TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Theoretische Informatik, Lehrstuhl für Automatentheorie

ENTWURF UND IMPLEMENTIERUNG EINES SYSTEMS ZUR ENTSCHEIDUNG VON SUBSUMPTION IN DER BESCHREIBUNGSLOGIK $\mathcal{FL}_0$

Friedrich Michel
Matrikelnummer: 4084573

BACHELOR-ARBEIT

zur Erlangung des akademischen Grades

BACHELOR OF SCIENCE (B.SC.)

Erstgutachter
Prof Dr. Franz Baader

Zweitgutachter
PD Dr. Anni-Yasmin Turhan

Betreuer
PD Dr. Anni-Yasmin Turhan

Eingereicht am: 15.08.2017

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Entwurf und Implementierung eines Systems zur Entscheidung von Subsumption in der Beschreibungslogik* $\mathcal{FL}_0$ selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 15.08.2017

Friedrich Michel

In dieser Fassung wurden am 12.11.2017 orthografische Korrekturen vorgenommen.

INHALTSVERZEICHNIS

1 EINLEITUNG UND MOTIVATION

Strukturierte Wissensrepräsentation war schon immer eine wichtige Aufgabe in der Informatik. Neben der bloßen Speicherung ist hier die Extraktion von impliziten Informationen, wie es mittels formaler Logiken möglich ist, von großem Interesse. Eine eine solche Wissensrepräsentation bietet die Prädikatenlogik erster Stufe. Leider fällt das Extrahieren vieler impliziter Informationen in dieser in die Klasse der unentscheidbaren Probleme. Eine weitere Möglichkeit bieten Beschreibungslogiken (BLs). Die meisten BLs bilden entscheidbare Fragmente der Prädikatenlogik erster Stufe und sind so einerseits relativ ausdrucksstark, während Probleme der Wissensextraktion entscheidbar werden. Sie erlauben dem Nutzer Sachverhalte der echten Welt durch Konzepte und Rollen zu modellieren. Konzepte modellieren dabei eine Klasse von Objekten, z.B. *Römer* und *Gallier*, während Rollen Beziehungen zwischen Objekten darstellen, wie z.B. *unterdrückt*. Konstruktoren erlauben aus Rollen und (einfachen) Konzepten komplexe Konzepte zu modellieren. Die Konstruktoren Konjunktion (⊓) und Disjunktion (⊔) erlauben ein Objekt mehreren Konzepten zuzuordnen.

$$Gallier \sqcap (Klein \sqcup Dick)$$

beschreibt alle Gallier die klein oder dick sind. Über existenzielle Restriktionen (∃) und Werterestriktionen (∀) lassen sich Konzepte durch Rollen in Relationen setzen.

$$Gallier \sqcap \exists besitzt.Hund \sqcap \forall bekämpft.Römer$$

beschreibt alle Gallier die einen Hund besitzen und niemanden bekämpfen, außer Römer. Über Negation (¬) lässt sich zum Beispiel darstellen,

$$\neg(\exists trinkt.Zaubertrank)$$

dass eine Person keinen Zaubertrank trinkt. Die auf diese Weise beschriebenen komplexen Konzepte können durch Axiome in Relation gesetzt werden. Das Unterklassenaxiom (⊑) sagt aus, dass alle Objekte eines bestimmten Konzepts, auch ein anderes Konzept erfüllen.

$$Römer \sqsubseteq \forall fürchtet.(Gallier \sqcap \exists lebt\_in.(Dorf \sqcap Klein))$$

sagt aus, dass das einzige was Römer fürchten Gallier sind, die in einem kleinen Dorf leben.

Der Unterschied zwischen einzelnen BLs besteht vor allem darin, welche Konstruktoren sie erlauben. $\mathcal{ALC}$, als bekannteste BL, enthält die fünf bisher vorgestellten Konstruktoren ($\sqcap$, $\sqcup$, $\neg$, $\exists$, $\forall$). Eines der wichtigsten Entscheidungsprobleme in BLs ist die Subsumption. Sie beantwortet für zwei gegebene Konzepte die Frage, ob alle Objekte des ersten Konzeptes auch immer dem zweiten Konzept angehören. Diese Frage stellt sich in $\mathcal{ALC}$ als EXPTIME schweres Problem heraus. Aus diesem Grund lohnt es sich weniger ausdrucksstarke BLs in der Hoffnung zu betrachten, dass in diesen Subsumption ein leichteres Problem darstellt. Die Beschreibungslogik $\mathcal{EL}$ beschränkt sich auf existentielle Restriktionen und Konjunktionen wodurch Subsumption in polynomieller Zeit entscheidbar wird. Dies hat in der Vergangenheit dazu geführt, dass für $\mathcal{EL}$ spezielle Reasoner umgesetzt wurden, welche schneller als allgemeine $\mathcal{ALC}$ Reasoner sein konnten. Aus diesem Grund entstanden, Ontologien welche sich bewusst auf die genannten Konstruktoren beschränkten. Eine zu $\mathcal{EL}$ ähnliche BL stellt $\mathcal{FL}_0$ dar. Diese ist auf Werterestriktionen und Konjunktionen beschränkt. Im Gegensatz zu $\mathcal{EL}$ bleibt in $\mathcal{FL}_0$ die Subsumption ein EXPTIME vollständiges Problem. Somit hat $\mathcal{FL}_0$, trotz geringer Ausdrucksstärke, die gleiche Komplexität in der Entscheidung von Subsumption wie $\mathcal{ALC}$. Aus diesem Grund wird Subsumption in $\mathcal{FL}_0$ bisher von Standard $\mathcal{ALC}$ Reasonern entschieden. In [1] wurde nun ein struktureller Ansatz zur Entscheidung von Subsumption vorgeschlagen. Dieser weckt die Hoffnung Subsumption in $\mathcal{FL}_0$, trotz gleicher theoretischer Komplexität, in praktischen Anwendungsfällen effizienter zu entscheiden, als bekannte $\mathcal{ALC}$ Reasoner dies ermöglichen würden.

Das Ziel dieser Arbeit ist es diese Hypothese zu überprüfen, indem der strukturelle Algorithmus nach einer Konkretisierung und Implementierung mit existierenden $\mathcal{ALC}$ Reasonern verglichen wird. Hierfür werden in Kapitel 2 zunächst die Grundlagen von BLs und insbesondere $\mathcal{FL}_0$ eingeführt. In Kapitel 3 wird der Algorithmus aus [1] genauer vorgestellt. Kapitel 4 wird sich mit der Konkretisierung und Implementierung des Algorithmus beschäftigen. In Kapitel 5 werden wir anhand einer empirische Auswertung die Hypothese überprüfen. Anschließend folgt in Kapitel 6 ein kurzer Ausblick auf mögliche anschließende Arbeiten.

2 GRUNDLAGEN

Im folgenden Kapitel wird die Beschreibungslogik $\mathcal{FL}_0$ formal eingeführt. Die Ausführungen sind aus der Literatur entnommen und stimmen inhaltlich und in den verwendeten Begriffen insbesondere mit [2] und [1] überein.

Sei N_C eine Menge von Konzeptnamen, N_R eine Menge von Rollennamen und $N_C \cap N_R = \emptyset$. Im folgenden nutzen wir A, $B \in N_C$ für Konzeptnamen, $r \in N_R$ für Rollennamen und C, D für komplexe Konzepte. Wir definieren Konzepte rekursiv. Wenn $A \in N_C$, $r \in N_R$ und C, D Konzepte sind, dann auch:

- A (Konzeptname)
- $\top$ (Top Konzept)
- $C \sqcap D$ (Konjunktion)
- $\forall r.C$ (Werterestriktion)

Ketten von Werterestriktionen $\forall r_1.\forall r_2 \ldots \forall r_n.C$ kürzen wir im Folgenden durch $\forall w.C$ ab, wobei $w = r_1 r_2 \ldots r_n$. Die Werterestriktion $\forall \epsilon.C$ entspricht hierbei C.

Um Wissen in BLs zu repräsentieren setzen wir (komplexe) Konzepte durch generelle Konzept Inklusionen (GCIs) in Relation. Eine GCI ist von der Form $C \sqsubseteq D$, wobei C und D Konzepte sind. Sie sagt aus, dass die durch das Konzept C repräsentierten Objekte auch durch D repräsentiert werden. Eine TBox $\mathcal{T}$ ist eine Menge von GCIs und bildet so unsere Wissensbasis (Ontologie). Im Allgemeinen könnten Ontologien weitere Formen von Axiomen enthalten. In dieser Arbeit beschränken wir uns jedoch auf GCIs.

Um die Semantik von $\mathcal{FL}_0$ Konzepten zu definieren führen wir Interpretationen ein. Eine Interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ besteht aus einer nicht leeren Menge $\Delta^{\mathcal{I}}$ (Domäne) und einer Funktion $\cdot^{\mathcal{I}}$, die jedem Konzeptnamen A eine Menge $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ und jeder Rolle eine binäre Relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ zuweist. Diese Funktion $\cdot^{\mathcal{I}}$ wird induktiv für komplexe Konzepte erweitert:

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$

- $(\forall r.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.\ (a,b) \in r^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$

Eine Interpretation $\mathcal{I}$ erfüllt eine GCI $C \sqsubseteq D$ gdw. $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Wenn $\mathcal{I}$ alle GCIs einer TBox $\mathcal{T}$ erfüllt, wird $\mathcal{I}$ ein *Modell* von $\mathcal{T}$ genannt.

Das Entscheidungsproblem, welches der in dieser Arbeit behandelte Algorithmus bearbeitet, ist Subsumption.

Definition 2.1 (Subsumption)**.** Sei $\mathcal{T}$ eine TBox und C und D Konzepte. C wird von D bezüglich $\mathcal{T}$ *subsumiert* ($C \sqsubseteq_{\mathcal{T}} D$) gdw. $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ für alle Modelle $\mathcal{I}$ von $\mathcal{T}$.

Mit diesen Definitionen sind die Grundlagen von $\mathcal{FL}_0$ gesetzt, sodass wir in den folgenden Kapiteln die Entscheidung von Subsumptionsrelationen im Bezug auf generelle TBoxen in $\mathcal{FL}_0$ betrachten können.

3 THEORETISCHER ALGORITHMUS

Nachdem im vorherigen Kapitel die Grundlagen von $\mathcal{FL}_0$ eingeführt wurden, betrachten wir nun den in [1] vorgestellten Algorithmus und die zugehörigen Grundlagen. Die Ausführungen stimmen inhaltlich mit den Betrachtungen in [1] überein.

3.1 FUNKTIONALE MODELLE

Definition 3.1. [1, Definition 3.4] $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ ist eine *funktionale Interpretation* gdw.

- $\Delta^{\mathcal{I}} = N_R^*$ (Struktur)
- für alle $r \in N_R$. $(u, v) \in r^{\mathcal{I}}$ gdw. $v = ur$ (Struktur)

Eine *funktionale Interpretation* $\mathcal{I}$ wird *funktionales Modell* eines Konzeptes C bezüglich einer TBox $\mathcal{T}$ genannt gdw.

- für alle $D \sqsubseteq E \in \mathcal{T}$. $D^{\mathcal{I}} \subseteq E^{\mathcal{I}}$ (Modell von $\mathcal{T}$)
- $\epsilon \in C^{\mathcal{I}}$ (Wurzel ist im Konzept C)

Funktionale Interpretationen können aufgrund ihrer Struktur als $|N_R|$-fach verzweigende Bäume betrachtet werden. Hierbei tragen Domänenelemente $w \in \Delta^{\mathcal{I}}$ die Menge aller Konzeptnamen $A \in N_C$ als Beschriftung, für die gilt, dass $w \in A^{\mathcal{I}}$. Ein funktionales Modell von A bezüglich der TBox $\mathcal{T} = \{A \sqsubseteq \forall r.B, B \sqsubseteq \forall s.A\}$ könnte wie folgt aussehen:

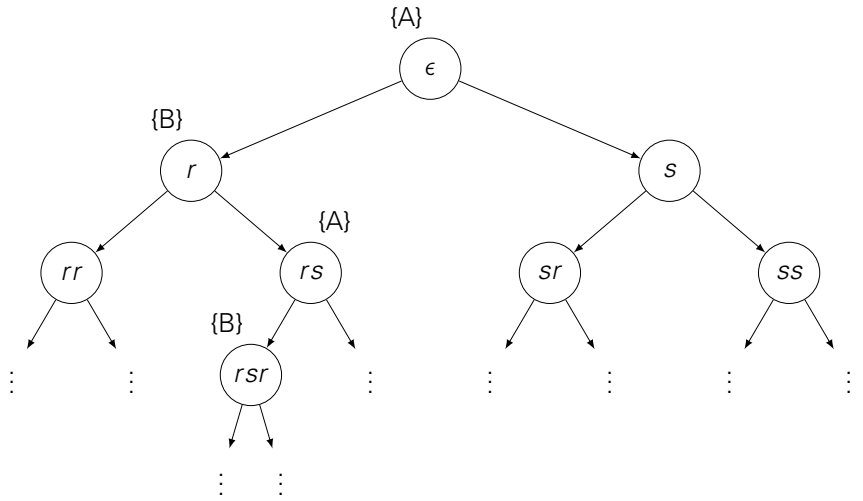

Abbildung 3.1: Funktionales Modell

Definition 3.2. [1, Definition 3.5] Für funktionale Modelle $\mathcal{I}, \mathcal{J}$ eines Konzeptes C bezüglich einer TBox $\mathcal{T}$ mit der selben Domäne N_R^* definieren wir

$$\mathcal{I} \subseteq \mathcal{J} \text{ gdw. für alle } A \in N_C.\ A^{\mathcal{I}} \subseteq A^{\mathcal{J}}.$$

In [1, S. 13] wird gezeigt, dass es ein *kleinstes funktionales Modell* $\mathcal{I}_{C,\mathcal{T}}$ gibt. Das heißt, dass $\mathcal{I}_{C,\mathcal{T}}$ ein funktionales Modell von C bezüglich $\mathcal{T}$ ist und für alle funktionalen Modelle $\mathcal{J}$ von C bezüglich $\mathcal{T}$ gilt $\mathcal{I}_{C,\mathcal{T}} \subseteq \mathcal{J}$.

Der folgende Algorithmus berechnet das *kleinste funktionale Modell* und ließt aus diesem Subsumptionsrelationen ab.

3.2 NORMALISIERUNG

Da der Algorithmus auf einer normalisierten TBox arbeitet, betrachten wir hier die Normalisierung von $\mathcal{FL}_0$ TBoxen.

Ein $\mathcal{FL}_0$ Konzept ist in *concept-conjunction Normalform* (CCNF), gdw. es von der Form

$$\forall w_1.A_1 \sqcap \forall w_2.A_2 \sqcap \ldots \sqcap \forall w_n.A_n$$

mit $A_i \in N_C$ und $w_i \in N_R^*$ $(1 \leq i \leq n)$ ist. Für $n = 0$ entspricht dieser Ausdruck $\top$. Eine GCI ist in CCNF, gdw. die linke und rechte Seite in CCNF sind. Eine $\mathcal{FL}_0$ TBox $\mathcal{T}$ ist in CCNF, gdw. alle GCIs in ihr in CCNF sind.

Eine $\mathcal{FL}_0$ TBox $\mathcal{T}$ wird durch erschöpfendes Anwenden der Regeln aus Tabelle 3.1 in CCNF umgeformt.

Tabelle 3.1: Normalisierung in CCNF

NF1.1	$C \sqcap \top$	$\rightsquigarrow$	C
NF1.2	$\top \sqcap C$	$\rightsquigarrow$	C
NF1.3	$\forall w.\top$	$\rightsquigarrow$	$\top$
NF1.4	$\forall w.(C_1 \sqcap \ldots \sqcap C_n)$	$\rightsquigarrow$	$\forall w.C_1 \sqcap \ldots \sqcap \forall w.C_n$

Eine $\mathcal{FL}_0$ TBox $\mathcal{T}$ ist in *plane-axiom Normalform* (PANF), gdw. sie in CCNF ist und für jede Werterestriktion $\forall w.A$ in $\mathcal{T}$ gilt, dass $|w| \leq 1$.

Eine $\mathcal{FL}_0$ TBox $\mathcal{T}$ in CCNF wird durch erschöpfendes Anwenden der Regeln aus Tabelle 3.2 in PANF umgeformt.

Tabelle 3.2: Normalisierung in PANF

NF2.1	$C_1 \sqcap \forall r w.A \sqcap C_2 \sqsubseteq D$	$\rightsquigarrow$	$C_1 \sqcap \forall r.B \sqcap C_2 \sqsubseteq D,\ \forall w.A \sqsubseteq B$
NF2.2	$D \sqsubseteq C_1 \sqcap \forall r w.A \sqcap C_2$	$\rightsquigarrow$	$D \sqsubseteq C_1 \sqcap \forall r.B \sqcap C_2,\ B \sqsubseteq \forall w.A$

Die Normalisierung besteht somit aus zwei Phasen. In der ersten Phase wird eine TBox durch Anwenden der Regeln NF1.1 bis NF1.4 in CCNF umgeformt und in der zweiten Phase wird die resultierende TBox durch die Regeln NF2.1 und NF2.2 in PANF normalisiert.

In [1, S. 17ff] wird gezeigt, dass für eine TBox $\mathcal{T}$ und die durch Normalisierung von $\mathcal{T}$ erhaltene TBox $\mathcal{T}'$ in PANF gilt, dass

$$C \sqsubseteq_{\mathcal{T}} D \text{ gdw. } C \sqsubseteq_{\mathcal{T}'} D$$

für alle Konzepte C und D aus der ursprünglichen TBox $\mathcal{T}$. Subsumption lässt sich also entscheiden indem man eine TBox in PANF normalisiert und die Subsumption für die normalisierte TBox entscheidet. Weiterhin braucht die hier beschriebene Normalisierung nur polynomielle Zeit, sodass auch dies für ein EXPTIME schweres Problem keine Einschränkungen ergibt.

3.3 SUBSUMPTION BZGL. GENERELLER $\mathcal{FL}_0$ TBOXEN

Der in [1] vorgeschlagene Algorithmus geht von einer $\mathcal{FL}_0$ TBox in PANF aus und entscheidet Subsumption zwischen zwei Konzeptnamen. Dies ermöglicht indirekt auch Subsumption zwischen komplexen Konzepten C und D zu entscheiden, indem man die TBox $\mathcal{T}$ auf $\mathcal{T}' = \mathcal{T} \cup \{A_C \sqsubseteq C, D \sqsubseteq A_D\}$ mit neuen Konzeptnamen A_C, A_D erweitert. Man sieht leicht, dass $C \sqsubseteq_{\mathcal{T}} D$ gdw. $A_C \sqsubseteq_{\mathcal{T}'} A_D$.

Für den Algorithmus benötigen wir einige Begriffe. Um auf die Konzepte, in denen ein Domänenelement enthalten ist verweisen zu können definieren wir

$$\mathcal{I}(w) = \{A \in N_C \mid w \in A^{\mathcal{I}}\}.$$

Definition 3.3. [1, Definition 7.1] Eine *head*-Struktur ist ein n+1 Tupel aus Mengen $(L, L_1, \ldots, L_n)$ mit $|N_R| = n$, wobei $L \subseteq N_C$ die Wurzel und $L_i \subseteq N_C$ $(1 \leq i \leq n)$ die Kinder der *head*-Struktur sind. Die Funktion *hd* bildet Knoten $w \in N_R^* = \Delta^{\mathcal{I}}$ einer funktionalen Interpretation $\mathcal{I}$ auf die *head*-Struktur $hd(w) = (\mathcal{I}(w), \mathcal{I}(wr_1), \ldots, \mathcal{I}(wr_n))$ ab.

Die Relation $\subseteq$ und die Vereinigung $\cup$ werden auf *head*-Strukturen erweitert.
Für *head*-Strukturen $h_1 = (L, L_1, \ldots, L_n)$ und $h_1 = (M, M_1, \ldots, M_n)$ gilt

$$h_1 \subseteq h_2 \text{ gdw. } L \subseteq M \wedge \bigwedge_{i=1}^{n} L_i \subseteq M_i$$

und

$$h_1 \cup h_2 = (L \cup M, L_1 \cup M_1, \ldots, L_n \cup M_n).$$

Für ein Konzept C in PANF mit $C = \forall w_1.A_1 \sqcap \ldots \sqcap \forall w_m.A_m$ und $A_i \in N_C$, $w_i \in (N_R \cup \{\epsilon\})$ $(1 \leq i \leq m)$ sei

$$\widehat{C} = \{\forall w_i.A_i \mid 1 \leq i \leq m\}$$

Wir definieren die Funktion *hd* auch für Konzepte

$$hd(C) = (\widehat{C} \cap N_C, \{A \in N_C \mid \forall r_1.A \in \widehat{C}\}, \ldots, \{A \in N_C \mid \forall r_n.A \in \widehat{C}\}).$$

Der Algorithmus beginnt iterativ eine funktionale Interpretation aufzubauen um zu einem bestimmten Zeitpunkt stoppen. Somit ist die Interpretation, im Gegensatz zu einer vollständigen funktionalen Interpretation, zu jedem Zeitpunkt endlich und wird deswegen *funktionaler Interpretationsstumpf* $\mathcal{I}^{\Delta}$ genannt. Um $\mathcal{I}^{\Delta}$ zu verändern, nutzen wir die Abkürzung $hd(w) := (L, L_1, \ldots, L_n)$, welche w zu allen $A \in L$ und wr_i zu allen $A \in L_i$ $(1 \leq i \leq n)$ hinzufügt. Falls $wr_i \notin \Delta^{\mathcal{I}^{\Delta}}$, wird zunächst wr_i in die Domäne eingefügt $\Delta^{\mathcal{I}^{\Delta}} := \Delta^{\mathcal{I}^{\Delta}} \cup \{wr_i\}$.

Definition 3.4. [1, Definition 7.2] Seien $u = r_{i_1}, \ldots, r_{i_k}$ und $v = r_{j_1}, \ldots, r_{j_l}$ zwei Domänenelemente mit $i_1, \ldots i_k, j_1, \ldots j_l \in \{1, \ldots, n\}$ und $N_R = \{r_1, \ldots, r_n\}$. Die Relation $\prec$ ist eine strenge Totalordnung der Domänenelemente, sodass $(u, v) \in \prec$ gdw. $|u| < |v|$ oder $k = l$ und $(i_1, \ldots, i_k) \prec_{\mathbb{N}} (j_1, \ldots, j_l)$, wobei $\prec_{\mathbb{N}}$ die lexikographische Ordnung von Tupeln natürlicher Zahlen ist.

Damit der Algorithmus terminiert, benötigen wir Bedingungen, welche verhindern, dass der funktionale Interpretationsstumpf $\mathcal{I}^{\Delta}$ unendlich lange erweitert wird. Hierfür verwenden wir „Anywhere Blocking".

Definition 3.5. [1, Definition 7.3] Ein Domänenelement $w \in \Delta^{\mathcal{I}^\Delta}$ gilt als *blockiert*, gdw.

$$\text{es gibt } u, u' \in \Delta^{\mathcal{I}^\Delta}, v \in N_R^*.\ u \prec u' \wedge w = u'v \wedge \mathcal{I}^\Delta(u) = \mathcal{I}^\Delta(u') \wedge \text{ u ist nicht blockiert.}$$

Nachdem alle nötigen Begriffe eingeführt wurden, wird an dieser Stelle der in [1] vorgeschlagene Algorithmus *SUBS* vorgestellt, welcher Subsumption zwischen zwei Konzepten A und B bezüglich einer TBox $\mathcal{T}$ entscheidet.

Algorithm 1 SUBS [1, Definition 7.4]

1: **procedure** SUBS($A, B, \mathcal{T}$) ▷ $A, B \in N_C, \mathcal{FL}_0$ TBox $\mathcal{T}$
2: $\Delta^{\mathcal{I}^\Delta} := \{\epsilon\}$
3: $A^{\mathcal{I}^\Delta} := \{\epsilon\}$
4: Wähle das kleinste $w \in \Delta^{\mathcal{I}^\Delta}$ bezüglich $\prec$ sodass:
- w nicht blockiert ist und
- es $E \sqsubseteq F \in \mathcal{T}$ gibt mit $hd(E) \subseteq hd(w)$ und $hd(F) \nsubseteq hd(w)$

5: **if** ein solches w existiert **then**
6: $hd(w) := hd(w) \cup hd(F)$
7: **goto** 4
8: **else**
9: **return** $A \sqsubseteq_{\mathcal{T}} B$ gdw. $B \in \mathcal{I}^\Delta(\epsilon)$
10: **end if**
11: **end procedure**

Die Funktionsweise von *SUBS* erinnert an regelbasierte Systeme, da aus $hd(E) \subseteq hd(w)$ folgt, dass $hd(w) := hd(w) \cup hd(F)$. Falls $w \in E^{\mathcal{I}^\Delta}$, wird also $\mathcal{I}^\Delta$ so verändert, dass $w \in F^{\mathcal{I}^\Delta}$ gilt. Dies kann als die Regel $hd(E) \rightarrow hd(F)$ verstanden werden. Aus diesem Grund werden wir im folgenden vom Anwenden von GCIs und Anwenden von Regeln sprechen und diesen Teil des Algorithmus meinen.

Nachdem in diesem Kapitel der Algorithmus *SUBS* eingeführt wurde, wird sich diese Arbeit im folgenden damit beschäftigen, wie sich dieser konkret implementieren lässt. Die Schritte 4 bis 7 bilden die Hauptschleife des Programms, welche potentiell sehr häufig ausgeführt wird und somit der meisten Optimierungen bedarf. Da die Schritte 5 bis 7 quasi direkt ausführbare Operationen darstellen, werden wir uns vor allem auf Schritt 4 konzentrieren, also auf das Finden von anwendbaren Regeln und das Entscheiden, ob ein Element blockiert ist.

4 ENTWURFSENTSCHEIDUNGEN

Der zuvor vorgestellte Algorithmus gibt einerseits einen sehr konkreten Rahmen zur Implementierung vor, aber lässt aber die Umsetzung einiger Schritte offen. In diesem Kapitel werden wir zunächst verschiedene Datenstrukturen betrachten, um Bestandteile des Algorithmus technisch zu repräsentieren. Hiernach werden wir uns mit der Normalisierung beschäftigen, welche im theoretischen Algorithmus durch Regeln beschrieben ist, welche nacheinander ausgeführt werden, bis keine von ihnen mehr anwendbar ist. Im Konkreten muss entschieden werden, wie anwendbare Normalisierungsregeln gefunden werden können. Im eigentlichen Algorithmus ist vor allem das Finden des kleinsten Elements, welches nicht blockiert ist, und auf welches eine GCI angewandt werden kann, eine zu lösende Aufgabe. Nachdem wir die Bausteine des Algorithmus einzeln betrachtet haben, fügen wir diese schließlich zu einem gesamten, Subsumption entscheidenden Algorithmus zusammen.

4.1 TECHNISCHE REPRÄSENTATION

4.1.1 KONZEPTE UND ROLLEN

Um Konzeptnamen einerseits speicherarm zu verwalten und diese außerdem in eine eindeutige Reihenfolge zu bringen, repräsentieren wir diese durch natürliche Zahlen. Das gleiche gilt für Rollennamen. Hierfür nutzen wir die bijektiven Funktionen $f_{N_C} : N_C \mapsto [1, |N_C|]$ und $f_{N_R} : N_R \mapsto [1, |N_R|]$. Zur leichteren Handhabung sei im folgenden $f_{N_C}(A_i) = i$, $f_{N_C}(B_i) = i$ und $f_{N_R}(r_i) = i$.

4.1.2 FUNKTIONALER INTERPRETATIONSSTUMPF

Wie bereits in Abschnitt 3.1 beschrieben, können funktionale Interpretationen als $|N_R|$-fach verzweigende Bäume betrachtet werden, wobei die Domänenenelemente $w \in \Delta^{\mathcal{I}^\Delta}$ die Beschriftung $\mathcal{I}^\Delta(w)$ tragen. Für den Algorithmus ist es sehr relevant, effizienten Zugriff auf $\mathcal{I}^\Delta(w)$ zu haben (Bestimmung und Veränderung von $hd(w)$) und irrelevant, $A^{\mathcal{I}^\Delta}$ zu bestimmen. Aus diesem Grund eignet sich die Baumdarstellung der funktionalen Interpretation

besser, als die der Definition von Interpretationen entsprechende Darstellung, welche jedem Konzept eine Menge von Domänenelementen zuweist. In dieser Baumdarstellung wird also jedem Domänenelement w die Konzeptmenge $\mathcal{I}^{\Delta}(w)$, also eine Menge natürlicher Zahlen, zugewiesen.

Der Interpretationsbaum ist wie bereits beschrieben $|N_R|$-fach verzweigend. Die Menge N_R ist jedoch unendlich abzählbar groß. Für das kleinste funktionale Modell sind jedoch nur die Rollen relevant, welche auch in der TBox vorkommen. Um auf die Menge der, in einer TBox $\mathcal{T}$ tatsächlich verwendeten, Konzepte und Rollen verweisen zu können definieren wir $N_C^{\mathcal{T}}$ und $N_R^{\mathcal{T}}$. Es sei $N_C^{\mathcal{T}} \subset N_C$ die Menge aller Konzeptnamen und $N_R^{\mathcal{T}} \subset N_R$ die Menge aller Rollennamen, die in $\mathcal{T}$ vorkommen. Der Baum, welchen wir implementieren wird somit ein $|N_R^{\mathcal{T}}|$-fach verzweigender.

In der Implementierung gibt es für die Baumdarstellung verschiedene Möglichkeiten. Die Knoten des Baumes können durch einzelne Objekte repräsentiert werden, welche neben einer Beschriftung eine Menge von Kindknoten enthalten. Diese Darstellung hat den Vorteil, dass leicht auf Kindknoten zugegriffen werden kann, aber den Nachteil, dass für den Zugriff, auf ein weit unten im Baum liegendes Element, durch den Baum traversiert werden muss.

Eine andere Möglichkeit besteht darin, den Baum indirekt zu repräsentieren. Das heißt, dass die einzelnen Knoten in einer Liste stehen und implizit feststeht, welche Knoten die entsprechenden Kindknoten sind. Hierfür können die Knoten entsprechend der Relation $\prec$ durchnummeriert werden (siehe Abbildung 4.1).

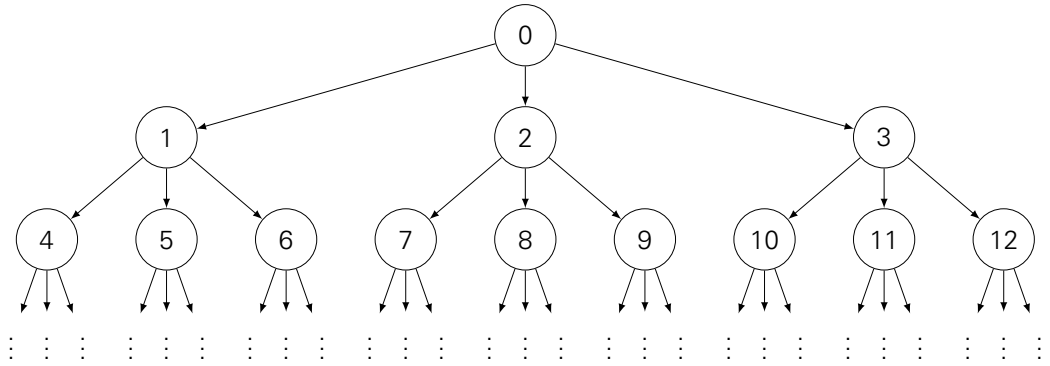

Abbildung 4.1: Nummerierte Darstellung funktionale Interpretation

Der r_i Nachfolger eines Knotens der Nummer x lässt sich durch $|N_R^{\mathcal{T}}| * x + i$ berechnen und der direkte Vorgänger durch $\lfloor \frac{(x-1)}{|N_R^{\mathcal{T}}|} \rfloor$. Schwerer zu überprüfen ist, ob ein Knoten ein nicht direkter Vorgänger ist.

Um ein Domänenelement mit der Zahl i zu verknüpfen, könnte das Domänenenelement in der Implementierung in einem Array an Stelle i abgelegt werden. Falls die vom Algorithmus konstruierte Interpretation jedoch nur die Elemente entlang eines Pfades der Länge n enthält, müsste das Array groß genug für einen Baum der Tiefe n sein und würde somit aus $|N_R^{\mathcal{T}}|^n$ Speicherstellen bestehen. Um dieses Problem zu umgehen nutzen wir in der Implementierung

in Java eine SortedMap (TreeMap). Diese erlaubt das Einfügen von Elementen und den Zugriff auf Elemente in $\mathcal{O}(\log n)$ [3]. Auf diese Weise wird für leere Domänenelemente kein Speicher verbraucht.

Durch diese Implementierung kann also einerseits effektiv auf die einzelnen Knoten zugegriffen werden und andererseits können durch die Wahl der SortedMap die Knoten auch in aufsteigender Reihenfolge bezüglich $\prec$ durchlaufen werden.

Um eine effiziente Überprüfung nicht direkter Vorgänger, welche für das Überprüfen von *blockierten* Knoten relevant ist, zu ermöglichen, betrachten wir eine weite Zuordnung von natürlichen Zahlen auf Domänenelemente. Hierfür wird jedem Knoten $w = r_{i_1} \dots r_{i_k}$ die Zahl $i_1 \dots i_k$ in der Basis $|N_R^{\mathcal{T}}| + 1$ zugeordnet.

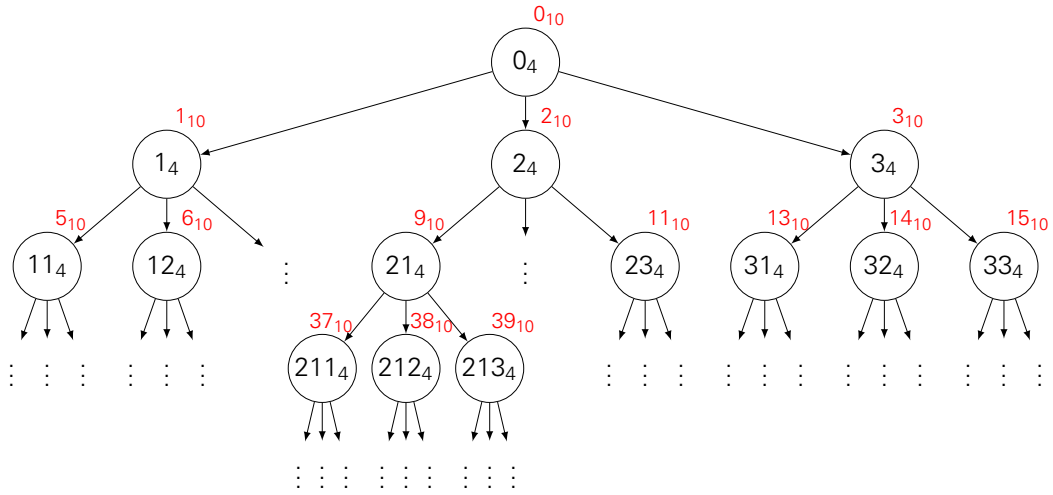

Abbildung 4.2: Basis $|N_R^{\mathcal{T}}| + 1$ Darstellung funktionaler Interpretationen

Der r_i Nachfolger eines Knotens der Nummer x lässt sich durch $(|N_R^{\mathcal{T}}| + 1) * x + i$ berechnen und der direkte Vorgänger durch $\lfloor \frac{x}{|N_R^{\mathcal{T}}|+1} \rfloor$. Ob ein Knoten x ein indirekter Nachfolger eines Knoten y ist, lässt sich somit herausfinden indem man

$$y = \left\lfloor \frac{x}{(|N_R^{\mathcal{T}}| + 1)^k} \right\rfloor$$

überprüft, wobei k die Differenz der Tiefen der Knoten ist. Die Tiefe eines Knotens x lässt sich durch $\log_{|N_R^{\mathcal{T}}|+1} x + 1$ bestimmen. Für k ergibt sich somit ein Wert von

$$k = (\log_{|N_R^{\mathcal{T}}|+1} y) - (\log_{|N_R^{\mathcal{T}}|+1} x).$$

Die Darstellung zur Basis $|N_R^{\mathcal{T}}| + 1$ behält somit alle Vorteile der durchnummerierten Variante und ermöglicht zusätzlich effizientes Überprüfen von Nachfolgerrelationen. Der Nachteil ist, dass dadurch, dass nicht alle Zahlen genutzt werden (Ziffer 0 wird nicht verwendet), die Werte

in tieferen Ebenen schneller wachsen (Faktor $|N_R^{\mathcal{T}}|+1$ statt $|N_R^{\mathcal{T}}|$). Dieser Nachteil wiegt jedoch nicht die Vorteile auf, sodass in der Implementierung zur Repräsentation von funktionalen Modellen eine *TreeMap* verwendet wird und die Domänenelemente durch natürliche Zahlen der Basis $|N_R^{\mathcal{T}}|+1$ identifiziert werden.

Da in einer Ontologie mit $|N_R^{\mathcal{T}}| = 15$ bereits nach 16 Ebenen die zugeordneten Zahlen die Größe des maximalen primitiven Datentyps „long“ (maximal 2^{64} [4]) erreichen, verwenden wir zur Zahlenrepräsentation die Java Klasse *BigInteger*, welche unbegrenzte Größen erlaubt.

4.2 NORMALISIERUNG

Der Algorithmus *SUBS* geht von einer normalisierten $\mathcal{FL}_0$ TBox in PANF aus, Ontologien liegen im Allgemeinen jedoch nicht normalisiert vor. Aus diesem Grund muss nach dem Einlesen einer Ontologie zunächst die Normalisierung vorgenommen werden. Wie bereits in Abschnitt 3.2 erwähnt, benötigt die in dieser Arbeit vorgestellte Normalisierung nur polynomiell viel Zeit. Außerdem muss eine Ontologie nur einmalig normalisiert werden und nicht für jede Subsumptionsanfrage einzeln. Da Subsumption bezüglich genereller TBoxen in $\mathcal{FL}_0$ EXPTIME schwer ist, fällt die Dauer der Normalisierung für genügend schwere Probleme nicht ins Gewicht. Aus diesem Grund lag das Hauptaugenmerk in der Implementierung nicht im Bereich der Normalisierung. Da sie jedoch in jedem Fall nötig ist, betrachten wir in diesem Abschnitt einige der getroffenen Entwurfsentscheidungen.

Die Regeln zur Normalisierung in CCNF arbeiten nur auf Konzepten. Um eine TBox $\mathcal{T}$ zu normalisieren, müssen für alle GCIs in $\mathcal{T}$ jeweils die linke und rechte Seite normalisiert werden. Im folgenden Betrachten wir nur die Normalisierung von Konzepten. Da die Normalisierung auf der syntaktischen Struktur von (komplexen) Konzepten arbeitet, bietet es sich an auf dem Syntaxbaum, welcher sich durch die rekursive Definition von Konzepten ergibt, zu arbeiten (vergleiche [1, S. 21]). Wir betrachten bei Ketten von Werterestriktionen jeweils die maximale Kette. Das Konzept $C = \forall r.(A \sqcap \forall sr.(\top \sqcap B) \sqcap \forall rrs.\top)$ wird zum Beispiel wie folgt repräsentiert:

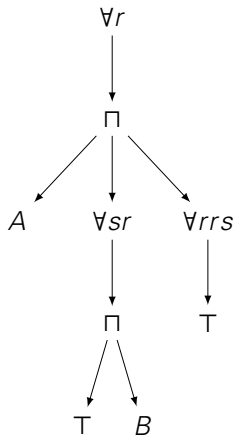

Abbildung 4.3: Syntaxbaum

Da $\top$ Konzepte keine Nachfolger haben können, sind diese stets Blattknoten. Dies erlaubt für die Normalisierungsregeln NF1.1 bis NF1.3 (siehe Tabelle 3.1) einen „Bottom up“ Ansatz zur Implementierung. Dieser lässt sich durch einen einzigen rekursiven Aufruf über die Baumstruktur realisieren. In diesem wendet jeder Knoten den rekursiven Aufruf zunächst auf seine Kindknoten an und wendet anschließend die Regeln NF1.1 bis NF1.3 auf sich selbst an. Nach diesem rekursiven Aufruf auf ein Konzept C gibt es entweder kein $\top$ mehr in C, oder $C = \top$.

Die Regel NF1.4 (siehe Tabelle 3.1) würde sich auch durch einen einzigen rekursiven Aufruf implementieren lassen. Da NF1.4 mit polynomiellen Wachstum an Speicherverbrauch verbunden ist, haben wir uns jedoch für einen „Top Down“ Ansatz entschieden, welcher die folgenden Punkte in gegebener Reihenfolge befolgt.

1. Falls NF1.4 sich auf den Wurzelknoten anwenden lässt, wende diese an. (Nach diesem Schritt ist der Wurzelknoten definitiv eine Konjunktion, ein Konzeptname oder $\top$)
2. Falls der Wurzelknoten ein Konzeptname oder $\top$ ist, terminiere.
3. Wähle aus den Kindknoten der Wurzel (Konjunktion) eine Werterestriktion $D = \forall w.C$, wobei C eine Konjunktion ist.
4. Wenn kein solches D existiert, terminiere.
5. Wende NF1.4 auf D an und fahre mit Punkt 3 fort.

Nach dem Anwenden von NF1.1 bis NF1.4 auf beide Seiten aller GCIs befindet sich die TBox in CCNF. Für die Normalisierung in PANF stellen wir folgende Beobachtung an. Wendet man NF2.1 auf die GCI

$$C_1 \sqcap \forall r_1 r_2 r_3 . A \sqcap C_2 \sqsubseteq D$$

einmal an, entstehen die folgenden GCIs:

$$C_1 \sqcap \forall r_1 . B_1 \sqcap C_2 \sqsubseteq D \text{ und } \forall r_2 r_3 . A \sqsubseteq B_1$$

mit

dem neuen Konzept B_1. NF2.1 ist hier erneut anwendbar, sodass insgesamt

$$C_1 \sqcap \forall r_1 . B_1 \sqcap C_2 \sqsubseteq D \text{ und } \forall r_2 . B_2 \sqsubseteq B_1 \text{ und } \forall r_3 . A \sqsubseteq B_2$$

mit dem neuen Konzept B_2 entsteht. Dieser Sachverhalt lässt sich abkürzen durch

$$\{C_1 \sqcap \forall r_1 \ldots r_n.A \sqcap C_2 \sqsubseteq D\}$$

$$\rightsquigarrow^*$$

$$\{C_1 \sqcap \forall r_1.B_1 \sqcap C_2 \sqsubseteq D\} \cup \bigcup_{i=2}^{n-1} \{\forall r_i.B_i \sqsubseteq B_{i-1}\} \cup \{\forall r_n.A \sqsubseteq B_{n-1}\}$$

wobei B_i mit $(1 \leq i \leq n-1)$ neue Konzepte sind.

Diese Abkürzung hat einerseits den Vorteil, dass die mehrfache Anwendung einer Normalisierungsregel, zu einer Anwendung zusammengefasst wurde. Der größere Vorteil ist jedoch, dass bis auf $\{C_1 \sqcap \forall r_1.B_1 \sqcap C_2 \sqsubseteq D\}$ nur GCIs entstehen, die bereits in PANF sind. Somit müssen diese für weitere Regelanwendungen nicht mehr betrachtet werden. Die originale GCI wird also quasi durch eine teilweise normalisierte GCI ersetzt und eine Menge von bereits normalisierten GCIs generiert. Eine solche Abkürzung ist analog für NF2.2 möglich. Insgesamt lässt sich eine GCI in CCNF zu PANF normalisieren, indem auf die linke und rechte Seite diese die verketteten Normalisierungsregeln erschöpfend angewandt werden. Eine TBox in CCNF wird zu PANF normalisiert, indem alle originalen GCI auf diese Weise bearbeitet und alle neuen GCIs in die TBox eingefügt werden.

4.3 FINDEN ANWENDBARER REGELN

In der Hauptschleife von *SUBS* muss das kleinste $w \in \Delta^{\mathcal{I}^\Delta}$ bezüglich $\prec$ gewählt werden, sodass w nicht blockiert ist und es $E \sqsubseteq F \in \mathcal{T}$ gibt mit $hd(E) \subseteq hd(w)$ und $hd(F) \nsubseteq hd(w)$.

Ein naiver Ansatz würde alle $w \in \Delta^{\mathcal{I}^\Delta}$ in aufsteigender Reihenfolge bezüglich $\prec$ durchlaufen und für alle $E \sqsubseteq F \in \mathcal{T}$ überprüfen, ob $hd(E) \subseteq hd(w)$ und $hd(F) \nsubseteq hd(w)$ gilt. Indem man sich für alle Domänenelemente w merkt, ob sie seit der letzten Änderung schon überprüft wurden, würde man in der Praxis so in den meisten Fällen nur ein oder wenige w's betrachten müssen um eine anwendbare GCI zu finden. Dies würde dennoch bedeuten, dass man in großen Ontologien pro Domänenelement hunderttausendfach Teilmengenüberprüfungen von *head*-Strukturen vornimmt, welche wiederum aus $|N_R^{\mathcal{T}}| + 1$ Teilmengentests von Konzeptmengen bestehen. Pro Domänenelement würden somit $|\mathcal{T}| * (|N_R^{\mathcal{T}}| + 1)$ Teilmengenüberprüfungen für jedes Anwenden einer GCI nötig sein. Da im schlechtesten Fall, wie in [1, S. 45] dargestellt, bevor *SUBS* terminiert, exponentiell viele GCIs angewandt werden müssen, lohnt es nach einem effizienteren Ansatz zu suchen.

Der effizientere Ansatz basiert auf dem in [5] vorgestellten Rete Algorithmus. Dieser ermöglicht „Many Pattern - Many Object Pattern matching“ und soll somit dazu dienen, alle linken GCI Seiten mit allen $w \in \Delta^{\mathcal{I}^\Delta}$ abzugleichen.

Der Rete Algorithmus wurde für regelbasierte Systeme entworfen. Mögliche Regeln sind

(Gallier, Name=<X>, kräftig=False, Haustier=<Y>)
→ <X> bekommt Zaubertrank,

welche aussagt, dass schwache Gallier Zaubertrank bekommen und

(Gallier, Name=<X>, kräftig=True, Haustier=<Z>) ∧ (Hund, Name=<Z>, Größe=klein)
→ <X> bekommt keinen Zaubertrank

welche aussagt, dass kräftige Gallier mit einem kleinen Hund keinen Zaubertrank bekommen. Die erste Regel trifft dabei auf folgendes Objekt zu (Gallier, Name=Asterix, kräftig=False, Haustier=keins), während für die zweite Regel zum Beispiel die Objekte (Gallier, Name=Obelix, kräftig=True, Haustier=Idefix) und (Hund, Name=Idefix, Größe=klein) passen würden. Anstatt alle Regeln mit allen Objekten zu vergleichen, wird ein so genanntes Rete Netzwerk aufgebaut. Ein Rete Netzwerk ist ein gerichteter azyklischer Graph mit verschiedenen Knotentypen. Diese sind im Wesentlichen *intra-* und *inter-Element* Knoten und *finale* Knoten. *Intra-Element* Knoten überprüfen Eigenschaften einzelner Objekte, wie *kräftig* und *Größe*, während *inter-Element* Knoten den Zusammenhang zwischen verschiedenen Objekten herstellen, welcher in der zweiten Regel durch die gemeinsam verwendete Variable Z festgelegt wurde. Nicht mehrfach verwendete Variablen müssen nicht überprüft werden, da diese beliebig belegt sein können. Beginnen mehrere Regeln gleich, werden Knoten wiederverwendet, sodass Knoten niemals zwei Kinder mit identischen Beschriftungen haben (zum Beispiel bei der Überprüfung, ob ein Objekt ein *Gallier* ist). Die beiden vorgestellten Regeln werden durch Abbildung 4.4 repräsentiert. Wird ein Objekt verändert oder neu hinzugefügt, wird dieses an den Wurzelknoten übergeben und von dort aus an alle Kindknoten propagiert. In einem *intra-Element* Knoten wird die geforderte Eigenschaft (zum Beispiel Größe = klein) überprüft und falls sie erfüllt ist, wird das Objekt an alle Kinder weiter propagiert. In einem *inter-Element* Knoten bleibt ein Objekt zunächst hängen und wartet, bis das zweite korrespondierende Objekt diesen Knoten erreicht. Sind zwei passende Objekte am *inter-Element* Knoten angekommen, wird dieses Paar an alle Kindknoten propagiert. Kommen Objekte an einem *finalen* Knoten an, ist eine Regel erfüllt und die rechte Regelseite kann „ausgelöst" werden.

Wird also das Objekt (Gallier, Name=Obelix, kräftig=True, Haustier=Idefix) an das Netz aus Abbildung 4.4 übergeben, wird dieses bis zum *inter-Element* Knoten propagiert. Da zu diesem Zeitpunkt kein korrespondierendes Objekt am selben Knoten ist, bleibt dieses zunächst dort. Als nächstes wird (Gallier, Name=Asterix, kräftig=False, Haustier=keins) an das Netz übergeben. Dieses Objekt wird bis zum *finalen* Knoten „Zaubertrank geben" propagiert und erfüllt somit die erste Regel. Als nächstes wird (Hund, Name=Idefix, Größe=klein) an das Netz übergeben und bis zum *inter-Element* Knoten propagiert. Da bereits das korrespondierende Objekt (Z Variable stimmt überein) in diesem Knoten ist, erfüllt das Objektpaar den Knoten

und wird an den *finalen* Knoten weitergegeben und die entsprechende Regel kann angewandt werden.

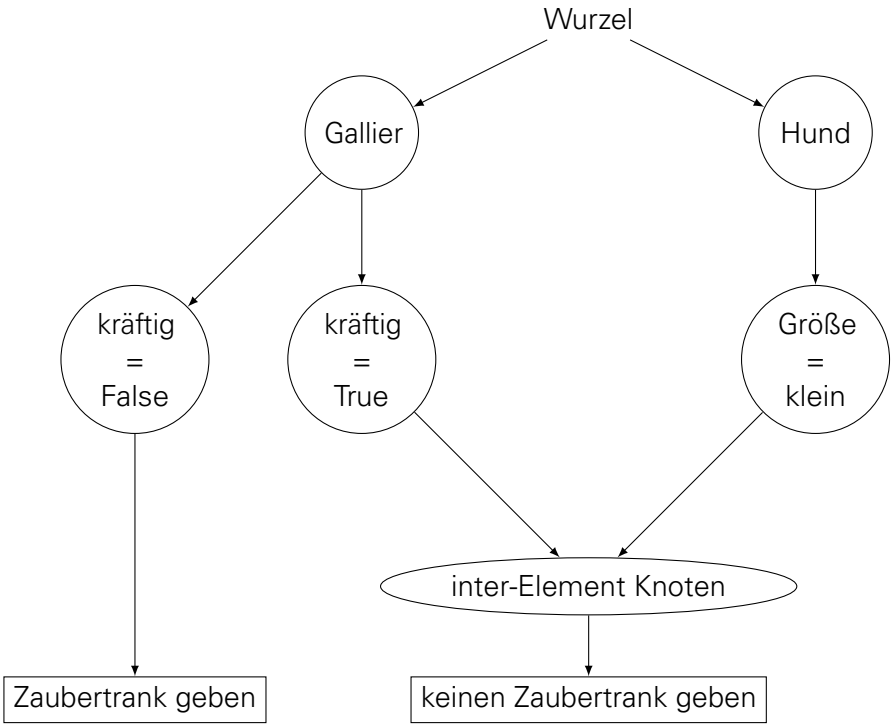

Abbildung 4.4: Beispiel für ein Rete Netzwerk

Da in komplexeren Netzen mehrere *finale* Knoten gleichzeitig erreicht werden können, werden erfüllte Regeln in einer Konfliktmenge gesammelt, aus welcher eine auszuführende Regel gewählt wird. Die in [5] vorgestellten Rete Netzwerke kennen weitere Knoten, welche zum Beispiel zur Darstellung von Negation dienen. Für den folgenden Ansatz werden aber die bisher vorgestellten Knotentypen ausreichen.

4.3.1 ANGEPASSTES RETE NETZWERK

Wie in Abschnitt 3.3 beschrieben wird ein $w \in \Delta^{\mathcal{I}^\Delta}$ und $E \sqsubseteq F \in \mathcal{T}$ mit $hd(E) \subseteq hd(w)$ gesucht. Die GCI kann als Regel $hd(E) \rightarrow hd(F)$ aufgefasst werden und w als Objekt. Die linke Regelseite lässt sich somit für hd(E) = $(L, L_1, \ldots, L_n)$ wie folgt formulieren:

$$\text{(Objektename = <W>, } L \subseteq \mathcal{I}^\Delta(\text{<W>})) \wedge$$
$$\bigwedge_{i=1}^{n} \text{(Objektename = <W>}r_i, L_i \subseteq \mathcal{I}^\Delta(\text{<W>}r_i)) \text{ mit <W> } \in N_R^*$$

Die Aufgabe der *intra-Element* Knoten wäre hierbei die Teilmengenüberprüfung. *Inter-Element* Knoten regeln das Zusammenführen aller zur selben *head*-Struktur gehörenden Objekte.

An dieser Stelle wird das, auf das Finden von anwendbaren GCIs angepasste, Rete Netzwerk vorgestellt. Dieses besteht aus einem *Wurzelknoten* und mehreren *intra-Element*, *Rollen-*, *inter-Element* und *finalen* Knoten, welche stets in dieser Reihenfolge durchlaufen werden. Die propagierten Objekte sind Domänenelemente w. Um die linke GCI-Seite $E = A_1 \sqcap A_2 \sqcap \forall r_1.A_3$ mit der *head*-Struktur $hd(E) = (\{A_1, A_2\}, \{A_3\})$ zu erfüllen, muss für ein $w \in N_R^*$ gelten, dass $\{A_1, A_2\} \subseteq \mathcal{I}^\Delta(w)$ und $\{A_3\} \subseteq \mathcal{I}^\Delta(wr_i)$. Um diese Untermengenrelationen zu überprüfen nutzen wir Ketten von *intra-Element* Knoten, welche jeweils für einen Konzeptnamen überprüfen ob ein bestimmtes Domänenelement in diesem enthalten ist. Da die Relation $\{A_3\} \subseteq \mathcal{I}^\Delta(wr_i)$) eigentlich für auf w (und nicht wr_i) anwendbare Regeln relevant ist, nutzen wir Rollenknoten. Dieser neue Knotentype überprüft einerseits, ob r_i die benötigte Rolle ist und erkennt andererseits, für welches Domänenelement die zuvor festgestellte Untermengenrelation relevant ist (wr_i oder w). *Inter-Element* Knoten testen, ob alle benötigten Untermengenüberprüfungen erfolgreich waren. *Finale* Knoten werden nur erreicht, wenn eine linke GCI Seite erfüllt ist und enthalten eine rechte GCI Seite, welche somit angewandt werden kann.

Der Wurzelknoten sammelt alle Untergraphen, welche einzelne GCIs repräsentieren und stellt somit selbst die TBox dar. Ein Domänenelement w, welches durch das Rete Netzwerk propagiert werden soll, wird stets an den Wurzelknoten gegeben, welcher es im originalen Rete Netzwerk an alle Kindknoten propagiert hätte. Die Kinder des Wurzelknotens sind stets *intra-Element* Knoten, wobei keiner doppelt auftritt, da gleich beginnende Knotenketten wiederverwendet werden. Somit hat er bis zu $|N_C^\mathcal{T}|$ viele Kinder. Da die meisten Konzeptnamen auch auf linken GCI-Seiten auftreten und die meisten GCIs relativ kurz sind (was relevant ist, da nur der Anfang einer Kette Kind des Wurzelknoten ist), sind es im Anwendungsfall nicht viel weniger als $|N_C^\mathcal{T}|$. Jedes w würde also an fast $|N_C^\mathcal{T}|$ viele Knoten propagiert und in fast allen Fällen wieder verworfen werden. Um diesen Aufwand zu verringern nutzen wir als Wurzelknoten ein $|N_C^\mathcal{T}|$-Tupel, in welchem an i-ter Stelle ein *intra-Element* Knoten steht, der A_i überprüft. Der Wurzelknoten propagiert w nur an die *intra-Element* Knoten, welche ein $A_i \in \mathcal{I}^\Delta(w)$ repräsentieren und die er somit über einen indexierten Zugriff effizient finden kann.

Die Betrachtungen bis zu diesem Zeitpunkt erfassen keine GCIs mit $\top$ als linker Seite, da diese keine *intra-Element* Knoten benötigen und somit nicht im *Wurzelknoten* repräsentiert sind. Deshalb enthält der Wurzelknoten zusätzlich eine Menge M_{top}, welche aus *finalen* Knoten besteht, deren Beschriftung $hd(F)$ die rechten Seiten von GCIs der Form $\top \sqsubseteq F$ repräsentieren.

Im Folgenden wird die Funktionsweise der einzelnen Knoten genauer erläutert. Ein Domänenelement wr_i wird durch das Rete Netzwerk propagiert, indem es an den Wurzelknoten gegeben wird.

- Der **Wurzelknoten** besteht aus dem Tupel $(x_1, \ldots, x_{|N_C^\mathcal{T}|})$ wobei x_j $(1 \leq j \leq |N_C^\mathcal{T}|)$ entweder ein *intra-Element* Knoten mit Beschriftung A_j und $f_{N_C}(A_j) = j$ ist, oder „NULL“, um auszudrücken, dass kein solcher Knoten existiert. Außerdem gehört zum Wurzelknoten

eine Menge *finaler* Knoten M_{top}, welche GCIs der Form $\top \sqsubseteq C$ repräsentiert. Wird ein $wr_i \in N_R \cup \{\epsilon\}$ an den Wurzelknoten gegeben, propagiert dieser das Paar (wr_i, r_i) an alle $x_j \neq NULL$, für die gilt $A_j \in \mathcal{I}^{\Delta}(wr_i)$ und $f_{N_c}(A_j) = j$. Außerdem wird es an alle finalen Knoten in M_{top} propagiert.

- **Intra-Element Knoten** mit Beschriftung $A \in N_C$ und Eingabe $(wr_i, r_i) \in (N_R^*, N_R \cup \{\epsilon\})$ überprüfen, ob $A \in \mathcal{I}^{\Delta}(wr_i)$. Im positiven Fall wird (wr_i, r_i) an alle Kindknoten propagiert und im negativen verworfen.

- **Rollenknoten** mit Beschriftung $r_j \in N_R \cup \{\epsilon\}$ und Eingabe $(wr_i, r_i) \in (N_R^*, N_R \cup \{\epsilon\})$ überprüfen, ob $r_i = r_j$. Im positiven Fall wird (w, r_i) an alle Kindknoten propagiert und im negativen verworfen. Wenn $r_j = \epsilon$, wird in jedem Fall (wr_i, ϵ) propagiert.

- **Inter-Element Knoten** mit Beschriftung $(r_{j_1}, \ldots, r_{j_m}) \in (N_R \cup \{\epsilon\})^m$ und Eingabe $(w, r_i) \in (N_R^*, N_R \cup \{\epsilon\})$ speichern für jedes w ein Tupel $\{True, False\}^m$. Ist noch kein solches Tupel vorhanden, wird es mit $\{False\}^m$ initialisiert. Wenn $r_{j_k} = r_i$ $(1 \leq k \leq m)$, wird der Wert des Tupels an Stelle k auf True gesetzt. Sind alle Werte innerhalb des Tupels True, wird dieses entfernt und (w, ϵ) an alle Kindknoten propagiert.

- **Finale Knoten** haben als Beschriftung eine *head*-Struktur hd(F) (einer rechten GCI Seite F). Bei Eingabe von $(w, r_i) \in (N_R^*, N_R \cup \{\epsilon\})$ wird $(w, hd(F))$ in eine sortierte Queue (Konfliktmenge) eingefügt. Es wird im Knoten gespeichert, für welche w er bereits ausgelöst wurde, um zu verhindern, dass er für ein w ein zweites Mal ausgelöst wird.

Im folgenden wird beschrieben, wie eine TBox $\mathcal{T}$ in PANF in ein Rete Netzwerk überführt wird. Hierfür werden die folgenden Punkte in gegebener Reihenfolge ausgeführt.

1. Der Wurzelknoten wird mit $\{NULL\}^{|N_C^{\mathcal{T}}|}$ und $M_{top} = \emptyset$ initialisiert.

2. Für jede GCI $E \sqsubseteq F \in \mathcal{T}$ mit $hd(E) = (L_\epsilon, L_1, \ldots, L_n)$ tu folgendes:

   a) Wenn $E = \top$ wird ein *finaler* Knoten mit Beschriftung $hd(F)$ in M_{top} eingefügt und mit der nächsten GCI fortgefahren.

   b) Erstelle aus jedem $L_i \subseteq N_C$ mit $i \in (\{\epsilon\} \cup [1, n])$ und $L_i \neq \emptyset$ eine Kette von *intra-Element* Knoten in beliebiger Reihenfolge, welche als Beschriftung Konzepte aus L_i tragen.

   c) Füge die Ketten entsprechend der Beschriftung des ersten Knotens in den Wurzelknoten ein (für einen ersten Knoten A wird die Kette in Stelle $f_{N_c}(A)$ eingefügt). Falls an dieser Stelle bereits ein Knoten eingetragen ist, wird die alte Kette bis zum ersten verschiedenen Nachfolger mitverwendet und erst ab diesem werden neue Knoten eingefügt.

   d) An den letzten Knoten der Ketten wird jeweils ein Rollenknoten mit der zu L_i korrespondieren Rolle r_i als Beschriftung angehängt.

e) Es wird ein *inter-Element* Knoten erstellt, welcher als Beschriftung die Menge aller in Punkt 2d verwendeten Rollen in beliebiger Reihenfolge nutzt. Alle Rollenknoten aus dem vorherigen Schritt bekommen diesen *inter-Element* Knoten als Kindknoten. (Falls es nur eine Kette gab, kann dieser Punkt übersprungen werden.)

f) An den *inter-Element* Knoten aus Schritt 2e wird ein *finaler* Knoten mit Beschriftung $hd(F)$ angehängt.

Beispiel 4.1. Für die TBox $\mathcal{T} = \{$
$A_2 \sqcap A_4 \sqcap A_5 \sqcap \forall r_1.A_3 \sqcap \forall r_1.A_4 \sqcap \forall r_2.A_1 \sqsubseteq B_7$,
$\forall r_2.A_3 \sqcap \forall r_2.A_4 \sqsubseteq B_8$
$\forall r_r.A_6 \sqsubseteq \forall r_1.B_9\}$

welche in *head* Schreibweise

$\{(\{A_2, A_4, A_5\}, \{A_3, A_4\}, \{A_1\}) \sqsubseteq (\{B_7\}, \emptyset, \emptyset)$
$(\emptyset, \emptyset, \{A_3, A_4\}) \sqsubseteq (\{B_8\}, \emptyset, \emptyset)$
$(\emptyset, \{A_6\}, \emptyset) \sqsubseteq (\emptyset, \{B_9\}, \emptyset)\}$

entspricht, entsteht somit folgendes Rete Netzwerk.

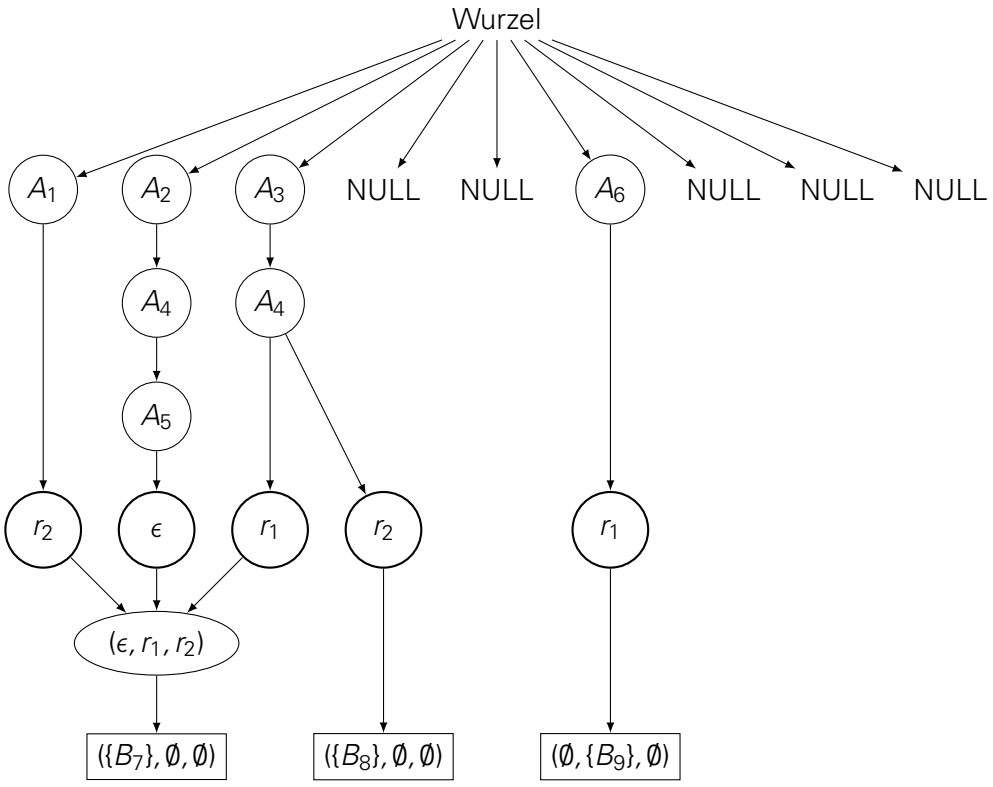

Abbildung 4.5: Rete Netzwerk für Beispiel 4.1

Wie bereits in der Beschreibung des finalen Knotens angedeutet, wird für eine GCI $E \sqsubseteq F \in \mathcal{T}$, deren linke Seite für ein $w \in \Delta^{\mathcal{I}^\Delta}$ erfüllt ist, $(w, hd(F))$ in eine sortierte Queue eingefügt.

Das Hauptkriterium, nach dem in dieser sortiert wird, ist die Größe von w bezüglich $\prec$ (vergleiche 3.4). Auf diese Weise ist gewährleistet, dass durch das Entnehmen des ersten Elementes der Queue ein Paar $(w, hd(F))$ gefunden wurde, sodass w kleinstmöglich bezüglich $\prec$ ist und $hd(E) \subseteq hd(w)$. Dies entspricht einem Teil der Bedingungen von *SUBS* in Zeile 4. Einerseits könnte w jedoch blockiert sein, was in Abschnitt 4.4 weiter besprochen wird und andererseits könnte $hd(F) \subseteq hd(w)$ sein. Zweiteres ist aber kein großes Problem, da somit *SUBS* Zeile 6 keine Auswirkungen hat und Mehrfachanwendungen von rechten GCI Seiten für dasselbe w durch die *finalen* Knoten ausgeschlossen sind. Weitere Kriterien für die Sortierung sind denkbar, um die Effizienz des Algorithmus zu steigern. Die tatsächlich gewählte Implementierung bevorzugt Regeln, welche Konzepte zum aktuellen Knoten hinzufügen gegenüber solchen, welche Kindknoten erweitern. Da auf diese Weise Änderungen weiter oben im Baum bevorzugt werden, ist die Hoffnung so schneller GCIs auszulösen, welche den Wurzelknoten betreffen. Wird hierdurch der gesuchte Subsumierer zum Wurzelknoten hinzugefügt, kann der Algorithmus früher terminieren. Andere Heuristiken sind durchaus denkbar und könnten in aufbauenden Arbeiten untersucht werden.

4.3.2 EFFIZIENZBETRACHTUNG

Das Rete Netzwerk wird im Algorithmus genutzt, indem abwechselnd das erste Element aus der Queue genommen und angewandt wird und danach alle geänderten $w \in \Delta^{\mathcal{I}^\Delta}$ durch das Netzwerk propagiert werden.

Im folgenden nehmen wir an, dass in praktisch genutzten Ontologien $|N_R^{\mathcal{T}}|$, die Größe einer GCI und für die meisten $w \in \Delta^{\mathcal{I}^\Delta}$ $|\mathcal{I}(w)|$ im Vergleich zu $|N_C^{\mathcal{T}}|$ und der Anzahl GCIs in $\mathcal{T}$ relativ klein ist. Diese Beobachtung trifft für alle, in der empirischen Auswertung verwendeten Ontologien zu.

Nach der Anwendung einer rechten GCI Seite müssen maximal $|N_R^{\mathcal{T}}| + 1$, im Normalfall aber nicht viel mehr als 2 bis 3 Domänenelemente w durch das Rete Netzwerk propagiert werden. Der Wurzelknoten gibt diese über einen Index direkt an $|\mathcal{I}(w)|$ viele *intra-Element* Knoten weiter. Die Länge der Pfade von *intra-Element* Knoten ist durch die Länge der GCIs begrenzt. Nach den *intra-Element* Knoten werden maximal noch ein *Rollenknoten*, ein *inter-Element* Knoten und ein *finaler* Knoten betreten. Somit ist der Zeitaufwand, um ein Domänenelement mit einer anwendbaren GCI zu finden, wesentlich geringer als bei einem naiven Ansatz, der über alle GCIs iteriert.

Dieser Geschwindigkeitsgewinn wird unter anderem durch erhöhten Speicheraufwand erlangt. *Inter-Element* Knoten und *finale* Knoten speichern für jedes bereits gesehene w geringe Datenmengen. Da jedoch $\Delta^{\mathcal{I}^\Delta}$ exponentielle Größe im Bezug auf $|N_C^{\mathcal{T}}|$ annehmen kann, können diese größere Mengen an Speicher belegen. Da jedoch für jedes $w \in \Delta^{\mathcal{I}^\Delta}$ auch andere Daten, wie $\mathcal{I}(w)$ gespeichert werden müssen, sollte der erhöhte Speicheraufwand im schlimmsten Fall ein kleiner konstanter Faktor sein.

Die Optimierung im Wurzelknoten kann in konstruierten Szenarien unwirksam sein. Man stelle sich eine TBox vor, in der jede GCI auf der linken Seite unter anderem ein Konzept *A* enthält. Der Wurzelknoten eines generierten Rete Netzwerks könnte somit einen einzigen, von „NULL" verschiedenen *intra-Element* Knoten als Nachfolger haben, nämlich *A*. Der Knoten mit Beschriftung *A* wiederum hätte eine hohe Zahl an Kindknoten. Somit würde *A* quasi die Rolle der Wurzel übernehmen, jedoch nicht für diese Position optimiert sein. Ein anderes mögliches Rete Netzwerk hätte *A* in jeder Knotenkette stets als letzten Knoten wodurch, außer im Wurzelknoten, jeder Knoten eine geringe Anzahl Kindknoten hat. Um zu gewährleisten dass stets das zweitere Rete Netzwerk generiert wird, sind Untersuchungen zur Balancierung von diesen modifizierten Rete Netzwerken von Interesse.

4.3.3 IMPLEMENTIERUNG

Aufgrund der nötigen Anpassungen an die Gegebenheiten in SUBS konnte keine Standart-Implementierung des Rete Netzwerks, wie sie in regelbasierten Systemen vorhanden sind, genutzt werden. Im Rahmen dieser Arbeit wurde das Rete Netzwerk, der Beschreibung entsprechend, als Graph aus Objekten implementiert. Das originale Rete Netzwerk wird in [5] jedoch nicht auf diese Weise umgesetzt. Anstelle dessen wird ausführbarer Programmcode generiert, welcher das Propagieren eines Domänenelementes durch ein gedachtes Netzwerk implementiert, ohne dieses explizit zu repräsentieren. Pfade in dem Netz werden durch Sprünge im generierten Code repräsentiert. Eine solche Implementierung des abgewandelten Rete Netzwerks könnte zu weiteren Effizienzsteigerungen führen und würde die Wiederverwendbarkeit, des einmalig für eine Ontologie generierten Rete Netzwerks, erleichtern und ist somit für anschließende Arbeiten von Interesse.

4.4 BLOCKIEREN

Um die Terminierung des Algorithmus *SUBS* zu gewährleisten, wird, wie in Abschnitt 3.3 beschrieben, „Anywhere Blocking" verwendet. Das heißt, dass eine GCI nur dann auf ein Domänenelement *w* angewandt wird, wenn *w* nicht *blockiert* ist. Dies haben wir bis jetzt unbeachtet gelassen und betrachten die Problematik in diesem Abschnitt. Hierzu teilen wir Definition 3.5, welche aussagt, wann ein Element *blockiert* ist, in zwei Arten von Blockierungen auf.

Definition 4.2. Ein Domänenelement $w \in \Delta^{\mathcal{I}^\Delta}$ gilt als

- *direkt blockiert*, gdw.

  es gibt ein nicht *blockiertes* $u \in \Delta^{\mathcal{I}^\Delta}$, sodass $u \prec w$ und $\mathcal{I}^\Delta(u) = \mathcal{I}^\Delta(w)$.

- *indirekt blockiert*, gdw.

  es gibt ein *direkt blockiertes* $u \in \Delta^{\mathcal{I}^\Delta}, v \in N_R^+,$ sodass $w = uv$.

- *blockiert*, gdw.

  w *direkt blockiert* oder *indirekt blockiert* ist.

4.4.1 ANALYSE DES PROBLEMS

Die Entscheidung, ob ein bestimmtes Domänenelement *w blockiert* ist, stellt sich als kompliziert heraus. Dies wird in Abbildung 4.6 deutlich gemacht. Das Element *w* könnte *direkt* oder *indirekt blockiert* sein. Um zu entschieden ob *w indirekt blockiert* ist, muss für alle Vorgänger *u* von *w* im funktionalen Interpretationsbaum untersucht werden, ob *w direkt blockiert* wird. Hierfür muss überprüft werden, ob *u* in den exakt gleichen Konzepten ist, wie ein beliebiges Domänenelement *v*, welches weiter oben im Baum steht. Ist ein solches *v* gefunden, muss anschließend für *v* entschieden werden, ob es *blockiert* ist. An dieser Stelle beginnt die gleiche anfängliche Überprüfung für *v*. Dieser Sachverhalt macht es vor allem für weit unten im Baum liegende Elemente aufwendig *Blockierungen* zu überprüfen. Auch eine statische Speicherung, welche Domänenelemente *blockiert* sind, stellt sich als aufwendig heraus, wie man in Abbildung 4.6 sieht. Einerseits ist die Bedingung in exakt den gleichen Konzepten zu sein sehr instabil. Anderseits führt die Eigenschaft *blockierter* Elemente, selbst nicht mehr blockieren zu können, zu möglichen Kettenreaktionen. Somit ist das dynamische Prüfen, ob ein Element *blockiert* ist, besonders komplex, wenn der Baum eher tief als breit ist und das zu überprüfende Element weit unten im Baum steht. Das statische Speichern hat Nachteile, wenn *direkt blockierte* Elemente viele Nachfolger haben und es häufig zu temporären Blockierungen kommt.

Eine weitere zu treffende Entscheidung ist, wie das Überprüfen von *Blockierungen* im Einklang mit dem im vorherigen Abschnitt 4.3 vorgestellten Finden von anwendbaren Regeln zusammenspielen kann.

4.4.2 GEWÄHLTE LÖSUNGEN

Das verwendete Rete Netzwerk platziert die anwendbaren Regeln und das Domänenelement *w*, auf das sie anwendbar sind, in einer sortierten Queue. Um eine Regel anzuwenden, wird das erste Element der Queue entnommen. An dieser Stelle setzen wir die Überprüfung, ob *w blockiert* ist, an. Ist *w blockiert*, wird die Regel nicht angewandt. Sobald *w* nicht mehr blockiert ist, muss das Paar aus anwendbarer Regel und *w* wieder in die Queue eingefügt werden. Um dies zu ermöglichen, wird das Paar mit der Blockierung verknüpft gespeichert. Was dies genau heißt, wird im folgenden besprochen.

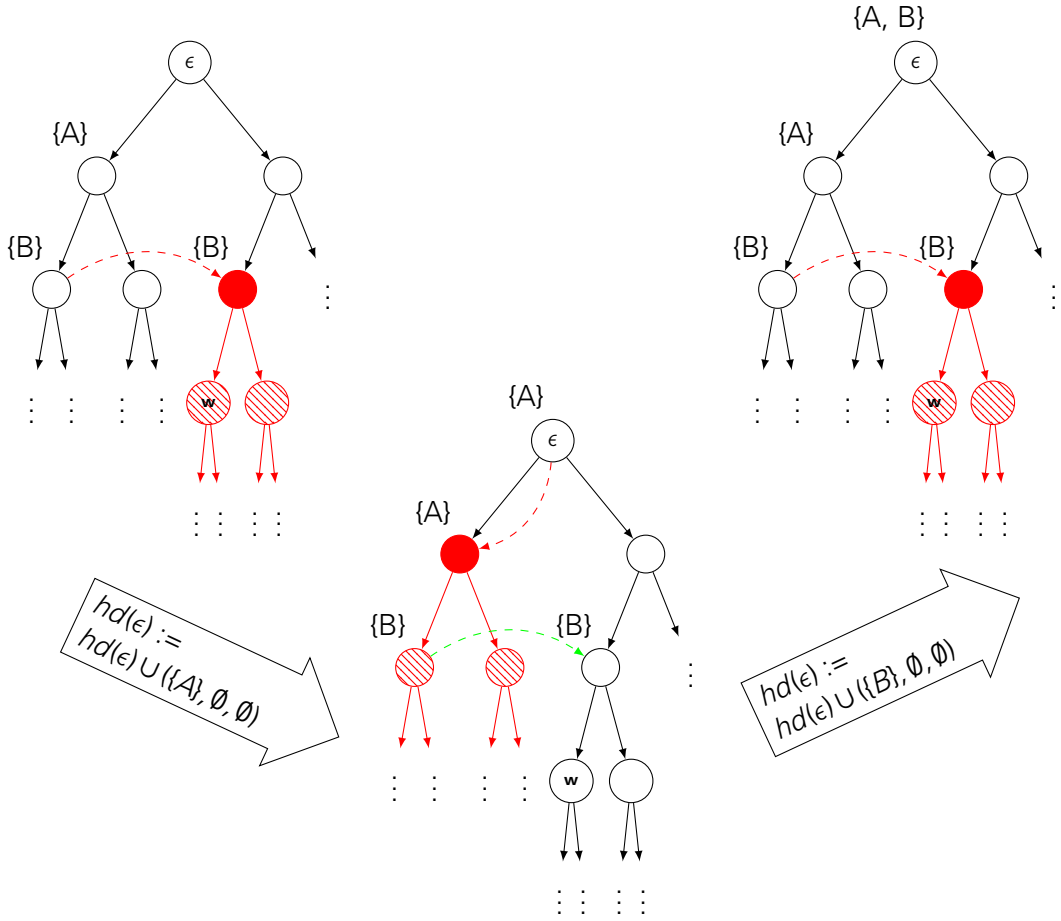

Abbildung 4.6: Schwierige Situationen beim Blockieren

Rot gefüllte Elemente sind *direkt blockiert* und rot gestrichelte *indirekt blockiert*. Trägt ein Knoten seine Konzeptmenge nicht als Beschriftung, ist diese für dieses Beispiel irrelevant.

Um zu entscheiden, ob ein Element *direkt blockiert* ist, muss dieses mit allen kleineren Domänenelementen bezüglich $\prec$ verglichen werden. Ein naiver Ansatz, welcher über alle Domänenelemente iteriert, wäre hierbei extrem aufwendig. Für die Überprüfung von *indirekten Blockierungen* sind viele Entscheidungen, ob ein Element *direkt blockiert* ist, nötig. Eine effiziente Möglichkeit zu entscheiden, ob ein anderes Domänenelement in exakt den gleichen Konzepten ist, wäre somit wünschenswert. Um dies zu realisieren, nutzen wir eine *Hash*-Tabelle. In diese wird für jedes Domänenelement w die Konzeptmenge $\mathcal{I}^{\Delta}(w)$ eingetragen. Ändert sich die Konzeptmenge eines Domänenelementes, wird diese aus der *Hash*-Tabelle entfernt und die aktualisierte Menge wieder eingetragen. Wird beim Eintragen festgestellt, dass der entsprechende Eintrag bereits existiert, wurde ein Domänenelement mit den gleichen Konzepten gefunden. Um feststellen zu können, welche Domänenelemente die gleichen Konzepte haben, werden diese in der *Hash*-Tabelle mit der Konzeptmenge verknüpft. Wenn für ein Domänenelement überprüft werden muss, ob andere Elemente die gleichen Konzeptmengen tragen, muss nur in der *Hash*-Tabelle nachgesehen werden, ob andere Elemente mit derselben Konzeptmenge verknüpft sind. Als Implementierung hierfür

nutzen wir Java *HashMaps* mit einem *Set* von Integern (Konzeptmenge) als *Key* und einer Menge von Domänenelementen als *Value*. Einfügen und Suchen in *HashMaps* hat bei gut gewählten Hashfunktionen eine Komplexität von $\mathcal{O}(1)$ und bei der schlechtest möglichen Hashfunktion eine Komplexität von $\mathcal{O}(\log n)$ [3]. Wir verwenden in der Implementierung die Standard Hashfunktion eines Java *Sets*. Diese Lösung ermöglicht effizientes Finden von anderen Domänenelementen mit der selben Konzeptmenge. Um zu entscheiden, ob ein Element *w* ein anderes Domänenelement *blockiert*, muss jedoch noch entschieden werden, ob *w* indirekt *blockiert* ist.

Um *indirekte Blockierung* zu entscheiden haben wir einen statischen Ansatz gewählt, welcher in jedem Element den aktuellen *Blockierungszustand* speichert. Wie bereits besprochen ist dieser Ansatz besonders aufwendig, wenn blockierte Elemente viele Nachfolgeknoten haben. Im Gegensatz zu einem dynamischen Ansatz steigert sich jedoch der Aufwand nicht mit der Tiefe der Elemente. Außerdem ändern sich die statischen Werte nur dann, wenn in der *Hash*-Tabelle eine neue Übereinstimmung festgestellt oder eine alte aufgelöst wurde. Die in jedem Element gespeicherten Werte sind: ob es *direkt blockiert*, *indirekt blockiert* und *blockierend* ist. Wenn durch eine Änderung in der *HashMap* eine neue Übereinstimmung festgestellt, oder eine alte aufgelöst wird, werden für alle betroffenen Domänenelemente die drei gespeicherten Werte neu evaluiert. Dies ist effizient möglich. Welche Elemente die gleichen Konzepte haben, ist durch die *HashMap* bekannt. Da in jedem Element gespeichert ist, ob es *indirekt blockiert* ist, lässt es sich ohne weiteres ermitteln, welche Elemente neu *direkt blockiert* sind und welche nicht mehr. Somit ergibt sich auch, welche Elemente *blockierend* sind. Für die betroffenen Elemente ändert sich an der *indirekten Blockierung* nichts. Hat sich für ein Element der *direkte Blockierungstatus* geändert, kann sich die *indirekte Blockierung* der Nachfolger ändern. Dies wird durch einen rekursiven Aufruf über die Kindknoten erreicht. Ist hierdurch ein *blockierendes* Element *indirekt blockiert* oder ein Element, welches Potential zum *blockieren* hat, nicht mehr *indirekt blockiert*, wird für dieses wie bei einer Änderung in der *HashMap* verfahren. Dieser Ansatz muss in der Implementierung weitere Sonderfälle beachten (zum Beispiel wenn ein *indirekt* und *direkt* blockiertes Element nur eine der beiden Eigenschaften verliert). Essentiell ändert der Ansatz jedoch nur den *indirekten Blockierungsstatus* von Nachfolgerknoten von Elementen, deren *direkte Blockierung* sich geändert hat. Des Weiteren beachtet er daraus folgende Änderungen von *direkten Blockierungen*, wie es in Abbildung 4.6 intuitiv deutlich wird.

Wird ein Paar aus Domänenelement *w* und einer GCI aus der Queue entnommen, muss somit nur noch abgelesen werden, ob *w blockiert* ist. Ist es *unblockiert*, kann die GCI angewandt werden. Ansonsten wird die GCI als zurückgehaltene Regel an *w* angehängt. Sobald sich der Status eines Domänenelementes von *blockiert* zu *unblockiert* ändert, werden alle angehängten GCIs wieder in die Queue eingetragen.

Auf diese Weise ist sichergestellt, dass Regeln nur für nicht blockierte Elemente angewandt werden. Somit haben wir alle Bestandteile eingeführt, welche wir benötigen um die Funktionsweise des Hauptprogramms vorzustellen.

4.5 HAUPTPROGRAMM

Das Hauptprogramm ist eine Konkretisierung von *SUBS* (Algorithmus 1). Es besteht im Wesentlichen aus einer Schleife, welche solange Regeln anwendet, bis keine weitere mehr anwendbar ist.

Algorithm 2 Konkreter SUBS Algorithmus

```
 1: procedure SUBS(A, B, 𝒯)                                   ▷ A, B ∈ N_C, 𝓕ℒ_0 TBox 𝒯
 2:     normalisiere 𝒯 zu 𝒯'
 3:     erstelle ein Rete Netzwerk ℛ aus 𝒯'
 4:     erstelle einen leeren Interpretationsstumpf ℐ^Δ        ▷ TreeMap<BigInteger, Element>
 5:     erstelle eine neue Blockierungsverwaltung ℬ            ▷ HashMap<Konzeptmenge, Element>
 6:     ℐ^Δ.put(0, neues Domänenelement mit Konzeptmenge {A})  ▷ A^{ℐ^Δ} := {ϵ}
 7:     ℛ.propagiere(0)                                        ▷ ℛ.queue wird befüllt
 8:     while ℛ.queue.not_empty() do
 9:         if B ∈ (ℐ^Δ.get(0).konzeptmenge) then
10:             return True
11:         end if
12:         (w, hd(F)) := ℛ.queue.pop_first()                  ▷ w ∈ ℕ
13:         w_elem := ℐ^Δ.get(w)                               ▷ Domänenelement mit Nummer w
14:         if w_elem.ist_blockiert() then
15:             hänge hd(F) an w_elem an
16:         else
17:             hd(w_elem) := hd(w_elem) ∪ hd(F)
18:             Sei M_geändert die Menge der Elemente, die in Zeile 17 geändert wurden
19:             ℬ.aktualisiere(M_geändert)
20:             ℛ.propagiere(M_geändert)                       ▷ ℛ.queue wird befüllt
21:         end if
22:     end while
23:     if B ∈ (ℐ^Δ.get(0).konzeptmenge) then
24:         return True
25:     end if
26:     return False
27: end procedure
```

Die Blockierungsverwaltung $\mathcal{B}$ enthält einerseits die *HashMap* und übernimmt andererseits die Aufgabe den *Blockierungsstatus* aller Domänenelemente zu aktualisieren wie in Abschnitt 4.4 beschrieben.

Alternativ könnte das *return-statement* aus der Schleife entfernt werden und nach Ende der Schleife die Menge aller Elemente, von welchen A subsumiert wird, ausgegeben werden.

Der konkretisierte Algorithmus wurde im Rahmen dieser Arbeit in Java implementiert. Diese Implementierung lieferte in manuellen Überprüfungen die korrekten Ergebnisse

und ist auch in der Lage mit großen Ontologien zu arbeiten. Aufgrund des Umfangs des Projektes und der zeitlichen Begrenzung für die Umsetzung, ist jedoch selbstverständlich mit Implementierungsfehlern, welche sich nur in selteneren Fällen zeigen, zu rechnen. Im folgenden werden wir die tatsächliche Leistungsfähigkeit des in dieser Arbeit betrachteten Algorithmus untersuchen.

5 EMPIRISCHE AUSWERTUNG

In den vorherigen Kapiteln haben wir den theoretischen Algorithmus *SUBS* betrachtet und diesen konkretisiert. In diesem Kapitel werden wir Messungen an der Implementierung vornehmen. Dazu werden wir zunächst die genutzten Testontologien betrachten und die Durchführung der Messungen besprechen. Hiernach folgt eine interne Analyse der Implementierung von *SUBS* , welche wir im Folgenden $SUBS_{\mathcal{I}}$ nennen werden. In dieser werden wir vor allem betrachten, wie viel Zeit das Programm für die einzelnen Schritte des Algorithmus tatsächlich braucht. Schließlich werden wir in der externen Analyse $SUBS_{\mathcal{I}}$ mit anderen bekannten Reasonern vergleichen.

5.1 TESTONTOLOGIEN UND ANFRAGEN

$SUBS_{\mathcal{I}}$ ist in der Lage OWL 2 [6] Ontologien, über die OWL API [7] einzulesen. OWL 2 erlaubt jedoch deutlich mehr Konstruktoren und Axiome, als in $\mathcal{FL}_0$ möglich sind. Aus diesem Grund gibt es OWL 2 Profile, welche die erlaubten Konstruktoren und Axiome limitieren. Es gibt allerdings kein $\mathcal{FL}_0$ Profil. Aus diesem Grund nutzen wir als Ausgangspunkt Ontologien im OWL 2 EL Profil [8]. Ersetzt man in einer $\mathcal{EL}$ Ontologie alle existenziellen Restriktionen durch Werterestriktionen, erhält man eine (nicht äquivalente) $\mathcal{FL}_0$ Ontologie. In OWL 2 Syntax bedeutet dies eine Ersetzung von „ObjectSomeValuesFrom“ durch „ObjectAllValuesFrom“. Des Weiteren erlaubt das OWL 2 EL Profil weitere Konstruktoren, welche in reinem $\mathcal{FL}_0$ und somit in $SUBS_{\mathcal{I}}$, nicht bekannt sind. Entfernt man diese ersatzlos, bleiben folgende Konstruktoren und Axiome in der Ontologie übrig: „ObjectAllValuesFrom“, „SubClassOf“, „EquivalentClasses“ und „ObjectIntersectionOf“. Das „EquivalentClasses“ Axiom lässt sich in $SUBS_{\mathcal{I}}$ durch zwei „SubClassOf“ Axiome interpretieren.

Als Quelle für die Ontologien wird die Ontologiensammlung Tones [9] und eine anonymisierte Version der SNOMED [10] $\mathcal{EL}$ Ontologie genutzt. Aus dieser Sammlung werden zunächst alle $\mathcal{EL}$ Ontologien ausgewählt. Diese werden wie oben beschrieben in reine (nicht äquivalente) $\mathcal{FL}_0$ Ontologien transformiert. Anschließend werden alle Ontologien verworfen, die aus weniger als jeweils 500 Konzepten und GCIs bestehen oder keine Rollen (und somit keine Werterestriktionen) enthalten. Somit blieben, neben SNOMED, 23 Ontologien mit jeweils über 1000 und unter 70000 Axiomen übrig.

Diese Herangehensweise gewährleistet, dass die Testontologien eine ähnlich Struktur haben, wie real eingesetzte $\mathcal{EL}$ Ontologien. Aus Mangel an realen $\mathcal{FL}_0$ Ontologien lässt sich jedoch nicht mit absoluter Sicherheit sagen, dass natürliche $\mathcal{FL}_0$ Ontologien eine ähnliche Struktur haben würden.

Die von $SUBS_{\mathcal{I}}$ implementierten Funktionen sind die Entscheidung einer Subsumptionsrelation zwischen zwei gegebenen Konzepten und das Finden aller Konzepte, welche ein gegebenes Konzept subsumieren. Da die Schwierigkeit der Überprüfung einer Subsumptionsrelation davon abhängig ist, ob diese gilt oder nicht, unterscheiden wir diese beiden Fälle. Somit entstehen drei verschiedene Problemstellungen, welche wir ab jetzt wie folgt nennen werden: *positive Subsumption*, *negative Subsumption* und Finden der *Subsumierermenge*. Für jede Ontologie werden zu jeder dieser drei Problemstellungen fünf zufällige, konkrete Probleme generiert. Die fünf resultierenden Ergebnisse werden in allen Fällen durch das arithmetische Mittel zusammengefasst. Um ein *positives Subsumptionsproblem* zu generieren wird zunächst ein Konzept zufällig aus der Ontologie gewählt. Für dieses wird die *Subsumierermenge* berechnet und aus diesen ein zufälliges Konzept gewählt. *Negative Subsumptionsprobleme* können äquivalent generiert werden. Eine Problem für das Finden der *Subsumierermenge* besteht lediglich aus einem zufällig gewählten Konzept.

5.2 DURCHFÜHRUNG

Die Problemstellungen werden automatisiert an den entsprechenden Reasoner gegeben. Dieser wird dabei über die Kommandozeile kontrolliert. Über GNU Time [11] werden

- die Dauer,
- der maximale Arbeitsspeicherverbrauch und
- die durchschnittliche Prozessorauslastung

gemessen. $SUBS_{\mathcal{I}}$ gibt zusätzlich aus, wie viel Zeit die verschiedenen internen Aufgaben benötigt haben.

Die Versuche werden hierfür auf einem Computer mit folgenden Eigenschaften ausgeführt.

Betriebssystem	GNU/Linux	Kernel: 4.12.6-1-ARCH, x86_64
Prozessor	Intel Core i5-7600K	4 Kerne, 3,80GHz
Arbeitsspeicher	16GB	2 * 8GB DDR4

Für Ausführung wurde den Reasonern ein Time-out von zehn Minuten gegeben. In den Fällen, in denen dieser Überschritten wurde, hat auch eine Erhöhung des Time-outs auf eine Stunde keine Änderungen ergeben.

5.3 INTERNE ANALYSE

Für die interne Analyse nutzen wir die größte Ontologie aus dem Testsatz, die SNOMED Ontologie. Dies gewährleistet, dass alle Zeiten im messbaren Bereich liegen und verringert bei Werten im größeren Bereich die Ungenauigkeit. Das Finden der *Subsumierermenge* entspricht für $SUBS_{\mathcal{I}}$ der *negativen Subsumption* mit dem einzigen Unterschied, dass für zweiteres keine Rückübersetzung der Konzepte aus der internen Repräsentation nötig ist. Die *positive Subsumption* unterscheidet sich für $SUBS_{\mathcal{I}}$ darin, dass *SUBS* durch eine Abbruchbedingung früher terminieren kann. Da das Finden der *Subsumierermenge* die anderen beiden Anfragen enthält, betrachten wir für die interne Analyse nur das Finden der *Subsumierermenge*. Diese Problemstellung wird, wie in der Durchführung beschrieben, für fünf zufällig generierte Probleme gelöst. Die präsentierten Ergebnisse entsprechen dem arithmetischen Mittel der fünf gemessenen Werte.

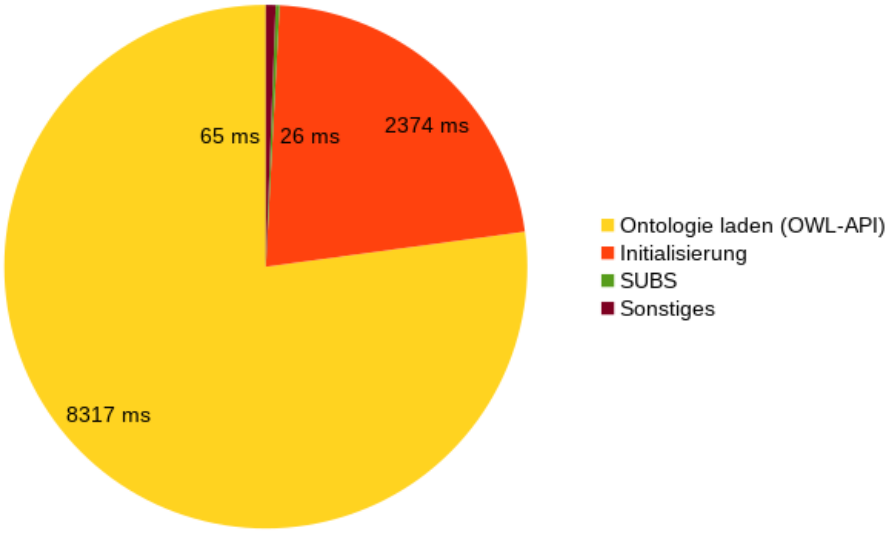

Abbildung 5.1: Gesamte Programmausführung für SNOMED

Insgesamt brauchte $SUBS_{\mathcal{I}}$ durchschnittlich 10,9 s für die gesamte Programmausführung. Mit 8,3 Sekunden dauerte das Einlesen der Ontologie über die OWL-API mit Abstand am längsten. Die zweitlängste Aufgabe ist mit 2,4 s die Initialisierung. In dieser wird die Ontologie in eine interne Repräsentation überführt und normalisiert. Anschließend wird sie in eine durch *head*-Strukturen inspirierte Repräsentation überführt. Als letzter Schritt der Initialisierung wird das Rete Netzwerk generiert. Das eigentliche berechnen der *Subsumierermenge* brauchte nur 26 ms.

Der besonders für größer werdende Probleme relevanteste Teil des Algorithmus macht also den kleinsten zeitlichen Anteil aus. Dies ist inhaltlich sehr positiv zu Betrachten, wird jedoch die externen Auswertungen erschweren, da der größte Teil der Zeit nicht für das Finden der *Subsumierermenge* benötigt wird.

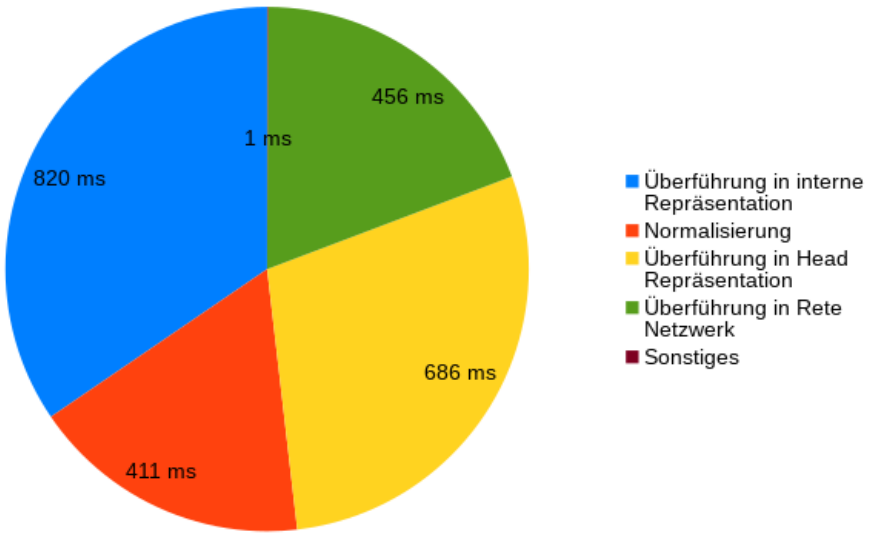

Abbildung 5.2: Initialisierung von SNOMED

Wie in Abbildung 5.2 zu erkennen ist, brauchen die Phasen der Initialisierung alle vergleichbar lang.

Will man die Laufzeit von $SUBS_{\mathcal{I}}$ optimieren bieten alle vier Phasen hierfür Potential. Auch steht zu überlegen, ob diese zu weniger Schritten zusammengefasst oder einzelne Repräsentationen übersprungen werden können. Für eine Ontologie ließe sich theoretisch auch nach einmaliger Initialisierung das Rete-Netzwerk speichern. Somit könnten alle bisherigen Schritte durch das einlesen des Rete-Netzwerks ersetzt werden.

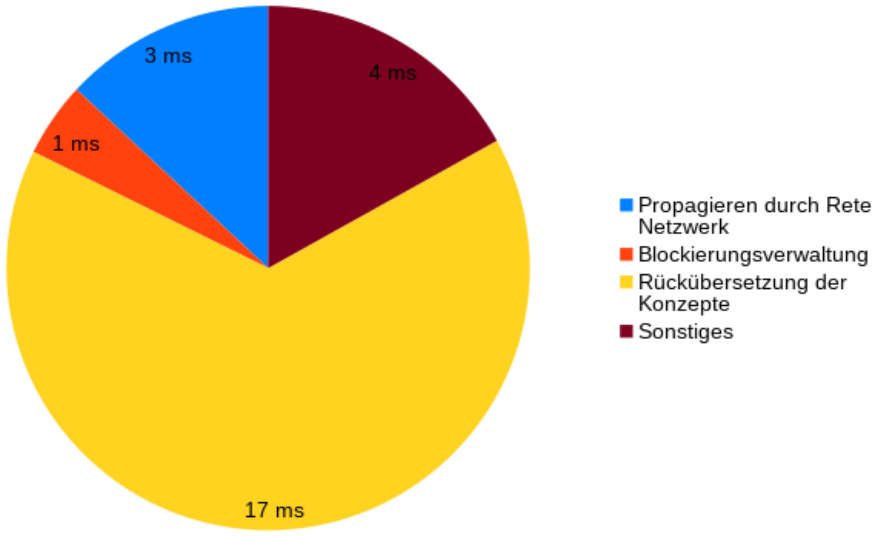

Abbildung 5.3: Finden der *Subsumierermenge* in SNOMED

Das Finden der *Subsumierermenge* dauert insgesamt 26 ms. Wie in Abbildung 5.3 deutlich wird, werden hiervon 15 ms für das Übersetzen der *Subsumierermenge* von interner

Repräsentation (natürliche Zahlen), in die originalen Konzeptnamen, benötigt. Der eigentliche Algorithmus zum Finden aller Subsumierer benötigt also insgesamt nur 8 ms um zwischen 5 und 31 Subsumierern aus insgesamt 291145 Konzepten zu finden. $SUBS_{\mathcal{I}}$ brauchte für das Propagieren aller Domänenelemente bis zur Terminierung insgesamt nur 3 ms. Für das Erkennen von Blockierungen wurde über die gesamte Ausführungszeit insgesamt nur 1 ms verwendet. Somit ist die Optimierung des Findens von anwendbaren Regeln und der Überprüfung von Blockierungsbedingungen effektiv. Vor allem die Zeit für die Blockierungsverwaltung könnte jedoch mit wechselnden Problemen variieren. Die 4 ms, die in Abbildung 5.3 als „Sonstiges“ markiert sind werden für noch genauer zu untersuchende, bisher unbestimmte, Schritte verwendet. Hier wären genauere Untersuchungen und Optimierungen interessant, da diese mit dem Finden von anwendbaren Regeln und der Blockierungsverwaltung bei schwerer werdenden Problem stärker ins Gewicht fallen würden.

Ansonsten ergeben die gemessenen Werte für $SUBS_{\mathcal{I}}$ eine durchschnittliche Prozessorlast von 220%. Hierbei könnten 100% bedeuten, dass ein Prozessorkern voll ausgelastet ist und 400% würden einer vollen Auslastung aller vier Kerne entsprechen. Somit ist $SUBS_{\mathcal{I}}$ teilweise in der Lage mehrkernige Prozessoren auszunutzen.

In einem weiteren Versuch vergleichen wir für alle Tones-Ontologien die Dauer des gesamten Programmes mit der Dauer zum Laden der Ontologien über die OWL-API. Die Zeiten werden hier in Relation zu der Anzahl der Axiome in der Ontologie dargestellt.

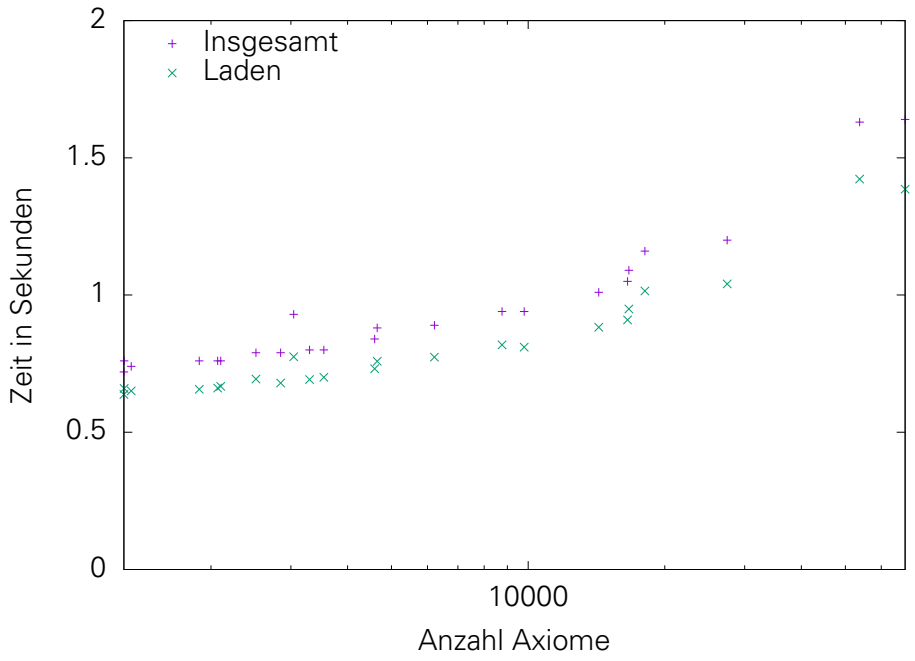

Abbildung 5.4: Dauer zum Laden der Ontologien im Vergleich zur gesamten Laufzeit

Wie in Abbildung 5.4 erneut deutlich wird, ist die Gesamtlaufzeit von $SUBS_{\mathcal{I}}$ wesentlich vom Laden der Ontologien dominiert. Diese Erkenntnis ist vor allem für die externen Analyse relevant.

5.4 EXTERNE ANALYSE

In der externen Analyse betrachten wir $SUBS_{\mathcal{I}}$ im Vergleich zu bekannten Reasonern. Hierfür werden wir die benötige Zeit und den Speicherverbrauch, für die drei bekannten Problemstellungen analysieren. Da wir für die Vergleichsreasoner nur die Gesamtlaufzeit messen können, vergleichen wir in allen Messungen die Werte für eine komplette Programmausführung. Diese enthält in allen Fällen das Laden der Ontologie und andere vom Reasoner vorgenommene Initialisierungen.

5.4.1 VERGLEICHSREASONER

Als Vergleichsreasoner werden HermiT [12] und Konclude [13] genutzt. Diese sind OWL 2 DL Reasoner und ermöglichen somit implizit auf $\mathcal{FL}_0$ Ontologien zu arbeiten. Die Entscheidung fiel auf diese beiden Reasoner, da sie zu aktuell zu den besten verfügbaren OWL 2 Reasonern zählen, wie die OWL Reasoner Evaluation (ORE) 2015 ergab [14]. Konclude führt die Liste der besten OWL 2 DL Reasoner an und Hermit belegt je nach Kategorie Platz zwei bis drei. HermiT bietet im Gegensatz zu Konclude das Berechnen der *Subsumierermenge* an, weshalb sich dieser besonders gut für den Vergleich mit $SUBS_{\mathcal{I}}$ eignet. Für die anderen im ORE-Wettbewerb vertreten Reasoner besteht die Problematik, dass diese, zumindest über die Kommandozeile keine Entscheidungen über Subsumption oder das Finden der *Subsumierermenge* erlauben. HermiT und Konclude erlauben nicht direkt Subsumption zu entscheiden, aber die Erfüllbarkeit von Konzepten. Da $C \sqsubseteq_{\mathcal{T}} D$ gdw. für $\mathcal{T}' := \mathcal{T} \cup \{E \sqsubseteq C \sqcap \neg D\}$ das neue Konzept E bezüglich $\mathcal{T}'$ unerfüllbar ist, ist die Subsumptionsprüfung indirekt möglich. Hierfür wurden die Ontologien entsprechend der Problemstellung um die zuvor beschriebene GCI erweitert.

5.4.2 ERGEBNISSE

Als erstes betrachten wir das Problem des Findens der *Subsumierermengen* für die TONES-Ontologien.

In Abbildung 5.5 fällt auf, dass HermiT für kleinere Ontologien zunächst ungefähr doppelt so schnell wie $SUBS_{\mathcal{I}}$ ist, jedoch in den größten Testontologien überholt wird. Diese Ergebnisse sind jedoch mit Vorsicht zu betrachten, da die von den Reasonern benötigte Zeit, vor allem bei den kleineren Ontologien, durch das Laden der Ontologien dominiert wird. So benötigt $SUBS_{\mathcal{I}}$ schon zum Laden der Ontologien für die kleineren Ontologien länger als HermiT insgesamt (vgl. Abbildung 5.4).

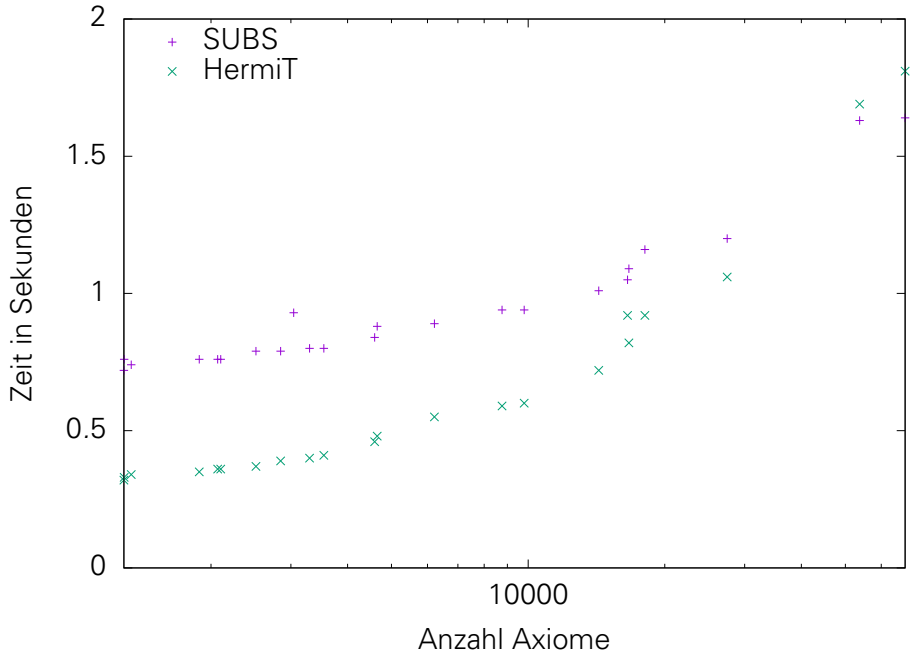

Abbildung 5.5: Finden der Subsumierermenge

Als nächstes betrachten wir die Entscheidung von Subsumptionsrelationen für die TONES-Ontologien.

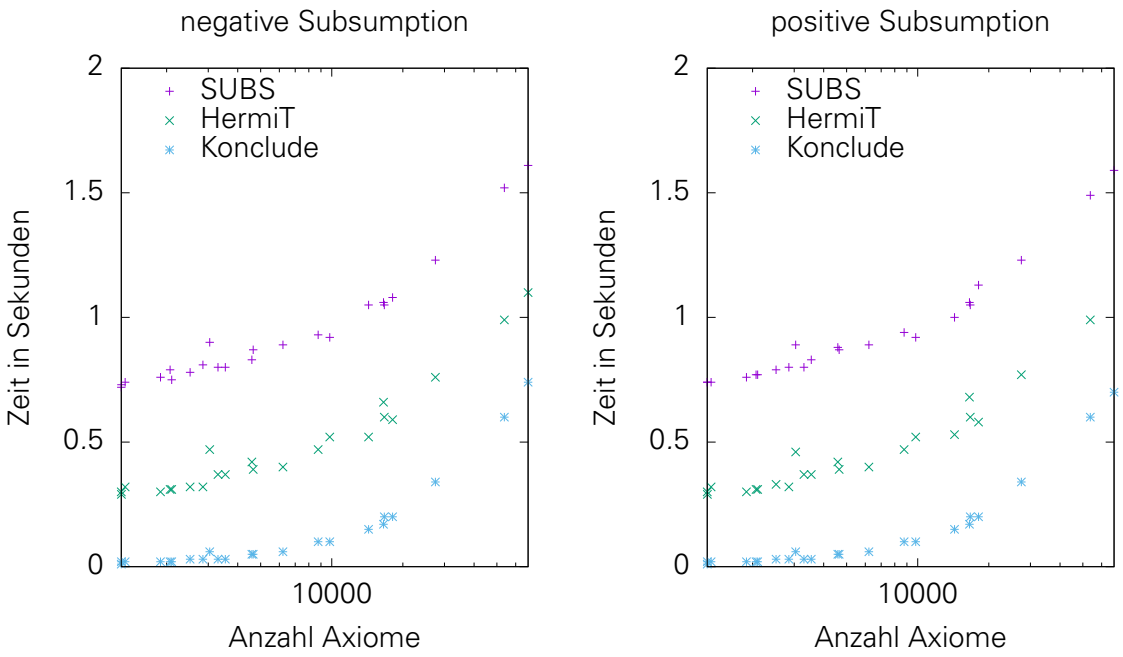

Abbildung 5.6: Benötigte Zeit zum Entscheiden von negativer und positiver Subsumption

Wie in Abbildung 5.6 deutlich wird, ist Konclude mit Abstand am schnellsten und $SUBS_{\mathcal{I}}$ am langsamsten. Des Weiteren fällt auf, dass die Abstände zwischen den Werten der einzelnen

Reasoner ungefähr konstant zu seien scheinen. Diese sind durch die Zeit zum Laden der Ontologien wieder mit Vorsicht zu betrachten. Vergleicht man die Resultate des HermiT Reasoners mit Abbildung 5.5 fällt jedoch auf, dass dieser Subsumption deutlich schneller entscheidet, als er *Subsumierermengen* findet. Die Zeitdifferenz stellt somit die Zeit dar, welche der HermiT Reasoner mit Sicherheit nicht für das Laden der Ontologie, sondern für das Finden der *Subsumierermenge* nutzte. Vergleicht man dieses mit der Zeit, welche $SUBS_{\mathcal{I}}$ nicht zum Laden der Ontologie benötigte Abbildung 5.4, liegt die Schlussfolgerung nahe, dass $SUBS_{\mathcal{I}}$, nach Abzug der Zeit für das Laden der Ontologie, dieses Problem tatsächlich schneller löst.

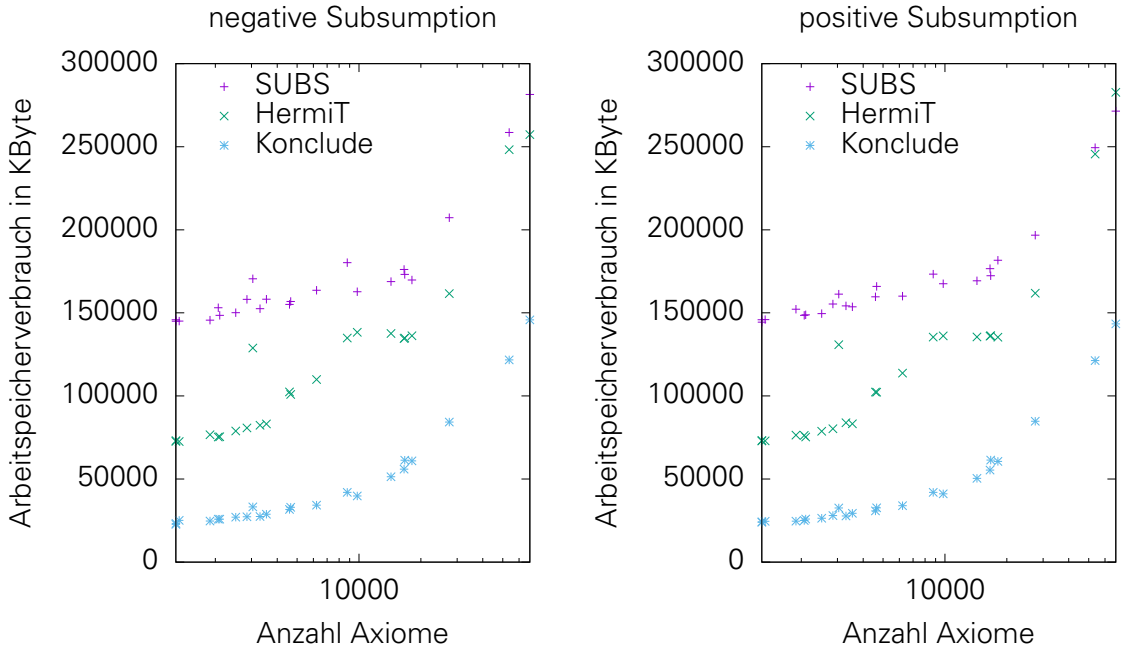

Abbildung 5.7: Maximale Arbeitsspeichernutzung beim Entscheiden von negativer und positiver Subsumption

Auch wenn man den maximalen Speicherverbrauch betrachtet, ergibt sich die selbe Rangfolge zwischen den Reasonern, wobei HermiT und $SUBS_{\mathcal{I}}$ für die größten Ontologien auf ähnliche Werte kommen. Auch hier ist zu beachten, dass unbekannt ist, ob die maximale Speichernutzung während des Ladens der Ontologie, oder während der eigentlichen Algorithmen zustande kam. Sowohl bei der Laufzeit, als auch bei der Speichernutzung sind die Unterschiede zwischen *positiver* und *negativer* Subsumption gering. Dies unterstützt wiederum die Beobachtung, dass die gemessenen Werte zu großen Teilen vom Laden der Ontologien dominiert werden.

Da SNOMED in der externen Analyse bis jetzt nicht betrachtet wurde, werden an dieser Stelle die Messwerte für SNOMED einzeln vorgestellt. SNOMED ist mit 291144 Axiomen die größte getestete Ontologie. Da HermiT beim Finden der *Subsumierermenge* nicht in unter zwei Stunden terminierte, entfallen hierfür die Ergebnisse.

	negative Subsumption		positive Subsumption	
	Laufzeit	Arbeitsspeichernutzung	Laufzeit	Arbeitsspeichernutzung
$SUBS_{\mathcal{I}}$	10,9 s	1,7 GByte	10,6 s	1,7 GByte
HermiT	17,9 s	2,3 GByte	17,6 s	2,2 GByte
Konclude	13,2 s	1,5 GByte	13,2 s	1,5 GByte

Am Schnellsten war in diesem Fall $SUBS_{\mathcal{I}}$ und Konclude am speichereffizientesten. Es ist immer zu beachten, dass die Messwerte durch das Laden der Ontologie dominiert sein können.

Neben den bisher betrachtete Ontologien enthielt TONES eine Ontologie („fma-conversion/-trunk“) mit 119558 GCIs, für die $SUBS_{\mathcal{I}}$ nicht terminierte. Genauere Beobachtungen ergaben, dass pro Domänenelement viele tausend GCIs anwendbar waren. Dies führte einerseits dazu, dass es trotz schnellem Finden von anwendbaren GCIs und effizienter Überprüfung von Blockierungen lange dauerte, bis alle möglichen GCIs auf ein Element angewandt waren. Hierdurch wurde die Konzeptmenge pro Domänenelement auch deutlich größer als in anderen Anwendungsfällen. Aus diesem Grund wurde die Erfüllung von Blockierungsbedingungen schwerer. Insgesamt führte dies dazu, dass $SUBS_{\mathcal{I}}$ nicht in unter einer Stunde terminierte. Weitere Untersuchungen dieser Problematik sind für weiterführende Arbeiten wünschenswert.

5.5 AUSWERTUNG

Insgesamt zeigen die Versuche, dass $SUBS_{\mathcal{I}}$ auch für große Ontologien in der Lage ist in kurzer Zeit Subsumptionsrelationen zu entscheiden und die *Subsumierermenge* zu berechnen. Die Optimierungen, welche Geschwindigkeitsgewinne durch erhöhten Speicherbedarf erlangten führten nicht dazu, dass $SUBS_{\mathcal{I}}$ einen auffällig hohen Speicherbedarf hatte. Da das Laden der Ontologien einen dominanten Einfluss auf die Messwerte hatte, lässt sich im Bezug auf die Entscheidung von Subsumptionsrelationen wenig über die Effizienz im Vergleich zu anderen Reasonern Schlussfolgern. Dies bedeutet jedoch auch, dass sich $SUBS_{\mathcal{I}}$ als nicht offensichtlich Nachteilhaft herausgestellt hat. Im Finden der *Subsumierermenge* hat sich $SUBS_{\mathcal{I}}$ als gegenüber HermiT überlegen herausgestellt, da HermiT im Gegensatz zu $SUBS_{\mathcal{I}}$ hierbei offensichtlich nicht nur vom Laden der Ontologie dominiert wurde und außerdem für SNOMED nicht terminierte. Die Schwächen von $SUBS_{\mathcal{I}}$ zeigten sich in der beschriebenen Ontologie, für die der Algorithmus nicht terminierte. Ob dies ein generelles Problem darstellt, oder sich durch weitere Optimierungen beheben lässt, gilt es in weiterführenden Arbeiten zu erforschen.

6 SCHLUSSBETRACHTUNG UND WEITERFÜHRENDE ARBEITEN

Um diese Arbeit zusammenzufassen, beginnen wir mit einem Rückblick auf die vorgenommenen Schritte. Wir formulierten in der Einleitung die Hoffnung, Subsumption für $\mathcal{FL}_0$ TBoxen mit einem strukturellen Ansatz effizienter entscheiden zu können als dies mit herkömmlichen $\mathcal{ALC}$ Reasonern möglich ist. Um diese These zu überprüfen, setzten wir uns mit dem hierfür vorgeschlagenen Algorithmus *SUBS* auseinander. In diesem identifizierten wir das Finden von anwendbaren GCIs als die zu konkretisierende Problematik. Aus diesem Grund entwickelten wir eine angepasste Variante eines Rete Netzwerks, welches diese Aufgabe effizient umsetzt. Da dieses die Überprüfung von Blockierungen zunächst ignoriert, betrachteten wir anschließend wie sich diese durch eine Hash-Tabelle und statisches Speichern von indirekten Blockierungen dennoch beachten lassen. Des Weiteren entwickelten wir eine technische Repräsentation für funktionale Interpretationsstümpfe, welche einerseits effizienten Zugriff auf die Domänenelemente ermöglicht und andererseits die Relationen zwischen diesen leicht überprüfbar hält. Nach diesen Entwurfsentscheidungen wurde der Algorithmus implementiert, sodass die anfängliche These in einer empirischen Auswertung überprüft werden konnte. In dieser stellte sich der strukturelle Algorithmus *SUBS* in der Entscheidung von Subsumption effizient genug heraus, um im Bezug auf Laufzeit und Speichernutzung mit aktuellen State of the Art Reasonern mithalten zu können. Eine exakte Feststellung, ob dieser Subsumption tatsächlich schneller entscheiden kann als bekannte $\mathcal{ALC}$ Reasoner, ließ sich jedoch nicht abschließend klären. Die empirische Auswertung ergab allerdings, dass *SUBS* in der Berechnung aller Subsumierer eines Konzeptes gegenüber $\mathcal{ALC}$ Reasonern überlegen ist. Somit hat sich gezeigt, dass sich strukturbasierte Ansätze für $\mathcal{FL}_0$ trotz gleicher theoretischer Komplexität als vorteilhaft gegenüber der Anwendung von Standard-$\mathcal{ALC}$-Algorithmen erweisen können.

Für zukünftige Arbeiten wären Effizienzuntersuchungen, welche nicht durch die Dauer zum Laden und Initialisieren der Ontologien beeinflusst werden von hohem Interesse. In Bezug auf die Optimierung von $SUBS_{\mathcal{I}}$ sind besonders Untersuchungen vielversprechend, welche andere, eventuell restriktivere, Blockierungsbedingungen betrachten oder das Rete Netzwerk durch Programmcodegenerierung umsetzen.

AKRONYME

$\mathcal{ALC}$	attributive language with Complex concept negation
BL	Beschreibungslogik
CCNF	*concept-conjunction Normalform* (concept-conjunction normal form)
$\mathcal{EL}$	Beschreibungslogik mit Konjunktion und existentiellen Restriktionen
GCI	generelle Konzept Inklusion (general concept inclusion)
$\mathcal{FL}_0$	Beschreibungslogik mit Konjunktion und Werterestriktion
PANF	*plane-axiom Normalform* (plane-axiom normal form)
ORE	OWL Reasoner Evaluation

LITERATURVERZEICHNIS

[1] M. Pensel. An Automata-Based Approach for Subsumption w.r.t. General Concept Inclusions in the Description Logic $\mathcal{FL}_0$. Master's thesis, Technische Universität Dresden, 2015.

[2] F. Baader. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[3] Blanche. Performance comparison of several Java Map. [Online] `http://www.programering.com/a/MzM4MjNwATE.html`, Sep. 2014. [Stand 14.08.2017].

[4] Oracle. Primitive Data Types. [Online] `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`. [Stand 14.08.2017].

[5] Charles L. Forgy. Expert systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[6] Keio MIT, ERCIM. OWL 2 Web Ontology Language. [Online] `https://www.w3.org/TR/owl2-overview/`, Dez. 2012. [Stand 14.08.2017].

[7] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

[8] Keio MIT, ERCIM. OWL 2 Web Ontology Language Profiles. [Online] `https://www.w3.org/TR/owl2-profiles/#OWL_2_EL`, Dez. 2012. [Stand 14.08.2017].

[9] Manchester OWL Repository. [Online] `http://mowlrepo.cs.manchester.ac.uk/datasets/tones/`, 2010. [Stand 14.08.2017].

[10] SNOMED International. SNOMED CT. [Online] `http://www.snomed.org/snomed-ct/`. [Stand 14.08.2017].

[11] GNU Time. [Online] `https://www.gnu.org/software/time/`. [Stand 14.08.2017].

[12] Univerity of Oxford. HermiT OWL Reasoner. [Online] `http://www.hermit-reasoner.com/`. [Stand 14.08.2017].

[13] Derivo. Reasoning System Konclude. [Online] `http://derivo.de/produkte/konclude/`. [Stand 14.08.2017].

[14] Bijan Parsia, Nicolas Matentzoglu, Rafael S. Gonçalves, Birte Glimm, and Andreas Steigmiller. The owl reasoner evaluation (ore) 2015 competition report. *Journal of Automated Reasoning*, Feb 2017.

[15] Jon Skeet. HashMap get/put complexity. [Online] `https://stackoverflow.com/questions/4553624/hashmap-get-put-complexity`, Jan. 2016. [Stand 14.08.2017].