A Description Logic Prover in Prolog

Ismail Faizi

Kongens Lyngby 2011
IMM-B.Sc.-2011-8

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

This work presents development of educational resources for teaching Description Logics and reasoning about them using Tableaux algorithms. The starting point of this work is the Master thesis of Thomas Herchenröder from the University of Edinburgh. Using Herchenröder's work and his implementation of Tableaux algorithm for reasoning in $\mathcal{ALC}$, two extensions are developed. One of these extensions introduces an alternative ontology format that is more human-friendly compared to the one used by Herchenröder. In addition, a converter is developed that converts expressions between the two formats. The second extension, *Xtableaux.pl*, is directly developed on top of the Prolog implementation of Herchenröder, *tableaux.pl*. This extension is meant to construct the intermediate steps in the proof process. The intention behind this extension is to help students understand the Tableaux algorithms and help in debugging the ontology.

Resumé

Dette rapport præsenterer udviklingen af uddannelsesresourcer til undervisning af beskrivelseslogik og ræsonnement omkring dette ved hjælp af Tableaux algoritmer. Udgangspunktet for dette arbejde er et speciale af Thomas Herchenröder fra University of Edinburgh. Ved at burge Herchenröder's arbejde og hans implementation af Tableaux algoritmen er der udviklet to udvidelser. En af disse udvidelser indfører en alternativ ontologi format, der er mere menneskevenlige i forhold til den, der bruges af Herchenröder. Hertil er der udviklet en konverter til at konvertere udtryk mellem de to formater. Den anden udvidelse, *Xtableaux.pl*, er en direkte udvidelse af Prolog implementationen udviklet af Herchenröder, den så kaldte *tableaux.pl*. Denne udvidelse er beregnet til at konstruere de mellemliggende trins i bevisførelsen. Hensigten bag denne udvidelse er til at hjælpe eleverne med at forstå Tableaux algoritmer og at hjælpe med at fejlfinde ontologier.

Preface

This bachelor project was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the B.Sc. degree in engineering.

The project deals with development of educational resources for teaching Description Logic and reasoning about it using Tableaux algorithm. The main focus is based on the work done in a Master thesis about the same topic.

The project consists of both the theoretical aspects of Description Logic and implementation of knowledge representation systems based on Description Logic. The work on this project began on 31th of January this year and was completed on 27th of June the same year. This project is credited with 20 ECTS points.

Lyngby, June 2011

[illegible]

Ismail Faizi

Acknowledgements

I thank my supervisors Jørgen Villadsen first by recommanding this project and second for his constructive ideas and his faith in my abilities to manage this project. It was both enjoyable and easy to work with him.

My heartfelt gratefulness goes to Thomas Herchenröder, whome I have never seen or have been in contact with in any way, for his great work which I have based my work on it.

I am grateful to have leaved in the era of researchers like Franz Baader, Diego Calvanese, Deborah L. MacGuinness, Daneile Nardi, Peter F. Patel-Schneider and Ian Horrocks. In the words of Isaac Newton "If I have seen further than others, it is by standing upon the shoulders of giants."

Contents

CHAPTER 1

Introduction

In Artificial Intelligence (AI) one has always been interested in representing knowledge in a manner so it is possible to reason about it, i.e. draw conclusions from the knowledge. The field of the AI which deals with it is Knowledge Representation (KR) and Reasoning.

This field was much popular in 1970s and it was in that time when the development of approaches to knowledge representations began. These developments are sometimes divided in two categories, logical-based formalisms and non-logical-based representations. The latter were developed based on network structures and rule-based representation – cognitive notions that were derived from experiments on recall from human memory and execution of tasks. These approaches were expected to be more general-purpose, although, they were often developed for specific domains. Unlikely, the logical-based approaches were general-purpose from the beginning since these were a variant of first-order logic (FOL) which provides very powerful and general machinery.

In the logical-based approach, reasoning is equivalent to verifying logical consequence, while in the non-logical approaches, it is accomplished by means of ad hoc procedures that manipulate the similarly ad hoc data structures that are meant to represent the knowledge. Among such representations we have *semantic networks* and *frames*. Both of these can be regarded as network structures where the aim of the network is to represent sets of individuals and their

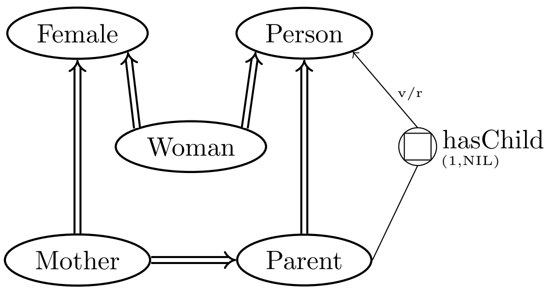


Figure 1.1: An example Network (cf. Fig. 1.1 in [1, p. 6]).

relationships. For this reason the term *network-based structures* is used to refer to these representation.

Although, network-structures were more appealing and effective from a practical point of view, they were not fully satisfactory since they lacked a precise semantic characterization. The recognition that frames could be given a semantics by relying on FOL led to a characterization that could not capture the constraints of semantic networks and frames with respect to logic. However, it turned out that semantic networks and frames could be regarded as fragments of FOL since they did not require all the machinery of FOL. Furthermore, it turned out that different features of the representation language led to different fragments of FOL which also resulted in computational problems of differing complexity with respect to reasoning in these fragments. However, the typical forms of reasoning in structured-based representations could be accomplished by specialized reasoning techniques.

Research in the area of Description Logics (DLs) began subsequent to above realization. In the beginning the research was under the label *terminological systems* which emphasized that the representation language was used in order to establish basic terminology in the modeled domain. Later, the label was changed to *concept languages* since the emphasize was on the set of concept-forming constructs declared in the language. In recent years the term *Description Logics* became popular since the attention was moved further towards the properties of the underlying logical systems.

1.1 From Networks to Description Logics

Using a simple example, I will provide some intuition about the ideas behind knowledge representation in network form. We will avoid references to any particular system and only speak in terms of a generic network.

A network consists of two elements, *nodes* and *links*. The first one is typically used to characterize concepts, i.e. sets of individual objects. The latter one is used to characterize relationships among concepts. Here, we will not consider the knowledge about specific individuals and restrict our attention to knowledge about concepts and their relationships.

The simple example, whose pictorial representation is given in figure 1.1 on the facing page, represents knowledge concerning family relationships. The link between `Mother` and `Parent` is a “IS-A” relationship. In this case it means that “mothers are parents”. The IS-A relationship provides the basis for inheriting properties among the concepts and thus defines a hierarchy over these. For instance the concept of `Person` is a more general concept than the concept of `Woman` – if `Person` has an age so has the `Woman`.

The ability of DLs to represent relationships beyond the IS-A relationship makes it their characteristic feature. For instance the property called a “role” is one of these relationships. The concept of `Parent` has such property which is expressed by a link to a node labeled `hasChild`. The role has a so called “value restriction”, denoted by v/r, which puts limitations on the range of types of objects that can fill it. The node has also a number restriction, expressed as (1,NIL), which puts a lower and upper bound on the number of children (NIL denotes infinity). The concept of `Parent` can be read as “A parent is a person having at least one child, and all of his/her children are persons.” [1, p. 6].

The concept of `Mother` inherits the restriction on its `hasChild` role from `Parent` since relationships of this kind are inherited from concepts to their subconcepts. There may also be implicit relationships between concepts. For instance, every `Mother` is a `Woman` since the concept of `Woman` is defined as a female person. These kind of inferences are a characterization of the properties of the network. It becomes more difficult to give a precise characterization of what kind of relationships can be computed when the established relationships among concepts becomes more complex. Although, a number of systems were implemented and used using the above ideas, the need for a more precise characterization of the meaning of a network emerged. This meaning can be given by defining a language and by providing an interpretation for the strings of the language [1, p. 7]. Such language is given in chapter 2 on page 7.

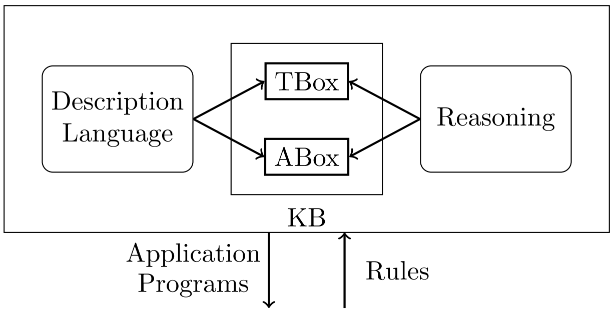


Figure 1.2: Architecture of a DL-based knowledge representation system (cf. Fig. 2.1 in [4, p. 50]).

1.2 DL-based KR Systems

There has been a tight connection between theoretical results and implementation of systems in the research on DL. This has been a characterization of research on DL.

A DL-based KR system provides facilities to set up knowledge bases, to reason about their contents and to manipulate them [4, p. 50]. The architecture of such a system is sketched in figure 1.2.

A knowledge base (KB) consists of two components, the TBox and the ABox. The first one introduces the *terminology* of an application domain, while the second one contains *assertions* about named individual in the application domain. These are further described in chapter 2 on page 7.

A DL system is not only a storage place for terminologies and assertions of an application domain. The system also offers services that *reason* about the stored terminologies and assertions. Typical reasoning tasks for TBox and ABox are described in more details in chapter 2 on page 7.

As shown in figure 1.2 a KR system is embedded into a larger environment. This is the case in any application. There is an interaction between other components and the KR system. These components query the KB and modify it, i.e. adding and retracting concepts, roles and assertions. A restricted mechanism for adding assertions uses rules [4, p. 51]. This notion is beyond the scope of this work and will not be further mentioned here.

1.3 Application Domains

Description Logics have many application domains. One of the first application domain for DLs was Software engineering where the basic idea was to implement a system that would support the software developer by finding out information about a large software system.

One very successful application domain has been configuration. This includes applications that support the design of complex systems created by combining a number of components, e.g. choosing computer components in order to build a home PC.

There are many other applications domain that have been addressed by the DL community. This work will not further discuss this aspect of Description Logics. For more information one is encouraged to look up [1].

1.4 Motivation

This work is based on a Master thesis of Thomas Herchenröder (from now on Herchenröder) from the University of Edinburgh. The thesis with the title "Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics" is from 2006.

A large portion of this work has been inspired from Herchenröder's work. The implemented reasoner in Prolog, *tableaux.pl*, has been used to develop services for better understanding the Tableaux algorithms.

1.5 Document Structure

Chapter 2 is dedicated to discuss the foundation of Description Logics. In chapter 3 the Tableaux algorithm for reasoning with DLs is introduced. Chapter 4 discusses various issues and design alternatives in the implementation of this algorithm and presents the implementation introduced by Herchenröder. In chapter 5, a formal specification of Herchenröder's implementation is given. Chapter 6 presents the extensions to this implementation and how they are tested. Chapter 7 closes with some conclusions.

Chapter 2

Description Logics

Description Logic (DL), or Description Logics, is a family of formal knowledge representation languages which has a set-theoretical foundation. Like propositional logic and unlike first-order logic, it has decision procedures. It is, however, more expressive than propositional logic.

DL is used to represent the knowledge of an application domain by defining the relevant concepts in the domain. And since it is equipped with a formal, logic-based semantics, it is possible to reason about the application domain.

2.1 Basic Description Logic

The basic DL is called *Attributive Concept Language with Complements* (abbreviated $\mathcal{ALC}$) [2, p. 2]. It is the basis for the more expressive DLs.

2.1.1 Syntax

In the following I will explain the grammar of the $\mathcal{ALC}$ which is shown in table 2.1 on the following page. Before doing so, I will describe the more fun-

Concept expression ::	$\bot \mid \top \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall R.C \mid \exists R.C$
Terminological axiom ::	$C \equiv D \mid C \sqsubseteq D$

Table 2.1: Grammar of the $\mathcal{ALC}$ DL [2, p. 8].

damental elements of DL which are common for all DLs. These are presented in both [2], [4] and [5]. Some of the elements are presented under different names among these works, and some are only presented in one or two of the three works. In the following I will be using the notation of Herchenröder unless other work is referenced.

In DL there are two kinds of symbols; denoted by *atomic symbols*. The first one is the so called *atomic concepts* (denoted by A, B) and the other one is *atomic roles* (denoted by R) [4, p. 51].

From the FOL point of view, atomic concepts are unary predicates while atomic roles are binary predicates [4, p. 49]. Beside atomic concepts and atomic roles, DL consists of so called *individuals* which from FOL point of view are constants.

One can construct arbitrary concept description (denoted by C, D) using atomic symbols and the so called *constructors* (*concept constructors* and *role constructors*). Here, we will only consider concept constructors since $\mathcal{ALC}$ only has atomic roles.

Now back to the grammar of the $\mathcal{ALC}$ (see table 2.1). As mentioned earlier, DL has a set-theoretical foundation. This is because the concepts represented in DL are interpreted as sets [2, p. 6]. Looking at grammar for forming *concept expressions* in $\mathcal{ALC}$, one can see that it provides two default concepts denoted by "bottom" ($\bot$), the empty set, and "top" ($\top$), the universe. These are not further defined in the logic, but receive their contents through an interpretation (e.g. model). According to [5], the $\top$ concept is an abbreviation for $C \sqcup \neg C$. A concept expression is also an "atomic concept" (A) (also know as *primitive concept* [2, p. 7] or *concept name* [5, p. 5]).

A concept expression is also one of the "negation" ($\neg C$), "conjunction" ($C \sqcap D$) and/or "disjunction" ($C \sqcup D$). These are the usual Boolean constructors known in logic. The concept description $\neg C$ means everything that is out side of C; from a set-theoretical point of view it is the complement of C. Like so, the concept descriptions $C \sqcap D$ and $C \sqcup D$ mean C 'and' D and C 'or' D (logically) respectively. From a set-theoretical point of view it is C intersected with D and

$$
\begin{array}{rcl}
\top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} & = & \emptyset \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} & = & \left\{a \in \Delta^{\mathcal{I}} \mid \exists b.\ (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\right\} \\
(\forall R.C)^{\mathcal{I}} & = & \left\{a \in \Delta^{\mathcal{I}} \mid \forall b.\ (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\right\} \\
(C \sqsubseteq D)^{\mathcal{I}} & = & C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \\
(C \equiv D)^{\mathcal{I}} & = & C^{\mathcal{I}} = D^{\mathcal{I}}
\end{array}
$$

Table 2.2: Formal semantic of $\mathcal{ALC}$-concepts and terminological axioms [4, p. 52-53,p. 56] [5, p. 6].

the union of C and D respectively [2, p. 7].

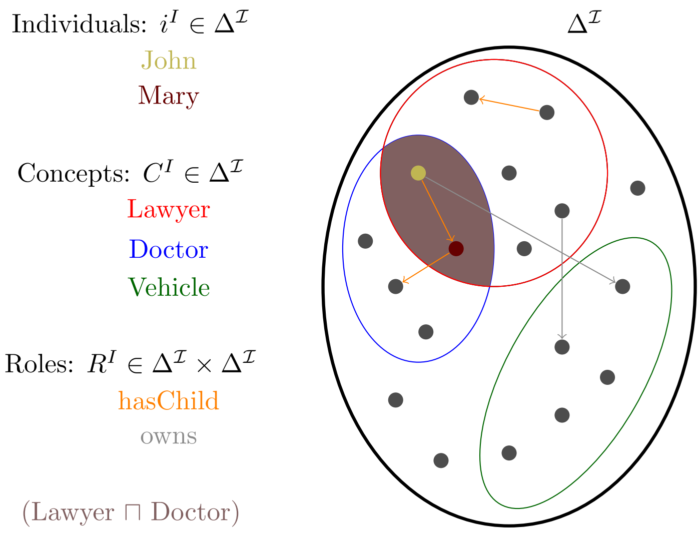
Last but not least, a concept expression is also one of the *value restriction* ($\forall R.C$) and *existential restriction* ($\exists R.C$). They are not interpreted the same way as we know them from FOL. According to Herchenröder , *"they describe concepts as sets of individuals that are characterized by the individuals they relate to through a given relation"* [2, p. 6]. These are further clarified in subsection 2.1.2.

2.1.2 Semantics

In the following I describe a formal semantics for $\mathcal{ALC}$ concept descriptions and terminological axioms. Such description is given in both [4, p. 52-56] and [5, p. 6].

The formal semantics for the grammar presented in table 2.1 on the preceding page is given as follows. We consider an *interpretation* $\mathcal{I}$ as a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where the domain (of interpretation) $\Delta^{\mathcal{I}}$ is a non-empty set and $\cdot^{\mathcal{I}}$ is the interpretation function that assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Figure 2.1 on the following page illustrate these notions using an extended Venn diagram. Here the text on the left is matched to the diagram elements on the right using colors.

The interpretation function is extended to the concept descriptions and the terminological axioms presented in table 2.1 on the preceding page. This results in inductive definitions shown in table 2.2. In the following I will elaborate on



Individuals: $i^I \in \Delta^{\mathcal{I}}$
John
Mary
Concepts: $C^I \in \Delta^{\mathcal{I}}$
Lawyer
Doctor
Vehicle
Roles: $R^I \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
hasChild
owns
(Lawyer ⊓ Doctor)
$\Delta^{\mathcal{I}}$

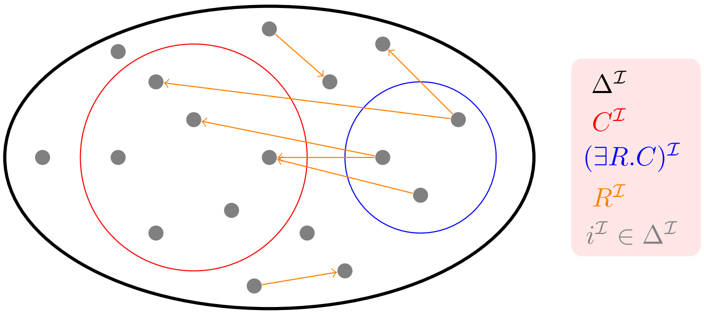
Figure 2.1: A diagram explaining the semantics of $\mathcal{ALC}$.

two of the definitions from this table; value and existential restriction.

2.1.2.1 Existential Restriction

The interpretation of existential restriction, as it is shown in table 2.2 on the previous page, is defined as:

$$(\exists R.C)^{\mathcal{I}} = \left\{a \in \Delta^{\mathcal{I}} \mid \exists b.\ (a,b) \in R^{\mathcal{I}} \land b \in C^{\mathcal{I}}\right\}$$



$\Delta^{\mathcal{I}}$
$C^{\mathcal{I}}$
$(\exists R.C)^{\mathcal{I}}$
$R^{\mathcal{I}}$
$i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$

Figure 2.2: A diagram illustrating the semantics of existential restriction.

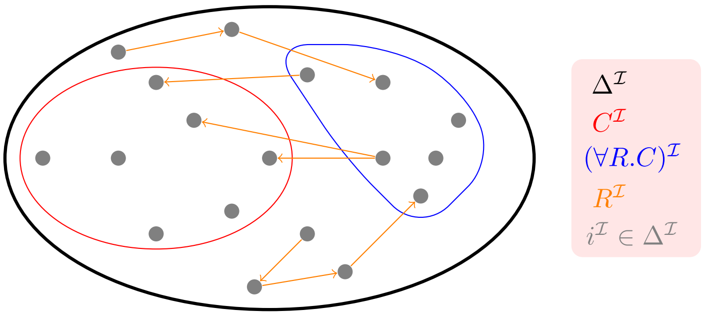

$\Delta^{\mathcal{I}}$
$C^{\mathcal{I}}$
$(\forall R.C)^{\mathcal{I}}$
$R^{\mathcal{I}}$
$i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$

Figure 2.3: A diagram illustrating the semantics of value restriction.

In set-theoretical terms it is the set of all individuals (a) that are related through R to at least one individual (b) from the concept C. This is illustrated by the diagram in figure 2.2 on the facing page where colors match symbols and shapes. For instance the concept $C^{\mathcal{I}}$ is represented by a red circle. The individuals in the domain, i.e. $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, are represented using gray dots.

In FOL, the notion of existential restriction is equal to:

$$\exists y.R(x,y) \wedge C(y)$$

where x is an arbitrary individual in the interpretation domain.

2.1.2.2 Value Restriction

The interpretation of value restriction, as it is shown in table 2.1.2 on page 9, is defined as:

$$(\forall R.C)^{\mathcal{I}} = \left\{a \in \Delta^{\mathcal{I}} \mid \forall b.\ (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\right\} \tag{2.1}$$

The FOL equivalent of above is:

$$\forall y.R(x,y) \rightarrow C(y)$$

In set-theoretical terms it is the set of all individuals that are related through (R) to only individuals from the concept (C). This is illustrated by the diagram in figure 2.3. As it is shown in the diagram, individuals out side the domain of R, i.e. $(a,b) \notin R^{\mathcal{I}}$, are also in this set. The reason for that is the implication in (2.1). Since the implication is only false when its premise is true, i.e. $\neg((a,b) \in R^{\mathcal{I}}) \vee (b \in C^{\mathcal{I}})$, the individuals not in the domain of R will fulfill this, thus they qualify for being in the set.

Woman	$\equiv$	Person $\sqcap$ Female
Man	$\equiv$	Person $\sqcap$ $\neg$Woman
Mother	$\equiv$	Woman $\sqcap$ $\exists$hasChild.Person
Father	$\equiv$	Man $\sqcap$ $\exists$hasChild.Person
Parent	$\equiv$	Father $\sqcup$ Mother
GrandMother	$\equiv$	Mother $\sqcap$ $\exists$hasChild.Parent
MotherWithoutDaughter	$\equiv$	Mother $\sqcap$ $\forall$hasChild.$\neg$Woman
Wife	$\equiv$	Woman $\sqcap$ $\exists$hasHusband.Man

Table 2.3: An example of a TBox [4, p. 56].

Example 2.1 *In this example we form some correct concept descriptions in the domain of family relationships. Lets assume that* `Female` *and* `Person` *are atomic concepts and* `hasChild` *is an atomic role. Using these atomic symbols and concept constructs and role restrictions we can describe more complex concepts. For instance the concept of* `Woman` *is described as:*

$$\texttt{Person} \sqcap \texttt{Female}$$

The concept of `Mother` *can be described likewise:*

$$\texttt{Woman} \sqcap \exists\texttt{hasChild.Person}$$

A more complex concept such as "mother not having any daughter" is described using all the above concept descriptions:

$$\texttt{Mother} \sqcap \forall\texttt{hasChild.}\neg\texttt{Woman}$$

For more examples on such concept descriptions, please refer to figure 2.3.

2.1.3 Knowledge Bases

In order to reason about any knowledge or manipulate it, a KR system based on Description Logic has to provide a facility. This is done by providing a so called

MotherWithoutDaughter(MARY)	Father(PETER)
hasChild(MARY,PETER)	hasChild(PETER,HARRY)
hasChild(MARY,PAUL)	

Table 2.4: An example of an ABox [4, p. 65].

knowledge base or KB. As mentioned earlier, a KB consists of two components, the **TBox** and the **ABox**.

2.1.3.1 TBox

The TBox contains the terminology, i.e. the vocabulary of an application domain [4, p. 50]. These are relations such as equivalence ($\equiv$) and subsumption ($\sqsubseteq$) between concepts forming terminological axioms. An example of a TBox is given in table 2.3 on the facing page.

It is possible to reason about the TBox and not only store terminologies. Typical reasoning tasks are *satisfiability* (whether a description is non-contradictory) and *subsumption* (finding a more general concept).

2.1.3.2 ABox

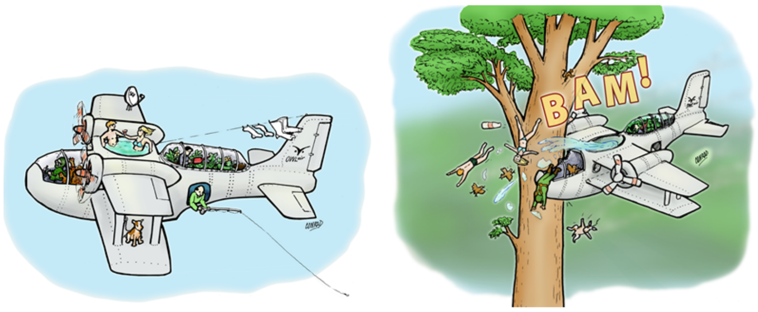
The ABox contains assertions about concrete individuals in terms of the terminology contained in the TBox. These assertions are formed by assigning individuals (e.g. `John`, `Mary`) to concepts (e.g. `Parent`) or relating them using roles (e.g. `hasChild`) in order to describe relations between individuals. An example of an ABox is given in table 2.4 on the preceding page.

For ABox, *consistency* of assertions is an important reasoning problem. Here one tests whether an assertion has a model, if it is the case then it is consistent [4, p. 50].

In a KB it is very useful to check for satisfiability of descriptions and consistency of sets of assertions. This way one can determine weather a knowledge base is meaningful at all. Testing whether a concept is subsumed by another one is also useful. Doing so, one can organize concepts in hierarchy according to their generality [4, p. 51].

2.2 Beyond $\mathcal{ALC}$

As mentioned earlier, $\mathcal{ALC}$ is not very expressive. In order to add expressiveness to DL, the basic Description Logic is extended carefully in order to preserve the good computational properties. By carefully I mean that not all extensions are



(a) Extending with feature X (b) Undecidability

Figure 2.4: Cartoons illustrating that extending DL with feature X would lead to undecidability. The cartoons are taken from [6, p. 21].

good. Some might be harmful and lead to undecidability which is illustrate through the cartoons in figure 2.4.

Extensions of $\mathcal{ALC}$ are indicated by additional letters. For instance the letter $\mathcal{N}$ stands for *number restrictions* (e.g., $\geqslant 2$ `hasChild`, $\leqslant 3$ `hasChild`). Adding this to $\mathcal{ALC}$, we get $\mathcal{ALCN}$ which is Attributive Language with Complements and Number restrictions. Often the letter $\mathcal{S}$ is used instead of $\mathcal{ALC}$ extended with transitive roles ($\mathcal{R}_+$) [6, p. 7]. Below is given a list of possible extensions to $\mathcal{ALC}$. This list is taken from [6, p. 7] and slightly modified.

- $\mathcal{H}$ for role hierarchy (e.g., `hasDaughter` $\sqsubseteq$ `hasChild`)
- $\mathcal{O}$ for nominals or singleton classes (e.g., {`ITALY`})
- $\mathcal{I}$ for inverse roles (e.g., `isChildOf` $\equiv$ `hasChild`$^-$)
- $\mathcal{Q}$ for qualified number restrictions (e.g., $\geqslant 2$ `hasChild.Doctor`)
- $\mathcal{F}$ for functional number restrictions (e.g., $\leqslant 1$ `hasMother`)

The complexity of reasoning in the various DLs resulting from extension of $\mathcal{ALC}$ can be found using the Complexity Navigator[1] of Evgeny Zolin from the University of Manchester.

The basis for W3C's OWL Web Ontology Language is the so called $\mathcal{SHIQ}$ DL. This acronym stands for $\mathcal{ALC}$ with transitive roles ($\mathcal{S}$), role hierarchy ($\mathcal{H}$),

[1]http://www.cs.man.ac.uk/ ezolin/dl/

inverse roles ($\mathcal{I}$) and qualified number restrictions ($\mathcal{Q}$). Extending $\mathcal{SHIQ}$ with nominals ($\mathcal{O}$) results in the so called $\mathcal{SHOIQ}$ which is the basis for the OWL DL. Likewise, we have $\mathcal{SHIF}$ which is $\mathcal{SHIQ}$ with functional restrictions instead of qualified number restriction. $\mathcal{SHIF}$ is the basis for the so called OWL Lite.

Chapter 3

Tableaux Algorithm

The DL reasoning algorithms, before the tableau-based algorithms, was the so called *structural subsumption algorithms*. These kind of algorithms compare the syntactic structure of concept descriptions. The problem with these algorithms, although usually being very efficient, is that they are only complete for languages that are not so expressive. For instance, these algorithms can not handle DLs with full negation and disjunction [4, p. 81].

With tableau-based algorithms which were first presented for satisfiability of $\mathcal{ALC}$-concepts, it is possible to have interesting hypotheses such as satisfiability, subsumption, equivalence and disjointness. These four kind of hypotheses can be reduced to only subsumption or unsatisfaibility [2, p. 11]. For instance testing that C is subsumed by D, i.e. $C \sqsubseteq D$, the formula is rewritten as $C \sqcap \neg D$ which is tested for satisfiability. This is possible since a concept C is subsumed by concept D iff $C \sqcap \neg D$ is empty. Testing $C \sqsubseteq D$ for satisfiability is not possible. In order to do it, as it is, one has to show that each concept in C is also in D.

3.1 Illustration of Tableau-based Algorithms

Before describing the actual Tableaux inference rules and how these are applied, I illustrate the underlying ideas behind the tableau-based algorithms. This is done using a simple example which is presented in [4] on pages 85-86.

Lets assume that we want to know whether

$$(\exists R.A) \sqcap (\exists R.B) \sqsubseteq \exists R.(A \sqcap B)$$

In order to find this out, the query is transformed and the following concept description is created:

$$C = (\exists R.A) \sqcap (\exists R.B) \sqcap \neg\exists R.(A \sqcap B)$$

Now, instead of checking for subsumption, above concept description is checked for unsatisfaibility. First, all the negation signs are pushed as far as possible into the description. This is done using De Morgan's rules and the usual rules for quantifiers. In case of C it means that we obtain:

$$C_0 = (\exists R.A) \sqcap (\exists R.B) \sqcap \forall R.(\neg A \sqcup \neg B)$$

The above concept description is now in *negation normal form.* This means that only concept names are negated.

Now, we construct a finite interpretation, $\mathcal{I}$, such that $C_0^{\mathcal{I}} \neq \emptyset$, i.e. there must exist an individual in $\Delta^{\mathcal{I}}$ (interpretation domain) that is an element of $C_0^{\mathcal{I}}$.

The tableaux algorithm just generates such an element, say b, and imposes the constraint $b \in C_0^{\mathcal{I}}$. This means:

$$b \in (\exists R.A)^{\mathcal{I}} \quad \text{and} \quad b \in (\exists R.B)^{\mathcal{I}} \quad \text{and} \quad b \in (\forall R.(\neg A \sqcup \neg B))^{\mathcal{I}}$$

From all three constraints, presented above, we can deduce the following. In the case of $b \in (\exists R.A)^{\mathcal{I}}$ there must exist an individual c such that $(b, c) \in R^{\mathcal{I}}$ and $c \in A^{\mathcal{I}}$. Analogously, in the case of $b \in (\exists R.B)^{\mathcal{I}}$ there must exist an individual d such that $(b, d) \in R^{\mathcal{I}}$ and $d \in B^{\mathcal{I}}$. The two individuals, c and d, are not the same since *"For any existential restriction the algorithm introduces a new individual as role filler, and this individual must satisfy the constraints expressed by the restriction."* [4, p. 86].

In the case of $b \in (\forall R.(\neg A \sqcup \neg B))^{\mathcal{I}}$ the individuals from the existential restrictions are used. This means:

$$c \in (\neg A \sqcup \neg B)^{\mathcal{I}} \quad \text{and} \quad d \in (\neg A \sqcup \neg B)^{\mathcal{I}}$$

In order to satisfy $c \in (\neg A \sqcup \neg B)^{\mathcal{I}}$ we can only choose $c \in (\neg B)^{\mathcal{I}}$ since $c \in (\neg A)^{\mathcal{I}}$ will be in conflict with $c \in A^{\mathcal{I}}$ from above.

The same way we can only choose $d \in (\neg A)^{\mathcal{I}}$ in order to satisfy $d \in (\neg A \sqcup \neg B)^{\mathcal{I}}$ since $d \in (\neg B)^{\mathcal{I}}$ will be in conflict with $d \in B^{\mathcal{I}}$ from above.

This concludes that C_0 is satisfiable since all constraints are satisfiable, i.e. $c \in A^{\mathcal{I}}$, $(b, c) \in R^{\mathcal{I}}$, $d \in B^{\mathcal{I}}$, $(b, d) \in R^{\mathcal{I}}$, $c \in (\neg B)^{\mathcal{I}}$ and $d \in (\neg A)^{\mathcal{I}}$. This means that $(\exists R.A) \sqcap (\exists R.B)$ is not subsumed by $\exists R.(A \sqcap B)$, since the interpretation, $\mathcal{I}$, is a witness of it. Formally this interpretation is as shown below. This is what the tableaux algorithm generated.

$$\Delta^{\mathcal{I}} = \{b, c, d\};\ R^{\mathcal{I}} = \{(b, c), (b, d)\};\ A^{\mathcal{I}} = \{c\};\ B^{\mathcal{I}} = \{d\}$$

This also means that $b \in ((\exists R.A) \sqcap (\exists R.B))^{\mathcal{I}}$, but $b \notin (\exists R.(A \sqcap B))^{\mathcal{I}}$.

3.2 Overview of Tableaux

Before the tableaux algorithm can start working the DL hypothesis or query must undergo some transformation. These transformations are:

1. First the *goal* is constructed. This means that the query is reduced to unsatisfaibility. For instance if the query is $C \equiv D$ the constructed goal is then to show that both $(C \sqcap \neg D)$ and $(\neg C \sqcap D)$ are unsatisfiable.
2. Next, all concepts in the goal are unfolded, i.e. TBox-elimination. This means that all terminological axioms are unfolded such that the goal contains only base symbols, i.e. primitive concepts.
3. The goal is written in Negation Normal Form, i.e. all the negations are pushed inwards such that only primitive concepts are negated.

When the transformation steps are completed the tableaux algorithm can start working. It manipulates the goal by applying four kinds of rules; intersection, union, existential and universal elimination. These rules are shown in table 3.1 on the next page and further explained in the following section.

3.3 Tableaux Inference Rules

In this section I give an informal description of each tableaux inference rule presented in table 3.1 on the following page. These rules are for the basic DL, i.e. $\mathcal{ALC}$. Additional rules for the proof algorithm exists that take care

$\sqcap - rule$	*if* 1.	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup - rule$	*if* 1.	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C_2\} \cap \mathcal{L}(x) =$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
		a. save $\mathbf{T}$
		b. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1\}$
		If that leads to a clash then restore $\mathbf{T}$ and
		c. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_2\}$
$\exists - rule$	*if* 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \in \mathcal{L}(y)$
	then	create a new node y and edge $\langle x, y\rangle$
		with $\mathcal{L}(y) = C$ and $\mathcal{L}(\langle x, y\rangle) = R$
$\forall - rule$	*if* 1.	$\forall R.C \in \mathcal{L}(x)$
	2.	there is some y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \notin \mathcal{L}(y)$
	then	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup C$

Table 3.1: Tableaux Inference Rules for $\mathcal{ALC}$ [2, p. 14].

of the additional operators and constructors added to the language. For more information on this subject please refer to [1].

The notions C, C_1 and C_2 in table 3.1 denote arbitrary DL concepts. A relation is represented by R. The whole proof tree is represented by $\mathbf{T}$, while x and y denote specific nodes in the tree. The set of DL formulas associated with a node x is denote by $\mathcal{L}(x)$ (node label).

Each rule in table 3.1 consists of two parts, an precondition part (the *if* part) and an action part (the *then* part). The precondition part consists of two conditions that both must be satisfied in order for the action part to take effect. The first condition is always a check for the presence of the term in the node label, while the second one is usually a check for the result not already being presence in the node label.

The first rule (denoted $\sqcap$–rule) is applied to intersection terms within a given node label. The second condition makes sure that both of the operands are not already presence in the node label. In the action part, both operands are added as new members to the current node label.

The second rule (denoted $\sqcup$–rule) is applied to union terms within a given node label. The second condition of this rule makes sure that none of the operands are already in the node label. The action part consists of three steps. In the first step the whole tree (denoted **T**) is saved. In the second step, the first operand of the union term is added to the current node label and the proof procedure proceeds. If this instance of the tree lead to a clash then the tree is restored. In step three the second operand is added to the just restored tree and the proof procedure is proceeded.

The third rule (denoted ∃–rule) is applied to the existential restriction terms ($\exists R.C$) in a given node label. The second condition in this rule makes sure that a node with the same relation (R) as edge label and containing the constrained concept (C) does not already exist. If this is not the case, such a node is created.

The last rule in table 3.1 on the preceding page (denoted ∀–rule) is applied to the value restriction terms. In this rule, the second condition makes sure that a node having the relation as the edge label and not containing the constraining concept exist. If this is the case, the constraining concept is added to the node label of that node.

Example 3.1 *Here the above rules are applied using a simple example. For the example we use following expression as the hypothesis*

$$\{[\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)]\}$$

where { } delimitate a tree and [ ] delimitate a node label. To begin with we apply the $\sqcap$*-rule. As one can see, there are three* $\sqcap$*-terms. It is arbitrary which one to choose. Let us choose the one such that* C_1 *is*

$$\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D)$$

and C_2 *is*

$$\exists R.C \sqcap \forall R.(\exists R.C)$$

Now the second precondition being true, i.e. both C_1 *and* C_2 *are not member of the current node label, we can move on to the action part of the* $\sqcap$*-rule. As one can see the action part does not tell to discard the affected term, i.e.* $C_1 \sqcap C_2$*. We will, however, discard this term in order to have a clean node label. This will result in the following expression.*

$$\{[\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D), \exists R.C \sqcap \forall R.(\exists R.C)]\}$$

Likewise, we apply the $\sqcap$*-rule to the remaining two* $\sqcap$*-terms and end up with the following expression*

$$\{[\exists S.C, \forall S.(\neg C \sqcup \neg D), \exists R.C, \forall R.(\exists R.C)]\}$$

which does not have any $\sqcap/\sqcup$*-terms that can be eliminated. Next, we apply the* $\exists$*-rule to all possible* $\exists$*-terms. We have two such terms available in the above expression. Expanding them we will get*

$$\{[\exists S.C, \forall S.(\neg C \sqcup \neg D), \exists R.C, \forall R.(\exists R.C)], [\langle S, [C]\rangle], [\langle R, [C]\rangle]\}$$

where $\langle R, C\rangle$ represents the expansion of a $\exists$-term with R being the edge and C the new created node. Next, we expand all possible $\forall$-terms. There are two of such terms which result in

$$\{[\exists S.C, \forall S.(\neg C \sqcup \neg D), \exists R.C, \forall R.(\exists R.C)], [\langle S, [C, \neg C \sqcup \neg D]\rangle], [\langle R, [C, \exists R.C]\rangle]\}$$

which is the result of adding the concept of each $\forall$-term to the node having the role as the edge. The two new nodes that resulted from the expansion of $\exists$- and $\forall$-terms are taken as the current nodes, i.e. the nodes that the proof rules are applied to. We have following current nodes:

$$\{[C, \neg C \sqcup \neg D], [C, \exists R.C]\}$$

The same way as above, the proof rules are applied to the above nodes one by one. We apply the $\sqcup$-rule to the first node and get

$$\{[C, \neg C], [C, \exists R.C]\}_L$$
$$\{[C, \neg D], [C, \exists R.C]\}_R$$

where we have two trees labeled L (left) and R (right). We proceed with the left tree and immediately realise that there is a clash, C and $\neg C$, which closes this branch of the Tableaux. The rest of the proof is left for the reader to complete. We can already conclude that the hypothesis is unsatisfiable since we already had a clash on one of the branches of the Tableaux.

3.4 Properties of Tableaux

According to Herchenröder [2, p. 16], the tableaux algorithm presented in table 3.1 on page 20 is sound and complete. It will also always terminate using exponential time and space complexity.

The *termination* property is derived from the fact that the algorithm only adds subexpressions that are strictly smaller than the parent expressions. There is always a limited number of these expressions, since the initial expression is finite. The branching and depth of the proof tree is also limited, since this depends on the number of existential in the initial formula, which again is limited. Moreover, the rules conditions are there to prevent loops in expanding terms [2, p. 16].

According to Herchenröder, *soundness* and *completeness* rely on the equivalence of the satisfiability of the initial formula and the consistency of the saturated proof trees. This can be shown partly using a *canonical model* and taking advantage of the *finite tree model* property.

Regarding the *complexity* of the algorithm, it is exponential in time and space in worst-case [2, p. 16].

Chapter 4

Implementing the Tableaux Algorithm

In this chapter, I will present the design and implementation decisions made in [2] in order to implement a more efficient implementation with regards to memory consumption and runtime. These decisions are:

- Applying tableaux rules in order instead of randomly.
- Replacing terms by their rule derivatives instead of keeping both.
- Instead of keeping the whole proof tree only the fringe of the proof is kept during the proof procedure.
- When eliminating union ($\sqcup$) a choice point is made and the resulting trees are explored sequentially instead of concurrently.

These decisions are explained more thorough in the following.

4.1 Application of Rules

Although the original definition of the Tableaux algorithm does not suggest any thing about the order of the application of rules, it is of great importance and provides significant advantages in a concrete implementation. As shown by Baader and Sattler (cf. [2]) it reduces the worst-case space requirements of the algorithm to polynomial.

The modified algorithm stated by Herchenröder operates in iteration consisting of two steps. These steps are first applied to the goal, denoted $C_0(x_0)$. In step one the $\sqcap$- and $\sqcup$-rules are applied as long as possible and the node is checked for clashes. In the second step the algorithm generates all the direct successors of x_0 using the $\exists$-rule and thoroughly applies the $\forall$-rule to the corresponding role assertions. The algorithm continues by applying step one and two to the successors in the same way.

This section summarizes the discussion of Herchenröder regarding this approach. I have used his example in order to make the arguments obvious.

Example 4.1 *We start with the goal node*

$$\{(\exists R.A \sqcap B) \sqcap \forall R.C\}$$

and apply the $\sqcap$-rule (which is the only applicable rule at this stage). This results in

$$\{(\exists R.A \sqcap B), \forall R.C\}$$

In this stage the only applicable rule is again the $\sqcap$-rule. The $\forall$-rule is blocked until we apply the hidden $\exists$-rule inside the $\sqcap$-rule. Applying the $\sqcap$-rule results in

$$\{\exists R.A, B, \forall R.C\}$$

Now it is possible to apply the $\exists$-rule and successively apply the $\forall$-rule which results in a new child node with two constraints.

$$\{\exists R.A, B, \forall R.C\}, \{A, C\}$$

As illustrated in example 4.1 it is suitable to expand all possible $\sqcap$- and $\sqcup$-terms in a node. This makes the hidden $\exists$-terms available for quantifier elimination.

The $\forall$-terms are always blocked and only available when the preconditions for the $\forall$-rule are met. This happens when the corresponding $\exists$-terms are expanded and new edges are created. Consequently, an exhaustive application of $\exists$-rule guarantees the availability of maximum number of edges for succeeding $\forall$-expansions.

$\sqcap - rule$	*if*	1.	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$
		2.	$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
	then		$\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcap C_2)) \cup \{C_1, C_2\}$
$\sqcup - rule$	*if*	1.	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$
		2.	$\{C_1, C_2\} \cap \mathcal{L}(x) =$
	then		$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
			a. save $\mathbf{T}$
			b. try $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcup C_2)) \cup \{C_1\}$
			If that leads to a clash then restore $\mathbf{T}$ and
			c. try $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcup C_2)) \cup \{C_2\}$
$\exists - rule$	*if*	1.	$\exists R.C \in \mathcal{L}(x)$
		2.	there is no y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \in \mathcal{L}(y)$
	then		create a new node y and edge $\langle x, y \rangle$
			with $\mathcal{L}(y) = C$ and $\mathcal{L}(\langle x, y \rangle) = R$ and
			$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \exists R.C$
$\forall - rule$	*if*	1.	$\forall R.C \in \mathcal{L}(x)$
		2.	there is some y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \notin \mathcal{L}(y)$
	then		$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup C$ for every applicable y and
			$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \forall R.C$

Table 4.1: Modified Tableaux Inference Rules for $\mathcal{ALC}$. Taken from [2, p. 14].

4.2 Parsimonious Rules

The ordered and exhaustive application of the tableaux inference rules to the DL expressions of a node label as discussed above makes it possible to deduce the so called *Parsimonious Rules.* According to Herchenröder these rules makes it possible to *kill to birds with one stone.* This means that parsimonious rules not only remove the redundant information from the proof tree and only keep the derivative terms, but also makes some of the preconditions for the original rules superfluous.

In this section I present the parsimonious rules derived by Herchenröder in [2, p. 24-29].

4.2.1 Replacement of Expressions by their Derivatives

In the standard tableaux algorithm the derivatives of the expressions are just added to the corresponding node label and new child nodes are added to the proof tree. This is shown in example 4.2 on the following page (cf. [2, p. 24]).

Example 4.2 *We begin with the initial DL expression that form the initial single member of the set, say*

$$\{A \sqcap (B \sqcap (C \sqcap D))\} \tag{4.1}$$

We apply the $\sqcap$-rule and get the set

$$\{A, (B \sqcap (C \sqcap D)), A \sqcap (B \sqcap (C \sqcap D))\}$$

that contains the operands of the intersection in addition to the initial member. Now we apply the $\sqcap$-rule to the second member of the above set. Notice that it is not possible to apply the $\sqcap$-rule to the initial expression any more. This is because of the precondition checking the presence of the operands (cf. table 3.1 on page 20). The resulting set is

$$\{A, B, (C \sqcap D), (B \sqcap (C \sqcap D)), A \sqcap (B \sqcap (C \sqcap D))\}$$

We proceed the same way as above and apply the $\sqcap$-rule, this time to the third element in the above set.

$$\{A, B, C, D, (C \sqcap D), (B \sqcap (C \sqcap D)), A \sqcap (B \sqcap (C \sqcap D))\}$$

No further rules are applicable since the proof tree is saturated.

As illustrated in example 4.2 the single proof node beside containing the fully expanded members (A, B, C and D) also contains the initial expression and all the intermediate expressions created during the rules application. According to Herchenröder the later ones “do not contribute any further to the decision procedure” [2, p. 25]. Furthermore, he claims that it is safe to discard these by replacing them with their derivatives. Doing so prevents one from checking these expressions over and over again in order to satisfy the second precondition of the $\sqcap$-, $\sqcup$- and $\exists$-rule and the second part of the second precondition of the $\forall$-rule. These preconditions validate the termination of the algorithm by helping to avoid expanding the same compound term over and over again.

Pruning the parent expressions has significant improvements on the implementation. It both saves memory footprint of the procedure and computing time [2, p. 26].

4.2.2 Derivation of the New Rules

The new rules (parsimonious rules) are simply derived from the original rules by replacing the complex expressions with their derivatives. For instance (4.1) in example 4.2 results in

$$\{A, (B \sqcap (C \sqcap D))\}$$

after applying the $\sqcap$-rule.

Formally this is done as follows (cf. [2, p. 25])

$$\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcap C_2)) \cup \{C_1, C_2\}$$

which means that the parent term is removed from the node label (expressed by the "$\setminus$" operator) before the derivative of a rule application is added to the node label (expressed by the "$\cup$" operator). Table 4.1 on page 27 presents the complete list of the modified rules.

Now the question is whether pruning of the parent expressions alter the semantics of the node label? Moreover, whether it renders the node empty? According to Herchenröder neither is the case [2, p. 27-29]. In the following I will present a summary of his argumentation.

In the case of $\sqcap$-expansion pruning the parent expression does not change the semantics of the node label. This is due do the equivalence of sets $\{A, B\}$ and $\{A \sqcap B\}$ semantically. In the set $\{A, B, A \sqcap B\}$ the parent expression $(A \sqcap B)$ just restates the requirement that individuals must satisfy A and B and is therefore redundant. The node label is not possible to render empty after removing the parent term, since the set will contain the operands of the term after the rule application.

The same holds in the case of $\sqcup$-expansion. Here the tree is duplicated and each operand of the complex $\sqcup$-term is added to the node label in each of the trees. This preserves fully the semantics of the $\sqcup$-term, since it is only satisfiable if either there exists a model fulfilling the tree with the first operand, or a model for the tree with second operand. Put in other words, the term does not add a new constraint to the respective node label, and is therefore redundant and can be discarded. The node label can not be rendered empty, since there will exists an operand after the rule application.

In the case of $\exists$-terms a new child node, containing the concept name, is created and new edge to it, with the name of the relation as the label, is added. This fully preserves the initial semantics of the $\exists$-term. Keeping the term is only retaining redundant information. While removing the term will result in prohibition of further expansion of it. It is also possible to render the node label empty by removing the term, but it is harmless, since the child node contains the relevant information in order to continue the decision procedure.

The same holds in the case of $\forall$-terms ($\forall R.C$). There is a possibility to render the node label empty by pruning the term, but it is harmless because of the same reason as mentioned above. It can, however, be damaging to the proof to prune the term. It is due to the fact that a $\forall$-term can be evaluated once for each existing child node with matching relation (R) edge and non-existent concept C. Therefore, we must make sure that it is maximally applied and then prune it from the node label.

4.3 Keeping the Fringe of the Proof

Analogous to complex DL expressions which can be pruned from the node label, once the rule is applied, the whole node can also be discarded, once it has been exhaustively transformed and expanded, checked for clashes and its all possible child nodes have been derived with their labels fully expanded. Therefore, it is sufficient only to keep track of the current set of leaf nodes, or better known as *fringe* [2, p. 30], instead of always maintaining the whole tree.

According to Herchenröder, keeping only the fringe of the proof tree provides some advantages without having any negative effects [2, p. 30]. One of these advantages is the need for less memory.

He also mentions that parent node must be fully expanded before it can be discarded. The expansion is both in terms of child expansion and clash detection.

4.4 Handling OR Trees

One has to make non-deterministic choice every time a $\sqcup$-connective has to be expanded. This leaves room for implementation variants. In his implementation, Herchenröder has chosen the sequential version of exploring $\sqcup$-trees. This is, every time a $\sqcup$-term has to be expanded, the proof tree containing the first operand of the term is explored until it fails. On failure, the proof tree containing the second operand of the term is explored in order to get a model.

An alternative way, is exploring the trees concurrently. This is, exploring the proof tree with the first operand and its duplicates containing the second operand simultaneously.

According to Herchenröder , the sequential version not only saves runtime memory, but is also a very attractive solution when developing with Prolog, since it fits naturally into Prolog's backtracking strategy.

Chapter 5

Specification of the Implemented Tableaux

In this chapter, I summarize the specification of the *tableaux.pl*, the implementation of tableaux algorithm proposed by Herchenröder. This chapter is basically the summary of chapter 5 in [2] with the addition of the running example that I have constructed by myself.

As mentioned in chapter 3 on page 17, the DL query must undergo some transformation before the actual Tableaux algorithm can start working. The first step in the transformation process is the construction of the goal [2, p. 34]. The complex DL concepts in the goal are replaced by their definition in order to construct an expression that only contains atomic concepts. This step of transformation is called *concept unfolding* [2, p. 35]. The resulting DL expression is now transformed into so called *Negative Normal Form* where the negations are pushed inwards until only the atomic concepts are negated. The actual proof starts with the result of this transformation by first eliminating the $\sqcap$- and $\sqcup$-connectives and then checking for clashes. Next, all possible $\exists$-terms are expanded and the corresponding (with the same relation) $\forall$-terms are applied. The process is then repeated for the resulting child nodes.

In the following sections I describe the above process and the specification of the *tableaux.pl* thoroughly and with use of a running example. This example is constructed by querying the ontology (TBox) shown in table 2.3 on page 12. The specific query is

$$\texttt{MotherWithoutDaughter} \sqsubseteq \texttt{Mother} \tag{5.1}$$

which means that the concept of `MotherWithoutDaughter` is subsumed by the concept of `Mother`, i.e. the concept of `Mother` is a more general concept then the concept of `MotherWithoutDaughter`.

In order to list the specification of certain predicates, I use the notation used by Herchenröder in his description. The capital letters $A, B, C, \ldots$ denote well-formed DL concept expressions. On the left of the arrow ($\leftarrow$) are goals that must be satisfied, while on the right of it are the subgoals that will allow one to derive the main goal if they are satisfiable. Predicates with two parameters, e.g. $\mathit{nnf}(A, B)$, can be understood as taking the first parameter (A) as input and constructing the second one (B) as output.

5.1 Goal Construction

As mentioned above, the first step of the transformation process is the construction of the goal from the initial query. In *tableaux.pl* it is done using the predicate $\mathit{query}/1$.

$$\begin{array}{ll}
\mathit{query}(A \equiv B) & \leftarrow \neg \mathit{tableaux}(A \sqcap \neg B) \wedge \neg \mathit{tableaux}(B \sqcap \neg A) \\
\mathit{query}(A \sqsubseteq B) & \leftarrow \neg \mathit{tableaux}(A \sqcap \neg B) \\
\mathit{query}(A \sqcap B \sqsubseteq \bot) & \leftarrow \neg \mathit{tableaux}(A \sqcap B) \\
\mathit{query}(\mathit{unsatisfiable}(A)) & \leftarrow \neg \mathit{tableaux}(A) \\
\mathit{query}(A) & \leftarrow \mathit{tableaux}(A)
\end{array}$$

If this predicate succeeds, it means that the query is satisfiable, otherwise it is unsatisfiable. The first clause matches equivalence of two concepts A and B. The two are equivalent if the tableaux proof can show that $A \sqcap \neg B$ and $B \sqcap \neg A$ are not satisfiable. The same way, goals for queries regarding subsumption, disjointness, unsatisfaibility and satisfiability are constructed.

The *tableaux/1* predicate is the basic tableaux proof predicate that is defined as follows:

$$\begin{array}{lll}
\mathit{tableaux}(A) & \leftarrow & \mathit{expand\_defs}(A, A1) \wedge \\
 & & \mathit{nnf}(A1, A2) \wedge \\
 & & \mathit{expand\_tree}(A2)
\end{array}$$

The three subgoals ($expand\_defs(A, A1)$, $nnf(A1, A2)$ and $expand\_tree(A2)$) which this predicate consists of, are the transformation of the goal and the actual proof procedure. Concept unfolding is done by the predicate $expand\_defs/2$, while transforming the goal to Negative Normal Form is done by the predicate $nnf/2$. The $expand\_tree/1$ predicate is the actual proof procedure which transforms and expands the initial node by well-defined proof steps. The goal fails if possible clashes are inspected during the expansion of the tree. If no clashes are detected when the tree is fully expanded, the goal succeeds.

Example 5.1 *This example is the first example in a series of examples illustrating the specification of the **tableaux.pl**. In this example we construct the goal for (5.1). The goal becomes*

$$\texttt{MotherWithoutDaughter} \sqcap \neg\texttt{Mother} \tag{5.2}$$

*according to the second clause of **query/1**. If the tableaux proof can show that the above expression is not satisfiable then (5.1) is true, otherwise it is false.*

In the following section we proceed with concept unfolding or expansion of the goal expression.

5.2 Concept Unfolding

In this step of the transformation all the so called *name concepts* that are defined in terms of other concepts are replaced by their definition. For instance the name concept `Mother` in the goal constructed in example 5.1 is replaced by its definition

$$\texttt{Woman} \sqcap \exists\texttt{hasChild.Person} \tag{5.3}$$

which is given in table 2.3 on page 12. In *tableaux.pl* this is done using the $expand\_defs/2$ predicate which is defined as:

$$\begin{aligned}
expand\_defs(A, A1) &\leftarrow atomic(A) \wedge ont(equiv(A, A2)) \wedge expand\_defs(A2, A1)\\
expand\_defs(A, A1) &\leftarrow atomic(A) \wedge \neg ont(equive(A, \_))
\end{aligned}$$

The definition states that each atomic concept A in the ontology is replaced by the expression in the right side of the $\equiv$ ($equiv/2$). Concept names which do not have a definition in the ontology are returned as primitives. This is applied recursively until the whole expression is unfolded.

Example 5.2 *This example is the second example in the series of examples illustrating the specification of the **tableaux.pl**. In this example we unfold the name concepts in (5.2) from example 5.1.*

We are concerned with two name concepts. These are `MotherWithoutDaughter` *and* `Mother`*. If we begin with the second name concept and replace it with its definition, we will get* (5.3) *which contains a name concept,* `Woman`*. Replacing it will result in the fully expanded DL concept that we called* `Mother`*.*

The concept `MotherWithoutDaughter` *can be expanded in the same way. The goal will then become a very long expression which is not suited to be presented here. Therefore, only the first letter of each concept (*`P` *for* `Person` *and* `F` *for* `Female`*) is presented and the role* `hasChild` *is replaced by the letter R. The fully expanded goal is than given as follows:*

$$((\mathtt{P} \sqcap \mathtt{F}) \sqcap \exists \mathtt{R}.\mathtt{P} \sqcap \forall \mathtt{R}.(\neg(\mathtt{P} \sqcap \mathtt{F}))) \sqcap \neg((\mathtt{P} \sqcap \mathtt{F}) \sqcap \exists \mathtt{R}.\mathtt{P}) \tag{5.4}$$

In the following section we proceed by pushing all the negation in the goal inwards such that only primitive concepts are negated.

5.3 Negative Normal Form

In order for the tableaux proof procedure to begin, the goal must undergo one last transformation. All the negations must be pushed inside such that only primitive concepts are negated. After this transformation the goal will be in negation normal form. In the *tableaux.pl* this transformation is done by the predicate *nnf*/2 which is defined as follows:

$$\begin{array}{ll}
nnf(\neg\neg C, C) & \\
nnf(\neg\forall R.C, \exists R.C1) & \leftarrow nnf(\neg C, C1) \\
nnf(\forall R.C, \forall R.C1) & \leftarrow nnf(C, C1) \\
nnf(\neg\exists R.C, \forall R.C1) & \leftarrow nnf(\neg C, C1) \\
nnf(\exists R.C, \exists R.C1) & \leftarrow nnf(C, C1) \\
nnf(\neg(A \sqcap B), A1 \sqcup B1) & \leftarrow nnf(\neg A, A1) \wedge nnf(\neg B, B1) \\
nnf(A \sqcap B, A1 \sqcap B1) & \leftarrow nnf(A, A1) \wedge nnf(B, B1) \\
nnf(\neg(A \sqcup B), A1 \sqcap B1) & \leftarrow nnf(\neg A, A1) \wedge nnf(\neg B, B1) \\
nnf(A \sqcup B, A1 \sqcup B1) & \leftarrow nnf(A, A1) \wedge nnf(B, B1) \\
nnf(\neg C, \neg C) & \leftarrow atomic(C) \\
nnf(C, C) & \leftarrow atomic(C)
\end{array}$$

As one can see the above clauses apply the rules such as $\neg\neg C \equiv C$, $\neg\forall R.C \equiv \exists R.\neg C$, De Morgan's laws etc. These rules are applied recursively until only the primitive concepts are negated.

Example 5.3 *This example is the third example in the series of examples illustrating the specification of the* ***tableaux.pl****. In this example we transform* (5.4) *from example 5.2 on page 33 into negative normal form.*

There are two concepts that are negated in (5.4)*. These are* $\neg(\mathtt{P} \sqcap \mathtt{F})$ *and* $\neg((\mathtt{P} \sqcap \mathtt{F}) \sqcap \exists \mathtt{R.P})$*. The first concept is transformed into negative normal form using De Morgan's rules and we get* $(\neg\mathtt{P} \sqcup \neg\mathtt{F})$*. The second concept is more complex than the first one. Here we must apply De Morgan's twice and beside that also the rule* $\neg\exists R.C \equiv \forall R.\neg C$*. This results in* $((\neg\mathtt{P} \sqcup \neg\mathtt{F}) \sqcup \forall\mathtt{R.}(\neg\mathtt{P}))$*. The goal is now transformed into negative normal form and looks like shown below.*

$$((\mathtt{P} \sqcap \mathtt{F}) \sqcap \exists\mathtt{R.P} \sqcap \forall\mathtt{R.}(\neg\mathtt{P} \sqcup \neg\mathtt{F})) \sqcap ((\neg\mathtt{P} \sqcup \neg\mathtt{F}) \sqcup \forall\mathtt{R.}(\neg\mathtt{P})) \tag{5.5}$$

It is now ready to be processed by the tableaux proof procedure.

In the following section we proceed with the actual tableaux proof procedure where the goal is expanded/transformed using the tableaux inference rules.

5.4 Expanding the Proof Tree

When the goal has been transformed such that all concepts names are replaced by their most basic definitions and all negations are pushed inwards such that only primitive concepts are negated, the tableaux proof procedure starts working. The procedure expands the tree in a standard agenda-style tree expansion [2, p. 36] where leaf nodes are examined, transformed ($\sqcap$- and $\sqcup$-elimination) or expanded ($\exists$- and $\forall$-expansion) and the resulting node(s) are put back into the list of current fringe nodes for further examination [2, p. 36].

In the *tableaux.pl* this process starts with the basic predicate, *process_node*/2, that is defined as follows:

$$\begin{aligned} process\_node(A, B) \leftarrow \quad & transform\_connectives(A, A1) \wedge \\ & expand\_node(A1, B) \end{aligned}$$

This predicate works on a single node from the proof tree. First all the $\sqcap$/$\sqcup$-connectives are eliminated using the *transform_connectives*/2 and next the transformed node is expanded by possible $\exists$/$\forall$-expansion using the *expand_node*/2 predicate.

The *transform_connectives*/2 predicate basically applies the $\sqcap$- and $\sqcup$-rule recursively to all $\sqcap$- and $\sqcup$-terms. This predicate is defined as follows:

$$\begin{aligned}
&\mathit{transform\_connectives}([A \sqcap B|R], R1) \leftarrow \\
&\qquad \mathit{transform\_connectives}([A, B|R], R1) \\
&\mathit{transform\_connectives}([A \sqcup B|R], R1) \leftarrow \\
&\qquad \mathit{transform\_connectives}([A|R], R1) \wedge \\
&\qquad \mathit{transform\_connectives}([B|R], R1)
\end{aligned}$$

The *expand_node*/2 predicate first checks for possible clashes in the just transformed node. Its core is, however, the expansion of the ∃ and ∀ quantifiers. The definition of this predicate is given below.

$$\begin{aligned}
\mathit{expand\_node}(N, \_) &\leftarrow \mathit{check\_clash}(N) \\
\mathit{expand\_node}(N, NL) &\leftarrow \mathit{expand\_exist}(N, [], E) \wedge \\
&\qquad \mathit{expand\_univ}(N, E, NL)
\end{aligned}$$

Example 5.4 *This example is the fourth example in the series of examples illustrating the specification of the* ***tableaux.pl****. In this example we eliminate all the possible ⊓- and ⊔-connectives in* (5.5) *from example 5.3 on page 34. The initial tree where the proof procedure starts is presented below.*

$$\big\{[((\mathtt{P} \sqcap \mathtt{F}) \sqcap \exists \mathtt{R}.\mathtt{P} \sqcap \forall \mathtt{R}.(\neg \mathtt{P} \sqcup \neg \mathtt{F})) \sqcap ((\neg \mathtt{P} \sqcup \neg \mathtt{F}) \sqcup \forall \mathtt{R}.(\neg \mathtt{P}))]\big\}$$

where { } and [ ] delimit trees and nodes in a tree, respectively. In the above tree there is only one node. This node has four ⊓-terms which all can be eliminated. The result is

$$\big\{[\mathtt{P}, \mathtt{F}, \exists \mathtt{R}.\mathtt{P}, \forall \mathtt{R}.(\neg \mathtt{P} \sqcup \neg \mathtt{F}), ((\neg \mathtt{P} \sqcup \neg \mathtt{F}) \sqcup \forall \mathtt{R}.(\neg \mathtt{P}))]\big\}$$

which can be further transformed by eliminating the possible ⊔-connectives. In the above node there are two of such connectives. The elimination of first of these connectives result in the following:

$$\begin{aligned}
&\big\{[\mathtt{P}, \mathtt{F}, \exists \mathtt{R}.\mathtt{P}, \forall \mathtt{R}.(\neg \mathtt{P} \sqcup \neg \mathtt{F}), (\neg \mathtt{P} \sqcup \neg \mathtt{F})]\big\}_L \\
&\big\{[\mathtt{P}, \mathtt{F}, \exists \mathtt{R}.\mathtt{P}, \forall \mathtt{R}.(\neg \mathtt{P} \sqcup \neg \mathtt{F}), \forall \mathtt{R}.(\neg \mathtt{P})]\big\}_R
\end{aligned} \tag{5.6a}$$

The whole tree is duplicated and the ⊔-term is pruned from the only node in the tree and the left operand of the ⊔-term is added to the first copy of the tree constructing the left tree (denoted by L), while the right operand is added to the second copy of the tree constructing the right tree (denoted by R). It is here a choice point is made and the proof procedure is proceeded using one of the trees and upon failure the alternative is tried. In his implementation Herchenröder proceeds with the left tree first and on failure the right tree is processed. We will proceed the same way.

The left tree is now transformed the same way as the initial tree— the ⊓/⊔-connectives are eliminated. In the left tree there is only one node which consists *of only one $\sqcup$-term that can be transformed— the other $\sqcup$-term is in the scope of the value restriction and can only be transformed after the expansion of the $\forall$. We get again two trees:*

$$\{[\mathtt{P}, \mathtt{F}, \exists\mathtt{R}.\mathtt{P}, \forall\mathtt{R}.(\neg\mathtt{P} \sqcup \neg\mathtt{F}), \neg\mathtt{P}]\}_{LL} \tag{5.7a}$$

$$\{[\mathtt{P}, \mathtt{F}, \exists\mathtt{R}.\mathtt{P}, \forall\mathtt{R}.(\neg\mathtt{P} \sqcup \neg\mathtt{F}), \neg\mathtt{F}]\}_{LR} \tag{5.7b}$$

Likewise, we have a choice point here. But here we notice that both nodes in the two trees contains clashes (`P`*,* $\neg$`P` *in* (5.7a) *and* `F`*,* $\neg$`F` *in* (5.7b)*) and thus both are closed.*

We return back to the first choice point and proceed with the right tree (see (5.6a)*). This is done in example 5.5 on the following page.*

The $expand\_exist/3$ and $expand\_univ/3$ predicates are, in my opinion, the most complex of them all. The first one is defined in the following:

$$\begin{aligned}
expand\_exist([\exists R.C|L], E, EE) &\leftarrow \neg member((R, \_, \_), E) \wedge id(I) \wedge \\
&\quad\ expand\_exist(L, [(R, I, [C])|E], EE) \\
expand\_exist([\exists R.C|L], E, EE) &\leftarrow member((R, \_, L1), E) \wedge \\
&\quad\ \neg member(C, L1) \wedge \\
&\quad\ id(I) \wedge expand\_exist(L, [], EE) \\
expand\_exist([], E, E) &
\end{aligned}$$

For every $\exists$-term in the list of terms, it creates a new node (e.g. $(R, I, [C])$) and connects the parent and the child node by an edge which is attributed with the relation in that term (R) and gets a unique identifier (I) in order to distinguish different child nodes that are connected through the same relation. This is only done if a child node with the same edge and having the same concept does not exist. In the case of no $\exists$-terms, the current node is returned. This will be the final leaf node and will ascertain an open branch since it is already checked for clashes.

When all the possible $\exists$-terms are expanded the $expand\_univ/3$ predicate will proceed by applying all the possible $\forall$-terms. This predicate is defined below.

$$\begin{aligned}
expand\_univ([\forall R.C|L], E, N) &\leftarrow expand\_univ\_exhaust((R, C), E, EE) \wedge \\
&\quad\ expand\_univ(L, EE, N) \\
expand\_univ([], E, N) &\leftarrow extract\_nodes\_from\_edges(E, N)
\end{aligned}$$

In order for the $\forall$-terms (e.g. $\forall R.C$) to be expanded, there must exist a node with the same label R. When this is the case the terms is transformed by adding the concept C to the node. The predicate, *expand_univ*/3, does this using a helper predicate, *expand_univ_exhaust*/3, which applies the given ∀-term **exhaustively** to the list of child nodes [2, p. 38]. This predicate is defined below.

```
$expand\_univ\_exhaust((R,C),E,E) \leftarrow$
        $extract((R,I,L1),E,E1) \wedge \neg member() \wedge$
        $expand\_univ\_exhaust((R,C),[(R,I,[C|L1])|E1],EE)$
$expand\_univ\_exhaust(\_,E,E)$
```

The *expand_predicate*/3 predicate, unlike, the *expand_exist*/3 returns the list of the child nodes without attributes when every ∀-term has been exhaustively applied to the list of child nodes. This is clarified in example 5.5. Moreover, this examples gives a very clear picture of how the two predicates operate.

Example 5.5 *This example is the last example in the series of examples illustrating the specification of the* ***tableaux.pl****. In this example we proceed with the right tree,* (5.6a)*, from example 5.4 on page 36.*

This tree consists of a single node which does not have any ⊓/⊔-connectives that can be eliminated. We, therefore, start with the expansion of the quantifiers. The result is:

$$\left\{[\mathtt{P},\mathtt{F},\forall\mathtt{R}.(\neg\mathtt{P}\sqcup\neg\mathtt{F}),\forall\mathtt{R}.(\neg\mathtt{P})],[(\mathtt{R},1,[\mathtt{P}])]\right\}$$

*The ∃-term, ∃*R.P*, is pruned from the parent node and a new child node,* (R, 1, [P])*, attributed with R is instead created. Now, if possible, the ∀-terms are expanded.*

It seems that both ∀-terms can be expanded since they scope over the same relation as the ∃-term expanded above, i.e. the relation R *is shared by both terms. The two terms are pruned from the parent node and their concepts are added to the child node. The result is then as shown below.*

$$\left\{[\mathtt{P},\mathtt{F}],[(\mathtt{R},1,[(\neg\mathtt{P}\sqcup\neg\mathtt{F}),\neg\mathtt{P},\mathtt{P}])]\right\}$$

Now the parent node, [P, F]*, is discarded, since it does not contribute any more to the proof procedure. The proof procedure proceeds with the new node instead (without the attributes). This node is*

$$\left\{[(\neg\mathtt{P}\sqcup\neg\mathtt{F}),\neg\mathtt{P},\mathtt{P}]\right\}$$

In this point in the proof procedure the node is transformed by eliminating the ⊓/⊔-connectives. Although, there is an obvious clash in the node, it is not discovered until all the connectives are eliminated. In the above node we have only one ⊔-connective which is eliminated. The result is

$$\left\{[\neg\mathtt{P},\neg\mathtt{P},\mathtt{P}]\right\}_{RL}$$
$$\left\{[\neg\mathtt{F},\neg\mathtt{P},\mathtt{P}]\right\}_{RR}$$

which gives two trees with an immediate clash in both. This closes also the right tree from the initial node. Thus, the tableaux is saturated and all trees are closed. This means that there is no model for the proof goal. The answer to (5.1) is 'Yes', i.e. the concept of `Mother` *is a more general concept than* `MotherWithoutDaughter`*.*

Chapter 6

Extending the Tableaux Implementation

In his conclusion, Herchenröder lists a number of possible ways to extend the tableaux implementation, *tableaux.pl*, put forth by him. The extensions put forth in this work are close to two of the extensions suggested by him. Since the focus of this work is to develop resources for use in education the development of the extensions was also focused in this direction.

In this chapter I present the two extensions I have developed for the *tableaux.pl*. Moreover I will also cover testing of these extensions here.

6.1 Ontology Format Converter

In his work, Herchenröder supports a specific format for expressing $\mathcal{ALC}$ expressions in Prolog. For instance (5.4) from example 5.2 on page 33 is expressed as shown below (negation is expressed using tilde).

```
and(and(and(and(P, F), exist(R, P)), forall(R, ~and(P, F))),
    ~and(and(P, F), exist(R, P)))
```

As one can see, this format is very difficult for humans to read. Moreover it is very easy to make mistakes when constructing complex expressions. In order to make this format more human-friendly, I have developed another format and a converter that can convert an expression between the two formats. For instance expressing the above expression in this format we will get:

```
((P /\ F) /\ R?P /\ R!(~(P /\ F))) /\ ~((P /\ F) /\ R?P)
```

where tilde still expresses negation, the forward- and backslash (/\) expresses conjunction, the question mark (`?`) expresses existential restriction, while exclamation mark (`!`) expresses value restriction. The operator that is missing in above expression is disjunction. This operator is expressed by a back- and forwardslash (\/).

In the following section the specification of this format is given. Moreover the Ontology Format Converter (OFC) that is developed to covert between the two formats is also specified and described.

6.1.1 Specification

The ontology format which is developed by Herchenröder (from now on *internal* format) enables one to express also axioms and hypotheses (queries) and not only concept description. The ontology format which is developed by me (from now on *external* format) only supports expressing of concept description, i.e. concepts that are constructed using negation, union, intersection, existential and value restriction or the combination of these.

The external format is very close to the syntax of the $\mathcal{ALC}$. For instance the expression $A \sqcap B$ is just expressed as it is, just by changing the operator to /\. Expressing the quantifier, however, differs a big deal. These are actually also implemented as binary operators. The normal term $\exists R.C$ is expressed as $R?C$ where the left side of the ? is the role and the right side is the concept. Likewise, the value restriction is implemented as binary operator. For instance the term $\forall R.C$ is expressed as $R!C$.

In order to remember that exclamation mark is the symbol for value restriction and question mark is the symbol for existential restriction in this format, one can think of existential quantifier as a question, i.e. is there exist a x such that ....

A predicate, *symbol_operator*/2, is used in order to map the operators in the internal and external formats. This predicate is defined as presented below.

$symbol\_operator(!, forall)$
$symbol\_operator(?, exist)$
$symbol\_operator(\backslash/, or)$
$symbol\_operator(/\backslash, and)$
$symbol\_operator(\sim, \sim)$

In addition to this predicate, there are two more predicates which converts an expression in internal format to the equal expression in external format and vise versa. These predicates are *to_external*/2 and *to_internal*/2. As the names suggest, the *to_external*/2 predicate takes an expression in internal format and returns it in external format. Likewise, the *to_internal*/2 predicate takes an expression in external format an returns it in internal format.

The *to_external*/2 predicate consists of three clauses. These are defined in the following.

```
$to\_external(Int, Ext) \leftarrow$
        $Int =.. [Opr, X1, X2] \wedge$
        $symbol\_operator(Sym, Opr) \wedge$
        $to\_external(X1, Y1) \wedge$
        $to\_external(X2, Y2) \wedge$
        $Ext =.. [Sym, Y1, Y2]$
$to\_external(Int, Ext) \leftarrow$
        $Int =.. [Opr, X] \wedge$
        $symbol\_operator(Sym, Opr) \wedge$
        $to\_external(X, Y) \wedge$
        $Ext =.. [Sym, Y]$
$to\_external(X, X)$
```

The two of the three clauses each convert binary and unary operators. The third claus is the basis case, i.e. returns the atomic concept. Looking at the first claus, one can see that it converts binary operators. The $Int =.. [Opr, X1, X2]$ term extracts the operator (Opr) and operands ($X1$ and $X2$) of an expression. The corresponding operator in external format is mapped to this operator using the *symbol_operator*/2 predicate. Recursively, the operands are converted to the external format and the expression is constructed using the term $Ext = .. [Sym, Y1, Y2]$. The second claus works in the same way as the first one, but only converting the unary operators.

For converting from external format to internal format the predicate, *to_internal*/2, is used. This predicate is defined exactly the same way as the *to_external*/2 predicate. The definition of it is presented in the following.

```
$to\_internal(Ext, Int) \quad \leftarrow$
        $Ext =.. [Sym, X1, X2] \wedge$
        $symbol\_operator(Sym, Opr) \wedge$
        $to\_external(X1, Y1) \wedge$
        $to\_external(X2, Y2) \wedge$
        $Int =.. [Opr, Y1, Y2]$
$to\_Internal(Ext, Int) \quad \leftarrow$
        $Ext =.. [Sym, X] \wedge$
        $symbol\_operator(Sym, Opr) \wedge$
        $to\_external(X, Y) \wedge$
        $Int =.. [Opr, Y]$
$to\_internal(X, X)$
```

It might be difficult to see the differences between the two predicates since they seems to be exactly the same definitions. The similarity of these predicate is due to they being symmetrical.

6.2 Proof Constructor

The *tableaux.pl* only returns a *true/false* (*satisfiable/unsatisfiable*) result for any given query. It is not possible to explore the intermediate steps toward the final answer—beside using the Prolog *trace* functionality. A convenient way to enable the user to explore the intermediate steps in a proof (a proof constructor, PC) has a lot of benefits.

A more important benefit of PC, in my opinion, is the ability to help one in understanding the Tableaux algorithm and how it works. This is especially important for those who are in the learning process, i.e. students.

Another benefit which is close to the above one, is the ability to have a correct answer in order to check ones own solution against it and thus find the intermediate step where one makes mistake, i.e. like having a mentor. Again, it is students that can benefit from this functionality.

In the following section a specification of the developed PC has been given.

6.2.1 Specification

The basic idea behind the development of the PC was to take the *tableaux.pl* as the starting point and output the result of some intermediate steps that together constructed the proof. It turned out that this idea could be easily realised and resulted in *Xtableaux.pl*—the extended tableaux implementation. In the following I give a specification of this extended version.

The challenges in developing the *Xtableaux.pl* were to output the constructed proof in a convenient way and find those intermediate steps that could result in a more constructive proof. The first challenge was easily overcome by outputting the constructed proof to a file. For each constructed proof the *Xtableaux.pl* creates a file named $proofX$ where X is a an auto-generated number distinguishing the proofs.

To overcome the second challenge the *tableaux.pl* was thoroughly examined and understood. The intermediate steps that resulted in a more constructive proof were chosen to be those presented in the following.

Initial Expression The expression given to the system as the hypothesis.

The Goal The reduction of the initial expression to satisfiability/unsatisfaibility, i.e. for instance the reduction of $C \sqsubseteq D$ to $C \sqcap \neg D$.

Expanded Expression The unfolding of all the name concepts, i.e. TBox-elimination.

Negative Normal Form The result of putting the expanded expression in negative normal form, i.e. pushing all the negation in the expanded expression inwards until only atomic concepts are negated.

Tableaux Inference Rules The results of applying tableaux inference rules to the initial node label in the proof tree. These results are further divided into intermediate steps which are as follows.

- **$\sqcap/\sqcup$-elimination** The result of eliminating all possible $\sqcap/\sqcup$-connectives in a node label.
- **$\exists$-elimination** The result of expanding all the existential restrictions in a node label.
- **$\forall$-elimination** The result of expanding all possible value restrictions in a node label.

Final Result The result of the proof, i.e. whether the goal is satisfiable or unsatisfiable.

The above steps means some changes to some of the predicates in *tableaux.pl*. These changes are highlighted in Appendix B where the complete listing of *Xtableaux.pl* is also given.

6.3 Testing the Extensions

In order to ensure that the correctness of the *tableaux.pl* is intact after the implementation of *Xtableaux.pl* and the ontology format converter works as intended, some tests needed to be conducted. To this end, I have constructed two different test suites.

These two test suites follow the same structure. They consists of a driver program, the data sets and the programs to be tested. In the following each of the two test suites are described in more details.

6.3.1 Testing Ontology Format Converter

The first test suite that I will describe in this section is intended to test the correctness of the ontology format converter, *grammar.pl*.

The structure of this test suite can be summarized as follows:

grammar.pl The implementation of the ontology format converter.

ofc_tester.pl The driver program that runs automatically all the test cases.

data The test cases for this test suite.

6.3.1.1 Construction of Test Data

One of the most important elements in testing any kind of software is the construction of test data. A constructive test data will help in revealing more bugs. It is always better that test data has been constructed by someone else then the one that has developed the software.

For this test suite, the test data has been partly taken from the Extended Mindswap Test in Herchenröder's work [2, p. 69] and partly has been constructed by myself.

The test data that has been constructed by myself, covers the very basic cases such as terms consisting of only one $\sqcap/\sqcup$-connectives. This way, the very basic bugs are revealed.

Normally, the test data consists of two parts. One part is the actual data to be fed to the software and the other part is the expected result. The test data constructed for this test suite only consists of one part—there is no data part for the expected result. This is due to the test data being used both as the actual and expected result. This is clarified further in the following section.

6.3.1.2 Conducting the Test

The way this test is conducted is not the same as the usual way—comparing the actual result of the test to the expected result. For this test suite the test data is only in the internal format which is converted to the external format using the *to_external*/2 predicate. The result is used as input to the *to_internal*/2 predicate which converts the data back to the internal format—its original form. If this goes well the test is passed. This way both predicates are tested using the same set of data.

The test is conducted using a driver program that takes each test case and runs it like described above.

6.3.2 Testing the Proof Constructor

The second test suite is intended to test the correctness of the proof constructor, *Xtableaux.pl*. This test suite, like the first one, consists of many different parts. These parts are listed below.

Xtableaux.pl The extended tableaux implementation, i.e. the proof constructor.

grammar.pl The implementation of the ontology format converter.

pc_tester.pl The driver program that runs automatically all the test cases.

data The test cases for this test suite.

As listed above, this test suite also contains the implementation of the ontology format converter. It is used by the proof constructor in order to convert the expressions in the internal format to the expressions in the external format before outputting them.

6.3.2.1 Construction of Test Data

For this test suite, the entire test data has been taken from the Extended Mindswap Test in Herchenröder's work [2, p. 69]. Each test case consists of a hypothesis and the expected result, i.e. whether the hypothesis is satisfiable or not. Half of the test cases are TBox free while the other half has TBox.

6.3.2.2 Conducting the Test

This test suite, like the other one, is conducted using a driver program that runs the proof constructor against every test case and tests the actual and the expected result. If they are the same the test is passed otherwise it is failed. The whole process is automated, i.e. both the proof constructor and the test cases are loaded by the driver program and the result of each test case is given back.

6.3.3 Test Results

The test suite for testing the ontology format converter passed error free the first time. This means that there were no bugs in the program that could be revealed by this test suite.

The test suite for testing the proof constructor revealed, though, some bugs. These bugs were caused by missing negation sign ($\sim$) in some of the predicates, e.g. *negnormform/2*, which resulted in wrong answers some of the times. By fixing these bugs the test suite was passed, thus ensuring the correctness of the reasoner.

CHAPTER 7

Conclusions

The goal of this work was to develop educational resources for teaching Description Logics and reasoning about them using Tableaux algorithms. The starting point of this work was the Master Thesis of Thomas Herchenröder [2] from the University of Edinburgh.

Using Herchenröder's work and his implementation of Tableaux algorithm for reasoning in $\mathcal{ALC}$, two extensions were developed. One of these extensions introduces an alternative ontology format, the external format, that is more human-friendly compared to the one used by Herchenröder, the internal format. In addition, a converter is developed that converts expressions between the two formats.

The second extension, *Xtableaux.pl*, is directly developed on top of the Prolog implementation of Herchenröder, *tableaux.pl*. This extension, called Proof Constructor, is meant to construct the whole proof by outputting the intermediate steps in the proof process to a file. The intention behind this extension is to help students understand the Tableaux algorithms and help in debugging the ontology.

7.1 Future Work

In his conclusion, Herchenröder mentions nine different ways to extend his work [2, p. 56-59]. This work, in addition to those extensions, can also be extended.

The ontology format converter can be extended with an interface that given a term in internal format or external format outputs the result to some medium, e.g. a file.

It is also possible to extend the proof constructor. One way would be to output the constructed proof as a graphical proof tree instead of pure text.

7.2 Reflection

Looking back at the past five months, I realize that working on this project has effected me in many levels. One important thing which I have learned in the course of this project was the ability to work independently. Moreover, I learned how to manage this kind of project – development based on reverse engineering.

Another important aspect of this project was learning more about Prolog and artificial intelligence in general. This had a profound effect on my choice of career.

Appendix A

Listing of tableaux.pl

This is the code for tableaux.pl, the Tableaux reasoner that was developed by Herchenröder . The listing below is the exact copy from his work.

```
% tableaux.pl -- tableaux reasoner for description logics
:- use_module(library(lists)).
:- op(100,fy,~).
:- dynamic ont/1.
:- dynamic id/1.

% Main Proof Goal
tableaux_proof(Exp) :- % true/fals = satisfiable/unsatisfiable
        proof(Exp).

%construct_goal
proof(equiv(A,B)) :-
        \+ tabl(and(A,~B)),
        \+ tabl(and(B,~A)).
proof(subsum(A,B)) :-
        \+ tabl(and(A,~B)).
proof(disjoint(A,B)) :-
        \+ tabl(and(A,B)).
```

```
proof(unsat(A)) :- % unsatisfiable A
        \+ tabl(A).
proof(A) :- % try to satisfy everything else
        tabl(A).

% main worker
tabl(Exp) :-
        expand_defs(Exp,Exp1), % expand expression into most basic
        negnormform(Exp1,Exp2), % NNF transformation
        setID(0),
        !,
        search([[Exp2]],df,_). % do the proof as an agenda search

negnormform(~ ~X,X1) :-
        negnormform(X,X1).
negnormform(~forall(R,C),exist(R,C1)) :-
        negnormform(~C,C1).
negnormform(forall(R,C),forall(R,C1)) :-
        negnormform(C,C1).
negnormform(~exist(R,C),forall(R,C1)) :-
        negnormform(~C,C1).
negnormform(exist(R,C),exist(R,C1)) :-
        negnormform(C,C1).
negnormform(~and(A,B),or(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(and(A,B),and(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~or(A,B),and(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(or(A,B),or(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~X,~X) :-
        atom(X).
negnormform(X,X) :-
        atom(X).
expand_defs(forall(R,C),forall(R,C1)) :-
        expand_defs(C,C1).
expand_defs(exist(R,C),exist(R,C1)) :-
        expand_defs(C,C1).
expand_defs(and(A,B),and(A1,B1)) :-
```

```
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(or(A,B),or(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(~A,~A1) :-
        expand_defs(A,A1).
expand_defs(X,Y) :-
        atom(X),
        ont(equiv(X,X1)),
        expand_defs(X1,Y).
expand_defs(X,X) :-
        atom(X),
        \+ ont(equiv(X,_)).

% search(+Goal,+Style,-ResultList) -- agenda style search
% -- transforms Goal into a list of [clash]/[model] elements
search([],_,[]).
search(Reduced, _, Reduced) :-
        Reduced = [H|_],
        H = [clash], % a clash leaf fails the proof
        !,
        fail.
search([Node| T], Style, Reduced) :-
        process_node(Node,NewNodes),
        filter_nodes(NewNodes,NewNodes1),
        merge_agendas(NewNodes1, T, Style, New),
        search(New, Style, Reduced).
filter_nodes(NewNodes,NewNodes1) :-
        ( setof(X,(member(X,NewNodes),X\=[model]),NewNodes1);
          NewNodes1 = []),
        !.

merge_agendas(A1, A2, df, New) :-
        append(A1, A2, New),!.
merge_agendas(A1, A2, bf, New) :-
        append(A2, A1, New),!.

% reduce a node of the proof tree
process_node([clash],[]) :- !.
process_node([model],[]) :- !.
process_node(ListOfDLExps,ResultListOfLists) :-
        transform_connect(ListOfDLExps,_,LoL3),
        expand_nodes(LoL3,ResultListOfLists).
expand_nodes([],[]).
expand_nodes([H|R],RLoL) :-
        expand_node(H,RLoL1),
        expand_nodes(R,RLoL2),
        append(RLoL1,RLoL2,RLoL).

% process a single node
expand_node([],[]).
expand_node([model],[[model]]) :-!.
expand_node([clash],[[clash]]) :-!.
expand_node(Node,[N1]) :-
        check_clash(Node,N1),!. % clash closes this branch
expand_node(Node,N1) :-
        expand_exist(Node,[],LoE), % expand existential restrictions
        LoE \= [], % only continue with non-empty edges
        expand_forall(Node,LoE,LoE2), % try eliminate value restrictions
        extract_nodes(LoE2,N1). % get the list of new fringe nodes
expand_node(Node,[N1]) :- % model closes this branch
        expand_exist(Node,[],LoE),
        LoE = [],
        N1 = [model].

% extract list of nodes from list of edges
extract_nodes([],[]).
extract_nodes([edge(_,_,N)|R],[N|R1]) :-
        extract_nodes(R,R1).

% transform and/or connectives
transform_connect([],_,[[]]).
transform_connect([and(A,B)|R],N,R1) :-
        ( member(A,R) ->
           ( member(B,R) ->
              transform_connect(R,N,R1);
              transform_connect([B|R],N,R1));
           ( member(B,R) ->
              transform_connect([A|R],N,R1);
              transform_connect([A,B|R],N,R1))).
transform_connect([or(A,B)|R],N,LoL) :-
        ( \+ (member(A,R) ; member(B,R)) ->
           ( transform_connect([A|R],N,LoL);
             transform_connect([B|R],N,LoL));
           transform_connect(R,N,LoL)).
transform_connect([H|R],N,LoL) :-
```

```
        H \= and(_,_),
        H \= or(_,_),
        transform_connect(R,N,LL1),
        LL1 = [LL2],
        LoL = [[H|LL2]].

% transform forall/exist quantifiers
expand_exist([],L,L).
expand_exist([exist(R,C)|T],T1,L) :-
        ( (member(edge(R,_,X),T1), member(C,X) ) -> % existing R edge
           expand_exist(T,T1,L);
           getID(Id),
           expand_exist(T,[edge(R,Id,[C])|T1],L)
        ).
expand_exist([H|T],T1,L) :-
           H \= exist(_,_),
           expand_exist(T,T1,L).

expand_forall(_,[],[]).
% push concept into exist. node
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N2)|R1]) :-
        setof(X,member(forall(R,X), Node),C1),
        append(C1,N,N1),
        remove_duplicates(N1,N2),
        expand_forall(Node,RoE,R1).
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N)|LoE]) :-
        \+ member(forall(R,_),Node),
        expand_forall(Node,RoE,LoE).

check_clash(Exp,Exp1) :-
        member(A,Exp),
        member(~A,Exp),
        Exp1 = [clash].

remove_duplicates([],[]).
remove_duplicates([H|T],[H|R]) :-
        r_d(H,T,[],R1),
        remove_duplicates(R1,R),!.
r_d(_,[],T,T).
r_d(E,[E|T],TT,T1) :- r_d(E,T,TT,T1).
r_d(E,[X|T],TT,T1) :- X \= E, r_d(E,T,[X|TT],T1).

getID(I):-
        id(I1),
        I is I1 + 1,
        retract(id(I1)),
        asserta(id(I)),!.
getID(I):-
        \+ id(_),
        I is 0,
        asserta(id(I)),!.
setID(X):-
        ( id(Y) ->
           retract(id(Y));
           true
        ),
        asserta(id(X)).
```

Appendix B

Listing of Xtableaux.pl

This is the Prolog code for *Xtableaux.pl* that extends *tableaux.pl* with the notion of proof construction.

```
% Xtableaux.pl -- The extended tableaux reasoner for description logics
:- use_module(library(lists)).
:- op(100,fy,~).
:- dynamic ont/1.
:- dynamic id/1.
:- dynamic fid/1.
:- ensure_loaded('grammar.pl').


% Unique file name generator
getFileID(I):-
        fid(I1),
        I is I1 + 1,
        retract(fid(I1)),
        asserta(fid(I)),!.
getFileID(I):-
        \+ fid(_),
        I is 0,
        asserta(fid(I)),!.

getFileName(X,N) :-
        getFileID(I),
        atom_concat(X, I, N).


% Main Proof Goal
tableaux_proof(Exp) :- % true/false = satisfiable/unsatisfiable
        getFileName(proof,F),
        telling(Old),
        tell(F),
        write('--- INITIAL EXPRESSION: ---'), nl,
        to_ext(Exp,Fml),
        write(Fml), nl,
        ( proof(Exp) ->
                write('- SATISFIABLE -')
                ;
                write('- UNSATISFIABLE -')
        ),
        nl,
        told,
        tell(Old).

%construct_goal
proof(equiv(A,B)) :-
        write('--- GOAL: ---'), nl,
        to_ext(and(A,~B),Fml1),
        write(Fml1),
        write(' AND '),
        to_ext(and(B,~A),Fml2),
        write(Fml2),
        nl,
        \+ tabl(and(A,~B)),
        \+ tabl(and(B,~A)).
proof(subsum(A,B)) :-
        write('--- GOAL: ---'), nl,
        to_ext(and(A,~B),Fml),
        write(Fml),
        nl,
        \+ tabl(and(A,~B)).
proof(disjoint(A,B)) :-
        write('--- GOAL: ---'), nl,
        to_ext(and(A,~B),Fml),
        write(Fml),
        nl,
        \+ tabl(and(A,B)).
proof(unsat(A)) :- % unsatisfiable A
        write('--- GOAL: ---'), nl,
        to_ext(A,Fml),
        write(unsatisfiable(Fml)),
        nl,
        \+ tabl(A).
proof(A) :- % try to satisfy everything else
        write('--- GOAL: ---'), nl,
        to_ext(A,Fml),
        write(Fml),
        nl,
        tabl(A).

% main worker
tabl(Exp) :-
        write('--- EXPANDED EXPRESSION: ---'), nl,
        expand_defs(Exp,Exp1), % expand expression into most basic
        to_ext(Exp1,Fml1),
        write(Fml1), nl,
        write('--- NEGATIVE NORMAL FORM: ---'), nl,
        negnormform(Exp1,Exp2), % NNF transformation
        to_ext(Exp2,Fml2),
        write(Fml2), nl,
        setID(0),
        !,
        write('===== TABLEAUX INFERENCE ====='), nl,
        search([[Exp2]],df,_). % do the proof as an agenda search

negnormform(~ ~X,X1) :-
        negnormform(X,X1).
negnormform(~forall(R,C),exist(R,C1)) :-
        negnormform(~C,C1).
negnormform(forall(R,C),forall(R,C1)) :-
        negnormform(C,C1).
negnormform(~exist(R,C),forall(R,C1)) :-
        negnormform(~C,C1).
negnormform(exist(R,C),exist(R,C1)) :-
        negnormform(C,C1).
negnormform(~and(A,B),or(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(and(A,B),and(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~or(A,B),and(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(or(A,B),or(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~X,~X) :-
        atom(X).
negnormform(X,X) :-
        atom(X).

expand_defs(forall(R,C),forall(R,C1)) :-
        expand_defs(C,C1).
expand_defs(exist(R,C),exist(R,C1)) :-
        expand_defs(C,C1).
expand_defs(and(A,B),and(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(or(A,B),or(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(~A,~A1) :-
        expand_defs(A,A1).
expand_defs(X,Y) :-
        atom(X),
        ont(equiv(X,X1)),
        expand_defs(X1,Y).
expand_defs(X,X) :-
        atom(X),
        \+ ont(equiv(X,_)).

% search(+Goal,+Style,-ResultList) -- agenda style search
% -- transforms Goal into a list of [clash]/[model] elements
search([],_,[]).
search(Reduced, _, Reduced) :-
        Reduced = [H|_],
        H = [clash], % a clash leaf fails the proof
        write('CLASH'), nl,
        !,
        fail.
search([Node| T], Style, Reduced) :-
```

```
        to_ext_dlist([Node|T],FmlList),
        write(FmlList), nl,
        process_node(Node,NewNodes),
        filter_nodes(NewNodes,NewNodes1),
        merge_agendas(NewNodes1, T, Style, New),
        search(New, Style, Reduced).
filter_nodes(NewNodes,NewNodes1) :-
        ( setof(X,(member(X,NewNodes),X\=[model]),NewNodes1);
          NewNodes1 = []),
        !.

merge_agendas(A1, A2, df, New) :-
        append(A1, A2, New),!.
merge_agendas(A1, A2, bf, New) :-
        append(A2, A1, New),!.

% reduce a node of the proof tree
process_node([clash],[]) :- !.
process_node([model],[]) :- !.
process_node(ListOfDLExps,ResultListOfLists) :-
        write('*** AND/OR-eleminations: ***'), nl,
        transform_connect(ListOfDLExps,_,LoL3),
        to_ext_dlist(LoL3,FmlList),
        write(FmlList), nl,
        expand_nodes(LoL3,ResultListOfLists).

expand_nodes([],[]).
expand_nodes([H|R],RLoL) :-
        expand_node(H,RLoL1),
        expand_nodes(R,RLoL2),
        append(RLoL1,RLoL2,RLoL).

% process a single node
expand_node([],[]).
expand_node([model],[[model]]) :-!.
expand_node([clash],[[clash]]) :-!.
expand_node(Node,[N1]) :-
        check_clash(Node,N1),!. % clash closes this branch
expand_node(Node,N1) :-
        write('*** EXIST-elemination: ***'), nl,
        expand_exist(Node,[],LoE), % expand existential restrictions
        LoE \= [], % only continue with non-empty edges
        write('*** FORALL-elemination: ***'), nl,
        expand_forall(Node,LoE,LoE2), % try eliminate value restrictions
        extract_nodes(LoE2,N1), % get the list of new fringe nodes
        write('*** NEW FRINGE NODES: ***'), nl,
        !. %%% ADDED by ISMAIL (closes this branch)
expand_node(Node,[N1]) :- % model closes this branch
        expand_exist(Node,[],LoE),
        LoE = [],
        N1 = [model].

% extract list of nodes from list of edges
extract_nodes([],[]).
extract_nodes([edge(_,_,N)|R],[N|R1]) :-
        extract_nodes(R,R1).

% transform and/or connectives
transform_connect([],_,[[]]).
transform_connect([and(A,B)|R],N,R1) :-
        ( member(A,R) ->
           ( member(B,R) ->
              transform_connect(R,N,R1);
              transform_connect([B|R],N,R1));
           ( member(B,R) ->
              transform_connect([A|R],N,R1);
              transform_connect([A,B|R],N,R1))).
transform_connect([or(A,B)|R],N,LoL) :-
        ( \+ (member(A,R) ; member(B,R)) ->
           ( transform_connect([A|R],N,LoL);
             transform_connect([B|R],N,LoL));
           transform_connect(R,N,LoL)).
transform_connect([H|R],N,LoL) :-
        H \= and(_,_),
        H \= or(_,_),
        transform_connect(R,N,LL1),
        LL1 = [LL2],
        LoL = [[H|LL2]].

% transform forall/exist quantifiers
expand_exist([],L,L).
expand_exist([exist(R,C)|T],T1,L) :-
        ( (member(edge(R,_,X),T1), member(C,X) ) -> % existing R edge
           expand_exist(T,T1,L);
           getID(Id),
           write(' '),
           to_ext(C,Fml),
           write([edge(R,Id,[Fml])]),
           nl,
           expand_exist(T,[edge(R,Id,[C])|T1],L)
        ).
expand_exist([H|T],T1,L) :-
           H \= exist(_,_),
           expand_exist(T,T1,L).

expand_forall(_,[],[]).
% push concept into exist. node
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N2)|R1]) :-
        setof(X,member(forall(R,X), Node),C1),
        append(C1,N,N1),
        remove_duplicates(N1,N2),
        write(' '),
        to_ext_list(N2,Fml),
        write([edge(R,I,Fml)]),
        nl,
        expand_forall(Node,RoE,R1).
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N)|LoE]) :-
        \+ member(forall(R,_),Node),
        expand_forall(Node,RoE,LoE).

check_clash(Exp,Exp1) :-
        member(A,Exp),
        member(~A,Exp),
        Exp1 = [clash].

remove_duplicates([],[]).
remove_duplicates([H|T],[H|R]) :-
        r_d(H,T,[],R1),
        remove_duplicates(R1,R),!.
r_d(_,[],T,T).
r_d(E,[E|T],TT,T1) :- r_d(E,T,TT,T1).
r_d(E,[X|T],TT,T1) :- X \= E, r_d(E,T,[X|TT],T1).

getID(I):-
        id(I1),
        I is I1 + 1,
        retract(id(I1)),
        asserta(id(I)),!.
getID(I):-
        \+ id(_),
        I is 0,
        asserta(id(I)),!.
```

```
setID(X):-
        ( id(Y) ->
           retract(id(Y));
           true
        ),
        asserta(id(X)).
```

In the following a list of modification to the *tableaux.pl* that results in *Xtableaux.pl* is given.

- The dynamic predicate *fid/1* has been added in order to generate a unique number—the same as predicate *id*/1.
- The two predicates *getFileID/1* and *getFileName/2* has been added for the purpose of generating a unique name for each file that contains a proof.
- The main predicate, *tableaux_proof/1*, has been extended to generate a file (if not exists already) for the constructed proof. It is also here that both the initial expression as the hypothesis and the result of the proof is outputted.
- Predicate *proof/1* has been extended to output the constructed goal.
- The main worker predicate, *tabl*/1, has been extended to output both the expended expression and the result of the *negnormform/2* (Negation Normal Form). It also outputs the beginning of the Tableaux inference rules.
- Predicate *search*/3 has been extended to output the current set of nodes or *clash* if one is found.
- Predicate *process_node*/2 has been extended to output the result of elimination of $\sqcap/\sqcup$-connectives.
- Predicate *expand_node*/2 has been extended to output the result of expanding $\exists$ and $\forall$ restrictions. Moreover, it also outputs the beginning of processing a new fringe nodes.
- Predicate *expand_exist*/3 has been extended to output the result of each $\exists$-expansion, i.e. the construction of each new edge to the current node.
- Likewise, the *expand_forall/3* predicate has been extended to output the result of each $\forall$-expansion.

Appendix C

The Internal Ontology Format

This is the complete ontology format which is developed by Herchenröder . It has been presented the same way as it is presented in appendix D in [2].

```
DLexpression :: DLconcept | DLaxiom | DLquery

DLquery      :: equiv(DLconcept, DLconcept) |
                subsum(DLconcept, DLconcept) |
                disjoint(DLconcept, DLconcept) |
                unsat(DLconcept) |
                DLconcept

DLaxiom      :: ont(equiv(PrimitiveConcept, DLconcept))

DLconcept    :: PrimitiveConcept |
                and(DLconcept, DLconcept) |
                or(DLconcept, DLconcept) |
                exist(Relation, DLconcept)|
                forall(Relation,DLconcept)|
                ~DLconcept

PrimitiveConcept :: PrologLiteral
```

```
Relation      :: PrologLiteral
```

Appendix D

Listing of grammar.pl

This is the Prolog code for the *grammar.pl*—the ontology format converter.

```
:- op(650,xfy,!).      /* universal   */
:- op(640,xfy,?).      /* existential */
:- op(630,xfy,\/).     /* disjunction */
:- op(620,xfy,/\).     /* conjunction */
:- op(610,fy,~).       /* negation    */

sym_opr(!,forall).
sym_opr(?,exist).
sym_opr(\/,or).
sym_opr(/\,and).
sym_opr(~,~).

%
% Convert to Externel Format
%
to_ext(Int, Ext) :-
        Int =.. [Opr, X1, X2],
        sym_opr(Sym, Opr), !,
        to_ext(X1, Y1),
        to_ext(X2, Y2),
        Ext =.. [Sym, Y1, Y2].

to_ext(Int, Ext) :-
        Int =.. [Opr, X1],
        sym_opr(Sym, Opr), !,
        to_ext(X1, Y1),
        Ext =.. [Sym, Y1].

to_ext(F, F).

%
% Convert to Internal Format
%
to_int(Ext, Int) :-
        Ext =.. [Sym, X1, X2],
        sym_opr(Sym, Opr), !,
        to_int(X1, Y1),
        to_int(X2, Y2),
        Int =.. [Opr, Y1, Y2].

to_int(Ext, Int) :-
        Ext =.. [Sym, X1],
        sym_opr(Sym, Opr), !,
        to_int(X1, Y1),
        Int =.. [Opr, Y1].

to_int(F, F).

%
% Convert a list of internal formulas to external format
%
to_ext_list([X|T],[Fml|L]):-
        to_ext(X,Fml),
        to_ext_list(T,L).

to_ext_list([],[]).

%
% Convert a two-diminsional list of internal formulas to external format
%
to_ext_dlist([X|T],[Y|L]) :-
        to_ext_list(X,Y),
        to_ext_dlist(T,L).
to_ext_dlist([],[]).
```

The two predicates *to_ext_dlist*/2 and *to_ext_list*/2 has been developed in order to process a two dimensional and one dimensional lists of nodes, respectively. They are used in the development of the proof constructor (see section 6.2 on page 44).

Appendix E

Test Data for Proof Constructor

This is the test data used in the test suite that tested the correctness of the proof constructor. There are in total 10 files where each file name is proceeded by its content.

```
ex1_1.pl:
sat(false).
query( and( wine, beer)).
ont(equiv(beer,and( drink,and(exist( hasingr, water),
and( exist( hasingr,hops),and( exist( hasingr, malt),
forall( hasingr, or( water, or(hops,malt))))))))).
ont(equiv(grapes,and( (~hops),and( (~malt), (~water))))).
ont(equiv(wine,and( drink, exist( hasingr, grapes)))).

ex1_2_1.pl:
sat(false).
query( and( exist( r, b), forall( r, (~b)))).

ex1_2_2.pl:
sat(true).
query( and( exist( r, b), forall( r, or( a, (~b))))).
```

```
ex1_2_3.pl:
sat(false).
query( and( exist( r, b), forall( r, (~b)))).
ont(equiv(aa,or( a, (~a)))).
ont(equiv(a,exist( r, exist( r, exist( r, c))))).
ont(equiv(b,and( a, and(exist( r, aa), or( aa, (~aa)))))).

ex1_2_4.pl:
sat(false).
query( and( exist( r, b), forall( r, or( a, (~b))))).
ont(equiv(a,( ~b))).

ex1_3.pl:
sat(true).
query( and( werewolf, human)).
ont(equiv(werewolf,and( animal, and(exist( haspower,
magical),forall( speaks, language))))).
ont(equiv(acramantula,and( beast,and(or( male,
female),exist( haspower, magical))))).
ont(equiv(wizard,and( male,and(human,exist(
haspower, magical))))).
ont(equiv(centaur,and( animal,and((~human),and(
or( male, female),and(exist(haspower, magical),
forall( speaks, language))))))).
ont(equiv(vampire,and( beast,and(or( male,
female),forall( haspower,magical))))).
ont(equiv(beast,and( animal,(~human)))).
ont(equiv(muggle,and( human,and( or( male,
female),forall( haspower,(~magical)))))).
ont(equiv(witch,and( female,and( human,
exist( haspower, magical))))).
ont(equiv(human,and( animal,exist( speaks, language)))).
ont(equiv(male,and( animal,(~female)))).
ont(equiv(merpeople,and( animal,and( (~human),
and( forall( haspower, magical),forall( speaks,
language)))))).

ex1_8.pl:
sat(true).
query( or( and( or( a, b), or( c, d)),
and( or( a, c), or( b, d)))).

ex1_10.pl:
```

```
sat(true).
query( and( veggiepizza, meatpizza)).
ont(equiv(veggiepizza,and( pizza, forall( hastopping,
(~meat))))).
ont(equiv(meatpizza,and( pizza, forall( hastopping,
(~veggie))))).
ont(equiv(veggie,or( mushroom, olive))).
ont(equiv(meat,or( pepperoni, sausage))).

ex1_11.pl:
sat(false).
query( and( exist( p, a), and(exist( p, b), and(
and( c, d), (~exist( p, (~and((~e), f)))))))).
ont(equiv(a,and( h, and( i, (~d))))).
ont(equiv(j,( ~k))).
ont(equiv(b,( ~g))).
ont(equiv(d,forall( q, j))).
ont(equiv(g,( ~e))).

ex2_7.pl:
sat(true).
query( and( b, and(c, and(exist( p, and( a,
and(c, exist( r, (~d))))), forall(r, d))))).
```

Appendix F

Test Data for Ontology Format Converter

This is the test data used in the test suite that tested the correctness of the ontology format converter. There are in total 10 files where each file name is proceeded by its content. The 4 first files are constructed by myself. The rest of the files indicate where their content come from using Prolog comments. The files that are indicated in the comments are from the Extended Mindswap Tests [2, p. 69-80].

```
exp_1.pl:
query(and( a, b)).

exp_2.pl:
query(or( a, b)).

exp_3.pl:
query(exist(r, b)).

exp_4.pl:
query(forall(r, a)).

exp_5.pl:
```

```
%
% The query from ex1_11.pl
%
query( and( exist( p, a), and(exist( p, b),
and(and( c, d), (~exist( p, (~and((~e), f)))))))).

exp_6.pl:
%
% The query from ex1_2_1.pl
%
query( and( exist( r, b), forall( r, (~b)))).

exp_7.pl:
%
% The query from ex1_2_2.pl
%
query( and( exist( r, b), forall( r, or( a, (~b))))).

exp_8.pl:
%
% The query from ex1_8.pl
%
query( or( and( or( a, b), or( c, d)),
and( or( a, c), or( b, d)))).

exp_9.pl:
%
% The query from ex1_9.pl
%
query( and( hydra, and( dragonet, exist( elemental, fire)))).

exp_10.pl:
%
% The query from ex2_1.pl
%
query( and( a, and(d, and(g, and((~m), and((~n),
and((~o), and((~p), and((~q), and((~r),
and((~s), and((~t), and((~u), and((~v1),
and((~w),(~x))))))))))))))).
```

Appendix G

Test Driver Scripts

The following two programs have been used as driver scripts for the two test suites.

G.1 Listing of pc_tester.pl

This is the code for the program that runs *Xtableaux.pl* against the test data presented in appendix E on page 71. The invocation of the program looked like

swipl -s pc_tester.pl -g run. -t halt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% pc_tester.pl                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Performs tests on the Xtableaux.pl
% in order to ensure the correctness of it.
% Inspired from reg_test.pl developed
% by Thomas Herschenroeder
:- consult('Xtableaux.pl').
```

```
% A small program to read data from a file
% and assert it to the database
%
% The program is taken from:
% http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/pt_framer.html
%
% It has been slightly modified
browse(File) :-
        seeing(Old),
        see(File),
        repeat,
        read(Data),
        process(Data),
        seen,
        see(Old),
        !.

process(end_of_file) :- !.
process(Data) :-  assert(Data), fail.

% dir interface
get_files(Dir):-
        consult(Dir), % get list of test files
        flist(Files), % into a variable
        eval_files(Files).

eval_files([F|T]):-
        eval_file(F),
        eval_files(T).

eval_file(File1):-
        retractall(query(_)),
        retractall(sat(_)),
        retractall(ont(_)),
        atom_concat('data/',File1,File),
        browse(File),
        write('Testing file: '), write(File), nl,
        query(Q),
        sat(S),
        tableaux_proof(Q,S1),
        ( S1 = S -> report(File,ok) ; report(File,err)).


report(File,ok):-
```

```
        write(File),write(': OK'),nl.

report(File,err):-
        write(File), write(': ERROR'),nl.

% Runs the test suite
run :- get_files('data/index.pl').
```

G.2 Listing of ofc_tester.pl

This is the code for the program that runs *grammar.pl* against the test data presented in appendix F on page 75. The invocation of the program looked like

swipl -s ofc_tester.pl -g run. -t halt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ofc_tester.pl                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Runs the test data for ontology format
% converter in order to test the correctness of it.
% Inspired from reg_test.pl developed by
% Thomas Herschenroeder
:- consult('grammar.pl').

% A small program to read data from a file
% and assert it to the database
%
% The program is taken from:
% http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/pt_framer.html
%
% It has been slightly modified
browse(File) :-
        seeing(Old),
        see(File),
        repeat,
        read(Data),
        process(Data),
        seen,
        see(Old),
        !.
```

```
process(end_of_file) :- !.
process(Data) :-  assert(Data), fail.

% dir interface
get_files(Dir):-
        consult(Dir), % get list of test files
        flist(Files), % into a variable
        eval_files(Files).

eval_files([F|T]):-
        eval_file(F),
        eval_files(T).

eval_file(File1):-
        retractall(query(_)),
        atom_concat('data/',File1,File),
        browse(File),
        write('Testing file: '), write(File), nl,
        query(Q),
        to_ext(Q,Fml), % convert to external format
        (   to_int(Fml,Q) -> % convert it back to internal
            report(File,ok);
        report(File,err)).

report(File,ok):-
        write(File),write(': OK'),nl.

report(File,err):-
        write(File), write(': ERROR'),nl.

% Runs the test suite
run :- get_files('data/index.pl').
```

Bibliography

[1] F. Baader, D. Calvanesse, D. L. McGuinness, D. Nardi and P. F. Patel-Schneider, *The Description Logic Handbook*: Theory, Implementation and Application, Cambridge University Press,
Second Edition, 2010.

[2] Thomas Herchenröder, Lightweight Semnatic Web Oriented Reasoning in Prolog: Tableaux Inferences for Description Logics, University of Edinburgh, 2006,

[3] D. Nardi and R. J. Brachman, *An Introduction to Description Logics* in The Description Logic Handbook, Cambridge University Press,
Second Edition, pp. 1–43.

[4] D. Nardi and R. J. Brachman, *Basic Description Logics* in The Description Logic Handbook, Cambridge University Press,
Second Edition, pp. 47–104.

[5] Franz Baader, *Description Logics*, Theoretical Computer Science, TU Dresden, Germany, 2009, volume 5689, URL: http://lat.inf.tu-dresden.de/research/papers-2009.html#Baader09

[6] Ian Horrocks, *OWL: A Description Logic Based Ontology Language*, Talk at CISA, Edinburgh, March 30th 2006 URL: http://www.cs.man.ac.uk/~horrocks/Slides/index.html