Großer Beleg

über das Thema

Entwicklung eines intelligenten Dialogsystems für iSuite

an der

Technischen Universität Dresden

Fakultät Informatik
Institut für Künstliche Intelligenz
Lehrstuhl Applied Knowledge Representation and Reasoning

bearbeitet von

Johannes Lötzsch

geboren am 05.08.1986 in Dresden

Matrikel-Nr.:	3244064
Hochschullehrer:	Sen.-Prof. Dr.habil. Uwe Petersohn
Betreuer:	René Balzer

Eingereicht am:

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich der von mir dem Institut für Künstliche Intelligenz der Fakultit Informatik eingereichte Große Beleg zum Thema „Entwicklung eines intelligenten Dialogsystems für iSuite“ selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Dresden, 09. Mai 2019

Zusammenfassung

Digitale Dialogsysteme ermöglichen es im Gegensatz zu Papierfragebögen, interaktiv zu entscheiden welche Detailfragen unter Beachtung des bereits vorhandenen Wissens sinnvoll sind. Diese Arbeit entwickelt solch ein intelligentes Dialogsystem und überprüft die Anwendbarkeit als Ersatz für bestehende medizinische Fragebögen.

Die wichtigste Architekturentscheidung besteht in der Auswahl eines geeigneten Kalküls zur Repräsentation der zwischen den einzelnen Fragen bestehenden Abhängigkeiten. Gesucht wird die schwächstmögliche noch hinreichend ausdrucksfähige Logik. Die Verwendung einer zu allgemeinen Sprache würde dazu führen, dass für die notwendigen Berechnungen nicht sichergestellt werden kann, dass Lösungen existieren und effizient bestimmbar sind.

Die Arbeit leitet her, warum sich eine Einschränkung auf monadische prädikatenlogische Hornformeln anbietet und stellt eine Implementierung zur Verfügung, die Inferenz mit einer Komplexität $O(n)$ garantiert.

Abstract

In comparison to paper questionnaires, digital dialog systems allow interactive decissions. Questions about details should only be queried in case of being consistent with already obtained knowledge. This thesis develops an intelligent dialog system and evaluates its practicality as a substitution for medical questionnaires.

Selecting a calculus for representing the dependencies between different questions is the design decission. From all logics providing the required expressiveness, the weakest one should be used. Such choice is essential to guarantee efficient determination for all required calculations.

We derive, why monadic first-order Horn clauses are recommendable. A proof of concept implementation is capable of providing inference with a complexity of $O(n)$.

Inhaltsverzeichnis

1 Einleitung

Das Programm *iSuite* [8] ist ein Informations- und Unterstützungssystem zur Behandlung von Schmerzpatienten. Es verarbeitet sowohl Daten die von Ärzten und Arzthelfern erfasst werden, als auch Aussagen die direkt vom Patienten über elektronische Fragebögen abgefragt werden. Das Patienten-Dialogsystem ermöglicht regelmäßige Erhebung von zur Diagnose maßgeblichen Informationen und hat das Ziel den dabei benötigten zeitlichen Personalaufwand zu minimieren. Die erfassten Daten werden zur Qualitätssicherung, Diagnostik-, Befundungs- und Therapieeffektivierung verwendet [3]. Kern von iSuite ist ein Expertensystem, welches medizinisches Fachwissen in Form von expliziten deklarativen Regeln benutzt, um mittels konzeptbasierter Inferenz [9] den Arzt bei der Beurteilung von Fällen zu unterstützen. Weiterhin ist es möglich auf Basis des gesammelten Datenbestände über möglichst viele ähnliche Fälle probabilistisch Schlüsse zu ziehen und dem Arzt zu assistieren, indem assoziiertes Wissen angeboten wird. *iSuite*

Die verwendete Eingabeschnittstelle zur Anamneseerhebung basiert auf standartisierten Fragebögen. Bisher ähnelt das Ausfüllen dem von Papierfragebögen und nutzt die Möglichkeiten digitaler Dialogsysteme nicht aus. Neben der Verwendung einer modernen, dem jeweiligen Nutzers angepassten, barrierefreien Nutzerschnittstelle, bietet vor allem die Bereitstellung einer intelligenten Benutzerführung großes Potential. Im Gegensatz zu herkömmlichen Fragebögen, soll das aus vorherigen Abfragen erhobene Wissen verwendet werden, um dynamisch zu entscheiden welche weitergehenden Fragen gestellt werden müssen. Dabei sollen Fragen ausgelassen werden, wenn aufgrund bereits existierendem Wissens auf deren Antwort geschlossen werden kann. Das System soll dafür sorgen, dass vor dem Erfragen von Details stets alle allgemeineren Fragen gestellt wurden.

Die Dialogführung soll nicht auf expliziten Fallunterscheidungen basieren, die das domainspezifische Expertenwissen in imperative Anweisungen übersetzen. Ein solcher Ansatz währe sehr inflexibel. Bereits kleine Änderungen der Fragebögen würden erfordern, dass Spezialisten alle Auswirkungen auf mögliche Abläufe der Dialoge prüfen und anpassen. Statt explizite imperative Fallunterscheidungen, wird mit dieser Arbeit eine möglichst allgemeine Lösung konstruiert, die nötiges Wissen implizit aus deklarativen Regeln ableitet. Das entwickelte Knowledge Retrieval für Dialogsysteme ist flexibel und kann unverändert weiterverwendet werden, auch wenn

neue Fragen und zwischen ihnen bestehende Abhängigkeiten hinzugefügt werden.

Für eine nutzerfreundliche Lösung ist es essentiell, dass keine zu großen Wartezeiten zwischen dem Senden von Antworten und dem Anzeigen der darauffolgenden Fragen entstehen. Um diese Anforderung erfüllen zu können, besteht der Fokus dieser Arbeit in der Auswahl einer geeigneten Logik zur Repräsentation der zwischen verschiedenen Fragen und ihren möglichen Antworten bestehenden Abhängigkeiten. Das verwendete Kalkül muss einerseits eine hinreichend starke Ausdrucksfähigkeit haben um alle notwendigen Abhängigkeiten darstellen zu können. Andererseits soll vermieden werden, dass eine zu allgemeine Logik erlaubt wird, da sonst keine hinreichenden Garantien bezüglich der für die benötigten Berechnungen zu erwartenden Zeit- und Platzkomplexitäten gegeben werden können.

2 Theoretische Grundlagen

Der Theorieteil vergleicht zunächst verschieden ausdrücksstarke Logiken und ihre Eigenschaften bezüglich Entscheidbarkeit. Als erstes findet auf Anregung des Lehrstuhl eine Auseinandersetzung mit F-Logik statt. Im Anschluss folgt eine Darlegung, warum sich in der Implementierung für eine Einschränkung auf monadische prädikatenlogische Hornformeln entschieden wurde.

Abschließend wird besprochen welche Arten von Abhängigkeiten zwischen Fragebögen gefunden wurden und welche weiteren Anforderungen an die zu verwendende Wissensdatenbank existieren.

2.1 Repräsentation mit Frames und F-Logik

2.1.1 Frames

Als Frames wird eine von Marvin Minsky [5] vorgestellte Klasse von Datenstrukturen bezeichnet, welche in eine Vererbungshierarchie eingebunden sind. Sie sind sehr ähnlich zu den Abstraktionen Klassen bzw. Prototypen in der objektorientierten Programmierung (*OOP*). Analog zu Attributen der OOP können Frames sogenannte Slots besitzen, welche vererbbare Merkmale beschreiben. Jeder Slot kann aus mehreren Facetten bestehen. Neben Name und Wert können dies beispielsweise Standardwerte oder Bedingungen für den Wertebereich sein. Als Wert eines Slots können neben primitiven Datentypen (z.B. Ganzzahl, Zeichenkette) auch Relationen zu anderen Frames verwendet werden. Typische Relationen sind subsets (Nachfolger in der Hierarchie), instance-of (Instanzbeziehung) und part-of (Aggregation) [12].

OOP

Minsky geht davon aus, dass Schlussfolgerungen auf Basis von Vererbung der menschlichen Wisensverarbeitung nahe kommt [5]. Sie gilt als leicht verständlich und anwendbar. Wie aus der OOP bekannt und in [4] beschrieben, ist die Darstellung von Konzepten durch abstrakte, vererbbare Datenstrukturen außerdem sehr ausdrucksstark.

Frames können als objektzentrierte Repräsentation semantischer Netze gesehen werden, wobei auf andere Frames verweisende Frame-Slots den Kanten der semantischen Netze entspricht [12].

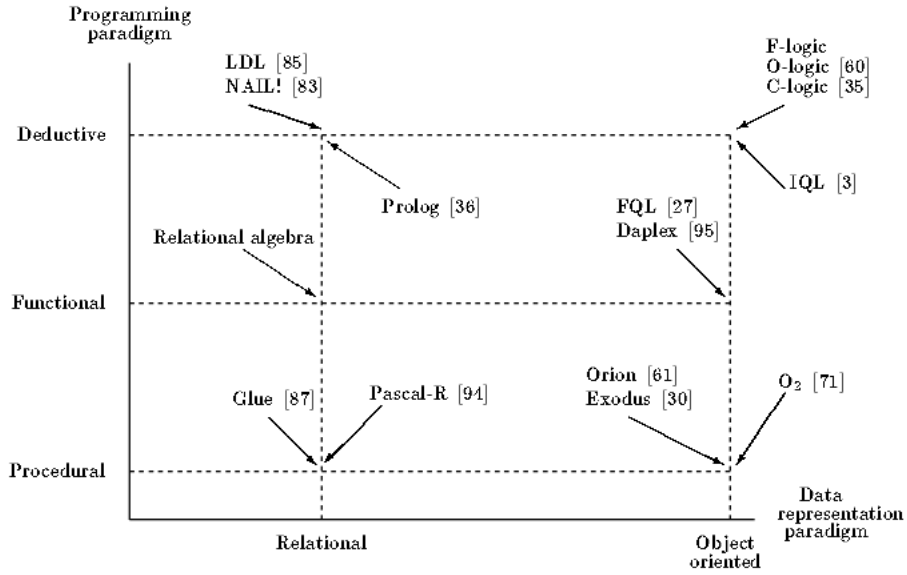

Abb. 2.1: Klassifikation von F-Logik aus [4]

2.1.2 F-Logik

F-Logik [4] steht für „Frame Logic“ und ist eine Sprache um framebasierte Daten, sowie Relationen zwischen ihnen formal darzustellen und damit deduktives Schließen auf der Basis von Frames zu ermöglichen. Als auf Logik und Frames basierende Sprache versucht sie, die Paradigmen von deduktiven und objektorientierten Systemen zu vereinen.

Der freie Ontologie-Editor Protégé [6] unterstützt F-Logik. Mit FLORA2 [13] steht eine freie und umfangreich dokumentierte Implementierung in XSB-Prolog [1] zur Vefügung.

Im Vergleich zu relationalen Sprachen wie Prolog kann F-Logik als Erweiterung durch Objekte verstanden werden. Dies kann einerseits als syntaktischer Zucker abgetan werden, dient jedoch andererseits deutlich der Lesbarkeit und vereinfacht die Modellierung, da die objektrelationale Abbildung statt vom Anwender automatisiert vorgenommen werden kann.

2.2 Repräsentation in Hornlogik

Horn-Klauseln sind Disjunktionen von maximal einem positiven mit beliebig vielen negativen Literalen. Dabei wird das positive Literal auch als Kopf und die Disjunktion der negativen Literale als Rumpf bezeichnet.

Jede Hornklausel kann auch als Implikation dargestellt werden. Dabei ergeben die Variablen aus dem Rumpf in positiver Form konjugiert die Prämisse und der Kopf die Konklusion.

$$\underbrace{h}_{Kopf} \quad \vee \quad \underbrace{\neg r_1 \vee \neg r_2 \vee \ldots \vee \neg r_n}_{Rumpf}$$

$$\equiv \quad h \quad \vee \quad \neg(r_1 \wedge r_2 \wedge \ldots \wedge r_n)$$

$$\equiv \quad \underbrace{h}_{Konklusion} \quad \leftarrow \quad \underbrace{(r_1 \wedge r_2 \wedge \ldots \wedge r_n)}_{Prämisse}$$

Horn-Klauseln können in drei verschiedenen Formen auftreten:

- Fakten h sind Hornklauseln, die aus genau einem positivem Literal (ohne negative Literale) bestehen. Damit diese Klauseln auf Wahr abgebildet werden, muss das eine Literal aus dem sie besteht Wahr sein.
- definite Hornklauseln $h \leftarrow (r_1 \wedge r_2 \wedge \ldots \wedge r_n)$ sind Klauseln mit genau einem positiven und mindestens einem negativen Literal. Ihre Semantik ist die einer herkömmlichen Implikation.
- Zielklauseln $\neg r_1 \vee \neg r_2 \vee \ldots \vee \neg r_n$ enthalten nur negative Literale. Sie können in einer Konjunktion mit weiteren Klauseln verwendet werden, um anzufragen ob (und welche) Modelle für diese Konjunktion existieren.

Unter Horn-Formeln werden Konjunktion von Horn-Klauseln verstanden. Sie sind folglich spezielle Konjunktionen von Disjunktionen von Literalen. Disjunktionen mit mehr als einem positivem Literal (bzw. Implikationen, die in der Konklusion eine Disjunktion von mindestens zwei positiven Literalen enthalten) sind nicht als Horn-Formeln darstellbar. Folglich handelt es sich bei Horn-Formeln um eine echte Untermenge allgemeiner logischer Formeln. Horn-Formeln können für verschiedene Klassen von Logik definiert werden, wir werden zunächst nur Horn-Formeln in der Aussagenlogik betrachten.

Das Entscheidungsproblem, ob für eine gegebene aussagenlogische Horn-Formel (*AHF*) eine gültige Lösung existiert, wird als *HORNSAT* bezeichnet. Es besitzt die außerst positive Eigenschaft, dass ihre Erfüllbarkeit in linearer Zeit (in Abhängigkeit zur Länge der Formel) entscheidbar ist. Insbesondere existieren konstruktive Algorithmen mit $O(n)$, welche im Falle der Erfüllbarkeit ein mögliches Modell berechnen. Der Markierungsalgorithmus ist ein solcher, dessen Grundidee dem im folgenden Abschnitt erklärtem Forward Chaining für AHF (2.2.1) ähnelt.

AHF
HORNSAT

Die Möglichkeit effizient Modelle zu finden, ist häufig ein großer Vorteil von AHF gegenüber allgemeinen aussagenlogischen Formeln, bei denen das *SAT-Problem* NP-vollständig ist. Wenn die Ausdrucksfähigkeit von AHF zum beschreiben eines Problems ausreicht, lohnt es sich daher diese Repräsentation zu verwenden.

SAT-Problem

2.2.1 Forward Chaining für aussagenlogische Horn-Formeln

Der Markierungsalgorithmus zum entscheiden der Erfüllbarkeit AHF, sowie das Forward Chaining zum berechnen eines Modells einer AHF basieren auf der gleichen Idee: Dem schrittweisen Generieren einer minimalen Interpretation für die gegebene Hornformel.

Im ersten Schritt beider Algorithmen, werden alle Variablen, die als Fakt in der Hornformel vorkommen, auf Wahr abgebildet. Dem liegt zu Grunde, dass keine Interpretation existieren kann, bei der einer der Fakten auf Falsch abgebildet wird. Im folgenden wird geprüft, ob sich aus den definiten Hornklauseln und der bisherigen Variablenzuordnung neue Fakten schliessen lassen. Üblicherweise wird hierfür die Inferenzregel Modus ponendo ponens angewendet. Dieser besagt, dass aus einem Fakt p und einer Implikation $q \leftarrow p$ auch ein Fakt q folgen muss.

$$\frac{p, q \leftarrow p}{q}$$

Eine effiziente Implementierung dieser Regel ist möglich, indem jede bekannte Unit aus allen Klauseln gestrichen werden. Die so erhaltenen neuen Klauseln sind in allen möglichen Modellen semantisch äquivalent zu den bisherigen Klauseln, da hiermit lediglich die Negationen wahrer Aussagen aus Disjunktionen entfernt werden. Wenn in Folge dessen eine Klausel nur noch den leeren Rumpf enthält (h), kann ihr Kopf als neuer Fakt betrachtet werden. In diesem Fall ist die Abbildung von h auf Wahr zur Interpretation hinzuzufügen und es kann geprüft werden, ob sich aus diesem neuen Fakt in Kombination mit den vorhandenen definiten Hornklauseln weitere Fakten ableiten lassen.

Durch transitives Anwenden der Inferenzregeln werden schrittweise alle zwangsläufig auf Wahr abzubildenden Variablen gefunden. Falls bei diesem Vorgehen alle Variablen aus einer Zielklausel entfernt wurden, entsteht eine leere Klausel. Da leere Disjunktionen immer Falsch sind, kann keine Konjunktion mit einer solchen Wahr sein. Hornformeln, aus denen sich eine leere Klausel ableiten lässt, sind folglich unerfüllbar. Der Algorithmus kann mit diesem Ergebnis beendet werden. Nach maximal $O(n)$ Schritten, wobei n die Länge der Hornformel ist, wird ein Fixpunkt erreicht, an dem auf keine neuen Fakten geschlossen werden kann. Bezüglich der Anwendung der Inferenzregeln handelt es sich um die Transitive Hülle. Der Algorithmus hat zu diesem Zeitpunkt alle zwangsläufig auf Wahr abzubildenden Variablen berechnet. Wenn alle anderen Variablen auf Falsch abgebildet werden, handelt es sich um ein minimales Modell. Der Algorithmus terminiert in diesem Fall mit der Ausgabe des Modells beziehungsweise der Antwort „Erfüllbar“.

Forward Chaining kann als datengetriebene Breitensuche nach einem Modell verstanden werden.

2.2.2 Backward Chaining für aussagenlogische Horn-Formeln

Bei großen Datenbasen kann Forward Chaining sehr ineffizient sein: Während der Berechnung eines Modells für eine gegebene Zielklausel, werden meist auch für viele weitere Variablen, die mit der Zielklausel nicht in direktem Zusammenhang stehen, Belegungen bestimmt. Backward Chaining versucht diesen Nachteil zu vermeiden.

Die Idee hinter Backward Chaining ist es per Tiefensuche, ausgehend von einer Anfrage (Zielklausel) Fakten zu finden, aus denen das Ziel folgt. Diesen Vorgehen wird als zielgetrieben bezeichnet.

Zentrale Datenstruktur für diesen Algorithmus ist die sogenannte Zielliste, welche Fakten enthält, die bewiesen werden müssen, um die Zielklausel zu beweisen. Initial enthält die Zielliste die Variablen der Anfrage.

In jedem Schritt wird eine Klausel, welche eine der Variablen aus der Zielliste als Konklusion enthält, verwendet um eine neue Zielliste zu generieren, welche als neue Anfrage für einen rekursiven Aufruf der Suchfunktion verwendet wird. Die neue Zielliste wird erzeugt, indem die gewählte Variable gestrichen und statt dessen durch die Prämisse ersetzt wird. Da die Prämisse eine hinreichende Bedingung für die gestrichene Variable darstellt, ist ein Beweis der neuen Anfrage immer auch ein Beweis der vorherigen und damit auch der ursprünglichen Anfrage. Für den Spezialfall, dass es sich bei der verwendeten Klausel um einen Fakt (Prämisse ist Wahr) handelt, darf die Konklusion ersatzlos aus der Zielliste gestrichen werden. Auf diese Weise kann eine leere Zielliste erreicht werden, dies ist gleichbedeutend mit dem Beweis der Anfrage.

Falls für eine Zielliste keine anwendbare Klausel gefunden werden kann, gilt sie als nicht beweisbar. In dem Fall wird Backtracking angewendet, um von der vorhergehenden Anfrage ausgehend, unter Wahl einer anderen Klausel, einen anderen Beweispfad zu suchen. Schlagen alle möglichen Beweisversuche fehl, gilt die Anfrage ebenfalls als unbeweisbar.

Backward Chaining kann in vielen Fällen mit weniger Schritten als Forward Chaining zu einem Ergebnis kommen. Ein weiterer Vorteil ist die Möglichkeit, eine modifizierte Variante zu implementieren, bei der nach fehlenden Fakten nachgefragt wird.

Bei Backward Chaining müssen allerdings auch die üblichen Probleme von Tiefensuchen beachtet werden: Die Existenz zueinander symmetrischer Klauseln (die Aquivalenz $p \leftarrow q$ und $q \rightarrow p$) könnte dazu führen, dass in einer Endlosschleife versucht wird eine Anfrage aus jeweils der anderen herzuleiten. Solche Schleifen müssen als Zirkelschluss erkannt und durch Backtracking beantwortet werden. Ein weiteres Problem ist die mögliche Ineffizienz durch Mehrfachberechnung gleicher

Teilbäume. Diese kann mit Hilfe dynamischer Programmierung umgangen werden.

2.2.3 prädikatenlogische Hornformeln

Für prädikatenlogische Hornformeln erster Stufe (*PHF*) muss beim Backward Chaining ein weiterer Fall beachtet werden: Der Suchraum für PHF ist abzählbar unendlich. Daher kann es passieren, dass die Tiefensuche in einen unendlich tiefen Ast des Suchbaumes absteigt (infiniter Regress). Dabei kann es passieren, dass keine Lösung gefunden wird, obwohl in einem anderen Zweig eine existiert. Dies kann durch iterative Tiefensuche vermieden werden, diese findet Lösungen falls existent. Es ist allerdings unbekannt, wie lange für eine Lösung gesucht werden muss. Falls keine gefunden wird, kann nicht entschieden werden ob eine Lösung an tieferer Stelle im Suchbaum existiert. Das Erfüllbarkeitsproblem und das Allgemeingültigkeitsproblem von PHF sind somit nur semi-entscheidbar. *PHF*

Dafür sind PHF sehr mächtig. Sie sind fähig SAT-Probleme der Aussagenlogik auszudrücken und damit NP-vollständig [7]. Das wahrscheinlich bekannteste Einsatzfeld für PHF stellt die Logik-Programmierung mit Sprachen wie Prolog dar.

2.3 Repräsentation in monadischer Prädikatenlogik

Monadische Prädikatenlogik bezeichnet die Teilmenge der Prädikatenlogik, bei der nur einstellige Relationssymbole und nur primitive Terme erlaubt sind. Primitive Terme enthalten keine Funktionssymbole, bestehen also lediglich aus einer Konstante oder einer Variable. Da weder mehrstellige Relationen, noch zusammengesetzte Terme verwendet werden dürfen, ist es nicht möglich die Peano-Axiome darzustellen. Dies ist wichtige Einschränkung in der Ausdrucksfähigkeit. In Kapitel 2.3.3 wird gezeigt, wie dieses Defizit für viele Praxisprobleme kompensiert werden kann.

2.3.1 Abbildung auf Aussagenlogik

Alle durch monadische Prädikatenlogik darstellbaren geschlossenen Formeln lassen sich ebenfalls durch aussagenlogische Formeln repräsentieren. Dafür müssen im ersten Schritt alle Quantoren und Variablen entfernt und nachfolgend alle atomaren Formeln durch aussagenlogische Atome ersetzt werden. Das Eliminieren eines Allquantors mit der durch sie gebundenen Variable x kann erfolgen, indem die Teilformel im Bereich des Quantors $F(x)$ durch eine Konjunktion ersetzt wird, die alle möglichen Substitutionen der Variable durch Elemente der Domain D aus der

ursprünglichen Teilformel verknüpft.

$$\frac{\forall x F(x)}{F(c_1) \wedge \ldots \wedge F(c_n) | c_i \in D}$$

Analog können Existenzquantoren durch Disjunktionen ersetzt werden.

$$\frac{\exists x F(x)}{F(c_1) \vee \ldots \vee F(c_n) | c_i \in D}$$

Im Falle von Herbrand-Interpretationen für monadische Prädikatenlogik handelt es sich bei der Domain um die Menge aller in den betrachteten Formeln auftretenden Konstantensymbolen.

Nach der Entfernung der Quantoren, können die erhaltenen variablenfreien Formeln als aussagenlogische Formeln dargestellt werden, indem alle atomaren Formeln (einstellige Prädikate angewendet auf Konstanten) durch jeweils eine aussagenlogische atomare Formel (entspricht einem nullstelligen Prädikat) ersetzt werden. Diese Art der Ersetzung ist ein modellerhaltender Isomorphismus.

Die hier beschriebene Transformation hat den Vorteil sehr einfach zu sein. Als Nachteil ergibt sich allerdings, dass die Länge von Formeln für jeden eliminierten Quantor um die Kardinalität der Konstantensymbole als Faktor wächst. Bei verschachtelten Quantoren führt dies zu exponentiell wachsenden Formellängen. Durch geeignete alternative Transformationen, wie der Nutzung von Skolemisierung, kann dies umgangen werden.

2.3.2 Monadische prädikatenlogische Hornformeln

Der Isomorphismus zwischen monadischer Prädikatenlogik und Aussagenlogik gilt auch, wenn sich jeweils auf Hornformeln beschränkt wird. Mit Hilfe der im vorherigen Abschnitt gezeigten Umformung können monadische prädikatenlogische Hornformeln (*MHF*) effizient in AHF übersetzt werden, um die verhältnismässig bekannten und effizienten Algorithmen für diese Klasse der Logik auf sie anzuwenden. Aus dem Isomorphismus ergibt sich, dass die positiven Eigenschaften bezüglich Entscheidbarkeit von AHF gleichermaßen für MHF gelten. Das entscheiden der Erfüllbarkeit und wenn vorhanden das finden eines Modells sind jeweils deterministisch in Polynomialzeit lösbar.

MHF

MHF kann als eine andere Schreibweise für AHF verstanden werden. Das erlauben einstelliger Relationssymbole auf primitiven Termen kann die Lesbarkeit verbessern ohne dass Inferenz auf dieser Darstellung in einer anderen Komplexitätsklasse wäre.

2.3.3 Erweiterung durch Procedural Attachments

Wie im folgenden Kapitel gezeigt wird, lassen sich die in 2.4.1 beschriebenen Regeln durch monadische Prädikatenlogik repräsentieren. Unter der Annahme, dass für die Fragebögen keine komplexeren Abhängigkeiten benötigt werden, eignet sie sich für die Darstellung hervorragend. Sie hat aufgrund ihrer Einfachheit viele wünschenswerte Eigenschaften (siehe 2.3.1). Für uns besonders wichtig ist die sehr geringe für das Schließen benötigte Komplexität. Weiterhin ist die einfache Syntax hilfreich beim aufstellen der Regeln für einen Fragebogen.

Das Defizit der schwachen Ausdrucksfähigkeit kann in weiten Teilen durch die Berechnung von Units durch ein mächtigeres System an anderer Stelle ausgeglichen werden. Nehmen wir als Beispiel die Regel, dass eine Frage übersprungen werden soll, falls der Patient jünger als ein bestimmtes Mindestalter ist. In einer höheren Logik könnte dies so aussehen:

$$antwort(alter, X) \wedge lt(X, 18) \rightarrow uebers pringe(fragex)$$

(falls eine Antwort auf die Frage nach dem *alter* existiert und ihr Wert X kleiner als 18 ist, soll die Frage mit dem Name $fragex$ übersprungen werden)
In monadischer Prädikatenlogik lässt sich die kleiner-als-Relation nicht darstellen, statt dessen kann aber eine Relation $lt18$ verwendet werden:

$$lt18(alter) \rightarrow uebers pringe(fragex)$$

Dabei bezeichnet $lt18(alter)$ den Fakt, dass eine Antwort auf die Frage mit dem Name *alter* existiert und der Wert kleiner 18 ist. Die Semantik der Relation $lt18$ wird nicht in den zu den Fragebögen gehörigen Regeln beschrieben, sondern wird als bereits vordefiniert vorausgesetzt. Für eine effiziente Implementierung bietet es sich an, den Interpreter der Inferenzmaschine in der Form zu modifizieren, dass die Interpretation der Relation $lt18$ von einer gesonderten Funktion (Procedural Attachment) statt durch den allgemeinen Inferenzalgorithmus ausgewertet wird.

Diese Vorgehensweise beruht auf einer Trennung des semantischen Hintergrundwissens der Wissensdatenbank in zwei Teile:

1. Den regulären für jeden Fragebogen einzeln definierten Regeln
2. Relationen, die in den Regeln der Fragebögen verwendet werden dürfen ohne vorher definiert zu werden, da sie der Inferenzmaschine bereits als Procedural Attachments bekannt sind

Auf diese Weise wird Komplexität aus den den Regeln der Fragebögen entfernt. Dadurch ergibt sich ein einfaches Design und effiziente Implementierbarkeit. Das

Aufstellen und lesen der Regeln der Fragebögen wird im Vergleich zur Beschreibung durch eine General Purpose Ontology, welche die Domain umfangreich beschreibt, stark vereinfacht. Beschränkt wird diese Vorgehensweise durch die fehlende Ausdrucksfähigkeit. Vor ihrer Verwendung sollte daher gut überlegt werden, ob tatsächlich keine komplexeren Regeln benötigt werden.

2.4 Analyse der vorhandenen Abhängigkeiten in Fragebögen

Im folgenden soll untersucht werden, welche Abhängigkeiten zwischen verschiedenen Fragen existieren können. Dabei soll der Fokus auf den vorhandenen Fragebögen der iSuite liegen.

2.4.1 Einfache Abhängigkeiten

In Fragebögen findet man üblicherweise viele Abhängigkeiten zwischen Antworten verschiedener Fragen. Besonders auffällig ist dies, wenn die Antwort auf eine Frage dazu führt, dass eine Teilmenge der folgenden Fragen übersprungen werden kann. Wenn die Daten des Hausarztes bereits abgefragt wurden, sowie der Fakt, dass der überweisende Arzt der Hausarzt ist, dann sollte nach den Daten des überweisenden Arztes nicht mehr gefragt werden. Eine Regel die dies in monadischer Prädikatenlogik ausdrückt könnte so aussehen:

$$\begin{aligned} & \mathit{wurdeAlsWahrBeantwortet}(\mathit{ueberweisenderArztIstHausarzt}) \\ \wedge\ & \mathit{istBekannt}(\mathit{hausarztName}) \\ \rightarrow\ & \mathit{istBekannt}(\mathit{ueberweisenderArztName}) \end{aligned}$$

Dabei sind *wurdeAlsWahrBeantwortet* und *istBekannt* Relationssymbole; *ueberweisenderArztIstHau*
hausarztName und *ueberweisenderArztName* Konstantensymbole, welche für die entsprechenden Fragen stehen.

Möglicherweise wird nicht nur der Fakt dass die Antwort auf eine Frage geschlossen werden konnte, sondern auch die Antwort selbst benötigt. Um solche Relationen auszudrücken werden zweistellige Prädikate und Variablen benötigt:

$$\begin{aligned} & \mathit{antwort}(\mathit{ueberweisenderArztIstHausarzt}, \mathit{ja}) \\ \wedge\ & \mathit{antwort}(\mathit{hausarztName}, X) \\ \rightarrow\ & \mathit{antwort}(\mathit{ueberweisenderArztName}, X) \end{aligned}$$

2.4.2 Erweiterte Darstellung von Abhängigkeiten

Wenn ein Patient antwortet, dass er keine Schmerzen im Kopfbereich spürt, können sämtliche Detailfragen die der genauerer Schmerzlokalisation im Kopfbereich ausgelassen werden. Dies ließe sich durch Regeln wie die im vorherigen Beispiel definieren. Allerdings will man diese Regeln nicht für jede Detailfrage einzeln beschreiben. Wenn die Relation $detailfrageVon$ bekannt ist, braucht nur eine einzelne allgemeine Regel definiert werden:

$$\begin{aligned} & detailfrageVon(schmerzenKopf, schmerzenStirn) \\ \wedge\ & detailfrageVon(schmerzenKopf, schmerzenOberkiefer) \\ \wedge\ & detailfrageVon(schmerzenKopf, \dots) \end{aligned}$$

$$\begin{aligned} & detailfrageVon(X, Y) \\ \wedge\ & antwort(X, keine) \\ \rightarrow\ & antwort(Y, keine) \end{aligned}$$

Deutlich komplizierter ist es die Regel aus dem vergangenen Abschnitt zu verallgemeinern, sodass für den Fall dass der überweisende Arzt der Hausarzt ist, alle Daten des Hausarztes auch für den überweißenden Arzt gelten sollen. Dafür reicht die bisherige Struktur der Fragebögen nicht aus, es ist weiteres Wissen über die Semantik der Fragen nötig um festzustellen welche Fragen über den Hausarzt mit welchen Fragen über den überweißenden Arzt korrespondieren. Im folgenden ein Beispiel für eine solche Umsetzung:

Die relevanten Objekte/Konzepte (in diesem Fall *hausarzt* und *ueberweisenderArzt*) und Ihre Eigenschaften/Slots (in diesem Beispiel die Daten *name* und *adresse*) müssen modelliert werden, hier geschieht dies in Form von prädikatenlogischen Konstantensymbolen. Die Relation *bedeutet* verknüpft sie mit den Namen der korrespondierenden Fragen:

$$\begin{aligned} & bedeutet(hausarztName, hausarzt, name) \\ \wedge\ & bedeutet(ueberweisenderArztName, ueberweisenderArzt, name) \\ \wedge\ & bedeutet(hausarztAdresse, hausarzt, adresse) \\ \wedge\ & bedeutet(ueberweisenderArztAdresse, ueberweisenderArzt, adresse) \\ \wedge\ & bedeutet(hausarzt\dots, hausarzt, \dots) \\ \wedge\ & bedeutet(ueberweisenderArzt\dots, ueberweisenderArzt, \dots) \end{aligned}$$

Mit einer Regel werden beantwortete Fragen mit einer Instanz in unserem Modell

verknüpft:

$$antwort(N, V) \wedge bedeutet(N, O, P) \leftrightarrow bekannteInstanz(O, P, V)$$

Die Beantwortung der Frage dass die beiden Ärzte die gleichen sind, bedeutet dass die Instanzen in denen die Ärzte als Objekte vorkommen gleiche Modelle besitzen:

$$\begin{aligned} & antwort(ueberweisenderArztIstHausarzt, ja) \\ \leftrightarrow \; (\; & bekannteInstanz(hausarzt, P, V) \\ & \leftrightarrow bekannteInstanz(ueberweisenderArzt, P, V) \quad) \end{aligned}$$

Der Aufwand die Semantik des Fragebogens in dieser Form zu beschreiben sollte nicht unterschätzt werden. Die Mühe kann in vielen Fällen gespart werden, da sich fasst alle in der Praxis auftretenden Abhängigkeiten auch durch unwesentlich mehr sehr einfache Regeln abbilden lassen (siehe 2.3).

Für komplexere Fragebögen ist der Ansatz jedoch interessant. Es sind Abhängigkeiten vorstellbar, die eine sehr ausdrucksstarke Logik benötigen. So zum Beispiel wenn die Antworten komplexe Datentypen sind, welche verglichen werden müssen.

2.4.3 Konsistenzbedingungen

Auch zwischen verschiedenen Antwortmöglichkeiten auf eine einzelne Frage kann es Abhängigkeiten geben: Nehmen wir als Beispiel die Frage „Zu welcher Tageszeit sind Ihre Schmerzen besonders intensiv?“ und die Antwortmöglichkeiten

1. Morgens
2. Tagsüber
3. Abends
4. Nachts
5. Die Schmerzintensität ist über den gesammten Tag konstant

Es ist möglich, dass bei einem Patient mehrere der ersten vier Antworten gleichzeitig zutreffen. Üblicherweise würde man diese Frage daher als Mehrfachauswahl implementieren. Allerdings gibt es auch Antwortkombination, deren Semantik nicht eindeutig ist. Die Kombination von „Die Schmerzintensität ist über den gesammten Tag konstant“ mit der Auswahl einer Tageszeit kann von verschiedenen Betrachtern unterschiedlich interpretiert werden. So könnte das Expertensystem annehmen, dass die einzelne genannte Tageszeit von der letzten Antwort bereits subsumiert wird, während der Patient meinte „Ich habe den ganzen Tag über hohe Schmerzen, zu

einer Tageszeit ist es aber noch etwas schlimmer“. Es ist nicht zielführend, wenn der ausfüllende Patient, das Expertensystem und der Arzt den gleichen Fragebogen verschieden deuten.

Um unnötigen Informationsverlust und Missverständnisse zu vermeiden wäre es das Beste, wenn ein Fragebogen erstellt werden könnte, dessen Semantik für alle — insbesondere den ungeschulten Patienten — intuitiv eindeutig ist. In manchen Fällen besteht eine geeignete Lösung darin Fragen zu teilen. Die oben genannte Frage könnte durch eine Entscheidungsfrage „Gibt es einen Tageszeit, zu der Ihre Schmerzen besonders intensiv sind“ und ausschließlich bei Beantwortung mit „Ja“ mit einer zusätzlichen Frage nach der Tageszeit ersetzt werden. Dieses Verfahren hat den Vorteil, dass die Semantik der Fragen für den Leser strukturell sichtbar und damit besser verständlich wird. Als Nachteil ergibt sich aber die Erhöhung der Anzahl der zu beantworteten Fragen. Auch können von Experten ausgearbeitete, normierte Fragebögen nicht einfach geändert werden. Schließlich ist es auf diese Art auch nicht möglich alle Mehrdeutigkeiten zu beseitigen. So wäre in dem beschriebenen Fall weiterhin die Auswahl aller Tageszeiten möglich.

Von einem intelligenten Dialogsystem kann erwartet werden, dass in solchen Fällen, in denen es die Eingabe nicht oder nicht eindeutig interpretieren kann, ein Fehler ausgeben wird und eine Korrektur der Eingabe verlangt wird. Dafür muss Wissen bereitgestellt werden, aus denen hervorgeht, welche Antwortmöglichkeiten und Kombinationen ungültig sind. Die Verletzung solcher Konsistenzbedingungen sollte an für den Benutzer aussagekräftigen Fehlermeldungen gebunden sein, für dieses Beispiel könnte das folgendermaßen aussehen:

$$
\begin{aligned}
& antwort(schmerzzeit, morgens) \\
\wedge\ & antwort(schmerzzeit, tagsueber) \\
\wedge\ & antwort(schmerzzeit, abends) \\
\wedge\ & antwort(schmerzzeit, nachts) \\
\rightarrow\ & fehler(schmerzzeit, fehlermeldungBitteNichtAlleTageszeiten)
\end{aligned}
$$

2.5 Monotonie

Während des beantwortens der Fragen wird die Wissensdatenbank stetig gefüllt. Für die Weiterverarbeitung des erfragten Wissens ist es irrelevant in welcher Reihenfolge die Fragen erhoben wurden [1]. Weiterhin kann es in Manchen Fällen erforderlich sein, dass bereits beantwortete Fragen zu späterem Zeitpunkt noch korrigiert werden sollen.

[1] Auch wenn die Reihenfolge für die redundanzarme Befragung eine wichtige Rolle spielt

In der Wissensdatenbank soll zu jedem Zeitpunkt nur Wissens enthalten sein, welches unabhängig von den Antworten der verbleibenden Fragen gültig bleibt. Zusätzliche Fakten können das vorhandene Wissen erweitern, jedoch in keinem Fall revidieren. Diese Eigenschaft wird als Monotonie bezeichnet. Dafür ist es nötig, dass keine vorzeitigen Schlussfolgerungen aufgrund der Nichtexistenz von Wissen (siehe Kapitel 2.5.1) gezogen werden.

Im Fall der Korrektur von Eingaben, müssen auch alle aufgrund der vorherigen Antworten geschlossenen Fakten revidiert werden, wenn sie nicht weiterhin gültig sind. Ein System mit dieser Eigenschaft wird als Truth Maintenance System (*TMS*) bezeichnet.

TMS

2.5.1 Monotonieerhaltendes negation-by-failure

Um eine Wissensdatenbank klein zu halten, werden häufig nur wahre Aussagen abgespeichert. Nicht gespeicherte Fakten werden dann implizit als falsch angenommen (closed world assumption). Mit dieser Vorgehensweise können wir beispielsweise die Antworten auf Mehrfachauswahl-Fragen effizient speichern. Angenommen die Frage „Wo am Kopf haben Sie schmerzen?“ kann mit „Rechts“, „Links“ und „Mitte“ beantwortet werden. Wenn ein Patient auf der rechten Seite des Kopfes Schmerzen empfindet ließe sich dies mit $answer(schmerzenKopf, rechts)$ beschreiben. Für die nicht gewählten Antwortmöglichkeiten wird kein zusätzlicher Fakt benötigt.

Die Anfrage $not\ answer(schmerzenKopf, links)$ würde in diesem Fall als wahr ausgewertet werden, da der Fakt $answer(schmerzenKopf, links)$ nicht in der Wissensdatenbank vorkommt. Dies wird als negation-by-failure bezeichnet.

Um in einer zweiwertigen Logik zwischen falschen Aussagen und nicht vorhandenem Wissen zu unterscheiden, werden zusätzliche Aussagen benötigt. So ließe sich im Beispiel der Mehrfachauswahl-Fragen nicht zwischen Fragen bei denen keine Antwortmöglichkeit ausgewählt wurde und unbeantworteten Fragen unterscheiden. Eine zusätzliche Relation *wurdeBeantwortet* ermöglicht diese Unterscheidung.

Eine Regel der Form

$$not\ answer(schmerzenKopf, \_) \rightarrow answer(schmerzenStirn, keine)$$

(mit der Semantik falls ein Patient keinerlei Kopfschmerzen hat, empfindet er auch keine Schmerzen an der Stirn) darf nicht verwendet werden, da sie die Monotonie verletzen würde: Bevor die Frage nach den Kopfschmerzen beantwortet wird, wäre die Prämisse $not\ answer(schmerzenKopf, \_)$ wahr und wurde zu der möglicherweise fälschlichen Konklusion führen, dass der Patient keine Schmerzen an der Stirn hat.

Um die Monotonie zu gewährleisten, muss bei jeder Verwendung von negation-by-failure zusätzlich getestet werden, ob die Fehlenden Fakten auch im endgültigen

Modell (wenn alle Fragen beantwortet wurden) nicht vorkommen werden. Die korrigierte Regel sieht in unserem Beispiel wie folgt aus:

$$\begin{aligned} & wurdeBeantwortet(schmerzenKopf) \\ \wedge\ & not\ answer(schmerzenKopf, \_) \\ \rightarrow\ & answer(schmerzenStirn, keine) \end{aligned}$$

2.6 Zusammenfassung und Bewertung

Logik	Erfüllbarkeitsproblem	
F-Logik / allgemeine Prädikatenlogik (PL)	semi-entscheidbar	NP
prädikatenlogische Hornformeln (PHF)	semi-entscheidbar	NP
allgemeine Aussagenlogik (AL)	entscheidbar	NP
aussagenlogische Hornformeln (AHF)	entscheidbar	P
monadische prädikatenlogische Hornformeln (MHF)	entscheidbar	P

Das Erfüllbarkeitsproblem prädikatenlogischer Hornformeln ist genau wie das der allgemeinen Aussagenlogik (AL) in der Komplexitätsklasse NP. Während PHF nur semi-entscheidbar ist, existieren bei SAT für AL immerhin Algorithmen wie das Anlegen einer vollständigen Wahrheitswertetabelle die entscheidbar sind.

Weder die Einschränkung auf Aussagenlogik (Verzicht auf Quantoren und Begrenzung auf nullstellige Prädikate) noch die Einschränkung auf Hornformeln (Konjunktionen von Disjunktionen mit maximal einem positiven Literal) reichen aus um eine effiziente Entscheidbarkeit zu gewährleisten. Aussagenlogische Hornformeln, die beide Einschränkungen kombinieren, haben die gewünschte Eigenschaft effizient entscheidbar zu sein. Es existieren Algorithmen, welche für jedes is AHF formulierte Problem in $O(n)$ ein Modell finden bzw. ermitteln wenn das Problem unerfüllbar ist. Monadische prädikatenlogische Hornformeln können effizient auf AHF abgebildet werden und besitzen somit die gleichen Eigenschaften bezüglich Erfüllbarkeit wie diese. Sie kann daher empfohlen werden um die Lesbarkeit komplexerer Formeln zu erhöhen.

MHF scheinen eine geeignete Logik zur Darstellung der nötigen Abhängigkeiten zwischen den Fragen zu sein, solange die beschränkte Ausdrucksfähigkeit ausreichend ist. Davon wird vom Autor ausgegangen, allerdings kann es nicht mit Sicherheit bestätigt werden und stellt eine sehr wichtige Entwurfsentscheidung dar. Die monadische Prädikatenlogik kommt im allgemeinen nicht ohne Procedural Attachments in einer höheren Logik aus.

MHF kann bezüglich Ausdrucksfähigkeit als ein Gegenteil von F-Logik verstan-

den werden. Während erstere noch eingeschränkter als AL ist, stellt F-Logik eine vollwertige Prädikatenlogik dar. Dies spiegelt sich auch in der für das Schließen nötigen Zeitkomplexität wieder.

Es bietet sich an die Vorteile beider Varianten zu kombinieren: die Einfachheit der MHF soweit sie ausreichend ist und die Ausdrucksfähigkeit von F-Logik für Ausdrücke, die mit MHF nicht dargestellt werden können. Das gewählte Datenformat zur Definition der Fragebögen berücksichtigt diese Überlegung, indem es beliebige Ausdrücke in F-Logik erlaubt, jedoch die Schnittmenge von F-Logik mit MHF als Best Practice empfiehlt.

3 Umsetzung

3.1 Architektur

3.1.1 Grundlegende Architekturentscheidungen

Aufgrund der Anforderungen, insbesondere dem Wunsch nach einem Mehrbenutzersystem, empfiehlt sich eine Realisierung als Client-Server-Architektur. Ein Daemon stellt als Backend die eigentliche Funktionalität zur Verfügung und sorgt für konsistente Datenspeicherung, während die Frontends zustandsfrei und außschließlich für die Darstellung zuständig sind. Dieses Design ermöglichst den parallelen Einsatz mehrerer, wenn gewünscht auch verschiedener Frontends. Da sämtliche Funktionalität mit höheren Anforderungen an Speicher-/Rechenkapazität im Backend implementiert ist, können als Platform für das Frontend auch Geräte mit begrenzten Ressourcen (z.B. PDAs, Tablet-PCs, ...) problemlos verwendet werden.

Das Sequenzdiagramm in Abbildung 3.2 zeigt die wichtigsten Methoden zur Kommunikation zwischen Front- und Backend und soll die Arbeitsteilung zwischen diesen Komponenten verdeutlichen. Wenn ein Fragebogen dargestellt werden soll, sendet das Frontend zunächst eine Anfrage *get_open_nodes*(), welche das Backend auffordert eine Liste der noch zu beantwortenden Knoten [1] zu übermitteln. Das Backend ermittelt daraufhin, welche Knoten unabhängig von anderen unbeantworteten Fragen beantwortet werden können und sortiert diese in einer geeigneten Reihenfolge (inhaltlich zusammenhängende Fragen sollten in der Liste nahe beieinander stehen). Das Frontend wählt aus der Antwort der *open_nodes* eine Menge von darzustellenden Fragen aus. Die Anzahl sollte an das verwendete Ausgabemedium (z.B. Displaygröße) und an nutzerabhängige Einstellungen (z.B. Schriftgröße) angepasst werden. Für die gewählten Knoten lässt sich das Frontend alle notwendigen Daten vom Backend übermitteln (*get_data_for_node*()) und stellt diese dar.

Wenn ein Nutzer Fragen beantwortet hat, wird ein Dictionary, welches für jede Frage die gewählten Antworten enthält, an das Backend gesendet. Dieses verifiziert die Konsistenz der Antworten und gibt gegebenenfalls aufgetretene Fehler zurück. Um die Interaktivität zu verbessern, kann das Frontend die Methode *set_answers*()

[1] Knoten meint an dieser Stelle Fragen oder Fragebögen. Für die allgemeine Verwendung des Begriffs Knoten in dieser Arbeit siehe Abschnitt 3.2.1.

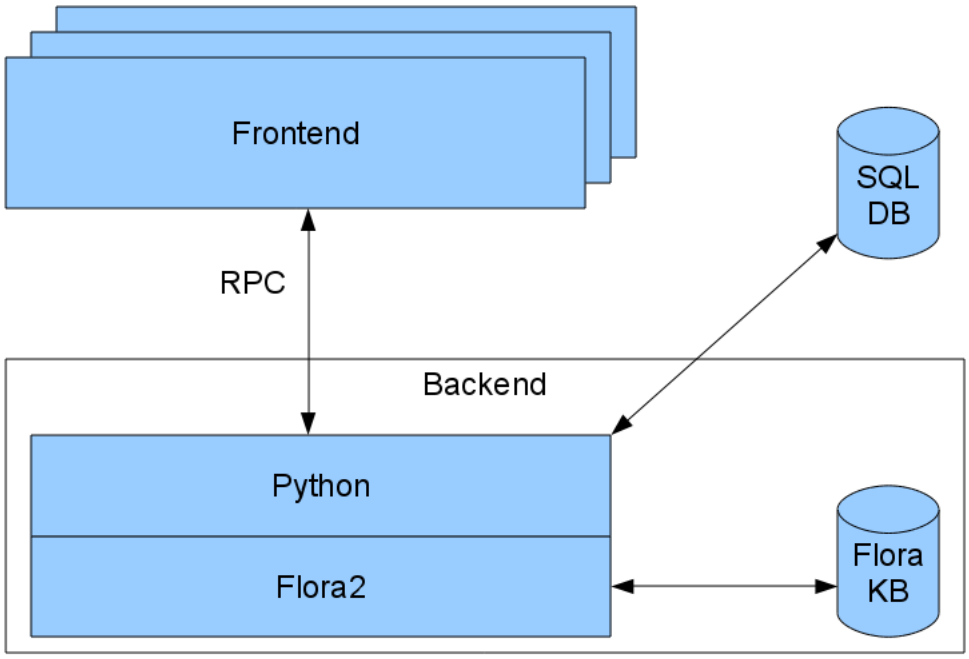

Abb. 3.1: Schematische Darstellung des Aufbaus

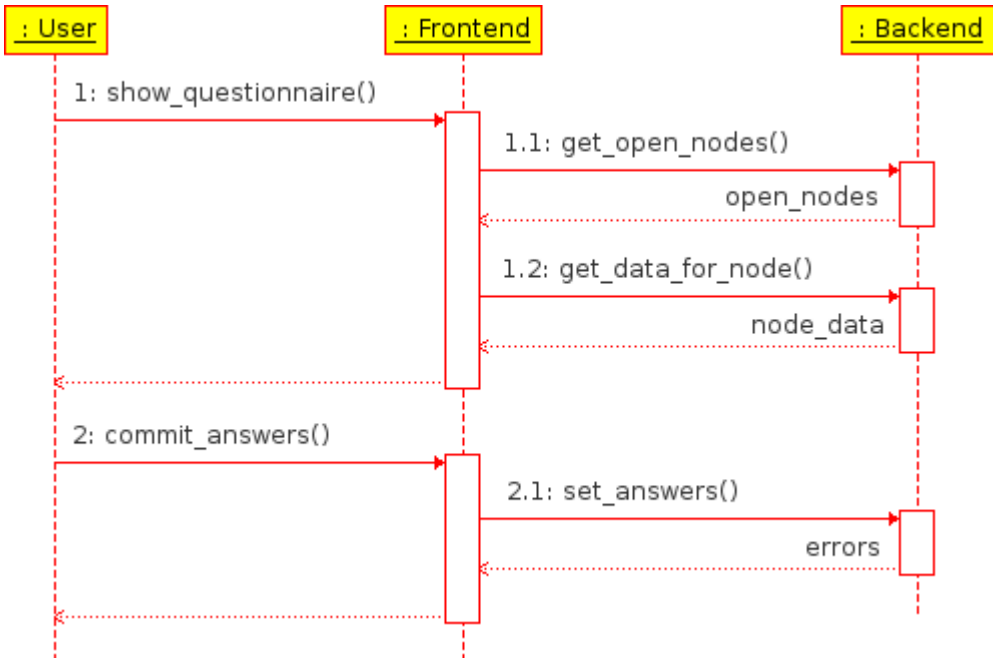

Abb. 3.2: Sequenzdiagramm: Kommunikation zwischen Front- und Backend (wichtigste RPC-Methoden)

nicht ausschließlich beim expliziten Senden der gegebenen Antworten durch den Nutzer, sondern jeweils beim Ausfüllen von (Teil-)Antworten verwenden. Auf diese Weise können automatisch geschlussfolgerte Antworten oder Fehler unverzüglich dargestellt werden. Die zuletzt genannte erhöhte Interaktivität kann im Frontend beispielsweise mittels *Ajax* implementiert werden. *Ajax*

Als Schnittstelle zwischen Frontends und Backend werden Remote Procedure Calls verwendet. Die dafür gewählte Bibliothek erlaubt es, den benutzten RPC-Standart einfach und für die Anwendung transparent zu wechseln. Dies erlaubt Verbindungen abzusichern (Authentifizierung und Verschlüsselung), sowie unkomplizierte Ergänzung weiterer Frontends in beliebigen Programmiersprachen.

Als Frontend wurde bisher eine Webanwendung implementiert, die sowohl Plattformunabhängigkeit, als auch Variabilität in der Darstellung bietet. Als Nebenprodukt der Entwicklung fiel außerdem ein kommandozeilenbasiertes Frontend für bequemes Debugging ab.

3.1.2 Verwendete Programmiersprachen

Große Teile des Backends lassen sich am besten deklarativ beschreiben. Dies gilt gleichermaßen für

- Definition der unterstützten abstrakten Klassen von Fragen (z.B. Einfachauswahl) und ihren Konsistenzbedingungen,
- Definition von Fragebögen als Menge von konkreten Fragen und Relationen zwischen verschiedenen Fragen und
- Logik zur Bestimmung der Abfragereihenfolge.

Da sich für diesen Teil der Implementation das deklarative Programmierparadigma anbietet und da für das Schlußfolgern auf Abhängigkeiten zwischen Fragen sowieso eine Inferenzmaschine benötigt wird, bietet es sich an eine Logik-Programmiersprache zu verwenden. Aufgrund der Vorteile von F-Logik (2.1), insbesondere dem für uns sehr hilfreichen Paradigma der Objektorientierung, wurde sich für die Verwendung dieser Sprache für den Hauptteil des Backends entschieden. Zur Anwendung kommt die unter *LGPL*-lizenzierte freie Implementation *FLORA2* [13]. *LGPL*

Andere Funktionen lassen sich besser unter Verwendung von Standartbibliotheken einer Imperativen Sprache implementieren. Dazu zählen: *FLORA2*

- die Kommunikation mit den Frontends per RPC,
- das persistente Speichern von Ergebnissen in eine SQL-Datenbank.

Für diese Aufgaben traf die Wahl auf Python. Als Schnittstelle zwischen FLORA2 und Python wird das *SWIG*-Interface von ReasonablePython [10] verwendet. *SWIG*

3.2 Verwendete Datenstrukturen

3.2.1 Repräsentation der Fragebögen als Relationen

Die einzelnen Fragebögen, Fragen und Antwortmöglichkeiten sind Konzepte, zwischen denen Relationen bestehen. Dieses Wissen über die zu stellenden Fragen kann als semantisches Netz verstanden werden. Es lässt sich als als Graph visualisieren, indem die Konzepte durch Knoten und die Relationen durch Kanten repräsentiert werden. Die genannten Konzepte werden im folgenden als *Knoten* bezeichnet. *Knoten*

Es werden verschiedene Relationen benötigt.

Offensichtlich bestehen Fragebögen aus Fragen. Für eine effiziente Darstellung ist es sinnvoll, dass Fragebögen zusätzlich aus weiteren Teilfragebögen bestehen dürfen. Manche Arten von Fragen besitzen eine Menge von möglichen Antworten. In der Praxis ist davon auszugehen, das manche Kombinationen von Antwortmöglichkeiten in gleicher Form für viele Fragen auftreten. Für eine redundanzarme Modellierung ist es daher nötig zwischen Antworten und Antwortkombinationen (im weiteren auch als Antwortmenge bezeichnet) zu unterscheiden. Die genannten Aggregationen können durch eine Relation $R_{childs} \subset (N \times N)$, wobei N für die Menge der Knoten steht, beschrieben werden. Jede Frage benötigt maximal eine Antwortmenge, alle anderen Beziehungen sollten allgemein mit der Kardinalität $n : m$ angenommen werden. Wenn Fragebögen oder Fragen, aus Teilfragen bestehen, müssen nicht immer alle Teilfragen beantwortet werden. Ob und welche Teilfragen zu stellen sind, hängt von den Antworten der allgemeineren, übergeordneten Fragen ab.

Die in 2.4 beschriebenen Abhängigkeiten zwischen Fragen bilden ebenfalls Relationen auf der der gleichen Menge von Knoten ($R_{deps} \subset (N \times N)$). Die logischen Ausdrücke aus **??** definieren diese Relationen implizit. Die Formeln, welche Abhängigkeiten definieren, sind immer von der gleichen Form: Sie sind Implikationen, in deren Konklusion genau ein Knoten $k_{Konklusion}$ und in der Prämisse eine Menge von Knoten $K_{Prämissen}$ vorkommen. Die Relation der sich aus einer Formel f ergebenden Abhängigkeiten berechnet sich durch:

$$R^*_{deps}(f) := \{(k_{Konklusion}(f), k_{Prämisse_i}) \mid k_{Prämisse_i} \in K_{Prämissen}(f)\}$$

R_{deps} ergibt sich aus der Vereinigung über die Menge aller Formeln F:

$$R_{deps} := \bigcup_{f \in F} R^*_{deps}(f)$$

Die Relationen zur Darstellung von Aggregationen R_{childs} und der Abhängigkeiten R_{deps} können auch in einer gemeinsamen Relation $R = R_{childs} \cup R_{deps}$ dargestellt werden. $(x,y) \in R$ bedeutet, dass die Beantwortung des Knotens x vom Knoten y

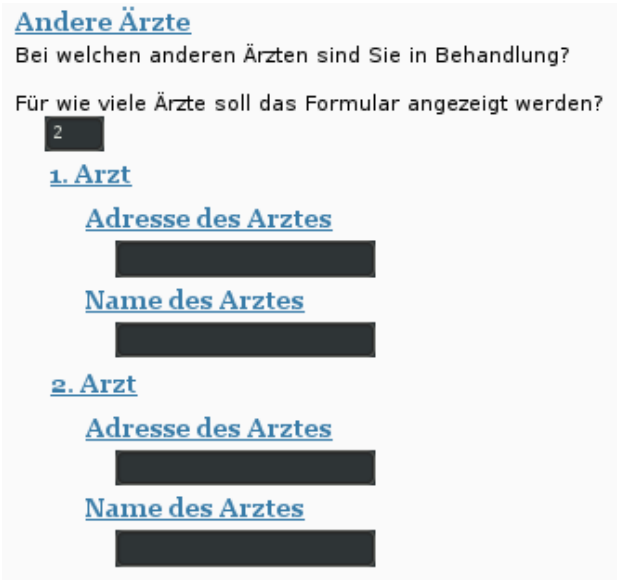

Abb. 3.3: Beispiel für einen Fragebogen mit einem „Multiplikator“

abhängt. Diese Abhängigkeiten müssen zyklenfrei sein, woraus folgt dass nicht nur R, sondern auch ihre Transitive Hülle $TC[R]$ antisymmetrisch sein muss. Es gilt $\forall x,y \in N : (x,y) \in TC[R] \rightarrow \neg(y,x) \in TC[R]$. Da $TC[R]$ transitiv, reflexiv und antisymmetrisch ist, handelt es sich um eine Partialordnung und kann als gerichteter azyklischer Graph (DAG) dargestellt werden.

3.2.2 Generische Datenstruktur zur Darstellung der Ausgabe

Fragebögen setzen sich aus bestimmten Basistypen von Fragen zusammen, dazu zählen:

- Texteingabefeld (Freitext)
- Mehrfachauswahl (Multiple Choice)
- Einfachauswahl

Diese Basistypen treten jedoch nicht nur in ihrer Reinform, sondern auch in vielseitigen Kombinationen auf. Ein typisches Beispiel dafür ist eine Einfach-/Mehrfachauswahl mit der Antwortmöglichkeit „Sonstige“, welche durch ein Texteingabefeld für diesen Fall ergänzt wird. Genauso ist es manchmal sinnvoll, mehrere Einfach-/Mehrfachauswahl-Fragen ineinander zu verschachteln (bei Auswahl einer Antwortmöglichkeit nach weiteren Details zu fragen).

Es wäre sehr aufwendig, die Darstellung der Fragen für jede beliebige Kombination von Fragetypen einzeln zu definieren. Statt dessen empfiehlt es sich eine rekursive Definition zu verwenden, die beliebig verschachtelte Fragen unterstützt.

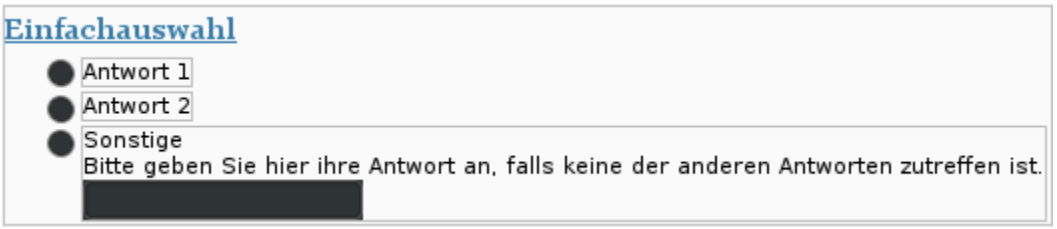

Abb. 3.4: Beispiel für verschachtelte Fragen

Ein weiterer Anwendungsfall für verschachtelte Fragen ist die Benutzung von „Multiplikatoren“. Bild 3.3 zeigt ein solches Beispiel, bei dem der Nutzer bestimmte Fragen (hier Name und Adresse des Arztes) beliebig oft beantworten kann. Diese Funktionalität lässt sich einfach abbilden, indem der neue Fragetyp „Multiplikator“ implementiert wird, welcher einen übergebenen Fragebögen als Klasse auffasst, die in gewünschter Anzahl instanziiert wird.

Alle gemeinsam darzustellenden Fragebögen, Fragen und Antwortmöglichkeiten lassen sich in einer gemeinsamen rekursiven Datenstruktur repräsentieren. Jeder Knoten ist dabei ein Dictionary, welcher vom Typ abhängige Attribute (z.B. „Name“, „Beschreibender Text“, . . .) sowie eine Menge von Kindknoten enthält.

Im Frontend ist für jeden Typ definiert, wie dieser in Abhängigkeit von seinen Attributen dargestellt und an welcher Stelle die Darstellung der Kindknoten plaziert wird. Grafik 3.4 zeigt eine mögliche Darstellung des oben beschriebenen Beispiels, wobei die Bereiche der einzelnen Knoten umrandet wurden.

Die verwendete rekursive Datenstruktur ermöglicht nicht nur eine flexible Verschachtelung von Fragetypen, sondern stellt auch eine einfache Schnittstelle zur Übermittlung der Daten von Knoten vom Backend an verschiedene Frontends dar.

3.2.3 Instanziierung und eindeutige Adressierung von Knoten

In diesem Abschnitt soll besprochen werden, inwieweit es sinnvoll ist, jeden in Abfragen existierenden Knoten einzeln zu definieren oder ob eine gemeinsame Definition für mehrfach verwendete Knoten möglich und zweckmäßig ist.

In üblichen Fragebögen existieren mehrfache Verwendungen des gleichen Knotens. Beispielsweise ist das Auftreten der Antwortmöglichkeit „Sonstige“ oder der Kombination der Anwortmöglichkeiten „Ja“ und „Nein“ gehäuft zu erwarten. Es ist wünschenswert, dass Informationen über diese Knoten nicht mehrfach und gegebenenfalls inkonsistent gespeichert werden. Diese Überlegung spielt insbesondere dann eine Rolle, wenn für die Knoten umfangreiche Datenmengen wie beispielsweise Übersetzungen in verschiedene Sprachen oder komplexe Abhängigkeiten (siehe 2.4) existieren.

Wenn alle dargestellte Fragen einzeln und unabhängig voneinander definiert werden sollen, führt dies nicht nur unter Umständen zu unnötiger Redundanz, sondern stellt auch eine Einschränkung der Funktionalität dar. Die in Abschnitt 3.2.2 beschriebenen „Multiplikatoren“ sind eine Art von Knoten, welche einmal definierte *Prototypen* von Knoten beliebig oft verwenden können.

Knoten-Prototyp

Jeder Knoten-Prototyp kann beliebig oft abgeleitet werden. Jede *Knoten-Instanz* ist eine Ableitung von genau einem Prototyp. Zwischen Prototypen und Instanzen besteht also eine $1 : n$-Beziehung.

Knoten-Instanz

$instancesOf$ und $prototypeOf$ sind zueinander inverse Relationen mit der Bedeutung $(i,p) \in instancesOf \leftrightarrow (p,i) \in prototypeOf$ genau dann, wenn i eine vom Prototyp p abgeleitete Instanz ist.

Am Beispiel der „Multiplikatoren“ wird schnell eine Schwierigkeit deutlich: Verschiedene Instanzen der gleichen Frage müssen in Inferenzregeln und zur Speicherung der Antworten unterschieden werden. Zur eindeutigen Identifizierung wird jeder Instanz eines Knotens ein Wert zugeordnet, welcher im Folgenden als *Adresse* bezeichnet wird.

Adresse einer Knoteninstanz

Trivialer Fall: Knoten mit genau einer Instanz

Übliche Fragebögen haben viele Knoten, bei denen von genau einer Instanz ausgegangen werden kann. Dies kann als Spezialfall von allgemein $n \in \mathbb{N}$ Instanzen betrachtet werden. Für eine einzige Instanz reicht es aus, wenn die Adresse ausschließlich von einer eindeutigen Knoten-ID des zugehörigen Prototyps abhängt:

$$Adresse(i) \coloneqq f(ID(p)) \mid (i,p) \in instanceOf \land p \in P_{oneInstance}, \text{ wobei}$$
$$P_{oneInstance} \coloneqq \{p \mid \forall i_1, i_2.(i_1,p) \in instanceOf \land (i_2,p) \in instanceOf \rightarrow i_1 = i_2\}$$

allgemeiner Fall

Für den allgemeinen Fall ist es naheliegend die Knoten-ID des Prototyps ebenfalls in die Adresse zu kodieren. Zusätzlich muss die Adresse jedoch von einem Merkmal abhängen, welche sich für jede Instanz des gleichen Prototyps unterscheidet. Im einfachsten Fall kann dafür ein einfacher Zähler C verwendet werden:

$$Adresse(i) \coloneqq f(ID(p), C(i)) \mid (i,p) \in instanceOf, \text{ mit}$$
$$C(i_1) = C(i_2) \rightarrow i_1 = i_2 \lor [\neg\exists p.(p,i_1) \in prototypeOf \land (p,i_2) \in prototypeOf]$$

mehrfache Instanzen von zusammengesetzten Knoten

Wenn ein Prototyp p mehrfach instanziiert wird und selbst aus weiteren Teilfragebögen oder Teilfragen besteht ($\exists x.(x,p) \in R_{childs}$, siehe Abschnitt 3.2.1), dann müssen

auch diese Knoten (implizit) mehrfach instanziiert werden und dabei unterschiedliche Adressen zugewiesen bekommen. Dies setzt sich transitiv fort.

Falls auf allen Pfaden des von R_{childs} aufgespannten DAG jeweils maximal ein Knoten explizit[2] mehrfach instanziiert wird, ist eine einfache Adressierung aller Knoten möglich: Ohne Beschränkung der Allgemeinheit soll ein einzelner Pfad betrachtet werden, dessen m-ter Knoten-Prototyp mehrfach explizit instanziiert wurde (z.B. indem p_m vom Typ „Multiplikator“ ist):

$$Pfad = (p_1, \ldots, p_{m-1}, p_m, p_{m+1}, \ldots, p_n)$$

Für den Pfad soll gelten, dass p_1 eine Wurzel[3] und p_n ein Blatt des DAG ist:

$$\neg\exists x.(p_1, p_x) \in R_{childs}$$
$$\forall x.(p_{x+1,p_x}) \in R_{childs}$$
$$\neg\exists x.(p_x, p_n) \in R_{childs}$$

Entsprechend der Annahme, dass mit Ausnahme von p_m alle Prototypen $p_x \in Pfad$ genau eine explizite Instanz besitzen, definieren wir:

$$P_{oneExplicitInstance} \coloneqq \{p_x \in Pfad \mid x \neq m\}$$

Für alle Knoten y mit $y < m$ gilt:

$$p_y \in P_{oneExplicitInstance} \land \neg\exists x.[p_x \notin P_{oneExplicitInstance} \land x < y]$$

Und folglich[4], weiterhin für den Fall das $y < m$, auch:

$$p_y \in P_{oneExplicitInstance} \land \neg\exists x.[p_x \notin P_{oneExplicitInstance} \land (p_y, p_x) \in R_{childs}]$$

Da alle Prototypen p_y mit $y < m$ genau eine explizite Instanz haben und wie gezeigt auch nicht Teil eines Knotens sein können, welcher mehrere Instanzen benötigt, sind auch keine weiteren impliziten Instanzen erforderlich. Die Adressierung für diese Knoten ist folglich trivial.

Für die Knoten $y > m$ gilt analog dazu, dass die Instanzen nur von der Instanziierung des Knotens m abhängen. Aus dem Original-DAG der Prototypen lässt sich ein DAG der impliziten Instanzen ableiten, indem alle Teilbäume unterhalb von m entsprechend der Anzahl der expliziten Instanzen von p_m (abzüglich des einen vorhandenen Originals) kopiert werden.

[2] im Gegensatz zum oben genannten impliziten Instanziieren
[3] Da es sich im allgemeinen um einen Multibaum handelt, kann es mehrere Wurzel-Knoten geben
[4] beweisbar durch vollständige Induktion über y

Für die Adressen von $y \geq m$ wird lediglich der Zähler C_m des Knotens p_m benötigt.

4 Fazit

Es konnte erfolgreich gezeigt werden, dass monadische prädikatenlogische Hornformeln eine geeignete Logik zur Darstellung von Fragen, Antworten und den zwischen ihnen befindlichen Abhängigkeiten sind. Die für iSuite zur Verfügung stehenden Fragebögen und alle zwischen ihnen bestehenden Abhängigkeiten, die gefunden wurden, konnten mit der Syntax von F-Logik und unter beschränkung auf MHF repräsentiert werdern. Dies erlaubt effektives Knowledge Retrieval in $O(n)$. Der Koeffizient n ist dabei überschaubar klein, da zur Berechnung der allgemeinsten noch unbeantworteten Fragen neben dem allgemeinen Wissen über die verwendeten Fragebögen ausschließlich das Wissen über den derzeitigen Nutzer, jedoch nicht das zu anderen Fällen, benötigt wird.

Das in F-Logik implementierte Backend des entwickelten intelligenten Dialogsystems befindet sich im Anhang der Arbeit. Bei der Umsetzung wurden die deduktive Datenbank XSB [1] und ihrer Erweiterung um F-Logik FLORA2 [13] als Inferenzmaschine verwendet. Die Ergebnisse dieser Arbeit legen nahe, dass sich MHF auch für die konzeptbasierte Inferenz des iSuite-Expertensystem anbietet. In vielen Fällen, in denen MHF nicht hinreichend ausdrucksstark sind, kann eine Erweiterung durch Procedural Attachments Abhilfe schaffen und gleichzeitig die niedrige Komplexität bei der Inferenz beibehalten.

Anhang

Anhang A — Original Lisp-Format zur Speicherung von Fragebögen

Dieses Beispiel zeigt das Format in welchem die Fragebögen in ISuite bisher dargestellt wurden:

```
(a$x0x0 "Mund / Gesicht / Kopf" (a$10100 a$10101
   a$10102 a$10103 a$10104 a$10105 a$10106 a$10107
   a$10108 a$10109))

(a$10105 "Ohr" (a$101xx11 a$101xx12))

(a$101xx11 "links")
(a$101xx12 "rechts")
```

Um die vorhanderen Fragebögen automatisch in ein neues Format zu konvertieren wurde ein Parser geschrieben. Er geht davon aus, dass das original Format durch diese EBNF-Regeln beschrieben werden kann:

```
<line>     := (<relevant>)[[<space>];<comment>]
<relevant> := <id> "<name>"[ <rest>]
<rest>     := ([<ref>{ <ref>}])[ <nodeType>]
<nodeType> := nu|oc|te
```

Anhang B — XML-Format zur Speicherung von Fragebögen

Das Original Format hatte für eine Implementierung in Lisp den Vorteil, dass diese nativen Datenstrukturen ohne externe Bibliotheken eingelesen werden können. Für andere Programmiersprachen wird jeweils ein Parser benötigt. Um beim Bearbeiten der Fragebögen die Unterstützung einer größeren Anzahl von vorhandenen Editoren und Werkzeuge wie Syntaxhighlighting, Validierung und Linter zu verwenden, bietet es sich an ein verbreiteteres Datenaustauschformat wie JSON, XML oder YAML zu verwenden.

Für den implementierten Prototyp wurde ein XML-Dokumentenformat entworfen, dessen Schema im Vergleich zum Original folgende zusätzliche Feature zulasst:

- Lokalisierung um Mehrsprachige Fragebögen/Dialoge zu ermöglichen.
- Unterscheidung zwischen Überschrift (name) und Freitextes (text) von Fragen wie in Abbildung 3.3 dargestellt.
- Eine zusätzliche Indirektion „AnswerList“ fasst die möglichen Antworten von Fragen die eine Auswahl verlangen zusammen und macht die Kombination von Antworten einfacher wiederverwendbar indem Redundanz eliminiert wird.

```
<?xml version="1.0" ?>
<xml>
  <Questionnaire id="a$x0x0">
    <lang id="de" name="Mund / Gesicht / Kopf"
       text="Körperregion: Mund / Gesicht / Kopf"/>
    <lang id="en" name="Mouth / Face / Head"
       text="Bodyregion: Mouth / Face / Head"/>
    <child id="a$10100"/>
    <child id="a$10101"/>
    <child id="a$10102"/>
    <child id="a$10103"/>
    <child id="a$10104"/>
    <child id="a$10105"/>
    <child id="a$10106"/>
    <child id="a$10107"/>
    <child id="a$10108"/>
    <child id="a$10109"/>
  </Questionnaire>
```

```
  <QuestionForMultipleChoise answerList=
     "53b054dac85fb849d73268c1380f3791ac958e5c"
     id="a$10105" minSelections="0">
    <lang id="de" name="Ohr" text="Im oder am Ohr:"/>
    <lang id="en" name="Ear" text="Inside or at ear:"/>
  </QuestionForMultipleChoise>
  <AnswerList
     id="53b054dac85fb849d73268c1380f3791ac958e5c">
    <answer id="a$101xx11"/>
    <answer id="a$101xx12"/>
  </AnswerList>
  <AnswerList
     id="7b6cd229fe4106a9219433e00a0db3bed701d940">
    <answer id="a$101xx11"/>
    <answer id="a$101xx12"/>
    <answer id="a$101xx13"/>
  </AnswerList>
  <Answer id="a$101xx11">
    <lang id="de" name="links"/>
    <lang id="en" name="left"/>
  </Answer>
  <Answer id="a$101xx12">
    <lang id="de" name="rechts"/>
    <lang id="en" name="right"/>
  </Answer>
  <Answer id="a$101xx13">
    <lang id="de" name="mitte"/>
    <lang id="en" name="center"/>
  </Answer>
</xml>
```

Anhang C — F-Logik Repräsentation der Fragebögen

Aus den Fragebögen im XML-Format kann eine Repräsentation in F-Logic bzw. Prolog compiliert werden. Für das obrige Beispiel sieht diese so aus:

```
a_24_x0x0:Questionnaire [
 node_orig -> 'a$x0x0',
 title("de") -> 'Mund / Gesicht / Kopf',
 title("en") -> 'Mouth / Face / Head',
 text("de") -> 'Körperregion: Mund / Gesicht / Kopf',
 text("en") -> 'Bodyregion: Mouth / Face / Head',
 childs -> {a_24_10100, a_24_10101, a_24_10102,
    a_24_10103, a_24_10104, a_24_10105, a_24_10106,
    a_24_10107, a_24_10108, a_24_10109}
].

a_24_10105:QuestionForMultipleChoice [
 node_orig -> 'a$10105',
 title("de") -> 'Ohr',
 title("en") -> 'Ear',
 text("de") -> 'Im oder am Ohr:',
 text("en") -> 'Inside or at ear:',
 forward_childs -> al_a_24_10105
].

al_a_24_10105:AnswerList [
 childs -> {a_24_101xx11, a_24_101xx12}
].
a_24_101xx11:Answer [
 node_orig -> 'a$101xx11',
 title("de") -> 'links',
 title("en") -> 'left'
].
a_24_101xx12:Answer [
 node_orig -> 'a$101xx12',
 title("de") -> 'rechts',
 title("en") -> 'right'
].
```

Das Beispiel von Abbildung 3.3 mit dem in 3.2.2 beschriebenen „Multiplikator“:

```
qn_aerzte:Questionnaire [
 title("de") -> 'Fragen zu behandelnden Ärzten',
 title("en") -> 'Questions about treating physicians',
 childs -> {q_ueberweisenderArzt,
       q_hausarzt,
       mu_andereAerzte}
].

q_hausarzt:QuestionForText [
 title("de") -> 'Name des Hausarztes',
 title("en") -> 'Family physician'
].

q_ueberweisenderArzt:QuestionForText [
 title("de") -> 'Name des Überweisenden Arztes',
 title("en") -> 'Name of transfering physician'
].

mu_andereAerzte:Multiplier [
 title("de") -> 'Andere Ärzte',
 title("en") -> 'Other physicians',
 text("de") -> 'Bei welchen anderen Ärzten sind Sie in
    Behandlung?\n\nFür wie viele Ärzte soll das
    Formular angezeigt werden?',
 instance_title("de") -> '%d. Arzt',
 instance_title("en") -> '%d. physician',
 inherit_childs -> {q_arzt_name, q_arzt_adresse}
].

q_arzt_name:QuestionForText [
 title("de") -> 'Name des Arztes',
 title("en") -> 'Name of physician'
].
q_arzt_adresse:QuestionForText [
 title("de") -> 'Adresse des Arztes',
 title("en") -> ' Address of physician'
].
```

Das folgende Beipiel zeigt ein typisches Beispiel für zwei Fragen zwischen denen eine Abhängkeit besteht. Nur wenn die erste, allgemeinere Frage nach dem Geschlecht mit „weiblich“ beantwortet wurde ist es sinnvoll die zweite Frage nach einer Schwangerschaft zu stellen:

```
q_gender:QuestionForOneChoice [
 title("de") -> 'Geschlecht',
 title("en") -> 'Gender',
 text("en") -> 'Which gender?',
 forward_childs -> al_gender
].

 al_gender:AnswerList [
  childs -> {a_gender_m, a_gender_w}
 ].

  a_gender_m:Answer [
   title("de") -> 'männlich',
   title("en") -> 'male'
  ].

  a_gender_w:Answer [
   title("de") -> 'weiblich',
   title("en") -> 'female'
  ].

q_schwanger:QuestionForOneChoice [
 title("de") -> 'schwanger',
 title("en") -> 'pregnant',
 text("de") -> 'Sind sie derzeit schwanger?',
 text("en") -> 'Are you pregnant?',
 forward_childs -> al_yesno
].
```

```
al_yesno:AnswerList [
 childs -> {a_yes, a_no}
].

 a_yes:Answer [
  title("de") -> 'Ja',
  title("en") -> 'Yes'
 ].

 a_no:Answer [
  title("de") -> 'Nein',
  title("en") -> 'No'
 ].
```

Die notwendige Abhängigkeit kann durch hinzufügen einer einzelnen Hornklausel zum Fragebogen implementiert werden:

```
inferredAnswer(q_schwanger(?Path), a_no, ?User, ?TS)
   :- wasValidAnswer(q_gender(?Path), a_gender_m,
   ?User, ?TS).
```

Ausgeschrieben bedeutet diese Regel: Wenn es für einen Patienten $?USER$ für eine Episode (zu einem Zeitpunkt) $?TS$ die Antwort auf die Frage nach dem Geschlecht als männlich bekannt ist, kann daraus gefolgert werden dass für den Patienten gilt, dass die Frage nach Schwangerschaft als mit Nein zu beantworten gilt.

Anhang D — Backend Implementierung

Im folgenden wird der vollständige FLORA2-Code aufgelistet, der notwendig ist, um alle gefundenen Abhängigkeiten (Kapitel 2.4) darzustellen, Monotonie (Kapitel 2.5) zu gewährleisten und effizientes Knowledge Retrieval unter Beachtung aller benötigten Inferenzregeln zu ermöglichen.

```
/** NodeInstances inherit everything from according
   Node **/

?N(?_Path):NodeInstance[
  ?Attribute -> ?Value
] :-
  ?N:Node[
    ?Attribute -> ?Value
  ],
  not ?Attribute = class.

/** Nodetypes as objects **/

Node::object [
 node => string,
 title(lang) => string,
 text(lang) => string,
 childs => Node
].
?Node[node -> ?Node] :- ?Node:Node.
?Node[class_all -> ?Class] :-
  ?Node:?Class,
  (?Class::Node; ?Class::NodeInstance).
?Node[class -> ?Class] :-
  ?Node[class_all -> ?Class],
  not (( ?Child::?Class, ?Node[class_all -> ?Child] )),
  not (( ?Class::Node, ?Node[class_all ->
     ?_InstanceClass::NodeInstance] )).

?N[childsRecursive -> ?C] :- ?N[childs -> ?C].
?N[childsRecursive -> ?CR] :-
  ?N[childsRecursive -> ?C],
  ?C[childs -> ?CR].
```

Für jeden Knotentyp wird eine Klasse definiert, welche insbesondere notwendige semantische Beschreibungen beinhaltet. So beispielsweise die Regel, dass bei einer Einfachauswahl wiedersprüchliche Antworten (vom Nuter gegeben oder inferiert) zu als ungültig interpretierten Antworten und einer Fehlermeldung führen.

```
Answer::Node.

AnswerList::Node [
 childs => Answer
].
?Node[childs -> ?Answer] :-
   ?Node[forward_childs -> ?AL],
   ?AL[childs -> ?Answer].

ToBeAnswered::Node.

Questionnaire::ToBeAnswered [
 childs => Question
].

Question::ToBeAnswered.

QuestionForText::Question [
 default *-> ''
].

QuestionForInt::QuestionForText.

QuestionForChoice::Question [
 forward_childs => AnswerList,
 default *-> []
].

QuestionForMultipleChoice::QuestionForChoice.

QuestionForOneChoice::QuestionForChoice.

invalidAnswer(?Question(?Path), ?User, ?_TS2,
    error_oneChoice) :-
   ?Question:QuestionForOneChoice,
```

```
  lastSuggestedAnswer(?Question(?Path), ?_A1, ?User,
     ?_TS1),
  lastSuggestedAnswer(?Question(?Path), ?_A2, ?User,
     ?_TS2),
  not ?_A1 :=: ?_A2,
  ?_TS1 =< ?_TS2.

Multiplier::QuestionForInt [
 inherit_childs => Question,
 default *-> 1,
 size *-> 2
].
MultiplierInstance::NodeInstance.
?Mu([1]):MultiplierInstance[childs -> ?Childs] :-
  ?Mu:Multiplier[inherit_childs -> ?Childs].
?Mu([?Count]):MultiplierInstance[childs -> ?Childs] :-
  ?Mu:Multiplier[inherit_childs -> ?Childs],
  validAnswer(?Mu(?_Path), ?Count, ?_User, ?_TS).
?Mu([?Count]):MultiplierInstance[childs -> ?Childs] :-
  ?Mu([?BiggerCount]):MultiplierInstance[childs ->
     ?Childs],
  ?Count is ?BiggerCount -1,
  ?Count > 0.
answeringExpectedRecursive(?Mu([?Count|?Path]), ?User)
   :-
  ?Mu([?Count]):MultiplierInstance,
  answeringExpectedRecursive(?Mu(?Path), ?User),
  validAnswer(?Mu(?Path), ?CountMax, ?User, ?_TS),
  ?Count =< ?CountMax.

invalidAnswer(?Question(?Path), ?User, ?TS,
   type_error) :-
  ?Question:Multiplier,
  suggestedAnswer(?Question(?Path), ?Answer, ?User,
     ?TS),
  not ?Answer:_integer.
```

Die folgenden beiden Seiten implementieren *getDataForNode*, um wie im Sequenzdiagramm 3.2 skizziert die bekannten statischen und dynamischen Daten eines Kontens abzurufen. Außerdem können per *getTimeline* Details über die Episode zu Debugzwecken abgefragt werden.

```
/** some getters for querying the internal logic **/


/* relation with all language specific attributes
   which are optional */

optionalAttrLang(text).

/* get language specific data of a node */

getDataForNode(?User, ?Node, ?Attribute, ?Value) :-
   ?User[lang -> ?Lang], ?Node[?Attribute(?Lang) ->
   ?Value].
getDataForNode(?_User, ?Node, ?Attribute, ?Value) :-
   ?Node[?Attribute -> ?Value], ?Attribute =..
   [?Attribute | []].
getDataForNode(?User, ?Node, ?Attribute, '') :-
  optionalAttrLang(?Attribute),
        ?User[lang -> ?Lang],
  not ?Node[?Attribute(?Lang) -> ?_Value].

/* get validAnswer or default value */

getDataForNode(?User, ?Node(?_Path), validOrDefault,
   ?Answer) :-
  validAnswer(?Node(?_Path), ?Answer, ?User, ?_TS).
getDataForNode(?User, ?Node(?_Path), validOrDefault,
   ?Default) :-
  ?Node[default -> ?Default],
  not validAnswer(?Node(?_Path), ?_Answer, ?User,
     ?_TS),
  answeringExpectedRecursive(?Node(?_Path), ?User).

/* get the last infered answer */
```

```
getDataForNode(?User, ?Node(?Path),
    lastInferredAnswer, ?Answer) :-
  inferredAnswer(?Node(?Path), ?Answer, ?User, ?TS),
  lastSuggestedAnswer(?Node(?Path), ?Answer, ?User,
      ?TS).

/* get NodePath as String or List, depending on being
   an NodeInstance or not */

getDataForNodeFormatNodes(?User, ?Node, ?Attribute,
    ?Value) :-
  getDataForNode(?User, ?Node, ?Attribute, ?Value),
  not ?Value:NodeInstance.
getDataForNodeFormatNodes(?User, ?Node, ?Attribute,
    [?N, ?P]) :-
  getDataForNode(?User, ?Node, ?Attribute, ?N(?P)),
  ?N(?P):NodeInstance.

/* get all events interesting to print a timeline (for
   debugging) */

getTimeline('givenAnswer',     ?Node(?Path), ?Answer,
    ?User, ?TS) :-
        givenAnswer(      ?Node(?Path), ?Answer, ?User,
            ?TS).
getTimeline('inferredAnswer', ?Node(?Path), ?Answer,
    ?User, ?TS) :-
        inferredAnswer(   ?Node(?Path), ?Answer, ?User,
            ?TS).
getTimeline('validAnswer',     ?Node(?Path), ?Answer,
    ?User, ?TS) :-
        validAnswer(      ?Node(?Path), ?Answer, ?User,
            ?TS).
getTimeline('invalidAnswer',   ?Node(?Path), ?Err,
    ?User, ?TS) :-
        invalidAnswer(    ?Node(?Path), ?User, ?TS,
            ?Err).
```

Der Verbleibende Code implementiert die interne Logik. Wichtigstes Interface ist *openNodeIndependent*, welche die Knoten zurück gibt, die noch unbekannt sind (*openNodes*) und gleichzeitig keine Abhängigkeiten zu weiteren *openNodes* haben.

```
/** This file defines the internal logic used to
    handle knowledge about answered/open questions **/


/* users are represented as strings */

User::string.


/* suggestedAnswers are all manual given Answers and
    inferred Answers */

suggestedAnswer(?Question, ?Answer, ?User, ?TS) :-
  givenAnswer(?Question, ?Answer, ?User, ?TS).
suggestedAnswer(?Question, ?Answer, ?User, ?TS) :-
  inferredAnswer(?Question, ?Answer, ?User, ?TS).

/* the last suggestedAnswer */

lastSuggestedAnswer(?Question, ?Answer, ?User, ?TS) :-
  /* Case: Only givenAnswer; (without simultaneous)
     inferedAnswer */
  suggestedAnswer(?Question, ?Answer, ?User, ?TS),
  not inferredAnswer(?Question, ?Answer, ?User, ?TS),
  /* Newest answer substitute old ones */
  not ((suggestedAnswer(?Question, ?_, ?User,
     ?TSLater), ?TSLater > ?TS)).
lastSuggestedAnswer(?Question, ?Answer, ?User, ?TS) :-
  suggestedAnswer(?Question, ?Answer, ?User, ?TS),
  /* There is no newer answer for any question */
  not ((suggestedAnswer(?_, ?_, ?User, ?TSLater),
     ?TSLater > ?TS)).

/* suggestedAnswers that have not been invalid at the
    same time */
```

```
wasValidAnswer(?Question, ?Answer, ?User, ?TS) :-
  suggestedAnswer(?Question, ?Answer, ?User, ?TS),
  not invalidAnswer(?Question, ?User, ?TS, ?_).

/* last suggested answer that was valid */

validAnswer(?Question, ?Answer, ?User, ?TS) :-
  lastSuggestedAnswer(?Question, ?Answer, ?User, ?TS),
  wasValidAnswer(?Question, ?Answer, ?User, ?TS).

/* Answers are accepted before and till given TS =>
   can be asked for (now-timeout) */

acceptedAnswer(?Question, ?Answer, ?User, ?TSBefore) :-
  validAnswer(?Question, ?Answer, ?User, ?TS),
  ?TSBefore =< ?TS.

/* Pathes of nodes which where expected to be answered
   explicite or implicit (child of to be answered
   node) */

answeringExpectedRecursive(?Node(?Path), ?User) :-
   answeringExpected(?Node(?Path), ?User).
answeringExpectedRecursive(?Node(?Path), ?User) :-
  answeringExpectedRecursive(?P(?Path), ?User),
  ?P(?Path):NodeInstance[childsRecursive ->
     ?Node:ToBeAnswered].

/* expected to be answered; but not answered now */

openNode(?NI, ?User, ?TS) :-
       answeringExpectedRecursive(?NI:NodeInstance,
          ?User:User),
       not acceptedAnswer(?NI, ?_, ?User, ?TS).
openNode(?Parent(?Path), ?User, ?TS) :-
        answeringExpectedRecursive(?Parent(?Path),
           ?User:User),
  ?Parent[childsRecursive -> ?Node:ToBeAnswered]:Node,
  openNode(?Node(?Path):NodeInstance, ?User, ?TS).
```

```
/* dependency-relation between questions
 * implicitly known from rules of type
    "inferedAnswer(...) :- wasValidAnswer(...), ..."
 */

?Question[dependency -> ?Dependency] :-
  clauselist(?_H, ?_T, ?_TR, [?_Term, ?Dependency |
     ?_]),
  ?_Term =.. [?_, wasValidAnswer, ?_],
  ?_H =.. [?_HF, ?Question | ?_],
  ?_HF =.. [?_, inferredAnswer, ?_].

/* openNodes without dependencies to other openNodes */

openNodeIndependent(?Node, ?User, ?TS) :-
  openNode(?Node, ?User, ?TS),
  not (( Node[dependency -> ?D], openNode(?D, ?User,
     ?TS) )).

/* openNode without childs that are partially known */

absolutelyOpenNode(?Node(?Path), ?User, ?TS) :-
  openNode(?Node(?Path), ?User, ?TS),
  not (( ?Node:ToBeAnswered,
         ?Node(?Path)[childsRecursive ->
            ?_ChildI:ToBeAnswered],
         not openNode(?_ChildI(?Path), ?User, ?TS) )).

/* absolutelyOpenNodes, without dependencies to other
   openNodes */

absolutelyOpenIndependent(?Node, ?User, ?TS) :-
  openNodeIndependent(?Node, ?User, ?TS),
  absolutelyOpenNode(?Node, ?User, ?TS).

/* absolutelyOpenIndependent Nodes without parent
   which is also absolutelyOpenIndependent */
```

```
absolutelyOpenIndependentTop(?Node(?Path), ?User, ?TS)
  :-
  absolutelyOpenIndependent(?Node(?Path), ?User, ?TS),
  not (( ?Parent(?_ParentPath)[childsRecursive ->
     ?Node],
     absolutelyOpenIndependent(?Parent(?ParentPath),
     ?User, ?TS), ?Path = ?ParentPath )).

/* This function should be called when an answer has
   been added.
 * It takes care about tabling/memoization and ensures
    that monotonie is maintained */

%refresh_answers(?User) :-
  %refresh(?_User),
        refresh{answeringExpected(?_NI, ?User)},
        refresh{answeringExpectedRecursive(?_NI,
           ?User)},
  refresh{openNode(?_NI, ?User, ?_TS)},
  refresh{openNodeIndependent(?_NI, ?User, ?_TS)},
  refresh{absolutelyOpenNode(?_NI, ?User, ?_TS)},
  refresh{absolutelyOpenIndependent(?_NI, ?User,
     ?_TS)},
  refresh{absolutelyOpenIndependentTop(?_NI, ?User,
     ?_TS)},
  refresh{givenAnswer(?_Q, ?_A, ?User, ?_TS)},
  refresh{inferredAnswer(?_Question, ?_Answer, ?User,
     ?_TS)},
  refresh{suggestedAnswer(?_Question, ?_Answer, ?User,
     ?_TS)},
  refresh{lastSuggestedAnswer(?_Question, ?_Answer,
     ?User, ?_TS)},
  refresh{invalidAnswer(?_Question, ?User, ?_TS, ?_)},
  refresh{wasValidAnswer(?_Question, ?_Answer, ?User,
     ?_TS)},
  refresh{validAnswer(?_Question, ?_Answer, ?User,
     ?_TS)},
  refresh{acceptedAnswer(?_Question, ?_Answer, ?User,
     ?_TS)}.
```

Nomenklatur

AHF	Aussagenlogische Horn-Formeln (Seite 5)
Ajax	Asynchronous JavaScript and XML (Seite 20)
FLORA2	Freie Implementation von F-Logik (Seite 20)
HORNSAT	Horn-satisfiability / Entscheidungsproblem ob eine aussagenlogische Horn-Formel erfüllbar ist (Seite 5)
Knoten	Der Begriff Knoten wird in dieser Arbeit allgemein für die Konzepte verwendet, aus denen sich Fragebögen zusammen setzen (Fragebögen, Fragen, Antwortmöglichkeiten) (Seite 21)
LGPL	GNU Lesser General Public License (Seite 20)
MHF	Monadische prädikatenlogische Hornformeln (Seite 9)
OOP	objektorientierten Programmierung (Seite 3)
PHF	Prädikatenlogische Hornformeln erster Stufe (Seite 8)
SAT-Problem	satisfiability problem / Entscheidungsproblem das fragt ob eine (aussagenlogische) Formel erfüllbar ist (Seite 5)
SWIG	Simplified Wrapper and Interface Generator — Werkzeug um von anderen Sprachen aus auf in C geschriebene Module zuzugreifen (Seite 20)
TMS	Truth Maintenance System (Seite 15)
iSuite	Am Lehrstuhl Applied Knowledge Representation and Reasoning entwickeltes Informations- und Unterstützungssystem zur Behandlung von Schmerzpatienten [8] (Seite 1)

Literaturverzeichnis

[1] *XSB Logic Programming and Deductive Database system*. http://xsb.sourceforge.net/, 2009.

[2] Beierle, Christoph und Gabriele Kern-Isberner: *Methoden wissensbasierter Systeme. Grundlagen, Algorithmen, Anwendungen*. Nummer 978-3528157234. Vieweg, 2005.

[3] Drzymajllo, Beata: *Entwicklung und Implementierung eines wissensbasierten Modells für die Evaluation und Empfehlung medikamentöser Therapien*. Diplomarbeit, TU Dresden, 2009.

[4] Kifer, Michael, Georg Lausen und James Wu: *Logical Foundations of Object-Oriented and Frame-Based Languages*. Journal of the Association for Computing Machinery, May, 1995.

[5] Minsky, Marvin: *FRAMES — A Framework for Representing Knowledge*. http://web.media.mit.edu/~minsky/papers/Frames/frames.html, 1974, ISBN 55860-013-2.

[6] Noy, Natalya Fridman, Ray W. Fergerson und Mark A. Musen: *The knowledge model of Protégé-2000: combining interoperability and flexibility*. Stanford University.

[7] Peter Norvig, Stuart Russel und: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[8] Petersohn, Uwe: *Projektdetails zu iSuite*. http://www.inf.tu-dresden.de/content/institutes/ki/awv/Research/isuite.de.html, 2009.

[9] Petersohn, Uwe, Steffen Guhlemann, Lars Iwer und Rene Balzer: *Problem solving with Concept- and Case-based Reasoning*.

[10] Schatten, Markus: *Reasonable Python or how to Integrate F-Logic into an Object-Oriented Scripting Language*. Technischer Bericht, Faculty of Organization and Informatics University of Zagreb, 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.3965&rep=rep1&type=pdf.

[11] Schatten, Markus: *Programming Languages for Autopoiesis Facilitating Semantic Wiki Systems*. Dissertation, SVEUCILISTE U ZAGREBU FAKULTET ORGANIZACIJE I INFORMATIKE, 2009.

[12] Stumme, Prof. Dr. Gerd. `http://www.kde.cs.uni-kassel.de/lehre/ws2008-09/KI/folien/4_09_Frames.pdf`, 2008.

[13] Yang, Guizhen, Michael Kifer, Hui Wan und Chang Zhao: *Flora-2 Users Manual*. `http://flora.sourceforge.net/docs/floraManual.pdf`, 2008.

Danksagung

Ich danke meiner Familie, meinen Freunden Paul Mosler, Nico Hoffmann, Christina Kleinau, Neda Sultova sowie Frau Dr. Borcea-Pfitzmann und Herrn Prof. Petersohn, ohne deren moralische Unterstützung ich das Studium nicht erfolgreich abschließen hätte können.