Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Neuroinformatik und Kognitive Robotik

Analyse der Einsetzbarkeit von Sprachassistenten auf mobilen Robotern

Bachelorarbeit zur Erlangung des akademischen Grades Bachelor of Science

Markus Paschke

Betreuer: Dr. Steffen Müller, FG Neuroinformatik und Kognitive Robotik

Verantwortlicher Hochschullehrer:
Prof. Dr. H.-M. Groß, FG Neuroinformatik und Kognitive Robotik

Die Bachelorarbeit wurde am 11.09.2017 bei der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau eingereicht.

Erklärung: „Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Alle von mir aus anderen Veröffentlichungen übernommenen Passagen sind als solche gekennzeichnet.“

Ilmenau, 11.09.2017 .....

Markus Paschke

Abstract

Personal Digital Assistants are playing a tremendous role in our daily life. There are several proprietary and open source projects on the market, e.g. Google Assistant, Apple Siri, Microsoft Cortana, Bixby, IBM Watson, Mycroft.

The system architecture, data flows, analysis of hardware and software requirements, as well as the comparison of different systems are described in this thesis. In addition, an abstract scheme, which pictures the process of recognizing, analyzing natural language and deliver well-defined answers is shown.

After the extensive analysis of the depicted systems, a created personal assistant, that is based on different available sub-components and extended with own features, is demonstrated.

Zusammenfassung

Sprachassistenten spielen zunehmend eine wichtige Rolle im täglichen Umfeld. Derzeit gibt es verschiedene proprietäre und Open Source Projekte am Markt, z.B. Google Assistant, Apple Siri, Microsoft Cortana, Bixby, IBM Watson, Mycroft.

In dieser Arbeit werden die System Architektur, die Datenflüsse, die Analyse der Hardware und Softwarevoraussetzungen sowie der Vergleich der einzelnen Systeme dargestellt. Zudem wird ein abstraktes Verfahren vorgestellt, welches anschaulich den Prozess darstellt, natürliche Sprache zu erkennen, zu analysieren und gegebenenfalls wohldefinierte Antworten auf Fragen oder Problemstellungen zu liefern.

Nach einer ausführlichen Analyse der Systeme wird ein erstellter Sprachassistent aufgezeigt, der auf bereits vorhandene Teilkomponenten aufbaut und bezüglich eigener Services und Leistungen erweitert wurde.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit wird zunächst ein Überblick über verschiedene Techniken zur Erkennung von natürlicher Sprache aufgezeigt. Welche Komponenten benötigt ein Sprachassistent, um erfolgreich Sprache zu analysieren, zu verarbeiten und eine Antwort durch *Text-to-Speech* (TTS) zu liefern? Zudem werden, soweit es möglich ist, mit Hilfe von Patenten und öffentlichen Dokumenten proprietäre Systeme (u.a. Siri, Google Assistent) untersucht und deren System Architektur dargestellt.
Nachdem die vorgestellten Sprachassistenten ausführlich analysiert und verglichen wurden, wird der Entwurf und die Implementierung eines Sprachassistenten aufgezeigt, der alle Anforderungen an den Betrieb in einem mobilen Roboter im Altenheim erfüllt.

Der Grundstein für die Spracherkennung wie wir sie heute kennen wurde in San Jose, Californien gelegt, indem ein Gerät namens Shoebox von IBM entwickelt wurde, welches bis zu 16 Wörter erkannte, die in ein Mikrophon gesprochen wurden. Das Gerät verstand die Zahlen 0-9, primitive Operationen (plus, minus und total) und wies einen Computer an, die Operationen auszuführen und danach auf einem Bildschirm auszugeben.

Erkennung und Verständnis natürlicher Sprache hat sich in den folgenden 50 Jahren gravierend verbessert und so sind derzeitige Assistenten in der Lage komplexe Anweisungen zu bewältigen. Apple Siri und Google Assistant, die derzeit den größten Markt-

anteil im Smartphone-Bereich besitzen [GARTNER], wurden Ende 2011 und Mitte 2012 vorgestellt. Weiterhin sind noch Microsoft Cortana, welches auf Windows Smartphones und Computern verfügbar ist, Viv, welches 2017 von Samsung aufgekauft wurde und teilweise in Bixby einging, IBM Watson und der Open-Source Ableger Mycroft derzeit am weitesten vertreten und werden deshalb analysiert.

Kapitel 2

Dialog System

Jeder dieser Sprachassistenten besitzt eine unterschiedliche System Architektur, welche jedoch schematisch auf ein Dialog System zurückgeführt werden kann. Dieses Beispiel Dialog System beinhaltet fünf Module, welche im Folgenden genauer betrachten werden. Die Dialog Management Komponente stellt dabei das Kernstück dar.

Eine Interaktion mit einem Dialog System könnte folgendermaßen aussehen [Jokinen und McTear, 2010, S. 3]:

1. Der Nutzer möchte wissen, wie das Wetter morgen wird und produziert ein akustisches Signal, welches von dem *Automatic Speech Recognition* Modul (ASR) aufgenommen und verarbeitet wird.

2. Das *Spoken Language Understanding Modul* (SLU) bekommt eine Wortfolge des ASR und versucht den syntaktischen Aufbau des Satzes, sowie die Bedeutung zu analysieren.

3. Die SLU Einheit hat die Semantik des Satzes, Wetter in [ORT] am [DATUM] herausgefiltert und übergibt diese Informationen an das Dialog Management Mo-

dul. Dieses führt nun eine Suche aus und greift auf den Kontext des Nutzers zu. Dies könnte z.B. der eingetragene Heimatort sein.

4. Die *Natural Language Generation* (NLG) strukturiert die Informationen, die dem Nutzer ausgegeben werden sollen, bezüglich einer linguistischen Struktur. Mithilfe von vordefinierten Grammatiken wird natürlicher Text erstellt. Dies könnte in einem Beispiel so aussehen: Es sind [DATUM] [TEMPERATUR] in [ORT].

5. Der finale Schritt ist nun diese Wortfolgen wieder als natürliche Sprache auszugeben. Die Realisierung erfolgt im TTS Modul.

In den folgenden Abschnitten werden die einzelnen Komponenten eines Dialog Systems genauer betrachtet und die Relevanz für die Sprachassistenten analysiert. Dabei werden u.a. State of the Art Techniken aufgegriffen, welche nicht in jedem Sprachassistenten manifestiert sein müssen.

2.1 Automatic Speech Recognition

Die automatische Spracherkennung ist der erste Schritt des Dialog Systems, um eine Anfrage des Nutzers an den Sprachassistenten zu bearbeiten. Für eine genaue Spracherkennung müssen mehrere Techniken und Algorithmen angewandt werden. Da in dieser Arbeit die einzelnen Sprachassistenten im Vordergrund stehen, wird nur schematisch auf einige der folgenden Komponenten, die sich auf [KRISNAWATI] beziehen, eingegangen.

- Signalverarbeitung
  - Signalabtastung
  - Quantisierung
  - Phonerkennung
- Dekodierung
  - akustisches Modell

  - Sprachmodell
  - linguistische Datenbank
  - Wörterbuch

- Adaption

Ziel der Spracherkennung ist es, die symbolische Darstellung als Wortfolge für ein vorliegendes akustisches Signal zu liefern. Dies geschieht durch ein trainiertes neuronales Netzwerk, welches einzelne Phoneme einem Fensterausschnitt des Ausgangssignals zuordnet. Das Resultat ist daraufhin ein Merkmalsvektor für diesen spezifischen Fensterausschnitt, welcher mit weiteren Merkmalsvektoren zu einem diskreten Symbol geclustet wird.

Der ASR Decoder beinhaltet eine linguistische Datenbank, ein Wörterbuch, ein trainiertes Sprachmodell und weitere akustische Modelle. Dabei wird eine Wortfolge gesucht, die der Merkmalsvektorenfolge möglichst ähnlich ist. Das Resultat ist eine Wortfolge oder Liste mit den n-wahrscheinlichsten Wörtern [Stolcke et al., 1997]. Ein Wortgraph ist ebenfalls möglich und wird häufig verwendet. Damit ist der Wortgraph oder die Liste der n-wahrscheinlichsten Wörter von besonderer Relevanz, denn dadurch können im Verlauf der Analyse einzelne Wörter stärker gewichtet werden. So ist zum Beispiel die korrekte Verwendung eines *Homophons* (phonetisch ähnlich klingendes Wort) nur mit Hilfe des Kontextes des Satzes möglich (z.B. 'uːrtsaɪt: Uhrzeit, Urzeit). Dieser Kontext kann sowohl lokal als auch global sein, deshalb können einige phonologischen Variationen schon im ASR/SLU Modul behandelt werden.

Weitere Probleme, bei dem das ASR Modul an seine Grenzen stößt, sind: Geräuschstörungen, Variationen der Sprachbenutzung sowie Störungen des Redeflusses, neue Wortschätze [Krisnawati], Dialekte und Jargons.

2.2 Spoken Language Understanding

Die SLU Einheit führt einen komplexen Prozess zur Identifizierung der logischen Struktur und Bedeutung eines Satzes bzw. Wortfolge durch. Im Gegensatz zur ASR ist dafür ein tieferes Verständnis für natürliche Sprache notwendig, um über die Bedeutung und Intention des Nutzers Aussagen treffen zu können. Herkömmlicherweise wird dies durch kontextfreie Grammatiken und Unifikation einzelner Merkmale vollbracht. Diese Methoden sind sehr anwendungsspezifisch und robust, benötigen jedoch eine gewisse Kompetenz und müssen zeitaufwendig manuell erstellt werden.
Da die Wortfolge des ASR Moduls bezüglich gesprochener Sprache nicht vergleichbar mit geschriebener Sprache ist und Pausen, Wiederholungen oder Verbesserungen enthalten kann, ist eine robuste Implementierung dieser Grammatiken notwendig. Dabei muss jedoch immer eine Balance zwischen Robustheit und Flexibilität gefunden werden. Ursache dafür ist, dass Robustheit immer auf Kosten der Genauigkeit geht und somit solche Systeme meist über-generalisieren [Wang et al., 2005].

Wie schon erwähnt gibt es mehrere Implementierungsmöglichkeiten einer SLU Komponente, dabei unterscheidet man hauptsächlich zwischen einem wissensbasierten und einer statistisch lernenden Vorgehensweise. Die herkömmliche Variante eines wissensbasierten Ansatzes bringt folgende Probleme mit sich:

- Die Erstellung einer Grammatik ist eine fehlerbehaftete Entwicklung.
- Es sind mehrere Entwicklungsperioden notwendig, um diese daraufhin genau abzustimmen.
- Eine gute Fachkenntnis bezüglich Linguistik und ingenieurwissenschaftliche Expertise ist notwendig, um eine Grammatik zu konstruieren, die ein weites Spektrum abdeckt, eine geringe Fehlerrate besitzt und zudem performant ist.
- Grammatiken sind in der Aufrechterhaltung und Erweiterung teuer und sehr domänenspezifisch [Wang et al., 2005].

Die wissensbasiert aufgebauten Systeme benötigen im Gegensatz zu statistisch lernenden jedoch keine große Menge an Trainingsdaten und können leichter evaluiert und validiert werden. Dabei werden in statistisch lernenden Systemen u.a. folgende Techniken verwendet: Lineare Regression, Klassifikation, Baumstrukturen, Neuronale Netzwerke, Cluster. Viele Probleme des wissensbasierten Ansatzes werde hier aufgegriffen und versucht zu lösen. Idealerweise sollte sich eine SLU Komponente an neue Domänen anpassen können. Dies ist mit solchen Systemen ohne großen Mehraufwand möglich. Aufgrund des meist fehlenden großen Trainingssatzes zur Entwicklungszeit, werden häufig beide Systeme miteinander kombiniert. Weiterführende Informationen finden sich in [Tur und De Mori, 2011].

2.3 Dialog Management

Die Dialog Management Komponente unterscheidet sich besonders in verschiedenen Dialog Systemen. Eine mögliche Implementierung wird anhand des Olympus Frameworks, welches auf RavenClaw [Bohus und Rudnicky, 2003] aufbaut, das wiederum auf einen aufgabenorientierte Dialog Engine basierenden AGENDA Dialog Manager setzt, aufgezeigt.

RavenClaw ist ein zweigeteiltes System mit der *Dialog Task Specification*, die jegliche Domänen spezifische Dialog Logik beinhaltet, und die *Dialog Engine*, welche domänenunabhängig agiert und den Dialog mit dem Nutzer kontrolliert, indem es sowohl die Dialog Task Specification ausführt, als auch Konversationsstrategien verfolgt (z.B. Zeiteinteilung, Terminierung, abwechselnde input/output phasen) [Bohus und Rudnicky, 2003].

2.3.1 Dialog Task Specification

Die genutzte Datenstruktur für die Dialog Task Specification ist eine Baumstruktur, die folgende Vorteile mit sich bringt [Bohus und Rudnicky, 2003]:

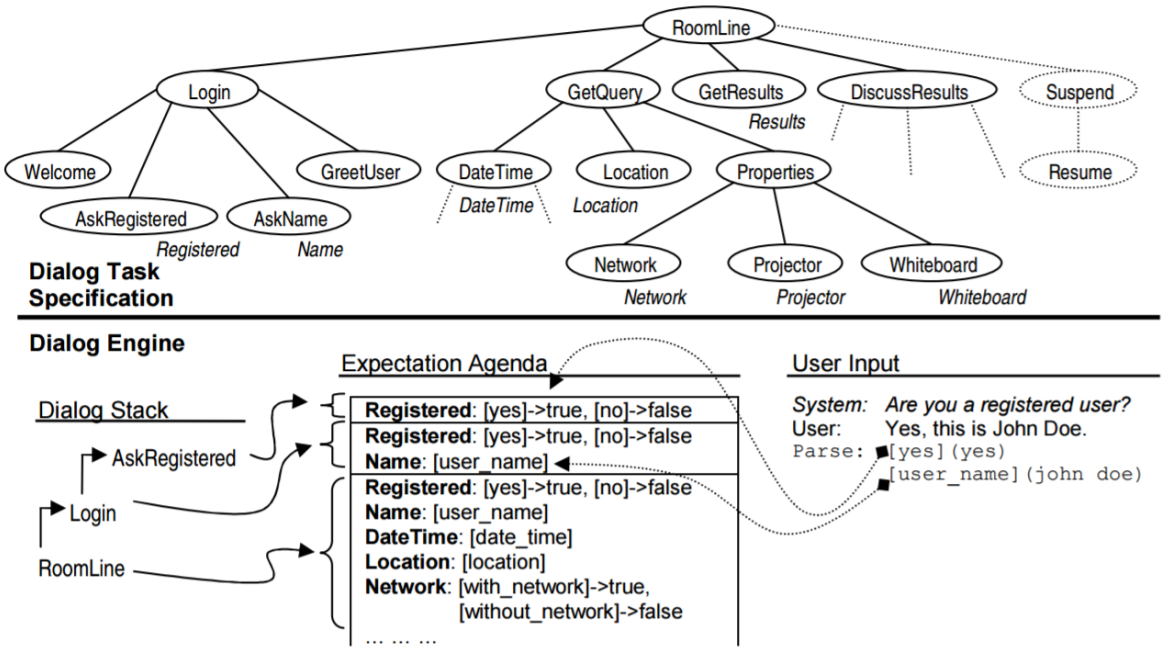


Abbildung 2.1: RavenClaw architectural details

- Eine natürliche hierarchische Interpretation für die meisten zielgerichteten Systeme.
- Jedes Subsystem, als Blatt repräsentiert, ist meist unabhängig und kann entsprechend einfach implementiert und gewartet werden.
- Eine solche Struktur ist skalierbar und kann während Laufzeiten entsprechend angepasst werden.
- Der Kontext und die Abhängigkeiten eines Blattes und inneren Knotens zu seinen Eltern sind jederzeit einsehbar
- Die Baumstruktur kann chronologisch durchlaufen werden (z.B. preorder)

Die inneren Knoten werden *dialog agencies* genannt, verkörpern ein höheres Level an Logik und überwachen den Ablauf der Kind-Knoten und deren Ausführreihenfolge. Die Blätter repräsentieren atomare Operationen, werden *fundamental agent* genannt und können einen von vier Typen besitzen.

1. **Informieren** *Inform* - Hiermit ist das senden einer Botschaft an den Nutzer (z.B. Welcome in Abbildung 3.1) gemeint.

2. **Anfrage** *Request* - Es werden Informationen des Nutzers angefordert (z.B. AskRegistered in Abbildung 3.1).

3. **Erwarten** *Expect* - Informationen des Nutzers werden ohne diese vorauszusetzen erwartet.

4. **Domänen Operation** *DomainOperation* - Diese Agents stellen die Schnittstelle zu anderen (Online-) Diensten dar und können auf Datenbanken, *Personal Knowledge Memory* o.ä. zugreifen und führen eine domänenspezifische Operation aus (z.B. GetResults in Abbildung 3.1)

Jeder Typ und dementsprechend jedes Subsystem besitzt weiterhin Voraussetzungen, Trigger, Abbruchkriterien, Erfüllungskriterien und die Ausführungs-Routine [Bohus und Rudnicky, 2003].

2.3.2 Dialog Engine

Die Dialog Engine startet die Dialog Task Specification und kontrolliert den Gesprächsfluss, indem sie Ausführungsphasen, in der möglicherweise mehrere *agencies* ausgeführt werden, und Informationseingabephasen verschachtelt.

Ausführungsphase

Initial wird ein Stack erstellt, der als Plan verwendet wird, um die einzelnen *agencies* auszuführen, wobei anfänglich nur der Wurzel-Agent darauf steht. In folgenden Schritten wird jeweils der *agent* ausgeführt der auf dem Stack oben steht. Dabei führt dieser wiederum eines seiner Kind-Knoten aus, indem er es zur Ausführung plant und auf den Stack schreibt, bis ein *fundamental agent* ausgeführt und eine System Antwort oder Operation generiert wird. Zusätzlich können fundamental agents mit dem Typ Anfrage jederzeit die Ausführungsphase unterbrechen und die Dialog Engine anweisen daraufhin in die Input Phase zu wechseln und die nötigen Informationen abzufragen.

Eingabephase

In der Eingabephase werden Informationen des Nutzers abgefragt. Diese startet entweder nachdem ein Agent, der unmittelbar davor ausgeführt wurde Eingaben erwartet und die Dialog Engine anwies in diese Phase zu wechseln oder nachdem alle Agenten des Stacks abgearbeitet wurden und weitere Befehle benötigen. Zur Analyse kann dieser Abschnitt in drei Teilphasen unterteilt werden:

1. **Agenda der erwarteten Informationen:** In diesem Teilabschnitt wird eine Datenstruktur für jegliche Eingaben, die ein fundamental agent (z.B. [user_name] für AskName in Abbildung 3.1) oder eine agency erwartet, angelegt. Dabei wird der Stack von oben durchlaufen und jedes Element teilt mit, welche Informationen benötigt werden. Agencies müssen dabei jeden einzelnen Kind-Knoten befragen.

2. **Variablenbindung:** Im zweiten Abschnitt werden die Eingaben auf die erwarteten Datenfelder abgestimmt, indem die Agenda ebenfalls von oben (top-down) durchlaufen wird. Felder die möglicherweise mehrmals durch verschiedene unabhängige agents abgefragt werden sollen, nutzen die Eingabe, die früher und damit hierarchisch höher eingeordnet wurde.

3. **Abfrage weiterer Agenten:**

   Nachdem alle Informationen erfolgreich gesammelt wurden, besitzen im Anschluss noch einmal alle Agenten des Dialog Task Specification Tree die Möglichkeit sich zu äußern und auf den Stack gepusht zu werden um etwaige letzte Informationen abzurufen. Dies funktioniert jedoch nur durch sogenannte Trigger, die vorher spezifiziert sind und während der Ausführungsphase nicht dynamisch geändert werden können. Wenn alle Agenten zufrieden gestellt sind und der Stack komplett durchlaufen wurde, wechselt die Dialog Engine wieder in die Ausführungsphase [Bohus und Rudnicky, 2003].

2.4 Natural Language Generation

Die Natural Language Generation versucht natürliche Sprache bzw. Schrift aus einer maschinellen Repräsentation von Daten zu erstellen. Es ist das logische Gegenstück zu dem bereits vorgestellten Natural Langue Understanding Modul.

Einfach strukturierte Sätze können mithilfe sogenannter *templates* erstellt werden, indem ein Muster eines grammatikalisch korrekten Satzes Platzhalter an den Stellen aufweist, an dem Informationen, die von dem System bereitgestellt werden, eingefügt werden. Mit anderen Worten, wird der nicht-linguistische Input direkt auf den Satz *gemapped* [Reiter und Dale, 1997].
Ein mögliches Beispiel (Zug ICE707 fährt aus Hamburg um 12:00 Uhr los) für eine template-basierte Generierung könnte folgendermaßen aussehen:

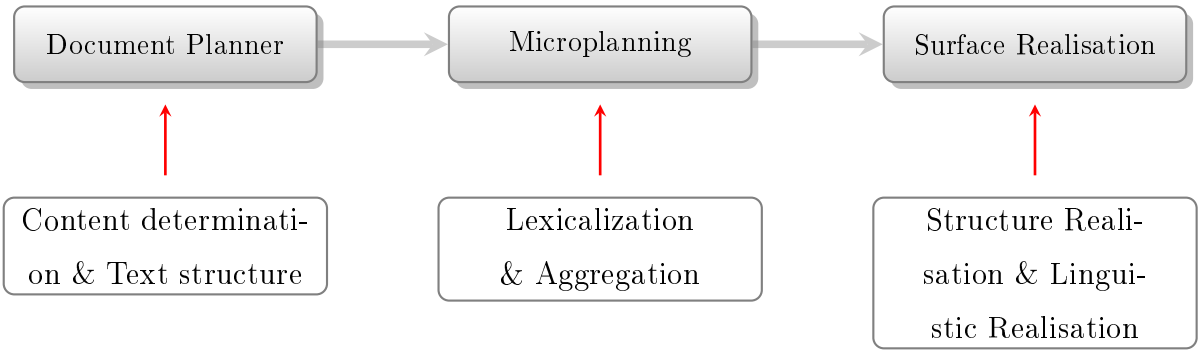
$\text{Abfahrt}(\text{zug}_{\text{ICE707}}, \text{ort}_{\text{Hamburg}}, \text{zeit}_{1200})$

Diese Informationen werden direkt mit dem nachfolgenden Template assoziiert.

Der [zug] verlässt [ort] um [zeit]

,wobei die Lücken [zug], [ort] und [zeit] mit Informationen aus der Datenbank gefüllt werden bzw. in diesem Exempel von dem Dialog Management bereitgestellt werden. Diese Art der Textgenerierung weißt jedoch u.a. folgende Probleme auf: (1) Die generierten Antworten sind auf die Domain und das jeweilige Template-Verhalten limitiert. (2) Auf Morphologie und Orthografie wird nicht geachtet, weil keine linguistischen Modelle verwendet werden. (3) Sätze können nicht optimiert werden. Falls eine Aufzählung stattfindet, wird für jeden Teil die gleiche Phrase verwendet. Ferner wird bei einer Beschäftigung mit diesen Modellen deutlich, dass es sich um einfache repetitive template-basierte Modelle handelt, welche keine komplexen Antworten generieren können.

Zu diesem Zweck führen Dale und Reiter für eine Natural Language Generation mehrere Phasen ein.



1. **Document Planner**: Bekommt als Input ein vier-tupel <k,c,u,d>, wobei k die *Knowledge Source*, c das *Communicative Goal*, u das *User Model* und d die *Discourse History* ist. Ein oder mehrere Input Variablen können als null angenommen werden.
   Aus diesen Daten

   - werden sogenannte *Messages* konstruiert.
   - wird entschieden, welche Informationen benötigt werden, um das Communicative Goal zu erfüllen.
   - wird eine Dokumentenstruktur entwickelt, die eine flüssige Texterstellung ermöglicht.

2. **Microplanning**: Bekommt als Input den *Document Plan* des Document Planner und produziert eine fertige Text-Spezifikation.
   Darunter fallen folgende Aufgaben:

   - Die Auswahl der Wörter und syntaktischen Strukturen für einzelne Messages.
   - Die Komposition der einzelnen Messages, um Textspezifikationen einzelner Sätze zu erstellen.

3. **Surface Realisation**

   Die Surface Realisation bekommt die Text Spezifikation in Form einer Baumstruktur von dem Micro Planner und generiert aus diesem den finalen Text. Dabei wird der Baum in preorder druchlaufen. Die inneren Knoten enthalten Strukturinformationen bezüglich des Dokuments (z.B. Überschrift, Kapitel, Abschnitt) und müssen im späteren Dokument nur entsprechend vermerkt werden. Die Blätter enthalten die einzelnen Phrasen und werden auf linguistische Formen zugeordnet.

Während das Microplanning und die Surface Realisation weitgehend auf linguistischen Wissen basieren, ist der Document Planner der applikationsabhängige Teil der NLG.

2.5 Text to Speech

Unter computerbasierter Sprachsynthese wird eine Komponente des Dialog Systems verstanden, welches als Eingabe einen Text bekommt und diesen in eine akustische Sprachausgabe umwandelt. Dabei muss es möglichst verständlich und natürlich klingen, wobei ersteres notwendig und letzteres eine nicht funktionale Anforderung ist. Ungeachtet dessen ist eine natürlich-klingende Sprache ein wichtiges Qualitätsmerkmal und differenziert die derzeitigen TTS Systeme am Markt. Weitere Qualitätsmerkmale sind u.a. eine gute Stimmlage, Sprachmelodie und korrekte Aussprache von Satzzeichen oder Abkürzungen.

Ein TTS-Synthesizer besteht generell aus zwei Komponenten, der *Natural Language Processing* (NLP) Einheit und der *Digital Signal Processing* (DSP) Einheit.

2.5.1 Die NLP-Komponente

In der NLP-Komponente wird der Text in seiner orthografischen Form in eine phonetische Repräsentation überführt und falls möglich auch prosodische Informationen hinzugefügt [Bouché]. Darunter wird die Wortintonation verstanden und sie ist für die Qualität eines gesprochenen Satzes von Bedeutung. So kann der Satz:

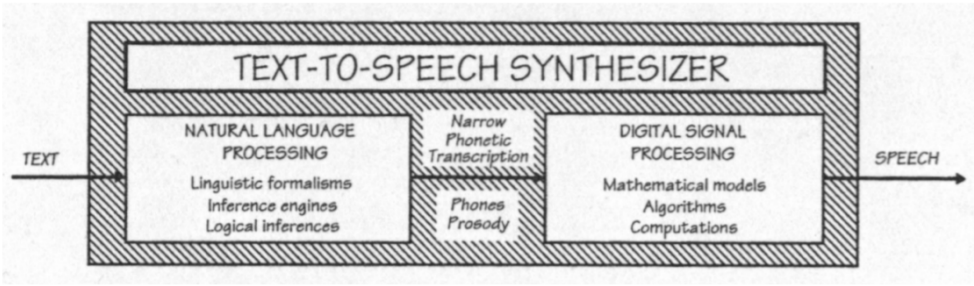


Abbildung 2.2: Aufbau eines TTS Systems. Aus [Dutoit, 1997, S.14]

Du gehst jetzt nach Hause

auf verschiedene weisen interpretiert werden. Mit Hilfe einer fallenden oder steigenden Tonhöhe kann aus dem Satz eine Aussage oder eine Frage formuliert werden. Zur Generierung einer phonetischen Repräsentation gibt es grundsätzlich zwei verschiedene Ansätze:

1. **Regelbasierter Ansatz:** Auf den Text wird eine komplexe Liste an Regeln zur *Graphem-Phonem* Bildung (Buchstaben-Laut-Zuordnung) angewendet, die zu einer Umwandlung in eine Phonemkette, in Form einer phonetischen Lautschrift führen. Ausnahmen für Fremdwörter oder Abkürzungen müssen zusätzlich definiert werden.

2. **Lexikonbasierter Ansatz:** Bei diesem Ansatz wird die Phonemkette mit Hilfe eines Lexikons erzeugt. Dabei kann ein Vollformen-Lexikon verwendet werden, welches alle möglichen Formen eines Wortes enthält, jedoch sehr speicherintensiv ist und eine große Redundanz aufweist.
   Falls nur Wortstämme bzw. Grundformen in einem Lexikon definiert sind, können weitere Wörter durch morphologische Prozesse gebildet werden, wodurch der Speicheraufwand stark reduziert wird. Für diese Art des Lexikons müssen Regeln definiert werden, um Wörter die nicht vorhanden sind bilden zu können.

Der Regelbasierte Ansatz ist für Sprachen mit regelmäßiger Wortintonation und Fluss

sehr gut geeignet (z.B. Französisch oder Spanisch), performt jedoch sehr schlecht für bsw. Englisch. Für Englisch muss hingegen ein lexikonbasierter Ansatz gewählt werden. Die Deutsche Sprache wird meist durch Einsatz beider Verfahren synthetisiert [Bouché].

2.5.2 Die DSP-Komponente

Der digitale Signal Prozessor wandelt die phonetische Repräsentation des Textes in gesprochene Sprache um und gibt ein akustisches Signal aus. Dabei können ebenfalls mehrere Ansätze verwendet werden [Karrer, 2010]:

- Computergenerierte Sounds nutzen ein akustisches Modell um aus den Phonemen einen Satz zu bilden. Diese einfachen Modelle sind performant und verständlich aber nicht sonderlich natürlich. Sie werden in der Praxis, wegen ihres niedrigen Speicherverbrauchs, meist in eingebetteten Systemen genutzt.

- Im Gegensatz dazu gibt es die *verknüpfende* Sprachsynthese, welche menschliche Sprache nutzt und sie in Phoneme, Silben, Wörter oder ganze Sätze aufbricht und in einer Datenbank speichert. Zum Satzbau werden alle nötigen Teile aus der Datenbank entnommen und entsprechend konkateniert. Typische TTS-Sprach Datenbanken wachsen in etwa auf 100-200Mb Größe an. Diese Art von Sprachsynthese ist für den Anwender vergleichbar mit natürlicher Sprache.

2.6 Beispiel Dialog System

Ein Open Source Framwork zur Implementierung eines Dialog Systems stellt *Olympus* [Bohus et al., 2007] dar. Es wurde in der Carnegie Mellon University (CMU) in den späten 2000ern entwickelt und steht unter aktiver Entwicklung.
Dieses Framework beinhaltet folgende Komponenten [University, 2008]:

- Das Dialog Management wird durch RavenClaw gesteuert, welches in dieser Arbeit bereits analysiert wurde.

- Das Low-level Interaktion Management (z.B. Timing von Start und Ende einer Aussage) wird durch den Apollo Interaktions Manager verarbeitet.
- Für die Spracherkennung unterstützt Olympus derzeit die CMU Sphinx Familie.
- Natural language understanding (NLU) wird von Phoenix realisiert. Dies ist ein robuster Parser der auf rechtslinearen Grammatiken basiert.
- Die Helios Komponente bekommt ein Set von Hypothesen von der NLU-Einheit und klassifiziert diese mit einer *confidence score* (spiegelt die Wahrscheinlichkeit wieder, dass die Eingabe korrekt semantisch verstanden wurde) und übergibt anschließend die Hypothese mit der höchsten Wahrscheinlichkeit an das Dialog Management.
- Zur Text Generierung (NLG) wird das Rosetta template basierte Verfahren genutzt.
- Das Modul Kalliope unterstützt derzeit u.a. SAPI 5-konforme TTS Engines und CMU's Flite Engines zur Sprachsynthese.
- Die Kommunikation zwischen den Modulen wird durch die MIT/MITRE Galaxy Communicator Architektur geschaffen.

Anhand der angeführten Analyse der Komponenten und dieser Liste eines Dialog System Frameworks wird deutlich, wie umfangreich ein funktionierendes System ist. Einige dieser Komponenten sind domain-unabhängig. Darunter fallen die Interaktion innerhalb des Systems, die Spracherkennung, die Sprachsynthese und die Natural Language Generation. Domain spezifisch muss je nach Einsatzziel das Dialog Management und dessen Interface zu Backend Services wie Datenbanken und Knowledge Graphen angepasst werden.

Nachdem in diesem Kapitel eine Übersicht über die einzelnen Komponenten eines Sprachassistenten dargestellt wurden, können mit diesem Hintergrundwissen die bereits vorgestellten Sprachassistenten bezüglich ihrer Prinzipien, System Architektur, Erweiterbarkeit und Implementierbarkeit erörtert werden.

Kapitel 3

Google Assistant

Google investierte in das Gebiet des *Digital Personal Assistant (DPA)* und brachte 2012 Google Now heraus. Diese App basiert auf der rudimentären Such-App, welche Speech-to-Text unterstützt und Bilder oder Antworten auf Suchanfragen, ähnlich der Google Webseiten Suche, präsentiert. Mit Google Now wurde ein verbessertes NLG Interface eingebaut, womit es möglich ist Antworten als natürliche Sprache wiederzugeben. Zudem werden Vorschläge basierend auf vorhergehenden Suchen angezeigt. Kontextabhängige und weiterführende Fragen können jedoch nicht behandelt werden, weshalb Anfragen mit diesem Sprachassistenten nicht mit einem natürlichen Gespräch verglichen werden können.

4 Jahre später wurde der Google Assistent vorgestellt, welcher anfänglich nur auf dem hauseigenen Pixel lief und dementsprechend keine weite Verbreitung erfuhr. Mit der Veröffentlichung von Google Home und der Öffnung des Sprachassistenten für weitere Android Smartphones mit mindestens Marshmallow, 1,5 GB RAM und einem 720p Bildschirm, als auch durch die Öffnung für iPhone Smartphones erzielt der Sprachassistent eine größere Reichweite.

Google Home, welches derzeit nur in den USA verfügbar ist, soll ab August 2017 in Deutschland angeboten werden. Weitere Sprachräume und Länder wie Frankreich und Japan sollen ebenfalls dieses Jahr noch folgen.

3.1 Prinzipien & System Architektur

Google Assistant als proprietäre Software erschwert die Analyse der Systemarchitektur, weshalb in diesem Abschnitt genauer auf die Prinzipien für eine gute Konversation des digitalen Assistenten und dessen Schnittstelle *Actions on Google*, mit der Entwickler die Möglichkeit haben eigene Skills zu implementieren, eingegangen wird.

Google stellt 6 Schritte für eine gute Unterhaltung vor [Google, 2017d]:

- **Einen gemeinsamen Startpunkt finden -** Sprecher A sendet eine Nachricht an Sprecher B.
- **Unterhaltung annehmen -** B nimmt die Unterhaltung mit A an.
- **Bedeutungsinhalt aufbauen -** A und B tauschen sich durch einen strukturierten Ablauf mit einem (meist unausgesprochenen) Kontext aus.
- **Weiterentwickeln -** A oder B (oder beide) lernen etwas aus der Interaktion
- **Zu einer Einigung kommen -** Sowohl A als auch B kommen zu einer Einigung. Falls dies nicht geschieht wird versucht, dass Problem zu beheben.
- **Weitere Handlungen -** Weitere Interaktionen oder Handlungen können als eine Folge der Konversation folgen. Zudem kann auch ein unterbewusstes Ziel (z.B. Zeitvertreib) erfüllt werden.

Ferner verwendet der Assistent ein *Turn-Taking* (Abwechslung von Eingabe und Ausgabephasen) basierten Ansatz. Eine Chance zur Vorherbestimmung eines Wechsels wird sowohl durch die Analyse des Syntax als auch durch die Analyse von Prosodie berechnet, wobei Letztere die Geschwindigkeit der Aussprache, Lautstärke und Tonhöhe beinhaltet. Durch die Einbindung dieser Aspekte wird u.a. der Punkt bestimmt für einen sogenannten Sprecherwechsel, um eine flüssige Konversation zu führen und keine lange Pausen entstehen zu lassen oder das Wort des Anderen abzuschneiden.

Die Prinzipien des linguistischen Philosophen Paul Grice [Bowe et al., 2014] für eine kooperative Konversation wurden ebenfalls angewandt und beschreiben, dass eine Unterhaltung wahrheitsgemäß, informativ, relevant und klar strukturiert sein muss.
Nach Analyse der Prinzipien des Assistenten soll nun eine sogenannte *Action* generiert werden. Dabei wird die Platform *Actions on Google* genutzt, um eine App zu erstellen, die eine Action definiert und eine Schnittstelle mit dem Google Assistenten herstellt. Actions werden in zwei Komponenten unterteilt: (1) einen *Intent*, welcher die Aktion beschreibt und einen *Fulfillment*, welcher einem Webservice entspricht ist und jegliche Logik beinhaltet,um die Aktion auszuführen.

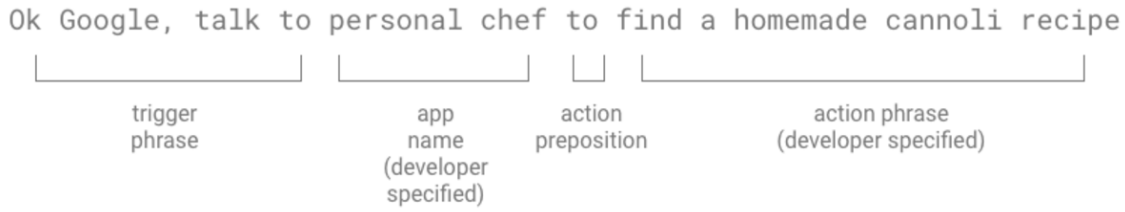


Abbildung 3.1: Actions on Google: Muster zum Aufruf eines Intents einer App [Google, 2017b]

Um einen Intent einer dafür entwickelten App aufzurufen, muss ein gewisses *Pattern* (Muster) eingehalten werden, damit die Anfrage von dem Google Assistent korrekt an die zutreffende App weitergeleitet werden kann. Dabei wird zu Beginn die *Trigger Phrase Ok Google, talk to* genannt, um den Google Assistenten aufzurufen und zu definieren, dass eine Aktion ausgeführt werden soll. Darauf folgt der Name der App (in diesem Beispiel *personal chef*), welche womöglich mehrere Intents definiert hat. Am Ende wird der eigentliche Zweck, den der Nutzer ausführen möchte genannt (*find a homemade cannoli recipe*). Dieser muss darüber hinaus vom Entwickler der App als Action definiert sein.

Anzumerken ist an dieser Stelle außerdem, dass nicht zwingend eine lokal programmierte App notwendig ist. Intents mithilfe der Webseite API.AI und selbst erstellter Webhooks zur Erfüllung der Intents (*Fulfillment*) genügt für einen Cloud-basierten Ansatz.

Nachdem der Nutzer eine Anfrage gestellt hat, werden folgende Aktionen ausgeführt [Google, 2017a]:

- Der Google Assistent nutzt Actions on Google und ruft die App auf, welche den Intent am besten erfüllen kann.
- Actions on Google sendet eine Anfrage an die Fulfillment Komponente der App und erwartet eine Antwort.
- Die Antwort wird in der UI des Google Assistenten angezeigt und die Konversation des Nutzers und der App beginnt.
- Weitere Anfragen werden direkt zwischen dem Nutzer und der App ausgetauscht und in der UI weiterhin angezeigt, bis der Nutzer keine weiteren Informationen oder Anfragen benötigt.

Um die Funktionalität des Google Assistenten um eigene Actions zu erweitern, kann eine lokale App mit Hilfe des Actions SDKs erweitert oder die Platform API.AI genutzt werden, welche die Funktionen des Actions SDKs einbindet und um weitere nützliche Funktionen, wie einer IDE, NLU Komponente oder maschinellen Lernen ergänzt. Aufgrund der zusätzlichen Funktionen wird im folgenden die Plattform API.AI genauer analysiert, wobei die Nutzung des Action SDKs analog zu betrachten ist.

3.2 Erweiterbarkeit

Um den digitalen Sprachassistenten um eigene Funktionen erweitern zu können, ist ein Google Konto erforderlich, womit ein Actions on Google Projekt erstellen kann. Dieses Projekt wird daraufhin mit der Platform API.AI verbunden und relevante Einstellungen (u.a. die Sprache der zu verarbeitenden Sprache) gesetzt. Google selbst bietet ein Sample Projekt an, dessen Struktur anhand der Abbildung 3.2 erkennbar ist.

Nachdem das Projekt erstellt wurde, kann dieses um Intents und *Entities* (Entitäten) erweitert werden. Ein *Default Welcome Intent* muss definiert sein, damit dieser als

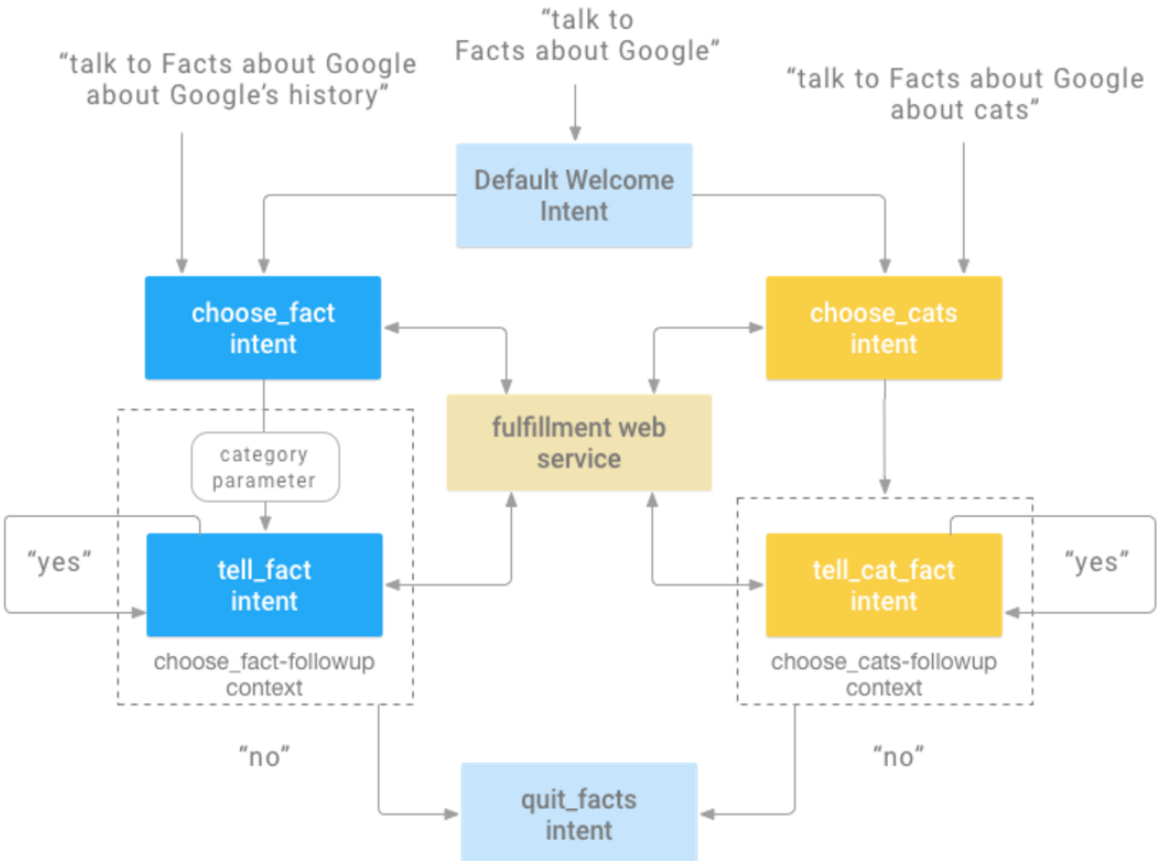


Abbildung 3.2: Intentstruktur des Actions on Google Beispiel Projekts [Google, 2017c]

erstes aufgerufen wird, sobald die App startet und kein deep linked Intent spezifiziert ist. Im angeführten Beispiel ist dies der Fall, wenn der Assistent die Aussage *talk to Facts about Google* bekommt. Ein deep-link würde aufgerufen werden (insofern die App dies unterstützt), wenn ein direkter vordefinierter Intent zusätzlich mitgegeben wird (z.B. *talk to Facts about Google about Google's history*). Des Weiteren gibt es kontextabhängige Intents, die es erleichtern weiterführende Fragen zu stellen. In der Abbildung 3.2 wäre dies im *choose_fact-followup context*. Dabei reicht es nach einer anfänglichen Frage bezüglich der Fakten über Googles Vergangenheit, weiterführende Aussagen in der Form von: *erzähle mir noch einen* zu stellen. Die Häufigkeit, damit eine solche Aussage weiterhin behandelt wird kann durch das Datenfeld *lifespan* (Lebensdauer) eines übergreifenden Intents festgelegt werden.

Entitäten werden dazu genutzt, Parameter Werte aus dem Text zu extrahieren. Dazu gibt es auf der Plattform API.AI drei verschiedene Typen [api, 2017]: (1) *System Entities* sind vordefiniert und enthalten bekannte Parameterextraktionen (z.B. Extraktion der Uhrzeit und des Datums aus dem Satz). (2) Die *Developer Entities* sind durch den Entwickler erstellt worden und lassen genauere Parameterextraktionen, sowie Synonyme für bekannte Phrasen zu. Ein Beispiel für Entitäten inklusive Synonyme für die Parametrisierung von Wetter-Zuständen ist in Listing A.2 zu finden. (3) Die *User Entities* werden auf Session Basis vergeben und können individuell an Nutzer angepasst werden.

Nachdem alle Intents und Entitäten erstellt wurden, muss die Logik zur Verarbeitung einer aufgerufenen Aktion implementiert werden. Angenommen der Wetter Intent in Listing A.1 wird aufgerufen, indem der Nutzer den Assistenten fragt: *Wie ist heute Morgen das Wetter in Berlin*, dann muss ein Cloud Service des Entwicklers existieren, der aus den Parametern Stadt und Datum eine Wettersuche durchführt. Dieser Service wird mit Hilfe eines Webhooks in der Form einer POST-Anfrage angesprochen. Ein möglicher Ausschnitt einer Anfrage könnte wie folgt aussehen:

```
1 "result": {
2         "parameters": {
3             "city": "Berlin",
4             "date-time": "2017-07-13T05:00:00Z/2017-07-13T10:00:00Z
    "
5         },
6         "contexts": [],
7         "resolvedQuery": "Wie ist heute Morgen das Wetter in Berlin
    ",
```

Listing 3.1: API.AI POST Request

Die Antwort des Cloud-Services muss ebenfalls im JSON Format per HTTP POST Methode zurückgeschickt werden und folgende Parameter enthalten:

- **speech** - Die Antwort auf die Anfrage.
- **displayText** - Die Antwort, die auf dem Gerät angezeigt werden soll.
- **data** - Weitere Daten, die für die klientseitige Aktion erforderlich sind.
- **contextOut** - Den Kontext anpassen (z.B. Lebensdauer eines Intents herunter-setzen).
- **source** - Beschreibung der Datenquelle.
- **followupEvent** - Das nächste Event/ Intent der ausgeführt werden soll (optional mit Parameterübergabe).

3.3 Implementierbarkeit

Für die Implementierung des Google Assistenten wird das Assistant SDK angeboten und kann in zwei verschiedenen Ausführungen genutzt werden: (1) Die Google Assistant Library und (2) die Google Assistant gRPC API. Dabei ist Erstere in Python geschrieben und benötigt eine Linux-ARMv7l Architektur (z.B. den Raspberry Pi 3 B). Letztere läuft auf allen Plattformen, die gRPC (general-purpose RPC infrastructure) unterstützen (Googles intern entwickelte Interprozesskommunikations-infrastruktur).

Da Actions on Google derzeit nur englische Sprache als Eingabe und Ausgabemedium akzeptiert, erfüllt diese Plattform nicht die Anforderungen für den Anwendungsfall zur Kommunikation mit einem Roboter in einem Altenheim. Daher wird dieses Projekt durch API.AI realisiert und dessen bereits vorgestellte Struktur genutzt. Intents und Entitäten ähneln sich jedoch dem Actions SDK sehr stark, weshalb für einen späterer Umstieg auf dieses nur marginalen Änderungen notwendig

wären. Dementsprechend sind für ein funktionsfähiges Assistenten-System folgende Komponenten notwendig:

- STT Modul
- TTS Modul
- API.AI Anbindung
- Logik zur Verarbeitung der Anfrage (lokal / cloud-service)

Kapitel 4

Apple Siri

Siri wurde 2003 als DARPA Projekt des Forschungsinstituts SRI International (Stanford Research Institute der Stanford Universität) [Harshita Phatnani, 2015] gegründet und finanziert. Nachdem 2008 dessen Finanzierung eingestellt wurde, richtete es sich neu als Siri International (SI) aus und sammelte in zwei Kaptialrunden über 24 Millionen Dollar. 2010 erschien der Siri Digital Assistant als kostenlose App im App Store und wurde vier Monate später von Apple für circa 200 Millionen Dollar aufgekauft. Siri wurde in das Betriebssystem iOS eingebunden und mit der Backend Struktur Apples verbunden, um auf dessen Services zuzugreifen. Weitere Services u.a. Wolfram Alpha und Wikipedia wurden ebenfalls mit eingebunden.
Siri wird auf den Geräten iPhone 4s oder neuer, iPad Pro, iPad Air oder neuer, iPad (3. Generation oder neuer), iPad mini oder neuer und iPod touch (5. Generation oder neuer) unterstützt. 'Hey Siri' wird auf dem iPhone SE, iPhone 6s, iPhone 6s Plus und iPad Pro (9,7"), ohne dass die Geräte an den Strom angeschlossen sind, sowie auf dem iPhone, iPad und iPod touch mit iOS 8 oder neuer, wenn die Geräte an den Strom angeschlossen sind, unterstützt [AppleEa].

4.1 Prinzipien & System Architektur

Bevor die System Architektur und Erweiterbarkeit durch Skills analysiert wird, werden zunächst einige *Human Interface Guidelines* [AppleEc] für Siri vorgestellt. Darunter fal-

len generelle Design- und Implementierungsentscheidungen zur Erstellung einer App, die den Sprachassistenten zur Unterstützung von Eingaben und Ausgaben verwendet.

- Strebe eine sprachliche Konversation an, die es nicht erfordert den Bildschirm zu betrachten oder darauf Aktionen zu vollführen.
- Versuche Interaktion zu minimieren und Antworten schnell zu liefern.
- Falls Nutzer auf die App weitergeleitet werden, soll dies unmittelbar zu dem Content führen.
- Erhöhe die Genauigkeit durch bereitgestellte Vokabulars.

Nachfolgend wird, soweit es aus Dokumenten und Patenten ersichtlich ist, die System Architektur Siris analysiert. Dabei nutzt Siri mehrere Technologien: Nuance Communications Spracherkennung und Text-to-Speech (TTS) Technologie, Siris artificial intelligence (AI) natural language processing engine und backend Services (z.B. *Knowledge Graph*). Ein Knowledge Graph kann als eine Repräsentation von Daten, Fakten und Informationen über die Welt beschreiben, verstanden werden. Erst im Mai 2017 kaufte Apple die Firma Lattice Data und fügte deren Datenbanken in den eigenen Knowledge Graph ein.
Die Abbildung 4.1 zeigt ein Block Diagramm einer möglichen Verkörperung der Konfiguration von Siri. Das Diagramm wurde im Januar 2012 durch das Patent [Gruber et al., 2012] veröffentlicht und zeigt sehr deutlich welche Komponenten für einen Sprachassistenten notwendig sind. Wie auch schon im Kapitel Dialog System, verwendet der *Intelligent Automated Assistant* viele Komponenten, die schon untersucht wurden. Darunter fallen u.a. der Sprachen Interpreter, das Vokabular, die Domain Modelle oder der Output Processor zur Generierung natürlicher Sprache. Unter Services 1084 können sowohl die Backend Services als auch die bereitgestellte API-Schnittstellen weiterer Apps z.B. der Kalender-App verstanden werden.

Die angeführte Abbildung 4.2 verdeutlicht die Kommunikation zwischen Klient und Server. Interessant ist die Nutzung eines lokalen Caches für Personal Memory und eines Vokabulars. Dadurch ist es möglich, netzwerkunabhängige Anfragen zu stellen und

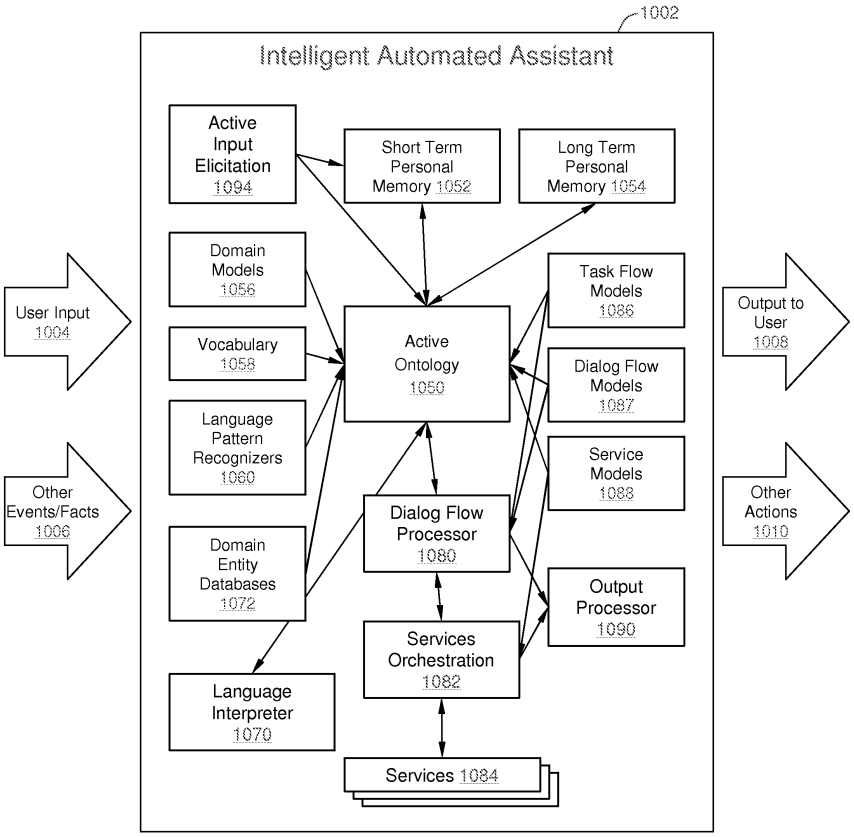


Abbildung 4.1: Eine mögliche Konfiguration der System Architektur von Siri [GRUBER et al., 2012]

lokal zu verarbeiten. Dies erscheint nützlich, falls der Nutzer z.B. einen Wecker stellen möchte und keine Netzwerkverbindung besitzt. Verglichen mit anderen Sprachassistenten ist dieses Feature natürlich in den meisten Systemen vertreten. Diese Abbildungen sind nur ein Ausschnitt einer möglichen System Architektur des Sprachassistenten Siri. Da das System wie die meisten Systeme proprietär ist, reicht diese Analyse und es wird nun die Erweiterbarkeit durch eigene Skills aufgezeigt.

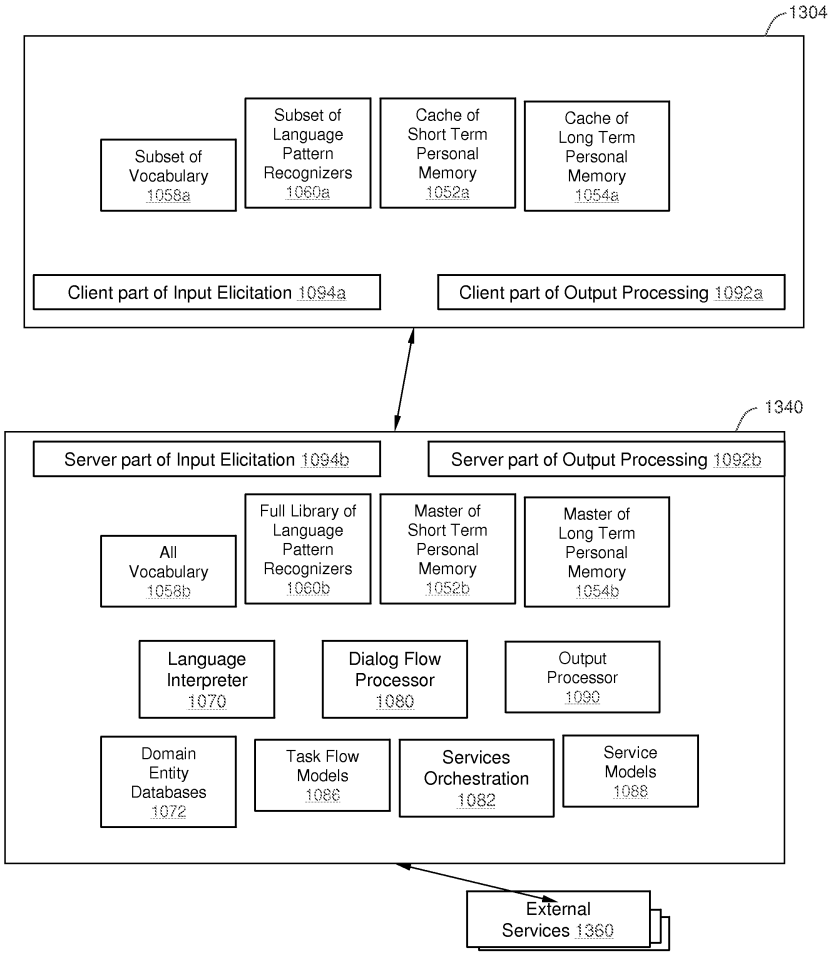


FIG. 7

Abbildung 4.2: Kommunikation eines Klienten und Servers inklusive der benötigten Module für Siri [GRUBER et al., 2012]

4.2 Erweiterbarkeit

Zur Entwicklung eigener Skills bietet Apple das SiriKit [APPLEb] an, welches auf das Intents- und Intents UI Framework zugreift, wodurch *app extensions* für die Realisierung eigener Services mit Siri oder Maps Funktionalitäten geschrieben werden können.

Diese Erweiterungen können in zwei Typen eingeteilt werden:

1. **Intents app extension**, die eine Nutzer Anfrage von SiriKit annimmt und direkt in eine Aktion in der lokalen App des Entwicklers ausführt. Zum Beispiel wenn der Nutzer Siri bittet ein Taxi zu rufen. Diese Anfrage könnte an eine Taxi App, mit den Parameter Ort weitergeleitet werden.

2. **Intents UI app extension**, die Informationen direkt im Siri- oder Maps Interface darstellt, nachdem die Anfrage von der Intents app extension beantwortet wurde. Eine Erstellung dieser Erweiterung ist optional.

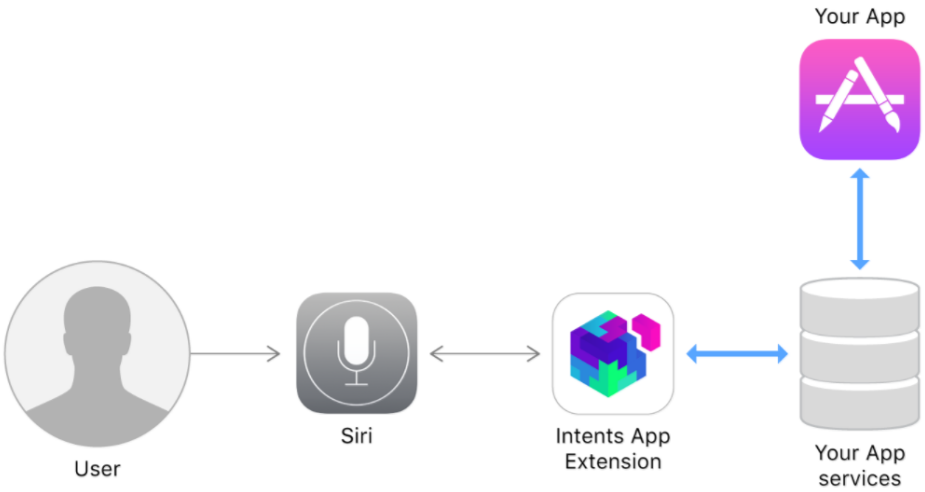


Abbildung 4.3: Eigene Services für Siri und Maps zur Verfügung stellen [APPLEb]

Vergleicht man die Möglichkeiten die Google Assistant und Apple Siri zur Verfügung stellen, um *third-party* Entwicklern eigene Skills und Aktionen für deren Assistenten zu entwickeln, wird schnell deutlich, dass die Funktionalitäten Siris sehr stark eingeschränkt werden. In Tabelle 4.1 wird gezeigt, welche Interaktionen von Siri unterstützt werden. Fällt die App in keine dieser Kategorien wird sie von Apple nicht in den App-Store zugelassen.

Zur Erstellung einer App Extensions App wird im Projekt Editor zunächst die *Capability* (Erweiterung) Siri ausgewählt und eine kurze Beschreibung der ausgetauschten Informationen mit Siri in den *NSSiriUsageDescription key* der Info.plist geschrieben. Der letzte Schritt der Autorisierung ist eine Benachrichtigung für den App-Nutzer, der

Tabelle 4.1: Unterstützte Intent Domänen von SiriKit [APPLEe]

Typ der App	**Unterstützte Siri Interaktionen**
Audio und Video Anrufe	Anruf starten Anruf Verlauf durchsuchen
Nachrichten	Nachricht senden Erhaltene Nachricht vorlesen Nachrichten durchsuchen
Bezahlung Dienste	Überweisung tätigen Rechnung bezahlen Rechnungen durchsuchen
Photo Management	Nach Photos suchen und anzeigen lassen
Fitness Aktivitäten	Workout starten, pausieren, beenden oder abbrechen
Transport Dienste	Fahrt buchen Informationen zu der Fahrt suchen
Automobil Apps	Diverse Steuerungen des Autos
Listen und Notizen	Erstellung und Management von Notizen und To-Do Listen
Scannen von Codes	Scannen von QR Codes zur Kontaktaufnahme oder Bezahlung
Restaurant Reservierungen	Reservierungen durch Maps

die Siri Anbindung akzeptieren muss. Der Quelltext zur Abfrage dieses Dialogs ist in Listing A.3 gegeben.
Vergleichbar mit den Intents des Google Assistant, läuft die Kommunikation zwischen Siri und der App so ab:

1. SiriKit untersucht die Intent Extensions *Info.plist* der App und stellt fest, welche Intents von der App unterstützt werden.

2. SiriKit öffnet daraufhin die NSExtensionPrincipalClass fordert den gewünschten Handler für diesen Intent an.

3. Nachdem SiriKit den Intent mit der gewünschten Anfrage und Parametern an den *Handler* übergeben hat, kontrolliert die App fehlende Parameter.

4. Die App hat nun die Möglichkeit fehlende Informationen zur Bearbeitung der

Anfrage zu stellen. Falls dies nicht der Fall ist, wird die Anfrage in der App bearbeitet und dessen Antwort zurück über einen weiteren Intent an Siri geschickt.

Um die Anschaulichkeit zu verdeutlichen wird im Anhang noch einmal genauer auf den Punkt zur Bearbeitung der Parameter des Intents eingegangen. Listing A.4 zeigt, wie eine Transport App überprüft, ob der Standort, den der Nutzer angegeben hat gültig ist. Die Funktion (Handler) bekommt den Transport Intent und testet ob dieser Standort in dem gültigen Areal ist und teilt dies anschließen mit.

4.3 Implementierbarkeit

Für den Einsatzzweck zur Erweiterung eines Sprachassistenten und Implementierung in einen Roboterkontext ist es nicht möglich Siri weitgehend anzupassen. Abgesehen davon ist es nur möglich Skills für diesen zu entwickeln, wenn eine Extensions angepasste App lokal auf einem iOS oder watchOS Gerät installiert ist. Mit der Einführung des Homepod von Apple wird sich diese Lage leider nicht verändern, da dieser nicht durch Skills erweitert werden kann und primär für die Home-Automatisierung und Musiksteuerung geplant ist.

Kapitel 5

Microsoft Cortana

Microsoft Cortana ist ein weiterer Mitstreiter unter den persönlichen Sprachassistenten und wurde von auf der Microsoft-Konferenz Build 2014 von Joe Belfiore im April vorgestellt. Ursprünglich als Applikation für das Windows Phone 8.1 und nur als Preview für Einwohner der USA in englischer Sprache verfügbar, fand er seit Anfang 2015 Einzug in Windows 10 Desktop PCs als vorinstallierter Assistent. Der Name Cortana stammt aus der Computerspiel Reihe *Halo* und verkörpert eine künstliche Intelligenz, die dem Protagonisten jederzeit zur Verfügung steht und nützliche Aufgaben erledigt. Die Stimme Cortanas basiert auf der Synchronsprecherin Jen Taylor [Mayword, 2015].
Derzeit werden die Plattformen Windows 10, Android, iOS, Xbox und monitorlose Geräte wie dem Harman Kardon Invoke, welcher im Wettbewerb mit Google Home, Amazon Alexa und Co. treten soll und im Mai 2017 vorgestellt wurde von Cortana unterstützt.

5.1 Prinzipien & System Architektur

Prinzipien und Richtlinien zur Erstellung von Skills für Cortana werden anhand der Design Guidelines von Microsoft [MicrosoftE] vorgestellt.

Gute Anwendungsszenarien:

- **Falls der Nutzer keine freie Hand zur Verfügung hat** - Z.B. kann eine sprachliche Interaktion beim Kochen genutzt werden um von dem Assistenten Antworten bezüglich des Vorgehens zu bekommen.
- **Eine schnelle und natürliche Art mit Sprache zu interagieren** - *Teile das mit John* ist sowohl schnell als auch natürlich und erleichtert die Bedienung.
- **Zeit sparen** - *Spiele die letzte Folge House of Cards* ist zeitlich effizienter, als die notwendige App zu öffnen und nach der letzten Folge House of Cards zu suchen.
- **Beim Multitasking assistieren** - Falls an einem Dokument oder in einer anderen Form gearbeitet wird, kann der Assistent genutzt werden um (eingehende) Nachrichten zu lesen oder zu verschicken.
- **Falls der Nutzer abgelenkt oder beschäftigt ist** - Wenn der Nutzer Auto fährt oder zu Fuß unterwegs ist, kann das Gerät mit Sprache bedient werden ohne eine Beeinträchtigung der primären Tätigkeit.

Schlechte Anwendungsszenarien:

- **Komplexe Anweisungen sind erforderlich** - Manche Aufgaben erfordern komplexe Anweisungen, etwa das Eintippen einer langen URL.
- **Private Informationen** - Informationen, die der Nutzer nicht in der Öffentlichkeit laut aussprechen möchte. Passwörter und andere vertrauliche Daten dürfen nicht abgefragt werden und ein Skill der diesen Parameter erwartet, wird die Überprüfung durch Microsoft zur Veröffentlichung nicht passieren.
- **Unnatürliche und schwer zu merkende Phrase** - Der Nutzer muss Phrasen sagen, die unnatürlich sind z.B. *spiele Genre Jazz* anstatt *spiele etwas Jazz.*
  Dies gilt auch, wenn vorausgesetzt wird, dass sich viele verschiedene und

komplexe Anfragen gemerkt werden.

Microsoft vergleicht die Entwicklung eines Skills mit der einer App oder einer Webseite, bei der das User Interface (UI) und die User Experience einen wesentlichen Bestandteil ausmachen und verweist auf den folgenden Design Prozess.

Abbildung 5.1: Cortana Skill Design Prozess [MICROSOFTe]

1. **Szenario definieren** - Was ist der Einsatzzweck des Skills und wird dieser mit Spracheingabe und Ausgabe harmonieren. Kann die Aufgabe schnell erledigt werden, benötigt wenig Interaktion und ist nicht von der Anzeige oder Bestätigung von Inhalten auf dem Bildschirm abhängig.

2. **Dialog** - Erstelle eine bidirektionale Konversation auf dem aufgestellten Szenario und evaluiere die Interaktion (, wenn möglich mit einer Person, die nicht in das Projekt involviert ist) und führe falls nötig Anpassungen durch.

3. **Voice User Interface (VUI) Diagramm** - Erstelle ein Sprach Nutzer Interface Diagramm basierend auf der erstellten Konversation. Erweitere dieses langsam um Hilfe und Fehlerzustände. Teste diesen Prototyp nach jeder Iteration und passe ihn gegebenenfalls an.

4. **Natural Language Processing** - Nutze das VUI um Entitäten, Intents und Aussagen zu erkennen. Identifiziere welche Entitäten und Intents definiert werden müssen und welche bereits vordefiniert sind.

5. **Design untersützende visuelle Karten** - Entwickle die visuelle Komponente des Skills, die in Cortana angezeigt werden soll. Überlade diese nicht mit unnötigen Informationen und überprüfe ob dies die Nutzererfahrung verbessert.

6. **Entwicklung** - Wiederhole einzelne Phasen, bis das Projekt keine weiteren Tests und Anpassungen mehr benötigt.

Da Cortana ebenfalls ein proprietäres System darstellt, wird auf Publikationen seitens Microsoft eingegangen, um dieses System zu beschreiben. Im Folgenden wird sich auf das Paper [SARIKAYA et al., 2016] bezogen, dass durch eine Reihe von Microsoft Corp. Angestellten im Dezember 2016 veröffentlicht wurde und eine abstrakte System Architektur eines *personal digital assistant* beinhaltet (siehe Abbildung 5.2). Ein PDA wird in diesem Paper in eine proaktive und reaktive Komponente unterteilt. Dabei wird die proaktive zu bestimmten Anlässen (Geofences o.ä.) ausgeführt und die reaktive nur, wenn der Nutzer explizit den Assistenten anfordert. Das System wird in drei Kategorien unterteilt:

- Input Verarbeitung (inkl. Language Understanding)
- Dialog State updaten
- Eine *rule-based policy* um die richtige System Aktion auszuführen

In der Input Phase wird die Anfrage (Text oder akustisches Signal) nach einem Domänen spezifischen Schema analysiert, welches *multi-domain*, *multi-turn* und *contextual query understanding* beherrscht. In der Update Phase entscheidet die SCO (Slot/Entity Carry Over), welche Slots von vorherigen Runden in der multi-turn Konversation noch relevant sind. Diese Slots können je nach Bedarf jederzeit oder zu einem Domänen/Intent Wechsel teilweise übergeben oder komplett gelöscht werden. Dieser gespeicherte Kontext wird u.a. dafür genutzt weiterführende Fragen zu unterstützen (vgl. Google Assistant Lifespan der Intents).

- Runde 1: *Wie ist das Wetter heute in New York?* (Wetter)
- Runde 2: *und am Wochenende?* (Wetter)

Die Flexible Item Selection (FIS) wird genutzt, um dem Nutzer mehrere Antwortmöglichkeiten zu liefern. Beispielweise könnte auf die Frage *Zeige mir chinesische Restaurants in der Nähe* der Assistent mehrere Restaurants anzeigen, aus denen der Nutzer

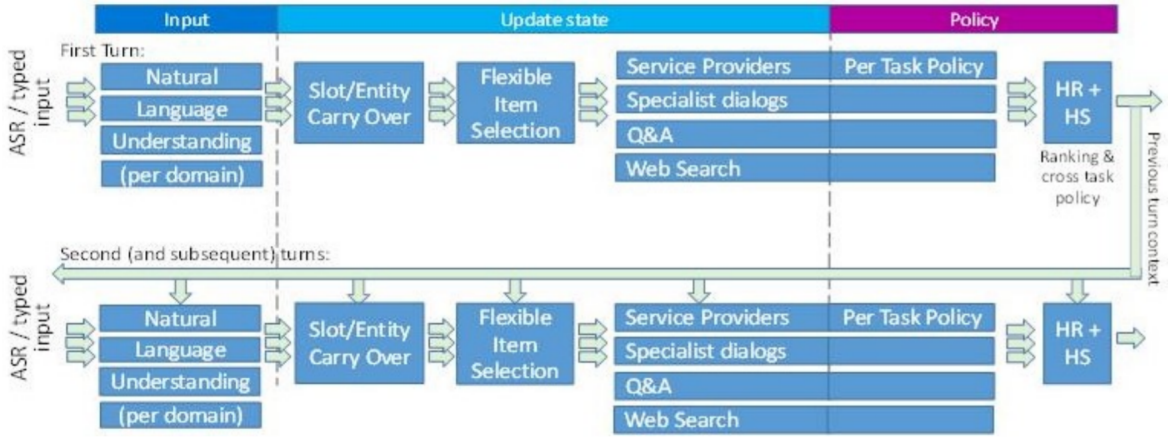


Abbildung 5.2: Cortana: Reaktive System Architektur mit Slot Carry Over und Flexible Item Selection [SARIKAYA et al., 2016]

'kein, ein oder mehrere' für weitere Anfragen auswählen kann. Zu guter Letzt wird anhand der Domain und dessen unterstützten Intents eine *Experience* (kleinste geschlossene Einheit, die ein Benutzer Bedürfnis erfüllt) ausgewählt. Diese Experiences sind auf mehrere Layer aufgeteilt und spezifizieren sich mit aufsteigendem Layer. Der unterste Layer (auch Notfall Experience genannt) führt eine Websuche (Bing-Websuche) auf die gestellte Frage aus. Darauf aufbauend wird u.a. ein von Microsoft betriebenes Q&A System befragt, spezialisierte Dialoge zurückgegeben oder eigene Service Leistungen genutzt. Diese könnten z.B. Yelp & Foursquare Vorschläge beinhalten.

Während jeder Phase werden mehrere Hypothesen parallel angepasst und weitergeleitet. Jede dieser Hypothesen wird am Ende durch den Mechanismus *Hypothesis Ranking and Selection* ausgewertet. Die Hypothese mit dem besten Wert wird verwendet und folglich dem Nutzer ausgegeben.

Erwähnenswert ist weiterhin, dass Cortana als Sprachassistent in der Cortana Intelligence Suite (ehemalig Cortana Analytics Suite) als Dienst eingebunden ist. Die Cortana Intelligence Suite ist weniger ein alleinstehendes Produkt als eine Reihe von Diensten zur Analyse von Datenmengen in der Microsoft Cloud um daraus interessante Fakten, Statistiken oder Aktionen zu generieren. Vergleichbar ist diese Plattform mit dem Ökosystem von IBM Watson. Die Architektur der Cortana Intelligence Suite

ist in der Abbildung 5.3 zu sehen. Auf einige dieser Komponenten wird im Folgenden eingegangen, da sie für Cortana (als Sprachassistent) als Backend Services fungieren [MICROSOFTc].

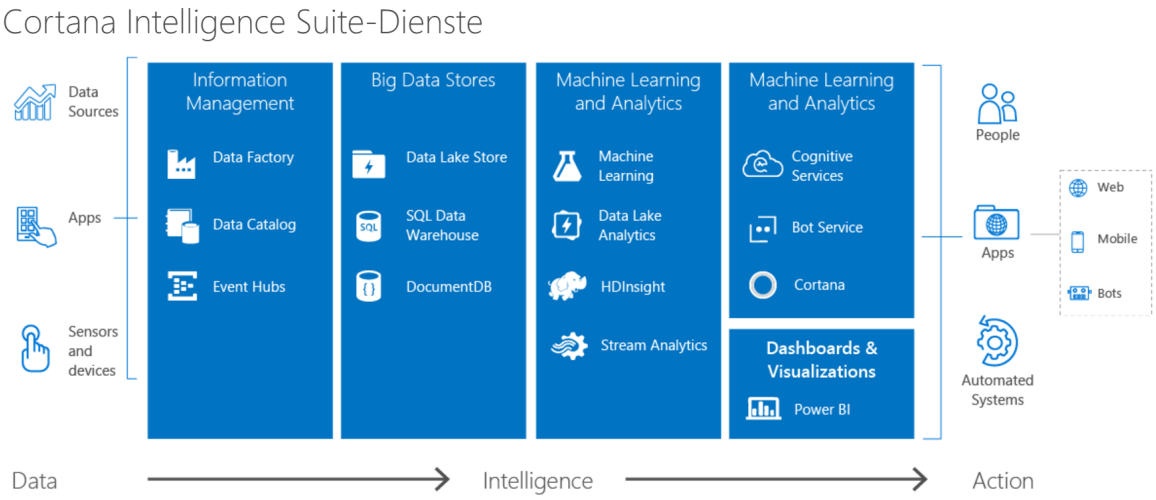


Abbildung 5.3: Cortana Intelligence Suite Dienste [MICROSOFTc]

- **Data Factory** - Hauptaufgabe ist die Koordinierung und Skalierung von Datenströmen einzelner Dienste auf die Azure Cloud.
- **Data Catalog** speichert Meta Daten für eine schnelle und effiziente Suche auf Datensätzen, wodurch mehr Zeit zur Analyse des Inhaltes verbleibt.
- **Data Lake Store** speichert große Mengen an Daten in unstrukturierter, semistrukturierter oder strukturierter Form in der Azure Cloud. Die maximale Datengröße ist nicht limitiert, während einzelne Dateien bis zu PetaByte Größe unterstützt werden.
- **Machine Learning** wird genutzt, um Vorhersagen aus bekannten Daten zu erstellen. Die Hauptaufgaben bestehen u.a. in Anomalie Detektion, Mehrklassen-Klassifikation und Regression.
- **Cognitive Services** sind ein Set aus APIs, SDKs und Cloud Diensten zur Erstellung von Intelligenten Systemen.

- **Bot Framework** ermöglicht Benutzern ein natürlicheren Umgang mit dem Computer durch intelligente Bots. Einige unterstützte Plattformen sind: Facebook Messenger, SMS, Skype und Slack.
- **Cortana** wird verwendet, um den Nutzer mit dem System und dessen Diensten über natürliche Sprache plattformübergreifend zu verbinden.

5.2 Erweiterbarkeit

Microsoft bietet seit Mai 2017 das Cortana Skills Kit [MICROSOFTd] in der *public preview* an, um Extensions bzw. Skills für Cortana zu entwickeln und ist derzeit nur über die Erstellung eines Bots durch das Microsoft Bot Framework möglich, welcher anschließend mit Sprach-, Nutzer- und Kontextinformations Diensten erweitert werden kann.
Einen Cortana Skill zu entwickeln umfasst folgende Schritte [MICROSOFTa]:

1. Erstelle oder verwende einen existierenden Bot durch das BotBuilder SDK. Das Microsoft Bot Framework bietet ein .NET SDK und ein Node.js SDK an und steht als open source Projekte auf GitHub. Zusätzlich ist es über die Azure Bot Plattform möglich, Bots über den Webbrowser zu erstellen.
2. Integriere die LUIS.ai für NLU, Cortana für Sprach- und Kontextinformationen, sowie Bing APIs für Wissensdatenbanken.
3. Lade den Bot in die Azure- oder eine andere -Cloud hoch.
4. Registriere diese anschließend mit dem Bot Framework.
5. Füge den bot zu dem Cortana Channel hinzu.
6. Veröffentliche den zugehörigen Cortana skill.

Da diese Methode derzeit die Einzige ist, nicht viel spezifisches Wissen über Cortana voraussetzt und sich primär auf die Erstellung eines Bots beschränkt, wird ein Einblick in die benötigten Datenstrukturen gegeben. Hierfür soll ein Beispiel mit einer LUIS.ai

Anbindung angeführt werden. Listing A.5 zeigt eine funktionsfähige Applikation für LUIS.ai, die im JSON Format importiert werden kann.

Intens werden aufgerufen, nachdem die Anfrage des Nutzers (*Utterance*) analysiert wurde und mit einem definierten Intent übereinstimmt.

```
1   "intents": [
2     {
3       "name": "SearchHotels"
4     }
5 ]
```

Listing 5.1: LUIS.ai Intent Definition

Entitäten dienen der Extrahierung wichtiger Parameter aus Aussagen. Zum Beispiel wäre es bei einer Hotelbuchung in der Nähe eines Flughafens wichtig, den Code des Flughafens zu parametrisieren (Frankfurt - FRA). LUIS.ai unterstützt derzeit bis zu 30 eigens definierte Entitäten zusätzlich zu den Vordefinierten.

```
1 "entities": [
2     {
3       "name": "Hotel"
4     },
5     {
6       "name": "AirportCode"
7     }
8 ]
```

Listing 5.2: LUIS.ai Entität Definition

Damit Aussagen und Anfragen des Benutzers an den richtigen Intent weitergeleitet werden, müssen die gebräuchlichsten Äußerungen mit den entsprechenden Intents deklariert werden.

```
"utterances": [
  {
    "text": "look for hotels in miami",
    "intent": "SearchHotels",
  }
]
```

Listing 5.3: LUIS.ai Aussagen

In der C# Implementierung des Bots muss die ID und der Subscription Key der Luis.ai Applikation eingetragen werden. Falls diese eine Anfrage des Nutzers entgegennimmt, wird die dazugehörige Funktion (Handler) des Bots ausgeführt und bekommt den Dialog Kontext, die ursprüngliche *Aktivity* mit der Anfrage, sowie die Antwort der LUIS Applikation inklusive analysierter und parametrisierter Intents und Entitäten.

```
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;

[LuisModel("YourModelId", "YourSubscriptionKey")]
public class RootLuisDialog : LuisDialog<object>
{
    ...

[LuisIntent("SearchHotels")]
public async Task Search(IDialogContext context,
  IAwaitable<IMessageActivity> activity, LuisResult result)
{
    ...
```

Listing 5.4: Codeausschnitt aus dem LUIS fähigen Bot [MICROSOFTb]

5.3 Implementierbarkeit

Zur Integration von Cortana in Drittanbieter Geräte bietet Microsoft das Cortana Devices SDK zur Zeit nur in einer private Preview an. Eine Öffnung des SDKs ist für Ende 2017 geplant und ermöglicht dementsprechend keine Implementierung des Assistenten in den Roboter. Eine Einbindung der NLU Komponente (LUIS.ai) in ein eigenes System mit TTS und STT Modulen sowie eigenen Backend Diensten wäre jedoch weiterhin möglich und wird künftig bei der konkreten Implementierung im Kontext eines mobilen Roboters berücksichtigt.

Kapitel 6

Bixby

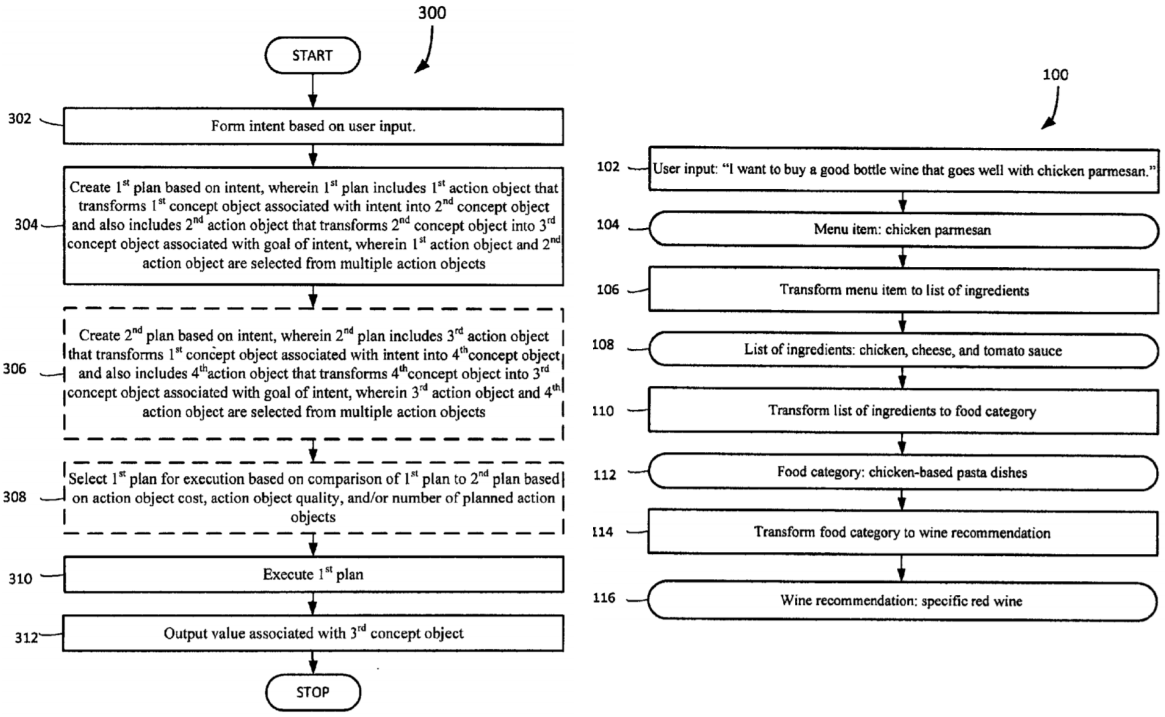
Samungs Assistent Bixby wurde im März 2017 während der Samsung Galaxy Unpacked Veranstaltung offiziell vorgestellt und ist seit April vorerst nur in Korea vollständig- und in Englisch teilweise nutzbar. Aufgrund der geringen Informationen seitens Samsung und einer derzeit nicht vorhandenen Entwickler Schnittstelle zur Erstellung von eigenen Skills wird dieser Assistenten nur kurz vorgestellt.

Bixby basiert zum einen auf seinem Vorgänger Samsung S-Voice, der ab 2012 für Samsung Geräte verfügbar war und nur rudimentäre Aufgaben (nach dem Wetter zu fragen oder einen Wecker stellen) beherrscht und zum anderen auf dem im Jahr 2016 akquirierten Assistenten Viv. Viv wurde seit 2012 von dem Entwicklungsstudio Six Five Labs, Inc. entwickelt, welches aus ehemaligen Siri Entwicklern besteht. Verglichen mit Siri, Google Assistant und Co soll dieser komplexere Anfragen beantworten und durch Drittanbieter sehr stark erweitert werden können. Wie viel Nutzen Samsung aus diesem Assistenten in Bixby gesteckt hat ist derzeit unklar und wird sich in Zukunft zeigen.

Bixby läuft aktuell offiziell nur auf dem Samsung Galaxy S8 und S8+, wurde jedoch auch erfolgreich auf ältere Samsung Geräte mit Android Nougat und höher portiert. Die Anzahl der Bixby kompatiblen Apps beläuft sich auf einige Samsung Apps, sowie u.a. folgende Drittanbieter Apps [Nasreen]: WhatsApp, Facebook, YouTube, Instagram, Twitter, Tumblr, Gmail, Google Play Store, Google Play Music, Google Maps, Uber, Pandora, Yelp.

6.1 Prinzipien & System Architektur

Da bezüglich der Prinzipien und System Architektur Bixbys keine Informationen vorliegen zeigt dieser Abschnitt eine soweit mögliche Analyse vom akquirierten Assistenten Viv. Die Abbildung 6.1 zeigt ein Blockdiagramm mit einer sich dynamisch entwickelnden System Architektur durch Einbindung von Drittanbieter Anwendungen. *Action Objects* werden als Rechtecke und *Concept Objects* als Ovale dargestellt.
Der Nutzer möchte von dem Assistenten in diesem Beispiel eine Flasche Wein kaufen, die zu seinem Essen - Parmesan Hähnchen - passt. Das System wertet diese Anfrage aus und erstellt einen Intent, dass der Nutzer einen Wein Empfehlung benötigt basierend auf dem Konzept Objekt **104** für ein Parmesan Hähnchen. Aufgrund dessen, dass kein Drittanbieter einen spezifischen Anwendungsfall zur Verfügung stellt,



(a) Konzept (b) Beispiel

Abbildung 6.1: Dynamisch entwickelnde kognitive System Architektur basierend auf Drittentwicklern [Gabel et al., 2014]

fertigt das System einen Plan für Transformationen des ursprünglichen Intents

mit Aktions Objekten an, die möglicherweise sequentiell ausgeführt werden, bis eine Erfüllung der Anfrage möglich ist. Im folgenden Beispiel wird das Parmesan Hähnchen zunächst in seine Zutaten zerlegt (Konzept Objekt **108**), welches dann in das Konzept Objekt *Food category: chicken-based pasta dishes* **112** transformiert wird. In der letzten Phase wird diese Essenskategorie auf die Anfrage nach einer Weinempfehlung angewendet und mit der Antwort *Red Wine* beantwortet und dem Nutzer ausgegeben.

Das Interessante an dieser System Architektur ist, dass obwohl das System die sehr spezielle Anfrage nach einer Weinempfehlung für ein spezifisches Gericht nicht direkt beantworten kann und kein Drittanbieter einen solchen Intent anbietet, kann das System eine Empfehlung durch einen Ablaufplan mit sequentiellen Transformationen des Intents synthetisieren kann.

Aus Übersichtlichkeit ist im vorherigen Beispiel ein sequentieller Plan mit drei Aktions Objekten und vier Konzept Objekten gegeben, jedoch erstellt das System unter normalen Bedingungen eine Reihe von Plänen, die sich durch unterschiedliche Iterationen, sequentiellen Abläufen, Teilungen, Verbünde oder der Anzahl an Transformationen unterscheiden können. Zum Zeitpunkt der Nutzeranfrage, kann das System seine volle Funktionalität nicht einschätzten und schreibt sich einen dynamischen Programm zur Erfüllung dessen. Bis auf einige wenige beabsichtigten Intents durch die Systementwickler ist dieses Verhalten gewünscht und bietet eine Möglichkeit komplexe Anfragen zu behandeln, die nicht expliziert definiert sind [Gabel et al., 2014].

6.2 Ausblick

Da sich der Assistent in einem frühen Stadium befindet, momentan nur in englischer und koreanischer Sprache nutzen lässt, kein SDK zur Entwicklung eigener Skills existiert, sowie nur auf einer kleinen Anzahl an Geräten (Samsung Galaxy S8/S8+) verfügbar ist, kann dieser Assistent sich bislang noch nicht mit den Alternativen messen und in dem Anwendungsfall eines mobilen Roboters noch nicht genutzt werden. Ein Release in Deutschland ist für das 4. Quartal 2017 geplant. Abgesehen davon bietet

Bixby einige neuartigen Features [SAMSUNG]:

- **Bixby Voice** - Gute Anbindung an eigene Samsung Apps zur Steuerung diverser Aufgaben (z.B. Erstelle ein Album mit Fotos von Hawaii)
- **Bixby Vision** - Nutzt die Kamera des Samsung Galaxy S8 um Informationen über Sehenswürdigkeiten oder Geschäfte zu suchen, einen Text in eine andere Sprachen zu übersetzten oder ein eingescanntes Produkt bei einem der Partner zu kaufen.
- **Bixby Reminder** - Erstellt Erinnerungen durch Bixby Voice oder direkt in unterstützten Apps. Video-, Foto- oder Webseiten Verknüpfungen sind ebenfalls möglich.
- **Bixby Home** - Zeigt kontextbasiert relevante Apps und Dienste auf einer einzelnen Seite an. Dies ist vergleichbar mit den Google Assistant oder Apple Siri Karten und integriert Voice, Vision und Reminder.

Kapitel 7

IBM Watson

IBM Watson ist im Gegensatz zu den anderen bereits vorgestellten Assistenten ein Ökosystem mit verschiedenen APIs, Diensten und Anwendungen. Einen dedizierten persönlichen Assistenten vergleichbar mit Siri gibt es nicht, denn Watson ist für Businesslösungen konzipiert und basiert auf anwendungsspezifischen Daten, die keinem Knowledge Graph angehören.

Watson wurde Ende der 2000er als *question answering* System für die Quiz Show Jeopardy unter der Bezeichnung *DeepQA Projekt* und der Leitung von David Ferrucci entwickelt [Ferrucci et al., 2013] und nach dem ersten CEO IBMs Thomas J. Watson benannt. 2011 schlug Watson Ken Jennings, die 74 Runden ungeschlagen war und Brad Rutter, der den bislang größten Gewinn von 3.25 Millionen Dollar erzielte, in dem Spiel Jeopardy und strich eine Million Dollar ein. Dafür standen dem System 200 Millionen Seiten strukturierter und unstrukturierter Daten zur Verfügung, die Offline eine Größe von 4 Terabytes einnahmen und durch eine Zusammenschaltung von 90 IBM Power 750 Servern mit 2880 Prozessoren und 15 Terabyte Arbeitsspeicher analysiert werden konnten, um eine Antwort in unter drei Sekunden zu liefern.

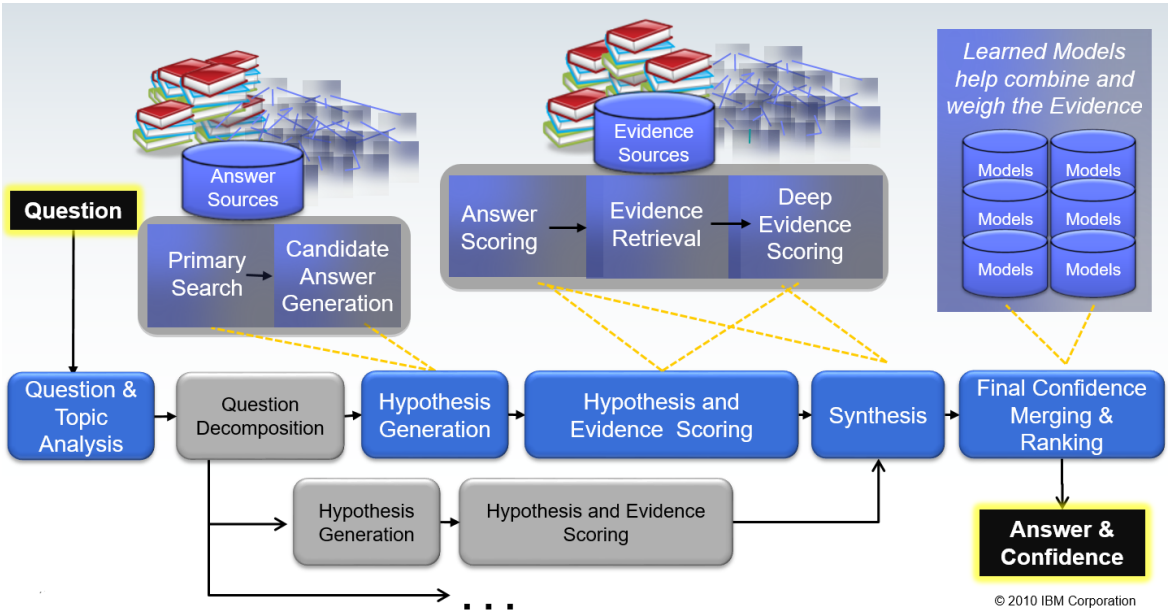


Abbildung 7.1: Massively Parallel Probabilistic Evidence-Based Architecture [Ferrucci et al., 2010]

7.1 Prinzipien & System Architektur

Da IBM Watson derzeit ein Ökosystem ist, dass mehr als 100 verschiedenen APIs umfasst, wird in diesem Kapitel auf die System Architektur des damaligen QA Systems eingegangen, sowie Prinzipien und Design Guidelines zur Erstellung eines Dialogs mit der Conversation API aufgezeigt.

Das finale Watson System, welches im Jahr 2011 im Jeopardy Wettbewerb antrat, bestand aus über 500.00 Zeilen Java Code und wurde von dem DeepQA Team in circa vier Jahren entwickelt. Der erste Schritt zur Erstellung eines solchen Systems bestand in der Datenakquirierung, der Analyse der Art der Fragestellung und der verschiedenen Domänen. Die Analyse der Semantik und Syntaktik von Beispielfragen wurde manuell erledigt, während die Charakterisierung der Domänen automatisiert werden konnte. Durch eine sogenannte *Lexical Answer Types* (LAT) Analyse konnten Statistiken bezüglich der Auftrittswahrscheinlichkeit einer Domäne festgestellt werden. Beispiele für solche Typen sind z.B. Fragen nach Ländern,Städten, Filmen oder Autoren. Als Basis

und Informationsquelle dienten Watson 2011 eine Reihe von Enzyklopädien, Thesauri, Nachrichtenartikel, Literaturen, sowie weitere Sammlungen. In der Abbildung 7.1 wird die DeepQA High-Level Architektur aufgezeigt, auf die genauer beleuchtet wird [Ferrucci et al., 2010].

Question Analysis ist der erste Schritt zur Beantwortung einer Anfrage und bestimmt den weiteren Verlauf durch eine initiale Analyse. Dabei wird der Satz auf logische Formen, Entitäten, Relationen, Kreuzverweise und viele weitere Strukturen untersucht und klassifiziert. Diese Klassifizierung wird durch mehrere Algorithmen realisiert, wovon die LAT Klassifizierung bereits vorgestellt wurde. Ein einzelnes Interrogativpronomen oder -adverb kann, ohne jegliche Untersuchung auf Semantik, den Typ der Frage festlegen.

Decomposition arbeitet nach dem Prinzip Teile und Herrsche und teilt eine Frage in mehrere Teilprobleme oder Teilfragestellungen auf und bearbeitet diese parallel. Nach einem regelbasierten *deep parsing* und einer statistischen Klassifikation wird überprüft, ob und wo die Frage zerteilt werden soll. In der Abbildung 7.1 werden die Subanfragen in grau dargestellt.

Hypothesis Generation erstellt mehrere Antwortmöglichkeiten, indem es alle möglichen Quellen durchsucht und kleine *Schnipsel* extrahiert. Diese werden im späteren verlauf genauer untersucht, fallen gelassen und bewertet. Die Hypothesen Generierung besteht aus zwei Phasen:

1. **Primary Search** - In dieser Phase werden so viele potentiellen Antworten in den verfügbaren Quellen gesucht, in der Hoffnung, dass sich eine von diesen im späteren Verlauf als richtig erweist.
2. **Candidate Answer Generation** - In der zweiten Phase werden aus den Ergebnissen wichtige Teile extrahiert, indem auf dem Text eine Substring Analyse oder Linkanalyse durchgeführt wird. Watson generiert mehrere hundert Kandidaten, denn auch hier gilt Quantität vor Qualität. Falls in der Hypothesen Generierung keine korrekte Antwort gefunden wird, kann das System die ursprüngliche Frage nicht beantworten.

Hypothesis and Evidence Scoring - Bevor die Kandidaten genauer evaluiert werden und weitere Informationen zur Unterstützung der einzelnen Hypothesen gefunden werden, unterziehen sich diese einem *Soft Filtering*. Hier werden alle Kandidaten ressourcensparend vor untersucht und anhand trainierter Modelle bewertet. Alle die diesen Test nicht bestehen werden direkt an die *Final Merging and Ranking* Komponente weitergeleitet. Nachdem in etwa 100 Kandidaten den Soft Filter passiert haben, werden für diese weitere Nachweise gesucht, um diese daraufhin genauer zu bewerten. Eine mögliche Art zur Durchführung ist den jeweiligen Kandidaten mit in die Frage einzubinden und darauf eine erneute Suche laufen zu lassen. Weitere Verfahren und die genaue Bewertung können in [Ferrucci et al., 2010] nachgeschlagen werden.

Final Merging and Ranking - In dieser letzten Phase werden aus allen Hypothesen die am besten unterstützten herausgesucht. Dies erfolgt indem alle Bewertungen und Nachweise gesichtet und analysiert werden. Die finale Antwort des Systems wird mit einer *confidence score* übergeben, welche ein Maß dafür darstellt, dass dies die korrekte Antwort auf die Frage ist.

Der QA-Dienst von IBM Watson wurde im November 2015 beendet und von vier neuen Diensten ersetzt. Diese werden kurz aufgelistet und im Abschnitt Erweiterbarkeit wird genauer auf den Conversations Dienst eingegangen.

1. **Natural Language Classifier** ermöglicht natürliche Sprache mit Hilfe maschinellem lernen zu interpretieren und kurze Texte zu klassifizieren, nachdem das System mit Trainingsdaten in Form von repräsentativen Texten und vordefinieren Klassen betrieben wurde. Mit dem Speech-to-Text Dienst ist es z.B. möglich, Fragen der Kunden zu klassifizieren und an die richtige Abteilung weiterzuleiten [IBMe].

2. **Conversation** versteht natürliche Sprache und nutzt maschinelles Lernen, um Konversationen und einen Dialogfluss mit Nutzern zu erzeugen. Eine mögliche Einbindung dieses Dienstes ist in der Abbildung 7.2 dargestellt.

3. **Retrieve and Rank** kombiniert zwei Informationsgewinnungs Komponenten. Zum einen Apache Solr, eine Open Source Enterprise Suche und anderer-

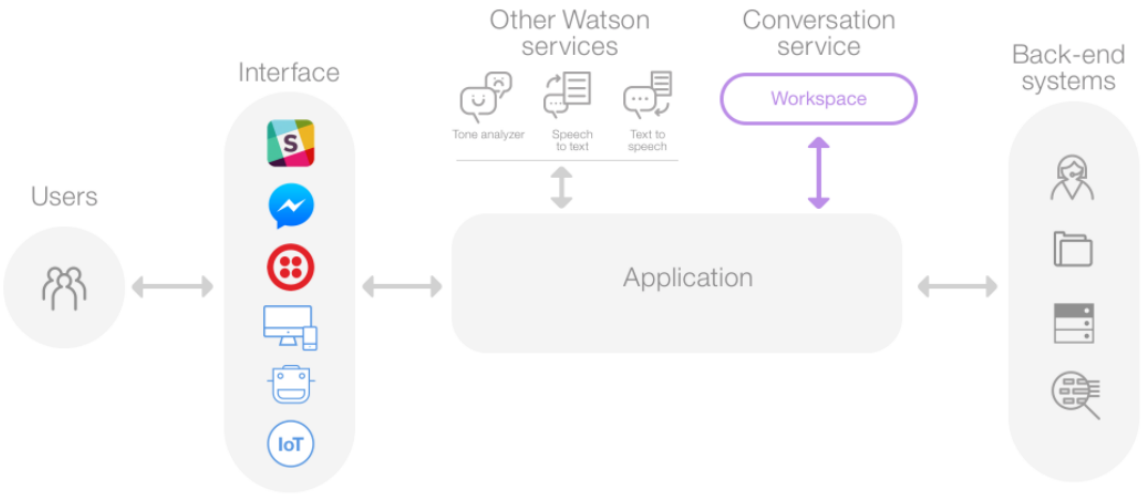


Abbildung 7.2: Architektur einer möglichen Einbindung der Conversation API [IBMb]

seits maschinelles Lernen, welches die beschafften Informationen anhand eines maschinell-lernenden Modells bezüglich ihrer Relevanz gewichten und dementsprechend bessere Vorschläge im Vergleich zu herkömmlichen Suchen anbieten können [IBMf].

4. **Document Conversion** konvertiert einzelne HTML,PDF oder Word Dokumente in HTML, Klartext oder einem Set an JSON-formattierten Antworten, die mit anderen Watson Diensten genutzt werden können [IBMd].

7.2 Erweiterbarkeit

Bis Ende 2015 wurde von IBM der Question and Answer (QA) Dienst angeboten, der Antworten auf gestellte Fragen lieferte. Dieser wurde jedoch eingestellt und durch die vier aufgezeigten Dienste ersetzt. Um einen funktionsfähigen Assistenten nachzubauen, der nur aus IBM Diensten besteht und Sprachein- und -ausgabe unterstützt, wird ein Zusammenschluss mehrerer Komponenten erforderlich. Zum einen wird der Speech-to-Text Dienst benötigt, in dessen Standard Plan die ersten Tausend Minuten kostenlos sind, den Text-to-Speech Dienst, der eine Million Zeichen pro Monat kostenlos in natürliche Sprache umwandelt und den Conversation Dienst, der bis zu 10.000 API

Aufrufe pro Monat ohne zusätzlichen Kosten anbietet und maximal 25 Intents und Entitäten unterstützt.
Desweiteren wird die Conversations API genauer untersucht und anhand eines Beispiels der Dialog Flow aufgezeigt. Auf die Anbindung der Conversations API mit einer Klient Applikation oder einem Bot durch Botkit wird am Ende kurz verwiesen.
Wie auch bei den bereits vorgestellten Sprachassistenten werden zunächst Intents und Entitäten zur Verarbeitung von natürlicher Sprache definiert und können über den Browser erstellt oder per CSV/JSON Datei importiert werden. In Listing 7.1, 7.2 ist ein Ausschnitt einer Intent und Entitäten Definition angeführt.

```
...
"intents": [{
"intent": "greetings",
"examples": [
  {"text": "aloha"},
  {"text": "bonjour"},
  {"text": "good morning"}
...
```

Listing 7.1: Codeausschnitt einer Intent Definition

```
"entities": [{
  "entity": "genre",
  "values": [{
    "value": "jazz",
    "synonyms": ["dixie", "swing"]
  },
  {
    "value": "pop",
    "synonyms": ["top 40", "top 10","popular"]
  }
```

Listing 7.2: Codeausschnitt einer Entitäten Definition

Da sich die Definitionen der Intents und Entitäten den anderen NLU Diensten ähneln, werden diese nicht weiter behandelt und stattdessen wird die Konstruktion einer Konversation aufgezeigt [IBMc]. Die verwendete Struktur ist ein Baum, bei dem ein einzelner Ast einem Intent entspricht und ein oder mehrere Kind-Knoten besitzt. Jeder dieser Knoten besitzt mindestens eine Bedingung und eine Antwort. Die Bedingung spezifiziert die Informationen, die nötig sind, damit dieser Knoten ausgelöst und ausgeführt wird. Dies kann ein spezieller Intent, ein Entitäten Wert oder eine Kontext Variable sein. Die Antwort kann als Text definiert werden, der dem Nutzer ausgegeben wird oder weitere Aktionen auslöst. Zudem können Variationen hinterlegt werden, um eine natürlich klingende Konversation zu ermöglichen. Falls ein Knoten erfolgreich ausgeführt wurde, kann entweder auf weiteren Input des Nutzers gewartet werden, dies wäre z.B. für Fragen mit einer Ja/Nein Antwortmöglichkeit wünschenswert oder zu einem anderen Knoten gesprungen werden. Falls zu einem anderen Knoten gesprungen wird, muss festgelegt werden, ob die Bedingung geprüft werden soll oder direkt eine seiner fest definierten Antworten ausgegeben werden sollen. Sollte die Bedingung nicht als wahr ausgewertet werden, wird der Knoten nicht ausgeführt und es werden die nächsten Geschwister Knoten überprüft. Sollte ebenfalls keiner dieser ausführbar sein, werden die Kinder der Wurzel überprüft.

Der Dialog selber enthält keinerlei Zustände und ist für jeden Nutzer der Appli-

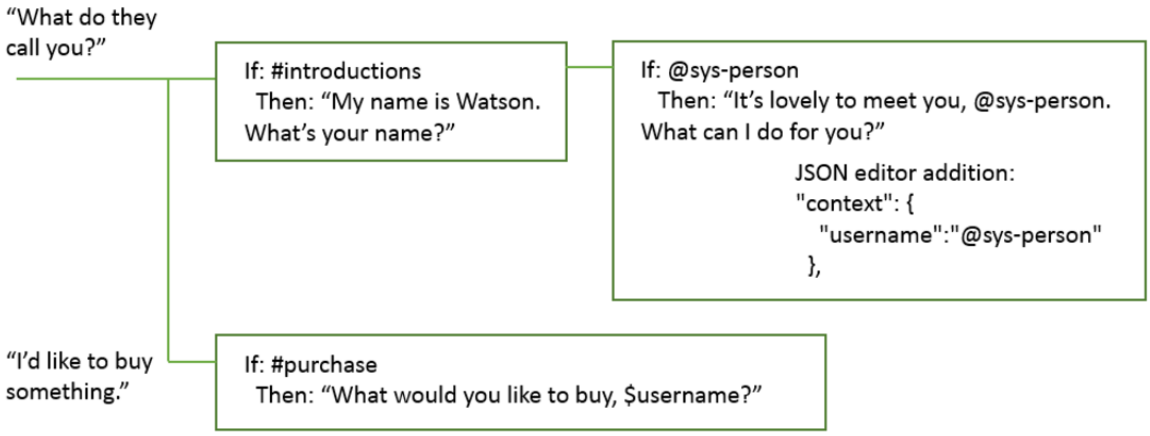


Abbildung 7.3: IBM Watson Dialog Flow [IBMc]

kation identisch. Dementsprechend ist die Klient Applikation dafür verantwortlich, kontextbezogene Informationen zu speichern. Diese Informationen können jedoch als Kontext Variablen dem Dialog übergeben, angepasst und zurück an die Applikation übermittelt werden. Diese Kontext Variablen können an Bedingungen geknüpft werden, um spezialisierte Antworten zu geben. Sie ermöglichen es zudem, den Kontext von einem Knoten zu einem weiteren Knoten zu übermitteln. Die Abbildung 7.3 zeigt, dass der Nutzername mit Hilfe der System Entität *@sys-person* extrahiert wurde und in der Variable *username* abgespeichert wurde, die im nächsten Knoten weiterverwendet werden kann, um den Nutzer anhand seines Namens personalisierter anzusprechen.

In den Kontext Variablen können Werte von Entitäten, ganze Eingabezeichenketten oder komplexere Objekte abgespeichert werden. Folgendes JSON Beispiel verdeutlicht dies:

```
1  {
2    "context": {
3      "place": "@ort"
4      "repeat": "<?input.text?>"
5      "complex_object": {
6        "vorname" : "Peter",
7        "nachname" : "Pan"
8        "alter" : "18"
9      }
10   }
11 }
```

Listing 7.3: Kontext Variablen Definition

Zur Einbindung der Conversation API bietet IBM das Watson SDK in den Sprachen Node.js, Java, Python, Swift, .NET und Unity an. Zur Erstellung der Applikation muss zunächst der Nutzername, das Passwort und die *Workspace ID* festgelegt werden. Einzelne Aktionen werden durch ein Vergleich der *response.output.action* mit den definierten Intents ausgeführt und ermöglichen es ohne großen Implementierungsauf-

wand eine Applikation zu erstellen. Eine Ausschnitt einer möglichen Implementierung ist in Listing 7.4 einzusehen.

```
1  // Set up Conversation service.
2  var conversation = new ConversationV1({
3    username: 'USERNAME',
4    password: 'PASSWORD',
5    path: { workspace_id: 'WORKSPACE_ID' },
6  });
7
8  // Conversation Antwort verarbeiten
9  function processResponse(err, response) {
10
11   // Intent überprüfen
12   if (response.output.action === 'uhrzeit_anzeigen') {
13     // Nutzer fragt nach der aktuellen Uhrzeit
14     console.log('Die aktuelle Uhrzeit ist ' + new
15     Date().toLocaleTimeString());
16   }
17 }
```

Listing 7.4: Codeausschnitt einer Watson Applikation in Node.js [IBMa]

7.3 Implementierbarkeit

IBM bietet mehrere SDKs zur Entwicklung von Applikationen für die bereitgestellten Dienste an. Zudem ist es möglich einen funktionsfähigen Assistenten mit Spracheinund -ausgabe, durch die STT und STT Dienste nachzubauen. Die einzigen Einschränkungen werden durch die verschiedenen kostenpflichtigen Pläne gesetzt. Eine Verwendung des TTS- oder des NLU Moduls der kostenfreien Variante wird in Kapitel 10 evaluiert.

Kapitel 8

Mycroft

In diesem Kapitel wird der Open Source Personal Assistant Mycroft vorgestellt, der unter der GNU Lesser General Public License version 3.0 veröffentlicht wird. Der digitale Assistent wurde nach einem Supercomputer in Robert A. Heinleins Roman *The Moon is a Harsh Mistress* benannt. Dieser Computer wurde mit leistungsstarker Hardware versehen und entwickelte im Laufe der Zeit ein eigenes Selbstbewusstsein. Durch eine Open Source Alternative zu den proprietären Systemen erhofft sich das Mycroft Entwicklerteam eine starke Basis an Drittentwicklern, die sowohl Mycrofts einzelne Module als auch Skills für Mycroft entwickeln und diesen dadurch schneller wachsen und in Zukunft eventuell den Turing Test bestehen lässt [Montgomeryb].

8.1 Prinzipien & System Architektur

Mycroft als *Intelligence Suite* besteht aus nachstehenden Komponenten:

- **Mycroft Core** verbindet NLU, TTS und STT Komponenten.
- **Mimic** ist eine schnelle und leichtgewichtige Text-to-Speech Engine.
- **Adapt** konvertiert natürliche Sprache in Datenstrukturen inkl. den extrahierten Intents.
- **OpenSTT** ist eine open source Speech-to-Text Engine, die hohe Genauigkeit und niedrige Latenz verbindet.

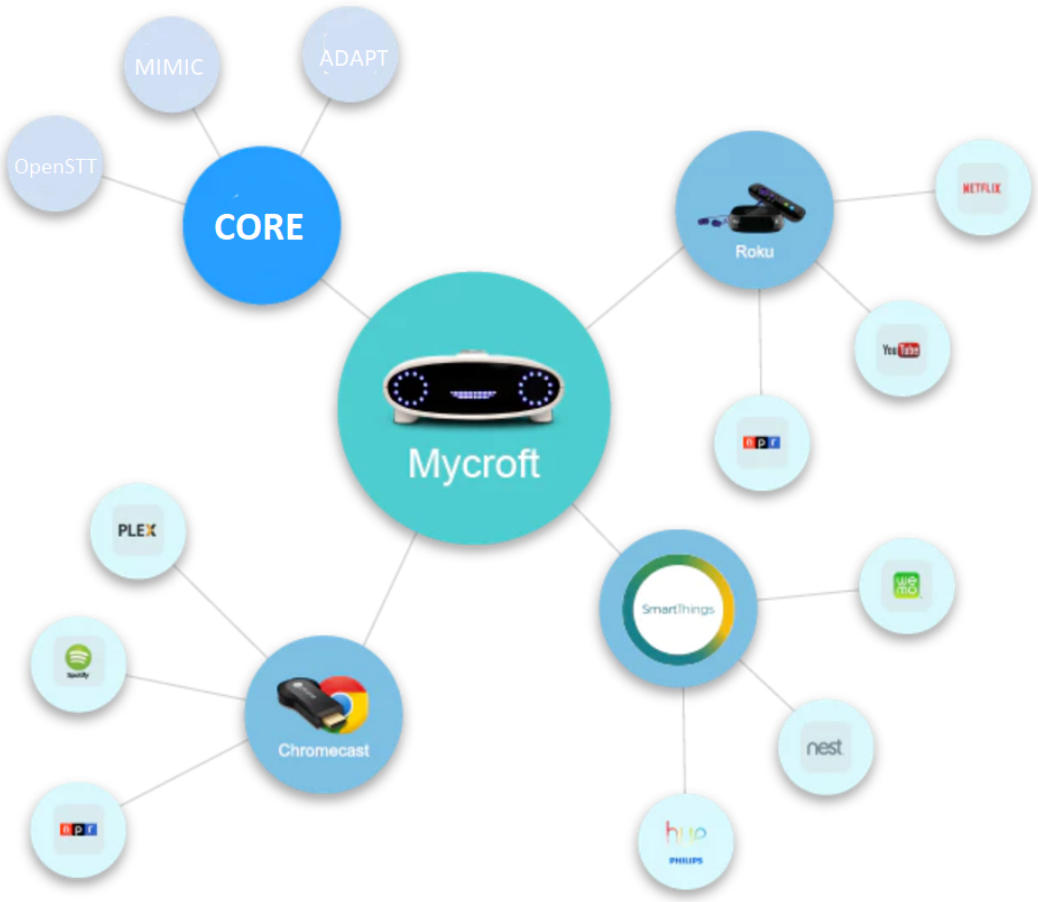


Abbildung 8.1: Mycroft Architektur und Fähigkeiten (verändert)[MONTGOMERYa]

Einige der in der Abbildung 8.1 dargestellten Fähigkeiten sind nicht *out of the box* verfügbar und müssen manuell nachgeladen werden.

Das Mycroft Core Modul beinhaltet hauptsächlich vier Dienste. Der Nachrichtenbus, der Skill-Dienst, der Sprachklient und die Kommandozeile *command line interface (CLI)*. Eine Anfrage an das Mycroft Framework sieht folgendermaßen aus [AIb]:

1. Der Nutzer sagt das Hotword (Hey Microft). Dieses kann nach eigenen Wünschen angepasst werden.

2. Der Sprachklient zeichnet die nachfolgende Sprache auf und schickt sie an das Mycroft Backend für eine STT Konvertierung.

3. Der STT Dienst schickt die Transkription zurück und der Sprachklient überträgt diese auf dem Nachrichtenbus.

4. Der Skill Klient greift die Transkription ab und nutzt das Adapt Modul um daraus einen Intent abzuleiten, der durch einen der Skills registriert ist.

5. Der Intent und etwaige Entitäten bzw. Keywords werden nach erfolgreichem *parsen* auf dem Nachrichtenbus weitergeleitet.

6. Der Skill Dienst leitet diese Daten an den Skill mit dem passenden definierten Intent Schema weiter.

7. Der entsprechende Skill verarbeitet die Daten, führt seine Aktionen aus und übergibt die Nachricht, die dem Nutzer ausgegeben werden soll, dem Nachrichtenbus.

8. Der Sprachklient erhält diese Nachricht und leitet sie an den TTS Dienst weiter (Mimic), welcher eine Audiodatei erstellt, die im Anschluss dem Nutzer ausgegeben wird.

Der **Nachrichtenbus** ist in dem Core Modul für sämtliche Interprozess Kommunikation verantwortlich. Z.B. könnte eine *speak* Nachricht übertragen werden und von jedem Dienst, der mit dem Bus verbunden ist, gelesen und verarbeitet werden.

Der **Sprachklient** besteht aus den Komponenten *AudioConsumer*, *AudioProducer* und einem *Listener*. Der Listener erkennt das Hotword durch Nutzung von Pocketsphinx, einer Spracherkennungs Engine und startet von diesem Zeitpunkt die Tonaufnahme bis das System sich dafür entscheidet, dass der Nutzer fertig ist mit sprechen. Zur Entscheidung darüber, wird die Heuristik verwendet: Der Listener nimmt solange den Ton auf, bis das *Noise level* unter null fällt. Dieses Level ist ein relativer Indikator für Umgebungsgeräusche und wird, bezüglich der letzten Sekunden permanent angepasst, wodurch Umgebungs- oder Störgeräusche die korrekte Länge einer Aufnahme nicht beeinflussen. Sollten nach der Aktivierung durch das Hotword

keine weiteren Geräusche folgen, wird eine Zeit von drei bis vier Sekunden abgewartet.

Der **Skill Dienst** besteht aus den aktivierten Skills, sowie dem Adapt Modul. Ein einzelner Skill besteht aus Vokabular Dateien (eine Liste von Phrasen oder Wörtern die erkannt werden sollen), Dialog Dateien (eine Liste aus Antworten die Mycroft als Antwort ausgeben soll) und einer Python Datei, welche zum einen das Vokabular auf die einzelnen Intents abbildet, als auch die jeweilig auszuführenden Aktionen definiert.

Der **Adapt Intent Parser** [AIa] überführt natürliche Sprache in eine Datenstruktur, die u.a. den Intent, die Wahrscheinlichkeit der Auswahl des korrekten Intents und eine Liste der Entitäten beinhaltet. Dieses Modul arbeitet regelbasiert, ist leichtgewichtig und kann lokal auf einem System (z.B. einem Rasberry Pi) laufen. Zur Erkennung der Intents aus der Anfrage verwendet das Mycroft Framework parallel zu diesem Modul noch maschinelles Lernen, um möglichst schnell genaue Ergebnisse zu liefern. Eine mögliche JSON Datenstruktur auf die Anfrage *Put on my Joan Jett Pandora Station* könnte so aussehen:

```
{
    "confidence": 0.61,
    "target": null,
    "Artist": "joan jett",
    "intent_type": "MusicIntent",
    "MusicVerb": "put on",
    "MusicKeyword": "pandora"
}
```

Listing 8.1: JSON Datenstruktur nach erfolgreichem Parsen einer Anfrage durch das Adapt Modul [AIa]

Details über das TTS Modul Mimic, welches auf der FLITE Software basiert und das STT Modul openSTT kann in [AId] und [AIe] nachgelesen werden.

8.2 Erweiterbarkeit

Bevor die genaue Implementierung zur Erstellung eines Skills analysiert wird, wird auf die Ordnerstruktur mit den notwendigen Dateien eingegangen [AIc]. Zuerst muss ein neuer Ordner mit dem Namen des Skills in /opt/mycroft/skills abgelegt werden. Dieser sollte einen Ordner dialog, eine Datei __init__.py, einen Ordner test und einen Ordner vocab beinhalten.

Der **dialog** Ordner muss für jede unterstützte Sprache eigene Unterordner mit den einzelnen Dialog Dateien enthalten. Z.B. müsste für die deutsche Sprache ein Unterordner mit Namen *de* erstellt werden, in dem ein Datei mit dem Namen *willkommen.dialog* existiert. Diese Willkommen Dialog Datei könnte eine Reihe von Aussagen enthalten, die in Listing 8.2 aufgeführt sind und durch Zeilenumbrüche separiert werden. Falls mehrere Antwortmöglichkeiten gegeben werden, wählt Mycroft einen dieser zufällig aus. Für jeden Intent muss eine spezifische Dialog Datei abgelegt werden.

```
1 Guten Tag, wie kann ich Ihnen Helfen.
2 Wie kann ich Ihnen helfen?
3 Schönen guten Tag, wobei kann ich Ihnen helfen.
```

Listing 8.2: Beispiel einer Dialog Datei [AIc]

Die **__init__.py** Datei erbt von der MycroftSkill Klasse und beinhaltet Funktionen, die den Skill auszeichnet.

Der **vocab** Ordner muss ebenfalls wie der Dialog Ordner für jede unterstützende Sprache einen Unterordner mit *.voc* Dateien besitzen, die eine Liste von Wörtern und Phrasen enthalten, um den jeweiligen Intent auszulösen.

Der generelle Aufbau der init Python Datei sieht wie folgt aus und wurde ebenfalls aus der Mycroft Documentation entnommen [AIc]:

```
1 class HelloWorldSkill(MycroftSkill):
2     def __init__(self):
3         super(HelloWorldSkill, self).__init__(name="HelloWorldSkill
```

```
    ")
4
5       def initialize(self):
6       thank_you_intent = IntentBuilder("ThankYouIntent").\
7           require("ThankYouKeyword").build()
8         self.register_intent(thank_you_intent, self.
    handle_thank_you_intent)
9         ...
10
11      def handle_thank_you_intent(self, message):
12        self.speak_dialog("welcome")
13      ...
14      def stop(self)
15        pass
16
17  def create_skill():
18      return HelloWorldSkill()
```

Listing 8.3: Codeausschnitt eines Mycroft Skills [AIc]

8.3 Implementierbarkeit

Das open source Mycroft Framework kann im Gegensatz zu den bisher vorgestellten Sprachassistenten gänzlich nach eigenen Wünschen angepasst werden und besitzt dementsprechend keine Limitierung in der Anzahl der Intents und Entitäten. Die Genauigkeit und Antwortzeit der entwickelten und verwendeten Komponenten (Adapter Intent Parser, STT und TTS Modul) muss jedoch genauer untersucht werden. Das Mycroft Core Modul wurde in Python geschrieben und kann plattformübergreifend verwendet werden. Mycroft AI, Inc. bietet ein Image für Linuxbasierte Desktop Systeme und das Raspberry Pi an. Zudem kann das Endgerät Mycroft Mark 1 mit vorinstallierter Software erworben werden. Leider sind ein Großteil der Drittentwickler Skills derzeit nur in englischer Sprache verfügbar und schränken deshalb den

Funktionsumfang in Deutsch stark ein. Diese müssen zudem manuell nachgeladen werden und funktionieren nur teilweise *out of the box*.

Für die Implementierung eines Sprachassistenten in einen mobilen Roboter können einzelne Module dieses Open-Source Projektes in Betracht gezogen werden, da es eine hohe Interoperabilität mit anderen Diensten (TTS oder STT Komponenten) bietet.

Kapitel 9

Vergleich der vorgestellten Sprachassistenten

In der nachstehenden Tabelle werden die bisher analysierten Sprachassistenten untereinander bezüglich einiger Gütekriterien untersucht und verglichen. Dabei werden jegliche Produkte und Geräte auf denen einer dieser Assistenten vertreten ist zusammengefasst. Beispielsweise werden bei dem Google Assistant sowohl Android Smartphones, Android TV's, Settopboxen und Google Home akzeptiert und deren Features vereinigt. Weiterhin sei angemerkt, dass viele der Mycroft Funktionalitäten auf Drittanbieter Anwendungen beruhen und deren Korrektheit nicht geprüft wurde. Außerdem ist IBM Watson in dem anschließenden Vergleich nicht einbezogen worden. Dies hat den Grund, dass diese Kriterien nicht auf Watson angewendet werden können.

Tabelle 9.1: Vergleich der vorgestellten Sprachassistenten bezüglich des Funktionsumfangs

	Google Assistant	**Apple Siri**	**Microsoft Cortana**	**Bixby**	**Mycroft**
Jederzeit zuhören	ja	ja	ja	ja	ja
Hotword	Ok Google, Hey Google	Hey Siri	Hey Cortana	Hey Bixby	Hey Mycroft (anpassbar)
Erweiterbarkeit durch Skills	ja	ja (eingeschränkt)	ja	nein (SDK geplant)	ja
Sprache zur Erstellung von Skills	java, kotlin, xml	swift, objective C, xml	C#, Node.js, xml	-	Python
Open Source	nein	nein	nein	nein	ja
Musik Streaming Optionen	Google Play Music, YouTube Music, Spotify, Pandora, TuneIn	Apple Music (weitere nicht bekannt)	Groove Musik, (en-us: TuneIn, iHeartRadio; (für Invoke geplant: Spotify, Pandora)).	Google Play Music (weitere nicht bekannt)	(3rd party Libraries) Google Music, Spotify, Mopidy, Lokale Musik, Quod Libet
Smart Home Partnerschaft	Nest, Honeywell, SmartThings, Wink, Belkin WeMo, Philips Hue, Lifx, Lutron, August, Logitech Harmony, Anova, IFTTT und weitere	HomeKit kompatible Produkte, u.a. Philips Hue, Nanoleaf, Lutron, Insteon, iHome, iDevices, Elegato, First Alert, Honeywell und weitere	Derzeit sind keine bekannt, mit Windows 10 IoT sollen Smart Home Geräte entwickelt werden. (z.B. Johnson Controls 'GLAS')	Smartthings kompatible Produkte, u.a. Philips Hue, Lutron, Leviton, Enerwave, Aeotec, Netgear, Honeywell, Ecolink, Bose und weitere	SmartThings, IFTTT, Belkin WeMo, Philips Hue, Nest, Lowe's Iris und Wink Hub
Multimediaausgabe Streaming (z.B. Audio Ausgabe)	ja, durch Chromecast	ja, durch AirPlay	nein, (*Cortana Speaker* geplant)	(nicht bekannt)	ja, an Chromcast, AirPlay Geräte (3rd Party Skill)
Synchronisierte ausgabe von Musik an mehrere Geräte	ja an jedes Chromecast Gerät	ja, mit Airplay2 Geräten	nein	(nicht bekannt)	(nicht bekannt)

Kapitel 10

Konzeption zur Implementierung eines Sprachassistenten in einen mobilen Roboter

Wie schon in vorangegangenen Kapiteln deutlich wird, müssen viele Komponenten verknüpft werden, um einen Sprachassistenten nachzubauen bzw. durch vorhandene APIs zu erweitern und nach eigenen Bedürfnissen anzupassen. Darunter fallen:

- Hotword Detektion
- Spracherkennungsmodul
- STT
- Natural Language Understanding
- Intent Verarbeitung
- TTS

Eine Konversation mit dem Roboter läuft auf diese Art und Weise ab:

Der Nutzer spricht die Phrase *Hey Mira* und triggert dadurch das Hotword des Detektionsmoduls, welches lokal auf dem Roboter ausgeführt wird und daraufhin das

Spracherkennungsmodul startet. Der Nutzer kann nach einem nachfolgenden Ton, welcher den Nutzer informiert das der Assistent zuhört, ein Kommando geben.

Das Spracherkennungsmodul nimmt solange die Sprache des Nutzers auf und speichert diese in einer Audiodatei ab, bis die aktuelle *Noise* Schwelle unter einen kritischen Bereich fällt, der das Ende seiner Aussage zu vermuten lässt. Diese Audiodatei wird daraufhin über das Speech-to-Text Modul in Text umgewandelt und an das NLU Modul weitergeleitet, in dem die Absicht des Nutzers (Intent) und dessen übergebene Parameter entnommen und in ein geeignetes Dateiformat umgewandelt werden. Der entsprechende Intent wird daraufhin lokal auf dem Roboter angesprochen, führt seine vordefinierte Aktion aus (z.B. Ausgabe der Uhrzeit) und generiert gegebenenfalls eine Antwort über das TTS Modul. Ein schematischer Ablauf ist in der Abbildung 10.1 zu sehen.

Für die **Hotword Detektion** wird Snowboy [KITT.AI] verwendet, welches als C++ Bibliothek geschrieben ist. Die entsprechenden Python Wrapper werden durch *SWIG* (Simplified Wrapper and Interface Generator) generiert. Alternative Python Bibliotheken zur Detektion (hotword_detection 1.2, PocketSphinx) werden jedoch nicht weiter getestet.

Das **Spracherkennungsmodul** verwendet die SpeechRecognition Bibliothek [Zhang], welche Unterstützung für diese APIs mitbringt:

- CMU Sphinx
- Google Speech Recognition
- Google Cloud Speech API
- Wit.ai
- Microsoft Bing Voice Recognition
- Houndify API

- IBM Speech to Text

Im hier angeführten Fall wird die Google Speech Recognition zur Umwandlung der gesprochenen Sprache in Text genutzt.

Für das **NLU Modul** stehen u.a. die LUIS.ai-, API.ai- und IBM Watson Conversation Lösungen zur Verfügung. Im Gegensatz zur Watson Conversation, welche Intents und Entitäten auf maximal 25 limitiert, bietet API.AI eine unbegrenzte Anzahl dieser an. Von einem Softlimit von 2000 Intents wird jedoch ausgegangen. Ferner stellt API.AI die System Entität *@sys.any* zur Verfügung, welche einen vorher nicht definierten Parameter aus dem Satz extrahiert. Dadurch ist es u.a. möglich den Satz *Suche im Internet nach Blockchain* zu bilden und das entsprechende Fremdwort herauszulösen. Der entsprechende Intent wird nach folgenden Muster erstellt, wobei Blockchain als *@sys.any* Entität deklariert wird:

Tabelle 10.1: Intent Aufbau (API.AI)

Suche im Internet nach Blockchain		
Parameter Name	Entity	Resolved Value
fremdword	@sys.any	Blockchain

Die **Intent Verarbeitung** verwertet den herausgelösten Intent, die übergebenen Parameter und führt entsprechende Aktionen aus. Dieses Python Modul kann nach belieben angepasst und erweitert werden.

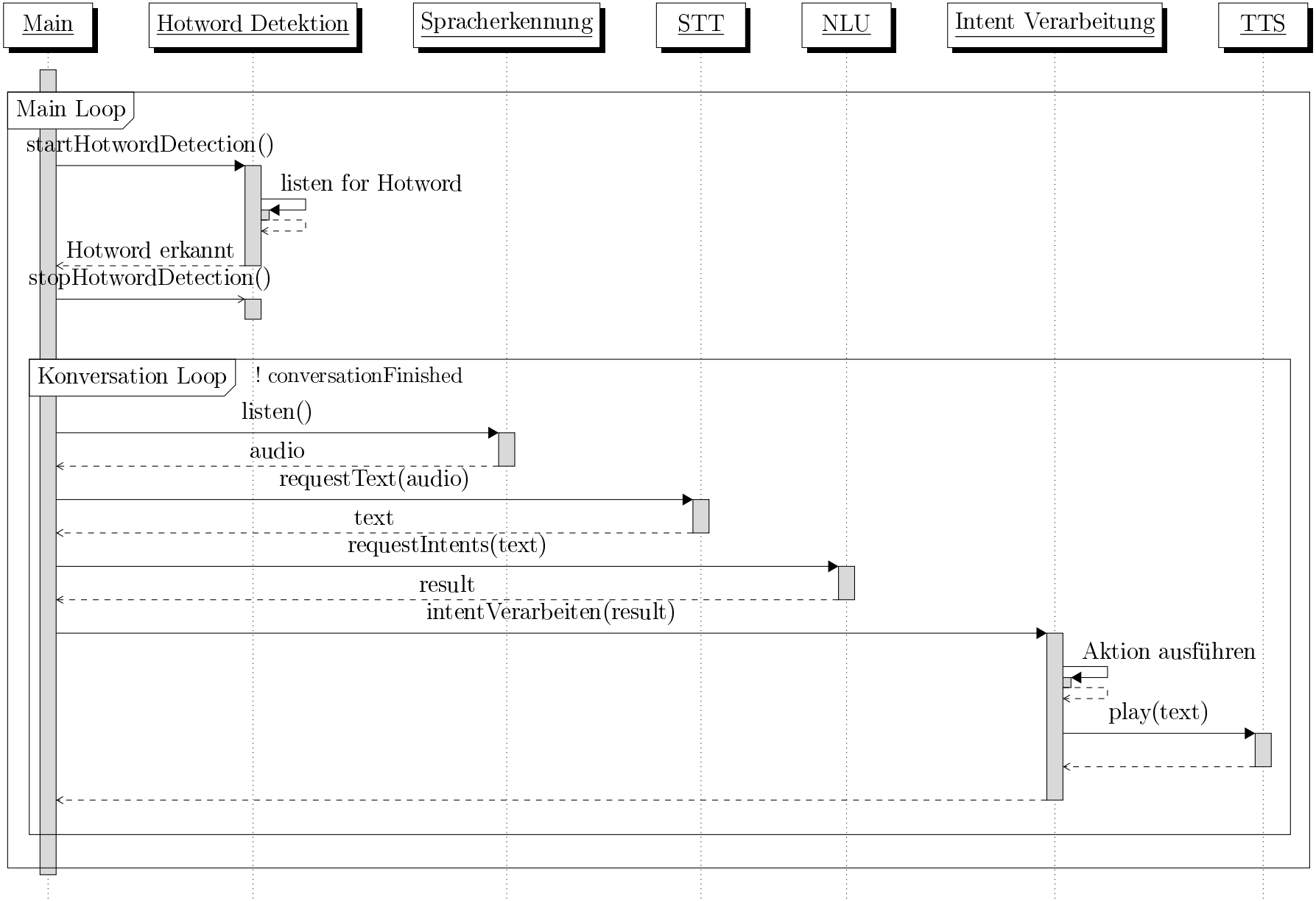
Als Python Modul für **TTS** stehen u.a. Amazon Polly (ehemalig IVONA Speech Cloud Beta), IBM Watson, Acapela und Google Text to Speech (gTTS) zur Verfügung. Acapela stellt dabei den teuersten Dienst dar mit circa € 1500 für 200.000 Sekunden transkribierter Sprache. IBM Watson TTS bietet eine kostenlose Transkribierung für eine Millionen Zeichen pro Monat und AWS Polly (in den ersten 12 Monaten) für fünf Millionen Zeichen pro Monat. Aufgrund der leichten Implementierung und

der Restriktionsfreiheit entschied man sich für Google TTS. Da alle Komponenten modular aufgebaut sind, ist es jederzeit möglich eine andere TTS API einzubinden.

Angesichts der Tatsache, dass das Mira Framework sowie viele der notwendigen APIs eine Python Schnittstelle bieten, wurde diese Sprache zur Erstellung des Sprachassistenten und zur Interkommunikation der einzelnen Module gewählt.

Ein sequentieller Ablauf einer Konversation inklusive der Module ist in der Abbildung 10.1 dargestellt. Im nächsten Kapitel wird die konkrete Umsetzung der getroffenen Design Entscheidungen aufgezeigt.

Abbildung 10.1: Sequenzieller Ablauf einer Konversation



Kapitel 11

Implementierung in einen mobilen Roboter

Die notwendigen Schritte zur Implementierung eines Sprachassistenten in einen mobilen Roboter auf Linux-Basis sind Bestandteil dieses Kapitels. Zuerst werden die notwendigen Voraussetzungen aufgelistet und die Python Paket-Struktur des Mira Assistenten aufgezeigt. Darüber hinaus ist eine vollständige Instruktion zur Installation aller Abhängigkeiten im Anhang in Listing A.6 gegeben.

11.1 Voraussetzungen

- System mit einer Mainstream Linux-Distribution
- gTTS: Python Interface für Googles Text-to-Speech API
- Pygame: Multimedia Bibliothek, notwendig zur Ausgabe von Audioströmen
- SpeechRecognition: Spracherkennungsbibliothek mit Unterstützung für mehrere Speech APIs
- PyAudio: Python Bindings für PortAudio
- DuckDuckGo2: Python Bibliothek für die DuckDuckGo API

- API.AI: Python SDK
- Python Wrapper für SoX
- SWIG: Simplified Wrapper and Interface Generator
- ATLAS: Automatische Generierung und Optimierung von numerischer Software

Die Python Paket Struktur der entwickelten Komponenten ist in der Abbildung 11.1 zu erkennen. Die Python Datei mira.py fungiert als Kernelement, steuert den Prozess des Sprachassistenten und greift auf die einzelnen Module zu.

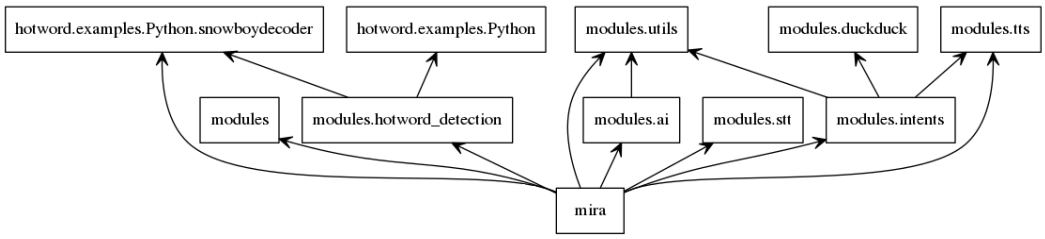


Abbildung 11.1: Mira Assistent - Python Paket Struktur

11.2 Module

Nachstehend werden die einzelnen Module des Mira Assistenten vorgestellt und einzelne Codeabschnitte zum leichteren Verständnis aufgeführt. Näheres dazu im Kapitel 11.3.

11.2.1 Snowboy Hotword Detection

Die Snowboy Hotword Erkennung ist eine Open Source Bibliothek, die lokal auf dem System ohne Internet Verbindung läuft, jederzeit zuhört und auf ein Hotword wartet. Wird das Wort oder die Wortfolge erkannt folgen weitere Aktionen.

Zur Erstellung eines eigenen Sprachassistenten wurde zunächst ein eigenes Sprachmodell erstellt, dass auf die Wortfolge *Hey Mira* reagiert. Dies ist möglich über

die grafische Oberfläche oder über einen RESTful API Aufruf. Zur Erhöhung der Genauigkeit der Hotword-Erkennung ist es möglich, das Sprachmodell mit bis zu 2000 Personen zu trainieren. Zudem kann die Sensitivität der Erkennung ebenfalls angepasst werden, wie Listing 11.1 zeigt.

```
1 detector = snowboydecoder.HotwordDetector(model, sensitivity=0.5)
```

Listing 11.1: SnowboyDeoder Initialisierung mit dem Sprachmodell und der Sensitivitätseinstellung

Dem Hotword Listener muss eine Callback Funktion mitgegeben werden, die ausgeführt wird, wenn das vorher definierte Sprachmodell erkannt wurde. Der Listener wird mit der Funktion gestartet:

```
1 def startHotwordListener(callback):
2   print('HotwordDetection läuft...')
3   detector.start(detected_callback=callback,
4              interrupt_check=interrupt_callback,
5              sleep_time=0.03)
```

Listing 11.2: Snowboy startHotwordListener Funktion mit übergebenen Callback Parameter

Nachdem die Bibliothek eingebunden wurde, waren nur marginale Änderungen an der Datei *snowboydecoder.py* nötig, um starten und stoppen des Hotword Listeners zu ermöglichen.

11.2.2 SpeechRecognition (STT)

Wie schon erwähnt unterstützt die SpeechRecognition Bibliothek eine Vielzahl von STT APIs und ermöglicht es dadurch ebenfalls auf eine andere STT Engine umzusteigen, falls dies erwünscht sein sollte. Die Aufnahme der Sprache und die Ermittlung des äquivalenten Textes findet durch die Funktion *listen()* statt:

```
1  def listen():
2    r = sr.Recognizer()
3    with sr.Microphone() as source:
4      print("Kommando geben!")
5      audio = r.listen(source)
6
7    # STT durch Google Speech Recognition
8    try:
9      text = r.recognize_google(audio, language='de')
10     print("Google Speech Recognition erkannte " + text)
11     return text
12 ...
```

Listing 11.3: Aufnahme und Umwandlung der Sprache in Text *(aus modules/stt.py)*

Weitere Optimierungen können u.a. durch diese Parameter gesetzt werden.

```
1
2 # Je höher die Sensitivität des Mikrophones ist,
3 # oder je lauter der Raum ist desto höher kann dieser
4 # Wert eingestellt werden.
5 recognizer_instance.energy_threshold = 300
6
7 # Kann genutzt werden um den energy_threshold an
8 # Hintergrund Geräusche anzupassen.
9 # Sollte nur genutzt werden, falls nicht gesprochen wird
10 recognizer_instance.adjust_for_ambient_noise(source, duration = 1)
11
12 # Aufnahme im Hintergrund
13 # und nach Abschluss Callback ausführen.
14 recognizer_instance.listen_in_background(source, callback)
```

Listing 11.4: Anpassungsmöglichkeiten der SpeechRecognition

Zudem kann mit Hilfe des *phrase_time_limit* Parameters eine maximale Länge einer Anfrage in Sekunden definiert werden. Dies kann zu einem vorläufigen Abbruch der Aufnahme des Kommandos führen, verhindert jedoch, dass eine Aufnahme kontinuierlich ausgeführt wird, falls angenommen wird, dass der Nutzer noch spricht.

11.2.3 API.AI (NLU)

Auf API.AI als NLU Komponente wurde schon ausführlich im Kapitel 3.2 eingegangen, deshalb wird im Folgenden nur auf den Verbindungsaufbau der Klient Anwendung eingegangen. Die Verarbeitung der Antwort, die im JSON Format vorliegt, findet im nachfolgenden Modul statt. Die erstellten Intents auf API.AI können aus der Tabelle 11.1 entnommen werden.

```
1  def request(text):
2
3    ai = apiai.ApiAI(CLIENT_ACCESS_TOKEN)
4    request = ai.text_request()
5
6    #Sprache Deutsch
7    request.lang = 'de'
8
9    #Falls SessionId existiert --> übergeben!
10   if utils.sessionId is not None:
11     request.session_id = utils.sessionId
12
13   request.query = text
14
15   #Antwort erhalten
16   response = request.getresponse()
17 ...
```

Listing 11.5: API.AI SDK - Intent und Parameter Analyse Anfrage auf einem vorliegenden Text

Die JSON Antwort wird daraufhin in ein Python Dictionary übertragen und dessen Variablen, die u.a. den erkannten Intent, die auszugebende Sprache und weitere Parameter enthalten, können im nächsten Modul weiterverarbeitet werden.

11.2.4 Intent Verarbeitung

Nachdem eine Antwort des API.AI Moduls geliefert wurde, kann der angesprochene Intent mit der entsprechenden Funktion verknüpft werden. Im nachfolgenden Beispiel könnte der Nutzer den Assistenten nach der aktuellen Uhrzeit gefragt haben. Der Intent **time** wurde seiner Phrase entnommen und die Funktion **timenow** kann ausgeführt werden.

```
1
2 #Führe den entsprechenden Intent aus
3 def executeIntent(speech, intent, result):
4
5   #Intents verknüpfen und ausführen
6   if intent == 'time':
7     timenow(speech, result)
8 ...
```

Listing 11.6: Verknüpfung der einzelnen Funktionen mit dem entsprechenden Intent *(aus modules/intents.py)*

Auf diese Weise kann jedem erkannten Intent lokal eine entsprechende Funktion zugewiesen werden. In diesen Funktionen können simple Sprachausgaben erfolgen oder dem Roboter Anweisungen gegeben werden.

```
1  #INTENT TIMENOW
2  def timenow(speech, result):
3
4    #Derzeitige Uhrzeit bestimmen
5    now = datetime.datetime.now()
6    hours = now.strftime("%H").lstrip('0')
7    minutes = now.strftime("%M").lstrip('0')
8
9    tts.voice('Es ist ' + hours + ' Uhr ' + minutes + '.')
10   tts.play(True)
11   utils.endConversation = True
```

Listing 11.7: Funktion timenow, die die aktuelle Uhrzeit bestimmt und dem Nutzer sprachlich ausgibt. *(aus modules/intents.py)*

11.2.5 gTTS

Als abschließenden Schritt eines Konversationszyklus wird die Antwort dem Nutzer sprachlich ausgegeben. Aus der Funktion timenow in Listing 11.7 wird auf das TTS Modul in Zeile neun & zehn zugegriffen.

```
1  #Ausgabe der mp3 Datei mit Threadblock option
2  #Verhindert weitere ausführung weiteren Codes
3  #bis die Sprachausgabe vollständig abgeschlossen wurde
4  def play(threadblock):
5    pygame.mixer.music.load(play_name)
6    pygame.mixer.music.play()
7    if threadblock:
8      while pygame.mixer.music.get_busy() == True:
9            continue
10
11 #Erstelle mp3 Datei aus Text (google TTS)
12 def voice(x):
```

```
13      text = gTTS(text=x, lang='de')
14      with open(save_name, 'wb') as tmp_file:
15          text.write_to_fp(tmp_file)
16      os.rename(save_name, play_name)
```

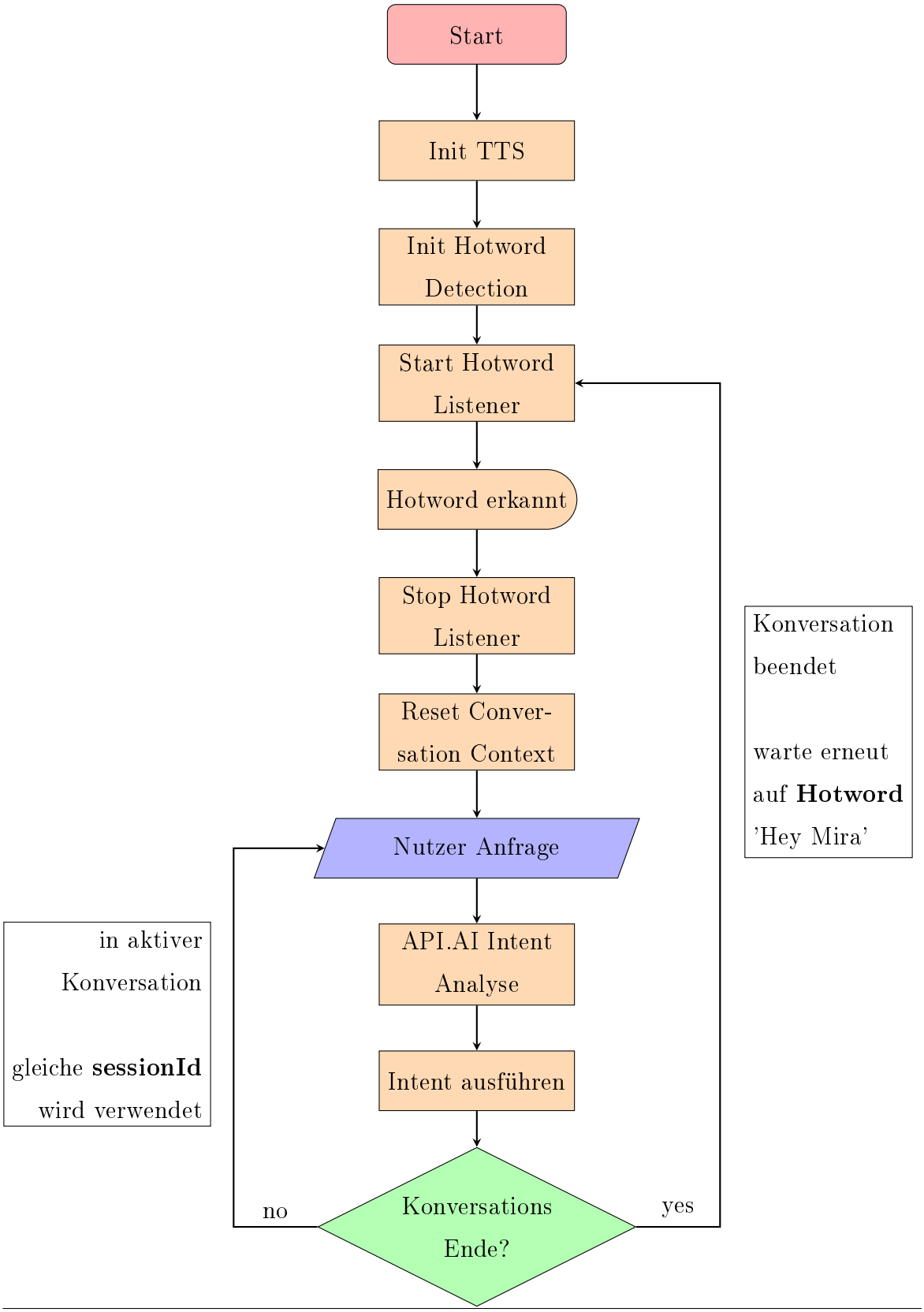
Listing 11.8: Erstellung und sprachliche Ausgabe einer, aus einem text generierten mp3. *(aus modules/tts.py)*

Auch hier ist es möglich, dass die Ausgabe den Thread nicht blockiert, wodurch weitere Anwendungsfälle möglich wären. Z.B. könnte die aktuelle Sprachausgabe durch weitere Kommandos unterbrochen werden. Eine Unterbrechung der Ausgabe ist derzeit nicht möglich und würde umfangreiche Maßnahmen betreffen, weil der ausgegebene Audiostrom zu viel Interferenz für die Spracherkennung erzeugt. Eine Möglichkeit könnte darin bestehen, die aktuelle Ausgabe aus dem Input des Mikrophones herauszurechnen und dadurch nur die Eingabe des Nutzers isoliert weiterzuverarbeiten.

11.3 Programmablauf

Der genaue Programmablaufplan ist in der Abbildung 11.2 gegeben. Der Assistent initialisiert zuerst das TTS Modul, die Hotword Detektion und startet daraufhin den Hotword Listener. Wenn die Hotword Detection die Wörter *Hey Mira* erkennt, führt es den vom Hauptprogramm übergebenen Callback *startConversation()* aus. Da der Assistent nun in eine aktive Konversation wechselt, wird der Hotword Listener gestoppt und jeglicher Kontext, aus vorherigen Konversationen, wird gelöscht. Nach diesem Zeitpunkt ertönt ein Signal, dass den Nutzer darüber informiert, dass dieser seine Anfrage dem Assistenten übermitteln kann. Wurde die Anfrage korrekt aufgenommen, wird der Inhalt durch API.AI analysiert und die entsprechenden Intents und Parameter extrahiert und dem Assistenten übergeben. Der Intent wird in dem Modul intents.py aufgerufen und die dazugehörige Funktion ausgeführt. Falls dieser Intent keine weitere Konversation benötigt, wird die Flag *conversationEnd* gesetzt.
Das Hauptprogramm beendet damit die aktive Konversation, springt aus der Schleife heraus und startet den Hotword Listener erneut, womit der Kreislauf erneut startet.

Abbildung 11.2: Mira Assistent - Programmablaufplan



Während der Assistent in der gleichen Konversation verweilt, wird die gleiche sessionId verwendet, um auf den aktiven Kontext zuzugreifen. Folgendes Beispiel legt offen, weshalb dies von Nutzen ist.

In Tabelle 11.1 ist der Intent **Timer** aufgelistet, welcher als Parameter **@number** und **@time_units** benötigt. Gibt der Nutzer nun den Befehl:

(Nutzer) : *Stelle Timer*

benötigt der Assistent weiterhin die Parameter **@number** und **@time_units**. Deshalb wird die Antwort wie folgt generiert:

(Assistent) : *Wie lange soll ich den Timer stellen?*
(Nutzer) : *30 Minuten*

Der Nutzer liefert die restlichen Parameter und der Assistent ordnet diese der aktuellen Konversation zu und stellt einen Timer auf 30 Minuten, bzw. liefert dem Klienten alle notwendigen Informationen. Nachdem die aktuelle Konversation beendet ist, startet der Assistent den Hotword Listener erneut und wartet auf das nächste Hotword. Eine Tabelle mit allen derzeit implementierten Intents ist in der Tabelle 11.1 angeführt. Zudem ist in Listing A.7 ein Codeausschnitt gegeben, der optionale Parameter verdeutlicht. Der Intent **move** erwartet als optionalen Parameter den Raum (@room). Falls dieser nicht übergeben wurde oder nicht einer der vordefinierten Räume entspricht, wird der Standard Intent ohne Parameter ausgelöst und bezweckt, dass der Roboter der Person aus dem Weg geht. Dies kann natürlich nach eigenen Wünschen angepasst werden, sodass eine andere Handlung in Kraft tritt (z.B. könnte dieser auf seinen Stellplatz fahren).

11.4 Vergleich mit proprietären Sprachassistenten

Nachdem nun alle Komponenten des Mira Assistenten sowie dessen Programmablauf anschaulich dargestellt wurden, stellt sich die Frage inwiefern sich dieser Assistent von kommerziellen bzw. proprietären Sprachassistenten unterscheidet.

Wie bei Open Source Projekten steht auch bei dem Mira Sprachassistenten die Flexibilität und Erweiterbarkeit der einzelnen Module im Vordergrund. Es bietet die Möglichkeit einzelne Komponenten leicht auszutauschen und nach eigenen Bedürfnissen anzupassen. Viele aktuelle Sprachassistenten liefern ein Komplettpaket, welches Hotword Detektion, Spracherkennung, -verarbeitung und -synthese bereitstellt. Diese Systeme sind jedoch sehr restriktiv und erlauben zumeist nur die Nutzung des eigenen Hotwords (z.B. OK Google). Zudem können systemeigene Intents nicht überschrieben werden und eigene Skills nur umständlich über eine zusätzliche Phrase angesprochen werden (siehe Abbildung 3.1). Eine Anfrage bezüglich eines Mira Skills im Google Assistant könnte demnach so aussehen:

(Nutzer) : *Ok Google, frage Mira: wie ist dein aktueller Akkustand*

Es wird deutlich, dass der Nutzer eine zusätzliche Phrase verwenden muss, um die Frage nach dem aktuellen Batteriestand zu erhalten.

Abgesehen davon, sind solche Systeme meist weitaus besser bezüglich ihrer eigenen Komponenten angepasst, als ein Zusammenschluss verschiedener Services von womöglich verschiedenen Unternehmen. Deshalb wird die Zeit von einer Anfrage bis zu einer Antwort, verglichen mit einer der proprietär vorgestellten Assistenten, im Mira Sprachassistenten höher ausfallen. Weiterhin müssen alle Aktionen, die einer der Sprachassistenten von Haus aus mitbringt selber implementiert werden. Dies könnte z.B. eine Wettersuche oder eine Anbindung an einen Kalender sein. Demnach ist offensichtlich, dass der Mira Assistent noch nicht den gleichen Umfang an Intents unterstützt. Eine Erweiterung des Assistenten mit weiteren Intents ist jedoch leicht

umzusetzen und wurde im vorherigen Kapitel besprochen.

Abschließend lässt sich sagen, dass bei dem Mira Assistent die meisten System Features der proprietären Sprachassistenten umgesetzt wurden. Der Assistent hört auf ein Hotword, erwartet ein Sprachkommando, verarbeitet dieses und führt eine entsprechend definierte Aktion aus.

Tabelle 11.1: Intent Definitionen

Intent Name	Aussage (Intent)	Parameter		Aktion	Konversation Ende
	(u.a.)	benötigt	optional	(u.a.)	
battery	Wie ist dein Akkustand?	-	-	Batteriestand (derzeit Zufallszahl) wird ausgeben.	Ja
calendar	Welcher ist mein nächster Termin?	-	-	Ausgabe: Ich kenne deine Termine noch nicht.	Ja
date	Welcher Tag ist heute?	-	-	Ausgabe : Heute ist der [Date]	Ja
follow	Folge mir!	-	-	Ausgabe: Alles klar.	Ja
mood	Wie geht es dir?	-	-	Mir geht es gut, danke. Was kann ich für dich tun?	Nein
move	Fahre in die Küche!	-	@room	Ich fahre in die [@room]	Ja
news	Erzähl mir die Nachrichten!	-	-	Tagesschau sagt: [news]	Ja
time	Wie viel Uhr ist es?	-	-	Es ist [hours] Uhr [minutes].	Ja
timer	Stell Timer auf 30 Minuten!	@number @time_units	-	Timer wird auf [hours] Uhr [minutes] gestellt.	Ja
wikipedia_search	Was ist Blockchain?	@fremdwort	-	Wikipedia sagt: [duckduck_response]	Ja / Nein
yes_intent	Ja	@yes_units	-	Okay was kann ich für dich tun?	Nein
stop_listening	Stopp	@stop_units	-	Falls Sie weitere Fragen haben, rufen Sie mich.	Ja
welcome	Hallo	-	-	Hallo, wie kann ich Ihnen helfen?	Nein
fallback	-	-	-	Ich habe Ihre Frage nicht verstanden.	Nein

Kapitel 12

Evaluierung

In diesem Kapitel wird eine Evaluation der Ergebnisse aus dem Kapitel 11 vorgenommen.
Der Sprachassistent und seine Komponenten werden in einer VirtualBox auf einem 64bit Ubuntu (16.04.3 LTS) getestet. Die einzelnen System Spezifikationen lauten wie folgt:

- Intel Core i5-4300U, Intel HD Graphics 4400, Hynix HFS128G3MNM
- 1933 MiB Arbeitsspeicher
- Dual-microphone mit Noise Cancellation

Zuerst wurde getestet, ob alle notwendigen Abhängigkeiten in Listing A.6 aufgeführt wurden und alle Python Module von dem Compiler korrekt übersetzt werden. Nachdem alle notwendigen Abhängigkeiten erfolgreich installiert wurden und sich der Sprachassistent durch den Befehl *python mira.py* erfolgreich starten lässt, wird zunächst die Hotword Erkennung genauer evaluiert.

Dabei wurde das Hotword *Hey Mira* als erstes in einer leisen Umgebung mit circa 10db getestet und später in einer Umgebung mit Hintergrundmusik.

Es wird deutlich, dass die Erkennung auf große Distanz und mit lauten Hintergrundgeräuschen sehr solide funktioniert. Dabei ist anzumerken, dass die eigene

Tabelle 12.1: Messung der Hotword Erkennungsrate bezüglich Distanz und Hintergrundlautstärke

Umgebungslautstärke	Distanz zum Gerät	gesprochene Sprache	Erkennungsrate	Anzahl der erkannten Hotwords
10db	1m	50db	95%	19/20
	2m	60db	90%	18/20
	5m	60db	90%	18/20
50db	1m	60db	100%	20/20
	2m	60db	90%	18/20
	5m	60db	70%	14/20

Sprache zu jeder Zeit lauter als die Musik sein muss, damit das Hotword erkannt wird. Dies passiert wie in einer normalen Konversation jedoch automatisch und ist vergleichbar mit einer Unterhaltung zwischen zwei Personen oder dem Rufen eines Namens während im Hintergrund störende Geräusche auftreten.

Im Folgenden wird die Erkennungsrate der Spracherkennung, die Analyse der Intents und die Sprachausgabe evaluiert. Jedes Kommando wurde in einem Abstand von einem Meter fünfmal in normaler Lautstärke ausgesprochen. Unter der Korrektheit wird in den Modulen dies verstanden:

- STT: Jedes Wort des Kommandos wurde in allen Durchgängen exakt transkribiert.
- NLU: Der korrekte Intent wurde in allen Durchgängen aus dem Text extrahiert.
- TTS: Die Antwort wurde in allen Durchgängen korrekt aus dem Text generiert.

Die einzelnen Laufzeiten beziehen sich auf die Zeit zwischen dem Ende des gesprochenen Kommandos und der Rückgabe der angeforderten Daten.

Die gesamte Laufzeit ist stark von der Länge des gesprochenen Kommandos abhängig und der dadurch übertragenen Audiodatei. Für weiterführende Test kann u.a. die sample-rate (samples per second (Hertz)) und weitere Parameter angepasst werden, wodurch die Audioqualität der Aufnahme geändert wird. Diese wirkt sich direkt auf

Tabelle 12.2: Messung der Laufzeiten der einzelnen Module des Mira Sprachassistenten

Kommando	STT		NLU (API.AI)		TTS		Gesamt Laufzeit einer Anfrage
	korrekt	Laufzeit	korrekt	Laufzeit	korrekt	Laufzeit	
Was kannst du	✓	747ms	✓	404ms	✓	643ms	1,794s
Stelle Timer auf 2 Stunden	✓	954ms	✓	403ms	✓	608ms	1,965s
Fahre auf deinen Stellplatz	✓	802ms	✓	426ms	✓	723ms	1,951s
Der wievielte Tag ist heute	✓	940ms	✓	410ms	✓	515ms	1,865s
Wie wird das Wetter (Fallback Intent)	✓	788ms	✓	401ms	✓	411ms	1,6s

die Bandbreite des Uploads aus und ist ein weiterer Merkmal für die Geschwindigkeit des STT Moduls. Ungeachtet dessen ist eine Gesamtlaufzeit von unter zwei Sekunden für den Betrieb eines Sprachassistenten geeignet.

12.1 Bewertung der Ergebnisse

Die Evaluation hat also gezeigt, dass es möglich ist den Sprachassistenten durch ein Hotword zu aktivieren, Kommandos zu stellen und wohldefinierte Antworten zu erhalten. Dies ist sowohl mittels Kommandozeile als auch mit natürlicher Sprache möglich. Zudem wurde bereits gezeigt, dass dieser Assistent durch Erstellung weiterer Intents persönlicher wird und einen größeren Funktionsumfang erhalten kann. Etwas das während des Entwurfs der Implementierung auffiel, war dass viele proprietäre Sprachassistenten zwar ein SDK anbieten, dieses jedoch in deutscher Sprache noch einige Schwächen aufweist und der Sprachassistent nicht gut anpassbar ist. So kann z.B. der Google Assistant mit Hotword Erkennung nur in englischer Sprache betrieben werden. Weiterhin war es schwierig eine gute und kostenlose STT Engine zu finden.

Kapitel 13

Zusammenfassung und Ausblick

13.1 Zusammenfassung

In dieser Arbeit wurde zunächst ein Dialog System in alle seine Module dekomponiert und detailgetreu analysiert. Dabei wurde speziell auf die Komponenten: Automatic Speech Recognition, Spoken Language Understanding, Dialog Management, Natural Language Generation und Text-to-Speech eingegangen. Es wurde deutlich, dass sich aktuelle Sprachassistenten bezüglich einer Implementierung dieser Komponenten voneinander unterscheiden und verschiedene Ansätze zur Umsetzung der einzelnen Funktionalitäten anstreben.

Im Folgenden wurden ausgewählte proprietäre und Open Source Sprachassistenten bezüglich ihrer Prinzipien, System Architektur und Implementierbarkeit dargestellt. Hier wurden zur Analyse Patente und öffentliche Dokumente seitens der einzelnen Unternehmen hinzugezogen. Zudem wurde eine anschauliche Darstellung zum Vergleich aller vorgestellten Sprachassistenten bezüglich einiger Kernmerkmale erstellt.

Während der Arbeit stellte sich heraus, dass das Gebiet der Sprachassistenten ein starkes Wachstum erfährt und ich deshalb den Funktionsumfang der einzelnen Assistenten mehrmals aktualisieren musste. Weiterhin entschied ich mich dafür,

dass ein proprietärer Sprachassistent nicht genügend angepasst werden kann, um die Anforderungen für einen mobilen Roboter im Altenheim zu erfüllen. Dies führte zu der Entscheidung einen eigenen Sprachassistent zu erstellen, der auf verfügbare Komponenten (z.B. Google TTS) aufbaut. Dieser besitzt die Möglichkeit ein personalisiertes Hotword zu setzten (in unserem Fall *Hey Mira*) und entsprechende Aktionen, nach einer Analyse der Intents in der Cloud, auszuführen.

13.2 Ausblick

Mit Hilfe der in dieser Arbeit entwickelt und implementierten Sprachassistenten, werden auf Kommandos des Nutzers wohldefinierte Antworten geliefert. Der Funktionsumfang des Mira Assistenten ist jedoch im Vergleich mit proprietären und kommerziellen Produkten sehr rudimentär. Einer Erweiterung der unterstützten Intens ist jedoch keine Limitierung gesetzt. Weitere Möglichkeiten zur Erweiterung des Mira Assistenten umfassen zum einen Multithreading, ein definierter Interrupt zum Neustarten der Applikation, sowie eine Maskierung der Sprachausgabe. Indem die Tonspur der Ausgabe aus der Mikrofon Ausgabe herausgerechnet wird, ist es möglich eine Tonaufnahme während der Sprachausgabe zu nutzen und dementsprechend weitere Kommandos zu geben. Eine andere Möglichkeit wäre ein entsprechendes Hotword (z.B. Stopp oder Mira Stopp) zu erstellen, welches auch in einer lauten Umgebung erkannt wird. In den vorgestellten Sprachassistenten ist es deshalb möglich das Kommando (*Google Stopp*) während der Aussage zu geben, um die aktuelle Ausgabe zu unterbrechen. Weiterhin kann evaluiert werden, ob einzelne Module (z.B. der Intent Parser) lokal auf dem Gerät genutzt werden können, um dadurch an Performance zu gewinnen. Für eine optimale Nutzung sollte bei dem mobilen Roboter ein Array von Fernfeld-Mikrofonen genutzt werden.

Anhang A

Ergänzende Unterlagen

```
{
  "userSays": [
    {
      "id": "8824091c-1733-4a18-80e8-b630f233192c",
      "data": [
        {
          "text": "wie ist "
        },
        {
          "text": "heute Morgen",
          "alias": "date-time",
          "meta": "@sys.date-time",
          "userDefined": true
        },
        {
          "text": " das Wetter in "
        },
        {
          "text": "Berlin",
          "alias": "address",
          "meta": "@sys.location",
```

```
      "userDefined": true
    }
  ]
 }
],
"id": "f1b75ecb-a35f-4a26-88fb-5a8049b92b02",
"name": "weather",
"responses": [
 {
  "resetContexts": false,
  "action": "weather",
  "affectedContexts": [
   {
    "name": "weather",
    "lifespan": 5
   }
  ],
  "parameters": [
   {
    "dataType": "@sys.location",
    "name": "address",
    "value": "$address"
   },
   {
    "dataType": "@unit-temperature",
    "name": "unit",
    "value": "$unit",
    "isList": false
   }
  ]
 }
],
```

```
54   "events": [
55     {
56       "name": "GOOGLE_ASSISTANT_WELCOME"
57     }
58   ]
59 }
```

Listing A.1: Sample json intent

```
1
2 {
3   "id": "34fe2fc7-eef1-490f-aba1-2605f6f079c3",
4   "name": "weather-condition",
5   "isOverridable": true,
6   "entries": [
7     {
8       "value": "rain",
9       "synonyms": [
10        "regen",
11        "regnen",
12        "regnet"
13      ]
14    },
15    {
16      "value": "snow",
17      "synonyms": [
18        "schnee",
19        "schneit",
20        "schneien"
21      ]
22    },
23    {
24      "value": "sun",
```

```
25        "synonyms": [
26          "sonne",
27          "sonnig",
28          "Sonnenschein"
29        ]
30      }
31    ]
32  }
```

Listing A.2: Sample json entities

```
1  import UIKit
2  import Intents
3
4  class ViewController: UIViewController {
5
6      override func viewDidLoad() {
7          super.viewDidLoad()
8
9          INPreferences.requestSiriAuthorization {
10             status in
11             if status == .authorized {
12                 print("Autorisierung erfolgreich!")
13             }
14             else {
15                 print("Autorisierung fehlgeschlagen!")
16             }
17         }
18     }
19 }
```

Listing A.3: Siri Autorisierungs Dialog

```
func resolveDropOffLocation(
        forRequestRide intent: INRequestRideIntent,
        with completion: (INPlacemarkResolutionResult) -> Void) {
    let location = intent.dropOffLocation
    var result : INPlacemarkResolutionResult = nil

    if location != nil {
        // If the location is valid, use it; otherwise,
        // let the user know it is not supported
        if self.locationIsInsideServiceArea(location) {
            result = INPlacemarkResolutionResult.
                            successWithResolvedValue(location)
        }
        else {
            result = INPlacemarkResolutionResult.unsupported()
        }
    }
    else {
        // Ask for the drop-off location.
        result = INPlacemarkResolutionResult.needsValue()
    }

    // Return the result back to SiriKit.
    completion(result)
}
```

Listing A.4: Siri Intent Parameter verarbeiten [APPLEd]

```
{
  "luis_schema_version": "1.3.0",
  "name": "LuisBot",
  "desc": "",
```

```
5   "culture": "en-us",
6   "intents": [
7     {
8       "name": "SearchHotels"
9     },
10    {
11      "name": "Help"
12    }
13  ],
14  "entities": [
15    {
16      "name": "Hotel"
17    },
18    {
19      "name": "AirportCode"
20    }
21  ],
22  "composites": [],
23  "closedLists": [],
24  "bing_entities": [
25    "geography"
26  ],
27  "actions": [],
28  "model_features": [
29    {
30      "name": "Near",
31      "mode": true,
32      "words": "near,around,close,nearby",
33      "activated": true
34    },
35    {
36      "name": "Show",
```

```
37        "mode": true,
38        "words": "show,find,look,search",
39        "activated": true
40      }
41    ],
42    "regex_features": [
43      {
44        "name": "AirportCodeRegex",
45        "pattern": "[a-z]{3}",
46        "activated": true
47      }
48    ],
49    "utterances": [
50      {
51        "text": "look for hotels in miami",
52        "intent": "SearchHotels",
53        "entities": []
54      },
55      {
56        "text": "search for hotels in seattle",
57        "intent": "SearchHotels",
58        "entities": []
59      },
60      {
61        "text": "help me",
62        "intent": "Help",
63        "entities": []
64      }
65    ]
66  }
```

Listing A.5: LUIS.ai Bot Applikation [Microsoftb]

```
1  $ pip install gTTS
2  $ pip install SpeechRecognition
3
4  $ sudo apt-get install portaudio19-dev && sudo pip install pyaudio
5
6  # FOR WIKIPEDIA
7  $ pip install duckduckgo2
8
9  # FOR NEWS
10 $ sudo pip install feedparser
11
12 # API.AI SDK
13 $ pip install apiai
14
15 # FOLLOWING INSTALL INSTRUCTIONS FOR SNOWBOY
16 $ sudo apt install sox
17
18 # Download swig 3.0.12
19 $ ./configure --prefix=/usr --without-clisp --without-maximum-
       compile-warnings && make
20
21 $ sudo make install &&
22 install -v -m755 -d /usr/share/doc/swig-3.0.12 &&
23 cp -v -R Doc/* /usr/share/doc/swig-3.0.12
24
25 # Libatlas
26 $ sudo apt-get install libatlas-base-dev
27
28 #Copmile a Python Wrapper for Snowboy
29 $ cd swig/Python
30 $ make
31
```

```
32 ### Code Abschnitt muss von def __init__ in def start verschoben
       werden
33         def audio_callback(in_data, frame_count, time_info, status)
       :
34             self.ring_buffer.extend(in_data)
35             play_data = chr(0) * len(in_data)
36             return play_data, pyaudio.paContinue
37 
38         self.audio = pyaudio.PyAudio()
39         self.stream_in = self.audio.open(
40             input=True, output=False,
41             format=self.audio.get_format_from_width(
42                 self.detector.BitsPerSample() / 8),
43             channels=self.detector.NumChannels(),
44             rate=self.detector.SampleRate(),
45             frames_per_buffer=2048,
46             stream_callback=audio_callback)
47 ###
```

Listing A.6: Mira Assistent Dependencies

```
1 {
2   "userSays": [
3 ...
4       "data": [
5         {
6           "text": "Geh mir aus dem Weg"
7 ...
8       "data": [
9         {
10          "text": "Fahre in die "
11        },
12        {
```

```
13            "text": "Küche",
14            "alias": "rooms",
15            "meta": "@rooms",
16            "userDefined": false
17 ...
18   "name": "move",
19   "responses": [
20     {
21       "parameters": [
22         {
23           "dataType": "@rooms",
24           "name": "rooms",
25           "value": "$rooms",
26           "isList": false
27         }
28       ],
29       "messages": [
30         {
31           "speech": "Wird gemacht."
32         }
33   ...
34 }
```

Listing A.7: Codeausschnitt des Mira Move Intents

Abbildungsverzeichnis

Literaturverzeichnis

[api, 2017] (2017). *API.AI Entities*. `https://api.ai/docs/entities` Accessed: 2017-06-01.

[AIa] AI, MYCROFT. *Mycroft Adapt Intent Parser*. `https://adapt.mycroft.ai/` Accessed: 2017-08-02.

[AIb] AI, MYCROFT. *Mycroft Core Overview*. `https://docs.mycroft.ai/core.overview` Accessed: 2017-08-02.

[AIc] AI, MYCROFT. *Mycroft Create a Skill*. `https://docs.mycroft.ai/skill.creation` Accessed: 2017-08-02.

[AId] AI, MYCROFT. *Mycroft Mimic Text to Speech*. `https://mimic.mycroft.ai/home` Accessed: 2017-08-02.

[AIe] AI, MYCROFT. *Mycroft Open STT*. `https://openstt.org/` Accessed: 2017-08-02.

[APPLEa] APPLE. *Apple Siri*. `https://www.apple.com/de/ios/siri/` Accessed: 2017-07-17.

[APPLEb] APPLE. *Apple Siri Kit*. `https://developer.apple.com/documentation/sirikit` Accessed: 2017-07-17.

[APPLEc] APPLE. *Siri Human Interface Guidelines*. `ttps://developer.apple.com/ios/human-interface-guidelines/features/siri/` Accessed: 2017-07-17.

[AppleEd] Apple. *Sirikit: Resolving the Parameters of an Intent.* https://developer.apple.com/documentation/sirikit/resolving_and_handling_intents/resolving_the_parameters_of_an_intent Accessed: 2017-07-17.

[AppleEe] Apple. *Supported Siri Interactions.* https://developer.apple.com/ios/human-interface-guidelines/features/siri/ Accessed: 2017-07-17.

[Bohus et al., 2007] Bohus, Dan, A. Raux, T. K. Harris, M. Eskenazi und A. I. Rudnicky (2007). *Olympus: an open-source framework for conversational spoken language interface research.* In: *Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies*, S. 32–39. Association for Computational Linguistics.

[Bohus und Rudnicky, 2003] Bohus, Dan und A. I. Rudnicky (2003). *RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda.*

[Bouché] Bouché, Katja. *Sprachsynthese: TTS Systeme.*

[Bowe et al., 2014] Bowe, Heather, K. Martin und H. Manns (2014). *Communication across cultures: Mutual understanding in a global world.* Cambridge University Press.

[Ferrucci et al., 2010] Ferrucci, David, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager et al. (2010). *Building Watson: An overview of the DeepQA project.* AI magazine, 31(3):59–79.

[Ferrucci et al., 2013] Ferrucci, David, A. Levas, S. Bagchi, D. Gondek und E. T. Mueller (2013). *Watson: Beyond Jeopardy!.* Artificial Intelligence, 199:93 – 105.

[Gabel et al., 2014] Gabel, M., C. Brigham, A. Cheyer und D. Kittlaus (2014). *Dynamically evolving cognitive architecture system based on third-party developers.* US Patent App. 14/306,856.

[Gartner] Gartner. *Worldwide Smartphone Sales to End Users by Operating System in 4Q16 (Thousands of Units).* `http://www.gartner.com/newsroom/id/3609817` Accessed: 2017-05-07.

[Google, 2017a] Google (2017a). *Actions are entry points into your app that define the invocation and discovery model for your app.* `https://developers.google.com/actions/components/actions` Accessed: 2017-05-21.

[Google, 2017b] Google (2017b). *Google Actions: Invocation and Discovery.* `https://developers.google.com/actions/discovery/` Accessed: 2017-05-20.

[Google, 2017c] Google (2017c). *Google Assistant SDK*. `https://developers.google.com/actions/get-started/create-project-agent` Accessed: 2017-07-20.

[Google, 2017d] Google (2017d). *Understanding How Conversations Work: The Key to a Better UI*. `https://developers.google.com/actions/design/how-conversations-work` Accessed: 2017-05-20.

[Gruber et al., 2012] Gruber, T.R., A. Cheyer, D. Kittlaus, D. Guzzoni, C. Brigham, R. Giuli, M. Bastea-Forte und H. Saddler (2012). *Intelligent Automated Assistant*. US Patent App. 12/987,982.

[Harshita Phatnani, 2015] Harshita Phatnani, Mr. Jyotiprakash Patra, Ankit Sharma (2015). *CHATBOT ASSISTING: SIRI*. In: *Proceedings of BITCON-2015 Innovations For National Development National Conference on : Research and Development in Computer Science and Applications.*

[IBMa] IBM. *IBM Client Applicaation.* `https://console.bluemix.net/docs/services/conversation/develop-app.html#building-a-client-application` Accessed: 2017-07-28.

[IBMb] IBM. *IBM Conversation.* `https://console.bluemix.net/docs/services/conversation/index.html` Accessed: 2017-07-26.

[IBMc] IBM. *IBM Conversation Dialog Build*. https://console.bluemix.net/docs/services/conversation/dialog-build.html#dialog-build Accessed: 2017-07-28.

[IBMd] IBM. *IBM Document Conversion*. https://console.bluemix.net/docs/services/document-conversion/index.html Accessed: 2017-07-26.

[IBMe] IBM. *IBM Natural Language Classifier*. https://console.bluemix.net/docs/services/natural-language-classifier/natural-language-classifier-overview.html Accessed: 2017-07-26.

[IBMf] IBM. *IBM Retrieve and Rank*. https://www.ibm.com/watson/developercloud/doc/retrieve-rank/index.html Accessed: 2017-07-26.

[Jokinen und McTear, 2010] Jokinen, K. und M. McTear (2010). *Spoken Dialogue Systems*. Synthesis lectures on human language technologies. Morgan & Claypool Publishers.

[Karrer, 2010] Karrer, Tony (2010). *Digital Signal Processor and Text-to-Speech*. http://elearningtech.blogspot.de/2010/06/digital-signal-processor-and-text-to.html Accessed: 2017-05-14.

[KITT.AI] KITT.AI. *Snowboy, a hotword detection engine*. https://snowboy.kitt.ai/ Accessed: 2017-08-15.

[Krisnawati] Krisnawati, Lucia D. *Einführung in die Spracherkennung & Sprachsynthese*. http://www.cis.lmu.de/~hs/teach/13w/intro/pdf/15asr.pdf Accessed: 2017-04-10.

[Mayword, 2015] Mayword, A. (2015). *Windows 10 Cortana: Tips and Tricks:*. 1. CreateSpace Independent Publishing Platform.

[Microsofta] Microsoft. *Cortana: Bot based Skill*. https://docs.microsoft.com/en-us/cortana/tutorials/bot-skills/creating-a-bot-based-skill Accessed: 2017-07-20.

[MICROSOFTb] MICROSOFT. *Cortana: Bot based Skill*. https://github.com/Microsoft/BotBuilder-Samples/blob/master/CSharp/intelligence-LUIS/ Accessed: 2017-07-20.

[MICROSOFTc] MICROSOFT. *Cortana Intelligence Suite*. https://social.technet.microsoft.com/wiki/contents/articles/36688.introduction-to-cortana-intelligence-suite.aspx Accessed: 2017-07-20.

[MICROSOFTd] MICROSOFT. *Cortana Skills Kit*. https://docs.microsoft.com/en-us/cortana/getstarted Accessed: 2017-07-20.

[MICROSOFTe] MICROSOFT. *Principles of Cortana Skill Design*. https://docs.microsoft.com/en-us/cortana/design-guides/skill-design-principles Accessed: 2017-07-20.

[MONTGOMERYa] MONTGOMERY, JOSHUA. *Mycroft: An Open Source Artificial Intelligence For Everyone*. https://www.kickstarter.com/projects/aiforeveryone/mycroft-an-open-source-artificial-intelligence-for?lang=de Accessed: 2017-08-02.

[MONTGOMERYb] MONTGOMERY, JOSHUA. *Why Name It Mycroft?*. https://mycroft.ai/why-name-it-mycroft/ Accessed: 2017-08-01.

[NASREEN] NASREEN, NUZHAT. *Bixby Supported Apps*. http://www.theandroidsoul.com/bixby-supported-apps/ Accessed: 2017-07-24.

[REITER und DALE, 1997] REITER, EHUD und R. DALE (1997). *Building applied natural language generation systems*. Natural Language Engineering, 3(1):57–87.

[SAMSUNG] SAMSUNG. *Samsung Bixby*. http://www.samsung.com/global/galaxy/apps/bixby/ Accessed: 2017-07-24.

[SARIKAYA et al., 2016] SARIKAYA, RUHI, P. A. CROOK, A. MARIN, M. JEONG, J.-P. ROBICHAUD, A. CELIKYILMAZ, Y.-B. KIM, A. ROCHETTE, O. Z. KHAN,

X. Liu et al. (2016). *An overview of end-to-end language understanding and dialog management for personal digital assistants*. In: *Spoken Language Technology Workshop (SLT), 2016 IEEE*, S. 391–397. IEEE.

[Stolcke et al., 1997] Stolcke, Andreas, Y. Konig und M. Weintraub (1997). *Explicit word error minimization in n-best list rescoring.*. In: *Eurospeech*, Bd. 97, S. 163–166.

[Tur und De Mori, 2011] Tur, Gokhan und R. De Mori (2011). *Spoken language understanding: Systems for extracting semantic information from speech*. John Wiley & Sons.

[University, 2008] University, Carnegie Mellon (2008). *Olympus architecture incorporates modules developed by researchers at Carnegie Mellon and by others*. `http://wiki.speech.cs.cmu.edu/olympus/index.php/Olympus` Accessed: 2017-05-12.

[Wang et al., 2005] Wang, Ye-Yi, L. Deng und A. Acero (2005). *Spoken language understanding*. IEEE Signal Processing Magazine, 22(5):16–31.

[Zhang] Zhang, Anthony. *Library for performing speech recognition, with support for several engines and APIs, online and offline.*. `https://github.com/Uberi/speech_recognition` Accessed: 2017-08-17.