in Quelltextausschnitt 6.6 dargestellte For-Schleife besondere Aufmerksamkeit. Durch die Verwendung von `range(i)` unterscheidet sich die Syntax zu Java. Die `toCode(...)`-Methode des Besuchers für For-Schleifen musste daher komplett neu geschrieben werden.

Für die Programmiersprache C wurde `CCodeVisitor` implementiert. Beim Besucher für die Programmiersprache C konnten die `toCode(...)`-Methoden vom Java-Besucher übernommen werden. Daher wurde ein weiterer abstrakter Besucher `CStyleVisitor` erstellt, der Syntax im C-Stil implementiert. Parallelität lässt sich in C leicht durch OpenMP modellieren. Da sich die Übersetzung der Parallelitätsstruktur zu Java unterscheidet, wurden beide separat implementiert. Die Version für `CCodeVisitor` ist in Quelltextausschnitt 6.7 zu dargestellt. In Quelltextausschnitt 6.8 ist synthetisierter C-Quelltext für das Zielsystem ARMAR-III dargestellt. Dies zeigt, dass Quelltext unabhängig der Programmiersprache des Zielsystem erzeugt werden kann.

6.3.3 CodeInjector

Zum Schluss wird der erzeugte Quelltext entweder ausgeführt oder zur Erweiterung der Anwendungsschnittstelle benutzt. Das Modul `CodeInjector` ist dafür zuständig den Quelltext entsprechend zu injizieren. Dazu wird das Schablonen-System *Jtwig* [2] verwendet. Eine vollständige Schablone für das Java-Zielsystem Legoroboter ist in Quelltextausschnitt 6.9 dargestellt. Soll die Befehlsbeschreibung ausgeführt werden, so wird der Quelltext in eine Schablone geschrieben und als neue Datei abgespeichert. Der Pfad der Schablone lässt

[2] `http://jtwig.org/`, zuletzt besucht am 09.12.2019

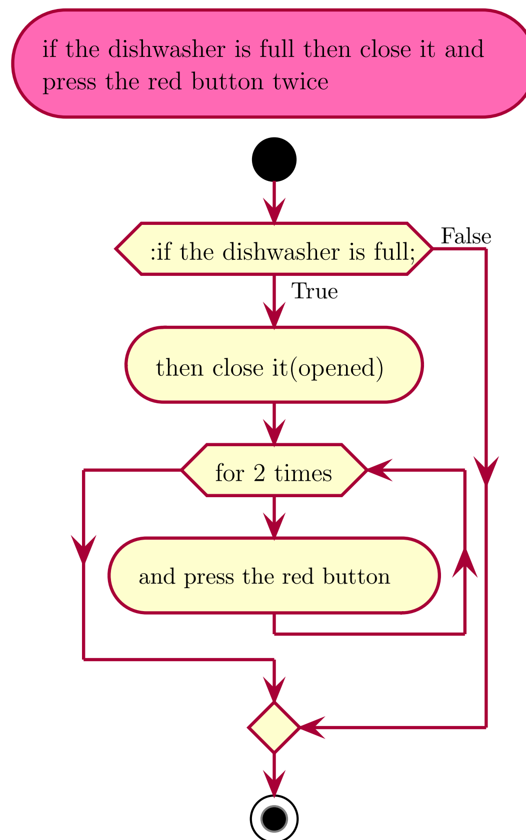


Abbildung 6.14: Beispiel für ein mit `PlantUMLCodeVisitor` erzeugtes Aktivitätsdiagramm

sich in der Konfigurationsdatei einstellen. Damit wird die Erweiterbarkeit auf andere Zielsysteme ermöglicht. Neben dem Platzhalter für den erzeugten Quelltext wurden in der Schablone auch Platzhalter für Kommentare, Variablen und Importe reserviert. Alle möglichen Importe und Variablen stehen aber in der Regel bereits in der Schablone.

Das Erlernen einer neuen Methode durch eine Lehrsequenz übernimmt ebenfalls das `CodeInjector`-Modul. Die Signatur der neuen Methode wird in `ASTPostPipelineData` gespeichert. Existiert die Signatur, so erzeugt `CodeInjector` die neue Methode mit ihrer Signatur als private Methode. Die Schablone besitzt hierfür am Ende einen speziellen Platzhalter für neue Methoden. Dieser Platzhalter wird nicht überschrieben, um das Hinzufügen weiterer Methoden zu erlauben.

6.3.4 ParseGraphDrawer

Das Modul `ParseGraphDrawer` ist dafür zuständig den PARSE-Graphen mit der Auszeichnugssprache Latex darzustellen. Es wurde im Prototyp entwickelt, um sich einen Überblick über den PARSE-Graphen zu verschaffen. Dadurch war es leichter möglich Heuristiken zu erkennen, die für die Mustersuche verwendet werden konnten. Auch konnte so der abstrakte Syntaxbaum im PARSE-Graphen betrachtet werden, was die Fehlersuche vereinfachte. `ParseGraphDrawer` durchläuft alle Kanten des Graphen und schreibt jede Kante und ihre dazugehörigen Knoten in eine Datei. Hierbei kann eine Tabelle verwendet werden, um Knoten abhängig von ihren Typen einzufärben. Da die Attribute wie beispielsweise Synonyme sehr lang werden können, existiert eine Schwarzliste, die die Darstellung dieser Attribute verhindert. Die Datei kann dann mit LuaLatex kompiliert und als PDF-Dokument betrachtet werden. Um mehrere Graphen leicht hintereinander zu kompilieren, wurde zusätzlich eine Stapelverarbeitung (engl. batch script) erstellt.

```java
public String toCode(ASTParallel node) {
    int threadIterator = 1;
    String threadJoins = "";
    String code = indent() + "//PARALLEL:\n";
    for (ASTBlock block : node.getSections()) {
        code += indent() + "//SECTION " + threadIterator++ + ":\n";
        code += indent() + "Thread t" + threadNumber + "= new Thread() 
            { ";
        code += "public void run() { \n";
        indentUp();
        code += block.visit(this);
        indentDown();
        code += indent() + "}};\n";
        code += indent() + "t" + threadNumber + ".start(); \n";
        threadJoins += indent() + "t" + threadNumber++ + ".join();\n";
    }
    code += indent() + "//END PARALLEL\n";
    code += threadJoins;
    return code.trim();
}
```

Quelltextausschnitt 6.4: Implementierung der Parallelität in `JavaCodeVisitor`

```java
// armar go to the kitchen
go2(Kitchen);
if (isFull(Dishwasher))
    {
    // then close it
    close(Dishwasher.Door);
    // and start the dishwasher
    start(Dishwasher);
} else {                    }
```

Quelltextausschnitt 6.5: Erzeugter Java-Quelltext für das Zielsystem ARMAR-III. Die Eingabe war „*armar go to the kitchen if the dishwasher is full then close it and start the dishwasher*“

```java
public String toCode(ASTFor node) {
    String code = indent() + "for " + node.getVariableName() + " in 
        range(" + node.getIterations() + "):\n";
    indentUp();
    code += node.getBlock().visit(this);
    indentDown();
    return code;
}
```

Quelltextausschnitt 6.6: Implementierung der For-Schleife in `PythonCodeGen`

```
public String toCode(ASTParallel node) {
    String code = indent() + "#pragma␣omp␣parallel␣sections␣\n␣{";
    indentUp();
    for (ASTBlock block : node.getSections()) {
        code += indent() + "#pragma␣omp␣section␣\n␣{";
        indentUp();
        code += block.visit(this);
        indentDown();
        code += indent() + "}\n";
    }
    indentDown();
    code += indent() + "}\n";
    return code.trim();
}
```

Quelltextausschnitt 6.7: Implementierung der Übersetzung der Parallelität in `CCodeVisitor`

```
// armar go to the kitchen
go2(Kitchen);
#pragma omp parallel sections
{
    #pragma omp section
    {
        // and bring me orange juice
        bring1(OrangeJuice);
    }
    #pragma omp section
    {
        // while reading the news
        read(News);
    }
}
```

Quelltextausschnitt 6.8: Beispiel für erzeugten parallelen C-Quelltext aus der Eingabe „*armar go to the kitchen and bring me orange juice while reading the news*“

```java
import edu.kit.ipd.lego.*;
// {{ header }}

class MyLegoRobot extends LegoRobot  {

// {{ classHead }}

    public static void main(String[] args) {
        LegoRobot robo = new MyLegoRobot();
        try {
            robo.execute();
        } catch (Exception e) {
            System.err.println("Error");
        }
        robo.close();
    }

@Override
public void execute(){

// {{ topComment }}
// {{ code }}

}

// New Methods: {{newMethod}}

}
```

Quelltextausschnitt 6.9: Legoroboter-Schablone für Java-Quelltext. Die geklammerten Begriffe sind Platzhalter.

7 Evaluation

In den vorherigen Kapiteln wurde eine Lösungsansatz zur zielsystemunabhängigen Synthese von Quelltext in natürlicher Sprache entworfen. In diesem Kapitel wird das Werkzeug evaluiert, dass den Lösungsansatz implementiert. So wird bestimmt, ob die die Anfangs festgelegten Anforderungen erfüllt wurden. Da das Werkzeug von den anderen PARSE-Agenten abhängt, erfordert eine Evaluation des Werkzeuges die Betrachtung etwaiger Einflussfaktoren. Um den Einfluss der Agenten und Vorverarbeitung von PARSE vollständig zu kontrollieren, müssten ideale PARSE-Graphen manuell erstellt werden. Da bereits kurze Sätze zu großen Graphen führen, ist dies nicht durchführbar. Deshalb kann eine Evaluation der Werkzeuge nur in Verbindung mit allen PARSE-Agenten geschehen.

Zu Beginn des Kapitels werden in Abschnitt 7.1 die Ergebnisse einer Bestandsaufnahme der Qualität vorhandener Agenten von PARSE vorgestellt. Anhand dieser Bestandsaufnahme werden Änderungen an Agenten von PARSE vorgeschlagen, um eine genauere Evaluation des Werkzeugs zu ermöglichen. Um die eigentliche Evaluation durchzuführen, werden in Abschnitt 7.2 die zu überprüfenden Hypothesen aufgestellt. Anschließend werden in Abschnitt 7.3 die Mustererkenner und der Aufbau des abstrakten Syntaxbaums anhand von erzeugten Aktivitätsdiagrammen in einer *Online*-Studie und manuellen Auswertung evaluiert. Darauf folgt in Abschnitt 7.4 eine Beurteilung der abstrakten Syntaxbäume als Zwischenrepräsentation. Die Übersetzung des Syntaxbaumes in Quelltext und damit die Betrachtung der Anforderung der Zielsystemunabhängigkeit wird in Abschnitt 7.5 begutachtet. Zum Schluss werden in Abschnitt 7.6 die Ergebnisse zusammengefasst.

Für die Evaluation werden die in Abschnitt 3.3 beschriebenen Zielsysteme verwendet. Die Evaluation findet dabei hauptsächlich auf dem PARSE-Korpus statt. Der PARSE-Korpus enthält 13 Szenarien mit insgesamt 238 textuelle Befehlsbeschreibungen an den Küchenroboter ARMAR-III. Die Szenarien unterscheiden sich in Länge, Detailgenauigkeit und Komplexität der Befehle. Die Länge der Szenarien variiert zwischen einem generellen Einzelbefehl und Beschreibungen, die aus über 20 ausführlichen Schritten bestehen. Daneben gibt es Szenarien mit Kontrollstrukturen wie Verzweigungen, Schleifen, Parallelität, Disfluenzen und temporaler Ausführungsreihenfolge.

7.1 Qualität vorhandener PARSE-Agenten

Nach Abschluss der Implementierungsphase konnte ein erster Überblick über die Qualität des Werkzeuges gewonnen. Hierbei wurde der PARSE-Korpus zu Quelltext umgewandelt.

Dabei wurde festgestellt, dass 78 der 238 (32.77%) Beschreibungen nicht in Quelltext umgewandelt werden konnten. Mehrere Arten von Fehler dominierten bei dieser Betrachtung. Der erste Fehler trat nur bei einem bestimmten Szenario auf. Ursprung dieses Fehlers war die Verbindung zur Ontologie bei der Kontextanalyse (siehe Abschnitt 3.2.2.3). Der Ursprung des Fehlers lag nicht in den implementierten Werkzeugen und betraf nur wenige Texte. Trotzdem konnte der Fehler, durch die Maßnahmen zur Beseitigung anderer Fehler, größtenteils behoben werden. Der zweite Fehler zeigte sich darin, dass kein vollständiger Syntaxbaum aufgebaut werden konnte. Zum Beispiel fehlte bei der While-Schleife oder der Verzweigung die Bedingung. Auch gab es Fälle, bei denen die Verbindung der Strukturen des Syntaxbaumes nicht möglich war. Ursprung dieser Fehlerart war der `MethodSynthesizer`-Agent (siehe Abschnitt 3.2.2.2). Der Agent war nicht in der Lage in der natürlichsprachlichen Befehlsbeschreibung alle Methoden zu extrahieren. Dadurch fehlten `ASTSynth` die Blätter zur Berechnung der Pfade und es konnte kein vollständiger Syntaxbaum aufgebaut werden. Besonders auffällig war, dass `MethodSynthesizer` keine Methoden für Bedingungen fand. Da `MethodSynthesizer` die gefundenen Methoden mit der Ontologie abgleicht, konnte die Ontologie als Grund für diesen Fehler bestimmt werden. Sie enthält nicht alle gesuchten Methoden. Die Ontologie muss also vor der letztendlichen Evaluation, um fehlende Methoden erweitert werden. Dadurch kann die Fehlerrate von `MethodSynthesizer` reduziert werden. Das dritte Problem lag ebenfalls im `MethodSyntehsizer`. Die Texte, die in Quellcode übersetzt werden konnten, waren in vielen Fällen fehlerhaft. Zum einen waren weiterhin syntaktische Fehler vorhanden, wie die Verwendung von Methoden ohne Rückgabetyp als Bedingungen. Zum anderen fanden sich auch semantische Fehler, wie die Zuordnung falscher Methoden zu Aktionen durch `MethodSynthesizer`. So wurde beispielsweise aus „*if you see the dirty dishes ...*“ das „*see the dirty dishes*“ als Methode `close(Dishes)` interpretiert. Dieser Fehler kann ebenfalls durch die Erweiterung der Ontologie um fehlende Methoden und Synonyme für Methoden reduziert werden. Problematisch waren hierbei aber auch fehlerhafte Parameter. So wurde beispielsweise der Roboter „*Armar*“ in den Paramter `warmed` übersetzt. Weitere Beispiele für solche fehlerhaften Parameter waren „*in the meantime*“ welches zu `InstantMeal` wurde und die Erkennung von „*News*“ als `Nesquik`.

Die Qualität des implementierten Werkzeugs hängt von der Qualität von PARSE ab. Deshalb fand vor Durchführung der Evaluation eine Qualitätssicherung der Agenten statt. Hierbei wurde die Ontologie um fehlende Methoden und Entitäten erweitert. Weiterhin wurde auch die Zerteilung der Instruktion optimiert. Dazu wurden mögliche Beispiele aus dem Korpus gesammelt. Da sich viele systematische Fehler in `MethodSynthesizer` materialisieren, wurde für eine syntaktische Evaluation Pseudo-Quelltext und Aktivitätsdiagramme verwendet. So kann die Qualität des erstellten Werkzeugs unabhängig von `MethodSynthesizer` bemessen werden.

7.2 Hypothesen

Das grundsätzliche Ziel dieser Arbeit ist die zielsystemunabhängige Quelltextsynthese aus natürlicher Sprache. Die unpräzise Formulierung der Zielsetzung erschwert eine wissenschaftliche Evaluation der Arbeit. Daher sollen aus der Zielsetzung klare und überprüfbare Hypothesen extrahiert werden. Anhand dieser Hypothesen lassen sich konkrete Aussagen bezüglich der Qualität der Werkzeuge treffen. Die Evaluation dieser Arbeit findet daher anhand dieser Hypothesen statt:

H_1: Das Erzeugen von funktionsfähigem Quelltext aus natürlicher Sprache ist grundsätzlich möglich.

H_2: Befehlssequenzen in natürlicher Sprache, die Kontrollstrukturen enthalten, können zu Quelltext übersetzt werden.

H_3: Abstrakte Syntaxbäume bilden eine geeignete Zwischenrepräsentation, um natürliche Sprache in Quelltext zu übersetzen.

H_4: Die Erzeugung von Quelltext aus natürlicher Sprache kann unabhängig vom Zielsystem stattfinden.

Um Hypothese H_1 und Hypothese H_2 zu überprüfen, wird der PARSE-Korpus mit den für PARSE implementierten Werkzeugen zu Quelltext übersetzt. Beide Hypothesen unterscheiden sich jedoch in den betrachteten Aspekten des erzeugten Quelltextes. Bei Hypothese H_1 liegt der Fokus auf der grundsätzlichen Übersetzung natürlicher Sprache in Quelltext. Die Syntax und Semantik des Quelltextes soll separat evaluiert werden. Die Korrektheit der Syntax hat hierbei Auswirkungen auf die Kompilierbarkeit und Lauffähigkeit des Quelltextes. Die Korrektheit der Semantik hingegen bewertet ob der erzeugte Quelltext den Anweisungen des Nutzers entspricht. Hypothese H_2 erweitert die Hypothese H_1 um Kontrollstrukturen. Daher wird die Syntax der erzeugten Kontrollstrukturen, wie Verzweigungen, Schleifen und Parallelität, betrachtet. Auch der Einfluss der Semantik auf Kontrollstrukturen wird evaluiert. Die Hypothese H_3 ist schwer zu messen und soll daher argumentativ untermauert werden. So soll zum Einen die Übersetzung von natürlicher Sprache in abstrakte Syntaxbäume betrachtet werden. Zum Anderen soll die Übersetzung von Syntaxbaum zu Quelltext verdeutlicht werden. Die letzte Hypothese H_4 zeigt, dass der Einfluss des Zielsystems minimiert ist, sodass die Verfahren unabhängig vom Zielsystem verwendet werden können. Dazu sollen die möglichen Einflüsse des Zielsystems benannt und ihre Auswirkung auf die Werkzeuge bemessen werden.

7.3 Synthese des abstrakten Syntaxbaumes

Der in dieser Arbeit entworfene Lösungsansatz beruht auf zwei Phasen: Der Synthese des Syntaxbaumes und die anschließende Übersetzung des Baumes in Quelltext. Idealerweise sollten beide Phasen separat betrachtet werden. Daher liegt in diesem Abschnitt der Fokus auf der Synthese des Syntaxbaumes.

7.3.1 Automatische Evaluation des Syntaxbaumes

Eine automatische Evaluation des erzeugten Syntaxbaumes wäre ideal, da es eine Rückkopplung der gefundenen Fehler zum Dialogagenten ermöglichen würde. Das heißt, die automatische Evaluation könnte mit einer Fehlerkorrektur verbunden werden. Um den abstrakten Syntaxbaum auf Lauffähigkeit zu überprüfen könnten Analyseverfahren aus dem Übersetzerbau verwendet werden. Eine Implementierung dieser Verfahren würde aber den Rahmen dieser Arbeit übersteigen. Da aber Quelltext erzeugt wird, können Übersetzer dazu verwendet werden die Lauffähigkeit des erzeugten Quelltextes zu überprüfen. Problematisch ist hierbei aber, dass diese Verfahren nur auf dem endgültig erzeugten Quelltext arbeiten und daher nur syntaktische Fehler erkennen. Da kein Bezug der Verfahren zur natürlicher Sprache existiert, werden keine semantischen Fehler erkannt. Eine Alternative hierzu könnte die Analyse des PARSE-Graphen darstellen. Hierbei stünde nicht nur der ursprüngliche Text in natürlicher Sprache zur Verfügung, sondern auch alle Ergebnisse der Agenten. Dies wäre aber eine große Herausforderung, da die korrekte Erkennung von Fehlern ein hohes Verständnis der Sprache benötigt, welches maschinell aktuell nicht möglich ist. Eine automatische Evaluation müsste also Heuristiken verwenden. Diese sind aber in der Regel zu ungenau, um die Werkzeuge geeignet zu evaluieren. So würde durch Heuristiken eine hohe Anzahl an falsch Positiven und falsch Negativen gefunden werden. Aus diesen Gründen ist nur eine manuelle Evaluation möglich.

7.3.2 Darstellung der Syntaxbäume

Um eine isolierte Betrachtung des Syntaxbaumes zu erreichen, gibt es mehrere Ansätze. Mit dem implementierten Modul `ParseDrawer` kann der abstrakte Syntaxbaum inklusive dem gesamten PARSE-Graphen bildlich dargestellt werden. Dies hat den Vorteil, dass die Ergebnisse aller Agenten von PARSE ebenfalls explizit dargestellt werden. Anhand dieser Ergebnisse können Agenten als Fehlerursache überprüft werden. Zeitgleich führt diese große Menge an Informationen dazu, dass der eigentliche Syntaxbaum schwer zu erkennen ist und erhöht auch den den Aufwand bei der manuellen Evaluation deutlich. Bei der Darstellung des Syntaxbaumes besteht weiterhin das Problem, dass dieser nicht intuitiv ist. Aus diesen Gründen sollte diese Form nur für einzelne Beispiele verwendet werden, bei der eine detaillierte Analyse sinnvoll ist. Daneben gibt es die Möglichkeit den Syntaxbaum indirekt, d.h. als UML-Aktivitätsdiagramm, darzustellen. Der Vorteil hierbei ist, dass das Aktivitätsdiagramm die Befehlsbeschreibung in natürlicher Sprache widerspiegelt. Bei der manuellen Auswertung muss also nur entschieden werden, ob die Sequenz der Aktivitäten der gegebenen Befehlsbeschreibung entspricht. Die Übersetzung des Syntaxbaum in ein Aktivitätsdiagramm ist aber äquivalent mit der Übersetzung in Quelltext. Daher lässt sich nicht garantieren, dass etwaige Fehler bei der Übersetzung tatsächlich ihren Ursprung in der Erzeugung des Syntaxbaumes haben. Um die Aktivitätsdiagramme zu erstellen wird die UML-Auszeichnungssprache *PlantUML* (siehe Abschnitt 3.3) verwendet. In Abbildung 7.1 ist ein auf natürlicher Sprache basierendes Aktivitätsdiagramm dargestellt. Der gesamte Eingabesatz ist hierbei in Pink hervorgehoben. Das gleiche Aktivitätsdiagramm mit den von `MethodSynthesizer` bereitgestellten Methoden ist in Abbildung 7.2 zu sehen. Da `MethodSynthesizer` bei der Bedingung die falsche Methode fand, enthält das Diagramm einen Fehler. Weiterhin ist für Menschen leicht ersichtlich, dass die Methode `pour()` nicht der gesuchten Methode `press(Button)` entspricht.

7.3.3 Aufbau der Studie

Aktivitätsdiagramme mit natürlicher Sprache sind selbst für Laien nach kurzer Einführung verständlich. Eine Auslagerung der Evaluation der erzeugten Aktivitätsdiagramme durch eine Studie hat einige Vorteile: Als erstes wird durch die Studie jedes Aktivitätsdiagramm mehrmals von unterschiedlichen Personen evaluiert. Dadurch steht nicht nur das Ergebnis einer einzelnen Person zur Verfügung sondern das von mehreren Probanden. Auch können durch die Befragung von Laien Einflüsse, die durch beispielsweise Kenntnisse der Informatik entstehen, beseitigt werden. Sind die Probanden zufällig gewählt, so erlaubt dies auch eine bessere Verallgemeinerung der Ergebnisse. Weiterhin ist durch die Auslagerung der Evaluation eine Befangenheit durch den Entwickler ausgeschlossen. Ein letzter Vorteil ist die zeitliche Machbarkeit. So können verschiedene Probanden zeitgleich unterschiedliche Aktivitätsdiagramme auf Fehler untersuchen. Dies erlaubt eine bessere Skalierung der Evaluation. Aus diesen Gründen wurde eine Online-Studie durchgeführt, bei der Probanden die erzeugten Aktivitätsdiagramme bewerteten und Fehler markierten.

Für die Online-Studie wurden automatisch 24 Formulare mit je zehn Aktivitätsdiagrammen mit Hilfe von *Google Forms* [1] erzeugt. Hierzu wurde die 238 Befehlsbeschreibungen des PARSE-Korpus (siehe Abschnitt 3.3) in 237 Aktivitätsdiagramme übersetzt. Bei einer Befehlsbeschreibung stürzte PARSE bereits bei der Kontextanalyse ab. Aus der Menge der Aktivitätsdiagramme wurden pro Formular in der Regel zehn Diagramme zufällig ausgewählt. Um eine gleichmäßige Verteilung der Szenarien zu garantieren, wurde hierbei zuerst zufällig ein Szenario und dann in einem zweiten Schritt ein zufälliges Aktivitätsdiagramm aus dem Szenario gewählt. Dabei wurde sichergestellt, dass kein Aktivitätsdiagramm mehrfach vorkommt. Um das Verständnis der Probanden auf eine möglichst einheitliche Stufe zu

[1] `https://www.google.com/forms/about/`, zuletzt besucht am 09.12.2019

bringen, beginnt jedes Formular mit einer Einführung. Die Einführung ist in Abschnitt B dargestellt. In dieser Einführung wurden die Strukturen des Aktivitätsdiagramm, mögliche Fehlerfälle und die Aufgabenstellung erklärt. Darauf folgten zehn Aktivitätsdiagramme mit Fehlerfällen zum Ankreuzen. Die Auswahlmöglichkeiten für die Fehlerfälle beinhalteten auch Fehler, die ihren Ursprung nicht in den entworfenen Werkzeugen haben. Dabei durften auch mehrere Fehlerfälle angekreuzt werden. Die Probanden hatten die Möglichkeit zwischen folgenden Fehlerfällen zu wählen. Der Bezugsrahmen beschreibt dabei die Aktivitäten, die von der Kontrollstruktur abhängen.

1. **Vollständig korrekt:** In dem Aktivitätsdiagramm konnten keine Fehler gefunden werden. Da dies die einzige Wahlmöglichkeit für Korrektheit war, schließt sie andere Wahlmöglichkeiten aus.
2. **Aktivität inkorrekt/unverständlich:** Eine Aktivität des Aktivitätsdiagramm enthielt eine fehlerhafte oder unverständliche Aktivität. Als fehlerhafte Anweisungen waren fehlende Informationen oder Informationen, die Teil einer anderen Aktivität sein sollten, gemeint.
3. **Fehlende Aktivität:** Im Aktivitätsdiagramm fehlte eine Aktivität. Hierbei waren auch Aktivitäten zu verstehen, die in mehrere Aktivitäten aufgetrennt werden sollten.
4. **Aktivität zuviel:** Das Aktivitätsdiagramm enthält eine Aktivität, die entfernt werden sollte. Hierunter war auch die Vereinigung mehrerer Aktivitäten zu einer einzelnen Aktivität.
5. **Aktivitätsreihenfolge inkorrekt:** Die Reihenfolge der Aktivitäten eines Aktivitätsdiagrammes entsprach nicht der semantischen Bedeutung. Aktivitäten sollten also vertauscht oder an anderer Stelle im Diagramm eingefügt werden.
6. **Koreferenz fehlerhaft:** Die Aktivität enthält eine Koreferenz, die auf die falsche Entität verweist oder fehlende Koreferenzen enthält.
7. **Verzweigung inkorrekt:** Das Aktivitätsdiagramm enthält Fehler in der Verzweigung. Als Fehler zählten inkorrekte Bedingungen, komplett fehlende Verzweigungen und ein fehlerhafter Bezugsrahmen der Verzweigung.
8. **Schleife inkorrekt:** Das Aktivitätsdiagramm enthält Fehler in der Schleife. Als Fehler zählten inkorrekte Bedingungen, komplett fehlende Schleifen und ein fehlerhafter Bezugsrahmen der Schleife.
9. **Parallelität inkorrekt:** Das Aktivitätsdiagramm enthält Fehler in der Parallelität. Als Fehler zählten komplett fehlende Anweisungen oder ein fehlerhafter Bezugsrahmen der Parallelität.
10. **Sonstige Fehlerfälle:** Das Aktivitätsdiagramm enthält Fehler, die sich keinem anderen Fehlerfall zuordnen lassen. Für diese Auswahlmöglichkeit musste eine Kurzbeschreibung gegeben werden.

Als Plattform für die Online-Studie wurde *Prolific* [2] verwendet. Prolific ist eine Online-Plattform, über die Probanden gegen Bezahlung an einer Studie teilnehmen können. Dadurch kann in kurzer Zeit eine große Anzahl an Probanden zur Ausfüllung der Formulare gefunden werden. Hierzu erstellt der Durchführende der Studie ein Formular und veröffentlicht dieses über Prolific. Bei der Veröffentlichung muss auf Prolifc eine Kurzbeschreibung (siehe Abschnitt A), die Anzahl an gesuchten Probanden, die geschätzte Bearbeitungszeit und die Belohnung für das Ausfüllen eines Formulars angegeben werden. Probanden sind frei bei der Wahl an welcher Studie sie teilnehmen. Der Durchführende der Studie ist in

[2] `https://www.prolific.co/`, zuletzt besucht am 09.12.2019

Tabelle 7.1: Inter-Annotator-Übereinstimmung für Ergebnisse der Studie

Formular	**Kappa**
1	0,24
2	0,39
3	0,19
4	0,19
5	0,34
6	0,15
7	0,32
8	0,32
9	0,39
10	0,37
11	0,33
12	0,33
13	0,24
14	0,12
15	0,41
16	0,31
17	0,27
18	0,36
19	0,38
20	0,32
21	0,41
22	0,41
23	0,59
24	0,39
Gesamt	0,32 (σ=0.10)

der Lage Abgaben aus Gründen wie geringe Datenqualität oder unrealistisch kurze Bearbeitungszeit abzulehnen. Wurde eine Abgabe abgelehnt, so kann über Prolific ein neuer Proband gesucht werden. Für jedes Formular wurde eine eigene Studie mit Beschreibung der Aufgabe erzeugt. Jedes Formular sollte von fünf Probanden bearbeitet werden, wobei es keine Voraussetzung für die Probanden gab. Als geschätzte Bearbeitungszeit wurden 20 Minuten eingestellt, welche auf maximal 67 Minuten verlängert werden durfte. Unabhängig der Zeit, erhielt der Proband für die akzeptierte Abgabe zwei britische Pfund als Belohnung.

7.3.4 Ergebnisse

Nach Abschluss der Online-Studie wurden die Abgaben der Probanden von Hand auf Sinnhaftigkeit überprüft, um Abgaben von geringer Qualität zu beseitigen. Kriterien für Ablehnung einer Abgabe waren widersprüchliche Antworten oder zufällig wirkende Abgaben. Als Indikator für zufällige Abgaben diente beispielsweise eine kurze Bearbeitungszeit, das Finden von unsinnigen Fehlern in trivialen Aktivitätsdiagrammen oder das Übersehen offensichtlicher Fehler. Dadurch wurden 19 Abgaben mit ungenügender Qualität aussortiert von neuen Probanden wiederholt. Insgesamt wurden 120 verwertbare und 19 aussortierte Formulare abgegeben. Über alle Fehlerfälle eines Aktivitätsdiagramms wurden die Inter-Annotator-Übereinstimmung, d.h. Fleiss' Kappa, berechnet. Fleiss' Kappa ist ein statistisches Maß, dass Auskunft über die Übereinstimmung mehrerer Probanden gibt. Ein höherer Wert bedeutet eine stärkere Übereinkunft. Die Kappa-Werte für alle Formulare sind in Tabelle 7.1 dargestellt. Im Mittel wurde eine ausreichende Übereinkunft

Tabelle 7.2: Ergebnisse der Online-Studie für Fehlerfälle für unterschiedliche Wahlstrategien

Fehlerfall	**Eindeutig (5)**	**Abweichung (4)**	**Mehrheit (3)**
Vollständig korrekt	29 (12,24%)	64 (27,00%)	113 (47,68%)
Anweisung inkorrekt	0	3 (1,27%)	21 (8,86%)
Fehlende Anweisungen	0	0	3 (1,27%)
Anweisungen zuviel	0	2 (0,84%)	9 (3,80%)
Reihenfolge falsch	0	0	5 (2,11%)
Koreferenz inkorrekt	0	4 (1,69%)	10 (4,22%)
Verzweigung inkorrekt	0	1 (0,42%)	3 (1,27%)
Schleife fehlerhaft	0	1 (0,42%)	4 (1,69%)
Parallelität falsch	0	0	0
Summe	29	75	168

Tabelle 7.3: Ergebnisse der Online-Studie für Aktivitätsdiagramme für unterschiedliche Wahlstrategien

Diagramme	**Eindeutig (5)**	**Abweichung (4)**	**Mehrheit (3)**
Korrekt	29 (12,24%)	64 (27,00%)	113 (47,68%)
Fehlerhaft	0	11 (4,64%)	50 (21,10%)
Bestimmt	29 (12,24%)	75 (31,65%)	163 (68,78%)
Unbestimmt	208 (87,76%)	162 (68,35%)	74 (31,22%)
Gesamt	237 (100%)	237 (100%)	237 (100%)

von 0,32 (σ=0,10) erreicht. Dieser Wert entspricht Landis und Koch [LK] zufolge einer ausreichenden Übereinstimmung. Durch Wahlverfahren dürfte es daher möglich sein, aus den Abgaben der Probanden über Korrektheit und Fehler für die einzelnen Aktivitätsdiagramme zu entschieden.

7.3.4.1 Fehleranalyse durch Wahlverfahren

Da die Probanden in der Regel selten eindeutig übereinstimmen, gibt für jedes Aktivitätsdiagramm mehrere Antworten. Von den 237 Aktivitätsdiagrammen wurden nur 29 (12,24%) von allen Probanden als vollständig korrekt markiert. Bei den Fehlern gab es keine vollständige Einigkeit. Es ist daher nötig die Ergebnisse durch verschiedene Wahlstrategien zusammenzufassen. Die Ergebnisse der Wahlstrategien sind in Tabelle 7.2 für Fehlerfälle und in Tabelle 7.3 für Diagramme dargestellt. Erlaubt man eine einzelne Abweichung, so werden 64 Diagramme (27,00%) als vollständig korrekt markiert. Zusätzlich lassen sich in 11 (4,64%) Diagrammen Fehler bestimmen. Wird die Abgaben der fünf Probanden durch eine einfache Mehrheitswahl zusammengefasst, so gibt es für 163 von 237 (68,78%) Diagramme ein Ergebnis. Dabei werden 113 Diagramme (47,68%) als vollständig korrekt markiert. Bei den fehlerhaften Diagrammen teilen sich 50 (21,10%) Diagramme auf die 55 gefundenen Fehlerfälle auf. Das bedeutet, dass 5 (2,11%) Diagramme mehrere Fehlerfälle enthalten. Bei den Fehlern sind fehlerhafte Aktivitäten mit 21 (8,86%) der häufigste Fehlerfall. Fehlerhafte Koreferenzen wurden in 10 (4,22%) der Diagramme gefunden. Die beiden Fehlerfälle können bei der Übersetzung in Quelltext zu fehlerhaften Methoden und Parametern führen. Der Quelltext enthält dadurch semantische Fehler. Syntaktische Fehler im Quelltext können durch die restlichen Fehlerfälle entstehen. Aktivitätsdiagramme mit unnötigen Aktivitäten gab es in 9 (3,80%) Fällen. Im Gegenteil dazu wurden fehlende Aktivitäten in 3 (1,27%) Aktivitätsdiagrammen gefunden. Die Reihenfolge der Aktivitäten war in 5 (2,11%) der Diagramme inkorrekt. Bei der Übersetzung in Quelltext kann es

Tabelle 7.4: Kategorisierung sonstiger Fehler in der Online-Studie

Fehlerfall	**Anzahl**
Disfluenzen und Rechtschreibung	17
Koreferenzen	5
Entitätenbezeichnung	5
Kontrollstrukturen	10
Zerteilung	4
Fehlende Aktivitäten	6
Kein Diagramm sichtbar	2
Gesamt	49

durch diese Fehlerfälle zu einer fehlerhaften Anzahl und Ausführungsreihenfolge der Methoden kommen. Bei den Kontrollstrukturen verteilen sich die Fehlerfälle mit 4 (1,69%) Aktivitätsdiagrammen auf fehlerhafte Schleifen und mit 3 (1,27%) Diagrammen auf fehlerhafte Verzweigungen. Es wurden keine fehlerhafte Parallelität gefunden. Durch diese Fehlerfälle entstehen inkorrekte Kontrollstrukturen bei der Übersetzung in Quelltext. Bei der Mehrheitswahl sind 74 (31,22%) Aktivitätsdiagramme unbestimmt. Das heißt, für diese Aktivitätsdiagramme konnte durch einen Mehrheitsentscheid kein Ergebnis bestimmt werden.

Neben den vorgegebenen Fehlerfällen gab es noch die Möglichkeit über den Fehlerfall „Sonstiges“ textuell Fehler zu beschreiben. Insgesamt wurden 49 Fehler über diese Möglichkeit annotiert. In Tabelle 7.4 sind die Fehler in Kategorien eingeteilt. Disfluenzen (11 Fälle) und Rechtschreibfehler (6 Fälle) im Eingabesatz sind mit 17 Fällen der häufigste Fehlerfall. Ebenfalls im Bezug zum Eingabesatz wurden in fünf Fällen Entitätsfehler genannt. Darunter fallen die Verwendung von „*Robo*“ statt „*Armar*“ (2 Fälle), fehlende (2 Fälle) Entitäten und fehlerhafte (1 Fall) Entitäten. Weiterhin fand in fünf Fällen eine Korrektur für fehlerhafte Koreferenzen statt. In vier Fällen, die in Beispiel 7.3.1 dargestellt sind, wurden Verbesserungsvorschläge für fehlerhafte Zerteilungen angegeben. Bei einem Probanden wurden aufgrund technischer Probleme zwei Aktivitätsdiagramme nicht geladen. Trotzdem wurde er akzeptiert, da seine sonstige Abgabe in Ordnung war. Interessant waren sechs Fälle, bei denen die Probanden weitere Aktivitäten erwarteten, die aber nicht im ursprünglichen Text vorhanden waren. Dabei wurde in fünf Fällen angegeben, dass sie Annahmen bezüglich der Fähigkeiten des Roboters trafen. In Beispiel 7.3.2 sind diese sechs Fälle dargestellt. Es ist daher wohl anzunehmen, dass weitere Probanden ähnliche Annahmen trafen, ohne dies explizit anzugeben. Als letztes wurden in zehn Fällen Kontrollstrukturfehler beschrieben. Sie sind in Beispiel 7.3.3 dargestellt. Die Ursachen dieser Fehler sind unterschiedlich. Sie verteilen sich meist auf einfache Spezifizierung und Erläuterungen (3 Fälle) der Antwort und fehlende Kontrollstrukturen (3 Fälle). Es gab aber auch Beschreibungen eines fehlerhaften Bezugsrahmen, einer unnötigen Schleife, einer Vereinigung von zwei Verzweigungen und einer unnötigen Aktivität bei der Parallelität, die durch einen Bezug zu einer vorherigen Aktion entstanden ist. Dies zeigt, dass ein Teil der Probanden auch Logikfehler der Befehlsbeschreibung selbst annotiert. Dadurch werden Diagramme, selbst bei vollständig korrekter Übersetzung durch die Werkzeuge als fehlerhaft interpretiert.

Unter der Annahme, dass die Ergebnisse des Mehrheitsentscheids stimmen, werden in 47,68% der Fälle vollständig korrekte Aktivitätsdiagramme erstellt. Im Vergleich dazu erreichten Arbeiten wie beispielsweise *SmartSynth* (58,7% Korrektheit) und *NLCI* (73,7% Genauigkeit) höhere Ergebnisse (siehe Abschnitt 4.1.1). Hierbei ist aber zu beachten, dass die Komplexität und Länge der Eingabetexte niedrig war. Zusätzlich gab es Einschränkungen beim Zielsystem und der Domäne. *SmartSynth* zum Beispiel war auf die Erzeu-

Beispiel 7.3.1: Verbesserungsvorschläge der Probanden für fehlerhafte Zerteilungen

1. „*Word ‚too‘ included in wrong activity also.*“
2. „ *‚put the plate‘ ‚inside activate the microwave‘ => ‚put the plate inside‘ ‚activate the microwave‘*“
3. „*3rd activity should remove the first word, ‚out‘ and place at end of 2nd activity.*“
4. „*This should be split into separate activities. Put dirty dishes in dishwasher. then Clean Dishes. then Put clean dishes into cupboard.*“

Beispiel 7.3.2: Fehlerbeschreibung der Probanden für fehlende Aktivitäten

1. „*If this is a task for robot, shouldn't it have activities for ‚finding popcorn‘ first then ‚bring it to the speaker‘?*“
2. „*everything seems correct, as ordered, but if I can add some details, it should close the fridge after taking the orange juice*“
3. „*should add an activity*“
4. „*Elaboration: Armar might not know where the popcorn bag is.*“
5. „*To elaborate: ‚activity missing‘: Unless we assume that the dishwasher is opened, it would need to be opened first. (I've seen this ‚issue‘ in previous ones, so might just be me going into detail).*“
6. „*Elaboration: Assuming the fridge is closed, it would have to be opened. Maybe a if-loop for fridge/cabinet/dishwasher open/close.*“

gung einfacher Skripte für Mobiltelefone in einer eigenen Domänensprache beschränkt und nicht in der Lage Befehlssequenzen mit Kontrollstrukturen zu übersetzen. In dieser Arbeit hingegen wurden weder die Eingabetexte noch die Domäne eingeschränkt. So gab es Befehlsbeschreibungen mit Kontrollstrukturen, einer großen Anzahl an Befehlen, Disfluenzen und Anweisungen die Einfluss auf die temporale Ausführungsreihenfolge haben. Für letzteres gab es keinen Agenten, weswegen sie auch nicht entdeckt werden konnten. Weiterhin war die Sprache der Eingabetexte nicht konstruiert, sondern beruhte auf Beschreibungen von Probanden. Zuletzt wurden die Aktivitätsdiagramme von mehreren Probanden annotiert, welche unterschiedliche Annahmen treffen konnten. Wie an den sonstigen Fehlern ersichtlich, wurden Aktivitätsdiagramme, die korrekt Übersetzt wurden, deren Befehlsbeschreibung aber Fehler enthält, als fehlerhaft annotiert. Die vollständige Korrektheit von 47,68% der Aktivitätsdiagramme ist daher positiv zu bewerten. Im Bezug auf die syntaktische Korrektheit des Quelltextes, kann Hypothese H_1 angenommen werden, da syntaktisch korrekter Quelltext erzeugt wurde.

7.3.4.2 Fehleranalyse durch Verschmelzung der Fehlerfälle

Während der Überprüfung der Ergebnisse fiel auf, dass die Probanden Schwierigkeiten hatten Fehlerfälle korrekt zuzuordnen. Viele Probanden erkannten zwar, dass es bei einer Kontrollstruktur ein Problem gab, aber sie konnten die Kontrollstruktur nicht korrekt

Beispiel 7.3.3: Fehlerbeschreibung der Probanden für Konstrollstrukturfehler

1. „*I think that ‚then please wash the green cup‘ should be with ‚and while you wash the cup*‘“ [Verdoppelung der Aktivität bei Parallelität]
2. „*I see no need for loop. Eliminate loop.*“ [Schleifenentfernung]
3. „*For loop missing*“ [Spezifizierung]
4. „*while loop missing*“ [Spezifizierung]
5. „*when ‚if there are no fresh oranges‘ is false, it means there are fresh oranges. It already checks once at the beginning of the task, so False check shouldn't be necessary*“ [Verzweigungsvereinigung]
6. „*I believe this needs a True/False check for ‚finding every piece‘, because even if there aren't any pieces of dishware it should still close the dishwasher*“ [Fehlende Kontrollstruktur]
7. „*The tasks don't make a lot sense. It should have IF or WHILE (concurrency) somewhere to check for dishwasher being full or empty.*“ [Fehlende Kontrollstruktur]
8. „*As written this would simply keep picking up dishes. It needs to be rewritten to place each dish into cupboard until dishwasher is empty.*“ [Bezugsrahmen der Syntax]
9. „*this works, but need to loop because there are more than 1 dish. and checking again for clean dish isn't necessary because if it's NOT dirty it means it's clean. (But I guess in real life, there is a chance of having broken dishes so...)*“ [Fehlende Kontrollstrukturen]
10. „*It seems like a straight command so I assume it does not require and True/False checks*“ [Erläuterung]

identifizieren. So wurde beispielsweise oft Schleifen, Verzweigungen und Parallelität verwechselt und vermischt. Aber auch bei den Aktivitäten selbst gab es Verwechslungen. Dies führt dazu, dass bei diesen Fehlerfällen keine Mehrheit bestimmt werden konnte. Daher sollen zusätzlich die verschiedenen Fehlerfälle zu Fehlerkategorien verschmolzen werden, um diese Verwechslungen der Probanden zu beseitigen. In Tabelle 7.5 sind die Ergebnisse für Fehlerkategorien nach Verschmelzung abhängig der Wahlstrategie dargestellt. Die Fehlerfälle werden zu drei Kategorien verschmolzen: Die erste Kategorie verschmilz fehlerhafte Schleifen, Verzweigungen und Parallelität zu der Kategorie „Kontrollstrukturfehler“. So sollen Unsicherheiten und Unverständlichkeiten der Probanden bei der Wahl der korrekten Kontrollstruktur entfernt werden. Als zweites werden syntaktische Fehler der Aktivitäten zu einer Kategorie „Sequenzfehler“ vereint. Hierbei werden die Fehlerfälle bezüglich der Anzahl und Reihenfolge der Aktivitäten verschmolzen. Die letzte Kategorie stellen die „Aktivitätsfehler“ dar. Hierbei wird der Fehlerfall „Aktivität inkorrekt/unverständlich“ und „Koreferenz inkorrekt“ verschmolzen. Die vollständige Korrektheit wurde nicht verschmolzen und entspricht einer eigenen Kategorie. Neben den verschmolzenen Kategorien wurden die Ergebnisse erneut durch drei Wahlstrategien bestimmt. Durch die Verschmelzung konnte eine größere Anzahl an Ergebnissen für Aktivitätsdiagramme (siehe Tabelle 7.6) bestimmt werden. Es wurden nun 3 (1,27%) Diagramme als eindeutig

Tabelle 7.5: Ergebnisse der Online-Studie für Fehlerkategorien nach Verschmelzung für unterschiedliche Wahlstrategien

Fehlerkategorie	**Eindeutig (5)**	**Abweichung (4)**	**Mehrheit (3)**
Vollständig korrekt	29 (12,24%)	64 (27,00%)	113 (47,68%)
Aktivitätsfehler	2 (0,84%)	15 (6,33%)	35 (14,77%)
Reihenfolgefehler	0	7 (2,95%)	31 (13,08%)
Kontrollstrukturfehler	1 (0,42%)	5 (2,11%)	14 (5,91%)
Summe	32	91	193

Tabelle 7.6: Ergebnisse der Online-Studie für Aktivitätsdiagramme nach Verschmelzung der Fehlerfälle für unterschiedliche Wahlstrategien.

Diagramme	**Eindeutig (5)**	**Abweichung (4)**	**Mehrheit (3)**
Korrekt	29 (12,24%)	64 (27,00%)	113 (47,68%)
Fehlerhaft	3 (1,27%)	27 (11,39%)	68 (28,69%)
Bestimmt	32 (13,50%)	91 (38,40%)	181 (76,37%)
Unbestimmt	205 (86,50%)	146 (61,60%)	56 (23,62%)
Gesamt	237 (100%)	237 (100%)	237 (100%)

fehlerhaft markiert. Dies ist eine Steigerung von 10,34%. Bei der Strategie „Abweichung“ können durch die Verschmelzung 16 (6,75%) weitere Diagramme als fehlerhaft identifiziert werden. Dies ist eine Steigerung von 21,33% im Vergleich zu den ursprünglichen 75 fehlerhaften Diagrammen ohne Verschmelzung. Bei der einfachen Mehrheitswahl können durch die Verschmelzung in 18 (7,59%) weitere Aktivitätsdiagrammen Fehler gefunden werden. Dies entspricht einer Steigerung von 11,04%. Von der Verschmelzung profitieren somit alle drei Wahlstrategien.

Da der Fehlerfall „Vollständig korrekt“ eine eigene Fehlerkategorie war, bleibt Anteil an korrekten Aktivitätsdiagrammen bei 47,68%. Die Ergebnisse zeigen, dass in 68 (28,69%) der Diagramme Fehler entdeckt wurden. Dies bedeutet, dass in 56 (23,62%) der Fälle die Probanden durch einen einfachen Mehrheitsentscheid nicht zu einem Ergebnis kamen. Die Verteilung der Fehler zeigt eine Dominanz der Aktivitätsfehler und Reihenfolgefehler auf. Aktivitätsfehler sind mit 35 (14,77%) die am häufigsten auftretende Fehlerkategorie. Bei diesem Ergebnis sollte aber beachtet werden, dass weniger Texte mit Kontrollstrukturen im Korpus vorhanden waren. So erkannte PARSE in 55 Diagrammen Kontrollstrukturen, während die Probanden in 14 (5,91%) der Diagramme Kontrollstrukturfehler fanden. Unter der Annahme, dass durch PARSE alle Kontrollstrukturen gefunden wurden, enthielten 25,46% der Diagramme mit Kontrollstrukturen Fehler. Die 31 (13,08%) Reihenfolgefehler waren hingegen zu erwarten. So enthielt ein Teil der Befehlsbeschreibungen diese Fehler, aber es gab keinen Agenten in PARSE, um sie zu erkennen und zu korrigieren. Die Fehler wurden daher nur übernommen und können durch die Implementierung eines solchen Agenten beseitigt werden. Trotz der Fehler kann Hypothese H_2 teilweise angenommen werden, da auch Aktivitätsdiagramme mit Kontrollstrukturen von den Probanden als korrekt akzeptiert wurden. Für eine vollständige Annahme müssten aber die 56 unbestimmten Diagramme entschieden werden. Durch die geringe Anzahl an Kontrollstrukturen im Korpus, kann es sein, dass die restlichen Kontrollstrukturen Teil der unbestimmten Diagramme sind.

7.3.5 Probleme im Aufbau der Studie

Der Wert für die berechnete Inter-Annotator-Übereinstimmung lag bei 0,32 (σ=0,10). Dies ist zwar eine ausreichende Übereinstimmung, liegt aber eher im niedrigeren Bereich. Des-

wegen gab es selbst nach der Verschmelzung für 56 (23,62%) Aktivitätsdiagramme durch den Mehrheitsentscheid kein Ergebnis. Dafür kann es einige Gründe geben. So wurden an die Probanden keine Teilnahmevoraussetzungen gestellt. Das bedeutet, dass Probanden mit verschiedenen Kenntnissen an der Studie teilnehmen konnten. Probanden könnten daher unterschiedliche Annahmen bezüglich Korrektheit haben. Ein Beispiel hierfür wäre die Annahme über das implizite Wissen des Roboters und seiner Umgebung. So gab es Probanden, die anzunehmen schienen, dass der Roboter in der Lage ist selbstständig Informationen zu implizieren. Im Gegenteil dazu gingen andere Probanden davon aus, dass dem Roboter Informationen explizit zur Verfügung gestellt werden müssen. Daher wurden in vollständig korrekten Diagrammen, die Einzelbefehle darstellten, von Probanden fehlende Aktivitäten erkannt. Hier hätte entweder die Einführung stärker auf die Fähigkeiten des Roboters eingehen müssen oder Probanden nach ihrem Kenntnisstand separiert werden. Da während der Studie keine persönlichen Informationen erhoben wurden, ist letzteres nachträglich nicht möglich. Ein weiteres Problem war die Belohnung zur erfolgreichen Abgabe der Studie. Es wurden 20 Minuten als Bearbeitungszeit geschätzt und unabhängig der Zeit, war die Belohnung immer gleich. Probanden konnten auch an mehreren Studien teilnehmen. Dies bedeutete, dass Personen dazu motiviert waren möglichst schnell viele Formulare auszufüllen. Die Qualität der Abgaben litt darunter, da die Personen die Aktivitätsdiagramme nicht extensiv untersuchten. Eine Umgestaltung der Studie mit einem Fokus auf Qualität könnte hier Abhilfe schaffen. Dies könnte erreicht werden, indem hochwertige Abgaben mit einer erhöhten Bezahlung belohnt werden. Auch kann ein Feld zur Erklärung der Entscheidung helfen die Abgabe nachzuvollziehen. Als letzter Grund wären auch Mehrdeutigkeiten zu nennen. So gab es Aktivitätsdiagramme, die je nach Interpretation unterschiedliche Antwortmöglichkeiten boten. Auch waren bestimmte Fehler auf mehrere Fehlerfälle übertragbar. Hier hätten Beispiele mit Fehlern die Probanden auf eine ähnliche Linie bringen können.

7.3.6 Evaluation der Methoden und Parameter

Die Korrektheit der Semantik basiert auf der Korrektheit der Methoden und ihrer Parameter. Um die Korrektheit zu bemessen wurde eine manuelle Evaluation der erzeugten Quelltexte durchgeführt. Um die Korrektheit der Semantik zu evaluieren, wurden Texte des PARSE-Korpus (siehe Abschnitt 3.3) zu Quelltext übersetzt. Die Aktivitäten entsprachen den von `MethodSynthesizer` gefundenen Methoden für das Zielsystem ARMAR-III. Von 238 Texten ließen sich 234 Befehlsbeschreibungen in PlantUML-Quelltext übersetzen und zu Aktivitätsdiagrammen kompilieren. Neben dem Absturz bei der Kontextanalyse, konnte zusätzlich `MethodSynthesizer` bei zwei Eingaben kein Ergebnis liefern. Gut hingegen funktioniert die Übersetzung des abstrakten Syntaxbaumes in Quelltext durch Besucher. Nur in einem der vier Texte für die keine Aktivitätsdiagramme erstellt werden konnten, wurde fehlerhafter PlantUML-Quelltext erstellt. Dies lag daran, dass in der Eingabe eine Parallelität entdeckt wurde, aber `MethodSynthesizer` keine Methoden in der Parallelität finden konnte. Dies führte dazu, dass syntaktisch unvollständiger Quelltext für die Parallelität erstellt wurde. Die 234 erzeugten Aktivitätsdiagramme wurden manuell ausgewertet. Dabei wurden die Fehler bezüglich der Methoden, Parameter und vollständigen Aufrufe separat gezählt. Von 237 Aktivitätsdiagrammen waren 77 (32,49%) Diagramme vollständig fehlerfrei. Dabei fällt auf, dass dies hauptsächlich Aktivitätsdiagramme mit weniger als sieben Aktivitäten zutraf. Enthielten Diagramme mehr als sieben Aktivitäten, so waren sie immer fehlerhaft. Weiterhin wurden 5 von 234 Befehlsbeschreibungen fehlerhaft als Lehrsequenz klassifiziert. Die Korrektheit von Lehrsequenzen hängt von der korrekten Erkennung durch `MethodSynthesizer` ab. Da die Werkzeuge die Ergebnisse nur übernehmen, gab es bei ihnen keine Probleme. Die Erkennung als Lehrsequenz führt dazu, dass bei der Injektion des Quelltext eine neue Methode angelegt wird, anstatt die gegebenen Befehle auszuführen. Wurde eine Lehrsequenz erkannt, so wird der erzeugte Quelltext in eine neue

Tabelle 7.7: Fehlerzählung und Gütekriterien bezüglich Methoden, Parameter und vollständiger Aufrufe.

Typ	Gesamt	r_p	f_p	f_n	Präzision	Ausbeute	F_1
Methoden	1402	1156	95	151	92,41%	88,45%	90,38%
Parameter	1572	1249	259	64	82,82%	95,13%	88,55%
Aufrufe	1402	920	331	151	73,53%	85,90%	79,24%

Tabelle 7.8: Kontrollstrukturfehler

Anzahl	Korrekt	Bedingung	Bezugsrahmen	Beides
73 (100%)	20 (27,40%)	51 (69,86%)	21 (28,77%)	19 (26,03%)

Methode in der Ausführungsschablone geschrieben. `MethodSynthesizer` ist auch dafür zuständig die neue Methodensignatur zu erzeugen. Problematisch ist hierbei insbesondere, dass in der Regel die gesamte Beschreibung dafür verwendet wurde die neue Methodensignatur zu erstellen. Dadurch werden keine Befehle gesucht und die neue Methode bleibt leer.

Die Ergebnisse der Evaluation sind in Tabelle 7.7 dargestellt. Insgesamt gab es 1402 Methoden und damit Aufrufe in den Befehlsbeschreibungen. Hiervon konnten 151 Methoden und Aufrufe durch `MethodSynthesizer` keiner Methode in der Ontologie zugewiesen werden. Zusätzlich wurde in 95 Fällen die falsche Methoden gefunden. `MethodSynthesizer` identifizierte daher in 1156 Fällen die korrekte Methodenbezeichnung. Insgesamt verteilten sich 1572 Parameter auf die 1402 Methoden. Hiervon wurden 1249 Parameter korrekt zugewiesen. Bei 64 Fällen fehlte der Parameter und in 259 Fällen wurde ein falscher Paramter gefunden. Dies führte dazu, dass 331 Aufrufe fehlerhaft waren. Es gab daher 920 vollständig korrekt identifizierte Aufrufe. Daraus ist ersichtlich, dass besonders bei Parametern eine niedrige Präzision von 82,82% erreicht wurde. Eine Ursache hierfür kann eine mangelhafte Auflösung der Koreferenzen sein. Die Ausbeute ist mit einem Wert von 95,13% hingegen deutlich höher. Daraus ergibt sich ein F_1-Maß von 88,55%. Den zweitgrößten Fehlerfall bilden die Methoden und Bedingungen. Hier wird eine Ausbeute von 88,45% und eine Präzision von 92,41% erreicht. Das F_1-Maß betrug 90,38%. Dabei unterscheiden sich Bedingungen und Methoden nach dem Ursprung der Lücke. Fehlt eine Methode so liegt das meist an einer fehlerhaften Zerteilung der Sätze in Aktionen. Bedingungen hingegen werden meist korrekt zerteilt. Stattdessen ist `MethodSynthesizer` nicht in der Lage, eine passende Methode der Schnittstelle in der Ontologie zu finden. Insgesamt konnte für vollständige Aufrufe eine Präzision von 73,54% und eine Ausbeute von 85,90% erreicht werden. Weiterhin ist in Tabelle 7.8 eine Aufteilung der Kontrollsturkturen nach Fehlerfällen ersichtlich. Obwohl es nur 73 Schleifen und Verzweigungen gab, werden in 51 (69,86%) davon die Bedingungen nicht gefunden. Der Bezugsrahmen der Kontrollstrukturen ist dabei in 21 (28,77%) von 73 Fällen fehlerhaft. Dabei gibt es eine Überschneidung beider Fehler in 19 (26,03%) Aktivitätsdiagrammen. Aktivitäten, die gefunden wurden, obwohl an der Stelle keine Aktion nötig ist stellt das drittgrößte Problem dar. Der technische Ursprung dieses Fehlers liegt in einer fehlerhaften Zerteilung der Sätze. So werden Zusatzinformationen als eigene Instruktionen interpretiert. Findet `MethodSynthesizer` eine passende Methode, die auf die Zusatzinformation passt, so wird eine unnötige Methode erzeugt. Ein weiter Ursprung dieser Fehler sind syntaktische Anweisungen. Die Referenzen auf vorherige Instruktionen werden meist als eigene Instruktionen erkannt und verarbeitet. Disfluenzen wurden auch oft als eigene Instruktionen auf unnötige Methoden abgebildet. Den geringsten Teil der Fehler machten die fehlerhaften Aktivitäten aus. Bei Bedingungen wurden hier in der Regel Methoden statt Bedingungen gefunden. Abseits der Bedingungen

lag der Ursprung dieser Fehler in der Regel darin, dass der Nutzer unklare Anweisungen gab. Auch gab es Fälle, bei denen die gewünschte Methode nicht existent war, aber eine andere Methode ähnlich, aber falsch war. Ein Beispiel hierfür ist „*empty the dishwasher*“, welches als `isEmpty(Dishwasher)` erkannt wurde.

Die implementierten Werkzeuge übernehmen die Ergebnisse der PARSE-Agenten und haben daher keinen Einfluss auf die Semantik des Quelltextes. Die Korrektheit der Methoden und Parameter wird daher von den Agenten von PARSE festgelegt. Bei vollständigen Methodenaufrufen konnte ein F_1-Maß von 79,24% erreicht werden. Bei Befehlsbeschreibungen mit weniger als sieben Methoden konnte in der Regel funktionsfähiger Quelltext erzeugt werden. Die Hypothese H_1 kann im Bezug auf die semantische Korrektheit daher angenommen werden. Schwierigkeiten gab es bei Kontrollstrukturen und langen Befehlsbeschreibungen. Bei Kontrollstrukturen konnte in 27,40% der Fälle bereits korrekter Quelltext erzeugt werden. Zeitgleich sind 69,86% der Kostrollstrukturfehler auf fehlerhafte Bedingungen, d.h. Methoden und Parameterfehler, zurückzuführen. Methoden- und Parameterfehler können durch eine Optimierung der Agenten und Ontologie reduziert werden. Trotzdem kann Hypothese H_2 bezüglich der Semantik nur teilweise angenommen werden. Zwar kann Quelltext mit Kontrollstrukturen synthetisiert werden, dieser kann aber noch Fehler enthalten.

7.3.7 Fehlerbeschreibung

Die implementierten Werkzeuge befinden sich am Ende der Verarbeitung durch PARSE und übernehmen die Ergebnisse und damit auch Fehler vorheriger Agenten. In diesem Abschnitt soll nach der quantitativen Evaluation ein qualitativer Überblick über Fehler und ihre Auswirkungen auf den Quelltext gegeben. Dabei soll auch versucht werden den Ursprung des Fehlers zu bestimmen. Anzumerken sei hierbei, dass Fehler in früheren Agenten zu Folgefehlern in späteren Agenten führen. Die hier beschriebenen Fehler sind daher teilweise Symptome und haben ihren tatsächlichen Ursprung in den früheren Agenten. Das vollständige Aufschlüsseln der Fehlerquellen erfordert detaillierte Einarbeitung in sämtliche Agenten von PARSE und übersteigt den Rahmen dieser Arbeit.

7.3.7.1 Syntaktische Fehler

Viele Agenten in PARSE sind dafür zuständig die Syntax aus natürlicher Sprache zu extrahieren. Dabei können an vielen Stellen Fehler entstehen, die zu einer fehlerhaften Syntax führen.

Ein Problem sind fehlende und fehlerhafte Zerteilung der Sätze in Instruktionen. Da alle Agenten diese Zerteilung als Grundlage für die Verarbeitung verwenden, führen Fehler hier zu einer großen Anzahl an Folgefehlern. Werden zwei Instruktionen nicht getrennt, so fehlt eine Instruktion. Daher wird zum Beispiel „*grab the water with one hand close the fridge with the other hand*“ als eine einzige Instruktion interpretiert. Die vereinten Instruktionen werden dann nur auf eine Methode `grab(WaterBottle)` abgebildet. Die Instruktion zum Schließen des Kühlschrankes geht dabei komplett verloren, wodurch die entsprechende Methode ebenfalls fehlt. Alternativ kann es bei fehlender Zerteilung auch vorkommen, dass Informationen beider Instruktion vermischt werden. Dies führt dazu, dass die erzeugte Methode fehlerhaft ist und und eine zweite Methode fehlt. Wird beispielsweise „*fill the water in the cup place the cup on the table*“ nicht zerteilt, so entsteht eine nicht durchführbare Methode `fill(Cup, Table)`. Zusätzlich fehlt hier die Methode `place(Cup,Table)`. Das gegenteilige Problem ist eine unnötige Zerteilung in mehrere Instruktionen. Ein Beispiel hierfür wären Sätze wie „*grab the juice located near the nesquik*“. Dies wird in die beiden Instruktionen „*grab the juice*“ und „*located near the nesquik*“ zerteilt. Die gegebene Information bezüglich der Position des Saftes ist hier nicht in der Methode `grab(Juice)`

enthalten. Zeitgleich wird aber aus „*located near the nesquik*“ eine nachfolgende Methode `locate(Nesquik)` erzeugt. Besonders auffällig ist, dass „*make sure to...*“ meist als eigene Instruktion erkannt wird. Sie wird in der Regel auf die Methode `make()` abgebildet.

Syntaktische Anweisungen, die sich auf vorherige Anweisungen beziehen, stellen ein Problem dar. Zum Beispiel enthält der Satz „*open the fridge grab the juice and bring it me but after you grab the juice close the fridge*“ zwei Fehler. Zuerst wird hier der Anweisungsbezug „*but after you grab the juice*“ als eigene Instruktion erkannt und eine neue Methode `grab(juice)` erstellt. Die Methode der Instruktion „*close the fridge*“ wird aufgrund fehlender Auflösung der temporalen Ausführungsreihenfolge ans Ende der Befehlssequenz gestellt. Syntaktische Anweisungen treten auch bei der Parallelität auf. Ein Beispiel hierfür wäre der Satz „*wash the cup while washing read the news*“, bei welchem zweimal die Methode `wash(...)` erzeugt wird.

Der fehlerhafte Bezugsrahmen von Schleifen und Verzweigungen ist ein auf Mehrdeutigkeit beruhendes Problem. Das Problem soll beispielhaft an der Befehlsbeschreibung „*if you meet someone look him in the eye wave your hand while greeting him*“ verdeutlicht werden. Es besteht die Möglichkeit, dass nur einige Phrasen Teil der Verzweigung sind und die restlichen Anweisungen nachfolgende Befehle darstellen. Die Entscheidung, welche Phrasen Teil der Verzweigung sind und welche Phrasen nachfolgenden Befehlen entsprechen ist syntaktisch mehrdeutig. Durch sein kulturelles Umfeld kann ein Mensch lernen, dass alle drei Befehle Teil des gleichen Vorgangs sind. Es wäre also implizites Wissen oder ein semantisches Verständnis der Sprache nötig, um den Bezugsrahmen korrekt aufzulösen. Ein ähnliches Problem entsteht auch bei den Bedingungen der Schleifen und Verzweigungen. So werden in 4 Aktivitätsdiagrammen Instruktionen, die nicht Teil der Bedingung sind, von den Agenten als Teil dieser erkannt. Die Instruktionen werden dann auf Methoden abgebildet, die keinen Rückgabetyp besitzen. Dadurch ist der Syntaxbaum und somit der erzeugte Quelltext syntaktisch nicht korrekt. Verschachtelte Verzweigungen sind ein weiteres Problem. In PARSE werden die Informationen bezüglich Verzweigungen in den Token als Attribute gespeichert. Da das Attribut einzigartig ist, kann ein Token nur eine einzige Verzweigung deklarieren. Dadurch sind verschachtelte Verzweigungen nicht möglich. Der Syntaxbaum wäre aber in der Lage diese zu modellieren. Die Ursprung dieser Fehler liegt in den Agenten `ConditionDetection` und `LoopDetection`.

Negierungen sind ebenfalls problematisch. Bei Teilsätzen wie „*don't press the red button*“, „*if they aren't dirty*“ oder „*bring the non empty cup*“ wird die Negierung nicht erkannt. Dies liegt daran, dass kein Agent zur Erkennung von Negationen existiert. `MethodSynthesizer` bildet die Teilsätze ebenfalls ohne die Negation zu erfassen auf Methoden der Schnittstelle ab. Dadurch wird an dieser Stelle nicht nach möglichen Antonym-Methoden gesucht. So wäre die gesuchte Methode beim Teilsatz „*if the dishes aren't clean*“ eigentlich `isDirty()`. Stattdessen wird auf `isClean()` abgebildet und die Negation ignoriert.

Eine Ausnahme sind Negierung von Schleifenbedingungen. Bei Schleifen kann zwischen Abbruch- und Wiederholungsbedingungen unterschieden werden. Bei Abbruchbedingungen wie „*repeat this until the dishes are clean*“ führt die positive Auswertung der Bedingung zu einem Verlassen der Schleife. Dahingegen wird die Schleife bei Wiederholungsbedingungen wie „*repeat this while the dishes are clean*“ bei positiver Auswertung der Bedingung wiederholt. Zur Unterscheidung beider Arten dient das Signalwort „*until*“. Wird es bei einer Schleife entdeckt, so wird die Bedingung negiert.

7.3.7.2 Semantische Fehler

In der Ontologie sind alle Methoden der Anwendungsschnittstelle gespeichert. `MethodSynthesizer` nutzt die Ontologie, um den Befehlen in natürlicher Sprache äquivalente Methoden zuzuordnen. Beim Aufbau des Syntaxbaumes werden die von `MethodSynthesizer`

erzeugten Methoden und Parameter von den implementierten Werkzeugen übernommen. Dies führt dazu, dass die Werkzeuge selbst nur geringen Einfluss auf die Korrektheit der Semantik haben. Deshalb sollen hier Fehler die Auswirkungen auf Methoden und Parameter haben vorgestellt werden.

Eine Fehlerquelle stellt eine lückenhafte Ontologie dar, d.h. eine Ontologie, die nicht alle Instruktionen und Entitäten enthält. Dies äußert sich oft darin, dass bei der Übersetzung des Befehls in natürlicher Sprache keine Methode durch `MethodSynthesizer` gefunden wird. Die Werkzeuge können in dem Fall für den abstrakten Syntaxbaum das entsprechende Blatt nicht erzeugen. Es fehlt also eine Methode, was die semantische und syntaktische Korrektheit des erzeugten Quellcodes zerstört. Ein Beispiel hierfür wäre „*place the green cup ...*“, welches auf keine passende Methode abgebildet werden konnte. Die Auswirkungen dieses Fehlerfalls auf den Syntaxbaum lassen sich noch in zwei Fälle einordnen: Ist die Methode Teil einer Sequenz, so fehlt sie vollständig. Der entsprechende Syntaxbaum ist aber in lauffähigen Quelltext übersetzbar. Eine syntaktische Analyse des Syntaxbaumes kann diesen Fehler daher nur schwer erkennen. Alternativ kann die fehlende Methode eine Bedingung wie „*if there are dirty dishes*“ sein. In dem Fall fehlt bei einer Verzweigung oder Schleife die Bedingung. Der aufgebaute Syntaxbaum ist in dem Fall syntaktisch nicht korrekt. Dies lässt sich früh durch eine statische Analyse erkennen. Deshalb sind fehlende Methoden besonders bei Bedingungen bemerkbar.

Neben der fehlenden Methode, kann auch ein Befehl in natürlicher Sprache von `MethodSynthesizer` der falschen Methode zugeordnet worden sein. Dies kann passieren wenn zwei Methoden oder ihre Synonyme sich ähneln. Ein Beispiel hierfür wäre „*pull it out of the fridge*“ welches auf die Methode `put(...)` abgebildet wird. Wird die falsche Methode höher bewertet oder fehlt die korrekte Methode in der Ontologie, so erzeugt `MethodSynthesizer` an der Stelle einen Fehler. Der Fehler wird dann von den Werkzeugen übernommen. Die falsche Methode lässt sich in der Regel nicht analytisch erkennen, da der erzeugte Syntaxbaum sich in kompilierbaren Quelltext übersetzen lässt. Erst zur Laufzeit wird durch den Menschen ersichtlich, dass falsche Aktionen durchgeführt werden. Eine Ausnahme bildet hierbei aber erneut die Bedingung. Eine Methode, die als Bedingung verwendet wird, muss einen booleschen Rückgabetyp besitzen. Besitzt die fehlerhafte Bedingung ebenfalls einen booleschen Rückgabetyp, so ist der Fehler weiterhin nicht erkennbar. Andererseits führt das Einsetzen einer Methode ohne booleschen Rückgabetyp zu einem Fehler während der statischen Analyse. Es wäre also möglich diesen Fehler beim Aufbau des Syntaxbaumes zu erkennen. Dazu müssten die Methoden aber mit Hilfe der Ontologie überprüft werden, da der PARSE-Graph den Rückgabetyp der gefundenen Methoden nicht speichert. So wird zum Beispiel statt der Bedingung `isClosed()` durch ihre Ähnlichkeit die Methode `close()` eingesetzt. Dies führt zu einem fehlerhaften Syntaxbaum. Da dies besonders bei Kontraktionen wie „*it's closed*“ vorkam, wurden diese einfach durch eine Heuristik zu „*it is closed*“ umgewandelt.

Die Argumente der Methoden werden durch `MethodSynthesizer` ebenfalls aus der Ontologie extrahiert und durch die Werkzeuge übernommen. Da Methoden überladen sein können und so eine unterschiedliche Anzahl an Parametern besitzen können, ist es möglich, dass die Methode mit einer geringeren Anzahl an Parametern gewählt wird. Dadurch gehen Informationen zur Ausführung der Methode verloren. Weiterhin werden die Parameter der Methoden oft durch allgemeine Typen wie `Grabable` und `Object` repräsentiert. Dadurch ist es möglich, dass verschiedene Objekte als Argument in Frage kommen. Wie auch bei Methoden, kann es hierbei zu falschen Argumenten kommen. Auch gibt es Methoden bei denen die Parameter mehrfach den gleichen allgemeinen Typ besitzen. Dies führt zu der Besonderheit, dass die Argumente für alle Parameter gleich bewertet werden. Die Reihenfolge kann dadurch fehlerhaft sein. Weiterhin fällt auf, dass bei Parametern keine Unterscheidung aufgrund der Eigenschaften der Entititäten durch `MethodSynthesizer`

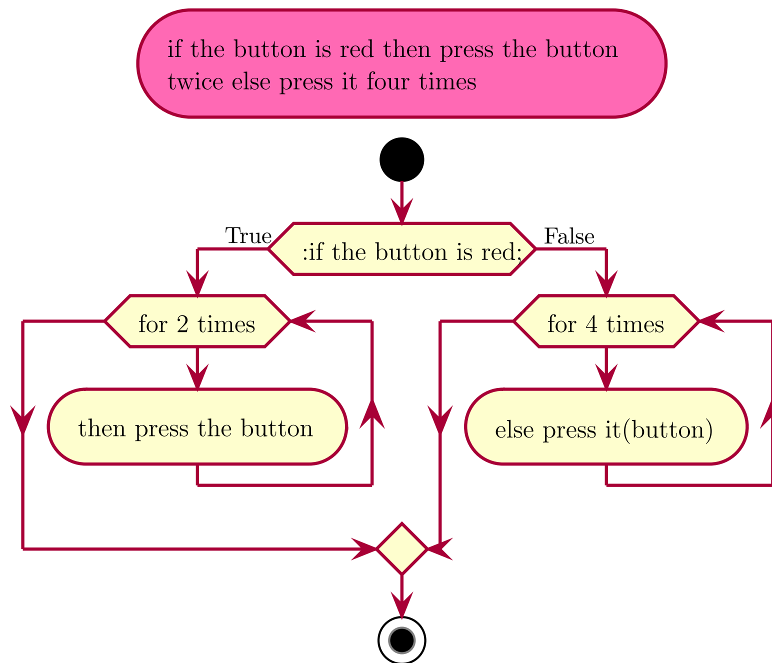


Abbildung 7.1: Beispiel eines durch PlantUML erzeugten Aktivitätsdiagramm, welches auf natürlicher Sprache basiert

stattfindet. So wird bei „*bring me the green cup*“ nur die Methode `bring1(Cup)` erzeugt. Dadurch wird nicht zwischen mehreren Entitäten unterschieden. Dies führt auch dazu, dass Zusatzinformationen, wie „*table at the window*“ für Parameter nicht verarbeitet werden. Bei Sätzen wie „*put it on to the table in the kitchen*“ können diese Zusatzinformationen zu falschen Parametern führen.

7.4 Syntaxbäume

In dieser Arbeit wurden abstrakte Syntaxbäume als Zwischenrepräsentation zur „zielsystemunabhängigen Quelltextsynthese aus natürlicher Sprache“ verwendet. In diesem Abschnitt soll Hypothese H_3 durch Argumente untermauert werden.

Befehlsanweisungen in natürlicher Sprache implizieren eine Abfolge von Aktivitäten. Die Abfolge der Aktivitäten wird von Verzweigungen, Schleifen, Parallelität und Sequenzen gesteuert. Sie lässt sich beispielsweise als Aktivitätsdiagramm darstellen, welches einem Syntaxbaum entspricht. Durch die Extraktion von Informationen über die Kontrollstrukturen aus natürlicher Sprache werden also Informationen über einen Syntaxbaum gesammelt. Diese Informationen sind nötig, um Quelltext aus natürlicher Sprache zu synthetisieren. Um möglichst fehlerfreien Quelltext zu erzeugen, muss die Menge an verfügbaren Syntaxbaum-Informationen maximiert werden. Idealerweise werden dabei ausreichend Informationen gesammelt, um einen vollständigen abstrakten Syntaxbaum aufzubauen. Das Ziel des Sprachverständnis zur Synthese von Quelltext ist daher bereits die Synthese eines Syntaxbaumes.

Wurde aus natürlicher Sprache ein Syntaxbaum erzeugt, so entstehen dadurch einige Vorteile. Ein großer Vorteil von Syntaxbäumen ist ihre Unabhängigkeit von einer konkreten Programmiersprache. Das bedeutet, dass sich aus dem gleichen Syntaxbaum Quelltext für unterschiedliche Programmiersprachen generieren lässt. Durch die Verwendung von Syntaxbäumen wird auch das Zielsystem von der natürlichen Sprache separiert. Der Syntaxbaum enthält ausreichend Informationen, um ihn leicht durch Besucher zu Quelltext

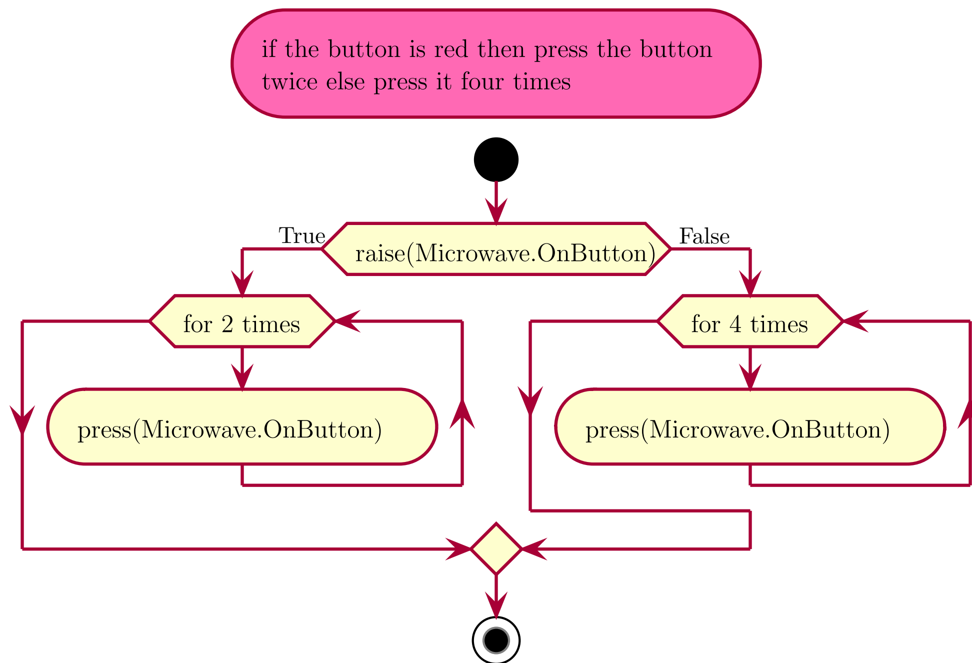


Abbildung 7.2: Beispiel eines durch PlantUML erzeugten Aktivitätsdiagramm, welches die von `MethodSynthesizer` erzeugten Methoden verwendet. Automatisch erzeugte Kommentare wurden zur besseren Lesbarkeit entfernt.

zu übersetzen. Ein erneuter Zugriff auf die aus natürlicher Sprache extrahierten Informationen ist nicht nötig. Der Aufwand, der durch die Verwendung von Syntaxbäumen als Zwischenrepräsentation entsteht, wird also durch den fehlenden Einfluss des Zielsystems ausgeglichen. Ein weiterer Grund, welcher für die Verwendung von Syntaxbäumen spricht, ist die Möglichkeit die syntaktische Korrektheit der Syntaxbäume bei der Erzeugung überprüfen zu können. Dazu können Analyseverfahren aus dem Übersetzerbau verwendet werden. So lassen sich Fehler bereits während dem Erstellen des Syntaxbaumes erkennen.

Die Übersetzung von natürlicher Sprache zu Quelltext mit Syntaxbäumen als Zwischenrepräsentation findet in zwei Phasen statt. Trotzdem ist der daraus entstehende Aufwand gering, da zur direkten Synthese des Quelltextes bereits Informationen über den Syntaxbaum aus der natürlichen Sprache gesammelt werden. Zeitgleich bieten Syntaxbäume als Zwischenrepräsentation einige Vorteile gegenüber der direkten Übersetzung in Quelltext. Syntaxbäume scheinen daher als Zwischenrepräsentation geeignet. Dies unterstützt Hypothese H_3.

7.5 Zielsystemunabhängigkeit

Ein Ziel dieser Arbeit war es sicherzustellen, dass die Werkzeug unabhängig vom Zielsystem eingesetzt werden können. Um zu zeigen, dass dies erreicht wurde, müssen die Einflüsse des Zielsystems auf die Werkzeuge offengelegt werden. Zuerst sollen daher die möglichen Einflüsse des Zielsystems aufgezählt werden. Einen starken Einfluss bildet die Methoden der Schnittstelle und Domäne des Zielsystems. In PARSE wird beides durch eine Ontologie repräsentiert. Die Ontologie hat großen Einfluss auf die PARSE-Agenten und insbesondere den `MethodSynthesizer`. Die Ontologie wird aber von den in dieser Arbeit erzeugten Werkzeugen nicht verwendet. Die Werkzeuge verwenden nur die Ergebnisse der Agenten von PARSE. Die Ergebnisse bilden somit eine Schicht zwischen der Ontologie und den Werkzeugen. Da die Ontologie des Zielsystems keinen direkten Einfluss auf die Werkzeuge hat, kann die Schnittstelle und die Domäne ausgetauscht werden ohne die Werkzeuge

Tabelle 7.9: Quelltextzeilen für implementierte Besucher-Klasse

Besucher	Quelltextzeilen	Bemerkungen
`PseudoCodeVisitor`	146	
`PlantUMLCodeVisitor`	212	
`AbstractVisitor`	71	Implementiert Einrückung
`PythonCodeVisitor`	157	Erbt von *`AbstractVisitor`* und ohne Parallelität
`CStyleVisitor`	142	Erbt von *`AbstractVisitor`*
`CCodeVisitor`	25	Erbt von *`CStyleVisitor`*
`JavaCodeVisitor`	30	Erbt von *`CStyleVisitor`*

ändern zu müssen. Der synthetisierte abstrakte Syntaxbaum ist also unabhängig von der Ontologie. Bei der Übersetzung in Quelltext hat die Programmiersprache und Verwendung der Schnittstelle Einfluss auf den Quelltext. Da der Quelltext vom Zielsystem ausgeführt wird, lässt sich dieser Einfluss nicht beseitigen. Es ist daher wichtig, dass eine schnelle und leichte Adaption auf andere Programmiersprachen möglich ist. Dies wird durch die Verwendung von Besuchern und eine Separierung der Quelltextübersetzung und Einspeisung ins Zielsystem erreicht. Dadurch ist zur Erzeugung des Quelltextes für ein Zielsystem durch Besucher nur die Programmiersprache des Zielsystems relevant. Für Zielsysteme mit gleicher Programmiersprache kann daher ein implementierter Besucher ohne Änderung wiederverwendet werden. Für Zielsysteme mit neuen Programmiersprachen können beispielsweise syntaktische Gemeinsamkeiten in abstrakten Besuchern gebündelt und wiederverwendet werden. Dies erlaubt eine schnelle Adaption der Quelltexterzeugung auf neue Zielsysteme. Der abstrakte Syntaxbaum enthält ausreichend Informationen, um ihn in eine Vielzahl an Programmiersprachen zu übersetzen. Um Quelltext für weitere Programmiersprachen zu erzeugen, reicht die Implementierung neuer Besucher. Änderungen an dem Werkzeug zur Synthese des Syntaxbaumes sind nicht nötig. Für Zielsysteme mit gleicher Programmiersprache können einmal implementierte Besucher direkt wiederverwendet werden. Während der Implementierung wurden Besucher für die Programmiersprachen Java, Python und C, Pseudo-Quelltext und PlantUML erzeugt. Besucher konnten meist innerhalb einer Stunde implementiert werden. In Tabelle 7.9 ist die Anzahl an Quelltextzeilen für die Besucher dargestellt. Hierbei sei noch auf Quelltextausschnitt 6.1, Quelltextausschnitt 6.4, Quelltextausschnitt 6.6 und Quelltextausschnitt 6.7 für Beispiele des Quelltextes der Besucher verwiesen. Folglich ist der Einfluss des Zielsystems bei der Programmiersprache gering. Um die Wiederverwendbarkeit der Besucher für Programmiersprachen zu verbessern, findet das Injizieren des Quelltextes in einer separaten Fließbandstufe statt. Durch das verwendete Schablonen-System muss nur die Ausführungsdatei-Schablone für das Zielsystem angepasst werden, in welche der Quelltext geschrieben wird. Der Einfluss des Zielsystems auf die Übersetzung des abstrakten Syntaxbaumes in Quelltext ist zwar vorhanden, lässt sich aber leicht handhaben. Aus softwaretechnischer Sicht ist der Einfluss des Zielsystems auf die implementierten Werkzeuge minimal. Es nur geringer Aufwand nötig ist, um weitere Zielsysteme einzubinden. Daher ist der Einfluss des Zielsystems auf die entworfenene Verfahren ebenfalls gering.

In Abschnitt 7.3.4 wurde eine Online-Studie und in Abschnitt 7.3.6 eine manuelle Evaluation von Hand durchgeführt. Die Online-Studie basierte auf vom Zielsystem unabhängigen Aktivitätsdiagrammen mit natürlicher Sprache. Bei der manuellen Evaluation wurden hingegen Methoden für ein konkretes Zielsystem erzeugt. Anhand einem Vergleich beider Studie dürfte sich daher der Einfluss des Zielsystems auf die Korrektheit bemessen lassen. In Tabelle 7.10 wird die vollständige Korrektheit der Aktivitätsdiagramme in Bezug gesetzt. Dabei entspricht die Differenz von 36 (15,19%) fehlerhaften Diagrammen dem

Tabelle 7.10: Differenz zwischen Online-Studie und Methodenevaluation.

Typ	Korrekt	Anteil
Online-Studie	113	47,68%
Methodenevaluation	77	32,49%
Differenz	36	15,19%

Einfluss des Zielsystems. Das bedeutet, dass 31,86% der Fehler auf den Einfluss des Zielsystems zurückzuführen sind. Im Vergleich zur Online-Studie wäre das Zielsystem damit die größte einzelne Fehlerquelle. Zeitgleich ist der Einfluss des Zielsystems auf Methoden und Parameter beschränkt. Bei der Messung muss aber erwähnt werden, dass bei den Einflüssen das Zielsystem nicht vollständig isoliert wurde. Das bedeutet, dass es weitere Einflüsse gibt, die diesen Unterschied erklären können. Die Erzeugung von Methoden und Paramter geschieht durch `MethodSynthesizer`. Dies führt dazu, dass bestimmte Fehler nur in der manuellen Evaluation vorkommen konnten. So war beispielsweise eine fehlerhafte Erkennung von Lehrsequenzen bei der Erzeugung der Aktivitätsdiagramme für die Online-Studie nicht möglich. Ein Teil der fehlerhaften Diagramme ließen sich womöglich dadurch erklären und somit den bemessenen Einfluss des Zielsystems weiter senken. Ein weiterer Einfluss sind die unterschiedlichen Evaluatoren beider Studie. Während bei der Online-Studie jeweils fünf Probanden die Korrektheit durch einen einfachen Mehrheitsentscheid feststellten, wurde dies bei der manuellen Evaluation nur durch einen Probanden durchgeführt. Das bedeutet, dass Annahmen über die Korrektheit bei der manuellen Evaluation starken Einfluss haben könnten. Weiterhin war die Übereinstimmung der Probanden nicht immer eindeutig. So konnten 56 (23,62%) der Fälle nicht durch einen Mehrheitsentscheid entschieden werden. Daher ist die Messung des Einflusses des Zielsystems nicht endgültig zu sehen.

Weiterhin wurde die Zielsystemunabhängigkeit praktisch überprüft. Dazu wurde mit PARSE für zwei verschiedene Zielsysteme Quelltext aus natürlicher Sprache übersetzt. Für das erste Zielsystem, den Küchenroboter Armar, wurde Java-Quelltext erstellt. Ein Quelltextbeispiel ist in Quelltextausschnitt 7.1 zu sehen. Die Parallelität wurde durch Java-Threads modelliert. Als Beispiel für die einfache Transferierbarkeit auf andere Sprachen wurde in Quelltextausschnitt 7.2 die gleiche Befehlsbeschreibung in C-Quelltext übersetzt. Dabei wurde zur Modellierung der Parallelität OpenMP verwendet. Zusätzlich ist in Quelltextausschnitt 7.4 eine Version mit Python-Quelltext dargestellt. Hierbei wurde die Parallelität ausgelassen, da die aktuelle Erzeugung von Quelltext für Python das nicht unterstützt. Das zweite Zielsystem war ein programmierbarer Legoroboter. Der Besucher für Java konnte zur Erzeugung des Quelltextes wiederverwendet werden. Ausgewechselt wurde die Schablone für die Ausführungsdatei und die verwendete Ontologie. In Quelltextausschnitt 7.3 ist erzeugter Quelltext für den Legoroboter zu sehen. Als drittes Zielsystem fungierte die Erzeugung von Aktivitätsdiagrammen mit PlantUML. Die Aktivititätsdiagramme konnten dabei auf den von `MethodSynthesizer` gefundenen Methoden oder natürlicher Sprache basieren. Ein Quelltextbeispiel für PlantUML mit Methoden als Aktivitäten ist in Quelltextausschnitt 7.5 dargestellt. Wurde natürliche Sprache verwendet, so konnte auch eine Funktion der Werkzeuge unabhängig der konkreten Ontologie gezeigt werden. Ein Quelltextbeispiel hierfür ist in Quelltextausschnitt 7.6 gezeigt. Hypothese H_4 kann daher teilweise angenommen werden. Zwar konnte keine Unabhängigkeit vom Zielsystem erreicht werden, aber der Einfluss des Zielsystems ist gering.

7.6 Zusammenfassung und Fazit

Die Synthese von Quelltext aus natürlicher Sprache war in fast allen Fällen möglich. Von 238 Befehlsbeschreibungen konnte nur eine Eingabe nicht in syntaktisch korrekten Quell-

text umgewandelt werden. Die gewählten Lösungsansätze führten daher grundsätzlich zu Erfolg. Die verwendeten Mustererkenner waren in der Lage sämtliche Strukturen des Syntaxbaumes zu extrahieren, sofern die Agenten von PARSE diese erkannten. Aufgrund Folgefehlern konnte Quelltext in 32,49% mit korrekter Syntax und Semantik erzeugt werden. Die semantische Korrektheit wurde durch Fehler reduziert, die sich in `MethodSynthesizer` materialisierten. Diese Fehler führten dazu, dass entweder Methoden im Quelltext nicht gefunden oder fehlerhaft zugewiesen wurden. Ohne Einfluss des Zielsystems wurden 47,88% der Texte durch einen Mehrheitsentscheid zwischen fünf Probanden als vollständig korrekt übersetzt akzeptiert. Syntaktische Fehler waren in der Regel ebenfalls auf die Agenten von PARSE zurückzuführen. Da die Werkzeuge die Ergebnisse dieser übernahmen, wurden auch Fehler übernommen. Nicht bearbeitet werden konnte das Problem der Ausführungsreihenfolge, da hierfür kein Agent existierte. Das Werkzeug kann zur Generierung von Aktivitätsdiagrammen genutzt werden, die Aktivitäten in natürlicher Sprache beschreiben. Dies zeigt, dass eine Übersetzung der natürlichen Sprache in Quelltext auch für Befehlssequenzen mit Kontrollstrukturen möglich ist.

Abstrakte Syntaxbäume als Zwischenrepräsentationen scheinen eine geeignete Wahl zu sein. Sie können über Besucher in Quelltext übersetzt werden, was die Übersetzung in verschiedene Programmiersprachen vereinfacht. Das Implementieren neuer Besucher dauert in der Regel nur eine Stunde. Die Verwendung von Besuchern und Schablonen erlaubt auch eine starke Unabhängigkeit vom Zielsystem. Da die Erzeugung des Quelltextes von der Injektion des Quelltext getrennt ist, können die Besucher leicht wiederverwendet werden. Dazu ist es nur nötig die Schablone für das Zielsystem anzupassen. Um die Unabhängigkeit vom Zielsystem zu zeigen, wurde Quelltext für einen Küchenroboter, einen Legoroboter und zur Erzeugung von Aktivitätsdiagrammen synthetisiert. Neben der Erzeugung von Quelltext welcher zur Ausführung dient, konnte auch die Anwendungsschnittstelle erweitert werden. Die Erweiterung der Schablone um private Methoden war hierzu ausreichend. Es wäre auch möglich erzeugte Methoden in die Ontologie zu schreiben. Dies kann durch Modifizierung von `CodeInjector` oder einer weiteren Fließbandstufe geschehen.

```java
import edu.kit.ipd.*;
//here be imports

class TestClass {

//here be variables
Dishes Dishes = Kitchen.DISHES;
NewsPaper News = Kitchen.NEWS;
Juice OrangeJuice = Kitchen.ORANGEJUICE;


// New Methods:

    public static void main(String[] args) {

// hey armar check if the dish is dirty then
//wash the dish twice after that get me
//orange juice while reading the news for
//me
// Generated Code:
        if (isDirty(Dishes))
                {
                for(int iter0=0;iter0 < 2;              iter0++) {
                        // then wash the dish twice
                        wash(Dishes);
        }
        }       else {      }
    //PARALLEL:
        //SECTION 1:
        Thread t0= new Thread() { public void run() {
                // after that get me orange juice
                get1(OrangeJuice);
        }};
        t0.start();
        //SECTION 2:
        Thread t1= new Thread() { public void run() {
                // while reading the news for me
                read(News);
        }};
        t1.start();
        //END PARALLEL
        t0.join();
        t1.join();


    }

// New Methods:

}
```

Quelltextausschnitt 7.1: Beispiel für erzeugten Java-Quelltext aus „*hey armar check if the dish is dirty then wash the dish twice after that get me orange juice while reading the news for me*“. Die Variablen mussten in die Schablone geschrieben werden. Für die Parallelität wurden Threads verwendet.

```
#include "header.h"
//here be imports

//here be variables
Dishes Dishes = Kitchen.DISHES;
NewsPaper News = Kitchen.NEWS;
Juice OrangeJuice = Kitchen.ORANGEJUICE;

int main(int argc, const char* argv[]) {

// hey armar check if the dish is dirty then
//wash the dish twice after that get me
//orange juice while reading the news for
//me
// Generated Code:
    if (isDirty(Dishes))
    {
        for(int iter0=0;iter0 < 2;iter0++) {
        // then wash the dish twice
        wash(Dishes);
        }
    } else {}
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // after that get me orange juice
            get1(OrangeJuice);
        }
        #pragma omp section
        {
            // while reading the news for me
            read(News);
        }
    }


}
```

Quelltextausschnitt 7.2: Beispiel für erzeugten C-Quelltext aus „*hey armar check if the dish is dirty then wash the dish twice after that get me orange juice while reading the news for me*“. Die Variablen mussten in die Schablone geschrieben werden. Die Parallelität wurde mit OpenMP modelliert.

```java
import edu.kit.ipd.lego.*;
// here be imports

class MyLegoRobot extends LegoRobot  {

// here be variables

    public static void main(String[] args) {
        LegoRobot robo = new MyLegoRobot();
        try {
            robo.execute();
        } catch (Exception e) {
            System.err.println("Error");
        }
        robo.close();
    }

@Override
public void execute(){

// Robo turn right twice grab the can then lower
//your head lift your head again then
//belch and throw it away then say goodbye
// Generated Code:
        for(int iter0=0;iter0 < 2;      iter0++) {
                // Robo turn right
                turn(right);
    }
        // twice grab the can
        grab();
        // then lower your head
        lowerHead();
        // lift your head again
        liftHead();
        // then belch
        belch();
        // and throw it away
        throwAway(backward);
        // then say goodbye
        sayGoodbye();


}

// New Methods:

}
```

Quelltextausschnitt 7.3: Beispiel für erzeugten Java-Quelltext für einen Legoroboter aus „*Robo turn right twice grab the can then lower your head lift your head again then belch and throw it away then say goodbye*“. Der automatisch erzeugte Quelltext steht in der Methode `execute`. Der restliche Quelltext wurde von der Schablone übernommen.

```
import KIT
# here be imports

# here be variables
Dishes = Kitchen.DISHES;
News = Kitchen.NEWS;
OrangeJuice = Kitchen.ORANGEJUICE;

# hey armar check if the dish is dirty then
#wash the dish twice otherwise read the
#news for me
# Generated Code:
if (isDirty(Dishes)):
        for x in range(2):
                # then wash the dish twice
                wash(Dishes)

else :
        # otherwise read the news for me
        read(News)



#
```

Quelltextausschnitt 7.4: Beispiel für erzeugten Python-Quelltext aus „*hey armar check if the dish is dirty then wash the dish twice otherwise read the news for me*“.

```
@startuml

#HotPink:hey armar check if the dish is dirty then
wash the dish twice after that get me orange
juice while reading the news for me;
detach

start

'Generated Code:
if (isDirty(Dishes)) then (True)
while ( for 2 times )
:wash(Dishes);
floating note left: then wash the dish twice

endwhile


else (False)
endif

fork
:get1(OrangeJuice);
floating note left: after that get me orange juice

fork again
:read(News);
floating note left: while reading the news for me

end fork


stop

'

@enduml
```

Quelltextausschnitt 7.5: Beispiel für erzeugten PlantUML-Quelltext aus „*hey armar check if the dish is dirty then wash the dish twice after that get me orange juice while reading the news for me*“. Die Aktivitäten wurden zu Methoden übersetzt.

```
@startuml

#HotPink:hey armar check if the dish is dirty then
wash the dish twice after that get me orange
juice while reading the news for me;
detach

start

'Generated Code:
:hey armar check;
if (:if the dish is dirty;) then (True)
while ( for 2 times )
:then wash the dish;
endwhile


else (False)
endif

fork
:after that get me(SPEAKER) orange juice;
fork again
:while reading the news for me(me);
end fork


stop

'

@enduml
```

Quelltextausschnitt 7.6: Beispiel für erzeugten PlantUML-Quelltext aus „*hey armar check if the dish is dirty then wash the dish twice after that get me orange juice while reading the news for me*“. Aktivitäten enthalten Text in natürlicher Sprache.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Thema der zielsystemunabhängigen Quelltextsynthese aus natürlicher Sprache behandelt. Nach einer Recherche bezüglich verwandter Arbeiten wurde die Problemstellung analysiert. Bei der Analyse wurde ein Verfahren entworfen, welches die Ergebnisse der Rahmenarichtektur PARSE in einem abstrakten Syntaxbaum akkumuliert und diesen Baum in einer zweiten Phase in Quelltext übersetzt. Die Synthese des Syntaxbaum geschieht in drei Schritten: Zuerst werden Mustererkenner dazu verwendet Strukturen des Syntaxbaum zu extrahieren. Die Strukturen werden im zweiten Schritt verbunden und gemäß der Ausführungsreihenfolge sortiert. Die Verbindung der Strukturen geschieht durch den Aufbau aller Pfade von den Blättern zur Wurzel. Um mögliche Fehler bereits frühzeitig zu erkennen, wird der erzeugte Syntaxbaum in einem letzten Schritt analysiert. Der Syntaxbaum wird durch Besucher in Quelltext für unterschiedliche Zielsysteme übersetzt. Als Zielsysteme dienten ein Küchenroboter, ein Legoroboter und die Erzeugung von Aktivitätsdiagrammen. Dabei wurden Besucher für die Sprachen Java, Python, PlantUML und C implementiert.

Die implementierten Werkzeuge wurden anhand einer Online-Studie evaluiert. Bei dieser Studie markierten 120 Probanden Fehlerfälle in 237 automatisch erzeugten Aktivitätsdiagrammen. Dabei wurde jedes Aktivitätsdiagramm von jeweils fünf Probanden untersucht. Die Inter-Annotator-Übereinstimmung der Studie betrug im Durchschnitt 0,32, bei einer Standardabweichung von 0,10. Durch einen Mehrheitsentscheid wurden in 47,68% der Aktivitätsdiagramme keine Fehler entdeckt. Fehler teilten sich in 14,77% Aktivitätsfehler, 13,08% Reihenfolgefehler und 5,91% Kontrollstrukturfehler auf. Für 23,62% der Diagramme konnten durch einen einfachen Mehrheitsentscheid keine Aussage getroffen werden. Weiterhin wurde eine manuelle Evaluation mit Fokus auf die Methoden durchgeführt. Dabei gab es verschiedene Probleme, die dazu führten, dass nur 32,91% aller Befehlsbeschreibungen vollständig korrekt in Quelltext übersetzt werden konnten. Korrekt übersetzt wurden vor allem Befehlssequenzen ohne Kontrollstrukturen mit weniger als sieben Befehlen. Zwar wurde die Existenz von Kontrollstrukturen in der Regel korrekt erkannt und übersetzt. Die meisten Kontrollstrukturen enthielten aber Fehler bezüglich ihrer Bedingung oder dem Bezugsrahmen. So wurden Bedingungen nur in 30,14% der Fälle korrekte Methoden zugewiesen. Zeitgleich wurden in 28,77% der Fälle Befehle dem Bezugsrahmen der Kontrollstruktur fehlerhaft hinzugefügt oder ausgelassen. Bei den Methodenaufrufen wurde eine Präzision 73,54% und eine Ausbeute von 85,90% erreicht. Als größte Fehlerquelle ließen sich die Parameterfehler identifizieren, d.h. fehlende und fehlerhafte Parameter der Methoden. Bei Parametern war die Präzision bei 82,82% und die Ausbeute bei 95,13%.

Weiterhin konnten nicht immer alle Befehle gefunden werden und es wurden auch zusätzliche Befehle gefunden. So lag für Methoden, bei einer Ausbeute von 88,45%, die Präzision bei 92,41%. Gründe hierfür waren meist eine fehlerhafte Zerteilung der Eingabe in einzelne Befehle oder fehlende Methoden in der Ontologie.

Auf dieser Arbeit aufbauend lassen sich weitere Forschungsthemen vorstellen: Als erstes könnte eine Verbesserung der Agenten von PARSE die Qualität des erzeugten Quelltextes steigern. Hier wären die Methodensynthese und die Zerteilung hervorzuheben. Bei der Methodensynthese wäre es nötig, dass auch der Akteur gefunden wird. Erst dadurch wäre eine vollständige Objektorientierung möglich. Aber auch neue Werkzeuge ließen sich in PARSE einbauen. So würde die Erweiterung um einen Agenten, der die temporale Reihenfolge feststellt bereits zu einer Reduktion von fehlerhaften Sequenzen führen. Hierbei wäre es auch wichtig, dass Bezüge zwischen den Aktionen aufgelöst werden. Dadurch ließen sich Teilsätze, die nur syntaktische Anweisungen darstellen, erkennen und bereinigen. Weiterhin sollte PARSE sicherstellen, dass der Bezugsrahmen von Kontrollstrukturen vollständig erkannt wird. Zum Schluss kann die aktuelle Analyse des Syntaxbaumes erweitert werden. Hierbei ist zu beachten, dass nicht nur eine statische Analyse durchgeführt wird wie bei Übersetzern für Programmiersprachen, sondern die Analyse mit Sprachverständnis kombiniert wird. Als erstes wäre es hierbei wichtig zu erkennen, ob der erzeugte abstrakte Syntaxbaum syntaktisch korrekt ist. Dies wäre die Aufgabe traditioneller Analyseverfahren. Neue Verfahren müssten hingegen entworfen werden um sicherzustellen, dass der Baum vollständig und semantisch korrekt übersetzt wird. Hierbei würde idealerweise überprüft, ob die erstellten Aktionen sinnvoll und durchführbar sind. Die Analyse sollte daher nicht nur den Syntaxbaum sondern den gesamten Graphen betreffen. So sollte zum Beispiel der Versuch „einen Tisch in ein Wasserglas zu füllen“ möglichst früh als Fehler erkannt werden. Dabei wäre es auch möglich, dass mit der Analyse eine automatische Korrektur vorgenommen wird. Hierbei sollten Information eigenständig inferiert und hinzugefügt werden. Verfahren die für eine solche Analyse entworfen wurden könnten auch außerhalb von PARSE Anwendung finden. Offensichtlich wäre die Erweiterung von PARSE, um weitere Programmiersprachen. Hierbei läge der Fokus auf Programmiersprachen mit anderen Programmierparadigmen. Beispiele hierfür sind funktionelle, logische und grafische Programmiersprachen. Besonders grafische Programmierung könnte bereits jetzt vom Einsatz mit PARSE profitieren. Grafische Programmierung wird oft dazu verwendet, um Anfängern das Programmieren zu vereinfachen. Anders als bei textueller Programmierung sind die Aktionen und Verbindungen definiert. Mit Hilfe von PARSE könnten daher schnell Aktionen und entsprechende Verbindungen erstellt werden. Das so erstellte Programm müsste zwar korrigiert werden, dies könnte einem Anfänger aber leichter fallen als das Schreiben von Grund auf. Abseits von PARSE könnte vertragsbasierte Programmierung bei der Erzeugung von Quelltext helfen. Verträge verlangen, dass Aktionen Vorbedingungen, Nachbedingungen und Invarianten besitzen. Die Bedingungen erlauben die Erzeugung von wohldefinierte Aktivitäten, die sicherstellen, dass die verwendeten Aktivitäten zusammenpassen. Dies könnte die Erzeugung und Korrektur von Quelltext vereinfachen. Die Computerlinguistik besteht aber nicht nur aus der Synthese von Quelltext. Es wäre daher Interessant zu untersuchen in wie weit die sich die Synthese invertieren ließe. So sollte statt der Übersetzung von natürlicher Sprache in Quelltext, aus Quelltext natürliche Sprache erzeugt werden. Dies könnte dabei helfen Quelltext automatisch zu dokumentieren. Die Konzepte für die Synthese von Aktivitätsdiagrammen könnten dazu verwendet werden, um aus Verfahrensbeschreibungen die verwendeten Verfahren zu extrahieren. Eine Idee wäre es die extrahierten Verfahren automatisch mit anderen Verfahrungsbeschreibungen zu vergleichen, um so Übereinstimmungen bei den verwendeten Verfahren zu finden. Anhand dieser Übereinstimmungen könnten automatisch konzeptionelle Verbindungen zwischen den Beschreibungen hergestellt werden. Dies könnte dabei helfen Plagiate zu identifizieren. Interessanter wäre aber wohl die automatische Identifizierung von Zusammenhängen

zwischen unterschiedlichen Fachdisziplinen. So könnte bekanntes Wissen leichter auf andere Bereiche übertragen werden. Auch wäre es durch die Extraktion von Verfahren möglich, die beschriebenen Verfahren mit anderen ähnlichen Verfahren zu vergleichen. Dies könnte dazu genutzt werden, um Begutachtungen der Verfahrensbeschreibung durch maschinell erstellte Fragen zu unterstützen. So könnten die Autoren beispielsweise auf fehlende Details und Aktionen in beispielsweise Beschreibungen von Algorithmen hingewiesen werden. Als letztes könnten die extrahierten Verfahren auch einfach dazu dienen, Menschen einen Überblick über das beschriebene Verfahren zu geben.

Literaturverzeichnis

[ARA+] Asfour, T. ; Regenstein, K. ; Azad, P. ; Schroder, J. ; Bierbaum, A. ; Vahrenkamp, N. ; Dillmann, R.: ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In: *2006 6th IEEE-RAS International Conference on Humanoid Robots*, S. 169–175 (zitiert auf Seite 23).

[Bit] Bitterwolf, Christian: *Bestimmung von Aktionsidentität in Gesprochener Sprache*. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/bitterwolf_ba (zitiert auf Seite 20).

[BV] Brand, Mark van d. ; Visser, Eelco: Generation of Formatters for Context-Free Languages. 5, Nr. 1, 1-41. http://dx.doi.org/10.1145/226155.226156. – DOI 10.1145/226155.226156. – ISSN 1049–331X (zitiert auf Seite 31).

[FDB] Feldman, Jerome ; Dodge, Ellen ; Bryant, John: Embodied Construction Grammar. In: *The Oxford Handbook of Linguistic Analysis* (zitiert auf Seite 28).

[FGS] Fernández, Santiago ; Graves, Alex ; Schmidhuber, Jürgen: An Application of Recurrent Neural Networks to Discriminative Keyword Spotting. In: *Proceedings of the 17th International Conference on Artificial Neural Networks*, Springer-Verlag (ICANN'07). – ISBN 3–540–74693–5 978–3–540–74693–5, 220-229 (zitiert auf Seite 11).

[Fil] Fillmore, Charles J.: Some Problems for Case Grammar. https://kb.osu.edu/bitstream/handle/1811/81803/WPL_10_August_1971_245.pdf (zitiert auf Seite 26).

[Gü] Günes, Zeynep: *Aufbau Eines Sprachkorpus Zur Programmierung Autonomer Roboter Mittels Natürlicher Sprache*. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/guenes_ba (zitiert auf Seite 23).

[Gru] Gruber, Thomas R.: Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In: *In Formal Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers, in Press. Substantial Revision of Paper Presented at the International Workshop on Formal Ontology*, Kluwer Academic Publishers (zitiert auf Seite 12).

[Has] Hasu, Tero: Programmatic Building of Models Just for Pretty Printing. In: *Proceedings of OOPSLA Workshop on Domain-Specific Modeling (DSM)* (zitiert auf Seite 31).

[HDY+] Hinton, G. ; Deng, L. ; Yu, D. ; Dahl, G. E. ; Mohamed, A. r ; Jaitly, N. ; Senior, A. ; Vanhoucke, V. ; Nguyen, P. ; Sainath, T. N. ; Kingsbury, B.: Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. 29, Nr. 6, S. 82–97. http://dx.doi.org/10.1109/MSP.2012.2205597. – DOI 10.1109/MSP.2012.2205597. – ISSN 1053–5888 (zitiert auf Seite 11).

[Hey] Hey, Tobias: *Kontext- Und Korreferenzanalyse Für Gesprochene Sprache.* `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/hey_ma` (zitiert auf Seite 20).

[Hoc] Hochweiss, Robert: *Analyse Und Korrektur von Disfluenzen in Gesprochener Sprache.* `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/hochweiss_ba` (zitiert auf Seite 21).

[Jon] Jonge, M. de: Pretty-Printing for Software Reengineering. In: *International Conference on Software Maintenance, 2002. Proceedings.*, S. 550–559 (zitiert auf Seite 31).

[Jur] Jurafsky, Dan: *[(Speech and Language Processing: International Version : an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition)].* Pearson Education (zitiert auf Seite 8).

[Kei] Keim, Jan: *Themenextraktion Zur Domänenauswahl Für Programmierung in Natürlicher Sprache.* `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/keim_ma` (zitiert auf den Seiten 21 und 22).

[Koc] Kocybik, Markus: *Projektion von Gesprochener Sprache Auf Eine Handlungsrepräsentation.* `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/kocybik_ba` (zitiert auf Seite 19).

[KTF] Khayrallah, Huda ; Trott, Sean ; Feldman, Jerome: Natural Language for Human Robot Interaction. In: *Proceedings of the Workshop on Human-Robot Teaming at the ACM/IEEE Conference on Human-Robot Interaction (HRI)* (zitiert auf Seite 28).

[KZGS] Kwiatkowski, Tom ; Zettlemoyer, Luke ; Goldwater, Sharon ; Steedman, Mark: Inducing Probabilistic CCG Grammars from Logical Form with Higher-Order Unification. In: *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics (EMNLP '10), 1223-1233 (zitiert auf Seite 29).

[LB] Loper, Edward ; Bird, Steven: NLTK: The Natural Language Toolkit. In: *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics* (zitiert auf Seite 9).

[LBK+] Lauria, S. ; Bugmann, G. ; Kyriacou, T. ; Bos, J. ; Klein, E.: Converting Natural Language Route Instructions into Robot Executable Procedures. In: *11th IEEE International Workshop on Robot and Human Interactive Communication Proceedings*, S. 223–228 (zitiert auf Seite 29).

[Les] Lesk, Michael: Automatic Sense Disambiguation Using Machine Readable Dictionaries: How to Tell a Pine Cone from an Ice Cream Cone. In: *Proceedings of the 5th Annual International Conference on Systems Documentation*, ACM (SIGDOC '86). – ISBN 0–89791–224–1, 24-26 (zitiert auf Seite 10).

[Lev] Levenshtein, V. I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. 10, S. 707 (zitiert auf den Seiten 52 und 65).

[LGS] Le, Vu ; Gulwani, Sumit ; Su, Zhendong: SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ACM (MobiSys '13). – ISBN 978–1–4503–1672–9, 193–206 (zitiert auf Seite 26).

[LK] Landis, J. R. ; Koch, Gary G.: The Measurement of Observer Agreement for Categorical Data. 33, Nr. 1, S. 159–174. – ISSN 0006341X, 15410420 (zitiert auf Seite 83).

[LV] Lincoln, Nicholas K. ; Veres, Sandor M.: Natural Language Programming of Complex Robotic BDI Agents. 71, Nr. 2, 211-230. http://dx.doi.org/10.1007/s10846-012-9779-1. – DOI 10.1007/s10846–012–9779–1. – ISSN 1573–0409 (zitiert auf Seite 28).

[LWP+] Lin, Xi V. ; Wang, Chenglong ; Pang, Deric ; Vu, Kevin ; Zettlemoyer, Luke ; Ernst, Michael D.: *Program Synthesis from Natural Language Using Recurrent Neural Networks* (zitiert auf Seite 28).

[LWT] Landhäusser, Mathias ; Weigelt, Sebastian ; Tichy, Walter F.: NLCI: A Natural Language Command Interpreter. 24, Nr. 4, 839-861. http://dx.doi.org/10.1007/s10515-016-0202-1. – DOI 10.1007/s10515–016–0202–1. – ISSN 1573–7535 (zitiert auf den Seiten 27 und 35).

[MHZF] Matuszek, Cynthia ; Herbst, Evan ; Zettlemoyer, Luke ; Fox, Dieter: Learning to Parse Natural Language Commands to a Robot Control System. https://doi.org/10.1007/978-3-319-00065-7_28. In: Desai, Jaydev P. (Hrsg.) ; Dudek, Gregory (Hrsg.) ; Khatib, Oussama (Hrsg.) ; Kumar, Vijay (Hrsg.): *Experimental Robotics: The 13th International Symposium on Experimental Robotics*. Springer International Publishing (Springer Tracts in Advanced Robotics). – ISBN 978–3–319–00065–7, 403-415 (zitiert auf Seite 29).

[Mih] Mihalcea, Rada: Using Wikipedia for Automatic Word Sense Disambiguation. In: *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, S. 196–203 (zitiert auf Seite 10).

[MML+] Mou, Lili ; Men, Rui ; Li, Ge ; Zhang, Lu ; Jin, Zhi: On End-to-End Program Generation from User Intention by Deep Neural Networks. http://arxiv.org/abs/1510.07211 (zitiert auf Seite 27).

[MMS] Marcus, Mitchell P. ; Marcinkiewicz, Mary A. ; Santorini, Beatrice: Building a Large Annotated Corpus of English: The Penn Treebank. 19, Nr. 2, 313–330. http://dl.acm.org/citation.cfm?id=972470.972475. – ISSN 0891–2017 (zitiert auf Seite 9).

[OMK] Otter, Daniel W. ; Medina, Julian R. ; Kalita, Jugal K.: A Survey of the Usages of Deep Learning in Natural Language Processing. http://arxiv.org/abs/1807.10854 (zitiert auf Seite 27).

[Ou] Ou, Yue: *Erkennung von Aktionen in Gesprochener Sprache*. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/ou_ba (zitiert auf Seite 19).

[Pas] Paskaran, Dinesh: *Evaluation Unterschiedlicher Spracherkennungssysteme in Der Domäne Humanoide Robotik*. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/paskaran_ba (zitiert auf Seite 19).

[PPJM] Pulido-Prieto, Oscar ; Juárez-Martínez, Ulises: A Survey of Naturalistic Programming Technologies. 50, Nr. 5, 70:1–70:35. http://dx.doi.org/10.1145/3109481. – DOI 10.1145/3109481. – ISSN 0360–0300 (zitiert auf den Seiten 1 und 25).

[PRWZ] Papineni, Kishore ; Roukos, Salim ; Ward, Todd ; Zhu, Wei-Jing: BLEU: A Method for Automatic Evaluation of Machine Translation. In: *Proceedings of the*

40th Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics (ACL ’02), 311–318 (zitiert auf den Seiten 15 und 30).

[PRZH] Price, David ; Riloff, 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*, ACM (IUI ’00). – ISBN 978–1–58113–134–5, 207–211 (zitiert auf Seite 26).

[RSK] Rabinovich, Maxim ; Stern, Mitchell ; Klein, Dan: Abstract Syntax Networks for Code Generation and Semantic Parsing. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, 1139-1149 (zitiert auf Seite 30).

[Scha] Schlereth, Mario: *Entwicklung Eines Dialogagenten Für Dialogbasierte Programmierung*. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/schlereth_ma` (zitiert auf Seite 22).

[Schb] Schneider, Michael: *Entwurf Einer Handlungsrepräsentation Für Gesprochene Sprache*. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/schneider_ba` (zitiert auf Seite 19).

[Ste] Steurer, Vanessa: *Synthese von Methodendefinitionen Aus Natürlichsprachlichen Äußerungen*. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/steurer_ma` (zitiert auf den Seiten 20 und 35).

[ULSA] Ullman, Jeffrey D. ; Lam, Monica S. ; Sethi, Ravi ; Aho, Alfred V.: *Compiler: Prinzipien, Techniken und Werkzeuge*. 2. Pearson Studium. – ISBN 978–3–8273–7097–6 (zitiert auf Seite 5).

[WFHP] Witten, Ian ; Frank, E ; Hall, M.A. ; Pal, C.J.: *Data Mining: Practical Machine Learning Tools and Techniques* (zitiert auf Seite 11).

[WHS] Weigelt, S. ; Hey, T. ; Steurer, V.: Detection of Conditionals in Spoken Utterances. In: *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, S. 85–92 (zitiert auf Seite 21).

[WSC+] Wu, Yonghui ; Schuster, Mike ; Chen, Zhifeng ; Le, Quoc V. ; Norouzi, Mohammad ; Macherey, Wolfgang ; Krikun, Maxim ; Cao, Yuan ; Gao, Qin ; Macherey, Klaus ; Klingner, Jeff ; Shah, Apurva ; Johnson, Melvin ; Liu, Xiaobing ; Kaiser, Lukasz ; Gouws, Stephan ; Kato, Yoshikiyo ; Kudo, Taku ; Kazawa, Hideto ; Stevens, Keith ; Kurian, George ; Patil, Nishant ; Wang, Wei ; Young, Cliff ; Smith, Jason ; Riesa, Jason ; Rudnick, Alex ; Vinyals, Oriol ; Corrado, Greg ; Hughes, Macduff ; Dean, Jeffrey: Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. abs/1609.08144. `http://arxiv.org/abs/1609.08144` (zitiert auf Seite 11).

[WT] Weigelt, S. ; Tichy, W.F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference On* Bd. 2, S. 819–820 (zitiert auf Seite 17).

[YN] Yin, Pengcheng ; Neubig, Graham: A Syntactic Neural Model for General-Purpose Code Generation. `http://arxiv.org/abs/1704.01696` (zitiert auf Seite 29).

Anhang

A Kurzbeschreibung bei *Prolific*

The study evaluates the similarity between commands given in English and computer generated activity diagrams.

The study begins with an Introduction to activity diagrams. After the introduction, you have to compare English command sequences with activity diagrams. If you identify differences between the commands and the activity diagram you check the appropriate boxes.

There will be 10 activity diagrams. Each diagram may differ in size and complexity.

You will need your ProlificID! Please copy the ID in advance. Please participate only once for the same study. However, you are welcome to participate in the other studies of the series.

B Einführung der Online-Studie

Activity diagrams represent a work flow through activities. They're also known as flow charts.

The activity diagrams and sentences in this questionnaire describe instructions to a kitchen robot called Armar.

The activity diagrams where constructed automatically.

Your task is to identify errors in the diagrams.

B.1 Sentence

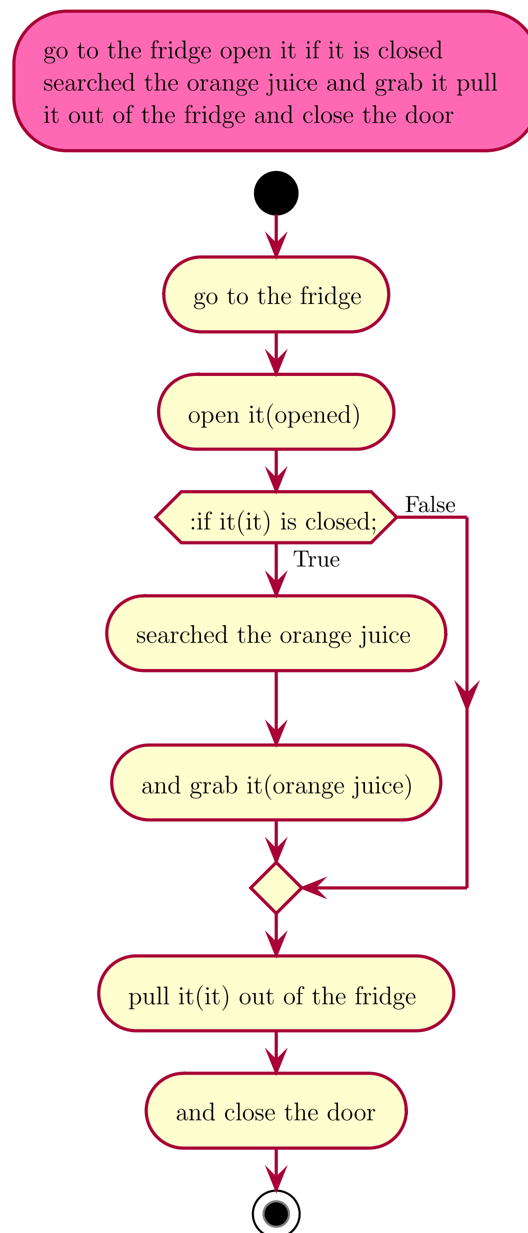


Abbildung B.1: Example of an activity diagram

The full sentence is written atop in pink. This is the sequence of commands which the activity diagram is compared to. Both should ideally represent the same sequence of actions.

The activity diagram starts with a full black dot. This is the start of the sequence of activities. The end of the activity diagram is symbolized by a black and white dot at the bottom.

B.2 Activities

An activity consists of a single action, command or condition. It is represented with a light yellow node. If a node contains activities that don't make sense, lack information or contain information which should be part of another node, then check the Äctivity incorrect/incomprehensible"box. It is also possible that an activity is missing or that another

activity should be split up. In that case check the Äctivity missing"box. If one of the nodes should be removed, check the Äctivity should be removed"box.

Activities are connected by arrows. Arrows point to the next activity. If the sequence of activities is incorrect (i.e. an activity should be placed before another) check the Äctivity sequence incorrect"box.

Some Activities contain words in round brackets after pronouns like it/her/she. These refer to an entity. If they refer to the wrong entity check the "Reference to wrong entity"box.

B.3 Conditions, Branches and Loops

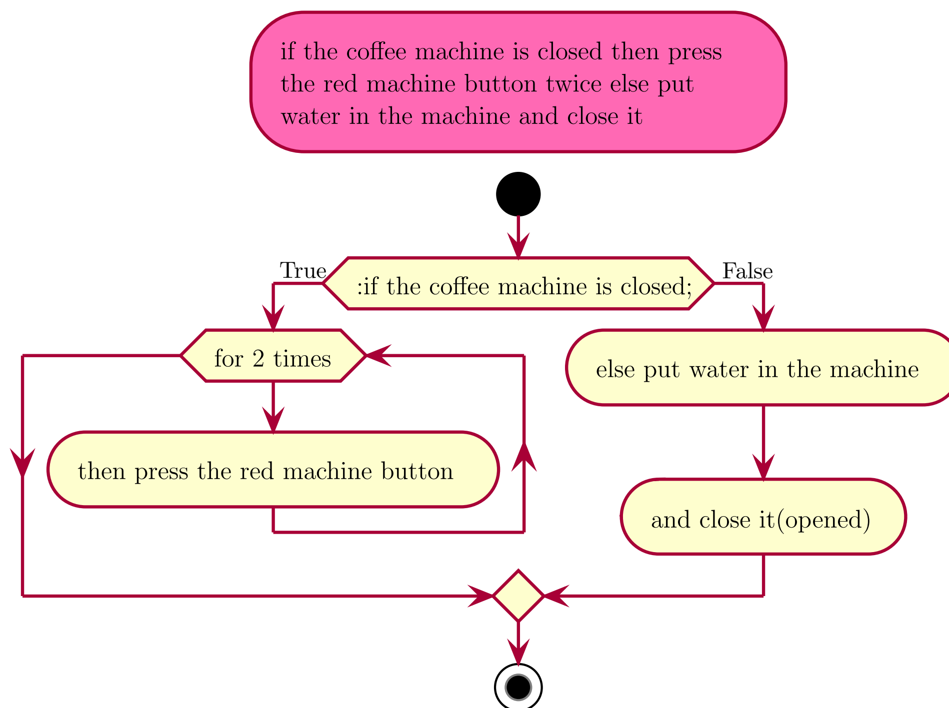


Abbildung B.2: Activity Diagram with Branch and For-Loop

In some cases a course of activities represents a conditional sequence. In such cases there may be multiple next activities and thus the path is split. Depending on the evaluation of the condition one of the next paths is chosen.

Conditions are usually part of a loop or branch. Branches split the sequence in two separate paths. Only one path can be followed. Branches may contain multiple types of errors, such as missing or additional activities. It is also possible that a path doesn't contain activities at all. In these cases check the "Branches incorrect"box.

In a loop one of the paths will be repeatedly followed until the condition is met to exit it. A special type of loop is a for-loop. Here the looping path will be followed for a number of times and then exited. If there are errors in any loop, check the "Loop incorrect"box.

B.4 Concurrency

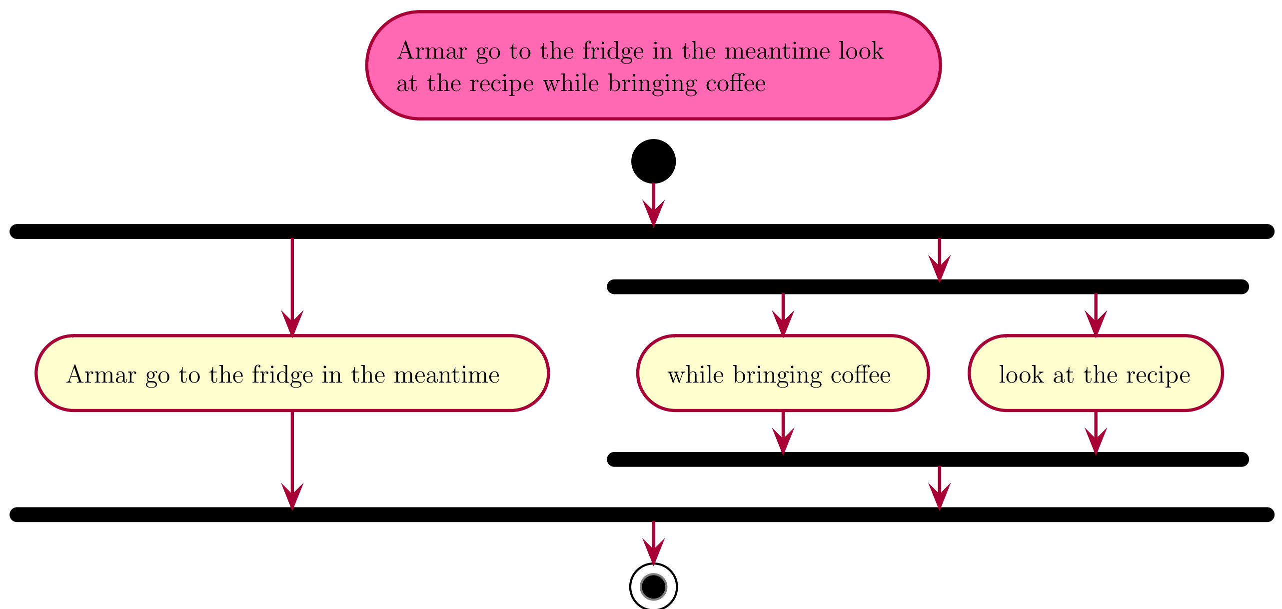


Abbildung B.3: Activity Diagram with Concurrency

Concurrency bars splits the path in a special way. Instead of following a single path, all paths are followed at the same time. Thus the activities in concurrency are done at the same time. The concurrency ends when all paths in the concurrency reach the ending bars. From now on, only a single path is followed.

B.5 No errors and other errors

Not all activity diagrams contain errors. It's possible that the diagram is entirely correct. In this case check ëverything correctönly.

In some cases you may identify errors that do not fit in one of the above categories. In cases, you may pick the öthers"check box and describe the error textually.