UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Fachgruppe Data Science
Warburger Straße 100
33098 Paderborn

Konzepte und Modellierung von zielorientierten Gesprächsstrategien für Chatbots

Bachelorarbeit

im Rahmen des Studiengangs Informatik
zur Erlangung des Grades
Bachelor of Science

von
Lukas Brandt

in Kooperation mit
Atos Information Technology GmbH

Betreut durch: Dr. Ricardo Usbeck & Dr. Wolfgang Thronicke

vorgelegt bei:
Prof. Dr. Axel Ngonga
und
Prof. Dr. Michaela Geierhos

Paderborn, 19. Dezember 2017

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Signatur

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund und Motivation

In der heutigen Zeit sind Chat-Plattformen wie WhatsApp, Telegram, Skype und Co. aus dem Alltag nicht mehr wegzudenken. Sie bilden somit eine Schnittstelle zu Kunden und Endanwendern, bei der diese sich nicht mehr an eine Webseite oder Anwendung gewöhnen müssen, sondern direkt über die ihnen vertrauten Chat-Plattformen mit einem Unternehmen oder einer Organisation kommunizieren können. Weiterhin reicht zur Kommunikation das Beherrschen der verwendeten Sprache aus und es wird kein tiefergehendes technisches Wissen benötigt. Diese Digitalisierung der Kommunikation kann weiter vorangetrieben werden, indem das Chatten auf Seiten des Unternehmens zur Kundenbetreuung nicht mehr durch einen dedizierten Mitarbeiter, sondern durch eine Software, einen sogenannten Chatbot, durchgeführt wird. Jedoch mangelt es an Herangehensweisen, komplexere Chatbots zu entwerfen, welche auch über einfache Aufgaben hinaus mit Menschen kommunizieren können und sich dabei deterministisch und nachvollziehbar verhalten.

1.2 Aktueller Stand der Chatbots

Mit Begriffen wie Chatbots oder Dialogsystemen werden häufig fälschlicherweise Systeme wie der Google Assistant, Microsoft Cortana, Amazon Alexa oder Apples Siri assoziiert. Diese haben jedoch die Aufgabe, auf Befehle und Anweisungen des menschlichen Benutzers zu reagieren und diese auszuführen. Beispiele dafür wären Befehle wie: „Sag mir, wie das Wetter morgen wird“ oder „Stell mir einen Wecker auf 9 Uhr“. Hierbei hat der Benutzer die Gesprächsführung inne und das System gibt nur Rückmeldung, weshalb auch von einem sogenannten Assistenzsystem gesprochen wird. Es findet kein tatsächlicher Dialog mit zwei gleichwertigen Teilnehmern statt.
In vielen Anwendungssituationen wäre es jedoch hilfreich, dass der Dialog von beiden Seiten beeinflusst werden kann, oder sogar nur der Chatbot souverän das Gespräch führt, um zum Beispiel einen Geschäftsprozess durchzuführen. Gerade für sehr simple Aufgaben wie die Beantwortung von häufig gestellten Fragen (zum Beispiel bezüglich Öffnungszeiten) oder das zielorientierte Durchführen von Standardabläufen (zum Beispiel das Reservieren eines Tisches als Ziel) können Chatbots eingesetzt werden.

1.3 Anforderungen aus dem industriellen und medizinischen Anwendungsfall

In der Forschung etabliert sich die Herangehensweise, maschinelle Lernverfahren auszuprobieren, wenn der Anwendungsfall komplexer wird und über simple Aufgaben hinausgeht. Da maschinelle Lernverfahren jedoch statistisch arbeiten und somit nur zu gewissen Wahrscheinlichkeiten wie gewünscht reagieren, sind sie nicht für jede Domäne geeignet. Gerade in sehr riskanten Anwendungsfällen, wenn es zum Beispiel in der Industrie um die Bedienung einer teuren Maschine oder in der medizinischen Betreuung um das Wohlergehen eines Patienten geht, wird das Umgehen mit Wahrscheinlichkeiten nach Möglichkeit vermieden. Deshalb wird ein Vorgehen benötigt, welche komplexere Aufgaben in Dialogform lösen kann. Jede Nachricht des Chatbots soll durch deterministische Regeln ausgelöst werden, sodass diese Dialoge zu jeder Zeit so ablaufen, wie sie spezifiziert wurden.
Innerhalb dieser Arbeit wird exemplarisch ein medizinischer Anwendungsfall zur Referenz und zur Bewertung eines Chatbot-Prototypen verwendet. Bei diesem Anwendungsfall handelt es sich um das europäische Forschungsprojekt Medolution[1], bei dem ein System zur telemedizinischen Betreuung von Patienten entwickelt werden soll. Wichtig für diese Art der Betreuung ist das Erheben und Verarbeiten von medizinischen Daten solcher Patienten. Durch die Verarbeitung kann ein behandelnder Arzt genaue Analyseergebnisse erhalten und bei kritischen Werteentwicklungen alarmiert werden. Ein möglicher Kommunikationskanal, durch den Patienteninformationen erhoben werden können, soll durch einen Chatbot realisiert werden. Durch eine solche telemedizinische Betreuung kann einem Arzt der Zeitaufwand für einfache Routinegespräche genommen werden, sodass er mehr Zeit für eine intensivere Behandlung kritischer Fälle hat. Der Patient wird ebenfalls entlastet, da er nur Termine vereinbaren muss, wenn eine tatsächliche Behandlung benötigt wird. Wenn er Fragen hat oder Informationen übermitteln möchte, kann er jederzeit einen Dialog mit dem Chatbot führen. Durch ein geeignetes Dialogmodell kann dieser Chatbot hierbei benötigte Informationen über den Patienten sammeln und diesem dazu passend Feedback und Ratschläge geben. Dabei ist es jedoch wichtig, dass ein Chatbot sich durchgehend einer Spezifikation entsprechend verhält, da der medizinische Umgang mit Patienten eine sehr riskante Domäne ist. Eine fehlerhafte Einschätzung eines medizinischen Wertes als ungefährlich könnte zum Beispiel dafür sorgen, dass der Patient schlechte Verhaltensweisen beibehält und sich der Wert verschlimmert, da im Dialogmodell des Chatbots keine klaren Ober- und Untergrenzen für diesen Wert definiert wurden.

[1] `https://itea3.org/project/medolution.html`

1.4 Problemstellung und Ziele

In dieser Arbeit soll eine Methode entwickelt werden, mit der komplexe, aber deterministische Dialoge modelliert werden können. Hierzu werden zwei vorhandene Modellierungsansätze, welche jedoch beide jeweils eigene Schwachpunkte haben, zu einem neuen hybriden Modellansatz vereint. Wie dieser hybride Ansatz gestaltet ist und wie dabei jeweils die Schwachpunkte der beiden zugrundeliegenden Modellierungsansätze ausgeglichen werden, wird im Laufe dieser Arbeit erläutert. Dazu zählt die Definition und Erklärung der einzelnen Elemente eines solchen Modells sowie eine zugängliche Syntax zum Entwerfen und zur Repräsentation dieser Modelle. Die Modelle sollen es einem Chatbot ermöglichen, sich innerhalb des Gesprächs differenziert zu verhalten und die Entscheidung zu einer Variante soll durch Bedingungen innerhalb des Modells definiert sein. Alternativ kann die Entscheidung ebenfalls durch externe Services durchgeführt werden, wie zum Beispiel Anfragen an eine Ontologie. Weiterhin wird ein Algorithmus entworfen, der auf der Grundlage dieser Modelle einen Dialog mit einem Menschen zielorientiert durchführen kann. Dabei soll der Algorithmus in ein modulares System eingebettet sein. Es steht nicht im Vordergrund der Arbeit, dass einzelne Module des Systems jeweils optimale Ergebnisse erreichen. Ein Beispiel dafür wäre eine perfekte Genauigkeit bei der sprachlichen Analyse der Nachrichten des Benutzers. Stattdessen dient das System als Prototyp zur Untersuchung der Qualität der Dialoge, welche mit dem neu entwickelten Modellierungsansatz entworfen und mit dem Algorithmus durchgeführt wurden.

Nach dieser Einleitung ist die Arbeit inhaltlich in folgende Kapitel gegliedert:

2 - Theoretische Grundlagen: Hier wird zuerst die Auswertung natürlicher Sprache erläutert. Die dabei erkannte Semantik wird benötigt, damit der Chatbot versteht, was der Benutzer ihm mitteilen möchte. Weiterhin wird erklärt, wie durch Wissensgraphen und Ontologien die Entscheidungsfindung des Chatbots unterstützt werden kann und welche Eigenschaften die geführten Dialoge haben sollten, damit ein menschlicher Benutzer an diesen so natürlich wie möglich teilhaben kann. Abschließend wird ein besonderer Fokus auf existierende Ansätze von Dialogmodellen für Chatbots gelegt, um den Ansatz dieser Arbeit von diesen abzugrenzen, aber auch den Ursprung einiger gemeinsamer Ideen darzustellen.

3 - Entwicklung und Modellierung von Gesprächszielen und -strategien: In diesem Kapitel wird der neue Ansatz für Dialogmodelle dieser Arbeit erläutert. Dafür werden zuerst die Elemente eines solchen Modells und die Wechselwirkungen der Elemente untereinander vorgestellt. Danach wird eine grafische Syntax definiert, die verwendet werden kann, um Dialoge in dieser Modellform zu entwerfen. Abschließend wird der Algorithmus erläutert, der den Benutzer Nachricht für Nachricht durch ein solches Dialogmodell führt.

4 - Architektur und Implementierung eines Chatbot-Prototypen: Nachdem im vorherigen Kapitel die theoretischen Ideen dargestellt wurden, wird in diesem Kapitel eine beispielhafte Implementierung eines Chatbots auf Basis der neuen Modelle mit dem neuen Algorithmus dokumentiert. Zuerst wird dargestellt, wie sich dieser Chatbot in das gesamte Umfeld des Medolution-Projektes eingliedern lässt. Danach wird erklärt, wie das beschriebene modulare System für den Chatbot aufgebaut ist und welche Software-Bibliotheken für die Implementierung des Prototyps verwendet wurden.

5 - Evaluation: Mit dem entwickelten Chatbot-Prototyp und einem beispielhaften Dialogmodell aus dem Medolution-Kontext soll bewertet werden, wie benutzerfreundlich der für diese Arbeit entwickelte Chatbot ist. Dies geschieht mit Hilfe der so genannten *System Usability Scale*.

6 - Zusammenfassung und Ausblick: Abschließend werden die Erkenntnisse der Entwicklung des neuen Modells mit besonderem Hinblick auf die Ergebnisse der Evaluation beschrieben. Weiterhin werden einige Ideen als Ausblick gegeben, welche zusätzlichen Funktionalitäten dem Chatbotsystem hinzugefügt und um welche Elemente die Dialogmodelle erweitert werden könnten.

2 Theoretische Grundlagen

Um einen Dialog, also eine zielorientierte und sprachliche Interaktion durch eine Folge von Äußerungen, maschinell durchzuführen, gilt es mehrere Aspekte zu beachten. Hamerich zählt hierzu Folgende auf: Linguistische Verarbeitung, sprachtechnologische Verarbeitung, Wissensverarbeitung und Ergonomie [11]. Von der sprachtechnologischen Verarbeitung wird in diesem Grundlagenkapitel abgesehen, da akustische Sprachinformationen für einen Chatbot nicht relevant sind. Alle anderen werden jedoch im Folgenden in der obigen Reihenfolge behandelt. Wie in Abbildung 2.1 zu sehen ist, könnte somit die Architektur eines Chatbots in die Komponenten Textauswertung, Textgenerierung, Dialogsystem, grundlegende Wissenbasis und die durch den Chatbot zu bedienende Anwendung aufgeteilt werden.

2.1 Maschinelle Auswertung von natürlicher Sprache

Damit das Dialogsystem des Chatbots die Nachrichten des Benutzers korrekt verarbeiten und eine Antwort generieren kann, gilt es zunächst, Informationen aus der Benutzernachricht durch die linguistische Verarbeitung zu extrahieren. Mit diesen kann die Logik der Gesprächsstrategie weiter fortfahren. Aus der Architekturperspektive betrachtet befindet sich dieser Bereich in der Textauswertungskomponente aus Abbildung 2.1. Innerhalb dieser Arbeit werden Informationen durch maschinelles Lernen extrahiert. Für die maschinelle Auswertung natürlicher Sprache gibt es zwar weitere Ansätze wie beispielsweise Systeme, die auf vordefinierten Regeln [4] oder Pattern Matching [19] basieren, jedoch wird in diesem Grundla-

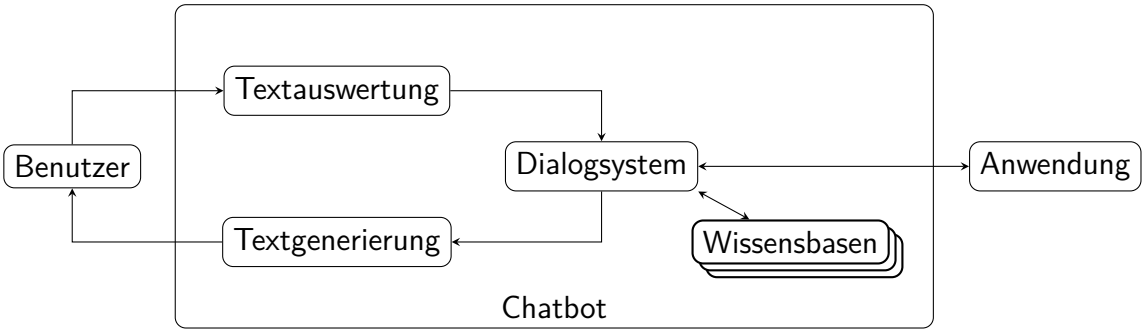


Abbildung 2.1: Komponenten eines Chatbots

genkapitel von solchen Ansätzen abgesehen.
Im Folgenden wird diese Auswertung durch maschinelles Lernen anhand des folgenden Dialogbeispiels erläutert:

Chatbot: Was haben Sie zuletzt gegessen?
Benutzer: Ich habe um 14 Uhr einen Apfel gegessen.

2.1.1 Klassifizierung der Intention

Zuerst ist die Intention, mit welcher der Benutzer die Nachricht geschickt hat, wichtig. In dem Dialogbeispiel wäre diese Intention eine Entitäten-Information, also die Antwort bezüglich der gegessenen Lebensmittel, nach denen zuvor vom Chatbot gefragt wurde. Wie dabei bereits zu beobachten ist, bilden innerhalb von gängigen Dialogen die Intentionen ein logisches Paar aus der geäußerten Intention des einen Gesprächspartners und einer Antwort mit einer passenden Gegenintention des anderen Gesprächspartners [11]. Hier sind diverse Kombinationen von Paaren möglich, wie zum Beispiel allgemein eine Begrüßung und die Erwiderung der Begrüßung oder eine Information zu erfragen und die Antwort zu liefern. Es sind jedoch auch sehr domänenspezifische Intentionen möglich und in vielen Fällen auch nötig. Für eine passende Antwort, die den Benutzer nicht verwirrt, ist es also wichtig, die Intention des Benutzers richtig zu kategorisieren, um mit der passenden und erwarteten Gegenintention reagieren zu können.
Laut Yang et al. sind Support Vector Machines (oder kurz auch: SVM) ein gängiges Werkzeug, um die Intention einer Benutzeräußerung mit ausreichenden Trainingsdaten kategorisieren zu können [24]. Bei einer SVM handelt es sich um ein Optimierungsproblem aus dem Bereich des maschinellen Lernens. Kategorisiert werden hierbei Vektoren aus $\mathbb{R}^n$ in eine Menge aus Labeln für die einzelnen Kategorien $L = l_1, l_2, ...l_n$. Optimiert wird eine separierende Hyperebene als Trenner zwischen den Kategorien, sodass alle Trainingsvektoren,die aus einer Kategorie stammen, zusammen sind und von den restlichen Vektoren getrennt werden [9]. Es werden als Trainingsdaten Vektoren betrachtet, die jeweils bereits ein Label, entsprechend ihrer Kategorie, haben. Zum Beispiel in Falle der Text-Klassifizierung eine vektorisierte Textäußerung und eine dazugehörige Intention. Wenn nun nach der Optimierung der Hyperebene ein neuer und unkategorisierter Eingabevektor $i \in \mathbb{R}^n$ betrachtet wird, kann durch die Lage zu dieser trennenden Hyperebene abgelesen werden, zu welcher Kategorie er mit welcher Wahrscheinlichkeit gehören würde. Also wird formal für die Eingabe i jedem Label $l \in L$ eine Wahrscheinlichkeit p zugeordnet. Dieses p_k modelliert die Wahrscheinlichkeit, dass auf Basis der trainierten Hyperebene die SVM die Eingabe i mit dem dazugehörigen Label l_k kategorisieren würde.
Für Textkategorisierungs-Aufgaben, wie zum Beispiel in diesem Fall die Kategorien der Intention, eignen sich nach Joachims die Support Vector Machines gut, da sie für das Problem der Überanpassung im Gegensatz zu anderen Modellen weniger anfällig sind [15].

2.1.2 Extrahieren von Entitäten

Die zweite wichtige Information sind die relevanten Entitäten einer Aussage. Diese Entitäten müssen nicht nur erkannt werden, sondern auch einer von mehreren zuvor bekannten Klassen zugeordnet werden. Hierbei ist es von der Domäne und Dialogart abhängig, welche Klassen von Entitäten für einen Anwendungsfall als relevant betrachtet werden. In dem Dialogbeispiel wären exemplarisch „14 Uhr“ mit der Klasse Uhrzeit und „Apfel“ mit der Klasse Lebensmittel relevante Entitäten. Das Problem, Wörter oder Phrasen innerhalb eines Textes als relevante Entitäten einzeln zu erkennen und einer von mehreren vorher bekannten Klassen zuzuordnen, heißt in der Forschung Named Entity Recognition (kurz: NER) [21]. Es gibt viele verschiedene Verfahren für NER, welche jeweils eine sehr unterschiedliche Herangehensweise haben. Eine Möglichkeit wäre es, rein heuristisch eine Erkennung und Zuordnung in Klassen mit vielen Trainingsdaten maschinell zu erlernen. Hierbei haben sich Conditional Random Fields (kurz: CRFs) als vielversprechender Ansatz erwiesen [21]. Bei vielen Klassen für NER-Probleme, wie beispielsweise Uhrzeiten oder Orte, sind die grammatikalischen Strukturen, in denen sie vorkommen können, jedoch sehr beschränkt. Somit sind sie auch bis zu einem gewissen Grad modellierbar und eingeschränkt mit diesem syntaktischen Modell durch einen Parser erkennbar. Anstatt die Entity Erkennung also ausschließlich zu trainieren wie bei CRFs, kann bei solchen bestimmten Klassen aus vorher bekannten Bereichen stattdessen auch ein hybrider Ansatz aus maschinellem Lernen und grammatikbasierten Verfahren benutzt werden. Dabei würde die Möglichkeit verloren gehen, jede Art von Entitäts-Kategorien erlernen zu können, ohne Grammatiken aufstellen zu müssen. Dafür werden jedoch weniger Trainingsdaten als vorher benötigt. Ein Beispiel für einen solchen hybriden Ansatz wäre eine Probabilistic Context Free Grammar (kurz: PCFG) [14].

2.2 Wissensgraphen und Ontologien

Da es in natürlicher Sprache viele unterschiedliche Möglichkeiten gibt, ein und denselben Fakt auszudrücken, braucht ein Chatbot eine Form von modelliertem Wissen, um diese unterschiedlich formulierten, aber semantisch identischen Aussagen als gleich erkennen zu können. Es kann vergleichbar sein mit dem menschlichen Allgemeinwissen oder gegebenenfalls spezielleres Domänenwissen für das Einsatzgebiet des Chatbots. Wenn der Benutzer zum Beispiel nach seiner Ernährung gefragt wird und überprüft wird, ob er ein Gemüse zu sich genommen hat, wäre es unvorteilhaft für die Erweiterbarkeit und die Wiederverwendbarkeit des Systems, wenn alle existierenden Gemüsearten direkt in den Quellcode geschrieben werden müssten, um die Eingabe mit diesen vergleichen zu können. Deshalb soll es dem Chatbot ermöglicht werden, Abfragen von solchen Fakten an vorhandenes Domänen- und Allgemeinwissen zu stellen. Die Abfragen werden an die Wissensbasen aus Abbildung 2.1 gestellt, in denen entsprechende Fakten, oder

auch Ontologien genannt, gespeichert sind.
Das Wort Ontologie (altgriechisch, etwa: „Die Lehre des Seins“) stammt aus dem Metaphysik-Bereich der Philosophie und ist eine Disziplin, die sich mit Aussagen über kategorisierte Enitäten und die Beziehungen zwischen diesen beschäftigt, also vereinfacht gesagt mit dem Aufstellen von Konzepten über die reale Welt [6]. Wenn ein solches Modell aus Konzepten einem Computer zur Verfügung gestellt werden soll, muss es in einer formalen und maschinenlesbaren Schreibweise vorhanden sein. Ein offener Standard für ein solches Format sind die Sprachen und Technologien des *Semantic Webs*. Beim *Semantic Web* werden die Daten einer solchen Ontologie spezifiziert (zum Beispiel mit dem RDF-Format), sodass durch Syntax und Semantik der Inhalt lesbar und interpretierbar für Maschinen wird, und durch formale Logik neue Informationen geschlussfolgert und extrahiert werden können [12].
Beim RDF-Format sind die Informationen als gerichteter Graph modelliert, bei dem jeder Knoten und jede Kante einen eindeutigen Bezeichner hat. Zwischen zwei eindeutig identifizierbaren Knoten wird durch eine Kante die Beziehung modelliert, wie zum Beispiel zwischen einem Buch und einem Verlag, dass das Buch bei diesem Verlag erschienen ist [12]. Diese Kombinationen aus den eindeutigen Bezeichnern, den sogennanten *URIs*, der beiden Knoten und der Beziehungskante werden im *Semantic Web* als Tripel aus dem *Subject*, dem ersten Knoten, dem *Predicate*, der Beziehung, und dem *Object*, dem zweiten Knoten, dargestellt [12]. Den Beziehungen lassen sich auch weitere Eigenschaften zuordnen wie zum Beispiel die Kardinalität oder Relationstypen (transitiv, symmetrisch, etc.), sodass der Wissengraph mit neuen Beziehungen, die automatisch inferiert werden, erweitert werden kann [12].
Wenn nun im Ernährungs Beispiel Unsicherheit herrscht, ob Mohrrüben auch Gemüse sind, ist in dem Graph eine Kante vorhanden zwischen Mohrrübe und Gemüse, wobei Mohrrübe eine Synonym-Beziehung zu Karotte hat und Karotte ein Gemüse ist. Dadurch ließe sich mit geeigneten Schlussfolgerungsregeln schließen, dass Mohrrübe ebenfalls ein Gemüse ist, da die Kanten des Pfades semantisch geeignete Beziehungen waren.

2.3 Eigenschaften natürlich wirkender Dialoge und Kommunikation

Um für den Benutzer eines Chatbots einen Mehrwert im Vergleich zu anderen Eingabemethoden zu bieten, muss der Chatbot einfacher bedienbar sein als sie. Dies wird erreicht, indem Eigenschaften natürlicher Dialoge zwischen Menschen beobachtet und Verhaltensweisen bestmöglich vom Dialogsystem imitiert werden, wodurch eine Dialog-Guideline für den Chatbot entsteht. Dieser Aspekt der einfachen Bedienbarkeit wird relevant für die Komponenten Dialogsystem und Textgenerierung aus der Architekturübersicht in Abbildung 2.1, da das Dialogsystem

auf den Benutzer möglichst entgegenkommend reagieren muss und die generierten beziehungsweise modellierten Texte der Antwort verständlich sein sollten.
Als eine Art Grundregelwerk der einfachen Bedienbarkeit und der natürlich wirkenden Kommunikation für Chatbors schlägt Hamerich die vier Grice'schen Maximen vor [11], welche im folgenden kurz zusammengefasst werden:

1. Maxim of Quantity - Äußerungen sollten so informativ wie möglich und nötig gestaltet sein, ohne dabei mit zu detaillierten Beiträgen Zeit zu verschwenden oder zu verwirren [10],

2. Maxim of Quality - Äußerungen sollten nur Informationen enthalten, von denen man der Meinung ist, dass sie wahr sind, und die man belegen könnte [10],

3. Maxim of Relation - Informationen in Äußerungen sollten immer relevant und wichtig für das aktuelle Thema des Dialoges sein [10],

4. Maxim of Manner - Äußerungen sollten so kurz wie möglich, geordnet, klar und eindeutig sein [10].

Menschen halten sich in aller Regel an diese Maxime, wenn beide Teilnehmer in ihrem Dialog ein gemeinsames Ziel erreichen wollen, oder genauer gesagt der Sprecher hält sich an die Maxime und der Zuhörer interpretiert die Informationen unter der Annahme, dass der Sprecher sich an die Maximen hält [10]. Wenn beispielsweise jemand gefragt wird „Was gab es zum Mittagessen?“ und die Person, von der die Antwort erwartet wird, kennt die genaue Bezeichnung des Gerichtes nicht, könnte diese Person daraufhin so genau antworten wie es ihr möglich ist, etwa mit „Irgendetwas mit Fisch“. Der Gesprächspartner wird nun ohne weitere Hinweise davon ausgehen können, dass der Gefragte den tatsächlichen Namen des Gerichts nicht kennt. Andernfalls hätte er die genaue Bezeichnung der groben Beschreibung vorgezogen, um nicht die Maxim of Quality zu verletzen. Deshalb braucht der Fragende keine vertiefenden Nachfragen bezüglich des Fischgerichtes zu stellen, da wahrscheinlich keine zufriedenstellende Antwort erwartet werden kann. Somit sollte es auch bei einem digitalen Dialog zum Erreichen der Ziele führen, wenn der Mensch und der Chatbot beide kooperativ agieren und sich an die vier Maximen halten. Wenn nun jede Nachricht, die der Chatbot versendet, keine der Maximen verletzt, hat er seinen Beitrag zur Kooperation erfüllt und deshalb sollten alle Nachrichten, Ziele und Strategiebestandteile mit Bedacht auf die Maximen definiert werden.

Zusätzlich gilt es noch standardisierte Anforderungen für Software zu betrachten. Hierfür existiert der internationale Standard DIN EN ISO 9241: „Ergonomie der Mensch-System-Interaktion“ und davon ist der Teil 110: „Grundsätze der Dialogsteuerung“ relevant mit den Anforderungen: Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit, Fehlertoleranz, Lernförderlichkeit, Steuerbarkeit, Erwartungskonformität und Individualisierbarkeit [20]. Dabei ist zu beachten, dass bei

der Verwendung des Wortes „Dialogsteuerung“ keine Chatbots oder andere Dialogsysteme gemeint sind, sondern allgemein grafische Oberflächen. Für Chatbots existieren keine eigenen standardisierten Anforderungen. Deshalb wurde dieser Ansatz für eine gute Bedienbarkeit von grafischen Oberflächen gewählt, weil zielorientierte Chatbots theoretisch mit grafischen Oberflächen austauschbar sind, welche dementsprechend viele grafische Elemente haben, um alle Interaktionen zu ermöglichen. Wenn also ein Chatbot eine austauschbare Alternative für eine solche Oberfläche darstellen soll, sollte er deshalb auch unter ähnlichen Aspekten und Anforderungen der Bedienbarkeit überprüft werden.
Aufgabenangemessenheit ist definiert als: „Ein interaktives System ist aufgabenangemessen, wenn es den Benutzer unterstützt, seine Arbeitsaufgabe zu erledigen, d. h., wenn Funktionalität und Dialog auf den charakteristischen Eigenschaften der Arbeitsaufgabe basieren anstatt auf der zur Aufgabenerledigung eingesetzten Technologie.“ [20]. Auf ein Dialogsystem übertragen würde dies also fordern, dass der Benutzer nicht durch für das Dialogziel unwichtige Aspekte abgelenkt oder behindert wird und dadurch keinen höheren Mehraufwand hat, als bei der Übermittlung der Daten ohne das System.
Selbstbeschreibungsfähigkeit ist so definiert: „Ein Dialog ist in dem Maße selbstbeschreibungsfähig, in dem für den Benutzer zu jeder Zeit offensichtlich ist, in welchem Dialog, an welcher Stelle im Dialog er sich befindet, welche Handlungen unternommen werden können und wie diese ausgeführt werden können.“ [20]. Bei natürlichsprachlichen Dialogen sollte der Benutzer somit im Falle von Nachrichten des Systems wissen, was für eine Reaktion von ihm verlangt wird und was valide Antworten an das System wären.
Fehlertoleranz wird in der Norm wie folgt definiert: „Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.“ [20]. Dies würde für ein Dialogsystem fordern, dass fehlerhafte Antworten erkannt werden und im besten Fall selbst korrigiert werden oder andernfalls der Benutzer einen Hinweis erhält, worin der Fehler besteht, und er erneut zu einer Eingabe aufgefordert wird. Hierbei sind sowohl Fehler bei den Benutzernachrichten an sich gemeint, zum Beispiel wenn der Benutzer ein Wort falsch verwendet, als auch bei den übermittelten Informationen, beispielsweise wenn ein angegebener Messwert im Kontext nicht möglich ist.
Unter *Lernförderlichkeit* gilt in der Norm: „Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen der Nutzung des interaktiven Systems unterstützt und anleitet.“ [20]. Dies ist jedoch für ein natürlichsprachliches Dialogsystem eher irrelevant, da bei guter Modellierung des Gesprächs nur die Beherrschung der verwendeten Sprache benötigt wird und es keine weiteren Aspekte des Systems zu erlernen gibt, weil diese Sprache das einzige Interaktionsmedium darstellt. Im Gegensatz zu klassischen grafischen Anwendungen gibt es keine Elemente wie Shortcuts oder vergleichbare Vereinfachungen für erfahrene Anwender. Die einzige Möglichkeit, ähnliche Hilfestellungen zur Lernförderung in einen Chatbot einzubauen, ist es, dem Benutzer bei einer Frage einige passende Antwortmöglichkeiten vorzu-

schlagen.
Steuerbarkeit wird mit der folgenden Definition erläutert: „Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.“ [20]. Auch bei natürlich wirkenden Dialog muss Steuerbarkeit zutreffen, da es möglich sein sollte, einzelne Themen in unterschiedlichen Detaillierungsstufen und somit auch in verschiedenen Geschwindigkeiten abzuarbeiten. Dazu sollte es dem Benutzer ermöglicht werden, die Richtung des Dialoges zu verändern, also einen Themenwechsel durchzuführen oder einzelne Themen vorzeitig von sich aus zu beenden, wenn er zum Beispiel die Antwort auf eine Frage nicht kennt.
Weiterhin wird *Erwartungskonformität* definiert als: „Ein Dialog ist erwartungskonform, wenn er den aus dem Nutzungskontext heraus vorhersehbaren Benutzerbelangen sowie allgemein anerkannten Konventionen entspricht.“ [20]. Grundlegend wird Erwartungskonformität erfüllt durch die bereits erwähnten logischen Paare der Intentionen.
Abschließend wird *Individualisierbarkeit* wie folg definiert: „Ein Dialog ist individualisierbar, wenn Benutzer die Mensch-System-Interaktion und die Darstellung von Informationen ändern können, um diese an ihre individuellen Fähigkeiten und Bedürfnisse anzupassen.“ [20]. Dies ist jedoch in modellierten Dialogen schwierig. Es wäre zwar zum Beispiel möglich, die Reihenfolge der bearbeiteten Ziele oder andere Präferenzen durch vergangene Gespräche mit dem Benutzer abzuändern, jedoch wäre dies keine bewusste Individualisierung durch den Benutzer.

2.4 Vorhandene Ansätze für strategische Gesprächsführung durch Dialogsysteme

Nachdem alle wichtigen Aspekte betrachtet wurden, um Nachrichten auswerten zu können und mit welchem Wissen und mit welchen Regeln die Antwort gestaltet werden sollte, werden hier als Teil der Grundlagen noch vorhandene Ansätze vorgestellt, wie das verwaltende Dialogsystem aus der Architektur-Übersicht in Abbildung 2.1 umgesetzt werden kann. Für die Umsetzung des Dialogsystems unterteilt Hamerich vorhandene Ansätze in vier Kategorien: Zustandsbasiert, Regelbasiert, Planbasiert und Statistisch [11].

2.4.1 Zustandsbasierte Dialogsysteme

In einem zustandsbasierten Dialogsystem wird der Dialog durch einen deterministischen endlichen Automaten modelliert [11]. Ein Zustand, also ein Knoten des Graphs, könnte je nach Modellierung einer Antwort des Systems entsprechen und jede Kante, die von diesem Zustand ausgeht, ist eine mögliche Nachricht des Benutzers auf diese System-Antwort, welche den Folgezustand bestimmt.
Da hierbei jede Nachricht des Systems klar definiert ist und auch weiterhin fest

modelliert wird, welche möglichen Antworten auf diese Nachricht durch den Benutzer folgen können, sind solche Systeme sehr statisch. Eine höhere Flexibilität der Dialoge wird nur durch einen sehr hohen Modellierungsaufwand erreicht, indem sehr viele Übergänge für sehr viele mögliche Benutzerantworten angegeben werden [11].
Jedoch sind diese zustandsbasierten Systeme dafür sehr simpel zu implementieren, da nur die Antwort des Benutzers mit allen ausgehenden Kanten des aktuellen Zustandes verglichen werden muss. Sobald eine Kante gefunden wurde, ergibt sich die Nachricht des Systems direkt aus dem Knoten, zu dem die Kante führt. Weiterhin lässt sich in einem solchen System das Gespräch sehr einfach erweitern oder abändern, indem der Dialogmodell-Graph bearbeitet wird, und dabei der Quellcode komplett unverändert bleiben kann. Ebenfalls ist das intuitive Verständnis eines Graphs sehr leicht erklärbar und verständlich, da es nur die Elemente Kante und Knoten gibt. Das führt dazu, dass der Dialog jederzeit durch Domänen-Experten oder andere Mitarbeiter des Dialogsystems mit Hilfe eines graphischen Editors verändert werden kann, ohne dass Erfahrung mit Sprachen wie zum Beispiel XML zur Definition des Graphs benötigt wird.

2.4.2 Regelbasierte Dialogsysteme

Bei einem regelbasierten Dialogsystem (in der Literatur auch manchmal als „frame-based“ oder „slot-based“ zu finden) werden die Antworten des Benutzers nicht für Übergänge in einem Graph betrachtet, sondern als Eingabe für einen gegeben Satz an Vorbedingungen, Regeln und Zielen [11]. Hierbei werden Aspekte zur Laufzeit des Dialoges ausgewertet wie die Erfüllung der Vorbedingungen, das Erreichen der Ziele des Dialoges oder ob Regeln erfüllt wurden, die einen Unterdialog mit eigenen Zielen aktivieren [11].
Ein historisches Beispiel für ein regelbasiertes Dialogsystem ist das Genial Understander System (oder auch kurz: GUS), bei dem von Frames gesprochen wird, welche aus Slots bestehen, die im Laufe des Dialogs gefüllt werden müssen [3]. Wenn beispielsweise ein Datum herausgefunden werden soll, würde der Frame dieses Datums aus den Slots Tag, Monat und Jahr bestehen. Sobald alle Datums-Slots gefüllt wurden, gibt es in GUS für Frames externe Prozeduren zur Überprüfung von Regeln, die in diesem Fall beispielsweise kontrollieren könnten, ob das eingegebene Datum ein valides Datum ist, was einen Unterdialog aktivieren kann und somit dynamische neue Frames hinzufügen würde [3].
Allgemein lässt sich nun der benötigte Modellierungsaufwand verringern, da nicht mehr alle möglichen Nachrichten des Benutzers durch Kanten modelliert werden müssen. Stattdessen wird strukturiert in der Form von Frames mit Slots angegeben, welche Informationen benötigt werden, und durch Prozeduren zur Ermittlung, welche Bedingungen die eingegebenen Werte erfüllen müssen. Hier ist die Füllreihenfolge der Slots aus dem Modell heraus beliebig. Bei einem zustandsbasiertem System benötigen solche flexiblen Dialoge zusätzlichen Modellierungsaufwand, da für jede mögliche Reihenfolge weitere Kanten und Knoten benötigt

werden.
Statt in der Modellierung des Dialoggraphs steckt nun die Arbeit in dem Programmieren der Frames und der Regeln. Wenn sich jedoch zu viele Regeln und Frames für Unterdialoge ergeben, die sich auf mehrere Daten beziehen, wird das entstehende Dialogmodell schnell unübersichtlich und somit schwer wartbar [11].
Im Gegensatz zu einem Graph enthält ein regelbasiertes Dialogsystem keine semantisch relevante Strukturierung der Reihenfolge. Der Zusammenhang der Ziele, der in einem Graph vorhanden ist, geht verloren durch eine Modellierung mittels einzelner losgelöster Ziele und Regeln. Insgesamt werden Dialoge von regelbasierten Dialogsystemen dadurch schwerer vorherzusehen.

2.4.3 Planbasierte Dialogsysteme

In einem planbasierten Dialogsystem geht es nicht darum, im Vorhinein mögliche Gesprächsverläufe vollständig durch Zustände oder durch Regeln zu modellieren, sondern das System zur Laufzeit ein regelmäßig aktualisiertes „mentales Modell“ des Gesprächs bilden zu lassen. Zu diesem Modell gehören dann zum Beispiel Informationen wie das Wissen, das das System vor dem Gespräch hatte, welche Informationen das System durch das Gespräch bereits gewonnen hat oder welche Aktionen in der näheren Zukunft durchgeführt werden könnten. Mit diesem Modell wird das Dialogsystem einen Plan für sein Verhalten ableiten [22].
Traum und Larsson beschreiben vier relevante Komponenten für ein planbasiertes Dialogsystem: Informationsstand, Menge der Dialogaktionen, Updatevorschriften und Updatestrategie [22]. Der Informationsstand enthält alle Informationen, die das Dialogsystem über den aktuellen Stand des Gesprächs hat, also Aspekte wie Verpflichtungen, Pläne, gesammelte Informationen, statisches Domänenwissen und Intentionen [22]. In der Menge aller möglichen Dialogaktionen werden Verhältnisse zwischen Nachrichten an den Benutzer und dazugehörigen Updatevorschriften definiert. Solche Updatevorschriften sind angegeben als kohärente Sammlung von Vorbedingungen an den Informationsstand für das Update und Änderungen an diesem als Folge auf das Update [22]. Zuletzt bedarf es noch einer Updatestrategie, welche die Vorbedingungen von Updatevorschriften überprüft und bei Erfolg einzelne oder eine Sequenz von Dialogaktionen auswählt, nach deren Durchführung die Änderungen der Updatevorschriften angewandt werden [22].
Da erst zur Laufzeit dynamisch entschieden wird, wie das Dialogsystem sich innerhalb der nächsten Nachrichtenwechsel verhalten wird, erscheinen planbasierte Systeme dem Benutzer intelligenter und sie können auch flexibler mit verschiedenen Gesprächsabläufen umgehen [11].
Jedoch sind planbasierte Dialogsysteme schwieriger zu spezifizieren als zustandsbasierte oder regelbasierte Systeme [11]. Ebenfalls stellt das Testen eine neue Art von Herausforderung dar, weil das System bei zwei semantisch äquivalenten Eingaben nicht zwangsläufig gleich reagiert, sondern es vorkommen kann, dass bei einer der beiden Eingabevarianten ein anderer Dialogplan durch die Strategie höher gewichtet wird und somit ab diesem Zeitpunkt der Dialog grundlegend anders

ablaufen wird [11].

2.4.4 Statistische Dialogsysteme

Bei statistischen Dialogsystemen wird im Gegensatz zu den bisher erläuterten Varianten kein Dialogablauf statisch im Vorhinein oder dynamisch zur Laufzeit modelliert, sondern das System erlernt ein Dialogmodell aus einer großen Menge vorhandener Dialogdaten [11]. Ein Bereich, der bei statistischen Dialogsystemen stark erforscht wird, ist das Erlernen von Dialog Strategien in einem Markov Decision Process [16] oder eine speziellere Form von diesem, dem Partially Observable Markov Decision Process [25].
Für das Erlernen einer Gesprächsstrategie in einem Markov Decision Process beschreiben Levin et al. vier wichtige Bestandteile: Aktionen des Systems, Zustände des Systems, Übergangswahrscheinlichkeiten zwischen den Zuständen und eine Kostenverteilung für alle mögkichen Zuständsübergänge [16]. Die vier Bestandteile werden im Folgenden erläutert. Aktionen des Systems sind alle Möglichkeiten, die das System hat, um mit dem Benutzer oder anderen Elementen des Systems zu interagieren. Dazu können zum Beispiel das Stellen einer Frage, die Bestätigen einer Aussage oder die Interaktion mit Ressourcen des Systems wie einer Datenbank zählen. Diese Aktionen sind jedoch nicht wie bei planbasierten Systemen an Regeln und Vorbedingungen geknüpft, sondern es ist jederzeit jede von diesen möglich und wann welche eingesetzt wird, erlernt das System. Weiterhin gibt es Zustände für jeden möglichen Status des Systems, also beispielsweise welche offenen Slots des Gesprächs bereits gefüllt wurden oder andere Systemressourcen. Um nun entscheiden zu können, welche Aktion in jedem Zustand ausgeführt werden sollte, also wie die Gesprächsstrategie ist, gibt es Übergangswahrscheinlichkeiten für einen Zustand s mit seinem Folgezustand s' und einer Aktion a, die wie folgt aussieht $P_T(S(t+1) = s' \mid S(t) = s, A(t) = a)$. Zuletzt gibt es noch eine Kostenverteilung, die beschreibt, welche Kosten c nach einem Zustandsübergang erwarteter Weise noch entstehen werden bis ein Zielzustand erreicht wird, in der Form $P_C(C(t) = c \mid S(t) = s, A(t) = a)$. Die Definition von $C(t)$ ist hierbei variabel und lässt sich gut einsetzen, um das strategische Verhalten des Dialogsystems beim Lernen in eine Richtung zu lenken. Wenn zum Beispiel gefordert wird, dass die Dialoge mit den Benutzern möglichst schnell abgeschlossen werden sollen und dabei mögliche Missverständnisse und Fehler in Kauf genommen werden, wäre eine Möglichkeit für die Kosten eines Zustandes die Entfernung zum nächsten Zielzustand [16]. Durch diese vier Bestandteile entsteht ein Optimierungsproblem, bei dem durch die Anpassung der Übergangswahrscheinlichkeiten die zu erwartenden Kosten gesenkt werden sollen. Um diese Entwicklung und Optimierung einer Strategie zu automatisieren, bieten sich vorhandene Algorithmen aus der Disziplin des Reinforcement Learnings an. Diese beschäftigen sich in erster Linie mit Modellen, in denen die Menge der Zustände nicht bekannt ist, wodurch weiterer Modellierungsaufwand wegfällt [16].
Laut Littman ist es jedoch in der Regel nicht der Fall, dass ein System, das auf

einem Markov Decision Process basiert, alle nötigen Eigenschaften seines Umfelds automatisch kennt, also in diesem Fall den Status des Gesprächs, sondern es erst Aktionen ausführen muss, um diese Informationen zu erhalten. Hier wird dann das angepasste Modell der Partially Obersvable Markov Decision Processes benötigt, bei denen diese Unwissenheit berücksichtigt wird [17]. Partially Obersvable Markov Decision Processeses wurden erfolgreich für Dialogsysteme unter anderem von Williams und Young eingesetzt [23].
Ein Vorteil ist, dass in solchen statistischen Dialogsystemen der Modellierungs- oder Spezifizierungsaufwand der vorherigen Ansätze größtenteils wegfällt, da keine festen Zustandsübergänge, Regeln oder Updatevorschriften im Vorhinein entworfen werden müssen. Sobald die möglichen Aktionen und die Kostenverteilung für die Markov Decision Processes modelliert und die darauf arbeitenden Lernverfahren implementiert sind, genügen Daten zum Trainieren und zum Testen, um ein Dialogsystem mit einer zielführenden Strategie generieren zu können [11].
Zusätzlich sind solche Dialogsysteme häufig sehr einfach zu verallgemeinern und wiederzuverwenden, da die meisten Aktionen in vielen verschiedenen Dialogdomänen nützlich sind und diese somit nur leicht bearbeitet werden müssen und daneben nur neue Trainingsdaten benötigt werden, um eine Strategie für eine andere Art von Dialog entwickeln zu können [16].
Dafür liegt das gravierende Hauptproblem bei solchen Verfahren in der großen Menge der benötigten Trainingsdaten, welche zuerst gesammelt werden müssen [11].

2.4.5 Bewertung der vorhandenen Ansätze

Zustandsbasierte Dialogsysteme sind leicht zu verstehen und das Erstellen leicht erlernbar. Jedoch ist es nicht möglich die Reihenfolge der Gesprächsthemen variabel zu gestalten oder den Übergängen zwischen den Zuständen nur unter definierten Bedingungen zu folgen, ohne sehr viele Kanten und Knoten zu verwenden. Für größere oder komplexere Dialoge sind zustandsbasierte Dialogsysteme somit nicht geeignet.
Bei Regelbasierte Dialogsysteme können hingegen können einfach Bedingungen für Gesprächsthemen eingeführt werden und die Dialoge sind sehr flexibel, da es keine definierte Reihenfolge gibt. Jedoch ist das Aufstellen der Regeln komplex und wird mit steigender Anzahl der Regeln unübersichtlich, weil die Regeln und Frames nur definiert werden und kein Zusammenhang unmittelbar erkennbar wird. Deshalb sind regelbasierte Systeme ebenfalls nicht geeignet für größere oder komplexere Dialoge, da die Definition des Systems nicht strukturiert genug ist.
Planbasierte Dialogsysteme wären in der Lage größere und komplexere Dialoge zu absolvieren, jedoch erfordert das Definieren der Updatevorschriften und -strategien deutlich mehr Arbeit als das Aufstellen von Modellen bei den beiden zuvor erläuterten Ansätzen. Weiterhin wird der tatsächliche Ablauf des Dialoges erst während des tatsächlichen Dialoges dynamische entworfen und ist somit nicht komplett korrekt vorhersagbar. Für riskante Domänen wie zum Beispiel die Medizin ist ein solches nicht deterministisches Verhalten nicht anwendbar.

Die statistischen Dialogsysteme wären ebenfalls geeignet größere oder komplexere Dialoge durchzuführen, aber es ist nicht in jeder Domäne problemlos möglich genug Trainingsdaten zu sammeln, um dem Dialogsystem ein realistisches Modell von solchen Dialogen anzutrainieren. Durch die zugrundeliegende Stochastik des Verhaltens des Dialogsystems, ist es ebenfalls nicht geeignet für riskante Domänen, weil zu gewissen Wahrscheinlichkeiten fehlerhafte Aktionen durchgeführt werden können.

2.5 Zusammenfassung der theoretischen Grundlagen

Für die Implementierung im weiteren Verlauf dieser Arbeit werden Frameworks verwendet, in denen die vorgestellten Verfahren zur maschinellen Auswertung von natürlicher Sprache implementiert wurden. Wissensgraphen und Ontologien werden zur Unterstützung der Dialogsstrategie mit Allgemein- und Domänenwissen eingesetzt. Der Aspekt der natürlichen Eigenschaften von Dialogen zur Steigerung der Ergonomie wird sowohl bei der Implementierung als auch bei der Modellierung eines Beispieldialogs relevant sein und zusätzlich im Evaluations-Kapitel mit diesem Beispieldialog empirisch überprüft. Im Bereich der Ansätze für strategische Dialogsysteme ist der Ansatz dieser Arbeit ein Hybrid aus einem zustandsbasierten und regelbasierten System und enthält Ideen und Funktionsweisen aus beiden Varianten. Die planbasierten und statistischen Dialogsysteme wurden zusätzlich beschrieben, um einerseits einen Überblick über den Stand der Forschung zu geben, aber auch, um eine Abgrenzung zu dem Ansatz dieser Arbeit zu ermöglichen. Eingesetzt werden Verfahren aus diesen Bereichen für diese Arbeit nicht, da Domänen-Experten Gespräche modellieren sollen, auf die sie direkten Einfluss haben und an die sich der Chatbot unmittelbar hält, ohne ihm ein geändertes Verhalten erst durch viele Trainingsdaten heuristisch beizubringen, was in Domänen wie zum Beispiel der Medizin ohne direkten Einfluss durch Experten zu riskant wäre.

2.6 Verwandte Arbeiten

Innerhalb dieser Arbeit wird ein Dialogsystem entwickelt und dessen Einsatz anhand eines medizinischen Beispiels gezeigt. Für Dialogsysteme in dieser Domäne gibt es bereits verschiedene erfolgreich getestete Ansätze, von denen drei mit einer ähnlichen Grundidee im Folgenden kurz vorgestellt und von dieser Arbeit abgegrenzt werden.

In ihrem Artikel „Whats's Up, Doc? A Medical Diagnosis Bot“ beschreiben Agrawal et al. ein Dialogsystem, welches den menschlichen Dialogpartner zuerst nach Alter, Geschlecht und Symptomen fragt, um ihm dann, unterstützt von einer externen Diagnose-Engine, eine mögliche Krankheit als Verursacher der Symptome

vorzuschlagen [1]. Der Grundgedanke sowohl bei ihrem als auch dem Ansatz dieser Arbeit ist es, im Vorhinein durch einen Graph modellierte Daten zu erfragen und daraufhin, in Abhängigkeit von diesen Daten, dem Benutzer ein Feedback zu geben. Das Dialogmodell in ihrem Ansatz wird strikt durch einen endlichen Automaten, also ein zustandsbasiertes Verfahren, modelliert. Es ist somit nicht möglich, von der modellierten Reihenfolge der Dialogthemen abzuweichen, also zum Beispiel Geschlecht vor dem Alter. Der Ansatz in dieser Arbeit hingegen wird regelbasierte und zustandsbasierte Verfahren kombinieren und dadurch solche Themenwechsel unterstützen, welche abhängig von Regeln innerhalb des Graphs springen, obwohl keine Kante zwischen dem aktuellen Knoten und dem Sprungziel existiert. Dafür hat die Vorgehensweise aus dem Artikel einen besseren Ansatz zur Analyse der Benutzer-Nachrichten, da extrahierte Entitäten mit Einträgen aus der lexikalischen Datenbank „WordNet“ und ebenfalls anhand der Kosinus-Ähnlichkeit der Vektor-Repräsentation der Entität mit vorhandenen Vektoren aus dem Datensatz von Wort-Vektoren „GloVe“ verglichen werden, um durch beide Quellen Synonyme und verwandte Wörter zu finden, falls die Diagnose-Engine die ursprüngliche Eingabe nicht kennt. Durch diese Vorgehensweise wird die Möglichkeit des Benutzers, sich variabel auszudrücken, ohne dass die Anzahl der Trainingsdaten für die Extraktionsmethoden durch maschinelles Lernen entsprechend erhöht werden muss, stark gesteigert. Von einer solchen Vorgehensweise wird jedoch im Rahmen dieser Arbeit abgesehen, obwohl eine Integration zukünftig denkbar wäre.

Fischer und Lam erläutern in ihrem Artikel „From Books to Bots: Using Medical Literature to Create a Chat Bot“ einen weiteren Ansatz zur medizinischen Betreuung durch einen Chatbot. Bei ihnen ist das Dialogmodell ein Graph, bestehend aus den Konzepten Symptom, Diagnose und Behandlung, wobei der Benutzer nach einigen Symptomen gefragt wird und ihm dann eine Diagnose sowie einige dazu passende Behandlungen vorgeschlagen werden [7]. Ähnlich wie bei der zuvor erwähnten Arbeit ist bei Fischer und Lam die Grundlage des Dialoges ein endlicher Automat, durch den Symptome in einem Dialog besprochen werden. Im Gegensatz zu der Arbeit von Agrawal et al. ist das Domänenwissen jedoch nicht in ein externes Programm ausgelagert, sondern direkt in das Dialogmodell integriert, also die Verlinkung von Symptomen zu einer Diagnose und von einer Diagnose zu möglichen Behandlungen. Für das Erstellen und die Verlinkung bietet der Ansatz von Fischer und Lam zusätzlich eine grafische Oberfläche als Editor, um ein zugängliches Modellieren durch Domänenexperten oder Crowdsourcing zu ermöglichen. Dafür ist die Interaktion mit dem Patienten während des Dialoges deutlich eingeschränkter als beim vorherigen Ansatz von Agrawal et al., da die möglichen Symptome nur mit Ja oder Nein beantwortet werden, bis der Chatbot weit genug im Graph vorgedrungen ist, damit eine eindeutige Diagnose gestellt werden kann. Ähnlich wie das Vorgehen von Fischer und Lam wird der Ansatz dieser Arbeit es ermöglichen Domänenwissen direkt in das Dialogmodell zu integrieren, jedoch auch die Möglichkeit bieten, es wie bei Agrawal et al. in externe Services auszulagern. Weiterhin soll es dem Benutzer ermöglicht werden, frei zu

formulieren und sie nicht auf Ja oder Nein als Antwort zu begrenzen. Von der Erstellung eines solchen zugänglichen grafischen Editors für die Dialogmodellierung wird innerhalb dieser Arbeit abgesehen. Es wäre jedoch ein guter Ansatz für eine spätere Erweiterung.

Neben den beiden faktenorientierten Ansätzen zur Diagnose von Agrawal et al. und Fischer und Lam gibt es in der medizinischen Forschung auch Ansätze zur psychologischen Betreuung von Patienten. In ihrer Studie „Delivering Cognitive Behavior Therapy to Young Adults With Symptoms of Depression and Anxiety Using a Fully Automated Conversational Agent (Woebot): A Randomized Controlled Trial“ [8] untersuchten Fitzpatrick et al. den Chatbot *Woebot* bezüglich seiner therapeutischen Wirkung auf Jugendliche, welche an Depressionen oder Angststörungen leiden. Hierbei konnte *Woebot* einer signifikanten Gruppe der Teilnehmer helfen, ihre Symptome zu reduzieren. Aus technischer Perspektive ist *Woebot* durch einen Entscheidungsbaum realisiert, der bei einzelnen Knoten durch eingebettete Sprachauswertungsprozeduren dynamisch zu anderen Knoten wechseln kann. Dieses Vorgehen, einem Entscheidungsbaum dynamische Sprünge zwischen den Dialogknoten zu ermöglichen, ist ein großer Unterschied bezüglich der Flexibilität der geführten Dialoge gegenüber den beiden zuvor erläuterten Ansätzen, bei denen das Dialogverhalten auf den Zustandsübergängen von statischen Automaten basiert. Es ähnelt somit dem Ansatz dieser Arbeit, mit dem Unterschied, dass dynamische Sprünge in dem Ansatz dieser Arbeit nicht nur durch Sprachauswertung ausgelöst werden können, sondern durch beliebige Bedingungen. Weiterhin verhält sich *Woebot* nicht zielorientiert. Der Chatbot hat zwar ebenso wie der Ansatz dieser Arbeit die meiste Zeit über die Gesprächsführung inne, versucht aber eher den menschlichen Benutzer zum Reden zu bringen und ihm durch Hinweise oder kleinere interaktive Dialogelemente weiterzuhelfen, solange der Benutzer dies wünscht. Dabei gibt es kein finales Resultat oder Ziel eines Prozesses, was der Chatbot und der menschliche Benutzer letztendlich gemeinsam erreichen wollen, worauf im Gegensatz dazu der Ansatz dieser Arbeit ausgelegt ist.

3 Entwicklung und Modellierung von Gesprächszielen und -strategien

Nachdem in den Grundlagen erläutert wurde, wie man wichtige Informationen aus den Nachrichten des Benutzers extrahiert und wie die Antworten des Chatbots darauf gestaltet sein sollten, werden in diesem Kapitel Konzepte beschrieben, um Gesprächsziele für einen Chatbot zu modellieren. Dazugehörig wird ebenfalls beschrieben wie eine Gesprächsstrategie aussehen muss, um diese Ziele zu erreichen, damit ein wie in der Einleitung beschriebener Chatbot implementiert werden kann.
Zuerst wird nun beschrieben, wie sich das Dialogsystem, dessen Modell und Strategie in diesem Kapitel entworfen werden, in die vorhandenen Ansätze einordnen lässt. Anschließend wird erläutert, in welcher Form und mit welchen Daten dieses Modell definiert wird, um alle Gesprächsziele darstellen zu können, sowie eine konkrete Syntax vorgestellt, mit der Domänen-Experten Gespräche in dieser Form modellieren können. Zuletzt wird eine Strategie für Dialoge dargestellt, mit der einer Reihe von Benutzernachrichten in eben beschriebener Form die modellierten Ziele erreichen kann.

3.1 Einordnung dieses Ansatzes für Gesprächsmodelle und -strategien

Wie in Abschnitt 2.4 bereits erläutert kann das interne Verhalten eines Dialogsystems als zustandsbasiert, regelbasiert, planbasiert oder statistisch eingeordnet werden. Der Ansatz dieser Arbeit vereint Ideen aus zustandsbasierten und regelbasierten Dialogsystemen, um die Vorteile beider zu erhalten, aber auch die jeweiligen Nachteile gegenseitig auszugleichen. Durch die strukturierte Modellierung eines zustandsbasierten Modells, wird eine gute Verständlichkeit und Erweiterbarkeit des Modells für Domänenexperten erreicht. Sie verhindert jedoch, dass der Dialog dynamischer werden kann, wie zum Beispiel bei der Reihenfolge von Dialogthemen, ohne zusätzliche Übergänge manuell modellieren zu müssen. Regelbasierte Modellierung hingegen hat keine Anforderungen an Reihenfolge und Zusammenhänge zwischen den Frames, die gefüllt werden müssen, sondern nur Regeln dafür, unter welchen Bedingungen Frames aktiviert werden und welche

Werte in die Frames eingetragen werden dürfen. Diese Regeln können jedoch bei einem größerem Dialog sehr komplex und unstrukturiert werden, was die Modellierung durch Domänenexperten schwierig und aufwendig macht. Wenn nun, wie in diesem Ansatz, ermöglicht wird, einen großen Satz an Frames und Regeln aufzuteilen und die einzelnen Teile einem semantisch passenden Kontext strukturiert zuzuordnen, in dem sie gelten und definiert werden, wird dieses Problem der Komplexität und Unübersichtlichkeit auf ein Minimum reduziert. Da die Frames und Regeln in die geordnete Strukturierung eines Graphs eingebettet werden, wird die leichte Bearbeitung eines zustandsbasierten Ansatzes benutzt, um den zustandsbasierten Ansatz durch die Dynamik eines regelbasierten Ansatzes zu erweitern. Hiermit können in der Strategie Zustandsübergänge ermöglicht werden, ohne dass eine entsprechende Kante existieren muss, dafür jedoch eine zutreffende Regel erfüllt wird. Durch diese Möglichkeit einer einfachen Modellierung eines dynamischen Gespräches, bei dem das Verhalten jedoch im Gegensatz zu statistischen Dialogsystemen deterministisch wie zuvor modelliert ist, kann man auch in riskanten Domänen wie der Medizin komplexe Dialoge effizient modellieren, bei denen die komplette Kontrolle behalten wird.

3.2 Benötigte Daten für die Modellierung von Gesprächszielen

Um die Vorstellung eines Gesprächszieles besser verbildlichen zu können, wird in diesem Kapitel zum Vergleich eine bereits vorhandene grafische Anwendung vorausgesetzt. Solche grafische Anwendungen bestehen aus einer Reihe von Eingabe-Formularen, die nun in ein Gesprächsmodell mit Gesprächszielen übertragen werden sollen. Das Ziel einer solchen Anwendung wäre es, dass der Benutzer die Eingabe-Felder korrekt ausfüllt, was sich auch auf einen Dialog mit einem Chatbot übertragen lässt. Ein Chatbot-Dialog hat ebenfalls als übergeordnetes Ziel, dass am Ende alle gestellten Fragen vom Benutzer beantwortet wurden, damit der Chatbot passend reagieren kann. Dies muss er durch eine geeignete Strategie erreichen, da die feste und lineare Bearbeitungsreihenfolge einer grafischen Oberfläche fehlt. Allgemein lassen sich solche Ziele natürlich auch frei modellieren, ohne dass eine grundlegende Anwendung als Basis existiert.

Definition 3.1 *Ein* Gesprächsziel *oder auch* Ziel *ist eine Konstellation von Metadaten, die ein Chatbot benötigt, um innerhalb eines freien Gesprächs einen Satz von semantisch zusammenhängenden Informationen vom Benutzer zu erfragen.*

Angenommen als Teil der hypothetischen Anwendung existiert ein Formular, auf dem der Benutzer Angaben zu seinem Gewicht machen soll, dann könnte das Formular zum Beispiel ein Eingabefeld für das Gewicht und ein weiteres Eingabefeld für den Zeitpunkt, wann der Benutzer dieses Gewicht gemessen hat, enthalten.

Weiterhin verfügt die Anwendung noch über Funktionalitätslogik. Mit der Funktionalitätslogik kann erstens überprüft werden kann, ob der eingegebene Wert ein gültiges Datum ist, und zweitens je nachdem, ob das eingegebene Gewicht eine gewisse Grenze überschreitet oder nicht, unterschiedliche Formulare als Nächstes zur Bearbeitung angezeigt werden.
Aus dieser Beschreibung einer solchen einfachen grafischen Anwendung kann man bereits mehrere relevante Bestandteile der Modellierung von Gesprächszielen ableiten. Weitere Bestandteile kommen hinzu, sobald Chatbots betrachtet werden. Beispiele dafür sind die Handhabung der Intentionen des Benutzers oder Definitionen, unter welchen Bedingungen Themenwechsel erfolgen sollen. Diese sind für grafische Anwendungen irrelevant und haben nur in einem freien Dialog Bedeutung. Alle diese Modellierungsaspekte eines Gesprächsziels werden nun einzeln näher erläutert.

3.2.1 Einleitender Text

Ein grafisches Formular hat im Normalfall nicht einfach nur die benötigten Eingabefelder, sondern auch eine passende Überschrift, einen einleitenden Text, eine übergeordnete Frage oder ähnliches, sodass der Benutzer weiß, was dieses Formular insgesamt von ihm verlangt und wie es im Zusammenhang der gesamten Anwendung einzuordnen ist. Auch ein Chatbot sollte deshalb eine oder mehrere solcher Nachrichten angegeben bekommen, die er abschickt, sobald mit der Bearbeitung des Ziels begonnen wird. Wenn zum Beispiel anstatt einer Nachricht wie „Gewicht?“besser etwas gesendet wird wie „Als nächstes werden Fragen zu ihrem Körpergewicht gestellt. Wissen sie ihr aktuelles Gewicht?“, ist der Benutzer besser darüber informiert, was aktuell von ihm erwartet wird und kann passend reagieren.

3.2.2 Intentions-Reaktionen

Auf eine relativ offene Frage wie das zuvor erwähnte „Wissen Sie Ihr aktuelles Gewicht?“ gibt es mehrere mögliche Antwort-Intentionen, mit denen der Benutzer logisch korrekt antworten könnte. So wäre zum Beispiel eine Verneinung, weil er sein Gewicht nicht kennt, genauso möglich wie Antworten in dem Fall, dass er sein Gewicht kennt, wie „Ja“ oder die tatsächliche Angabe seines konkreten Gewichtes wie „Ich wiege 80 Kilogramm“. Wenn nun mehrere Intentionen von Seiten des Benutzers logisch möglich und auch zu erwarten sind, müssen für jede dieser Intentionen geeignete Reaktion des Chatbots angeben werden, um die in Abschnitt 2.1 erwähnten logischen Paare aus Intentionen beider Gesprächsteilnehmer modellieren zu können. Als Reaktionen werden hier konkrete Typen angegeben, welche einem Verhalten des Chatbots entsprechen, wie zum Beispiel ein Gesprächsthema zu beenden, detaillierter nachzufragen oder die Antwort zu akzeptieren und zu speichern.

3.2.3 Erklärungen/Hilfe-Funktionen

Das grafische Formular muss verhindern, dass der Benutzer nicht weiß, wie er mit der Anwendung zu interagieren hat. Dazu könnte es zum Beispiel seinen Eingabefeldern in der Form von Tooltipps mehr abrufbare Informationen hinterlegen oder einen Button hinzufügen, der einen weiteren Hilfe-Dialog öffnet. Bei einem Chat ist die Interaktion an sich durchgehend gleich, nämlich eine empfangene Nachricht lesen und eine passende Antwort schicken. Jedoch können auch hier Szenarien auftreten, in denen der Benutzer zum Beispiel ein Wort einer Nachricht nicht kennt oder sogar die gesamte Aussage der Nachricht nicht versteht. Je nach konkretem Ziel kann es eine passende und hilfreiche Reaktion sein, mit einer erklärenden Nachricht auf manche Intentionen des Benutzers zu reagieren. Hier kommen mehrere Intentionen in Frage wie zum Beispiel eine Aussage, dass er etwas nicht verstanden hat, oder wenn er bei einer wichtigen Frage angibt, dass er die Antwort nicht kennt. Es kann dann versucht werden mit einer erklärenden Hilfeantwort ein besseres Verständnis davon beim Benutzer zu erzeugen, worum es geht oder den Rahmen möglicher Antworten einzugrenzen. Um nun also bei der Modellierung von Zielen als Reaktionstyp auf manche Intentionen des Benutzers das Absenden einer Erklärung zu ermöglichen, sollte jedem Ziel ein detaillierter Erklärungstext beigefügt werden. Mit einer solchen Methodik wird die Selbstbeschreibungsfähigkeit aus den Grundsätzen der Dialogsteuerung aus Abschnitt 2.3 verbessert.

3.2.4 Überspringbare Ziele

Die übermittelten Informationen mancher Ziele sind je nach Szenario zwar praktisch in Erfahrung zu bringen, jedoch theoretisch auch verzichtbar für eine erfolgreiche Durchführung des Dialogs. In einem solchen Fall wäre es sehr unnatürlich, wenn der Chatbot immer wieder die Fragen zu einem Ziel stellt, obwohl der Benutzer nicht darauf eingeht, das Thema wechselt oder ähnliches, obwohl die eventuell eher geringe Wichtigkeit der Information eine solche Aufdringlichkeit nicht rechtfertigt. Für solche Situationen wäre es praktisch, bei den Zielen eine Wahrscheinlichkeit anzugeben, ob die Bearbeitung des Ziels abgebrochen werden sollte, falls ein solches Szenario eintritt und der Benutzer mehr oder weniger bewusst nicht auf das Ziel eingeht. Mit einer solchen Wahrscheinlichkeit für jedes Ziel kann die Wichtigkeit modelliert werden, bei der dann zum Beispiel 0% bedeutet, dass das Ziel so wichtig ist, dass es in gar keinem Fall übersprungen werden darf und immer weiter gefragt wird, während höhere Prozentzahlen bedeuten, dass es immer weniger schlimm wäre, wenn die Informationen des Ziels nicht erhalten werden und das Bearbeiten des Ziels bei entsprechendem Benutzerverhalten abgebrochen werden kann.

3.2.5 Benötigte Informationen

Die Kernelemente eines Gesprächszieles sind die Informationen, die vom Benutzer erfragt werden sollen. Solche gesuchten Informationen würden den konkreten Eingabefeldern der grafischen Eingabemaske entsprechen. Hierzu werden Entitäten aufgezählt, die benötigt werden, und ihre jeweilige Klasse der NER. Mit solchen Informationen kann die Übereinstimmung überprüft werden, ob die strukturierten Daten der aktuellen Benutzernachricht oder einer Nachricht, die zu einem früheren Zeitpunkt geschickt wurde, aber bisher nicht passend verwendet werden konnte, dieses Ziel komplett oder nur teilweise erfüllen. Zum Beispiel, wenn für ein Ziel eine Zahl als Gewicht und eine Zeitangabe, wann dieses gemessen wurde, gefordert sind, kann so erkannt werden, dass die Informationen „80 Kilogramm“ vom Typ Gewicht und „Gestern“ vom Typ Zeitangabe dieses Ziel direkt erfüllen können.
Da nicht immer davon auszugehen ist, dass ein Benutzer alle Informationen, die zu einem Ziel gehören, auf einmal innerhalb einer Nachricht äußert, sollte zu jeder benötigten Information auch eine oder mehrere mögliche Nachrichten hinterlegt sein, mit der konkret nach dem Wert dieser Information gefragt werden kann. Im weiteren werden diese benötigten Informationen eines Ziels als die Slots eines Ziels bezeichnet, in Anlehnung an die Slots eines Frames des regelbasierten Hintergrunds dieses Systems.

Definition 3.2 *Ein* Slot *eines Ziels ist eine vom Benutzer benötigte Information, die in den inhaltlichen Kontext des Ziels passt, und worin innerhalb des Gesprächs ein geeigneter Wert gespeichert wird.*

3.2.6 Korrektheit von Informationen

Es ist möglich, sowohl bei einem Eingabefeld einer grafischen Oberfläche als auch einem Slot eines Gesprächsziels direkt Beschränkungen anzugeben, wann eine Benutzerangabe valide ist und mit ihr weiter fortgefahren werden kann oder wann sie nicht gültig ist und es sich um einen Tippfehler, eine fehlerhafte Erinnerung des Benutzers oder Ähnliches handeln muss. Im weiteren Verlauf wird bei diesen Überprüfern der Korrektheit von Revisoren gesprochen.

Definition 3.3 *Ein* Revisor *eines Slot überprüft die Korrektheit von möglichen Werten mit einer vorgegebene Bedingung und entscheidet so, ob der Wert in dem Slot gespeichert werden darf.*

Angenommen die Nachricht des Benutzers enthält das Wort oder die Phrase x, welches durch die NER der Klasse zugeordnet wird, die ein aktueller aktiver Slot erwartet. Zusätzlich hat dieser Slot einen Revisor angehängt bekommen. Ein solcher Revisor hat eine prädikatenlogische Bedingung $R(x)$, und nur, wenn diese Bedingung mit der Eingabe x als wahr ausgewertet wird, darf der Slot mit x gefüllt werden. Es ergibt sich also eine Menge an validen Eingaben für diesen Slot.

Wenn zum Beispiel nach einem Geburtsdatum gefragt wird, so kann bei einem Datum, das in der Zukunft liegt, durch einen Revisor direkt von einem Fehler des Benutzers ausgegangen werden. In einem solchen Fall sollte der Chatbot diesen übermittelten Wert ignorieren und sowohl mit einem Hinweis antworten, welches Problem die letzte Nachricht des Benutzers hatte, als auch seine ursprüngliche Frage zu der benötigten Information wiederholen. Je nach Szenario kann es weiterhin auch hilfreich sein, bei korrekten Informationen ebenfalls eine spezielle Nachricht zur Bestätigung oder zu vergleichbaren Zwecken zu schicken. Von Revisoren werden somit die konkreten Beschränkungen und eine oder mehrere mögliche Nachrichten für den fehlerhaften beziehungsweise fehlerfreien Fall benötigt. Sie sind eine Möglichkeit, die Regeln des regelbasierten Teils des Modells einzubauen und werden hier benutzt, um daneben zusätzlich den Aspekt der Fehlertoleranz der DIN EN ISO 9241 zu berücksichtigen, der in Abschnitt 2.3 beschrieben wurde.

3.2.7 Beziehungen zwischen Zielen

Durch das Gespräch sollte ein roter Faden erkennbar sein und Ziele, die innerhalb der Domäne zusammenhängen, sollten auch direkt nacheinander besprochen werden, sofern der Benutzer nicht das Thema von sich aus wechselt, worauf in Abschnitt 3.2.9 eingegangen wird. Es soll also möglich sein, dass Gesprächsziele eine Beziehung untereinander haben und die Knoten dieser Ziele einen oder mehrere nachfolgende Knoten haben können. Für den roten Faden wird ähnlich einer Tiefensuche zuerst ein einzelner nachfolgender Knoten und gegebenenfalls seine nachfolgenden Knoten komplett bearbeitet, bevor andere offene Knoten betrachtet werden. Dies wird getan, da nachfolgende Knoten eine Fortführung beziehungsweise alternativ eine Verfeinerung des Themas des Ziels ihres Vorgängers sein sollten, und dieser Themenkomplex als erstes vollständig bearbeitet werden sollte, anstatt andere Themenkomplexe zwischendurch teilweise anzusprechen und den Benutzer durch fehlenden Kontext zu verwirren. Je nach konkretem Gesprächskontext ist es ebenfalls möglich, dass nachfolgende Ziele nur relevant werden, wenn ein Ziel mit einer bestimmten Intention des Benutzers beantwortet wurde. So kann es zum Beispiel für einen Dialog relevant sein, dass ein Ziel nur besprochen wird, wenn eine Frage, die zuvor gestellt wurde, verneint wurde. Diese Vernetzung von Zielen ist der zustandsbasierte Aspekt der Kombination aus zustandsbasierten und regelbasierten Modellen von dem Ansatz dieser Arbeit.

3.2.8 Auswirkung auf den weiteren Gesprächsverlauf

Der Vernetzung der Gesprächsziele sollte das Dialogsystem in einigen Fällen aber nicht einfach ohne Einschränkungen folgen, da manche Ziele nur relevant werden, wenn bei einem anderen Ziel die übermittelten Daten vorher definierte Bedingungen erfüllen. So ist es auch bei grafischen Oberflächen möglich, dass manche Eingabemasken optional sind und nur in manchen Fällen angezeigt werden, und ein solches Verhalten muss auf die Modellierung des Gesprächs übertragbar sein.

Es ist zum Beispiel vorstellbar, dass es innerhalb der Domäne nur Sinn ergibt, nach der Ernährung des Benutzers zu fragen, wenn seine medizinischen Werte gegebene Schwellenwerte überschreiten und die Informationen zur Ernährung ansonsten irrelevant sind und das Gespräch nur unnötig in die Länge ziehen. Damit solche Zusammenhänge modellierbar sind, sollte es möglich sein, den Slots eines Ziels Bedingungen anzuhängen, in welchen Fällen nachfolgende Ziele des aktuellen Ziels für den Dialog aktiviert werden sollen. Diese Bedingungen, in welchen Fällen nachfolgende Ziele aktiviert werden, wurden Aktivatoren benannt.

Definition 3.4 *Ein* Aktivator *eines Slots überprüft mit Hilfe einer vorgegebenen Bedingung, ob der in dem Slot gespeicherte Wert neue Gesprächsziele aktiviert und somit dem aktuellen Gespräch als mögliche Themen hinzufügt werden.*

Aktivatoren haben analog zu Revisoren ebenfalls eine prädikatenlogische Bedingung $A(y)$ für den bereits gespeicherten Inhalt y des Slots, dem der Aktivator angehängt. Weiterhin enthält ein Aktivator Antworten an den Benutzer für die Fälle, dass die Bedingung erfüllt ist oder nicht, vergleichbar mit dem Revisor. Falls ein Ziel Z mehrere Slots $S_1, ..., S_n$ mit jeweiligen Aktivatoren $A_1, ..., A_n$ hat, werden die nachfolgenden Ziele von Z auch nur aktiviert und somit vom Dialogsystem in Betracht gezogen, falls alle diese Aktivatoren ihre Bedingung $A_n(y_n)$ mit den übermittelten Daten y_n erfüllen. Selbst wenn keine nachfolgenden Gesprächsziele vorhanden sind, kann man diese Aktivatoren benutzen, um in Abhängigkeit von den übermittelten Informationen differenzierte Antworten an den Nutzer zu schicken, je nachdem ob die Bedingung des Aktivators erfüllt ist oder nicht. Aktivatoren sind somit die zweite Möglichkeit, die Regeln des regelbasierten Teils des Modells einzubauen.
Revisoren und Aktivatoren sind zwar ähnlich definiert, also aus einer prädikatenlogischen Bedingung an ihren Slot und einem Satz an Nachrichten, sie haben jedoch einen sehr unterschiedlichen Einfluss auf das Dialogmodell und werden deshalb im Folgenden voneinander abgegrenzt. Ein Ziel Z hat die Slots $S_1, ..., S_n$, und die haben jeweils einen Revisor mit Bedingung R_k und einen Aktivator mit Bedingung A_k. Durch die NER wurde eine Entität x gefunden, die durch ihre NER-Klasse für Slot S_i in Frage kommt. Mit $R_i(x)$ wird nun entschieden, ob x eine valide Eingabe für S_i ist. Bei einer korrekten Auswertung wird x in S_i gespeichert und die Bearbeitung des Slots abgeschlossen. Andernfalls bleibt der Slot in Bearbeitung, und der Benutzer wird weiterhin zu diesem befragt. Sobald nach möglicherweise mehreren Nachrichten alle Slots $S_1, ..., S_n$ mit korrekter Auswertung durch die entsprechende Revisor-Bedingung $R_1, ..., R_n$ einer Entität $y_1, ..., y_n$ befüllt wurden, werden alle Aktivatoren überprüft. Nur wenn bei jedem Slot $S_1, ..., S_n$ die Bedingung $A_1, ..., A_n$ mit der entsprechenden bereits gespeicherten Entität $y_1, ..., y_n$ als wahr ausgewertet wird, werden nachfolgende Ziele von Z aktiviert.

3.2.9 Auslöser für einen Themenwechsel

In einem freien Gespräch kann es jederzeit zu einem Themenwechsel von Seiten des Benutzers kommen, wenn er zum Beispiel anstatt seines INR-Wertes (eine Messgröße für die Blutgerinnung) lieber zuerst über sein Gewicht reden möchte. Um auf einen solchen Themenwechsel reagieren zu können, braucht ein Ziel Schlüsselwörter, die erkennen lassen, dass der Benutzer sich gerade zu diesem Ziel äußert. In dem Fall würde das Dialogsystem zu diesem Knoten im Ziel-Graph springen und mit diesem Ziel weiter fortfahren. Solche Schlüsselwörter könnten beim Beispiel des Gewichts unter anderem sein: „Gewicht“, „Kilogramm“ und „Kg“. Damit solche Schlüsselwörter nicht versehentlich innerhalb des Gespräches verwendet werden, wird als Kontrollmechanismus überprüft, ob das Schlüsselwort auch durch die NER in eine erwartete Klasse kategorisiert wurde. Dadurch werden Fälle vermieden, in denen ein Schlüsselwort zwar auftaucht, aber durch einen anderen Kontext innerhalb der Nachricht eine andere Bedeutung hätte, die keinen Themenwechsel auslösen sollte. Durch diese Möglichkeit des freien Wechseln des Themas wird die Steuerbarkeit aus den Grundsätzen der Dialogsteuerung aus Abschnitt 2.3 verbessert. Im weiteren Verlauf werden solche Auslöser für Themenwechsel als Trigger bezeichnet.

Definition 3.5 *Ein* Trigger *eines Ziels ist eine Angabe von Wörtern oder Phrasen, bei welchen ein Themenwechsel zu dem Ziel durchgeführt wird, sobald der Benutzer sie in einem passenden Kontext verwendet.*

3.2.10 Angabe von Informationen für nicht fokussierte Ziele

Zu jedem Zeitpunkt hat das Gespräch genau ein Ziel im Fokus, nach dem zu diesem Zeitpunkt gefragt wird. Jedoch ist es ebenfalls möglich, dass bereits alle Informationen durch vorhergehende Nachrichten übermittelt wurden, um alle Slots eines Ziels direkt zu füllen, sobald dieses Ziel aktiviert wird. In einem solchen Fall kann mit diesen Informationen aus früheren Nachrichten, die bisher nicht verwendet werden konnten, das Ziel direkt abgeschlossen werden, ohne dass der Benutzer eine Nachricht zu diesem Ziel erhält. Dieses Verhalten ist jedoch nicht immer wünschenswert, da teilweise Informationen nur innerhalb des konkreten Kontextes des Ziels, also wenn das Ziel aktuell im Fokus ist, korrekt verwendet werden können. Zum Beispiel könnte es ein Ziel sein, einen Arzttermin mit dem Benutzer zu vereinbaren, sodass eine Zeitangabe die benötigte Information wäre. Wenn der Benutzer zu einem früheren Zeitpunkt des Gesprächs bereits eine Zeitangabe geäußert hat, wäre diese theoretisch als Information zum Beenden dieses Ziels ausreichend. Jedoch ist diese Zeitangabe ohne den konkreten Kontext, dass nach einem Terminvorschlag gefragt wird, höchst wahrscheinlich nicht den Wünschen des Benutzers für einen Arzttermin entsprechend und somit nicht verwendbar. Damit solche Probleme verhindert werden können, werden entsprechende Ziele gekennzeichnet, sodass sie nur mit Informationen befüllt werden, die der Benutzer

neu innerhalb des Kontextes dieses Ziels übermittelt. Ältere Nachrichten, welche der Benutzer geäußert hatte, als ein anderes Ziel im Fokus war, und somit nicht geeignet sein könnten, werden dann nicht in Betracht gezogen.

3.3 Syntax zum Darstellen und Modellieren von Gesprächszielen

Nachdem nun erläutert wurde, welche Daten für ein zielorientiertes Dialogmodell benötigt sind, wird eine formale Syntax beschrieben, damit Domänen-Experten ein ähnliches Modell für konkrete Dialoge entwerfen können, wie es im realen Einsatzszenario der Fall wäre. Es gibt zwar bereits vorhandene Ansätze für das Modellieren von Gesprächen, jedoch kommt keiner von diesen für den Anwendungsfall dieser Arbeit in Frage. Die meisten dieser vorhandenen Ansätze basieren auf Auszeichnungssprachen wie XML oder JSON als Basis zur Definition des Modells, wie zum Beispiel AIML[1] beziehungsweise RiveScript[2]. Jedoch haben die Experten in vielen Domänen keine oder nur geringe Erfahrung mit solchen Auszeichnungssprachen, und deshalb sollte eine einfachere und besser überschaubare Syntax zum Erstellen und späteren Modifizieren gewählt werden. Wie zuvor bereits erläutert wurde, sind die Gesprächsziele untereinander verbunden. Dadurch bietet sich eine grafische Darstellungsform eines Graphs mit einigen textuellen Attributen innerhalb der Knoten des Graphs an. Mit dieser grafischen Notation ist eine schnelle Bearbeitung und verständliche Repräsentation des Dialogmodells möglich. Grafische Darstellungen sind übersichtlicher und anschaulicher und die Kanten- und Knotennotation eines Graphs ist schnell zu erlernen, was diese Syntax zu einer guten Wahl macht. Deshalb wurde für diese Arbeit die Entscheidung getroffen, einen Ansatz für eine neue und grafische Syntax zum Modellieren von Gesprächen zu entwerfen, welche alle benötigten Metadaten des Gesprächsmodells mit allen Zielen und den dazugehörigen Daten, die in Abschnitt 3.2 aufgezählt werden, enthält. Ein Beispiel für ein komplettes Dialogmodell in dieser neuen Notation ist in Anhang B in vereinfachter Form zu sehen. Die einzelnen Elemente dieser Syntax werden im Folgenden exemplarisch gezeigt und ihre Semantik wird erläutert.

3.3.1 Repräsentation von Gesprächszielen

Als Kernelemente einer solchen grafischen Repräsentation eines Dialogmodells durch einen Graph lassen sich die Gesprächsziele identifizieren. Diese haben einige relevante Metadaten, welche zusammen mit dem Ziel definiert und dargestellt werden sollten, wie zum Beispiel in Abbildung 3.1 zu sehen ist. Ein Ziel wird durch eine rechteckige Box repräsentiert, die vergleichbar mit einem UML-Klassendiagramm oben seinen Namen hat und getrennt darunter eine Aufzählung

[1] `http://www.alicebot.org/aiml.html`
[2] `https://www.rivescript.com/`

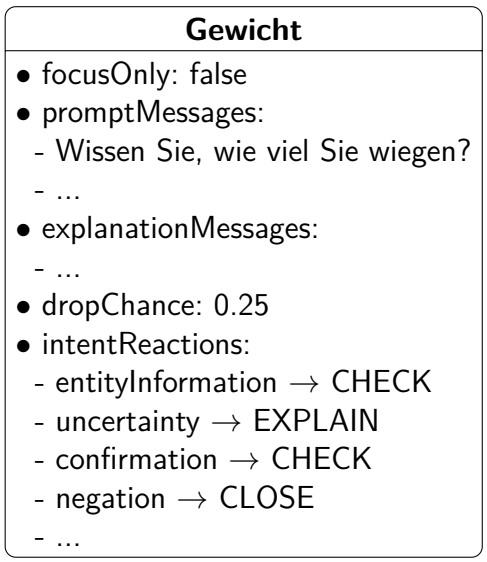


Abbildung 3.1: Beispiel für die grafische Repräsentation eines Gesprächsziels

der Metadaten des Ziels:

- `focusOnly`: Ein boolscher Wert, der ausdrückt, ob die Slots des Ziels nur gefüllt werden dürfen, wenn dieses Ziel aktuell aktiv im Fokus steht (siehe Abschnitt 3.2.10).
- `promptMessages`: Eine Auswahl von Nachrichten, von denen eine zufällig ausgewählt und dem Nutzer geschickt wird, sobald dieses Ziel fokussiert wird (siehe Abschnitt 3.2.1).
- `explanationMessages`: Eine Auswahl von Nachrichten, die detaillierter als die `promptMessages` dieses Ziel erklären, von denen eine zufällig ausgewählt und dem Nutzer geschickt wird, wenn als Intentions-Reaktion eine `EXPLAIN`-Reaktion ausgelöst wird (siehe Abschnitt 3.2.3).
- `dropChance`: Die Wahrscheinlichkeit, mit der dieses Ziel ohne Bearbeitung fallen gelassen wird, falls der Benutzer nicht auf dieses eingeht (siehe Abschnitt 3.2.4).
- `intentReactions`: Eine Zuordnung von den möglichen Kategorien der Benutzer-Intentionen zu den möglichen Reaktions-Typen des Systems (siehe Abschnitt 3.2.2).

3.3.2 Repräsentation der Beziehungen zwischen Gesprächsthemen

Wie in Abschnitt 3.2.7 erwähnt haben Gesprächsziele Beziehungen untereinander, welche die Reihenfolge der Ziele innerhalb des Gesprächs definieren, sofern der

Benutzer nicht das Thema wechselt. Mit einer Menge an Zielen Z sind solche Verbindungen zwischen Zielen Elemente aus $Z \times Z$. Weiterhin ist es möglich, dass eine Kante um ein Intentions-Label erweitert wird, also $(Z \times Z) \cup (Z \times Z \times Label)$. Bei einer Kante mit einem Label (z_1, z_2, l) gibt l an, dass das nachfolgende Ziel z_2 nur aktiviert wird, falls z_1 von dem Benutzer mit einer Nachricht mit der Intention l abgeschlossen wurde. Mit einem Graph als Grundlage für die grafische Notation wären diese Beziehungen die Kanten und sollten deshalb grafisch als Pfeile dargestellt werden. Dies ist in Abbildung 3.2 exemplarisch zu sehen, wo modelliert wird, dass nach dem Gewicht und bei passender Aktivierung nach Problemen beim Urinieren gefragt wird. Außerdem ist hier ein Intentions-Label an einer Kante zu sehen, da hier das Ziel einen Arzt-Termin zu vereinbaren nur in Betracht gezogen wird, wenn der Benutzer die Frage nach Problemen beim Urinieren bestätigt.

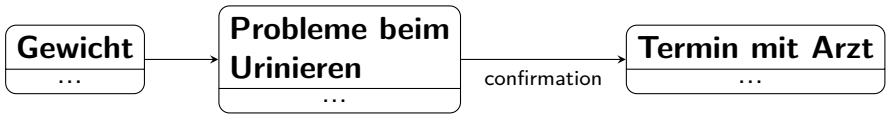


Abbildung 3.2: Beispiel für die grafische Repräsentation von Beziehungen zwischen Gesprächszielen

3.3.3 Repräsentation der Trigger

Die in Abschnitt 3.2.9 beschriebenen Trigger sind die einzige Möglichkeit, bewusst die Reihenfolge der Bearbeitung der Ziele innerhalb des Gesprächs abzuändern. Unabhängig von der Vernetzung der Ziele kann so direkt zu einem Ziel gesprungen werden. Deshalb sollten die Trigger auch grafisch direkt bei diesem Ziel dargestellt werden, wie in Abbildung 3.3 zu sehen ist. Um einen Trigger zu definieren, sind drei Metadaten relevant:

- `expectedValues`: Eine Aufzählung von Schlüsselwörtern, auf die der Trigger reagiert.
- `type`: Die Kategorie, in der dieses Schlüsselwort von der NER eingeordnet wurde.
- `multiOccurrence`: Falls mehrere Entitäten mit der gleichen NER-Kategorie innerhalb einer Nachricht auftauchen, zeigt dieser boolsche Wert an, ob alle diese Entitäten von diesem Trigger benutzt werden sollen oder ob er nur eine verbraucht und alle anderen für andere Trigger oder Informations-Slots übrig bleiben.

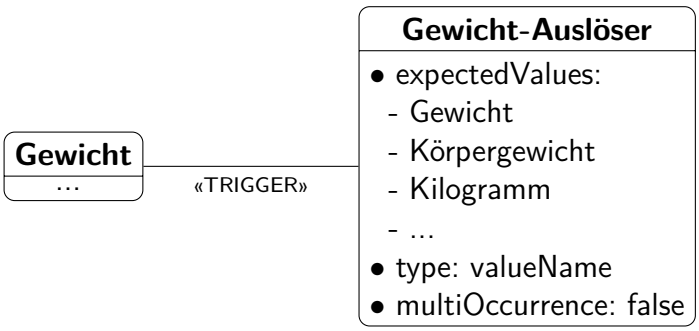


Abbildung 3.3: Beispiel für die grafische Repräsentation von einem Auslöser für einen Gesprächsthemawechsel

3.3.4 Repräsentation der Slots eines Ziels

Zu einem Ziel gehören auch die benötigten Informationen. Erst sobald all diese Slots der Informationen gefüllt worden sind, kann das Ziel als erreicht betrachtet werden, und es kann entweder direkt oder abhängig von einer Aktivierung mit den nachfolgenden Zielen fortgefahren werden. Um eine Information gezielt zu erfragen und die Antwort korrekt zuordnen zu können, sind mehrere Metadaten notwendig, wie in Abschnitt 3.2.5 bereits angeschnitten:

- `onMissing`: Eine Sammlung von Nachrichten, mit denen gezielt nach dieser Information gefragt werden kann und von denen der Benutzer eine zufällig ausgewählte geschickt bekommt, wenn das dazugehörige Ziel aktiv ist, aber der Slot dieser Information noch nicht gefüllt wurde.

- `type`: Die Klasse, aus der Entitäten von der NER eingeordnet sein müssen, um diesen Slot füllen zu können.

- `multiOccurrence`: Falls eine Entität aus der NER-Klasse mehrfach in der Benutzer-Nachricht vorkommt, gibt dieser boolsche Wert an, ob diese alle von dem Slot dieser Information verbraucht werden oder ob dieser Slot maximal eine Entität verbraucht und alle anderen Entitäten dieser Klasse für andere Slots oder Trigger übrig bleiben.

Ein Beispiel dafür sieht man in Abbildung 3.4, in der für das Ziel Gewicht der konkrete Gewichtswert und der Zeitpunkt, an dem gemessen wurde, modelliert sind.

3.3.5 Repräsentation von Revisoren und Aktivatoren

Zusätzlich zu den Definitionen, welche Informationen von einem Ziel benötigt werden, gibt es auch noch zu modellierende Logik, die hinter diesen Informationen steckt und den Verlauf des Gesprächs beeinflusst. Diese Logik besteht aus den bereits erläuterten Revisoren (siehe Abschnitt 3.2.6) und Aktivatoren (siehe

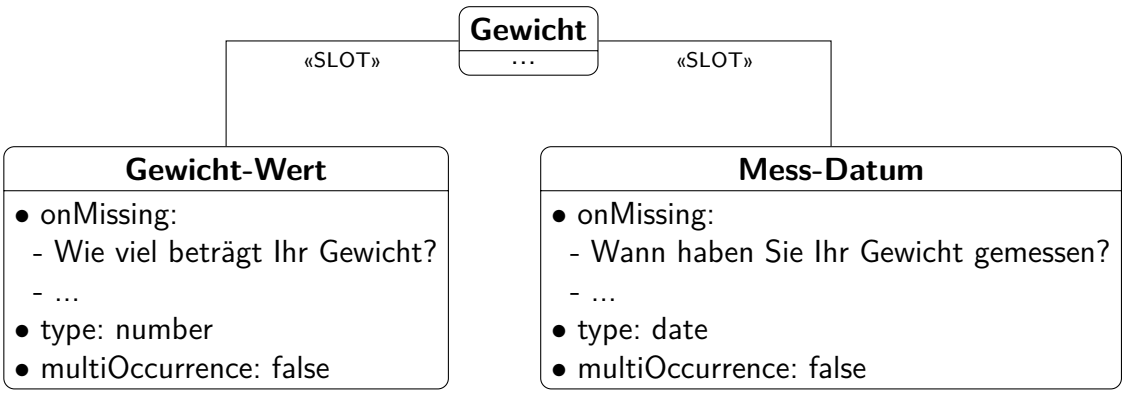


Abbildung 3.4: Beispiel für die grafische Repräsentation einer benötigten Information

Abschnitt 3.2.8). Beide haben eine prädikatenlogische Bedingung, um zu überprüfen, ob ein Wert für den Slot in Betracht gezogen werden kann beziehungsweise ob ein Wert neue Ziele aktiviert. Damit Domänen-Experten nicht den Umgang mit formaler Prädikatenlogik erlernen müssen, gibt es vordefinierte Typen von parametrisierten Bedingungen. Somit muss bei der Modellierung des Gesprächs nur noch der Typ ausgewählt werden und dazugehörige Parameter als Metadaten angegeben werden. Beide Bausteine sind in der grafischen Repräsentation des Gesprächsmodells mit ihrem entsprechenden Slot verbunden, und beide haben einen ähnlichen Aufbau ihrer Metadaten, weshalb sie zusammen in diesem Abschnitt beschrieben werden. Wie in Abbildung 3.5 zu sehen ist, enthalten Revisoren und Aktivatoren folgende Metadaten:

- `condition`: In der `condition` wird die prädikatenlogische Bedingung des Revisors oder Aktivators definiert. Es gibt weitere Typen von `conditions`, als die in der Abbildung dargestellten, die auch andere Parameter als die hier verwendeten `lower` und `upper` haben.

- `onActivation`: Eine Sammlung von Nachrichten, von denen eine zufällig ausgewählte oder keine, falls keine angegeben sind, an den Nutzer geschickt wird, sofern die `condition` erfüllt ist.

- `onEnd`: Eine Sammlung von Nachrichten, von denen eine zufällig ausgewählte oder keine, falls keine angegeben sind, an den Nutzer geschickt wird, sofern die `condition` nicht erfüllt ist.

- `onDataMissing`: Einige `condition types` vergleichen die übermittelte Information mit Informationen aus früheren Dialogen. Falls keine beziehungsweise nicht genug von solchen Informationen gespeichert sind, wird eine der Nachrichten aus dieser Sammlung zufällig ausgewählt und an den Nutzer geschickt, sofern welche vorhanden sind.

Im Beispiel in Abbildung 3.5 wird mit einem Revisor vom type `BOUNDS` geprüft, ob sich das übermittelte Gewicht zwischen 0 und 500 Kilogramm befindet. Andernfalls wird es nicht in den Slot gefüllt und die Warnung wird geschickt. Falls der Revisor das Schreiben des Werts in den Slot zulässt, wird der Aktivator vom `type RELATIVE_CHANGE`, nachdem auch alle weiteren Slots des Ziels gefüllt wurden, überprüfen, ob sich das Gewicht im Vergleich zu dem letzten bekannten Gewicht um mehr als 10 % verändert hat. Dann wird die entsprechende der drei Nachrichten-Typen geschickt, und bei erfolgreicher Aktivierung eventuelle nachfolgende Ziele aktiviert.

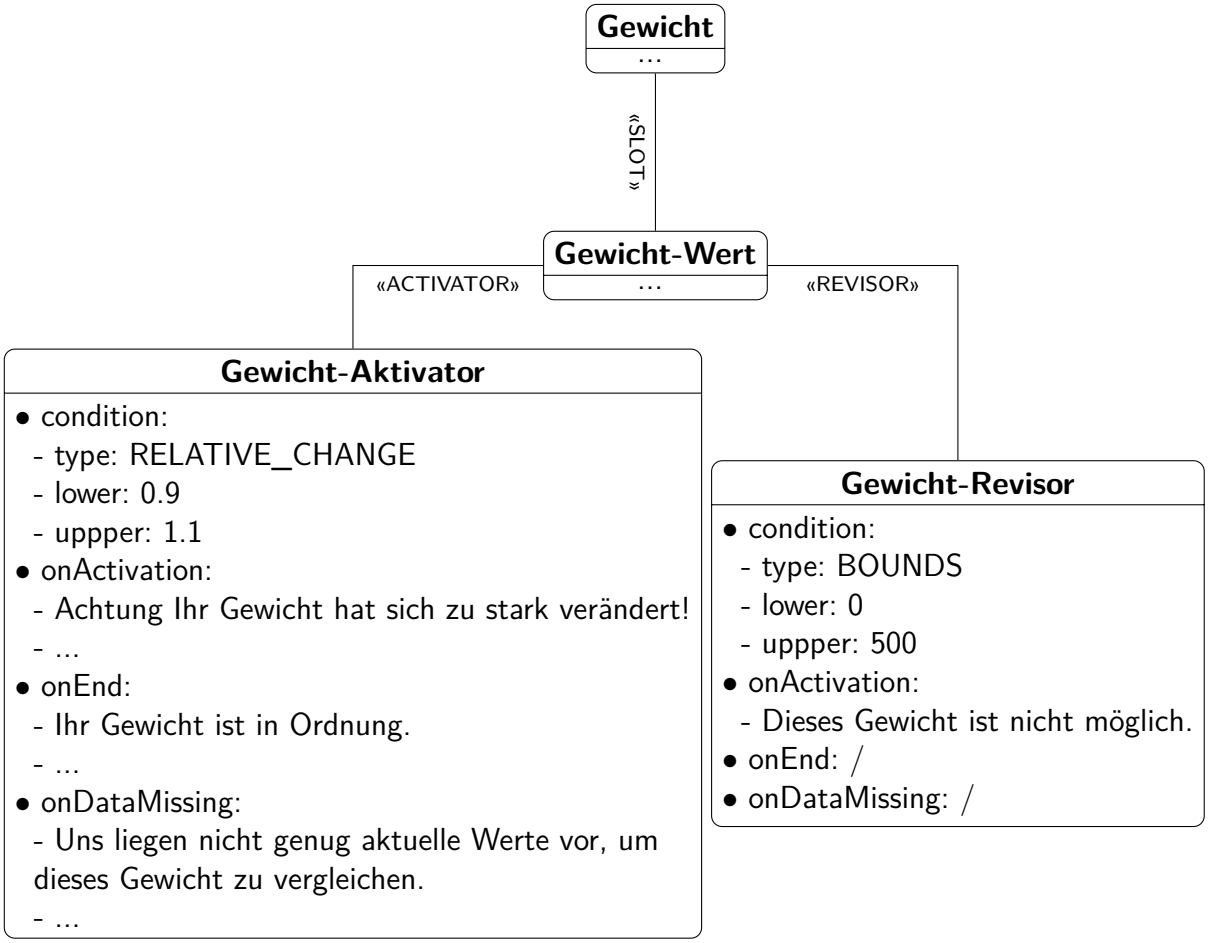


Abbildung 3.5: Beispiel für die grafische Repräsentation von Aktivatoren und Revisoren

3.4 Funktionsweise einer Gesprächsstrategie zum dynamischen Bearbeiten von Gesprächszielen

Bisher wurde erklärt, welche Daten benötigt werden, um Gesprächsziele zu definieren und wie diese Gesprächsziele gemeinsam grafisch in einem Gesprächsmodell dargestellt werden können. Eine algorithmische Strategie, mit der alle Ziele des Modells zielorientiert erfüllt werden können, komplettiert den Ansatz. Dabei muss sowohl die zustandsbasierte als auch die regelbasierte Grundidee erhalten bleiben. Hierzu gilt es, zwei Aspekte zu betrachten. Zuerst ist es wichtig, während des Dialogs Daten über den Zustand des Gesprächs zu halten. Zweitens muss mit diesen Daten und dem Gesprächsmodell zusammen algorithmisch eine Antwort an den Nutzer generiert werden. Diese Antwort sollte so gewählt sein, dass die Ziele des Dialogmodells bestmöglich erreicht werden können. Beide Punkte werden im Folgenden erläutert.

3.4.1 Datenhaltung während des Dialogs

Für die zielorientierte Bearbeitung des Dialogmodells sind neben dem grundlegenden Graph auch noch folgende Daten bezüglich des aktuellen Status des Dialogs relevant:

- Das Ziel, das aktuell im Fokus ist und behandelt wird, und seine Slots, die zum Teil schon befüllt sein können. Hierzu gehören insbesondere alle Metadaten des Ziels (`focusOnly, promptMessages, explanationMessages, dropChance, intentReactions`) und der jeweiligen Slots (`onMissing, type, multiOccurrence`). Die Slots werden hierbei um einen Eintrag beziehungsweise ein Array an Einträgen erweitert, in dem die Entitäten gespeichert werden, mit denen dieser Slot befüllt wurde.

- Alle erfolgreich erreichten Ziele in der Form: Name des Ziels, Tupel aus dem Namen eines Slots des Ziels und der Entität beziehungsweise den Entitäten mit denen der Slot befüllt wurde und das Intentions-Label, mit dem die Nachricht klassifiziert wurde, welche dieses Ziel final abgeschlossen hat.

- Die Namen aller offenen Ziele. Ein Ziel gilt als offen, wenn es im Laufe des aktuellen Dialoges aktiviert wurde, aber nicht aktuell im Fokus ist und seine Slots noch nicht befüllt wurden, es also auch noch nicht in der Liste der erreichten Ziele enthalten ist. Bei diesen offenen Zielen ist zusätzlich die Reihenfolge wichtig, in welcher sie aktiviert beziehungsweise erreicht wurden, um die logischen Zusammenhänge zwischen den Zielen und somit den roten Faden des Gesprächs aufrecht erhalten zu können. Damit dies erreicht wird, werden die offenen Ziele in einer Datenstruktur gehalten, die die Reihenfolge bewahrt. In dieser Reihenfolge werden sie nun auch bearbeitet, außer in zwei Fällen. Wenn der Benutzer das Thema wechselt, wird zu diesem Punkt in

der Datenstruktur gesprungen und von dort aus mit der ab dort geltenden Reihenfolge fortgefahren. Weiterhin gibt es einen Sonderfall, falls ein Ziel ein anderes Ziel aktiviert. Angenommen aktuell ist die Reihenfolge innerhalb der Datenstruktur $x_1, ..., x_n$ und es gibt kein aktives Ziel im Fokus. Deshalb wird das Ziel x_1 an erster Stelle der Datenstruktur in den Fokus geladen, sodass regulär das nachfolgende Ziel x_2 als nächstes behandelt würde. In diesem Fall aktiviert nun jedoch x_1 bei seinem Erreichen die Ziele y_1 und y_2. Daraufhin werden y_1 und y_2 an das Ende der Datenstruktur angehängt, damit die Reihenfolge der zuvorkommenden Ziele $x_2, ..., x_n$ nicht verletzt wird. Als nächstes fokussiertes Ziel wird y_1 gewählt und sofern dieses nicht seinerseits Ziele aktiviert, danach y_2. Wenn nun das Ende der Datenstruktur erreicht wird und es beim letzten Ziel keine Aktivierungen gab, wird wieder zu dem Anfang der Datenstruktur gesprungen und $x_2, ..., x_n$ in ihrer Reihenfolge bearbeitet. Da Aktivierungen durch Kanten zwischen Zielen modelliert werden, ähnelt diese Bearbeitungsreihenfolge einer Tiefensuche, da zuerst bei einem Ziel-Knoten v_1 mit den nachfolgenden Knoten v_2 und v_3, v_2 und alle nachfolgenden Knoten von ihm untersucht werden, bevor v_3 betrachtet wird.

- Von jeder früheren Nachricht, welche Entitäten enthielt, die nicht vollständig für Slots oder Trigger verbraucht wurden, werden ihre Entitäten und ihre Intention gespeichert. Damit wird verhindert, dass der Benutzer Informationen gegeben hat, die ein Ziel erfüllen könnten, das zu diesem Zeitpunkt nicht aktiv war. Durch dieses Speichern kann in so einem Fall ein neues Ziel direkt mit diesen vorhandenen Informationen gefüllt werden, und der Benutzer muss nicht erneut nach Informationen befragt werden, die er bereits gegeben hat. Es kann bei einem Ziel kritische Auswirkungen haben, wenn der unwahrscheinliche Fall eintritt und alle Slots durch die Entitäten einer nicht verwendeten Nachricht gefüllt werden können, welche zwar durch die NER richtig kategorisiert wurden aber dennoch in einem falschen Kontext sind. Hierfür lässt sich bei einem Ziel `focusOnly` auf Wahr setzen, und es kann nicht mit einer dieser nicht vollständig verbrauchten Nachrichten gefüllt werden.

3.4.2 Phasenweise Verarbeitung einer Benutzer-Nachricht

Die empfangene strukturierte Nachricht des Benutzers nach der linguistischen Analyse durch maschinelles Lernen besteht aus den extrahierten Entitäten der NER und der Klassifizierung der Intention. Mit den strukturierten Daten als Eingabe werden in mehreren Phasen die Elemente des Dialogmodells bearbeitet, um schlussendlich alle Ziele zu erreichen. Diese Phasen werden nun in chronologischer Reihenfolge ihrer Ausführung entsprechend vorgestellt und textuell erläutert. Eine exemplarische Implementierung des Algorithmus in JavaScript, welcher sich aus diesen Phasen ergibt, lässt sich zum besseren Verständnis in Anhang A finden.

1. Zuerst wird untersucht, ob der Benutzer mit dieser Nachricht versucht, das

Thema zu wechseln, also ob ein offenes Ziel einen zur Nachricht passenden Trigger hat. Falls ja, wird dieses offene Ziel zum neuen Fokus-Ziel.

2. Zu der Nachricht wird als nächstes eine passende Reaktion durchgeführt. Hierbei wird für das Label $l \in L$ der Intentionsklassifizierung der Benutzernachricht eine Reaktion der möglichen Reaktionen R durchgeführt. Die Auswahl einer Reaktion geschieht durch die modellierte Zuordnung des aktuellen Fokusziels `intentReactions`: $L \rightarrow R$. Für diese Arbeit sind die Reaktionen (`EXPLAIN`, `ACCEPT`, `CLOSE` und `CHECK`) am wichtigsten und werden im Folgenden einzeln kurz vorgestellt. Theoretisch sind auch weitere Reaktionen möglich und leicht hinzuzufügen.

   Die `EXPLAIN`-Reaktion ist für den Fall gedacht, dass der Benutzer die Frage zu dem aktuellen Ziel nicht versteht oder nicht weiß, was er antworten kann. Dann wird als Antwort an den Benutzer eine zufällig ausgewählte Nachricht aus den `explanationMessages` des Fokus-Ziels und die `onMissing`-Frage des ersten leeren Slots geschickt.

   Eine `ACCEPT`-Reaktion ist passend, wenn allein durch eine Intention ein Ziel erfolgreich geschlossen werden kann, wie zum Beispiel bei einer Ja/Nein Frage. Das Fokus-Ziel wird bei einer solchen Reaktion zusammen mit der Intention, die zur `ACCEPT`-Reaktion geführt hat, den geschlossenen Zielen hinzugefügt und überprüft, ob nachfolgende, neue Ziele aktiviert werden.

   Bei einer `CLOSE`-Reaktion wird davon ausgegangen, dass der Benutzer nicht in der Lage ist das aktuelle Ziel zu bearbeiten, zum Beispiel wenn er zwar versteht, wie eine Frage gemeint ist, die Antwort aber nicht kennt. Das Ziel wird in diesem Dialog nicht weiter behandelt und deshalb aus der Liste der offenen Ziele entfernt. Anschließend wird das nächste offene Ziel in den Fokus geladen.

   Die wichtigste Reaktion ist die `CHECK`-Reaktion, da bei dieser versucht wird, die Slots des Fokus-Ziels mit den Entitäten der Nachricht zu füllen. Mit Berücksichtigung der Typen der Slots und der Entitäten der Nachricht sowie der `multiOccurrence`-Angabe eines Slots, wird überprüft, ob sich ein offener Slot mit einer beziehungsweise im `multiOccurrence`-Fall mehreren Entitäten der Benutzer-Nachricht füllen lässt. Falls sich ein oder mehrere Slots füllen lassen und zusätzlich an einem Slot ein Revisor angehängt ist, wird dieser vor dem Füllen die entsprechenden Werte prüfen und gegebenenfalls die passende Nachricht an den Nutzer senden. Sofern danach alle Slots gefüllt sind, werden analog zum `ACCEPT`-Fall alle Aktivatoren und Kanten mit Begrenzung der Intention überprüft und nachfolgende Ziele den offenen Zielen hinzugefügt, während das aktuelle Ziel erfolgreich geschlossen wird. Sollte jedoch kein Slot in diesem Durchlauf gefüllt werden können, obwohl

dies durch die Zuweisung der `CHECK`-Reaktion erwartet wurde, wird mittels der `dropChance` überprüft, ob das Fokusziel übersprungen wird.

3. Falls es Entitäten in der Nachricht gibt, die für keinen Slot oder Trigger des Fokusziels innerhalb von Phase Zwei verbraucht werden konnten, muss überprüft werden, ob diese Nachricht eines der offenen Ziele, sofern diese nicht `focusOnly` sind, vollständig füllen kann, bevor dieses offene Ziel als Fokus-Ziel behandelt wird. Das Verfahren ist ähnlich der `CHECK`-Reaktion inklusive des Überprüfens der Revisoren und der Aktivatoren, jedoch mit dem Unterschied, dass in dieser Phase alle Trigger, die ein offenes Ziel haben, angesprochen werden müssen, um sicherzugehen, dass die Kontexte des Ziels und der Nachricht identisch sind und die Übereinstimmung von Slots und Entitäten der Nachricht kein Zufall ist. Dadurch wird in dieser dritten Phase der Fall behandelt, dass der Benutzer nebenbei, eventuell sogar unwissentlich, genügend Informationen gibt, die ein bisher offenes Ziel direkt abschließen, ohne dass es erst detailliert wie in Phase 2 durchlaufen wird. Wenn dadurch ein offenes Ziel mit seinen Slots durch die Nachricht vollständig bearbeitet werden konnte, wird es den geschlossenen Zielen hinzugefügt, zusammen mit den befüllten Slots und der Intention der Nachricht. Falls ein offenes Ziel der Nachrichten nur teilweise beantwortet werden konnte und innerhalb dieses Durchlaufs der Phasen das Fokusziel nicht bereits geändert wurde, wird dieses offene Ziel zum neuen Fokusziel für den nächsten Durchlauf, um die noch verbleibenden offenen Slots durch Nachfragen an den Benutzer füllen zu können. Falls nach Phase 3 die Entitäten der Benutzernachricht immer noch nicht alle verbraucht werden konnten, werden sie gebündelt den nicht benutzten Informationen hinzugefügt.

4. Nun werden die bisher aktivierten neuen Ziele dieses Durchlaufs behandelt. Bei diesen muss überprüft werden, ob sie mit einem der Sets aus den nicht benutzten Informationen direkt gefüllt werden könnten und sie somit gar nicht erst den offenen Zielen hinzugefügt werden müssen, sondern direkt den Geschlossenen. Alle Ziele, die von diesem neuen und jetzt geschlossenen Ziel aktiviert wurden, werden ebenfalls der Liste der neuen Ziele hinzugefügt, deren Elemente aktuell in dieser Phase durchlaufen werden. Wie bei der vorhergehenden Phase ist es hier wichtig, dass auch alle Trigger korrekt getroffen werden und nicht `focusOnly` gilt, um einen fehlerhaften Kontext auszuschließen. Falls ein neues Ziel nur zum Teil gefüllt werden kann, wird es, ebenfalls, wie in der vorherigen Phase, zum neuen Fokus-Ziel, sofern in diesem Durchlauf das Fokus-Ziel noch nicht geändert wurde.

5. Zuletzt muss nun noch die vollständige Antwort an den Benutzer für den Fall generiert werden, dass es sich um keine `EXPLAIN`-Reaktion gehandelt hat. Ansonsten enthält die Antwort bisher eventuell eine Nachricht oder einen Hinweis von einem Revisor beziehungsweise Aktivator. Da solche Nachrichten jedoch rein informativer Natur sind, weiß der Benutzer noch nicht mit

absoluter Sicherheit, was er als nächstes antworten kann. Deshalb wird diesem informativen Anteil der nächsten Nachricht noch eine Frage angehängt. Diese wird entweder zufällig den `promptMessages` des aktuellen Fokusziels entnommen, falls das Ziel noch nicht bearbeitet wurde, oder alternativ ist es die `onMissing`-Frage eines ungefüllten Slots.

4 Architektur und Implementierung eines Chatbot-Prototyps

Dieser hybride Ansatz zur Modellierung von zielgerichteten Dialogen und dazugehörigem Strategiealgorithmus wurde zuvor theoretisch erläutert, sollen jedoch auch in realen Einsatzdomänen verwendbar sein. Dazu wird im folgenden Kapitel zuerst ein Szenario für eine solche Einsatzdomäne beschrieben und anschließend erläutert, wodurch in diesem Szenario ein Chatbot einen Mehrwert liefert. Dies geschieht am Beispiel des Medolution-Projektes[1], welches in der Einleitung bereits vorgestellt wurde. Danach wird eine tatsächliche Implementierung eines Prototypen beschrieben. Hierfür werden die Architektur des Prototypensystems und die als Grundlage verwendeten Bibliotheken vorgestellt. Anschließend wird zusätzlich erläutert, wie Domänenexperten innerhalb dieses Prototyps ein Dialogmodell entwerfen können, zum Beispiel ein Mediziner die Fragen an den Patienten innerhalb des Medolution-Projektes.

4.1 Überblick über die Komponenten des Medolution-Projektes

Wie in Abbildung 4.1 zu sehen ist, werden im Medolution-Projekt primär medizinische Daten verarbeitet, welche entweder automatisch durch Sensoren oder von Mitarbeitern manuell als Datenintegratoren an einen Webserver übermittelt werden. Diese Daten werden durch eine *Apache Storm*[2] Topologie auf definierte Events untersucht. Falls ein solches Event auftritt, kann sofort ein medizinischer Experte benachrichtigt werden. Diese Ereignisse werden zusammen mit den ursprünglich eingegangen Daten als strukturierte *Apache Kafka*[3] Datenströme in ein verteiltes *Apache Cassandra*[4] Datenbank-Cluster geleitet und dort gespeichert. Aus diesen gespeicherten Daten kann ein Analyse-Bericht für medizinische Experten oder Datenwissenschaftler generiert werden.
Es ist jedoch nicht möglich, alle relevanten Daten durch Sensoren herauszufinden. Zum Einen gibt es für einige Faktoren, zum Beispiel das Wohlbefinden des Patienten oder seine Ernährung, keine zuverlässigen Sensoren. Weiterhin würden

[1] `https://itea3.org/project/medolution.html`
[2] `https://storm.apache.org/`
[3] `https://kafka.apache.org/`
[4] `https://cassandra.apache.org/`

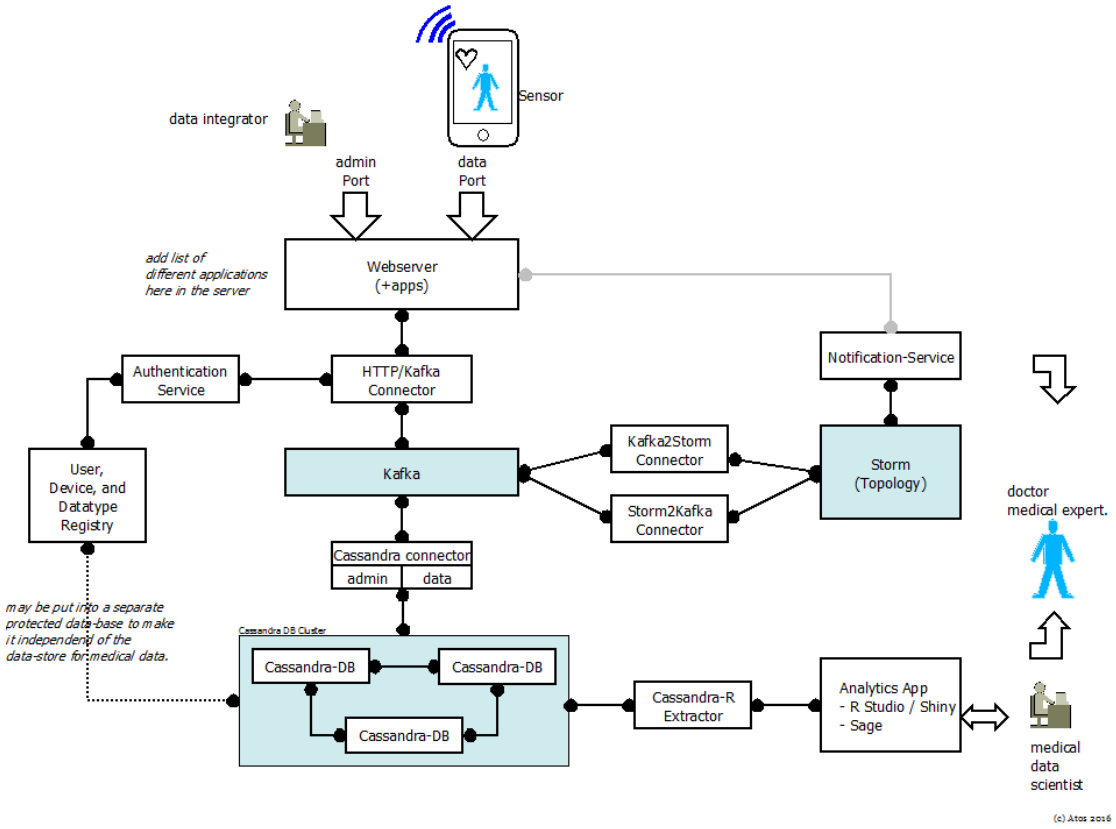


Abbildung 4.1: Geplanter Datenfluss im Medolution-Projekt (©Atos 2016)

zu viele Sensoren, die der Patient am Körper tragen muss, für Unannehmlichkeiten in seinem Alltag sorgen. Alternativ wäre es möglich, diese nicht automatisch erhebbaren Daten durch Kommunikation zwischen dem Patienten und Datenintegratoren zu sammeln und manuell einzutragen. Ein solcher Ansatz skaliert jedoch nicht gut, da mit steigender Anzahl an Patienten und relevanten Daten mehr Datenintegratoren benötigt werden, weil jeder nur mit einer begrenzten Anzahl von Patienten gleichzeitig kommunizieren kann. Anstatt mehr Datenintegratoren einzustellen und so die Kosten für die Patientenbetreuung in die Höhe zu treiben, stellt ein zielorientierter Chatbot eine gute Alternative dar, um ebenfalls in direkter Kommunikation mit den Patienten Daten zu erheben. Im Gegensatz zu einem Menschen kann ein Chatbotsystem auf einem leistungsfähigen Server mehrere Patienten gleichzeitig betreuen und durch Anpassung der Hardware entsprechend zu einer veränderten Anzahl der Patienten skalieren. In Abbildung 4.1 würde dieser Chatbot innerhalb des Medolution-Projektes als weitere Datenquelle für den Webserver dienen und könnte auch eigenständig Berichte für medizinische Experten liefern.

4.2 Wissensbasis für die Gesprächsstrategie

Ein Ziel eines solchen Chatbots ist es allerdings, nicht nur die Daten des Patienten zu erheben, sondern auch dynamisch Feedback zu geben. Relevant sind zum Beispiel Hinweise bei falscher Medikation oder Ernährung, sodass das Erreichen von kritischen medizinischen Werten präventiv verhindert werden kann. Solche Hinweise lassen sich mit dem vorgestellten Dialogmodell einfach einbauen, indem nach Übermittlung der entsprechenden Informationen, zum Beispiel zu gegessenen Lebensmitteln, ein Aktivator den Zusammenhang zu medizinischen Werten überprüft und je nach Aktivierung dem Nutzer eine Warnung zurückschickt. Jedoch ist es kein einfach wartbarer Ansatz in jeden Aktivator, der zum Beispiel für einen medizinischen Wert die beeinflussenden Lebensmittel überprüft, jeweils eine Liste von entsprechenden Lebensmitteln einzufügen. Deshalb wird dieses Wissen nicht direkt in das Dialogmodell integriert, sondern gesammelt als Ontologien in einem externen Wissensgraph gehalten. Für die Aktivatoren muss durch diese Separierung nur noch modelliert werden, welche Anfrage sie an den Wissengraph stellen müssen, um seine `condition` auszuwerten. Wenn sich Änderungen bezüglich des Wissens zum Beispiel über Lebensmittel ergeben, müssen sie dadurch nicht mehr in jedem Aktivator einzeln angewandt werden, sondern nur zentral im Wissensgraph gespeichert werden. Die Ontologie, welche für das Evaluations-Beispiel des Medolution Chatbots verwendet wird, lässt sich in Anhang B zusammen mit dem Dialogmodell, das sie unterstützen soll, finden.
Dieses Wissen soll jedoch nicht vollständig manuell modelliert werden müssen, sondern teilweise durch automatische Schlussfolgerungen ergänzt werden. Wenn zum Beispiel Inhaltsstoffe, wie zum Beispiel manche Vitamine, einen medizinischen Wert beeinflussen, können Regeln definiert werden, die besagen, dass jedes Lebensmittel, das diesen Inhaltsstoff enthält, somit ebenfalls den Wert beeinflusst. Die Verbindung von jedem entsprechenden Lebensmittel zum Wert muss nicht explizit angegeben werden, sondern dieser Zusammenhang wird automatisch geschlussfolgert. Für solche Schlussfolgerungen werden Regeln formuliert. Als Vorbedingung für diese Regeln werden Prädikate zwischen Subjekten und Objekten gefordert. Sofern diese Vorbedingungen innerhalb der Ontologie existieren, werden neue Relationen hinzugefügt. Wenn zum Beispiel das `istEin`-Prädikat transitiv sein soll, also wenn gilt: `a istEin b` und `b istEin c`, dann soll auch gelten `a istEin c`, ohne dass diese dritte Aussage eigenständig modelliert werden muss. Eine solche Regel würde wie folgt aussehen:

```
@prefix md:   <http://www.c-lab.de/projekte/Medolution#> .
[ruleIstEin: (?s md:istEin ?o) (?o md:istEin ?r) -> (?s md:istEin ?r)]
```

Alle Schlussfolgerungsregeln, die für die Ontologie des Evaluations-Beispiels verwendet werden, lassen sich ebenfalls in Anhang B finden.

4.3 Verwendete Bibliotheken und Frameworks

Innerhalb dieser Arbeit werden nicht alle benötigten Komponenten neu entwickelt, sondern es wurden quelloffene Bibliotheken und Frameworks als Grundlage genommen. Im Folgenden werden diese Bibliotheken und ihr Beitrag zum Chatbot erläutert.

4.3.1 Rasa NLU

Rasa NLU[5] ist eine Open-Source Python Bibliothek zur Verarbeitung natürlicher Sprache. Nach eigenen Angaben ist sie speziell für Entwickler von Chatbots gedacht. Durch die Open-Source Lizenz ist sie kostenlos, und da das Training für das maschinelle Lernen vom Anwender auf eigener Hardware durchgeführt wird, behält der Anwender die volle Kontrolle und alle Rechte an den eigenen Trainingsdaten. Sie unterstützt als Bestandteile ihrer Processing Pipeline sowohl das Kategorisieren der Intention mittels einer SVM[6] als auch NER durch CRFs[7] und somit beide Arten von wichtigen Informationen einer Nachricht, wie in Abschnitt 2.1 erläutert. Für beide Verfahren benötigt *Rasa NLU* geeignete Trainingsdaten, welche im JSON-Format in folgender Form definiert werden:

```
{
  "text": "Ja ich habe gestern Rosenkohl gegessen",
  "intent": "confirmation",
  "entities": [
    {
      "entity": "lebensmittel",
      "value": "Rosenkohl",
      "start": 20,
      "end": 29
    }
  ]
}
```

4.3.2 Duckling

Wie in Abschnitt 2.1 bereits erläutert ist es für manche NER-Klassen möglich, ein hybrides Verfahren aus maschinellem Lernen und Grammatiken zu benutzen, wie zum Beispiel durch Probabilistic Context Free Grammars. Konkret wurde hier die von Facebook entwickelte Bibliothek *Duckling*[8] verwendet, die eine Open-Source Implementierung einer Probabilistic Context Free Grammar in der Programmiersprache Haskell ist. *Duckling* bietet fertige Grammatiken und Trainingsdaten für

[5] https://github.com/RasaHQ/rasa_nlu
[6] https://nlu.rasa.ai/pipeline.html#intent-classifier-sklearn
[7] https://nlu.rasa.ai/pipeline.html#ner-crf
[8] https://github.com/facebookincubator/duckling

einige NER-Klassen wie zum Beispiel Geldbeträge, Temperaturen, Datumsangaben, Uhrzeiten, E-Mail Adressen oder URLs. Theoretisch wäre es auch möglich gewesen, mit genug Trainingsdaten diese NER-Klassen durch *Rasa NLU* umzusetzen, jedoch konnte durch die Ergänzungen von *Duckling* diese manuelle Arbeit vermieden werden. Weiterhin hat *Duckling* die Funktionalität, relative Angaben wie zum Beispiel „Nächster Sonntag“ in absolute Angaben umzuwandeln, was in diesem Beispiel ein konkretes Datum wäre.

4.3.3 ArangoDB und Foxx

Für die Durchführung des Dialoges müssen das Dialogmodell und ebenfalls Informationen zu den Benutzern des Chatbots, wie Name und ID sowie eine Historie der Ergebnisse abgeschlossener Dialoge, persistent in einer Datenbank gespeichert werden. Sowohl die Informationen zu den Benutzern als auch die Elemente des Dialogmodells lassen sich im JSON-Format serialisieren. Um diese JSON-Dokumente und die Kanten zwischen den Elementen des Dialogmodells in einer Datenbank zu vereinen, wird eine Datenbank für beide Typen von Schemata benötigt. Hier fiel die Wahl auf die NoSQL-Datenbank *ArangoDB*[9], die nach eigenen Angaben als Multi-Modell-Datenbank Dokumente, Graphen und Key-Value Paare speichern kann[10]. Neben dem Multi-Modell Aspekt war der ausschlaggebende Punkt für diese Wahl die *ArangoDB* interne Bibliothek *Foxx*. Mit *Foxx* lassen sich von der Datenbank verwaltete REST-Microservices implementieren, die REST-Anfragen zur Ausführung von eigener Logik bezüglich der gespeicherten Daten erlauben. Dadurch kann der Quellcode, der sich ausschließlich mit dem Dialogmodell und nicht mit einem konkreten Gespräch beschäftigt, wie zum Beispiel das Suchen nach Triggern für ein bestimmtes Schlüsselwort, direkt innerhalb der Datenbank-Komponente verwaltet werden.

4.3.4 Apache Jena und Fuseki

Anfragen an einen Wissensgraph sind ein Beispiel für einen externen Service, der während der Durchführung des Dialogmodells bei der Entscheidungsunterstützung helfen kann. Ein Wissensgraph entsteht durch das Aufstellen von Ontologien, und die Anfragen werden über die dafür konzipierte Abfrage-Sprache *SPARQL* realisiert. Eine Plattform für das Laden und Speichern von Ontologien ist *Apache Jena*[11], welche durch die interne Anfrage-Engine *ARQ* auch *SPARQL*-Queries durchführen kann. *Jena* bietet weiterhin die dazugehörige Server-Plattform *Fuseki*, mit der Anfragen an eine *Jena*-Instanz per HTTP möglich sind. Durch *Fuseki* lässt sich *Apache Jena* eigenständig als externer Service in das System einbinden und von Aktivatoren oder Revisoren ansprechen. Die Bibliothek ist somit eine gute Wahl für einen Expertendienst durch Ontologien.

[9] `https://github.com/arangodb/arangodb`
[10] `https://www.arangodb.com/why-arangodb/multi-model`
[11] `https://github.com/apache/jena`

4.4 Zusammenfassung der Architektur des Chatbots

Die vorgestellten Bibliotheken und Frameworks müssen mit den selbstständig implementierten Teilen der Dialogstrategie kombiniert werden, um ein Gesamtsystem zur Durchführung von modellierten Gesprächen aufbauen zu können, und diese Architektur wird hier beschrieben. Wie auch in dem Schaubild in Abbildung 4.2 zu sehen ist, gib es somit insgesamt die folgenden Komponenten:

- Dialogsystem: Zentraler Baustein des Systems, der Nachrichten des Nutzers bekommt, mit diesen die Dialogstrategie verfolgt und eine Antwort zurückschickt.
- Textanalyse: Die beim Dialogsystem eingehenden Nachrichten werden von der Textanalyse mit *Rasa NLU* und *Duckling* analysiert und es werden aus dieser Nachricht die in Abschnitt 2.1 aufgezählten strukturierten Daten extrahiert, also die Intention und das NER-Ergebnis.
- Dialogmodell: Daten zu den Benutzern und das konkret entworfene Dialogmodell werden hier in einer *ArangoDB* Datenbank gespeichert und die Anfrage-Logik für Interaktionen mit dem Dialogsystem durch *Foxx*-Microservices implementiert.
- Expertendienst: Es werden Externe Dienste eingesetzt, um komplexere Aktivatoren und Revisoren modellieren zu können und so einen sehr detaillierten Decision-Support innerhalb des Dialogs zu ermöglichen, wie zum Beispiel durch *Apache Jena* mit *Fuseki* als Ontologie Service. Das Dialogmodell schickt in einem solchen Fall eine Anfrage an den Dienst und benutzt die Antwort als Ergebnis des Aktivators beziehungsweise Revisors. Es wird hier zwar von einem Expertendienst als Singular gesprochen, um die Erklärung der Architektur zu vereinfachen, jedoch sind beliebig viele von diesen externen Diensten unterschiedlichster Art als Teil des Systems denkbar und werden je nach Einsatzdomäne auch benötigt.
- Reportmodul: Je nach Einsatzdomäne können hier unterschiedliche Report-Möglichkeiten der abgeschlossenen Dialoge implementiert werden. Im Falle des Medolution-Projektes wäre es das Generieren eines PDFs mit allen Ergebnissen der aktuellen Dialog-Historie eines Benutzers sowie das Übermitteln der Werte des neuen abgeschlossenen Dialogs an den Webserver zur Datenauswertung mit der Topologie des Medolution-Projektes.

Wenn von der Dialogstrategie im Dialogsystem und den *Foxx*-Microservices im Dialogmodell abgesehen wird, sind die Aufgaben der übrigen Komponenten vergleichsweise simpel. In diesen werden die beschriebenen Bibliotheken eingebaut und Schnittstellen zum Ansprechen der Bibliotheken mit übermittelten Daten definiert. Von einer genaueren Beschreibung dieser Komponenten wird deshalb abgesehen. Zu den beiden komplexeren Komponenten lassen sich in Abschnitt 3.4 Informationen für die Funktionsweise sowie in Anhang A eine Beispiel-Implementierung

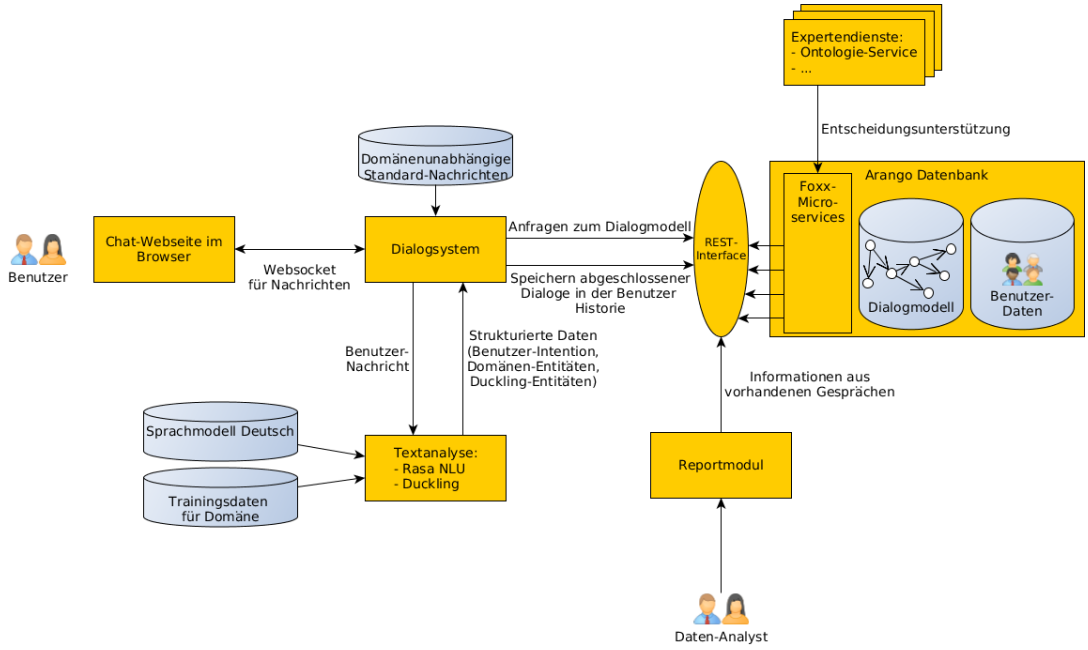


Abbildung 4.2: Überblick über die Komponenten des Chatbots

der Dialogstrategie des Dialogsystems finden In Anhang C ist die Spezifikation der *Foxx* REST-Schnittstelle des Dialogmodells aufgeführt. Deshalb wird im Folgenden nur auf die Interaktion der Komponenten untereinander näher eingegangen.
Das Dialogsystem hat eine Websocket-Verbindung zu einer Chat-Webseite, worüber die Nachrichten beider Gesprächsteilnehmer ausgetauscht werden. Hierfür wurde die Socket-Bibliothek *socket.io*[12] verwendet. Andere Client-Plattformen und Übertragungs-Kanäle neben Webseite und Websocket sind ebenfalls möglich und einfach austauschbar, solange das Dialogsystem über einen Kommunikationskanal Textnachrichten als Eingabe bekommt und ebenfalls mit solchen antworten kann. Weiterhin besitzt das Dialogsystem als externe Datenquelle einige Standardnachrichten, die unabhängig von der Domäne des Einsatzszenarios sind und generell in jedem Dialog benötigt werden. Beispiele für solche Standardnachrichten sind eine Verabschiedung, oder Fehlernachrichten. Die Textanalyse-Komponente mit *Rasa NLU* und *Duckling* benötigt zur Extraktion der strukturierten Daten aus der Benutzer-Nachricht ebenfalls externe Datenquellen, nämlich ein Deutsches Sprachmodell, in dem zum Beispiel Wort-Vektoren [18] vorhanden sind, und die Trainingsdaten für *Rasa NLU* zum Erlernen der Kategorisierung von Intentionen und von NER-Klassen der Domänen-Entitäten.
In Abbildung 4.3 ist als UML-Sequenz-Diagramm der Austausch von Nachrichten zwischen den Komponenten während der Verarbeitung einer Benutzer-Nachricht zu sehen, was in den folgenden Schritten erfolgt:

[12] `https://socket.io/`

1. Der Benutzer schickt seine Nachricht an das Dialogsystem.
2. Das Dialogsystem lässt von der Textanalyse die strukturierten Daten dieser Nachricht extrahieren.
3. Mit den strukturierten Daten und dem aktuellen Dialogzustand folgt die Dialogstrategie dem Dialogmodell, wobei Anfragen an das REST-Interface des Dialogmodells geschickt werden, um Informationen zu dem Dialogmodell zu erhalten. Obwohl hierbei auch mehrere Anfragen benötigt werden könnten, wird es im Sequenz-Diagramm zur Vereinfachung als eine Anfrage an das Dialogmodell dargestellt.
4. Wenn bei einer der Anfragen von dem Dialogsystem an das Dialogmodell eine Anfrage bezüglich der Auswertung eines Revisors oder Aktivators vorkommt, kann je nach Typ der Bedingung des Revisors beziehungsweise Aktivators eine Anfrage an einen externen Expertendienst gestellt werden, welcher zur Entscheidungsunterstützung die Bedingung des Revisors oder Aktivators prüft. Als Antwort wird das Ergebnis der Auswertung sowie in manchen Fällen eine spezifische Nachricht von dem Expertendienst an den Benutzer übermittelt.
5. Das Dialogsystem hat mit den Informationen des Dialogmodells seinen Zustand aktualisiert, und falls das Gespräch damit ein Ende erreicht hat, werden alle abgeschlossenen Ziele im Dialogmodell in der Historie der Dialoge des Benutzers gesichert.
6. Der Benutzer erhält abschließend die zum aktuellen Zustand ermittelte Nachricht als Antwort des Dialogsystems.

4.5 Definition und Modellierung von Gesprächszielen

Nachdem ein System aufgebaut wurde, mit dem zielorientierte Dialoge anhand von Dialogmodellen durchgeführt werden können, und die Anforderungen und die Wissensbasis für das Medolution-Projekt bekannt sind, wird abschließend ein beispielhaftes Dialogmodell für den Medolution-Anwendungsfall entworfen. Dieses wird zur Durchführung der nachfolgenden Evaluation benutzt, um die Tauglichkeit und die Benutzerfreundlichkeit des Ansatzes dieser Arbeit zu analysieren. Im Rahmen dieser Arbeit war es nicht vorgesehen, einen Editor zu entwickeln, der exakt mit der in Abschnitt 3.3 vorgestellten Syntax arbeitet. ArangoDB verfügt jedoch über ein Web-Interface, das durch die Möglichkeit des Erstellens von JSON-Dokumenten und des Verbindens der Dokumente untereinander als Graph, der vorgeschlagenen Syntax aus Abschnitt 3.3 sehr nahe kommt. Das Gesprächsmodell mit den Zielen für den Medolution-Anwendungsfall, das zur Evaluation

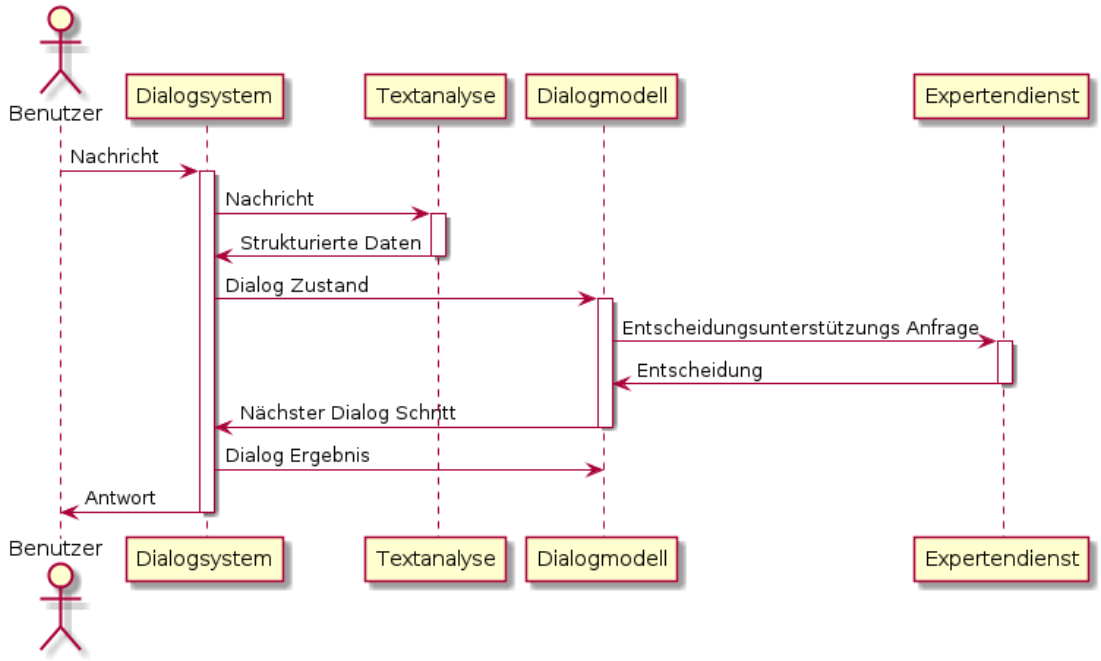


Abbildung 4.3: Sequenz-Diagramm während der Generierung einer Antwort an den Benutzer

verwendet wird, ist deshalb auch mit diesem Web-Interface entworfen worden. In der in Abschnitt 3.3 entworfenen Syntax ist es in vereinfachter Form in Anhang B gemeinsam mit der entscheidungsunterstützenden Ontologie zu finden, wo beides auch ebenfalls kurz erläutert wird.

5 Evaluation

Das in Anhang B dargestellte Dialogmodell wird in diesem Kapitel als Instanziierung verwendet, um das bisher entworfene System für zielorientierte Chatbots zu evaluieren. Bei einer empirischen Untersuchung der Benutzerfreundlichkeit werden einige Probanden einen Dialog mit dem Chatbot führen, den sie anhand der System Usability Scale [5] bewerten. Einige Testteilnehmer werden experimentell mit dem Chatbot auf ihre persönliche Art kommunizieren und ihre Erfahrungen bewerten. Dadurch wird die Interaktion des Chatbots mit unterschiedlichen Benutzertypen untersucht. Diese haben unterschiedliche Fehlerpotenziale und dabei wird auch die Möglichkeit geboten, einen Zusammenhang zwischen der Erfahrung der Benutzer mit Technologie, sozialen Netzwerken und weiteren relevanten Aspekten und dem Empfinden der Zugänglichkeit des Chatbots zu analysieren. Im Folgenden wird zuerst das Verfahren für diese empirische Untersuchung vorgestellt, die System Usability Scale, sowie der konkrete Fragebogen, welchen die Probanden ausfüllen werden. Anschließend werden die Ergebnisse, die sich aus einer Analyse der ausgefüllten Fragebögen ableiten lassen, präsentiert und erläutert.

5.1 Erläuterung der System Usability Scale und des verwendeten Fragebogens

Die System Usability Scale wurde von John Brooke mit dem Ziel entwickelt, eine Möglichkeit zu haben, im industriellen Umfeld unabhängig von der Einsatzdomäne, schnell und somit kosteneffizient die Benutzerfreundlichkeit von Systemen analysieren zu können [5]. Es werden 10 Fragen gestellt, die jeweils auf einer Skala zwischen 1 - *Entschieden Widersprechen* und 5 - *Volle Zustimmung* beantwortet werden. Alle diese Punktzahlen werden dann je nach zugehöriger Frage unterschiedlich gewichtet aufaddiert. Jede Frage erhält einen Wert zwischen 0 und 4, indem von der angekreuzten Antwort 1 subtrahiert wird. Die Werte der positiv gestellten Fragen (1, 3, 5, 7) werden aufaddiert und bei den negativ gestellten Fragen (2, 4, 6, 8) wird von 5 der Wert der Frage subtrahiert. Das Ergebnis dieser Addition wird mit 2,5 multipliziert, und es ergibt sich insgesamt eine Gesamtpunktzahl zwischen 0 und 100, wobei eine höhere Zahl ein besseres Ergebnis repräsentiert. Die 10 Fragen wurden frei aus dem Englischen übersetzt, da es sich um deutsche Probanden handelt, und lauten somit wie folgt:

1. Mir würde es gefallen, dieses System regelmäßig zu benutzen
2. Ich empfand das System als komplexer, als es angebracht wäre

3. Ich empfand die Benutzung des Systems insgesamt als einfach
4. Ich glaube, dass ich die Hilfe eines technischen Experten bräuchte, um dieses System benutzen zu können
5. Die Funktionen dieses Systems wirkten auf mich gut integriert
6. Ich glaube, das System enthält zu viele Unstimmigkeiten
7. Ich könnte mir vorstellen, dass die meisten Leute schnell lernen würden, mit diesem System umzugehen
8. Die Benutzung des Systems erschien mir umständlich
9. Ich hatte das Gefühl, das System souverän benutzen zu können
10. Ich musste viele Dinge lernen, bevor ich dieses System benutzen konnte

Diese Fragen werden von der jeweiligen Testperson ausgefüllt kurz nachdem sie den Chatbot benutzt hat. In dieser Evaluation werden die Probanden sich in einem Rollenspiel in die Rolle eines Patienten mit einem Kunstherz, also einem Mitglied der Zielgruppe des Medolution-Chatbots, versetzen. Innerhalb dieser Rolle haben sie die Aufgabe zu erfüllen, dem Chatbot einige medizinische Werte mitzuteilen. Zuerst werden den Probanden folgende allgemeine Fragen zu ihrer Person gestellt:

- Geschlecht?
- Alter?
- Beruflicher/fachlicher Hintergrund?
- Wie viel Erfahrung haben Sie mit Computern und Smartphones von 1 - *Absoluter Neuling* bis 10 - *Erfahrener Anwender*?
- Wie viel Erfahrung haben Sie mit sozialen Netzwerken und Chat-Plattformen von 1 - *Keinerlei Erfahrungen* bis 10 - *Täglicher Benutzer*?

Die Aufgabenstellung für das eigentliche Experiment wird auf dem Fragebogen durch den folgenden Text erläutert:

„ *Stellen Sie sich vor, Sie wären Maria Müller. Maria ist eine 57 Jahre alte Frau mit einem Herzfehler und dadurch Patientin mit einem Kunstherz. Da Kunstherzen sehr sensibel sind, ist es wichtig, dass Ärzte ein paar von Marias medizinischen Werten im Auge behalten, damit mögliche Risiken schnell entdeckt und behandelt werden können. Bisher sollte Maria ihre medizinischen Werte telefonisch an Mitarbeiter Ihres Arztes weitergeben, aber jetzt wird dafür alternativ ein Chatbot angeboten. Sie werden gleich in dem Experiment als Maria mit diesem Chatbot chatten und haben die Aufgabe, folgende medizinische Werte zu übermitteln:*

- *Benutzer-ID des Systems: 124*
- *INR-Wert: 4 (gemessen um 13 Uhr)*
- *Letzte Mahlzeit: Kartoffeln und Spinat (zum Mittagessen um 12 Uhr)*
- *Gewicht: 63 Kilogramm (zuletzt gestern gewogen)*

Je nachdem wie Sie mit dem Chatbot interagieren, kann es vorkommen, dass der Chatbot Ihnen Fragen stellt, deren Antwort in diesem Text nicht erläutert wird. In einem solchen Fall können Sie sich entweder eine passende Antwort ausdenken oder dem Chatbot sagen, dass Sie es nicht wissen. Sobald das Experiment gestartet ist, darf der Versuchsleiter Ihnen keine Fragen mehr beantworten. Versuchen sie zuerst die Aufgabe zu erfüllen und dem Chatbot Ihre medizinischen Werte mitzuteilen. Bitte beantworten Sie anschließend die Fragen zu Ihrer Erfahrung mit dem System auf der nächsten Seite. Viel Erfolg! “

Der Versuchsablauf sieht wie folgt aus:

1. Die Probanden füllen die allgemeinen Fragen zu ihrer Person aus.
2. Die Probanden lesen die Aufgabenstellung und haben die letzte Möglichkeit, dem Versuchsleiter Fragen zu stellen.
3. Die Probanden führen den Dialog mit dem Chatbot und versuchen, die Aufgabe zu erfüllen.
4. Die Probanden füllen abschließend die Fragen der System Usability Scale aus.

Dieser Versuchsablauf wird von mehreren Probanden durchgeführt und die Ergebnisse im Folgenden weiter diskutiert.

5.2 Analyse der Ergebnisse des Fragebogens

An dem zuvor beschriebenen Experiment haben insgesamt 28 Probanden teilgenommen. Diese waren im Schnitt 25,142 Jahre alt und die Aufteilung nach Geschlechtern entsprach 57 % weiblich und 43 % männlich. Nach eigenen Angaben lag die Vorerfahrung mit Computern und Smartphones durchschnittlich bei 6,928 und bei sozialen Netzwerken und Chat-Plattformen bei 8,321. Die Probanden können also mit einem Computer oder Smartphone im Durchschnitt mittelmäßig bis gut umgehen und die Benutzung von sozialen Netzwerken oder Chat-Plattformen findet annähernd täglich statt. Wenn die fachlichen beziehungsweise beruflichen Hintergründe in die Kategorien Technik/Mechanik, Wirtschaft, Verwaltung/Öffentlicher Dienst, Kulturwissenschaft, Sozialwissenschaft und Naturwissenschaft

eingeteilt werden, ergibt sich bei den Teilnehmern des Experimentes folgende Aufteilung:

Kategorie	Absolut	Relativ
Technik/Mechanik	4	14,286 %
Wirtschaft	4	14,286 %
Verwaltung/Öffentlicher Dienst	4	14,286 %
Kulturwissenschaft	6	21,429 %
Sozialwissenschaft	3	10,143 %
Naturwissenschaft	7	25,000 %

Die Verteilungen der Antworten der Probanden auf die 10 Fragen lassen sich gut in Abbildung 5.1 erkennen. Wie den Boxplots dort zu entnehmen ist, ist das obere Quartil bei den negativ gestellten Fragen gegen 1 bis 2 orientiert und bei den positiv gestellten Fragen das untere Quartil bei 4-5. Schwankungen sind eher minimal und Ausreißer treten auch nur vereinzelt auf. Diese Ergebnisse lassen auf ein einheitliches positives Empfinden des Chatbots bei den Probanden schließen.
Die Endergebnisse des Chatbots auf der System Usability Scale lagen im Bereich

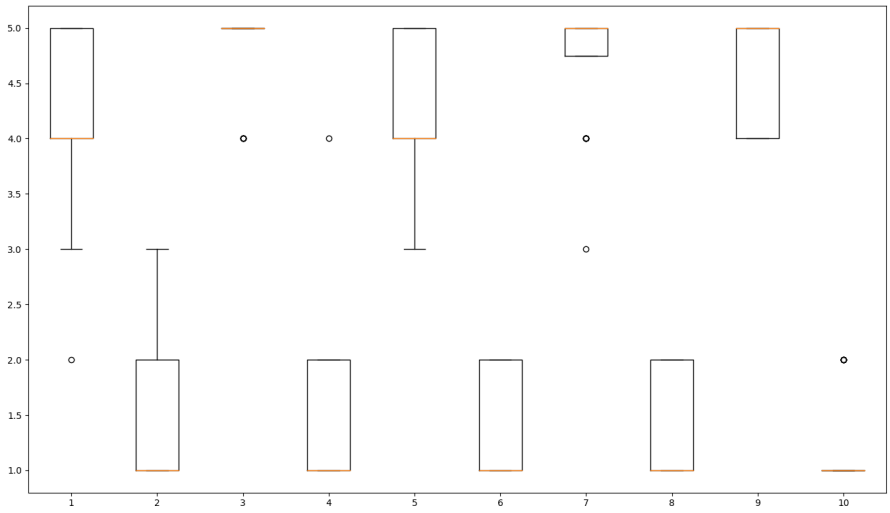


Abbildung 5.1: Boxplots der Antworten aller Probanden auf die 10 Fragen

zwischen 72,5 und 100. Durchschnittlich lagen sie bei 89,107. Wie in Abbildung 5.2 der Boxplot aller Gesamtergebnisse zeigt, fokussiert sich eine Mehrheit der Scores auf den Bereich zwischen 85 und 95. Eine empirische Studie mit über 2300 veröffentlichten Ergebnissen von Durchführungen der System Usability Scale hat

ergeben, dass diese im Durchschnitt ein Gesamtergebnis von 70,14 hatten [2]. Sowohl das durchschnittliche Gesamtergebnis von 89,107 als auch das schlechteste Gesamtergebnis von 72,5 liegen über diesem Durchschnitt der Studie. Aufgrund dieser Ergebnisse kann man bei der Auswertung der Bedienbarkeit des Chatbot von einem sehr guten Ergebnis sprechen.

Weiterhin wurde nach Korrelationen gesucht, um mögliche statistische Zusam-

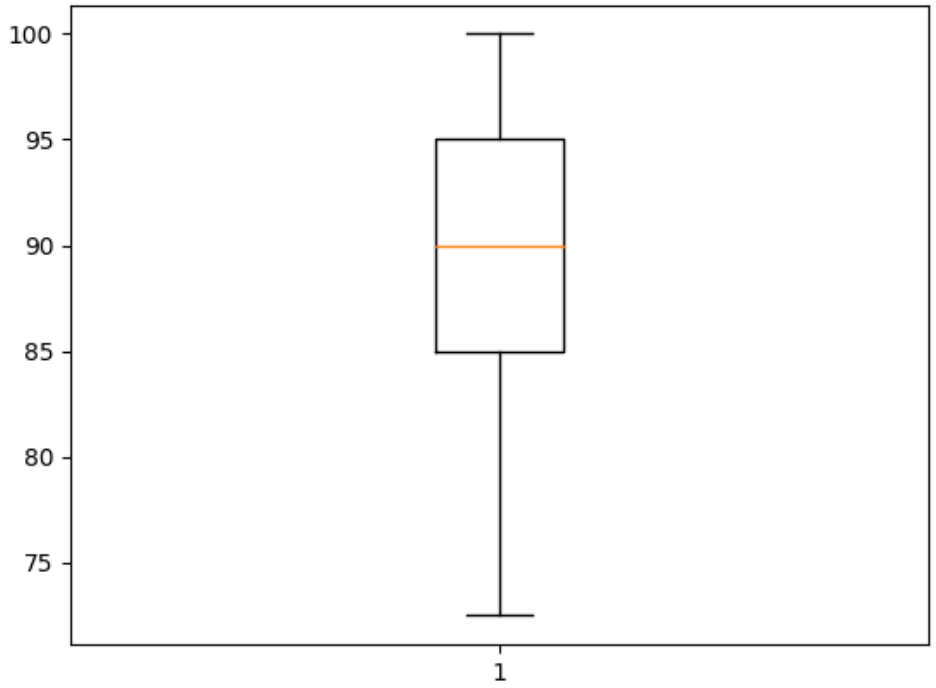


Abbildung 5.2: Boxplot aller Gesamtergebnisse der Probanden

menhänge zwischen den Eigenschaften des Benutzers wie zum Beispiel Alter oder Vorerfahrungen und seiner Bewertung des Chatbots zu finden. Zuerst wurde allgemein überprüft, ob das Alter eines Probanden a_i einen Einfluss auf des Endergebnis der System Usability Scale s_i des jeweiligen Probanden gehabt haben könnte. Hierfür wurde mit der Pearson-Korrelation ρ und den beiden Stichproben A und S der Korrelationskoeffizient $\rho_{A,S} \approx -0,157$ berechnet. Die Pearson-Korrelation wurde hier gewählt, da es sich bei dem Alter einer Person um eine Verhältnisskala handelt. Durch einen Korrelationskoeffizienten, der sehr nahe bei 0 liegt, lässt sich schließen, dass bei dieser Evaluation das Alter der Probanden keinen signifikanten Einfluss auf das Endergebnis der System Usability Scale hatte. Ob der Chatbot generell unabhängig von dem Alter des Benutzers zugänglich ist, sollte in einer größeren Untersuchung mit einem höheren Altersdurchschnitt weiter untersucht werden.

Weiterhin wurden die Stichproben für die Vorerfahrung mit Computern und Smartphones C beziehungsweise mit sozialen Netzwerken und Chat-Plattformen N im Hinblick darauf betrachtet, ob sie eine Korrelation zu den Antworten auf einige gezielt darauf anspielende Fragen der System Usability Scale haben könnten. Folgende Fragen wurden betrachtet:

- Mir würde es gefallen, dieses System regelmäßig zu benutzen (R).
- Die Funktionen dieses Systems wirkten auf mich gut integriert (I).
- Ich könnte mir vorstellen, dass die meisten Leute schnell lernen würden, mit diesem System umzugehen (L).
- Die Benutzung des Systems erschien mir umständlich (U).

Hierbei wurde die Kendall-Rangkorrelation τ verwendet, da es sich bei den persönlichen Empfindungen als Antworten auf die einzelnen Fragen um eine Ordinalskala handelt. Folgende Koeffizienten wurden berechnet:

- $\tau_{C,R} \approx 0,214$
- $\tau_{N,R} \approx 0,035$
- $\tau_{C,I} \approx 0,051$
- $\tau_{N,I} \approx 0,179$
- $\tau_{C,E} \approx 0,004$
- $\tau_{N,E} \approx -0,069$
- $\tau_{C,U} \approx -0,093$
- $\tau_{N,U} \approx -0,231$

Auch hier ließen sich also keine signifikanten Zusammenhänge bei dem durchgeführten Experiment finden.

5.3 Nicht-empirische Beobachtungen

Unabhängig von statistischen Ergebnissen waren jedoch noch weitere Verhaltensweisen der Probanden auffällig. Um diese Auffälligkeiten darzustellen, werden im Folgenden auch Ausschnitte aus Dialogen von Benutzern gezeigt, die hierbei in ihrer ursprünglichen Formulierung belassen wurden.
In keinem Dialog wurde eine `EXPLAIN`-Reaktion benötigt, weil keiner der Probanden um eine Erklärung gebeten hat oder anders ausgedrückt hat, dass er eine Nachricht nicht versteht. Es bleibt unklar, ob die Ursache hierfür ist, dass durch den Einleitungstext des Fragebogens keine Fragen mehr offen waren, oder ob es

den Probanden nicht bewusst war, dass es möglich gewesen wäre, sich Fragen genauer erklären zu lassen.
Am Anfang des Dialoges wurden die Probanden nach ihrem Wohlbefinden gefragt, wobei eine Antwort darauf nicht durch die Aufgabenstellung vorgegeben war, sondern den Probanden freigestellt war. Von allen 28 Teilnehmern hat nur ein einziger geantwortet, dass es ihm nicht gut gehen würde:

Chatbot: Hallo Maria Müller. Wie geht es Ihnen heute?
Benutzer: Mir geht es nicht so gut.

Gängiger waren kurze, positive Antworten der Probanden wie zum Beispiel:

Chatbot: Hallo Maria Müller. Wie fühlen Sie sich aktuell?
Benutzer: ganz gut

Es lässt sich hier als Verhaltensmuster vermuten, dass die Teilnehmer möglicherweise entweder unangenehmen Smalltalk über die eigenen Gefühle vermeiden oder vielleicht den Chatbot nicht mit ihren Sorgen belasten wollten.
Weiterhin haben Probanden mit einem zunehmenden Alter teilweise angefangen, ausschweifendere und höflichere Formulierungen und Phrasen zu verwenden, beispielsweise:

Chatbot: Hallo Maria Müller. Wie geht es Ihnen heute?
Benutzer: Mir geht es sehr gut, dankeschön!

Sie haben sich in einzelnen Situationen bedankt oder entschuldigt, obwohl solche höflichen Nachrichten erkennbar keinen Einfluss auf das Verhalten des Chatbots hatten.
Es ist auch vorgekommen, dass Benutzer sich gegenüber dem Chatbot passiv aggressiv gezeigt haben, wenn die Textauswertung nicht fehlerfrei funktioniert hat. So ein Verhalten würde zwar bei einem menschlichen Gesprächspartner eine Reaktion auslösen, aber der Chatbot kann solche Verhaltensweisen nicht erkennen und sich deshalb auch nicht dadurch beeinflussen lassen. Zum Beispiel:

Chatbot: Ihr INR-Wert ist außerhalb des normalen Bereichs. Haben Sie in letzter Zeit Medikamente eingenommen?
Benutzer: Ich habe heute mittag um 13 Uhr 2 Tabletten Marcumar eingenommen.
Chatbot: In was für einer Dosierung haben Sie dieses Medikament eingenommen?
Benutzer: Wie ich bereits sagte habe ich 2 Tabletten eingenommen.

Zusätzlich hat die Oberfläche des Chatbots das Eingabefeld für Nachrichten deaktiviert, sobald der Chatbot das Gespräch als abgeschlossen erachtet und sich verabschiedet hat, woraufhin einige Probanden sich enttäuscht gezeigt haben, dass

sie sich nicht ebenfalls verabschieden konnten.
Alle diese Verhaltensweisen deuten auf eine gute Akzeptanz des Chatbots bei den Teilnehmern hin, da es sich um Umgangsformen aus der zwischenmenschlichen Kommunikation handelt. Dass solche Verhaltensweisen auf den Chatbot übertragen wurden kann darauf hin deuten, dass er von den Probanden ansatzweise vermenschlicht und somit zum Teil als gleichwertiger Gesprächspartner akzeptiert wurde. Um diese Vermutungen jedoch empirisch zu bestätigen, wären weitere Untersuchungen nötig.

6 Zusammenfassung und Ausblick

Abschließend wird ein Fazit aus dieser Arbeit gezogen und ein Ausblick auf weitere offene Punkte und Fragestellungen beziehungsweise die sich dadurch ergebenden Verbesserungsmöglichkeiten des Chatbot-Systems gegeben.

6.1 Zusammenfassung des entwickelten Systems, der Ergebnisse der Evaluation und der daraus gewonnenen Erkenntnisse

Das Ziel dieser Arbeit war es, ein zugängliches Verfahren zur Dialogmodellierung zu entwickeln, das Domänenexperten ohne Programmiererfahrung ermöglicht, zielorientierte Chatbots in ihren jeweiligen Domänen einzusetzen. Dies wurde durch die Entwicklung eines hybriden Modells aus zustandsbasierten und regelbasierten Verfahren erreicht. Entworfene Dialogmodelle lassen sich dynamisch und vollständig mit allen modellierten Elementen durch den Chatbot durchführen, und im Hinblick auf die Benutzerfreundlichkeit beim menschlichen Gesprächspartner ergab sich mit einem durchschnittlichen Ergebnis auf der System Usability Scale von 89,107 auch ein sehr guter Wert. Weiterhin ließen sich keine signifikanten Korrelationen zwischen dem Alter der Benutzer sowie ihrer Vorerfahrung mit Computern, Smartphones, sozialen Netzwerken und Chat-Plattformen und der Benutzerfreundlichkeit des Chatbots finden. Die daraus resultierende Annahme, dass sich die zielorientierten Chatbots dieser Arbeit von unterschiedlichen Benutzergruppen unabhängig von Alter und Vorerfahrung gut bedienen lassen, sollte jedoch in einem größer angelegten Experiment weiter untersucht werden.
Die Referenzimplementierung dieser Arbeit für ein solches System kann durch ihre modularisierte Architektur in Zukunft unkompliziert erweitert und verbessert werden. Wenn zum Beispiel ein performanterer Ontologie-Service, als der in dieser Arbeit verwendete, eingesetzt werden soll, kann ein anderer über passende Schnittstellen problemlos integriert werden. Weiterhin sorgt die Modularisierung zum Beispiel bei der Vereinigung der Datenhaltung und der dazugehörigen Datenlogik für eine gute Skalierbarkeit. Dieser Aspekt der *Separation of Concerns* [13] hat sich hier aus einer Architektur-Perspektive als guter Leitfaden für die Entwicklung erwiesen.

6.2 Ausblick auf weitere Verbesserungsmöglichkeiten für zielorientierte Chatbots

Das in dieser Arbeit entwickelte System hat sich primär auf das Entwerfen der Ziele innerhalb des Dialogmodells und die passende Dialogstrategie für das Erreichen dieser Ziele konzentriert. Außerhalb der Entwicklungen aus dieser Arbeit gibt es Möglichkeiten, sowohl die Modellierung der Dialoge als auch weitere Komponenten des Gesamtsystems weiter zu optimieren. Ansätze für einige dieser Möglichkeiten werden hier zuletzt vorgestellt und sollen einen Ausblick darauf geben, wie sich das Chatbot-System und der theoretische Ansatz dieser Arbeit weiter entwickeln lassen.
Das Dialogmodell an sich hat Möglichkeiten für weitere Interaktionselemente wie zum Beispiel Nachrichten an den Benutzer, die geschickt werden, wenn eine Kante zwischen Zielen verfolgt wird. Weiterhin wäre eine Form von Feedback informativ, wenn ein Ziel abgeschlossen wird, das keine Nachfolger hat und somit ein Thema innerhalb des Dialoges abgeschlossen ist. In einem solchen Fall würde aktuell direkt mit dem nächsten offenen Ziel fortgefahren werden, das in einem vollständig anderen Pfad des Dialoggraphs liegen könnte, und dem Benutzer könnte die Verbindung zwischen dem alten, abgeschlossenen Ziel und dem neuen Ziel nicht sofort erkennbar sein. Ebenfalls ermöglichen es Dialogmodell und Dialogstrategie in ihrer aktuellen Version nicht, abgeschlossene Ziele wieder gezielt zu öffnen, wenn zum Beispiel der Benutzer einen übermittelten Wert korrigieren möchte. Es fällt ebenfalls auf, dass häufig Ziele ähnlich gestaltet sind. Zum Beispiel hat die Abfrage von medizinischen Werten in der Regel einen Wert und ein Messdatum. Für solche Fälle würde sich ein Vererbungsmechanismus für ähnliche Ziele anbieten, bei denen erst gemeinsame Eigenschaften einmalig in schematischen Prototypen beschrieben werden und die konkreten Unterschiede dann spezialisiert in den Kinder-Zielen definiert werden. Zusätzlich könnten auch globale Events für das komplette Gespräch nützlich sein. Bei den globalen Events wird nicht der eigentliche Ablauf des modellierten Gesprächs geändert, sondern nur einmalig auf das Event reagiert und danach das Gespräch mit dem eigentlichen fokussierten Ziel fortgesetzt. So könnte zum Beispiel die Beantwortung von FAQs zur Bedienung des Chatbots oder Ähnlichem einfach integriert werden, ohne dass die Beantwortung dieser Fragen den Gesprächsverlauf vollständig unterbrechen würde. Bei der Version des Chatbots dieser Arbeit wird überprüft, ob ein Thema fallengelassen werden soll, wenn der Benutzer nicht auf das Thema eingeht. Alternativ könnte man in so einem Fall auch eine Antwort, die durch ein maschinelles Lernverfahren generiert wurde, zurück schicken. Damit kann in solchen Fällen Smalltalk geführt werden, der nicht vorher modelliert werden muss, bis der Benutzer wieder auf das eigentliche aktuelle Ziel eingeht.
In dieser Arbeit wurde nur eine vereinfachte Form der Auswertung von natürlicher Sprache betrachtet. Mit dem aktuellen Stand der Forschung sind technisch weitere

Arten der Auswertung außer der Klassifizierung der Intention und dem Erkennen der Entitäten möglich. Das Einbinden von weiteren dieser Sprachauswertungsmethoden könnte die Interaktion mit dem Benutzer vereinfachen, da dieser eine größere Freiheit hätte, wie er seine Nachrichten formulieren kann. Wenn zum Beispiel aus einer komplexeren Nachricht nur die Entitäten extrahiert werden und nicht die Beziehung dieser untereinander, wird es bei zu vielen Entitäten pro Satz nicht mehr auswertbar. Beispielsweise werden aus einem Satz Messwerte und ihre jeweiligen Maßeinheiten extrahiert. Sobald nun innerhalb eines Satzes mehr als ein Paar aus Messwert und Maßeinheit auftreten, wird es aufwändig, allein mit einem NER-Verfahren zuzuordnen, welcher Wert zu welcher Maßeinheit gehört. Für solche Fälle sollte die Sprachauswertung zusätzlich die Abhängigkeiten zwischen den Entitäten extrahieren oder sogar einen kompletten Syntax-Baum aufbauen. Hier müsste dann auch die Art der Definition der Slots des Dialogmodells entsprechend angepasst werden.
Eine stärkere Einbindung von Ontologien bietet ebenfalls Potenzial für den Chatbot. Unbekannte Wörter während der Sprachauswertung könnten dadurch zugeordnet werden. Ebenfalls wäre es denkbar, logische Zusammenhänge wie die benötigten Slots eines Ziels oder die Bedingung eines Aktivators oder Revisors, direkt automatisch aus einer gegebenenfalls vorhandenen Ontologie extrahieren zu können. In einem solchen System müsste ein Modellierer diese Elemente nicht mehr manuell entwerfen, sondern es würde reichen, wenn er ein Dialogziel zu passenden Knoten in einem Wissensgraph verknüpfen würde. Technisch wäre es weiterhin möglich, dass eine Ontologie über den Benutzer aufgebaut wird, mit der dann der nächste Dialog angereichert wird. Wenn zum Beispiel der Benutzer bei einem Dialog äußert, dass er unter Kopfschmerzen leidet, könnte dies als Krankheit des Benutzers in einer Ontologie gespeichert werden und bei dem nächsten Dialog gefragt werden, ob seine Kopfschmerzen besser geworden sind.
Insgesamt gibt es also einige Ansätze, um sowohl das Modellieren der Dialoge zu vereinfachen als auch die daraus resultierende Dialoge zu verbessern. Die vorliegende Arbeit hat für diese Ideen eine gute Grundlage in Form eines theoretischen Modellierungsansatzes und eines modularen Gesamtsystems geliefert.

Literaturverzeichnis

[1] Monica Agrawal, Janette Cheng, and Caelin Tran. What's up, doc? a medical diagnosis bot. 2017.

[2] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale. *International Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[3] Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry S. Thompson, and Terry Winograd. GUS, A frame-driven dialog system. *Artif. Intell.*, 8(2):155–173, 1977.

[4] Eric Brill. A simple rule-based part of speech tagger. In *3rd Applied Natural Language Processing Conference, ANLP 1992, Trento, Italy, March 31 - April 3, 1992*, pages 152–155, 1992.

[5] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[6] Coral Calero, Francisco Ruiz, and Mario Piattini, editors. *Ontologies for Software Engineering and Software Technology*. Springer, 2006.

[7] Michael Fischer and Monica Lam. From books to bots: Using medical literature to create a chat bot. In *Proceedings of the First Workshop on IoT-enabled Healthcare and Wellness Technologies and Systems*, IoT of Health '16, pages 23–28, New York, NY, USA, 2016. ACM.

[8] Kara Kathleen Fitzpatrick, Alison Darcy, and Molly Vierhile. Delivering cognitive behavior therapy to young adults with symptoms of depression and anxiety using a fully automated conversational agent (woebot): A randomized controlled trial. *JMIR Ment Health*, 4(2):e19, Jun 2017.

[9] Herve Frezza. Support vector machines tutorial. *Frezza. Buet@ supe. lec. fr*, 2013.

[10] H Paul Grice. Logic and conversation. *1975*, pages 41–58, 1975.

[11] Stefan W. Hamerich. *Sprachbedienung im Automobil: Teilautomatisierte Entwicklung benutzerfreundlicher Dialogsysteme*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2009.

[12] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. Springer-Verlag, 2007.

[13] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995.

[14] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[15] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Machine Learning: ECML-98, 10th European Conference on Machine Learning, Chemnitz, Germany, April 21-23, 1998, Proceedings*, pages 137–142, 1998.

[16] Esther Levin, Roberto Pieraccini, and Wieland Eckert. Learning dialogue strategies within the markov decision process framework. In *Automatic Speech Recognition and Understanding, 1997. Proceedings., 1997 IEEE Workshop on*, pages 72–79. IEEE, 1997.

[17] Michael L Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119–125, 2009.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[19] Roger C. Parkison, Kenneth Mark Colby, and William S. Faught. Conversational language comprehension using integrated pattern-matching and parsing. *Artif. Intell.*, 9(2):111–134, 1977.

[20] Wolfgang Schneider. Grundsätze der Dialoggestaltung DIN EN ISO 9241-110. *URL: http://www.ergo-online.de/site.aspx (accessed May 2017)*, 2006.

[21] Burr Settles. Biomedical named entity recognition using conditional random fields and rich feature sets. In *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications*, pages 104–107. Association for Computational Linguistics, 2004.

[22] David R Traum and Staffan Larsson. The information state approach to dialogue management. In *Current and new directions in discourse and dialogue*, pages 325–353. Springer, 2003.

[23] Jason D. Williams and Steve J. Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech & Language*, 21(2):393–422, 2007.

[24] Xuesong Yang, Yun-Nung Chen, Dilek Z. Hakkani-Tür, Paul Crook, Xiujun Li, Jianfeng Gao, and Li Deng. End-to-end joint learning of natural language understanding and dialogue manager. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*, pages 5690–5694, 2017.

[25] Steve J. Young, Milica Gasic, Simon Keizer, François Mairesse, Jost Schatzmann, Blaise Thomson, and Kai Yu. The hidden information state model: A practical framework for pomdp-based spoken dialogue management. *Computer Speech & Language*, 24(2):150–174, 2010.

Abbildungsverzeichnis

Anhänge

A Implementierung der Gesprächsstrategie

```
1  'use strict'
2
3  const rp = require('request-promise')
4
5  function InformationManager () {
6    // Activated but not filled nodes of the current dialog
7    let openNodes = []
8    // Successfully filled nodes of the current dialog
9    let closedNodes = []
10   // Information sets from previous user messages that could not
       be used yet
11   let unusedInformations = []
12   // Currently active node with all its metadata
13   let focusNode = null
14   // Entities of the currently active focus node
15   let focusEntities = null
16   // Has the dialog system started to check the focus node
17   let startedChecking = false
18   // ID of the user
19   let id
20
21   this.init = async function (dialogRootNode) {
22     // Initialize information state with the given start node
23     await loadFocusNode(dialogRootNode)
24     openNodes.push(dialogRootNode)
25   }
26
27   // Process the input sentence and try to fill nodes with it
28   this.fillSlots = async function (sentence) {
29     console.log('Slot filling for sentence: ' + sentence.text)
30     // Extract entities from the given sentence
31     let sentenceEntities = []
32     for (let entity of sentence.entities) {
33       sentenceEntities.push({
34         type: entity.entity,
35         submittedValue: entity.value
36       })
37     }
38     // Names of nodes that were newly activated in this processing
         iteration
39     let newNodeNames = []
```

```
40      // Has the sentence entitites that were unused in this
            processing interation
41      let unusedEntitiesExisting = false
42      // Answer that activators or revisors may give
43      let answer = null
44      // Can the focus node be changed by the while this processing
            iteration
45      let canChangeFocusNode = true
46      if (this.isDialogFinished()) {
47        console.log('No open nodes left')
48      } else {
49        // Since the dialog is not finished the sentence needs to be
              processed
50        let sentenceEntitiesClone = []
51        for (let e of sentenceEntities) {
52          sentenceEntitiesClone.push(Object.assign({}, e))
53        }
54        // Ask the dialog model if one of the open nodes is
              triggered by this
55        // sentence to detect topic changes
56        let triggerResult = await rp({
57          method: 'PUT',
58          uri: 'http://data_services:8529/dialog-services/
                dialog_nodes/triggered',
59          headers: { 'Content-Type': 'application/json' },
60          json: true,
61          body: {
62            nodes: openNodes,
63            input: sentenceEntities
64          }
65        })
66        if (triggerResult.triggered && (triggerResult.node != null)
              && (triggerResult.node !== focusNode.name)) {
67          // Change the focus node and entities if a topic change
                was detected
68          console.log(triggerResult.node + ' was triggered and will
                be the new focus node')
69          await loadFocusNode(triggerResult.node)
70        }
71        if (focusNode.intentReactions.hasOwnProperty(sentence.intent
              .name)) {
72          // React with the proper intent reaction defined in the
                focus node
73          switch (focusNode.intentReactions[sentence.intent.name]) {
74            case 'CHECK':
75              console.log(focusNode.name + ' reacts to ' + sentence.
                    intent.name + ' with CHECK of the entities: ' +
                    JSON.stringify(sentenceEntitiesClone))
76              // Save the intent that caused the reaction and check
                    the focus entities
77              focusNode.submittedIntent = sentence.intent.name
78              for (let keyEntity of focusEntities) {
```

```
79                // Try to fill this entity with the sentence
80                if (!keyEntity.hasOwnProperty('submittedValue') ||
                      keyEntity.submittedValue == null) {
81                  if (keyEntity.multiOccurence) {
82                    let submittedValue = []
83                    for (let i = 0; i < sentenceEntitiesClone.length
                        ; i++) {
84                      console.log('Try ' + sentenceEntitiesClone[i].
                          submittedValue + ' for ' + keyEntity.name
                          + ' with multi occurence')
85                      if (sentenceEntitiesClone[i].type ===
                          keyEntity.type) {
86                        submittedValue.push(sentenceEntitiesClone[i
                            ].submittedValue)
87                        console.log('Wrote in node ' + focusNode.
                            name + ' ' + sentenceEntitiesClone[i].
                            submittedValue + ' into ' + keyEntity.
                            name)
88                        sentenceEntitiesClone.splice(i, 1)
89                        i--
90                      } else {
91                        console.log('In focus node ' + focusNode.
                            name + ' was ' + sentenceEntitiesClone[i
                            ].submittedValue + ' of type ' +
                            sentenceEntitiesClone[i].type + ' not
                            fitting for ' + keyEntity.name)
92                      }
93                    }
94                    if (submittedValue.length > 0) {
95                      keyEntity.submittedValue = submittedValue
96                    }
97                  } else {
98                    for (let sentenceEntity of sentenceEntitiesClone
                        ) {
99                      console.log('Try ' + sentenceEntity.
                          submittedValue + ' for ' + keyEntity.name)
100                     if (sentenceEntity.type === keyEntity.type) {
101                       keyEntity.submittedValue = sentenceEntity.
                            submittedValue
102                       console.log('Wrote in node ' + focusNode.
                            name + ' ' + sentenceEntity.
                            submittedValue + ' into ' + keyEntity.
                            name)
103                       sentenceEntitiesClone.splice(
                            sentenceEntitiesClone.indexOf(
                            sentenceEntity), 1)
104                       break
105                     } else {
106                       console.log('In focus node ' + focusNode.
                            name + ' was ' + sentenceEntity.
                            submittedValue + ' of type ' +
                            sentenceEntity.type + ' not fitting for
```

```
                        ' + keyEntity.name)
107                   }
108                 }
109               }
110               // If a fitting value was found in the sentence
                     check the dialog model for possible revisors
111               if (keyEntity.submittedValue != null) {
112                 let revisorResult = await rp({
113                   method: 'PUT',
114                   uri: 'http://data_services:8529/dialog-
                         services/dialog_nodes/' + keyEntity.name +
                          '/check_revisor',
115                   headers: { 'Content-Type': 'application/json'
                         },
116                   json: true,
117                   body: {
118                     value: keyEntity.submittedValue,
119                     id: id
120                   }
121                 })
122                 // Delete the value if the revisor does not
                       accept it
123                 if (!revisorResult.successful) {
124                   keyEntity.submittedValue = null
125                 }
126                 // Save a possible answer from the revisor
127                 if (answer == null) {
128                   answer = revisorResult.answer
129                 }
130               }
131             }
132           }
133           // Check if the sentence contains entities that could
                 be used for triggers of the focus node
134           console.log('Request triggers for node ' + focusNode.
                 name + ' while entity checking')
135           let triggers = await rp({
136             method: 'GET',
137             uri: 'http://data_services:8529/dialog-services/
                    dialog_nodes/' + focusNode.name + '/triggers',
138             json: true
139           })
140           for (let trigger of triggers) {
141             for (let sentenceEntity of sentenceEntitiesClone) {
142               if ((sentenceEntity.type === trigger.type) &&
143                   ((!trigger.hasOwnProperty('expectedValue')) ||
144                     (trigger.expectedValue.indexOf(
                           sentenceEntity.submittedValue) !== -1)))
                           {
145                 console.log('Used in focus node ' + focusNode.
                       name + ' ' + sentenceEntity.type + ' for
                       trigger' + trigger.name)
```

```
146                     sentenceEntitiesClone.splice(
                            sentenceEntitiesClone.indexOf(sentenceEntity
                            ), 1)
147                     break
148                   } else {
149                     console.log('In focus node ' + focusNode.name +
                            ' ' + sentenceEntity.type + ' was not
                            fitting for ' + trigger.name)
150                   }
151                 }
152               }
153               // Check if there are filled focus entities
154               let filledFocusEntities = 0
155               for (let keyEntity of focusEntities) {
156                 if (keyEntity.submittedValue != null) {
157                   filledFocusEntities++
158                 }
159               }
160               // If no focus entity is filled and filling the focus
                     node was not started in previous processing
                     iterations
161               // presume the user does not want to talk about this
                     topic and check for a drop of this topic
162               if (filledFocusEntities === 0) {
163                 console.log('User did not give a needed entity for '
                        + focusNode.name)
164                 if (startedChecking) {
165                   await checkFocusNodeForDrop()
166                 } else {
167                   startedChecking = true
168                 }
169               } else if (filledFocusEntities < focusEntities.length)
                      {
170                 canChangeFocusNode = false
171                 startedChecking = true
172                 console.log('Cannot change focus node anymore
                        because current focus node was not filled
                        completly')
173               } else {
174                 // If the focus node was filled completly try to
                        extract the user ID from it and add it to the
                        closed nodes
175                 startedChecking = false
176                 let entities = []
177                 for (let e of focusEntities) {
178                   entities.push({
179                     name: e.name,
180                     type: e.type,
181                     submittedValue: e.submittedValue
182                   })
183                 }
184                 if (focusNode.name === 'ID') {
```

```
185                 id = entities[0].submittedValue
186               }
187               closedNodes.push({
188                 name: focusNode.name,
189                 entities: entities,
190                 intent: sentence.intent.name
191               })
192               // Check if new nodes ware activated by the filled
                     entities or the submitted intent
193               console.log('Load from ' + focusNode.name + '
                     activated nodes because it was filled while
                     entity checking')
194               let activationResult = await rp({
195                 method: 'PUT',
196                 uri: 'http://data_services:8529/dialog-services/
                       dialog_nodes/' + focusNode.name + '/activates'
                       ,
197                 json: true,
198                 body: {
199                   id: id,
200                   entities: entities,
201                   intent: sentence.intent.name
202                 }
203               })
204               // Save a possible answer from the activator
205               if (answer == null) {
206                 answer = activationResult.answer
207               }
208               // Add newly activated nodes to the open nodes
209               for (let n of activationResult.nodes) {
210                 newNodeNames.push(n)
211               }
212               // If no node is activated load the next node from
                     the open nodes as the focus node
213               if (newNodeNames.length === 0) {
214                 await nextFocusNode()
215               } else {
216                 // Simply remove the filled focus node from the
                       open nodes otherwise
217                 console.log('Remove ' + openNodes[openNodes.
                       indexOf(focusNode.name)] + ' from open nodes
                       in CHECK reaction')
218                 openNodes.splice(openNodes.indexOf(focusNode.name)
                       , 1)
219               }
220             }
221             break
222           case 'ACCEPT':
223             // Save the submitted intent that caused this reaction
224             console.log(focusNode.name + ' reacts to ' + sentence.
                   intent.name + ' with ACCEPT')
225             focusNode.submittedIntent = sentence.intent.name
```

```
226             // Check if closing this node activated other nodes
227             console.log('Load from ' + focusNode.name + '
                    activated nodes because it was accepted')
228             let activationResult = await rp({
229               method: 'PUT',
230               uri: 'http://data_services:8529/dialog-services/
                      dialog_nodes/' + focusNode.name + '/activates',
231               json: true,
232               body: {
233                 id: id,
234                 entities: [],
235                 intent: sentence.intent.name
236               }
237             })
238             // Save a possible answer from the activator
239             if (answer == null) {
240               answer = activationResult.answer
241             }
242             // Add newly activated nodes to the open nodes
243             for (let n of activationResult.nodes) {
244               newNodeNames.push(n)
245             }
246             // Add the node with the submitted intent to the
                    closed nodes
247             closedNodes.push({
248               name: focusNode.name,
249               entities: [],
250               intent: sentence.intent.name
251             })
252             // If no node is activated load the next node from the
                    open nodes as the focus node
253             if (newNodeNames.length === 0) {
254               await nextFocusNode()
255             } else {
256               // Simply remove the filled focus node from the open
                      nodes otherwise
257               console.log('Remove ' + openNodes[openNodes.indexOf(
                      focusNode.name)] + ' from open nodes IN ACCEPT
                      reaction')
258               openNodes.splice(openNodes.indexOf(focusNode.name),
                      1)
259             }
260             startedChecking = false
261             break
262           case 'EXPLAIN':
263             console.log(focusNode.name + ' reacts to ' + sentence.
                    intent.name + ' with EXPLAIN')
264             // Instruct the dialog system to send the user the
                    explaination message of the focus node as the
                    answer
265             answer = 'EXPLAIN'
266             startedChecking = false
```

```
267             break
268           case 'CLOSE':
269             console.log(focusNode.name + ' reacts to ' + sentence.
                    intent.name + ' with CLOSE')
270             // Remove this node and load the next one from the
                    open nodes
271             await nextFocusNode()
272             startedChecking = false
273             break
274           case 'ASKAGAIN':
275             console.log(focusNode.name + ' reacts to ' + sentence.
                    intent.name + 'with ASKAGAIN')
276             // Instruct the dialog system to send the current
                    prompt or onMissing message again
277             answer = 'ASKAGAIN'
278             startedChecking = false
279             break
280           default:
281             console.log(focusNode.name + ' has the unknown
                    reaction type ' + focusNode.intentReactions[
                    sentence.intent.name] + ' for the intent ' +
                    sentence.intent)
282             // Since there is a unknown reaction type defined the
                    focus node will be checked for a drop
283             startedChecking = false
284             await checkFocusNodeForDrop()
285         }
286         console.log('Unused sentence entities: ' + JSON.stringify(
                sentenceEntitiesClone))
287         unusedEntitiesExisting = sentenceEntitiesClone.length > 0
288       } else {
289         // User did not reacted as expected and a drop has to be
                considered
290         startedChecking = false
291         console.log(focusNode.name + ' has no reaction defined for
                ' + sentence.intent.name + '. A drop will be
                considered.')
292         await checkFocusNodeForDrop()
293         console.log('Unused sentence entities: ' + JSON.stringify(
                sentenceEntitiesClone))
294         unusedEntitiesExisting = sentenceEntitiesClone.length > 0
295       }
296
297       // Check if there are unused informations that can be used
              to fill other open nodes
298       if (unusedEntitiesExisting && sentenceEntities.length > 0) {
299         console.log('Sentence has unused informations left and it
                will be tried to fill other open nodes with them')
300         let sentenceEntitiesUsed = false
301         // Try to fill on of the open nodes with this sentence
302         for (let openNode of openNodes) {
303           console.log('Load fill result for open node ' + openNode
```

```
        )
304     let fillResult = await rp({
305       method: 'PUT',
306       uri: 'http://data_services:8529/dialog-services/
            dialog_nodes/' + openNode + '/fill',
307       headers: { 'Content-Type': 'application/json' },
308       json: true,
309       body: {
310         entities: sentenceEntities,
311         id: id
312       }
313     })
314     if (fillResult.canFill) {
315       // Open nodes could be filled completly and will be
            added to the closed nodes
316       openNodes.splice(openNodes.indexOf(openNode), 1)
317       closedNodes.push({
318         name: openNode,
319         entities: fillResult.filledEntities,
320         intent: sentence.intent.name
321       })
322       console.log(openNode + ' was filled and activated
            nodes will be loaded')
323       // Check if closing this node activated other nodes
324       let activationResult = await rp({
325         method: 'PUT',
326         uri: 'http://data_services:8529/dialog-services/
              dialog_nodes/' + openNode + '/activates',
327         json: true,
328         body: {
329           id: id,
330           entities: fillResult.filledEntities,
331           intent: sentence.intent.name
332         }
333       })
334       // Add newly activated nodes to the open nodes
335       for (let n of activationResult.nodes) {
336         newNodeNames.push(n)
337       }
338       // Save a possible answer from the activator or
            revisor
339       if (answer == null) {
340         answer = fillResult.anwer
341       }
342       if (answer == null) {
343         answer = activationResult.answer
344       }
345     } else if (fillResult.started) {
346       // The node could only be filled partially and it will
            be checked if it can be the new focus node
347       if (canChangeFocusNode) {
348         console.log(openNode + ' filling was started and it
```

```
                will be loaded as the new focus node')
349             // Load the open node as the new focus node
350             focusNode = await rp({
351               method: 'GET',
352               uri: 'http://data_services:8529/dialog-services/
                    dialog_nodes/' + openNode,
353               json: true
354             })
355             // Set the focus entity the filled and open entities
                    of the open node
356             focusEntities = fillResult.filledEntities.concat(
                    fillResult.openEntities)
357             console.log('New focus node: ' + JSON.stringify(
                    focusNode, null, 4))
358             console.log('New focus entities: ' + JSON.stringify(
                    focusEntities, null, 4))
359             console.log('Cannot change focus node anymore
                    because a focus only new node was made the new
                    focus node')
360             canChangeFocusNode = false
361             startedChecking = true
362             console.log('New focus node is ' + focusNode.name)
363             // Save a possible answer from the revisor
364             if (answer == null) {
365               answer = fillResult.anwer
366             }
367           }
368         }
369         // If this open node used all of the sentence entities
                at once consider this sentence as used and stop
370         if (fillResult.allUsed) {
371           sentenceEntitiesUsed = true
372           break
373         }
374       }
375       if (!sentenceEntitiesUsed) {
376         // No open node could use the whole sentence so it will
                be added to the unused informations
377         unusedInformations.push({
378           entities: sentenceEntities,
379           sentence: sentence.text
380         })
381       }
382     }
383   }
384
385   // Handle newly opened nodes
386   if (newNodeNames.length > 0) {
387     console.log('New nodes: ' + newNodeNames)
388     // If new nodes were activated check if they can be filled
          with unused informations
389     for (let name of newNodeNames) {
```

```
390           console.log('Load new node ' + name)
391           let newNode = await rp({
392             method: 'GET',
393             uri: 'http://data_services:8529/dialog-services/
                  dialog_nodes/' + name,
394             json: true
395           })
396           if (newNode.focusOnly) {
397             // Because the new node is focus only it is not allowed
                  to fill it outside of focus with unused informations
398             // It will be just added to the open nodes and (if
                  possible) loaded as the new focus node
399             openNodes.push(name)
400             console.log(newNode.name + ' needs focus to be filled')
401             if (canChangeFocusNode) {
402               await loadFocusNode(name)
403               console.log('Cannot change focus node anymore because
                    a focus only new node was made the new focus node'
                    )
404               canChangeFocusNode = false
405               startedChecking = false
406               console.log('New focus node is ' + focusNode.name)
407             }
408           } else {
409             // try to fill the new node with unused informations
410             let addedNewNodeToOpenNodes = false
411             console.log('New node ' + newNode.name + ' can be filled
                    outside of focus')
412             if (unusedInformations.length > 0) {
413               for (let informationSet of unusedInformations) {
414                 console.log('Load fill result for new node ' + name)
415                 let fillResult = await rp({
416                   method: 'PUT',
417                   uri: 'http://data_services:8529/dialog-services/
                        dialog_nodes/' + name + '/fill',
418                   headers: { 'Content-Type': 'application/json' },
419                   json: true,
420                   body: {
421                     entities: informationSet.entities,
422                     id: id
423                   }
424                 })
425                 if (fillResult.canFill) {
426                   // New node could be filled completly with this
                        unused information set
427                   closedNodes.push({
428                     name: name,
429                     entities: fillResult.filledEntities,
430                     intent: null
431                   })
432                   console.log('New node ' + name + ' was filled and
                        activated node names will be loaded')
```

```
433                 let activationResult = await rp({
434                   method: 'PUT',
435                   uri: 'http://data_services:8529/dialog-services/
                          dialog_nodes/' + name + '/activates',
436                   json: true,
437                   body: {
438                     id: id,
439                     entities: fillResult.filledEntities,
440                     intent: null
441                   }
442                 })
443                 // Add nodes that are activated by filling this
                       new node to this new nodes loop
444                 for (let n of activationResult.nodes) {
445                   newNodeNames.push(n)
446                 }
447                 // Save possible answers from revisors or
                       activators of this node
448                 if (answer == null) {
449                   answer = fillResult.anwer
450                 }
451                 if (answer == null) {
452                   answer = activationResult.answer
453                 }
454               } else if (fillResult.started) {
455                 // New node could be filled partially with the
                       unused information set
456                 // Add it to the open node (if not already
                       happened) and make it the focus node if
                       allowed
457                 if (!addedNewNodeToOpenNodes) {
458                   openNodes.push(name)
459                   addedNewNodeToOpenNodes = true
460                 }
461                 if (canChangeFocusNode) {
462                   console.log('New node ' + name + ' filling was
                          started and it will be loaded as the new
                          focus node')
463                   focusNode = await rp({
464                     method: 'GET',
465                     uri: 'http://data_services:8529/dialog-
                            services/dialog_nodes/' + name,
466                     json: true
467                   })
468                   focusEntities = fillResult.filledEntities.concat
                          (fillResult.openEntities)
469                   console.log('New focus node: ' + JSON.stringify(
                          focusNode, null, 4))
470                   console.log('New focus entities: ' + JSON.
                          stringify(focusEntities, null, 4))
471                   console.log('Cannot change focus node anymore
                          because a new node was made the new focus
```

```
                    node')
472                 canChangeFocusNode = false
473                 startedChecking = true
474                 console.log('New focus node is ' + focusNode.
                        name)
475                 if (answer == null) {
476                   answer = fillResult.anwer
477                 }
478                 break
479               }
480             }
481             if (fillResult.allUsed) {
482               // If this information set is used completly
                      remove it from the unused sets
483               unusedInformations.splice(unusedInformations.
                      indexOf(informationSet), 1)
484               break
485             }
486           }
487           if (!addedNewNodeToOpenNodes) {
488             // if the node was not added to the open nodes yet
                    add it now
489             console.log('New node ' + newNode.name + ' could not
                    be filled with unused informations')
490             openNodes.push(name)
491             addedNewNodeToOpenNodes = true
492             // If allowed make it to the focus node to preserve
                    the topically flow in the dialog
493             if (canChangeFocusNode) {
494               canChangeFocusNode = false
495               console.log('Cannot change focus node anymore
                      because a new node was made the focus node
                      without filling')
496               await loadFocusNode(name)
497               console.log('New focus node is ' + focusNode.name)
498             }
499           }
500         } else {
501           // If there are no unused informations simply add the
                  new node to the open nodes
502           console.log('New node ' + newNode.name + ' could not
                  be filled because there are no unused informations
                  ')
503           openNodes.push(name)
504           if (canChangeFocusNode) {
505             await loadFocusNode(name)
506             canChangeFocusNode = false
507             console.log('Cannot change focus node anymore
                    because a new node was made the focus node')
508             console.log('New focus node is ' + focusNode.name)
509           }
510         }
```

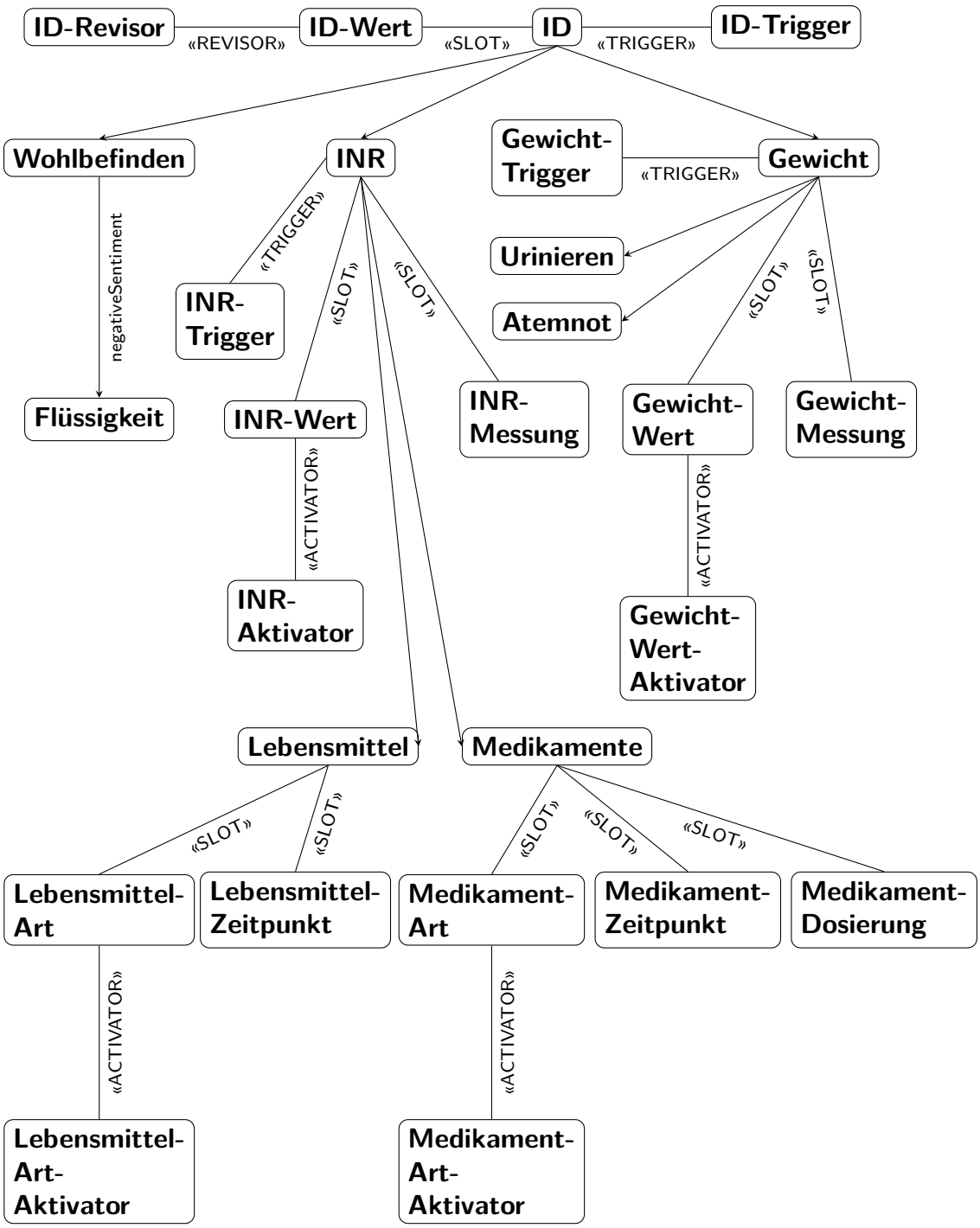
```
511           }
512         }
513       }
514
515       if (canChangeFocusNode && focusNode == null && !this.
              isDialogFinished()) {
516         // If no focus node existing load the next one of the open
              nodes for the next processing iteration
517         await loadFocusNode(openNodes[0])
518       }
519
520       return answer
521     }
522
523     // Returns the appropriate message of the focus node as an
        answer for the user
524     this.getPromptMessage = function () {
525       if (focusNode != null) {
526         let allEmpty = true
527         let onMissingQuestion = null
528         if (focusEntities) {
529           // Search for a on missing question of an empty focus
                entity
530           console.log('Try to find on missing question')
531           for (let entity of focusEntities) {
532             if (entity.submittedValue == null || ((entity.
                    submittedValue instanceof Array) && (entity.
                    submittedValue.length === 0))) {
533               if ((entity.onMissing != null) && (entity.onMissing.
                      length > 0)) {
534                 onMissingQuestion = entity.onMissing[Math.floor(Math
                        .random() * entity.onMissing.length)]
535                 break
536               } else {
537                 console.log('Focus entity ' + entity.name + ' has no
                        on missing question')
538               }
539             } else {
540               console.log('Did not ask about ' + entity.name + '
                      because it was already filled')
541               allEmpty = false
542             }
543           }
544         }
545
546         if (((!allEmpty) || startedChecking) && onMissingQuestion !=
              null) {
547           // Return an appropriate on missing question if found
548           console.log('Returning on missing question of ' +
                focusNode.name + ': ' + onMissingQuestion)
549           return onMissingQuestion
550         }
```

```
551       // If no focus entity is filled return the general prompt
              question of the focus node
552       console.log('Returning prompt message of focus node ' +
              focusNode.name)
553       return focusNode.promptMessage[Math.floor(Math.random() *
              focusNode.promptMessage.length)]
554     }
555   }
556
557   // Returns the explanation message of the first open node
558   this.getExplanationMessage = function () {
559     if (!this.isDialogFinished()) {
560       return focusNode.explanationMessage[Math.floor(Math.random()
              * focusNode.explanationMessage.length)]
561     }
562   }
563
564   // Check if the dialog is finished or if open nodes are left
565   this.isDialogFinished = () => openNodes.length === 0
566
567   // Getter for the closed nodes of this dialog
568   this.getClosedNodes = () => closedNodes
569
570   // Log debug informations about the current dialog
571   this.logInformationState = function (includeClosedNodes) {
572     console.log('Open nodes:\n\t' + openNodes.join('\n\t'))
573     if (includeClosedNodes) {
574       console.log('Closed nodes: ' + JSON.stringify(closedNodes,
              null, 4))
575     }
576     console.log('Unused slot informations: ' + JSON.stringify(
          unusedInformations, null, 4))
577   }
578
579   // Check if the current focus node should be dropped and if yes
        remove it
580   async function checkFocusNodeForDrop () {
581     let r = Math.random()
582     if (r < focusNode.dropChance) {
583       console.log('Dropped ' + focusNode.name + ': ' + r + ' < ' +
              focusNode.dropChance + ' => Drop')
584       openNodes.splice(openNodes.indexOf(focusNode.name), 1)
585       focusNode = null
586       focusEntities = null
587       return true
588     } else {
589       console.log('Did not dropp ' + focusNode.name + ': ' + r + '
              > ' + focusNode.dropChance + ' => No drop')
590       return false
591     }
592   }
593
```

```
594   // Load a focus node and its entities by name
595   async function loadFocusNode (nodeID) {
596     console.log('Load data and entities for ' + nodeID + ' in
          loadFocusNode function')
597     focusNode = await rp({
598       method: 'GET',
599       uri: 'http://data_services:8529/dialog-services/dialog_nodes
            /' + nodeID,
600       json: true
601     })
602     focusEntities = await rp({
603       method: 'GET',
604       uri: 'http://data_services:8529/dialog-services/dialog_nodes
            /' + nodeID + '/entities',
605       json: true
606     })
607     console.log('New focus node: ' + JSON.stringify(focusNode,
          null, 4))
608     console.log('New focus entities: ' + JSON.stringify(
          focusEntities, null, 4))
609   }
610
611   // Replace the current focus node with the next one from the
        open nodes
612   async function nextFocusNode () {
613     console.log('Moving to next focus node after ' + focusNode.
          name)
614     if (openNodes.length > 1) {
615       let focusNodeName = focusNode.name
616       let focusNodeIndex = openNodes.indexOf(focusNodeName)
617       console.log('Remove ' + openNodes[openNodes.indexOf(
            focusNodeName)] + ' from open nodes in nextFocusNode
            function')
618       openNodes.splice(openNodes.indexOf(focusNodeName), 1)
619       if (focusNodeIndex < openNodes.length) {
620         console.log('Loading next focus node in line')
621         await loadFocusNode(openNodes[focusNodeIndex])
622       } else {
623         console.log('Starting with the first node because there
              was no next node in line')
624         await loadFocusNode(openNodes[0])
625       }
626     } else {
627       console.log('No next focus node available')
628       openNodes.splice(0, 1)
629     }
630   }
631 }
632
633 module.exports = InformationManager
```

B Ontologie und Dialogmodell für die Evaluation

Zuerst wird das Medolution-Dialogmodell erläutert. Am Anfang des Gesprächs wird der Benutzer durch das einzige aktive Ziel nach seiner Benutzer-ID befragt und mit einem Revisor wird überprüft, ob zu dieser ID ein Eintrag in der Datenbank existiert. Dadurch ist die ID von Anfang an bekannt und ab diesem Zeitpunkt ist es möglich, Revisoren und Aktivatoren Bedingungen kontrollieren zu lassen, welche sich auf Daten früherer Dialoge mit dem Benutzer beziehen.
Der Benutzer wird allgemein nach seinem Wohlbefinden befragt. Falls er dieses als schlecht beschreibt, ist eine wahrscheinliche Ursache, dass er nicht genug Flüssigkeit zu sich nimmt und deshalb wird ihm in dem Fall zusätzlich eine Frage danach gestellt. Diese Frage hat zwar keinen Einfluss auf den Verlauf des Gespräches, ist aber eine wichtige Information für den Arzt und ebenfalls eine passende Erinnerung für den Patienten auf seine Flüssigkeitszufuhr zu achten.
Zentral für den Dialog ist der INR-Wert, also eine Kennzahl für die Blutgerinnung, da diese bei Kunstherzen bei zu hohen beziehungsweise niedrigen Werten schnell gefährlich werden kann. Hierbei wird der INR-Wert an sich und der Zeitpunkt, zu dem die Messung erfolgt ist, erfragt. Wenn also der Benutzer einen INR-Wert von unter 2 oder über $3,5$ angibt, stellt dies ein erhöhtes Risiko dar und es wird nach möglichen Ursachen gesucht. Hierfür kommen primär Vitamin K-haltige Lebensmittel und einige Medikamente in Frage. Bei den Lebensmitteln ist auch der Zeitpunkt der Mahlzeit eine gesuchte Information und bei den Medikamenten ebenfalls der Zeitpunkt der Einnahme sowie die Größe der eingenommen Dosierung. Nach beiden wird gefragt und bei einer in der Ontologie modellierter Beeinflussung des INR-Wertes durch das Medikament oder das Lebensmittel, wird eine Warnung gesendet.
Abschließend ist es auch wichtig auf Gewichtsschwankungen zu achten. Da der Chatbot sich regelmäßig im Abstand von wenigen Tagen mit den Patienten unterhalten wird, gilt als Faustregel, dass $\pm$ 2 Kilogramm innerhalb so kurzer Zeit für Patienten mit einem Kunstherz ein Symptom sind, welches weiter untersucht werden sollte. Auch hier wird nach dem Wert und der Zeitpunkt der Messung gefragt. Falls eine verdächtige Gewichtsschwankung auftritt, wird empfohlen einen Arzt zu kontaktieren und zusätzlich nach den Symptomen Atemnot und Probleme beim Urinieren gefragt.
Auf die Metadaten innerhalb der Elemente des Modells wurde zur Verbesserung der Übersicht hier in diesem Beispiel verzichtet:


ID-Revisor
«REVISOR»
ID-Wert
«SLOT»
ID
«TRIGGER»
ID-Trigger
Wohlbefinden
INR
Gewicht-Trigger
«TRIGGER»
Gewicht
negativeSentiment
«TRIGGER»
Urinieren
Atemnot
«SLOT»
«SLOT»
«SLOT»
«SLOT»
INR-Trigger
Flüssigkeit
INR-Wert
INR-Messung
Gewicht-Wert
Gewicht-Messung
«ACTIVATOR»
«ACTIVATOR»
INR-Aktivator
Gewicht-Wert-Aktivator
Lebensmittel
Medikamente
«SLOT»
«SLOT»
«SLOT»
«SLOT»
«SLOT»
Lebensmittel-Art
Lebensmittel-Zeitpunkt
Medikament-Art
Medikament-Zeitpunkt
Medikament-Dosierung
«ACTIVATOR»
«ACTIVATOR»
Lebensmittel-Art-Aktivator
Medikament-Art-Aktivator

Weiterhin sind hier die verwendete Ontologie und der Regelsatz für die Schlussfolgerungen zu sehen, welche benutzt werden, um abfragen zu können, ob Medikamente oder Lebensmittel einen Einfluss auf den INR-Wert haben, damit entsprechende Warnungen angezeigt werden können. Ontologien, welche für dieses Chatbot-System verwendet werden können, sind in realen Einsatz-Domänen signifikant größer, jedoch genügt diese deutlich kleinere Ontologie zur Demonstration der Funktionalität und zur Evaluation.
Zuerst die Schlussfolgerungs-Regeln:

```
@prefix md:   <http://www.c-lab.de/projekte/Medolution#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

[ruleIstEin: (?s md:istEin ?o) (?o md:istEin ?r) -> (?s md:istEin ?r)]
[ruleBeeinflusst: (?s md:beeinflusst ?o) (?o md:beeinflusst ?r)
    -> (?s md:beeinflusst ?r)]
[ruleEnthält: (?s md:enthält ?o) (?o md:enthält ?r) -> (?s md:enthält ?r)]
[ruleEnthaltenIn: (?s md:enthält ?o) -> (?o md:enthaltenIn ?s)]
[ruleBeeinflusstDurch: (?s md:beeinflusst ?o) -> (?o md:beeinflusstDurch ?s)]
[ruleBeeinflusstChain: (?s md:enthält ?o) (?o md:beeinflusst ?r)
    -> (?s md:beeinflusst ?r)]
[ruleIstBeeinflussung: (?s md:istEin ?o) (?o md:beeinflusst ?r)
    -> (?s md:beeinflusst ?r)]
```

Anschließend die Beispiel-Ontologie zur Einstufung der Beeinflussung des INR-Wertes durch Lebensmittel mit einem hohen Vitamin K Gehalt oder Medikamente, welche Gerinnungshemmer sind, beziehungsweise die Wirkungsweise von Gerinnungshemmern beeinflussen:

```
@prefix md:   <http://www.c-lab.de/projekte/Medolution#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

# Lebensmittel
md:vitamink       md:beeinflusst  md:inr .
md:broccoli       md:enthält      md:vitamink .
md:grünkohl       md:enthält      md:vitamink .
md:sushi          md:enthält      md:algen .
md:algen          md:enthält      md:vitamink .
md:blattgemüse    md:enthält      md:vitamink .
md:blumenkohl     md:enthält      md:vitamink .
```

```
md:spinat          md:istEin       md:blattgemüse .
md:rosenkohl       md:enthält      md:vitamink .
md:chicoree        md:istEin       md:blattgemüse .
md:chinakohl       md:enthält      md:vitamink .
md:eisbergsalat    md:istEin       md:blattgemüse .
md:endivie         md:istEin       md:blattgemüse .
md:feldsalat       md:istEin       md:blattgemüse .
md:fenchel         md:enthält      md:vitamink .
md:zwiebel         md:enthält      md:vitamink .
md:knoblauch       md:enthält      md:vitamink .
md:kopfsalat       md:istEin       md:blattgemüse .
md:kresse          md:enthält      md:vitamink .
md:linsen          md:enthält      md:vitamink .
md:löwenzahn       md:istEin       md:blattgemüse .
md:mangold         md:istEin       md:blattgemüse .
md:poree           md:enthält      md:vitamink .
md:rotkohl         md:enthält      md:vitamink .
md:sauerampfer     md:enthält      md:vitamink .
md:schnittlauch    md:enthält      md:vitamink .
md:schnittsalat    md:istEin       md:blattgemüse .
md:wirsing         md:enthält      md:vitamink .
md:zwiebeln        md:enthält      md:vitamink .
md:petersilie      md:enthält      md:vitamink .


# Medikamente
md:marcumar        md:beeinflusst  md:inr .
md:falithrom       md:beeinflusst  md:inr .
md:aspirin         md:beeinflusst  md:inr .
md:ibuprofen       md:beeinflusst  md:inr .
md:carbamazepin    md:beeinflusst  md:inr .
md:rifampicin      md:beeinflusst  md:inr .
md:allopurinol     md:beeinflusst  md:inr .
md:metformin       md:beeinflusst  md:inr .
md:amiodaron       md:beeinflusst  md:inr .
md:propafenon      md:beeinflusst  md:inr .
md:furosemid       md:beeinflusst  md:inr .
md:antibabypille   md:beeinflusst  md:inr .
md:amitriptylin    md:beeinflusst  md:inr .
```

C REST-Schnittstelle für Anfragen an das Dialogmodell

Im Folgenden ist die REST-Schnittstelle der FOXX-Microservices zu sehen, welche benutzt wird, um Anfragen zu Benutzern des Chatbots oder zu dem hinterlegten Dialogmodell zu stellen. Die Dokumentation aller möglichen REST-Anfragen ist automatisch durch *Swagger*[1] erfolgt, was ein Satz an Open-Source Tools für den offenen Spezifikations-Standard *OpenAPI Specification*[2] ist.

[1] `https://swagger.io/`
[2] `https://github.com/OAI/OpenAPI-Specification`

Paths

Checks if one of the currently open nodes was triggered by the user input entities

```
PUT /dialog_nodes/triggered
```

Parameters

Type	Name	Description	Schema
Body	**body** *optional*	Names of the currently open nodes and the entities of the user input	object

Responses

HTTP Code	Description	Schema
200	Informations if one of the open nodes was triggered	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Checks if the submitted values are correct with the revisors of the entity

```
PUT /dialog_nodes/{entity}/check_revisor
```

Parameters

Type	Name	Description	Schema
Path	**entity** *required*	Name of the entity of the dialog node	string
Body	**body** *optional*	The submitted value for the revisors entitiy and the User ID if the revisor needs to request past dialog records from this user	object

Responses

HTTP Code	Description	Schema
200	Status informations from the revisor	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Returns the dialog node document with the given name

```
GET /dialog_nodes/{name}
```

Parameters

Type	Name	Description	Schema
Path	**name** *required*	Name of the requested dialog node	string

Responses

HTTP Code	Description	Schema
200	Dialog node document	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Produces

- `application/json`

Returns the dialog node names that are activated by the node with the given name and the filled slots of the node after checking all activators of connected entities and edges with a intent constraint

```
PUT /dialog_nodes/{name}/activates
```

Parameters

Type	Name	Description	Schema
Path	**name** *required*	Name of the requested dialog node	string
Body	**body** *optional*	Filled entities of the node, intent of the message used for final filling and the User ID if a activator needs to request past dialog records from this user	object

Responses

HTTP Code	Description	Schema
200	New activated dialog nodes	string

HTTP Code	Description	Schema
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Returns the connected entities of the dialog node with the given name

```
GET /dialog_nodes/{name}/entities
```

Parameters

Type	Name	Description	Schema
Path	**name** *required*	Name of the requested dialog node	string

Responses

HTTP Code	Description	Schema
200	Entities documents	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Produces

- `application/json`

Tries to fill the dialog node entities with the unused slot informations

```
PUT /dialog_nodes/{name}/fill
```

Parameters

Type	Name	Description	Schema
Path	**name** *required*	Name of the dialog node	string
Body	**body** *optional*	Unused slot informations destined for this filling try	object

Responses

HTTP Code	Description	Schema
200	Status informations about the filling try	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Returns the triggers of connected entities of the dialog node with the given name

```
GET /dialog_nodes/{name}/triggers
```

Parameters

Type	Name	Description	Schema
Path	**name** *required*	Name of the requested dialog node	string

Responses

HTTP Code	Description	Schema
200	Triggers documents	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Produces

- `application/json`

Checks if a user ID is existing for a user ID revisor

```
PUT /users/check_id
```

Parameters

Type	Name	Description	Schema
Body	**body** *optional*	User ID	object

Responses

HTTP Code	Description	Schema
200	Check results with either a personalized greeting or an error message	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Returns the user document with the given ID

```
GET /users/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the requested user	number

Responses

HTTP Code	Description	Schema
200	User object	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Produces

- `application/json`

Adds a new dialog record for the user with the given ID

```
POST /users/{id}/record
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the user	number
Body	**body** *optional*	New record for the user containing the filled nodes, a completed flag and the date of the dialog	object

Responses

HTTP Code	Description	Schema
200	Updated user object	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer
errorMessage *optional*	string
errorNum *optional*	integer

Consumes

- `application/json`

Produces

- `application/json`

Returns the latest entry for the requested value of the records from the user with the given ID if available

```
GET /users/{id}/{node}/{entity}
```

Parameters

Type	Name	Description	Schema
Path	**entity** *required*	Entity name of the requested value	string
Path	**id** *required*	ID of the user of the requested value	number
Path	**node** *required*	Node name of the requested value	string

Responses

HTTP Code	Description	Schema
200	Latest entry for the requested value	string
500	Default error response.	Response 500

Response 500

Name	Schema
code *optional*	integer

Name	Schema
errorMessage *optional*	string
errorNum *optional*	integer

Produces

- `application/json`