Practical Reasoning in Expressive Description Logics Using Alternating Automata

A Dissertation submitted to
Faculty of Computer Science,
Free University of Bozen-Bolzano
in partial fulfillment of the degree of
MASTER OF SCIENCE

Submitted By:
Kumar Abhinav

Supervisors:
Prof. Diego Calvanese
Dr. Peter F. Patel-Schneider

March 2009
Abstract

Acknowledgements

1 Introduction 1
1.1 Aim of the Thesis ................................. 2
1.2 Related Work ................................... 3
1.3 Outline of the Thesis ............................. 3

2 Knowledge Representation with Description Logics 4
2.1 The Description Logic $\mathcal{ALC}$ ............................. 5
2.1.1 Syntax ....................................... 5
2.1.2 Semantics ................................... 5
2.2 Knowledge Base ................................... 6
2.3 Inference Problems ............................... 7

3 Reasoning in Description Logics 9
3.1 Reasoning Paradigms ............................. 9
3.1.1 Structural Algorithms .......................... 9
3.1.2 Tableau Algorithms ........................... 9
3.1.3 Translational Approaches ....................... 10
3.1.4 Automata-based Approach ....................... 10
3.2 Conversional Approach .......................... 11
3.3 Translational Approach .......................... 12
3.3.1 BDD-based Decision Procedure for $K$ .......... 12
3.4 Modal $\mu$-calculus Approach ....................... 13
3.4.1 Encoding into TWATA .......................... 13
3.4.2 Incremental Decision Procedure ................. 13
3.5 Inverse Method Approach ........................ 13
3.6 MONA ........................................ 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>7.1 Outlook</td>
<td>52</td>
</tr>
<tr>
<td>7.1.1 Optimizations</td>
<td>52</td>
</tr>
<tr>
<td>7.1.2 DL Language Extensions</td>
<td>52</td>
</tr>
<tr>
<td>7.1.3 Ontological Representations</td>
<td>52</td>
</tr>
<tr>
<td>7.1.4 Proof Explanation</td>
<td>52</td>
</tr>
<tr>
<td>A Test Queries</td>
<td>53</td>
</tr>
<tr>
<td>A.1 Queries without TBox</td>
<td>53</td>
</tr>
<tr>
<td>A.2 Queries with Acyclic TBox</td>
<td>55</td>
</tr>
<tr>
<td>A.3 Queries with Cyclic TBox</td>
<td>64</td>
</tr>
<tr>
<td>Bibliography</td>
<td>66</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 A TBox with Concepts about Family Relationships .......................... 6  
2.2 The Family ABox ................................................................. 7  
3.1 BDD-represented Automaton .................................................. 14  
4.1 A Deterministic Finite State Automaton ......................................... 17  
4.2 A nondeterministic Finite State Automaton ........................................ 18  
4.3 A Büchi Automaton ................................................................. 19  
4.4 Σ-labelled Tree ........................................................................ 20  
4.5 Looping Tree Automaton ............................................................. 21  
4.6 Accepting Run of Alternating Automaton ........................................... 23  
5.1 Incremental Satisfaction Approach ............................................... 34  
6.1 Config. 1 Performance Results ..................................................... 48  
6.2 Config. 2 Performance Results ..................................................... 48  
6.3 Config. 3 Performance Results ..................................................... 50
List of Tables

5.1 ALT Transition Relation ........................................ 27
6.1 Negation Normal Form Transformation Rules .................... 39
6.2 Rules for Sub-Concept Computation .............................. 39
6.3 Predicates for ALT Automata Construction ....................... 40
6.4 Predicates for ULT Automata Construction ....................... 41
6.5 Predicates for NLT Automata Construction ....................... 42
6.6 Satisfiability Results of Queries without TBox .................. 46
6.7 Satisfiability Results of Queries with Acyclic TBox ............. 47
6.8 Satisfiability Results of Queries with Cyclic TBox .............. 49
Abstract

Description Logic (DL) languages have become a de-facto standard for Knowledge Representation in recent times. They help us in capturing the terminological knowledge of the targeted domain in a precise manner. By using DLs, the knowledge of the application domain can be represented in a formal way enabling the Knowledge Representation Systems based on DLs to perform various kinds of inference tasks like satisfiability checking, instance checking, subsumption etc. These reasoning capabilities allow us to find implicit consequences of the explicitly represented knowledge and thereby facilitating the development of smart applications. The traditional way of reasoning in Description Logics is based on the tableau algorithms. Though the tableau algorithms have performed well for reasoning in DLs, they do not have the optimal worst-case complexity for a given DL. Due to this the worst-case complexity bounds for a logic are usually proven by the theoretically optimal automata-based algorithms. Thus, the researchers often end up in creating two algorithms for a new logic, one automata-based for proving the exact complexity bounds of the logic and another a tableau-based for a practical implementation. The problem with the automata-based algorithms is that in spite of having nice theoretical properties they have best-case exponential behavior. Due to this, to the best of our knowledge, there was no attempt made for developing an automata-based reasoning tool for DLs.

In this work we investigate the novel possibility of developing automata-based reasoning tool for DL $\mathcal{ALC}$. We obtain our results by an innovative approach of dividing the input concept into smaller sub-concepts and then checking the satisfiability of these sub-concepts in an incremental fashion, inspired by the recent work on checking the satisfiability of $\mu$-calculus formulas using the automata-based algorithms. This involves handling of technical challenges like deciding subsumption in the presence of a cyclic TBox. We introduce the definitions of alternating looping tree automata (ALT), universal looping tree automata (ULT), non-deterministic looping tree automata (NLT) and investigate the decomposition of ALT to ULT and ULT to NLT transformations. Then we provide an incremental satisfaction algorithm for deciding satisfiability using automata-based technique. When combined together, they provide a decision procedure for DL $\mathcal{ALC}$. We also present a prototype implementation of the proposed algorithm in programming language Prolog. This preliminary implementation is then compared with an optimized tableau algorithm implementation for characterizing the strengths and weaknesses of the proposed approach.
First of all I would like to thank God for providing me the opportunities and strength to endure the hardships of life.

Several people have been instrumental in allowing this work to be completed. I would like to especially thank my supervisors Prof. Diego Calvanese and Dr. Peter F. Patel-Schneider for their expert guidance and timely advises that saved me from numerous pitfalls and made this work possible.

I would like to acknowledge the efforts of Prof. David Toman and Gulay Unel, University of Waterloo, for their generous help and numerous advises.

I also wish to thank Prof. Enrico Franconi and Dr. Peter F. Patel Schneider for providing me an opportunity to spend my summer at Bell Labs Research. I would also like to express my gratitude towards Prof. Sergio Tessaris and the faculty of Free University of Bolzano, Technical University Dresden for introducing me to the fascinating world of computational logic. Prof. Franz Baader summons a special mention since he was the first person who introduced me to Description Logics and laid the foundations for this work.

Prof. Viktor Kuncak and the wonderful group of LARA, EPFL deserves my kind thanks for bearing with my frequent absences (due to this work) and always encouraging me.

I would like to thank Vikash Kumar, Sana Faraz, Ravi Nandan and all my friends for extending me their valuable support and advice in making important decisions.

Most importantly, I would like to thank my parents and sister for being the pillars of love and faith during my whole life, without their unwavering and unconditional support I couldn’t have made this far.

Kumar Abhinav
CHAPTER 1

Introduction

What makes our human race so different from the other animals? Different people will provide a different answer to this question. But if we critically analyze those answers, we will definitely find our ability to reason contributing significantly to our rational behavior. Animals handle their problems by instinct, they do what their instinct tells them to do while humans reason their way out of the problems. This argument highlights the importance of reasoning in our intelligent behavior. Inspired by the role of reasoning in human intelligence, there had been an untiring attempt by mankind to enable machines so that they can reason about the tasks instead of just doing meaningless computations.

Description Logics (DLs) are a family of knowledge representation languages that allow to capture the background knowledge of an application domain in a well structured and formal way. They help us in capturing the terminological knowledge of the targeted domain in a precise manner. Since, by using DLs, the knowledge of the application domain can be represented in a formal way, the Knowledge Representation Systems based on DLs can perform various kinds of inference tasks like satisfiability checking, instance checking, subsumption etc. These reasoning capabilities allow us to exploit the implicit knowledge hidden in the explicitly represented knowledge, thereby facilitating the development of smart applications. In a specific DL, the terminologies of the domain under consideration are described by concepts which represent the classes of individuals in the domain and roles, which stand for the relations between the individuals. A DL is characterized by the constructors it provides for building complex concepts from atomic ones. For e.g., by using constructors like conjunction, negation and existential quantification the concept “Father” can be represented as follows:

\[
\text{Father} \equiv \text{Human} \sqcap \neg \text{Female} \sqcap \exists \text{hasChild} \cdot \text{Human}
\]

i.e., a human being who is not female and has a child who is also a human being.

The core inference problems in DLs are: satisfiability checking, i.e., answering the question if a concept \( C \) can be interpreted as a non-empty set, and subsumption checking, i.e., answering the question if every individual in the interpretation of a concept \( C \) necessarily belongs to the interpretation of another concept \( D \). The inference problems are reducible into each other. For checking subsumption between concepts, the subsumption problem can be first reduced to checking the satisfiability of concepts and then the satisfiability problem.
CHAPTER 1. INTRODUCTION

can be given to a DL reasoner. One of the most widely used approaches to decide the satisfiability problem in DLs is by using \textit{tableau-based} algorithms. A tableau algorithm attempts to answer a satisfiability problem by generating a \textit{tree-shaped} structure, that can be put into correspondence with a logical model, called tableau. Tableau algorithms break a complex input problem into much simpler and smaller sub-problems and then try to satisfy each of the sub-problems (sometimes by breaking them further). If no contradictions are encountered then the original input concept is satisfiable and there exists a model for it. Although the tableau-based approach is used by most of the modern DL reasoners, theoretically due to their non-deterministic nature the tableau algorithms have a worst-case complexity that in general is not worst-case optimal. However in the recent years, several complex yet highly efficient optimizations techniques have been developed to achieve a reasonable performance level of a tableau-based reasoner, e.g., Absorption [Horrocks, 2003; Baader et al., 2007].

Another important class of reasoning algorithms are \textit{automata-based} algorithms. They translate the input DL concept either into a \textit{Non-deterministic Tree Automaton} (NTA) or into an \textit{Alternating Tree Automaton} (ATA). Non-emptiness of the language accepted by the automaton is then used to decide the satisfiability of the input concept. While tableau-based algorithms are usually not suited to prove tight (e.g., $\text{ExpTime}$) upper-bounds (for an exception, see [Donini and Massacci, 2000]), generally $\text{ExpTime}$ upper bounds are easily established by using automata-based techniques [Baader et al., 2007]. So, the researchers have to work on two algorithms for an $\text{ExpTime}$ DL, one automata-based for proving the exact complexity bounds and another tableau-based for the implementation purposes. However, this overhead can be eliminated if an automata-based DL reasoner can be developed with a practical performance level. Another advantage of using automata-based algorithm is that the proof of \textit{termination} is elegant. Since the emptiness test is performed on the (finite) model rather than a (possibly infinite) model, the algorithm terminates automatically without the need for special conditions for termination. The main hindrance in developing a purely automata-based DL inference engine comes from the fact that the automata-based reasoning algorithms are best-case $\text{ExpTime}$ for any given DL. So till date, to the best of our knowledge, no attempt has been made to develop an inference engine by directly implementing an automata-based algorithm [Hladik, 2008; Baader, 2003].

1.1 Aim of the Thesis

Most of the contemporary reasoners are based on the tableau algorithms. The tableau algorithms inherently belong to the $\text{NExpTime}$ complexity class since a tableau algorithm has to always “guess” the right path. Moreover, if the expressive DLs with cyclic terminologies are considered, then in order to ensure termination, i.e., avoiding the generation of an infinite tableau, a special mechanism known as \textit{blocking} has to be used. In a tree generated by tableau algorithm, a node is said to be \textit{blocked} if it can be replaced by (a copy of) a predecessor node. Also, to obtain an acceptable performance several sophisticated optimizations have been devised by the developers over a period of time.

The recent developments in the field of other logics like Modal Logic and Propositional $\mu$-calculus have proposed automata algorithms, that do not require the exponential step of constructing the complete automata from input before performing the emptiness test [Unel and Toman, 2007b]. So, the main objective of this thesis is to develop an incremental satisfaction algorithm for for DL $\text{ALC}$ based on automata techniques and verifying the feasibility of implementing an inference engine using the proposed algorithm. However, in order to achieve this, the following sub-tasks need to be dealt with:

- Developing an understanding of the existing automata-based algorithms for reasoning
1.2 Related Work

A translation of DL $\mathcal{ALC}$ into monadic second order theory of two successors (WS2S) and then using the automata-based MONA tool [Elgaard et al., 1998] was presented in Karabaev and Lutz [2004]. The Binary Decision Diagram (BDD) based decision procedure for modal logic $K$ was presented in Pan et al. [2002]. The authors have proposed an on-the-fly emptiness checking algorithm implemented using BDDs. In Kupferman and Vardi [2005] the authors have sketched the idea of using incremental procedure for automata construction. This idea was further developed by Unel and Toman [2007a] and they used the incremental technique to propose an algorithm for checking the satisfiability of $\mu$-calculus formulas. Baader and Tobies [2001] presents the Vornokov’s inverse method for deciding the modal logic $K$ formulas as an on-the-fly realization of the emptiness test done by the automata approach for $K$.

1.3 Outline of the Thesis

In Chapter 2, we introduce DLs and provide the syntax and semantics of DL $\mathcal{ALC}$ that will be useful in the remainder of the thesis. Chapter 3 gives the relevant background on the automata theory, describing the automata models which are useful for DLs. Chapter 4 will provide an insight into various reasoning paradigms used for reasoning in DLs.

In Chapter 5, we describe our incremental approach for automata construction from the input concept. Use of this approach will ensure that we avoid the exponential step of constructing automata before performing the emptiness test for checking the satisfiability of input concept.

We discuss about the prototype implementation of automata-based inference engine based on the proposed incremental satisfaction algorithm for DL $\mathcal{ALC}$ with general TBoxes in Chapter 6. The results of performance and related issues also discussed towards the end of the chapter. Chapter 7 concludes this work by summarizing the work done, presenting the conclusions and laying the outline for the future work.
“A knowledge representation (KR) is most fundamentally a surrogate, a substitute for the thing itself; it is a set of ontological commitments, e.g., an answer to the question: In what terms should I think about the world?; it is a fragmentary theory of intelligent reasoning; it is a medium for pragmatically efficient computation, e.g., the computational environment in which thinking is accomplished; it is a medium of human expression, e.g., a language in which we say things about the world.” R. Davis, H. Shrobe, and P. Szolovits.

Generally, the main focus of research in the field of knowledge representation and reasoning is the study of how the high-level descriptions of the world can be provided in such a way that they can be effectively utilized in building intelligent applications. Here the word “intelligent” points towards the capability of a system to find implicit consequences of the explicitly represented knowledge. These systems are therefore characterized as knowledge-based systems [Baader et al., 2007]. The approaches to knowledge representation can be roughly divided into two categories:

- **logic-based formalisms**, they evolved out of the predicate calculus on an assumption that it can be used unambiguously to capture facts about the world.

- **non-logic-based formalism**, they are built on more cognitive notions like network structures and rule-based formalism derived from the human approach towards knowledge representation.

While the non-logical systems usually were created from very specific lines of thinking and later evolved to be treated as general-purpose tools, the logic-based approaches were more general purpose from the very beginning. In a logic-based approach, the representation language is usually a variant of first-order predicate calculus and reasoning amounts to verification of logical consequences. But reasoning in different fragments of first-order logic leads to computational problems of differing complexity. So, in recent years the attention was moved towards studying the properties of the underlying logical systems, giving popularity to the field of Description Logics.
CHAPTER 2. KNOWLEDGE REPRESENTATION WITH DESCRIPTION LOGICS

2.1 The Description Logic \( ALC \)

In this section, we provide a brief introduction to Description Logics. We will discuss about the basic description language \textit{Attributive Language with Complement (ALC)} [Schmidt-Schauß and Smolka, 1991] and will present its syntax and semantics.

2.1.1 Syntax

Let \( N_c \) and \( N_R \) be disjoint sets of \textit{concept names} and \textit{role names}, respectively. The set of \( ALC \) concept (descriptions), over \( N_c \) and \( N_R \), is inductively defined as follows:

- Every concept name is an \( ALC \) concept.
- \( \top \) (top concept) and \( \bot \) (bottom concept) are \( ALC \) concept.
- If \( C, D \) are \( ALC \) concepts and \( r \) is a role name, then the following are \( ALC \) concepts:
  - \( C \sqcap D \) (conjunction)
  - \( C \sqcup D \) (disjunction)
  - \( \neg C \) (negation)
  - \( \exists r.C \) (qualified existential restriction)
  - \( \forall r.C \) (qualified value restriction)

2.1.2 Semantics

An \textit{interpretation} \( I = (\Delta^I, \cdot^I) \) consists of a non-empty set \( \Delta^I \), the \textit{interpretation domain} and a mapping \( \cdot^I \) that maps:

- each \( A \in N_c \) to a set \( A^I \subseteq \Delta^I \).
- each \( r \in N_R \) to a binary relation \( r^I \subseteq \Delta^I \times \Delta^I \).

The mapping \( \cdot^I \) can be extended to concepts as follows:

- \( \top^I = \Delta^I \)
- \( \bot^I = \emptyset \)
- \( (\neg C)^I = \Delta^I \setminus C^I \)
- \( (C \sqcup D)^I = C^I \cup D^I \)
- \( (C \sqcap D)^I = C^I \cap D^I \)
- \( (\exists r.C)^I = \{ d \in \Delta^I \mid \exists e \in \Delta^I \text{ such that } (d,e) \in r^I \text{ and } e \in C^I \} \)
- \( (\forall r.C)^I = \{ d \in \Delta^I \mid \forall e \in \Delta^I, (d,e) \in r^I \rightarrow e \in C^I \} \).

\( ALC \) has \textit{tree model property} i.e., any satisfiable concept \( C \) is satisfiable in an interpretation \( I \) that has the shape of tree whose root belongs to \( C \). However, if we consider \textit{general axioms} then the tree model can be (possibly) infinite since unravelling of a graph will result in an infinite tree if it contains a cycle.
2.2 Knowledge Base

A Knowledge Base (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$.

TBox and GCIs

A **Terminology Box** (TBox) is a finite set of concept inclusions of the form $A \sqsubseteq B$, where $A$ and $B$ are concepts. The inclusions of the form $A \sqsubseteq B$ are called **primitive concept definitions** as $B$ specifies the necessary condition for instances of $A$, such an inclusion axiom is called a definition of $A$. A **General Concept Inclusion** (GCI) has the form $C \sqsubseteq D$ where $C, D$ are complex concepts. They are also called general axioms. A TBox containing general axioms together with the primitive ones is known as general TBox. An interpretation $I$ satisfies the GCI $C \sqsubseteq D$, denoted by $I \models C \sqsubseteq D$, iff $C^I \subseteq D^I$. $I$ is a **model** of a TBox $\mathcal{T}$ denoted by $I \models \mathcal{T}$, if it satisfies all the GCIs in $\mathcal{T}$.

For TBoxes that consists of primitive concept definitions only, we define the notions of **cyclic** and **acyclic** TBox, which will be used later in this thesis. For that, we define first when a concept uses another one. A concept $A$ **directly uses** $B$ in $\mathcal{T}$ if $B$ appears in the definition of $A$, and let $\text{uses}$ be defined as the transitive closure of the relation directly uses. Then $\mathcal{T}$ contains a cycle iff there exists an atomic concept in $\mathcal{T}$ that directly or indirectly uses itself. Otherwise, $\mathcal{T}$ is called acyclic.

An axiom $A \sqsubseteq B$ is called **unique** if $\mathcal{T}$ doesn’t contain other definition of $A$. A TBox $\mathcal{T}$ that is both unique and acyclic is called **unfoldable** TBox.

Figure 2.1 shows a TBox with concepts concerned with family relationships. It is a cyclic TBox due to the axiom,

$$\text{Momd} \sqsubseteq \text{Man} \sqcap \forall \text{hasChild.Momd}$$

since the concept $\text{Momd}$, which stands for Male with only male descendants, refers itself in its definition.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man</td>
<td>$\sqsubseteq$ Person $\sqcap$ Male</td>
</tr>
<tr>
<td>Woman</td>
<td>$\sqsubseteq$ Person $\sqcap$ $\neg$Man</td>
</tr>
<tr>
<td>Mother</td>
<td>$\sqsubseteq$ Woman $\sqcap$ $\exists$hasChild.Person</td>
</tr>
<tr>
<td>Father</td>
<td>$\sqsubseteq$ Man $\sqcap$ $\exists$hasChild.Person</td>
</tr>
<tr>
<td>Parent</td>
<td>$\sqsubseteq$ $\exists$hasChild.Person</td>
</tr>
<tr>
<td>Grandfather</td>
<td>$\sqsubseteq$ Man $\sqcap$ $\exists$hasChild.Father</td>
</tr>
<tr>
<td>Brother</td>
<td>$\sqsubseteq$ Man $\sqcap$ $\exists$hasSibling.Person</td>
</tr>
<tr>
<td>Sister</td>
<td>$\sqsubseteq$ Person $\sqcap$ $\neg$Brother $\sqcap$ $\exists$hasSibling.Person</td>
</tr>
<tr>
<td>Uncle</td>
<td>$\sqsubseteq$ Man $\sqcap$ $\exists$hasSibling.Parent</td>
</tr>
<tr>
<td>Aunt</td>
<td>$\sqsubseteq$ Person $\sqcap$ $\neg$Uncle $\sqcap$ $\exists$hasSibling.Person</td>
</tr>
<tr>
<td>Momd</td>
<td>$\sqsubseteq$ Man $\sqcap$ $\forall$hasChild.Momd</td>
</tr>
</tbody>
</table>

Figure 2.1: A TBox with Concepts about Family Relationships
ABox

An Assertion Box (ABox) contains extensional knowledge about the domain of interest, i.e., assertions about individuals, called membership assertions. Figure 2.2 shows an ABox example based on family terminology.

Let \( N_i \) be the set of individual names, disjoint from \( N_c \) and \( N_R \). For individual names \( a, b \), concept \( C \), and role name \( r \), we can say:

- \( C(a) \) is a concept assertion.
- \( r(a, b) \) is a role assertion.

A finite set of such assertions is known as an Assertion Box (ABox). The assertions in an ABox are interpreted as follows:

- An interpretation \( I \) satisfies \( C(a) \), if \( a^I \in C^I \).
- An interpretation \( I \) satisfies \( r(a, b) \), if \( (a^I, b^I) \in r^I \).

\( I \) is a model of an ABox denoted \( I \models A \), if it satisfies all the assertions in \( A \).

An interpretation \( I \) is a model of \( K \) if it is a model of \( T \) and \( A \) i.e., \( I \) satisfies \( K \) iff \( I \models T \) and \( I \models A \).

```
Man(James)
Woman(Anna)
hasChild(James, Will)
hasChild(Anna, Jack)
hasSibling(Jack, Will)
Grandfather(John)
hasChild(John, James)
```

Figure 2.2: The Family ABox

2.3 Inference Problems

The different kinds of reasoning performed by a DL system are defined as logical inferences. In the following, we will first discuss the inferences for concepts and then for TBoxes. We will assume that 'T' stands for a TBox from now on.

Concept Satisfiability

A concept \( C \) is satisfiable iff there exists an interpretation \( I \) such that the extension of \( C \) is not empty, i.e., \( C^I \neq \emptyset \). Concept Satisfiability involves checking whether a concept \( C \) is satisfiable. This allows us to determine if concept \( C \) is contradictory i.e., whether there cannot be any instance of \( C \) in any interpretation. When we consider also a TBox \( T \) then we mean to restrict the attention to the interpretations that are models of \( T \). This leads to the following definition:
Definition 2.1. (The satisfiability problem w.r.t a TBox) An $\mathcal{ALC}$ concept $C$ is 
$satisfiable$ w.r.t a TBox $T$ if there is a model $I$ of $T$ with $C^I \neq \emptyset$. In this case, $I$ is also 
called a model of $C$ w.r.t $T$.

**Example 2.1.** Concept $\text{Man} \sqcap \text{Father}$ is satisfiable w.r.t to our Family TBox while concept 
$\text{Man} \sqcap \text{Woman}$ is not.

Concept satisfiability in DL $\mathcal{ALC}$ w.r.t a general TBox is an $\text{ExpTime-complete}$ problem.

**Concept Subsumption**

A concept $C$ is $\text{subsumed}$ by a concept $D$ (denoted by $C \sqsubseteq D$) iff for every interpretation 
$I$, $C^I \subseteq D^I$. Checking subsumption between two concepts or computing subsumption 
hierarchy is one of the main tasks of modern DL reasoners [Haarslev and Moller, 2001; 
Tsarkov and Horrocks, 2006; Sirin et al., 2007]. The tasks of checking subsumption and 
(un-)satisfiability are mutually reducible since $\mathcal{ALC}$ is closed under the boolean operations. 
A concept $C$ is $\text{subsumed}$ by a concept $D$ with respect to a TBox $T$, denoted by $C \sqsubseteq_T D$, 
if $C^I \subseteq D^I$ for every model $I$ of $T$.

**Example 2.2.** Concept $\text{Grandfather}$ is subsumed by concept $\text{Father}$ w.r.t Family TBox 
while concept $\text{Mom}$ is not subsumed by concept $\text{Father}$.

Concept subsumption in DL $\mathcal{ALC}$ w.r.t a general TBox is an $\text{ExpTime-complete}$ problem [Baader et al., 2007].

**Concept Equivalence**

Two concepts $C, D$ are $\text{equivalent}$ with respect to $T$ if $C^I = D^I$ for every model $I$ of $T$. In 
this case, we write $C \equiv_T D$.

**Concept Disjointness**

Two concepts $C$ and $D$ are $\text{disjoint}$ with respect to $T$ if $C^I \cap D^I = \emptyset$ for every model $I$ of 
$T$.

**Example 2.3.** Concept $\text{Brother}$ and concept $\text{Sister}$ are disjoint concepts w.r.t to our 
Family TBox.

For expressive DLs like $\mathcal{ALC}$ the various reasoning problems are mutually reducible to 
each other. Let $C, D$ denotes the $\mathcal{ALC}$ concepts, then

- $C \sqsubseteq D$ is equivalent to showing $C \sqcap \neg D$ is unsatisfiable.
- $C \equiv D$ is equivalent to showing both $C \sqcap \neg D$ and $\neg C \sqcap D$ are unsatisfiable.
- $C$ and $D$ are disjoint is equivalent to showing $C \sqcap D$ is unsatisfiable.

So, in the following work it is sufficient to consider satisfiability problem only.
In this chapter we present various techniques used for reasoning in DLs. The first half of the chapter presents a short introduction to different types of reasoning algorithms used in DLs. The latter half of the chapter provides an overview of different approaches proposed over the years for reasoning in DLs.

3.1 Reasoning Paradigms

This section presents a brief outline of the major approaches to reasoning in DLs.

3.1.1 Structural Algorithms

In order to decide the inference problems the structural algorithms compare the syntactic structure of (possibly normalized) concept descriptions. A formal description of an algorithm based on this approach was given by Nebel [1990]. These type of algorithms have a polynomial run-time but it came with a trade-off in terms of the expressive power of the chosen DL. Complete structural algorithms are known only for DLs with very limited expressiveness. For expressive logics a structural approach could not be adopted and these logics were considered unusable in practical algorithms [Levesque and Brachman, 1987]. Hence, several researchers tried to circumvent these problem adopting different approaches. Two major techniques were adopted by researchers, first, the use of incomplete algorithms to preserve the good run-time behavior of their systems. Second, tailoring the DL to maximize its expressiveness while maintaining sound and complete structural algorithms. The former approach was adopted in systems like BACK [Quantz and Kindermann, 1990] while the developers of the CLASSIC system [Patel-Schneider et al., 1991] have adopted the latter approach.

3.1.2 Tableau Algorithms

The first tableau algorithm was developed by Schmidt-Schauß and Smolka [1991] for concept subsumption in the DL $\mathcal{ALC}$ in absence of a TBox and the approach proved to be useful for deciding subsumption and other inference problems like concept satisfiability for other
DLs also. Since then, extensions of the tableau methods have been employed to obtain sound and complete satisfiability algorithms for several DLs extending ALC with number restrictions [Hollunder, 1990; Hollunder and Baader, 1991; Baader and Sattler, 1999], transitive closure of roles [Baader, 1991], transitive roles [Horrocks et al., 2000b] and concrete domains [Haarslev et al., 1999]. Tableau algorithms have also been extended to ALC with TBoxes allowing general sets of inclusion assertions [Buchheit et al., 1993]. In spite of the fact that the inference problems for the latter case are ExpTime, systems implementing the tableau approach for e.g., KRIS [Baader and Hollunder, 1991], CRACK [Bresciani et al., 1995] have shown a reasonable run-time performance on practical application problems. Modern systems equipped with sophisticated optimization techniques like FaCT [Horrocks, 1998], DLP [Patel-Schneider, 2000] and Racer [Haarslev and Moller, 2001] can efficiently deal with problems of considerable size, even for ExpTime DLs.

3.1.3 Translational Approaches

Schild [Schild, 1991] pointed out that many DLs are syntactic variants of modal logics. This fact lead to the application of inference procedures from modal logics to their counter parts in description logics. This approach has been refined for more expressive DLs and a number of (worst-case) optimal decision procedures for very expressive DLs were obtained by sophisticated translation into PDL [De Giacomo and Lenzerini, 1994] or to modal µ-calculus [de Giacomo et al., 1994]. But the experiments in [Horrocks et al., 2000b] indicate that it will be very difficult to obtain implementations based on these kinds of translations. A different approach involves translation of DL into FOL. Since FOL is undecidable in general, a decision procedure can not be obtained by simply expressing the inference problems of DLs in terms of the satisfiability problems for (extensions of) FOL. Another variation of this approach involves a non-standard translation of DLs into FOL and then using the Resolution based theorem proving to obtain a decision procedure [Hustadt and Schmidt, 2000; Hustadt et al., 2004]. Performance studies [Horrocks et al., 2000a] indicate that the translational approach leads to an acceptable but inferior run-time performance compared to tableau algorithms.

3.1.4 Automata-based Approach

Several description logics and modal logics exhibit the tree model property, i.e., every satisfiable concept has (under a suitable abstraction) a tree shaped model. It can be (possibly) infinite since unravelling of a graph will result in an infinite tree if it contains cycle. Tree model property makes it possible to reduce the satisfiability of a concept to the existence of a tree with certain properties dependent on the formula. In cases, where it is feasible to capture these properties using a tree automaton [Gécseg and Steinby, 1984], satisfiability and hence, subsumption of the logic can be reduced to the emptiness problem of the corresponding class of tree automata [Vardi and Wolper, 1984] (see Section 4.3). Automata-based algorithms first construct a non-deterministic tree automaton or an alternating tree automaton from the input DL concept. Non-emptiness of the language accepted by the automaton is then used to decide the satisfiability of the input concept. Generally, the upper complexity bounds are easily established using automata-based techniques. In particular, for the expressive DLs with the ExpTime inference problems, where it is difficult to obtain the tableau algorithms with optimal complexity, exact complexity results can be obtained elegantly using automata-based approach [Calvanese et al., 1999]. Still, there doesn’t exists a pragmatic implementation based on automata algorithms, since this approach usually involves an exponential construction step of automata, which occurs in every case. Recently,
it has been shown that it is possible for the $\mu$-calculus to construct the automata from the input formula incrementally, thus avoiding the need of construction of exponential sized (in the formula) automata in the first step itself [Unel and Toman, 2007a]. This incremental approach of automata construction has not yet been established for Description Logics. The aim of this thesis is to propose an incremental approach based algorithm for the DL $\mathcal{ALC}$ satisfiability problem.

### 3.2 Conversional Approach

This section will present the conversional approaches proposed to use automata-based algorithms for reasoning. By “conversional approach” we denote those approaches in which a conversion procedure from one algorithm to another is developed, so that a single algorithm can be used for establishing theoretical complexity results as well as for practical implementation.

The proposal of translating Tableau algorithms to Automata algorithms was first presented in [Baader, 2003]. The intuition behind this translation is to save the overhead in developing two algorithms for the same logic, one for proving the exact upper complexity bounds and another one for a practical implementation of the reasoning procedures. Tableau algorithms are considered bad in proving the ExpTime upper complexity bounds for expressive DLs while these bounds can be easily established by automata-based algorithms. But it leads to the problem of developing two algorithms for the same logic. A tableau-based algorithm for implementation and an automata-based algorithm for proving the complexity bounds. Baader et al. investigates the relationship between automata and tableau-based algorithms in [Baader, 2003] with an aim of developing a framework for conversion of tableau-based algorithms to automata-based algorithms.

The core notion of this approach is the tableau system. The purpose of a tableau system is to capture all the details of a tableau algorithm. The set $\mathcal{T}$ of the input consists of all the pairs $(C, T)$ of the input concept $C$ and the TBox $T$ of the DL under consideration.

**Definition 3.1. (Tableau System)** Let $\mathcal{T}$ be the input. A tableau system for $\mathcal{T}$ is a tuple

$$S = (\mathcal{NLE}, \mathcal{EL}, ^S, R, C)$$

where $\mathcal{NLE}$ and $\mathcal{EL}$ stands for the node label elements and edge label elements respectively and $^S$ is a function, mapping each input $\Gamma \in \mathcal{T}$ to a tuple

$$\Gamma^S = (nle, el, ini)$$

such that

- $nle \subseteq \mathcal{NLE}$ and $el \subseteq \mathcal{EL}$ are finite.
- $ini \subseteq 2^{nle}$.

The definitions of $R$ and $C$ depends on the definition of the notion of $S$-pattern (or $S$-tree) which is a finite labelled tree

$$(V, E, n, l)$$

with depth at most one with $n$: $V \to 2^{\mathcal{NLE}}$ and $l$: $E \to 2^{\mathcal{EL}}$ node and edge labelling functions.
• $R$, the collection of completion rules, is a function, mapping each $S$-pattern to a finite set of non-empty finite sets of $S$-patterns.

• $C$, the collection of clash-triggers, is a set of $S$-patterns.

Using these definitions of tableau system and $S$-tree, an $S$-tree compatible with $\Gamma$ is defined as follows,

**Definition 3.2.** Let $\Gamma$ be the input and $t = (V, E, n, l)$ an $S$-tree with root at $v_0$. Then $t$ is compatible with $\Gamma$ iff it satisfies the following conditions:

• $n(x) \subseteq 2^{nle}$ for each $x \in V$

• $l(x, y) \in cl_S$ for each $(x, y) \in E$

• $\exists \Lambda \in ini_S(\Gamma)$ such that $\Lambda \subseteq n(v_0)$

• the out degree of $t$ is bounded by $2^{|\Gamma|}$

For checking whether an input has a property $P$ (for e.g., satisfiability) or not, it suffices to verify the existence of a saturated, clash-free $S$-tree. This can be done by a reduction to looping automata and checking for emptiness\(^1\). In this way one can design a tableau-based algorithm, use these conversions to obtain an EXPTime upper-bound and then use the same algorithm for the practical implementation. This approach does not lie in our area of interest since we are interested in a pure automata-based decision procedure.

### 3.3 Translational Approach

Many DLs can be translated to weak monadic second order theories of two successors ($WS2S$). Using this translation, an existing reasoner or theorem prover for $WS2S$ can then be used for reasoning in DLs. One such application is the MONA tool [Elgaard et al., 1998]. The translation of the DL $ALC$ into $WS2S$ formulas and evaluating MONA’s performance on these formulas was done in [Karabaev and Lutz, 2004]. The results of this translation were reasonable in the case where no TBoxes were involved. However, in the presence of TBoxes, MONA’s performance was extremely poor, rendering the Mona approach to DL reasoning useless.

#### 3.3.1 BDD-based Decision Procedure for K

Intuitively, it is clear that to get a practical reasoning time from an automata based algorithm one has to avoid the exponential construction involved in transforming the input to an automaton. So, the approach suggested by Pan et al. [2002] uses on-the-fly emptiness checking of the automaton. They have used Binary Decision Diagrams (BDDs) to represent and manipulate fix points of a set of types and investigate different kinds of representations as well as a level-based representation scheme. Although, promising results are presented in Pan et al. [2002], the early attempts made by researchers\(^2\) to translate the DL $ALC$ into monadic formulas and then using a BDDs based tool for reasoning ended in a futile attempt. But the BDDs can be used to efficiently represent automata in an implementation of a reasoning tool based on automata techniques.

---

\(^1\)Details of the reduction can be found in [Baader, 2003].

\(^2\)Gulay Unel and David Toman, University of Waterloo, Canada.
3.4 Modal $\mu$-calculus Approach

In this section we will discuss about the decision procedures for $\mu$-calculus involving the use of automata-based algorithms.

3.4.1 Encoding into TWATA

In Calvanese et al. [1999] an automata-based decision procedure for DL with fixpoints was proposed. Fixpoints on concepts in their full generality have been investigated in [Schild, 1993; DeGiacomo and Lenzerini, 1997] on the lines of modal $\mu$-calculus. But those logics lacked inverse roles which are essential to deal with $n$-ary relations. So, the work in [Calvanese et al., 1999] addresses these issues and included all the constructs mention above. The reasoning in DL with fixpoints was found to be ExpTime-complete. A decision procedure based on reducing the inference to non-emptiness of two-way alternating automata on infinite tress is presented by the authors. But this encoding of an expressive DL into TWATA doesn’t addressed the issue of exponential blowup while translating the alternating automata to non-deterministic automata for non-emptiness checking. However, it should be noted that the encoding of input into TWATA have only polynomial complexity.

3.4.2 Incremental Decision Procedure

A novel idea for incremental construction of automata was presented by Kupferman and Vardi [2005]. The highlight of this approach was that they avoided the use of Safra’s construction [Safra, 1988], which over the years, proved quite resistant to the efficient implementation [Tasiran et al., 1995]. This idea was further developed by Unel and Toman [2007a] and they showed how to construct an automata to check the $\mu$-calculus satisfiability problem in an incremental fashion and without the need for re-computation of automata transitions. The authors aimed at an early detection of an inconsistency in the set of theories $\{\varphi_1, \ldots, \varphi_n, \neg \varphi\}$, by proposing an incremental and interleaved approach towards constructing the automaton while simultaneously testing for satisfiability of the so far constructed fragments. This incremental approach involves several steps. First, the decision procedure is split into a sequence of simpler problems, so that the larger problems can be constructed from the simple ones while avoiding re-computation. Second, checking whether formula $\varphi_i$ is satisfiable is further divided into the following steps:

1. construct an alternating automaton $A_\varphi$ that accepts tree models of $\varphi$,
2. reduce $A_\varphi$ to a universal automaton $A'_\varphi$,
3. reduce $A'_\varphi$ to a nondeterministic tree automaton $A''_\varphi$,
4. check that the language of $A''_\varphi$ is nonempty.

These steps can be interleaved into each other since explicit construction of complete automaton is not required if one performs a top-down emptiness test, thus yielding an incremental satisfaction decision procedure.

3.5 Inverse Method Approach

In Baader and Tobies [2001] the Vornokov’s inverse method for deciding the satisfiability of modal logic $K$ is presented as an on-the-fly realization of the emptiness test done by
the automata approach for $\mathbf{K}$. Thus the inverse method yields an ExpTime algorithm for satisfiability in $\mathbf{K}$ w.r.t global axioms. The authors have proposed a bottom-up algorithm for the non-emptiness checking of an automaton. The algorithm works by computing inactive states. A state $q \in Q$ is active iff there exists a tree $t$ and a run of $A$ on $t$ in which $q$ occurs: otherwise $q$ is inactive. A looping tree automata accepts a tree iff it has an active initial state. In spite of avoiding exponential overhead by the on-the-fly construction of the automata this approach is rendered useless in the presence of terminological cycles which can lead to an infinite tree in DL $\mathcal{ALC}$ with cyclic TBoxes.

3.6 MONA

MONA is a tool that translates formulas to finite-state automata. The formulas may express search patterns, temporal properties of reactive systems, parse tree constraints, etc. MONA analyzes the automaton resulting from the compilation and prints out ”valid” or a counter-example [BRICS]. MONA translates WS1S and WS2S formulas into minimum DFAs (Deterministic Finite Automata) and GTAs (Guided Tree Automata), respectively. The automata are represented by shared, multi-terminal BDDs.

BDDs represented Automata

A simple generalization of BDDs result in a canonical graph if shared, multi-terminal BDDs are used. These BDDs have multiple leaves, one corresponding to each state. Each state, in turn, is associated with such a BDD, and sharing means that isomorphic subtrees are identified [Klarlund, 1998]. For e.g., the property “there is more than one element in the intersection of $X$ and $Y$” i.e.,

$$\exists p, q : p \neq q \land p \in X \cap Y \land q \in X \cap Y$$

can be represented by the automaton shown in Figure 3.1, here each state $r, s$ or $t$ is described in an array with information about whether it is final and with a pointer to a BDD node defining its transition function.

WS2S, deals with elements and finite subsets of infinite, binary tree. A deterministic,
bottom-up tree automaton can be used to represent the WS2S. Its transition function determines for each pair \((r, s)\) of states and each letter \(a\) what the next state is. Tree automaton is for that reason at-least quadratically more difficult to work with than DFAs.

As stated earlier in Section 3.3, translating a DL into monadic second order theory and using MONA for reasoning didn’t returned encouraging results if TBoxes with terminological cycles were considered. But the BDD-based automaton representation used in MONA can be used as an optimization in any independent automata-based implementation.
CHAPTER 4

Automata for Description Logics

The central idea of this work is to employ automata as devices to solve the decision problem for DLs. If we can reduce the satisfiability problem of the targeted DLs to the emptiness problem of the language accepted by an automaton, then we will get an algorithm matching the upper complexity bound of the targeted DL. We will start by discussing various types of automata. First, we will describe the deterministic and nondeterministic finite state automata on finite strings. Then we will present Büchi automata in Section 4.2.2. We proceed to define the tree automata in Section 4.3. Finally, we discuss alternating tree automata in Section 4.3.3.

4.1 Automata over Finite Strings

In this section, we will introduce automata on finite strings. A string (or word) over alphabet \( \Sigma \) is a finite sequence of characters from \( \Sigma \). For e.g., if \( \Sigma = \{0, 1\} \), then '0101' is a string over \( \Sigma \).

4.1.1 Deterministic Finite State Automata

A Finite State Automaton (FSA) is a computational device whose input is a string and whose output is one of two values that we can call Accept and Reject [Rich, 2008]. If \( A \) is an FSA, an input string is fed to \( A \) one character at a time, left to right. When it receives a character, \( A \) considers its current state, the new character and chooses a next state. One or more of \( A \)'s states may be marked as accepting states. If \( A \) runs out of input and is in an accepting state, it accepts. If, however, \( A \) runs out of input and is not in an accepting state, it rejects.

Definition 4.1. A deterministic FSA (or DFA) \( A \) is a quintuple \((Q, \Sigma, \delta, q_{\text{in}}, F)\), where:

- \( Q \) is a finite set of states,
- \( \Sigma \) is the input alphabet, which is a finite set of symbols,
A configuration of a DFA $A$ is an element of $Q \times \Sigma^*$. It can be thought of as a snapshot of $A$'s computation. A computation by $A$ is a finite sequence of configurations $C_{i-1} \vdash_A C_i$ for $i \in 1, \ldots, n$, where $\vdash_A$ is defined as follows:

$$\forall q \in Q, a \in \Sigma, w \in \Sigma^*: (q, a \cdot w) \vdash_A (p, w) \text{ iff } \delta(q, a) = p.$$ 

We denote with $\vdash_A^*$ the reflexive transitive closure of $\vdash_A$.

- $A$ accepts $w$ iff $(q_{in}, w) \vdash_A^* (q, \epsilon)$, for some $q \in F$. A configuration $(q, \epsilon)$, for any $q \in F$, is called an accepting configuration of $A$.

It follows that $A$ rejects $w$ iff $(q_{in}, w) \vdash_A^* (q, \epsilon)$, for some $q \notin F$. A configuration $(q, \epsilon)$, for any $q \notin F$, is called a rejecting configuration of $A$.

The language accepted by $A$, denoted $L(A)$, is the set of all strings accepted by $A$. For example, consider the language $L = \{w \in \{a, b\}^* | w \text{ contains no more than one } b\}$. $L$ is accepted by the DFA $A$ shown in Figure 4.1. The nodes of the automaton are called states. The edges are labelled by $a \in \Sigma$ and connect a state $q \in Q$ to a state $q' \in Q$ iff $q' \in \delta(q, a)$. The initial state is marked with an arrow and a final state is identified by a doubly circled node.

### 4.1.2 Nondeterministic FSA

**Definition 4.2.** A nondeterministic FSA (or NFA) $A_N$ is a quintuple $(Q, \Sigma, \delta, q_{in}, F)$, where:

- $Q$ is a set of states,
- $\Sigma$ is an alphabet,
- $q_{in} \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states, and
- $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\})) \times Q$ is the transition relation.
A configuration is defined analogously as one for DFA in the previous section. Let \( w \) be an element of \( \Sigma^* \). For an NFA \( A_N \), we can define \( \vdash_{A_N} \) as for DFAs, except that the condition \( \delta(q, a) = p \) is replaced by \( (q, a, p) \in \delta \), where additionally \( (q, \epsilon, q) \in \delta \). Then we say that:

- \( A \) accepts \( w \) iff at least one of its computations accepts.

It follows that \( A \) rejects \( w \) iff none of its computational accepts.

The language accepted by \( A_N \), denoted \( L(A_N) \), is the set of all strings accepted by \( A_N \). The primary difference between a DFA and NFA is that in every configuration a DFA can make exactly one transition to move to the next state while this is not necessarily true for an NFA. For e.g., consider the language \( L = \{ w \in \{a, b\}^* \mid w \text{ is made up of an optional 'a' followed by 'aa' followed by zero or more b's} \} \). The NFA \( A_N \) shown in Figure 4.2 accepts \( L \).

### 4.2 Automata over Infinite Strings

In this section we will discuss about automata on infinite strings. An infinite string over \( \Sigma \) is an infinite sequence of characters from \( \Sigma^\omega \), i.e., \( a_0a_1\ldots \in \Sigma^\omega \).

#### 4.2.1 Looping Automata

**Definition 4.3.** A *looping word automaton* over a alphabet \( \Sigma \) is of the form \( A = (Q, \Sigma, q_{in}, \Delta) \), where:

- \( Q \) is a finite set of states,
- \( \Sigma \) is the input alphabet,
- \( q_{in} \subseteq Q \) is the set of initial states,
- \( \Delta \subseteq Q \times \Sigma \times Q \) is the transition relation.

A looping automaton doesn’t have a set of final states, i.e., in looping automaton, an input is accepted if the automaton doesn’t reach a “dead end”, a state for which there is no transition defined in the current state.

**Definition 4.4.** A run of \( A \) on an infinite word \( w = a_0a_1\ldots \in \Sigma^\omega \) is a function \( \rho : \mathbb{N} \rightarrow Q \) such that \( \rho(0) = q_{in} \) and \( \rho(i + 1) \in \delta(\rho(i), a_i) \) for all \( i \in \mathbb{N} \).

A run \( \rho \) of \( A \) is accepting if there exists one. The language accepted by \( A \), \( L(A) \), is the set of words \( w \) such that there exists a successful run of \( A \) on \( w \).
4.2.2 Büchi Automata

Büchi automata were first introduced by Büchi in [Büchi, 1960] for obtaining a decision procedure for the monadic second-order theory of structures with one successor.

**Definition 4.5.** A (nondeterministic) Büchi word automaton over an alphabet $\Sigma$ is a quintuple $A = (Q, \Sigma, \delta, q_{in}, F)$, where:

- $Q$ is a finite non empty set of states,
- $q_{in} \in Q$ is the initial state,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

A Büchi automaton $A$ can have more than one initial states and it doesn’t allows for $\epsilon$ transitions. It can be represented as an edge-labelled directed graph. Figure 4.3 shows a Büchi automaton over an alphabet $\Sigma = \{a, b, c\}$. The automaton operates on infinite input words. It starts in its initial state and proceed by nondeterministically choosing, when in a state $q \in Q$ and reading $a \in \Sigma$, a successor state among those in $\delta(q, a)$.

**Definition 4.6.** A run of $A$ on an infinite word $w = a_0a_1\ldots \in \Sigma^\omega$ is a function $\rho : \mathbb{N} \to Q$ such that $\rho(0) = q_{in}$ and $\rho(i + 1) \in \delta(\rho(i), a_i)$ for all $i \in \mathbb{N}$.

**Example 4.1.** A run of the automaton shown in Figure 4.3 on the word $a(baba)^\omega$ is given by the sequence $q_0q_1q_2q_3q_4^\omega$.

The set of states visited infinitely often in a run $\rho$, denoted by $\text{Inf}(\rho)$, is defined as:

$$\text{Inf}(\rho) = \{ q \in Q \mid \text{for infinitely many } k \in \mathbb{N}, \text{ we have } \rho(k) = q \}$$

A run $\rho$ of $A$ is accepting iff $\text{Inf}(\rho) \cap F \neq \emptyset$. The language accepted by $A$, $\mathcal{L}(A)$, is the set of words $w$ such that there exists a successful run of $A$ on $w$. 

![Figure 4.3: A Büchi Automaton](image-url)
4.3 Automata over Infinite Trees

4.3.1 Looping Tree Automata

Definition 4.7. A (Nondeterministic) Looping Tree automaton (NLT) over k-ary trees is of the form \( A = (Q, \Sigma, q_\text{in}, \delta) \), where:

- \( Q \) is a finite set of states,
- \( \Sigma \) is the input alphabet,
- \( q_\text{in} \subseteq Q \) is the set of initial states,
- \( \delta : Q \times \Sigma \rightarrow 2^{Q^{k+1}} \) is the transition function.

Definition 4.8. A run of this automaton on a k-ary tree \( t \) is a labelled k-ary tree \( r : K^* \rightarrow Q \) such that \( (r(v_1), \ldots, r(v_k)) \in \delta(r(v), \tau(v)) \) for all \( v \in K^* \).

The run is successful if \( r(\epsilon) \in q_\text{in} \). The language accepted by \( A \), \( \mathcal{L}(A) \), is the set of all trees \( t \) such that there exists a successful run of \( A \) on \( t \).

Example 4.2. Figure 4.5 shows an example of looping tree automaton that accepts the language \( \mathcal{L}(A) = \{ t \mid a \text{ never occurs below a } b \text{ in } t \} \), where the automaton \( A = (Q, \Sigma, I, \delta) \) is defined as follows:

- \( Q = \{ q_a, q_b \} \),
- \( \Sigma = \{ a, b \} \),
- \( I = \{ q_a, q_b \} \),
• The transition function $\delta$ is defined as follows:
  \begin{align*}
    - \delta(q_a, a) &= \{(q, q')\} \text{ for } q, q' \in Q, \\
    - \delta(q_b, b) &= \{(q_b, q_b)\}.
  \end{align*}

Figure 4.5: Looping Tree Automaton

4.3.2 Büchi Tree Automata

Definition 4.9. A Büchi tree automaton over an alphabet $\Sigma$ is a quintuple $A = (Q, \Sigma, \delta, q_{in}, F)$, where:

- $Q$ is a finite non empty set of states,
- $q_{in} \in Q$ is the set of initial states,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta \subseteq Q \times \Sigma \times Q^k$ is the transition relation.

Definition 4.10. A run of $A$ on tree $t$ is a function $\rho : \mathbb{N} \rightarrow Q$ such that $\rho(0) = q_{in}$ and $\rho(i + 1) \in \delta(\rho(i), a_i)$ for all $i \in \mathbb{N}$. A run $\rho$ of $A$ is accepting iff $\text{Inf}(\rho) \cap F \neq \emptyset$. The language accepted by $A$, $\mathcal{L}(A)$, is the set of trees $t$ such that there exists a successful run of $A$ on $t$.

4.3.3 Alternating Tree Automata

Alternating tree automata extend the tree automata by universal choices. The transition function no longer denotes a set of successor states but a (positive) boolean combination. Alternating tree automata were introduced by Muller and Schupp in [David and Paul, 1987].

Definition 4.11. For a finite set $X$ of variables, let $B^+(X)$ be the set of positive Boolean formulas over $X$, i.e., the smallest set such that

- $X \subseteq B^+(X)$,
- true, false $\in B^+(X)$.
- if $\varphi, \psi \in B^+(X)$ then $\varphi \land \psi \in B^+(X)$ and $\varphi \lor \psi \in B^+(X)$.
Let $K_0 = \{0, 1, \ldots, k\}$. For a word $w \in K_0^*$ and $c \in K_0$, by convention, we have $w \cdot 0 = w$.

**Definition 4.12.** An *Alternating Looping Tree automaton* (ALT) $A$ is a tuple $(Q, \Sigma, q_{in}, \delta)$, where:

- $Q$ is a set of states,
- $\Sigma$ is the input alphabet,
- $q_{in} \in Q$ is the initial state,
- $\delta : Q \times \Sigma \rightarrow B^+(K_0 \times Q)$ is the transition function.

The width $w(A)$ of an automaton $A$ is the number of literals that can appear on the right-hand side of a transition [Hladik, 2008].

**Definition 4.13.** A run $r$ of $A$ on a tree $t$ is a $w(A)$-ary infinite tree over $(K_0^* \times Q) \cup \{\#\}$ such that, for each node $x$ with $r(x) = (v, q) \neq \#$. For $\delta(q, \tau(v)) = \varphi$, there is a set $S = \{(v_1, q_1), \ldots, (v_n, q_n)\} \subseteq K_0 \times Q$ that satisfies the following conditions:

1. $S$ satisfies $\varphi$
2. For all $1 \leq i \leq n$, $r(x \cdot i) = (v \cdot v_i, q_i)$.

A run $r$ is successful if $r(\epsilon) = (\epsilon, q_0)$ holds. The language accepted by $A$, $L(A)$, is the set of all trees $t$ such that there exists a successful run of $A$ on $t$.

The automaton $A$ is called *Universal Looping Tree automata* (ULT), if all the formulas that appear in $\delta$ are conjunctions of atoms in $K_0 \times Q$ [Kupferman and Vardi, 2005; Muller and Schupp, 1995]. Notice that an NLT can also be represented as an ALT in which, for all formulas $\varphi$ that appear in $\delta$, if $(v_1, q_1)$ and $(v_2, q_2)$ are conjuncts in the disjunctive normal form of $\varphi$, then $v_1 \neq v_2$ (i.e., if the transition is rewritten in disjunctive normal form, then in each disjunct there is at most one element of $\{v\} \times Q$, for each $v \in K_0$).

**Example 4.3.** Let $A = (Q, \Sigma, q_{in}, \delta)$ where

- $Q = \{q_0, q_1, q_2, q_3\}$,
- $\Sigma = \{a, b, c\}$,
- $q_{in} = q_0$,
- The transition function $\delta$ is defined as follows:
  
  $- \delta(q_0, a) = ((0, q_3) \land (2, q_1)) \lor (3, q_2)$
  $- \delta(q_1, b) = (0, q_0) \land (3, q_3)$
  $- \delta(q_3, a) = \delta(q_0, b) = \delta(q_0, c) = true$

Figure 4.6 shows a successful run $r$ of $A$ on input tree $t$. 
Figure 4.6: Accepting Run of Alternating Automaton
In this chapter we will present the incremental reasoning approach. First, we will discuss the automata construction algorithms for DL \( \mathcal{ALC} \). Then we will present various automata translations and methods to decompose them. In section 5.3 we will present the incremental satisfaction algorithm. We will conclude this chapter by summarizing the incremental reasoning approach with the help of an example.

5.1 NTA for ALC with General TBoxes

We will use infinite \( k \)-ary trees as data structures on which non-deterministic tree automata will operate. Propositionally expanded and clash-free sets of concepts, so called Hintikka sets, can be used to label nodes of the trees accepted by an automaton. The idea for using Hintikka sets is justified by the fact that we do not need to consider nodes whose label contains an obvious contradiction (also called a clash) because in a model \( I \), every individual will belong to either \( C^I \) or to \( (\neg C)^I \).

**Definition 5.1.** (Sub-concept): Let \( C \) be an \( \mathcal{ALC} \) concept. The set of sub-concepts of \( C \), denoted by \( \text{sub}(C) \), is the minimal set \( S \) that contains \( C \) and has the following properties:

- if \( \neg A \in S \) then \( A \in S \),
- if \( D \cup E \in S \) or \( D \cap E \in S \) then \( \{D, E\} \subseteq S \),
- if \( \exists r.D \in S \) or \( \forall r.D \in S \) then \( D \in S \).

For a TBox \( T \), the concept \( C_T \) is defined as

\[
C_T = \bigcap_{C \subseteq D \in T} \neg C \cup D
\]

**Definition 5.2.** (Hintikka set): Let \( C \) be an \( \mathcal{ALC} \) concept and \( T \) be a TBox. A set \( H \subseteq \{\text{sub}(C) \cup \text{sub}(C_T)\} \) is called a Hintikka set if the following conditions are satisfied:
• if \( D \cap E \in H \), then \( \{D, E\} \subseteq H \),

• if \( D \cup E \in H \), then \( \{D, E\} \cap H \neq \emptyset \),

• there is no concept name \( A \) with \( \{A, \neg A\} \subseteq H \).

A Hintikka set \( H \) is called \( T \)-expanded if, for every GCI \( D \sqsubseteq E \in T \), it holds that \( \neg D \sqcup E \not\in H \).

**Definition 5.3.** (Hintikka Tree):

For a concept \( C \) and TBox \( T \), fix an ordering of the existential concepts in \( \text{sub}(C) \cup \text{sub}(C_T) \) and let \( \varphi : \{\exists r.D : \exists r.D \in \{\text{sub}(C) \cup \text{sub}(C_T)\}\} \rightarrow K \) be the corresponding ordering function, where \( K \) is the set \( \{1, \ldots, k\} \). Then, the tuple \( (\tau_0, \tau_1, \ldots, \tau_k) \) is called \( C, T \)-compatible if, for all \( 0 \leq i \leq k \), \( \tau_i \) is a \( T \)-expanded Hintikka set and, for every existential restriction \( \exists r.D \in \{\text{sub}(C) \cup \text{sub}(C_T)\} \) with \( \varphi(\exists r.D) = i \), the following holds

• if \( \exists r.D \in \tau_0 \), then
  - \( \tau_i \) contains \( D \),
  - \( \tau_i \) contains all concepts \( E_j \) for which there is a value restriction \( \forall r.E_j \in \tau_0 \),

• if \( \exists r.D \not\in \tau_0 \), then \( \tau_i = \# \).

Nodes labeled with \( \# \) are called dummy nodes.

**Definition 5.4.** (NTA for \( \mathcal{ALC} \)) For an \( \mathcal{ALC} \) concept \( C \) and a TBox \( T \) with \( k \) existential restrictions in \( \{\text{sub}(C) \cup \text{sub}(C_T)\} \), fix an ordering of these existential restrictions and let \( \varphi : \{\exists r.D : \exists r.D \in \{\text{sub}(C) \cup \text{sub}(C_T)\}\} \rightarrow K \) be the corresponding ordering function. Then the looping tree automaton \( A_{C, T} = (Q, \Sigma, I, \delta) \) is defined as follows:

- \( Q = \Sigma = \{\tau \in 2^\{\text{sub}(C) \cup \text{sub}(C_T)\} | \tau \text{ is a } T \text{-expanded Hintikka set}\} \cup \{\#\}, \)

- \( \delta \) consists all tuples \( (\tau_0, \tau_1, \ldots, \tau_k) \) that are \( C, T \)-compatible,

- \( I = \{\tau \in Q | C \in \tau\} \).

From previous definition we now have,

**Theorem 5.1.** The language \( \mathcal{L} \) of nondeterministic looping tree automaton \( A_{C, T} \), defined over concept \( C \) and TBox \( T \), is empty iff concept \( C \) is unsatisfiable w.r.t \( T \).

### 5.2 ALT for ALC with General TBoxes

In the following section, an \( \text{ExpTime} \) algorithm for the satisfiability problem of \( \mathcal{ALC} \) concepts w.r.t. general TBoxes that uses alternating automata is presented. Similar algorithms for various DLs were developed by [Calvanese et al., 1999] and [Calvanese et al., 2002].
5.2.1 Handling TBoxes

We handle the TBox axioms in the transition conditions of the automata by ensuring that input concept \( C_0 \) holds at the root of a tree-model, and that \( C_T \) is propagated to every node of the tree.

**Definition 5.5.** (ALT for ALC concept w.r.t. TBox): Let \( C \) be an ALC concept and \( T \) a TBox in negation normal form, with \( k \) being the number of existential restrictions in \( \text{sub}(C) \cup \text{sub}(C_T) \). Let \( N_c(C, T) \) be the set of concept names occurring in \( C \) or \( T \), and let \( R_c(C, T) = \{s_r \mid r \text{ occurs in } C \text{ or } T\} \). The ALT \( A_{C, T} = (Q, \Sigma, q_{\text{in}}, \delta) \) running over \( k \)-ary trees is defined as:

- \( Q = \text{sub}(C) \cup \text{sub}(C_T) \cup R_c(C, T) \cup \{s_r \mid s_r \in R_c(C, T)\} \cup \{\text{Start}, \text{GCI}, \#\}, \)
- \( \Sigma = \{S_c \cup \{r\} \mid S_c \subseteq N_c(C, T), r \in R_c(C, T)\} \cup \{\#\}, \)
- \( q_{\text{in}} = \text{Start}, \)
- \( \delta(q, \sigma) \) for each \( q \in Q \) and each \( \sigma \in \Sigma \), is defined in Table 5.1.

**Theorem 5.2.** The language \( L \) of alternating looping tree automaton \( A_{C, T} \), defined over concept \( C \) and TBox \( T \), is empty iff concept \( C \) is unsatisfiable w.r.t \( T \).

5.3 Incremental Automata Construction

In this section, we will present the incremental approach for constructing the target automaton and checking its emptiness incrementally.

The idea for incremental construction of automata was presented by Kupferman and Vardi [2005]. Highlight of this approach was that they avoided the use of Safra’s construction [Safra, 1988], which over the years, proved quite resistant to the efficient implementation [Tasiran et al., 1995]. This idea was further developed by Unel and Toman [2007a] and they showed how to construct an automaton to check the \( \mu \)-calculus satisfiability problem in an incremental fashion and without the need for re-computation. But both of these papers dealt with the alternating parity automaton since they were dealing with more expressive logics. In order to use this approach in our case, a similar mechanism for ALT needed to be devised. In the following, an incremental construction of ALT to ULT transformation is discussed.

5.3.1 From ALT to ULT

In the following we present the translation of ALT to ULT. The intuition behind the translation is to remove the nondeterminism in \( \delta \) by choosing the atoms that are going to be satisfied. So, ULT are special case of ALT i.e., the transition function of ULT contains only conjunctions.

**Definition 5.6.** (ALT to ULT) : Consider an ALT \( A = (\Sigma, Q, q_{\text{in}}, \delta) \), where \( \delta : Q \times \Sigma \rightarrow B^+(K_0 \times Q) \). A restriction of \( \delta \) is a partial function \( \eta : Q \rightarrow 2^{K_0 \times Q} \). A restriction \( \eta \) is relevant to \( \sigma \in \Sigma \) if for all \( q \in Q \) for which \( \delta(q, \sigma) \) is satisfiable and \( \eta(q) \) satisfies \( \delta(q, \sigma) \).
For each $\sigma \subseteq N_C(\mathcal{C}, T)$, i.e., containing no basic role, there is a transition from the initial state

$$\delta(\text{Start}, \sigma) = (0, C) \land (0, \text{GCI})$$

For each $\sigma \subseteq N_C(\mathcal{C}, T)$ there is a transition

$$\delta(\text{GCI}, \sigma) = (0, C_T) \land \bigwedge_{i=1}^{k} ((i, \text{GCI}) \lor (i, \#))$$

For each $\sigma \in \Sigma$ and each $s_r \in R_c(\mathcal{C}, T)$ there are transitions

$$\delta(s_r, \sigma) = \begin{cases} true, & \text{if } s_r \in \sigma \\ false, & \text{otherwise} \end{cases}$$

$$\delta(\overline{s_r}, \sigma) = \begin{cases} true, & \text{if } s_r \not\in \sigma \\ false, & \text{otherwise} \end{cases}$$

For each $\sigma \in \Sigma$ and each $D \in N_C(\mathcal{C}, T)$ there are transitions

$$\delta(D, \sigma) = \begin{cases} true, & \text{if } D \in \sigma \\ false, & \text{otherwise} \end{cases}$$

$$\delta(\neg D, \sigma) = \begin{cases} true, & \text{if } D \not\in \sigma \\ false, & \text{otherwise} \end{cases}$$

For the $q \in Q$ (that is not an atomic concept or a basic role), and each $\sigma \in \Sigma$, there are transitions

$$\delta(D \cap E, \sigma) = (0, D) \land (0, E)$$

$$\delta(D \cup E, \sigma) = (0, D) \lor (0, E)$$

$$\delta(\exists r. D, \sigma) = \bigvee_{i=1}^{k} ((i, s_r) \land (i, D))$$

$$\delta(\forall r. D, \sigma) = \bigwedge_{i=1}^{k} ((i, \overline{s_r}) \lor (i, D) \lor (i, \#))$$

$$\delta(\#, \sigma) = \begin{cases} true, & \text{if } \sigma = \# \\ false, & \text{otherwise} \end{cases}$$

$$\delta(q, \#) = \begin{cases} true, & \text{if } q = \# \\ false, & \text{otherwise} \end{cases}$$

Table 5.1: ALT Transition Relation
Let $R$ be the set of restrictions of $\delta$. For $A = (\Sigma, Q, q_{in}, \delta)$, the ULT is defined as $A' = (\Sigma', Q, \langle 0, q_0 \rangle, \delta')$ where:

- $\Sigma' = \{ (\sigma, \eta) \in \Sigma \times R \mid \eta \text{ is relevant to } \sigma \}$
- For all $q \in Q$, $\sigma \in \Sigma$, and $\eta \in R$,
  \[
  \delta'(q, \langle \sigma, \eta \rangle) = \bigwedge_{(c, s) \in \eta(q)} (c, s)
  \]

A running strategy of $A$ for a $\Sigma$-labeled tree $\langle t, \tau \rangle$ is a $R$-labeled tree $\langle t, f \rangle$. We say that $\langle t, f \rangle$ is relevant to $\langle t, \tau \rangle$ if for all $x \in t$, the restriction $f(x)$ is relevant to $\tau(x)$. A running strategy $\langle t, f \rangle$ is good for $\langle t, \tau \rangle$ if $\langle t, f \rangle$ is relevant to $\langle t, \tau \rangle$ and the run on $R$-labeled tree $\langle t, f \rangle$ is accepting [Kupferman and Vardi, 2005].

**Theorem 5.3.** The ALT $A$ accepts $\langle t, \tau \rangle$ iff there exists a running strategy $\langle t, f \rangle$ that is good for $\langle t, \tau \rangle$.

The proof is similar to the one presented in Emerson and Jutla [1991] for alternating automata with parity acceptance condition.

### 5.3.2 Decomposing ALT to ULT Translation

Let ALT $A = (\Sigma, Q, q_i, \delta)$ accepts the tree models of $\varphi$. This automaton can be constructed as in Section 5.2. The following theorem states that we can reuse the transitions computed for a ULT $A_k'$ for a subformula $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_k$ of $\varphi$ in the computation of the ULT $A_{k+1}'$ for $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_{k+1}$.

**Theorem 5.4.** Let $\varphi' = \varphi_1 \land \varphi_2 \land \ldots \land \varphi_k$ be a subformula of $\varphi$. The ALT $A_k$ for $\varphi'$ is of the form $A_k = (\Sigma_k, Q_k, q_{ik}, \delta_k)$. Let $A'_k = (\Sigma'_k, Q_k, q_{ik}, \delta'_k)$ be the ULT translation of $A_k$ and $A' = (\Sigma', Q, \langle q_i, 0 \rangle, \delta')$ be the ULT translation of $A$. Then for every $q_k \in Q_k$, $\sigma_k \in \Sigma_k$, $\eta_k \in R_k$, $\eta_l \in R_l$:

\[
\delta'(q_k, \langle \delta_k, \eta_k \cup \eta_l \rangle) = \delta'(q_k, \langle \delta_k, \eta_k \rangle)
\]

where $R_k$ is the set of restrictions $\eta_k : Q_k \to 2^{K_0 \times Q}$ such that for all $\langle \delta_k, \eta_k \rangle \in \Sigma'_k$, $\eta_k$ is relevant to $\delta_k$, and $R_l$ is the set of restrictions $\eta_l : Q \setminus Q_k \to 2^{K_0 \times Q}$.

The proof is similar to the one presented in Unel and Toman [2007a] for alternating automata with parity acceptance condition.

### 5.3.3 From ULT to NLT

The translation of ULT to NLT is presented in the following definition. This translation was originally described by Miyano and Hayashi [1984] for translating Alternating Büchi Word automata (ABW) to Nondeterministic Büchi Word automata (NBW). A technical variant of this construction adapted to tree automata was presented in [Kupferman and Vardi, 2005]. The translation presented below is devised for looping automata considering the above translations as reference.
The intuitive idea behind the ULT to NLT translation is that if the universal automaton have the transitions $\delta'(q, \sigma) = (0, q_1) \land (1, q_2)$ or $\delta'(q, \sigma) = (0, q_3) \land (2, q_4)$, where $\{0, 1, 2\}$ is the set of directions, then $\delta''(q, \sigma) = \{\{(q_1), \#\}, \{(q_3), \#, \{q_4\}\}\}$. Here, $\#$ represents the dummy node since there is no transition in the direction $c = 2$ or $c = 1$, where $c \in \{0, 1, 2\}$. Before proceeding further we need to define the following notation. Let $A' = (\Sigma', Q, q_{in}, \delta')$ with $n$ states. For a state $q \in Q$ and symbol $\sigma \in \Sigma$, let $\text{sat}(q, \sigma)$ be the set of subsets of $K_0 \times Q$ that satisfy $\delta'(q, \sigma)$. Let $\gamma = \{(d_1, q_1), \ldots, (d_n, q_n)\}$. We define,

$$\Lambda(\gamma) = \{(Q_0, \ldots, Q_k) | Q_i = \begin{cases} \#, & \text{if for no } j \in \{1, \ldots, n\}, d_j = i \\ \{q_j \mid d_j = i\}, & \text{otherwise} \end{cases} \text{ for } i \in \{0, \ldots, k\}\}$$

**Definition 5.7.** (ULT to NLT) Let $A' = (\Sigma', Q, q_{in}, \delta')$ be the ULT with $n$ states, then $A'' = (\Sigma', Q'', q_{in}', \delta'')$ is the resulting NLT where:

- $Q'' = 2Q \cup \{\#\}$,
- $q_{in}' = \{q_{in}\}$, where $q_{in}$ appears in literal $(0, q_{in})$,
- The transition relation $\delta''$ is defined as follows:

$$\delta''(S, \sigma) = \{\Lambda(\gamma) \mid \gamma \in \text{sat}(q, \sigma)\}$$

### 5.3.4 Decomposing ULT to NLT Translation

The following theorem states that we can reuse the transitions computed for a NLT $A''_k$ for $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_k$ in the computation of $A''_{k+1}$ for $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_{k+1}$.

**Theorem 5.5.** Let $A''_k = (\Sigma_k', Q''_k, q''_{in}, \delta''_k)$ be the NLT translation of $A'_k$ (i.e., for $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_k$), and $A'' = (\Sigma', Q'', q''_i, \delta'')$ be the NLT translation of $A'$. Then for all $S \in Q''$, $\sigma' = (\sigma_k, \eta_k) \in \Sigma_k'$, $\sigma = (\sigma_k, (\eta_k \cup \eta_i) \in \Sigma'$, $\delta''(S, \sigma) = \delta''(S, \sigma')$.

The proof is similar to the one presented in Unel and Toman [2007a] for alternating automaton with parity acceptance condition.

### 5.4 Incremental Satisfaction Algorithm

The incremental satisfaction algorithm for the satisfaction problem of $\mu$-calculus formulas was proposed by Unel and Toman [2007a]. The algorithm proposed below is technical variant adapted to use looping automata for reasoning in the DL $\mathcal{ALC}$.

Let $A_n$ represents the ALT automaton of the input formula $\varphi = \varphi_1 \land \varphi_2 \land \ldots \land \varphi_n$. The corresponding NLT of $A_n$ is represented by $A''_n$. Let $A_i$ be the ALT for $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_i$ and let $A'_i$ be the ULT translation of $A_i$ and $A''_i$ the NLT translation of $A'_i$. The algorithms works towards checking the emptiness of an automaton $A''_i$ while simultaneously checking the emptiness of the targeted automaton $A''_n$. If some $A''_i$ is empty then the algorithm stops by pointing that the language of $A''_i$ is empty which in turn implies the unsatisfiability of the original formula $(\varphi_1 \land \varphi_2 \land \ldots \land \varphi_n)$ represented by the automaton $A_n$ (see Algorithm 1).
Algorithm 1 Incremental Satisfaction Algorithm

Input: $\text{ALC}$ concept $C_0$

Output: Satisfiable if $L(A) \neq \emptyset$, where $A$ is the ALT of $C_0$

Unsatisfiable otherwise

1: procedure $\text{isa}(C_0)$ $\triangleright$ Incrementally check satisfiability of input concept $C_0$
2: for $i = 1$ to $n$ do
3: construct ALT $A_i$
4: if $i \geq 1$ then
5: construct ULT $A'_i$ using ULT $A'_{i-1}$
6: end if
7: construct NLT $A''_i$
8: if $A''_i$ is not empty then
9: return Satisfiable
10: else
11: return Unsatisfiable
12: end if
13: end for
14: end procedure

5.5 Emptiness Checking Algorithm

Unlike Büchi automata, looping automata don’t have any acceptance condition. Instead an input is accepted if the automaton doesn’t reach a “dead end”, i.e., a state for which there is no successor state or the only successor state is “false”. A bottom-up algorithm for checking the emptiness of an automaton works by computing a set of “bad” states. A state $q \in Q$ is “good” iff there exists a tree $t$ and a run of $A$ on $t$ in which $q$ occurs; otherwise, $q$ is bad. But the bottom-up emptiness test is inefficient since we have to generate the complete automaton before doing any emptiness test. Instead, we can try to solve the emptiness problem by constructing a successful run in a top-down manner i.e., starting with the root node, then applying the transition function to go to successor nodes proceeding in the same way. The top-down approach has the advantage that it does not require the automaton to be explicitly given. It can be constructed on-the-fly during the run of the algorithm. Algorithm 2 describes the procedure (Empty/1) which takes NLT $A$ and returns whether the language accepted by $A$ is empty or set of states reachable from the initial state. The algorithm starts from the initial state and proceeds by nondeterministically choosing a transition from the transitions given by transition relation. Since the input automaton have looping leaf nodes it computes a fixpoint of the set of reachable states for termination. It returns ‘empty’ if all possible transitions from the input state leads to the non looping leaf nodes.

5.6 An Example

The following example illustrates the various steps involved in the incremental satisfaction and will help reader to understand it. Since, the overall construction is quite large we will
Algorithm 2 Emptiness Checking Algorithm

**Input:** NLT $A = (\Sigma, Q, \{q_0\}, \delta)$

**Output:**
- YES if $L(A) \neq \emptyset$
- NO otherwise

1: procedure non-empty($A$)  \Comment{Check emptiness of NLT $A$}
2: Initialize $reachable = \{q_0\}$ and $reachable' = reachable$
3: for each $q$ in $reachable$ do
4:     choose a $trans \in \delta(q, \sigma)$, for some $\sigma \in \Sigma$
5:     for each $q' \in trans$ do
6:         if there exists a transition $\delta(q', \sigma)$ then
7:             $reachable' = reachable' \cup \{q'\}$
8:         else
9:             backtrack to last choice point
10:     end if
11: end for
12: if $reachable = reachable'$ then
13:     return YES
14: else
15:     $reachable = reachable'$
16: end if
17: return NO
18: end procedure
discuss a simple case only.

Consider an $\mathcal{ALC}$ concept $C_0 = \text{Man} \sqcap \text{Woman}$. Let $T$ be a TBox\(^1\) containing the following terminological axioms,

$$\text{Man} \sqsubseteq \text{Human} \sqcap \text{Male}$$

$$\text{Human} \sqcap \text{Male} \sqsubseteq \text{Man}$$

$$\text{Woman} \sqsubseteq \neg \text{Man}$$

Then by unfolding the input concept $C_0$ w.r.t to TBox $T$, we will get, $C_0 = (H \sqcap M \sqcap \neg H) \sqcup (H \sqcap M \sqcap \neg M)$. Here we will use $H$ instead of Human and $M$ instead of Male for brevity. Now, construct the automaton $A_0$ for the input concept $C_0$ using the rules described in Table 5.1. Let $A_0 = (\Sigma, Q, q_{in}, \delta)$, then

- $Q = \text{sub}(C_0) \cup \text{sub}(C_T) \cup R_c(C, T) \cup \{ s_r \mid s_r \in R_c(C, T) \} \cup \{ \text{START}, \text{GCI}, \# \}$,
- $\Sigma = \{ \sigma \mid \sigma \subseteq \{ H, M, \text{Man}, \text{Woman} \} \} \cup \{ \# \}$,
- $q_{in} = \text{START}$,
- Transitions $\delta$ are defined as follows,

$$\delta(\text{START}, \sigma) = (0, ((H \sqcap M \sqcap \neg M) \sqcup (H \sqcap M \sqcap \neg H))) \land (0, \text{GCI})$$

$$\delta((H \sqcap M \sqcap \neg M) \sqcup (H \sqcap M \sqcap \neg H), \sigma) = (0, (H \sqcap M \sqcap \neg M)) \lor (0, (H \sqcap M \sqcap \neg H))$$

Now, we can branch the automaton $A_0$ into $A_1 = (\Sigma_1, Q_1, q_{in1}, \delta_1)$ and $A_2 = (\Sigma_2, Q_2, q_{in2}, \delta_2)$ for $C_1 = (H \sqcap M \sqcap \neg M)$ and $C_2 = (H \sqcap M \sqcap \neg H)$ respectively. We will apply the proposed incremental satisfaction algorithm to automaton $A_1$ first and then to $A_2$. Based on output we will be able to decide the emptiness problem for $A_0$ thus solving the satisfiability problem for input concept $C_0$. Considering, $A_1 = (\Sigma_1, Q_1, q_{in1}, \delta_1)$ we have

- $Q_1 = \{ H \sqcap M, H, M \} \cup \{ \text{Start}_1, \# \}$,
- $\Sigma_1 = \{ \{ H \}, \{ M \}, \{ H, M \} \} \cup \{ \# \}$,
- $q_0 = \text{Start}_1$,
- The transition relation $\delta_1$ is defined as:

$$\delta_1(\text{Start}_1, \sigma) = (0, H \sqcap M),$$

$$\delta_1(H, \{ H \}) = \text{true},$$

$$\delta_1(H, \{ M \}) = \text{false},$$

$$\delta_1(H, \{ H, M \}) = \text{true},$$

$$\delta_1(M, \{ H \}) = \text{false},$$

$$\delta_1(M, \{ M \}) = \text{true},$$

$$\delta_1(M, \{ H, M \}) = \text{true},$$

$$\delta_1(H \sqcap M, \sigma) = (0, H) \land (0, M).$$

Let $A'_1 = (\Sigma'_1, Q_1, q_{in1}, \delta'_1)$ be the ULT translation of $A_1$ where

- $\Sigma'_1 = \Sigma_1 \times R_1$ where $R_1$ is the set of relevant restrictions defined below. Suppose, $\eta_{\sigma}$ represents the restriction related to the alphabet $\sigma \in \Sigma_1$ then,

\(^1\) Please note that the example illustrates procedure using an acyclic TBox only.
\[ \eta_{H,M}(H \cap M) = \{(0, H), (0, M)\}, \]
\[ \eta_{H, M}(H) = \{\text{true}\}, \]
\[ \eta_{H, M}(M) = \{\text{true}\}, \]
\[ \eta_H(H \cap M) = \{(0, H), (0, M)\}, \]
\[ \eta_H(H) = \{\text{true}\}, \]
\[ \eta_H(M) = \text{undefined}, \]
\[ \eta_M(H \cap M) = \{(0, H), (0, M)\}, \]
\[ \eta_M(H) = \text{undefined}, \]
\[ \eta_M(M) = \{\text{true}\}. \]

- The transition relation \( \delta'_1 \) is defined as,
  \[ \delta'_1(H, \langle H, \eta_H \rangle) = \text{true}, \]
  \[ \delta'_1(H, \langle M, \eta_M \rangle) = \text{undefined}, \]
  \[ \delta'_1(H, \langle \{H, M\}, \eta_{H,M} \rangle) = \text{true}, \]
  \[ \delta'_1(M, \langle H, \eta_H \rangle) = \text{undefined}, \]
  \[ \delta'_1(M, \langle M, \eta_M \rangle) = \text{true}, \]
  \[ \delta'_1(M, \langle \{H, M\}, \eta_{H,M} \rangle) = \text{true}, \]
  \[ \delta'_1(H \cap M, \langle \sigma, \eta_\sigma \rangle) = (0, H) \land (0, M). \]

Since there exists a successful run of \( A'_1 \), we will expand our input concept to include \( \neg M \), i.e., now the input concept will be \( C_1 = (H \cap M \cap \neg M) \). Let automaton \( A_{12} = (\Sigma_{12}, Q_{12}, q_{in}, \delta_{12}) \) represents the input concept \( C_1 \) where:

- \( Q_{12} = \{(H \cap \neg M), (M \cap \neg M), (H \cap M \cap \neg M)\} \cup Q_1, \)
- \( \Sigma_{12} = \Sigma_1, \)
- The transition computed during construction of \( A_1 \) will be utilized in constructing \( A_{12} \). However, we will need to compute the following new transitions.

\[ \delta_{12}(\neg M, \{H\}) = \text{true}, \]
\[ \delta_{12}(\neg M, \{M\}) = \text{false}, \]
\[ \delta_{12}(\neg M, \{H, M\}) = \text{false}, \]
\[ \delta_{12}(H \cap \neg M, \sigma) = (0, H) \land (0, \neg M), \]
\[ \delta_{12}(M \cap \neg M, \sigma) = (0, M) \land (0, \neg M), \]
\[ \delta_{12}(H \cap M \cap \neg M, \sigma) = (0, H) \land (0, M) \land (0, \neg M). \]

Let \( A'_{12} = (\Sigma'_{12}, Q_{12}, q_{in}, \delta'_{12}) \) be the ULT translation of \( A_{12} \). Then we have,

- \( \Sigma'_{12} = \Sigma_{12} \times R_{12}, \) where \( R_{12} \) is the set of restrictions relevant to the new alphabet.
Suppose, \( \eta_\sigma \) represents the restriction related to the alphabet \( \sigma \in \Sigma_{12} \) then new restrictions are,

- \( \eta_{H,M}(H \cap \neg M) = \{(0, H), (0, \neg M)\}, \)
- \( \eta_{H,M}(M \cap \neg M) = \{(0, M), (0, \neg M)\}, \)
- \( \eta_{H,M}(H \cap M \sqcup \neg M) = \{(0, H), (0, M), (0, \neg M)\}. \)

\[^2\text{The strategy related to this transition is not defined}\]
\begin{itemize}
  \item \( \eta_{H,M}(\neg M) = \text{undefined} \),
  \item \( \eta_H(H \sqcap \neg M) = \{(0, H), (0, \neg M)\} \),
  \item \( \eta_H(M \sqcap \neg M) = \{(0, M), (0, \neg M)\} \),
  \item \( \eta_H(H \sqcap M \sqcup \neg M) = \{(0, H), (0, M), (0, \neg M)\} \),
  \item \( \eta_H(\neg M) = \text{true} \),
  \item \( \eta_H(H \sqcap \neg M) = \{(0, H), (0, \neg M)\} \),
  \item \( \eta_M(M \sqcap \neg M) = \{(0, M), (0, \neg M)\} \),
  \item \( \eta_M(H \sqcap M \sqcup \neg M) = \{(0, H), (0, M), (0, \neg M)\} \),
  \item \( \eta_M(\neg M) = \text{undefined} \).
\end{itemize}

But \( R_{12} \) will also contain the restrictions \( R_1 \) (see section 5.6). Then, one can easily verify that there exists no relevant restriction which satisfy the following transition relation

\[ \delta_{12}(M \sqcap \neg M, (\sigma, \eta_{\text{sigma}})) = (0, M) \land (0, \neg M) \]

for any \( \sigma \in \Sigma_{12}' \). Proceeding in a similar way, after constructing \( A_2 \) one will arrive at conclusion that there does not exists any relevant strategy for the transition that can lead to a successful run.

\[ \delta_{22}'(H \sqcap \neg H, (\sigma, \eta_{\text{sigma}})) = (0, H) \land (0, \neg H) \]

Since, the languages of both automata \( A_1 \) and \( A_2 \) are empty, the language of the disjunctive automaton \( A_0 \) will necessarily be empty. Thus, leading to the conclusion that the input concept \( C_0 \) is unsatisfiable w.r.t to the TBox \( T \).

Figure 5.1 graphically represents the steps involved in testing the satisfiability of \( ALC \) concept using incremental satisfaction approach.

\begin{center}
\begin{tikzpicture}
  \node (C0) at (0,0) {$C_0 \equiv C_1 \sqcap C_2 \sqcap ... \sqcap C_n$};
  \node (C1) at (-2,-2) {$C_1$};
  \node (C2) at (0,-2) {$C_2$};
  \node (C3) at (2,-2) {$C_n$};
  \node (C4) at (-3,-4) {$C_1'$};
  \node (C5) at (0,-4) {$C_2'$};
  \node (C6) at (3,-4) {$C_n'$};

  \draw[->] (C1) -- (C0);
  \draw[->] (C2) -- (C0);
  \draw[->] (C3) -- (C0);
  \draw[->,dashed] (C1) -- (C4);
  \draw[->,dashed] (C2) -- (C5);
  \draw[->,dashed] (C3) -- (C6);

  \node (ALT A1) at (-4,-6) {ALT \( A_1 \)};
  \node (ULT A1') at (-3,-8) {ULT \( A_1' \)};
  \node (NLT A1'') at (-2,-10) {NLT \( A_1'' \)};

  \node (C1') at (-2,-4) {$C_1'$};
  \node (C2') at (0,-4) {$C_2'$};
  \node (C3') at (2,-4) {$C_n'$};
  \node (C1'') at (-1,-8) {$C_1''$};
  \node (C2'') at (0,-8) {$C_2''$};
  \node (C3'') at (1,-8) {$C_n''$};
  \node (C1''') at (-2,-10) {$C_1'''$};
  \node (C2''') at (0,-10) {$C_2'''$};
  \node (C3''') at (2,-10) {$C_n'''$};

  \node (L) at (-1,-12) {$L(A_1'')$ is Empty?};

\end{tikzpicture}
\end{center}

Figure 5.1: Incremental Satisfaction Approach
In this chapter we will present the implementation of an inference engine for the DL $\mathcal{ALC}$ based on the proposed incremental satisfaction algorithm. We will also discuss about the various issues involved in such an implementation. Towards the end of chapter, we will summarize the performance results of the prototype implementation.

6.1 Implementing the Incremental Satisfaction Algorithm

Various implementations both in the commercial and non-commercial sectors have deployed tableau based reasoning, such as Racer [Haarslev and Moller, 2001], Pellet [Sirin et al., 2007] and FaCT [Horrocks, 1998]. These systems use Lisp, Java or C++ as their implementation language. The use of Prolog for implementing tableaux-based reasoning was proposed by Beckert and Posegga [1995] without targeting DL as the application field. Meissner [2004] describes the implementation of DL $\mathcal{ALCN}$ in Prolog however, the author has implemented the tableaux-based algorithm. In Hechenroeder [2006] the author has presented an optimized version of tableaux reasoning in Prolog with the goal of reducing the number of inferences performed by the earlier tableau-based inference engines implemented in Prolog.

To the best of our knowledge, there has been no other work that implements the DL reasoning with automata-based techniques in Prolog. In fact, there has been no attempt to implement a DL reasoning tool purely based on automata-based algorithms. However, the following efforts in this direction need to be mentioned.

- Baader and Tobies [2001] show that the inverse method [Voronkov, 2000] for deciding satisfiability for $\mathcal{K}$ formulas can be viewed as an implementation of the automata-based approach, which suggests that it could also be useful for DLs.

- The satisfiability algorithm for $\mathcal{K}$ presented by Pan, Sattler, and Vardi [2002], which is based on binary decision diagrams (BDDs), can also be regarded as an optimized automata emptiness test.

Other than these efforts for the modal logic $\mathcal{K}$, the notational variant of DL $\mathcal{ALC}$, there has been no published study of developing an automata-based decision procedure for DL and implementing the same.
6.1.1 Interpretation of Algorithm

The basic idea behind the Incremental Satisfaction Algorithm (ISA) is the need to avoid the explicit construction of the complete automaton for input concept before performing the emptiness check of the language accepted by automaton. The algorithm starts by breaking the complex input concept into smaller conjuncts. This is done by analyzing the input concept in a bottom-up manner. The algorithm then takes the first such conjunct and starts by constructing the ALT $A_1$ for the same. Next, the algorithm proceeds to the construction of ULT $A_1'$ from $A_1$. The $A_1'$ is then used to construct the NLT $A_1''$ for the first sub-concept of the input. Then the emptiness test is done on $A_1''$. The outcome of this emptiness test decides the next step of the algorithm. If the emptiness test succeeds, it implies that the language of $A_1''$ is empty and the input sub-concept is not satisfiable. The ISA will terminate immediately implying that the original input concept is unsatisfiable. This result is valid since, if the first conjunct of the input concept is not satisfiable itself then we do not have any chance to satisfy the input concept. Hence, the ISA works in two directions, first it checks the emptiness of a particular automaton $A_1''$ while simultaneously checking the emptiness of the $A_i''$ for $1 \leq i \leq n$, $n$ being the total number of conjuncts of input concept.

On the other hand, if the first conjunct turns out to be satisfying then the second conjunct is also taken into consideration. The conjunction obtained by combining the first and the second conjuncts will now make the next input and the cycle is repeated again. But this time the transitions of the $A_1'$ and $A_1''$ will not be recomputed, instead, the previous ones will be used to construct the automaton for new input. In this way, ISA avoids the re-computation and increases efficiency. Moreover, if the unsatisfiability is detected at an early stage, then the construction of complete automata of the input happens in the worst case only.

Handling TBoxes

One of our aims is to reason over the application domain by taking the terminological knowledge into account. So, we should handle the TBoxes while doing the reasoning over the input concept. This is done by using the following notions,

1. **Unfolding** is a recursive substitution procedure using which, given an unfoldable TBox $T$ and a concept $C$, it is possible to eliminate from $C$ all concept names occurring in $T$. The satisfiability of the resulting concept is independent of the axioms in $T$. The ISA can then be used to find a satisfying interpretation $I$ for the unfolded concept and any such interpretation will also satisfy $T$. However, if $T$ contains cyclic axioms, e.g., $(A \sqsubseteq \exists R.A) \in T$, then trying to unfold $A$ would lead to non-termination.

2. Cyclic and general axioms can be handled by making a single concept from such axioms of TBox, i.e., if each axiom of the form $C \sqsubseteq D$ can be changed to the equivalent $\top \sqsubseteq \neg C \sqcup D$, then we can define $C_T$ as follows:

$$C_T = \prod_{(C_i \sqsubseteq D_j) \in T} (\neg C_i \sqcup D_j)$$

While testing the satisfiability of a concept $C_0$ with respect to $T$, this constraint on possible interpretations can be imposed by testing the satisfiability of $C \cap C_T$. This will ensure that any interpretation satisfying the input concept $C$ will also satisfy every cyclic axiom in the TBox.
We will use both unfolding and $C_T$ in our implementation for efficiency purposes (see Section 6.4.4).

### 6.1.2 XSB Prolog System

XSB is a research-oriented logic programming system which allows the evaluation of queries according to the *Well-Founded Semantics* [Gelder et al., 1991] through full SLG resolution [Sagonas et al., 2007]. It uses a transformation technique called *unification factoring* that can improve program speed and indexing for compiled code.

#### Well-Founded Semantics

Prolog is based on a depth-first search through trees that are built using program clause resolution (SLD). As such, Prolog is susceptible to getting lost in an infinite branch of a search tree, where it may loop infinitely. SLG evaluation, available in XSB, can correctly evaluate many such logic programs. For e.g., any query to the following program:

\[
\texttt{:- table ancestor/2.}
\]

\[
\texttt{ancestor}(X,Y) :- \texttt{ancestor}(X,Z), \texttt{parent}(Z,Y).
\]

\[
\texttt{ancestor}(X,Y) :- \texttt{parent}(X,Y).
\]

will terminate in XSB, since \texttt{ancestor/2} is compiled as a tabled predicate. Prolog systems, however, would go into an infinite loop. The user can declare that SLG resolution is to be used for a predicate by using \texttt{table} declarations, as done here. Alternately, an \texttt{auto_table} compiler directive can be used to direct the system to invoke a simple static analysis to decide what predicates to table\(^1\). This ability to solve recursive queries has proven very useful in a number of areas, including deductive databases, language processing, program analysis, model checking and diagnosis. For efficiency, the SLG is implemented at the abstract machine level so that tabled predicates will be executed with the speed of compiled Prolog.

#### Unification Factoring

For compiled code, XSB offers unification factoring, which extends clause indexing methods found in functional programming into the logic programming framework. Summarily, unification factoring provides not only the complete indexing through non-deterministic indexing automata, but it also factors elementary unification operations [Sagonas et al., 2007].

### 6.1.2.1 Tabling

Consider the Prolog program

\[
\texttt{path}(X,Y) :- \texttt{path}(X,Z), \texttt{edge}(Z,Y).
\]

\[
\texttt{path}(X,Y) :- \texttt{edge}(X,Y).
\]

\(^1\)See Sagonas et al. [2007] for details
together with the query \( ?- \text{path}(1,Y) \). This program has a simple, declarative meaning: there is a path from \( X \) to \( Y \) if there is a path from \( X \) to some node \( Z \) and there is an edge from \( Z \) to \( Y \), or if there is an edge from \( X \) to \( Y \). Prolog, however, enters into an infinite loop when computing an answer to this query. The inability of Prolog to answer such queries comprises one of its major limitations as an implementation of logic.

A number of approaches have been developed to address this problem by reusing partial answers to the query \( \text{path}(1,Y) \) [Tamaki and Sato, 1986; Banchilhon et al., 1986]. The ideas behind these algorithms can be described in the following manner. Calls to tabled predicates, such as \( \text{path}(1,Y) \) in the above example, are stored in a search-able structure together with their proven instances. This collection of tabled subgoals paired with their answers, generally referred to as a table, is consulted whenever a new call, \( C \), to a tabled predicate is issued. If \( C \) is sufficiently similar to a tabled subgoal \( S \), then the set of answers, \( A \), associated with \( S \) may be used to satisfy \( C \). If there is no such \( S \), then \( C \) is entered into the table and is resolved against program clauses as in Prolog, i.e., using SLD resolution. As each answer is derived during this process, it is inserted into the table entry associated with \( C \) if it contains information not already in \( A \).

Predicates can be declared tabled in a variety of ways. A common form is the compiler directive

\[
:- \text{table } p_1/n_1,\ldots,p_k/n_k.
\]

where \( p_i \) is a predicate symbol and \( n_i \) is an integer representing the arity of \( p_i \). For static predicates, this directive is added to a file containing the predicate(s) to be tabled and they are compiled to use tabling by the compiler. For dynamic predicates, the executable directive

\[
?- \text{table } p/n.
\]

causes \( p/n \) to be tabled if no clauses have been asserted for \( p/n \).

### 6.2 Inference Engine Specification

This section presents an overview of the prototype Prolog implementation of the inference engine, based on the proposed incremental satisfaction algorithm. In the following, \( A, B, C, \ldots \), denote well-formed DL concepts expressions. Prolog variables are represented by \( X, Y, \ldots \). Expression on the left of the left-pointing arrow (\( \leftarrow \)) is the goal which we want to satisfy, while expressions on the right of it represent subgoals, that if satisfied, allow one to derive the main goal.

#### 6.2.1 Negation Normal Form

The first step in processing the sub-concept \( C_i \) of the input concept \( C_0 \) is the conversion of \( C_i \) into negation normal form (NNF). In NNF, the negation occurs only in front of concept names. Any DL concept can be converted to its equivalent negation normal form in linear time by the recursive application of the rules given in Table 6.1.

The first argument to \( \text{nnf}/2 \) represents some initial DL concept and the second argument is the resultant form, given that the subgoals on the right-hand side succeed. The rules have to be applied recursively until the negation occurs in front of atomic concepts only.
\[
\text{nnf}(\neg\neg C, C_1) \leftarrow \text{nnf}(C, C_1).
\]
\[
\text{nnf}(\neg\forall R.C, \exists R.C_1) \leftarrow \text{nnf}(\neg C, C_1).
\]
\[
\text{nnf}(\forall R.C, \forall R.C_1) \leftarrow \text{nnf}(C, C_1).
\]
\[
\text{nnf}(\neg\exists R.C, \forall R.C_1) \leftarrow \text{nnf}(\neg C, C_1).
\]
\[
\text{nnf}(\exists R.C, \exists R.C_1) \leftarrow \text{nnf}(C, C_1).
\]
\[
\text{nnf}(\neg(A \cap B), (A_1 \sqcup B_1)) \leftarrow \text{nnf}(\neg A, A_1), \text{nnf}(\neg B, B_1).
\]
\[
\text{nnf}((A \cap B), (A_1 \cap B_1)) \leftarrow \text{nnf}(A, A_1), \text{nnf}(B, B_1).
\]
\[
\text{nnf}(\neg(A \sqcup B), (A_1 \cap B_1)) \leftarrow \text{nnf}(\neg A, A_1), \text{nnf}(\neg B, B_1).
\]
\[
\text{nnf}((A \sqcup B), (A_1 \sqcup B_1)) \leftarrow \text{nnf}(A, A_1), \text{nnf}(B, B_1).
\]
\[
\text{nnf}(\neg C, \neg C) \leftarrow \text{atom}(C).
\]
\[
\text{nnf}(C, C) \leftarrow \text{atom}(C).
\]

Table 6.1: Negation Normal Form Transformation Rules

6.2.2 ALT Automata Construction

After transforming the \( C_i \) into NNF, the next step is to construct the ALT automata \( A_i \) for the \( C_i \). In order to construct \( A_i \) we need extra predicates as detailed below.

Sub-concept computation

The set of sub-concepts of \( C_i \), \( \text{sub}(C_i) \) is the minimal set \( S \) which contains \( C_i \) and has the properties shown in Table 6.2. For a TBox \( T \), \( \text{sub}(C, T) \) is defined as follows:

\[
\text{sub}(C) \cup \bigcup_{C \subseteq D \in T} \text{sub}(\neg C \cup D)
\]

Identifying Concepts and Roles

The next step in the construction of \( A_i \) is to identify and form the sets of concept \( N_c \) and roles \( N_r \) respectively. Also, in-order to identify that the child node is an \( r \)-successor we will add the symbol \( s_r \) in a node label denoting that the node is an \( r \)-successor. For a set \( N_r \) of role names, let \( RC \) be the set such that:

\[
\begin{align*}
\text{if } \neg A \in S, & \text{ then } A \in S \\
\text{if } A \cap B \in S, & \text{ then } A, B \subseteq S \\
\text{if } A \sqcup B \in S, & \text{ then } A, B \subseteq S \\
\text{if } \exists R.C \in S, & \text{ then } C \in S \\
\text{if } \forall R.C \in S, & \text{ then } C \in S
\end{align*}
\]

Table 6.2: Rules for Sub-Concept Computation
\[ RC = \{ s_r \mid r \in N_R \} \]

If \( s_r \) is present in a node label, it will symbolize the fact that the corresponding node is not an \( r \)-successor.

Now, proceeding our construction of \( A_i \), we can construct the set of states and alphabet as defined in Section 5.2 (Definition 5.5). The transitions of \( A_i \) are computed by using the definition of transition relation presented in Table 5.1. The predicates used for the construction of \( A_i \) are described in the Table 6.3.

1. The set of states of \( A_i \) is computed by \texttt{nodeATA0/2} predicate. It takes the input \( C \) and returns the set of states \( Q \).

\[
\text{nodeATA0}(C, Q) \leftarrow \text{subcon}(C, Sc), \text{subCt}(St), \text{union}(Sc, St, CtC), \\
\text{allRoles}(C, Nr), \text{union}(Nr, CtC, CtCR), (Nr \neq \emptyset \rightarrow \text{negRole}(Nr, NNr)), \\
\text{union}(NNr, CtCR, CtCNR), \text{union}([\text{start}, \text{gci}, #], CtCNR, Q).
\]

The predicate \texttt{subcon/2} computes the sets of sub-concepts of the input concept \( C_i \) and a TBox \( T \). The predicate \texttt{allRoles/2} computes the set of the roles present in \( C_i \) or \( T \). The predicate \texttt{union/3} takes two lists as inputs and produces the union of two lists as output. Finally, the states “start”, “gci”, “#” are added to form the set of states \( Q \).

2. Alphabet of \( A_i \) is computed by the predicate \texttt{alphATA0/2}. Upon taking the input \( C \) it returns the set of alphabet symbols, where each such symbol is itself a set comprising the members from \( N_c \) and \( RC \).

\[
\text{alphATA0}(C, A) \leftarrow \text{concept}(C, \_), \text{allRoles}(C, Nr), \text{alphT}(Nt), \\
\text{union}(Nc, Nr, U), \text{union}(U, Nt, At), \text{power}(At, AList), \\
\text{union}(AList, [[#]], A).
\]

The predicate \texttt{alphT/1} returns the alphabet from the TBox component. The predicate \texttt{power/2} computes the powerset of the input.

3. The start state \( q_0 \) is denoted by the predicate \texttt{startATA0/1}.

4. Transitions of \( A_i \) are computed by the \texttt{transitionATA0/5} predicate. For each of the rule shown in Table 5.1 a corresponding body of the transition predicate is defined.

Table 6.3: Predicates for ALT Automata Construction

6.2.3 ULT Automata Construction

For the construction of ULT \( A'_i \), we need to compute the set of restrictions \( R \) relevant to \( q \in Q \) and any \( \sigma \in \Sigma \) (see Definition 5.6). We have to do the additional book-keeping while
constructing $A'_i$ since we want to use the transitions of $A'_i$ while constructing $A'_{i+1}$. The predicates used for the construction of $A'_i$ are mentioned in Table 6.4.

1. The set of states for $A'_i$ will be the same as of $A_i$ so a straight call to the predicate for computing the states of $A_i$ will be made.

$$\text{nodeULT0}(C, Q) \leftarrow \text{nodeATA0}(C, Q).$$

2. The alphabet for $A'_i$ is computed by the predicate $\text{alphULT0}/5$. It takes as input the $Q$ and $\Sigma$ of $A'_{i-1}$ (remember we want to use the transition of previous automata in the construction of incremental automata). A tuple $(\sigma, \eta) \in \Sigma \times R$ is returned for each $q \in Q'$.

$$\text{alphULT0}(C, NListPrev, NList, InAlph, A) \leftarrow \text{tsetof}((\text{Alph}, \text{Res}), \text{alphNodRes}(C, NListPrev, NList, InAlph, \text{Alph}, \text{Res}), A).$$

3. The start node of $A'_i$ is computed by the predicate $\text{startULT0}/1$.

$$\text{startULT0}(0, Q) \leftarrow \text{startATA0}(Q).$$

4. The transitions of $A'_i$ are calculated by the $\text{transitionULT0}/3$ predicate. It takes $q \in Q$ and $\sigma \in \Sigma'$ as input and returns the related transition.

$$\text{transULT0}(Q, (A, R), X) \leftarrow \text{member}((Q, X), R).$$

$$\text{transitionULT0}(Q, A, T) \leftarrow \text{transULT0}(Q, A, T).$$

Table 6.4: Predicates for ULT Automata Construction

### 6.2.4 NLT Automata Construction

The next step is to construct the NLT automata $A''_i$ from $A'_i$. This is the last step in the transformation process, as after constructing $A''_i$ we can perform a language emptiness test to determine whether the input concept $C_i$ is satisfiable or not. The predicates used to construct $A''_i$ are presented in Table 6.5.

### 6.2.5 Emptiness Checking Procedure

Now, we have to check the emptiness of the language accepted by $A''_i$ which will help us to decide about the satisfiability of $C_i$. This is done by the $\text{search}/6$ predicate. It takes the input concept, states and alphabet of $A''_{i-1}$ as input. The alphabet of $A''_i$ and the set of states occurred in the successful proof are returned if the language of $A''_i$ is not empty, otherwise, the predicate fails implying that the input concept is unsatisfiable.

$$\text{search}(C, NListPrev, NList, N, InAlph, Alph, OutAlph, Sol) \leftarrow$$
1. The set of states for $A''_i$ are defined by the predicate $\text{nodeNLT0}/3$. It takes the input and returns the set of $A_i$ states along with its powerset. The set of $A_i$ states are used in the construction of $A'_{i+1}$ while the powerset represents the states of $A''_i$.

$$\text{nodeNLT0}(C, N\text{List}, Qn) \leftarrow \text{nodeAT0}(C, N\text{List}), \text{power}(N\text{List, Qn}).$$

2. The alphabet of $A''_i$ is same as that of $A'_i$ and is computed by the predicate $\text{alphNLT0}/5$.

$$\text{alphNLT0}(C, N\text{ListPrev, NList, InAlph, A}) \leftarrow \text{alphULT0}(C, N\text{ListPrev, NList, InAlph, A}).$$

3. The start node of $A''_i$ is computed by $\text{startNLT0}/1$.

$$\text{startNLT0}([Q1]) \leftarrow \text{startULT0}(0, Q1).$$

4. The transition of $A''_i$ are computed according to the Definition 5.7 by the predicates $\text{transitionNLT0}/3, \text{nExist}/2$ and $\text{sat}/4$. The predicate $\text{transitionNLT0}/3$ takes the input set of states along with the alphabet and returns the corresponding transitions. Predicate $\text{nExist}/2$ computes the total number of existential restrictions in $\text{sub}(C)$ and $\text{sub}(C_T)$ which in turn determines the arity $k$. The $\text{sat}/4$ predicate is used to recursively compute the transition for each member of the input set of states $S$.

$$\text{sat}(S, A, K, Q) \leftarrow \text{transitionULT0}(S, A, R), \text{sat2}(R, 0, K, Q).$$

$$\text{transitionNLT0}(S, A, T) \leftarrow \text{nExist}(S, K), \text{sat}(S, A, K, T).$$

Table 6.5: Predicates for NLT Automata Construction

$$\text{alphNLT0}(C, N\text{ListPrev, NList, InAlph, OutAlph}), \text{member}(\text{Alph, OutAlph}), \text{path}(\text{Alph, N, true, Sol}).$$

6.2.6 ISA Procedure

The input concept $C_0$ has to be processed before beginning the procedure for satisfiability testing. The conjuncts of $C_0$ are processed by $\text{input}/2$ and used for the construction of incremental input, which is then used for satisfiability testing. The predicate $\text{isa}/2$ takes $C_0$ as input and returns the results by calling the $\text{search}/6$ predicate for the incremental input.

$$\text{isa}(C, \text{Sol}) \leftarrow \text{isa1}([C], [], [], \text{Sol}).$$

6.2.7 Optimizations

To speed up the performance of the inference engine, a wide range of optimizations have been looked into. In this section, we discuss these optimizations. They can be broadly divided into two categories as follows:
Program Optimizations

1. The transitions of the automata increases with the increment in the size of the input concept. Thus, unless one of the early conjuncts turn out to be unsatisfiable, the re-computation and the memory requirements for the transitions increase very rapidly. So, utilizing the transitions of $A_{i-1}$ in the construction of $A_i$ is an important optimization.

2. An axiom $(A \sqsubseteq C) \in T$ results in the disjunction $(C \sqcup \neg A)$ being added to the label of every state, leading to non-deterministic expansion and search, contributing to significant overheads in terms of computation and performance. The solution of this problem is to optimize the handling of the TBox by dividing it into two components, an unfoldable part $T_u$ and a general part $T_g$, such that $T_g = T \setminus T_u$ and $T_u$ contains unique, acyclic definition axioms [Baader et al., 2007]. Although, this does not solve the problem entirely, it improves the performance if there are few cyclic axioms (see Figure 6.3).

Prolog-system Optimizations

XSB Prolog uses memoization, an optimization technique used to speed up computer programs by having predicate calls avoid repeating the calculation of results for previously-processed inputs. A memoized function ”remembers” the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it, thus moving the cost of a call with given parameters to the first call made to the function with those parameters. Memoization is a means of lowering a function’s time cost in exchange for space cost; that is, memoized functions become optimized for speed in exchange for a higher use of computer memory space. It can be enabled by using the compiler directive auto_table, which performs a static analysis to determine which predicates may loop under Prolog’s SLD evaluation. These predicates are compiled as tabled predicates, and SLG evaluation is used instead. However, using this compiler directive can be quite expensive in terms of memory, as it may lead to the tabling of the whole run of program. So, another compiler directive suppl_table is used in addition. The intention of this is to direct the system to table for efficiency, rather than termination. The compiler uses tabling to ensure that no predicate will depend on more than three tables. But in order to achieve a good balance between memory and execution speed, we had to decide with the help of profiler whether a predicate should be tabled or not depending on the amount of CPU time spent in the predicate. The results are presented in Section 6.4.1.

6.3 Testing the System

In this section, we present the evaluation methodology of the runtime behavior of prototype implementation of incremental satisfaction algorithm.

6.3.1 Platform

All tests were performed on the Intel 1.66GHz Core Duo machine with 1GB RAM running Linux kernel 2.6.25.1. Though the program will run on various Prolog implementations like SICStus or SWI Prolog with little or no change, XSB\textsuperscript{2} was chosen for the implementation and testing, since it provides various kinds of optimizations (see section 6.2.7).\footnote{http://xsb.sourceforge.net/}
6.3.2 Test Data

The Maryland Information and Network Dynamics Lab Semantic Web Agents Project, Mindswap\(^3\), is a group within a lab of Maryland Institute for Advanced Computer Studies of the University of Maryland. The group focuses on Semantic Web research and has developed the Description Logic reasoner Pellet. As a part of “The Semantic Web” course the students have to develop an \( \mathcal{ALC} \) reasoner in Python \(^4\). To provide for test cases the course maintainers have collected a set of DL queries, together with the necessary TBoxes. These examples come from various sources, the course maintainers, the participants, or from the relevant sources on the Internet. They have the known outcomes and they stress the various parts and potential pitfalls of the tested reasoners. Some of the test queries were taken from Hechenroeder [2006]. A complete list is given in Appendix A. They have been translated from their original Python format to Prolog-styled terms. The test cases contains a predicate \texttt{query/1} which holds the actual query. They also contain the terminological axioms using \texttt{tbox/1} predicate constituting the corresponding TBox. The input query is processed before starting the satisfiability testing. During preprocessing, the TBox is scanned and splitted into cyclic and acyclic axioms. The query is then unfolded w.r.t. acyclic axioms and converted into subgoals by the predicate \texttt{input/2} in a bottom-up manner. The resulting list of subgoals along with the cyclic axioms is then provided to the core inference engine for satisfiability checking.

During the testing, each of the test files is \textit{consulted} into the Prolog interpreter, the query is run, using the TBox (if given) and the outcome of the proof is then used to decide the satisfiability. The query can be run into two modes. First mode just checks the satisfiability and returns “True” if satisfiable, while the second mode extends the first one and also prints the witness found during the proof, if the concept is satisfiable.

6.3.3 Testing Configurations

The inference engine can be run in two different ways. The primary difference in the two ways is the handling of TBoxes. In the acyclic execution mode, the TBox is handled by the means of unfolding, while in cyclic execution mode the TBox is splitted and cyclic axioms are handled as discussed in Section 6.1.1. The acyclic axioms are again handled by unfolding. Depending on the execution modes the test cases are divided into three test configurations mentioned below.

6.3.3.1 Config.1 : Empty TBox

The first configuration handles the queries with empty TBoxes, i.e., there are no application domain axioms and the input concept alone needs to be tested for satisfiability. However, the input concept is divided into a number of sub-concepts before testing. Since there are no axioms at all, the inference engine is run in the acyclic mode.

6.3.3.2 Config.2 : Acyclic TBox

In the second configuration, queries with acyclic TBoxes are dealt by unfolding the input concept w.r.t. TBox. The unfolded concept is then converted into sub-concepts which are then used for satisfiability testing in an incremental fashion. The inference engine is again run in acyclic mode in this configuration.

\(^3\)The MIND Lab, http://www.mindlab.umd.edu

\(^4\)http://www.mindswap.org/2004/cmsc498w/pa3.shtml
6.3.3.3 Config.3: General TBox

The third configuration tests the queries with general TBoxes. Since these TBoxes can potentially contain cyclic queries, the inference engine is run in cyclic mode and the axioms are handled as discussed in 6.3.3.

6.3.4 Reference Systems

6.3.4.1 tableux.pl

`tableux.pl` is a lightweight and optimized tableaux implementation in Prolog by Hechenroeder [2006]. The supported language is $\mathcal{ALC}$. Each of the test cases were put in a separate file starting with $q_n.pl$ or $atq_n.pl$. The performance was measured by calling the `time` command on the top-level predicate `proof/1` e.g.,

```
    time(pl -g test -s q_n.pl -t halt)
```

From the returned values, user time was chosen since it represents the actual computation time taken by the program.

6.3.4.2 RacerPro

RacerPro is a commercial DL reasoner developed by Racer Systems. The version deployed for tests was 1.9. The reference system `tableux.pl` cannot handle the cyclic TBoxes, so for comparing our inference engine in config.3, RacerPro was used. RacerPro is written in Lisp and is highly optimized for speed and performance. The supported DL language is $\mathcal{SHIQ}$. RacerPro supports different ontology formats (such as OWL), interfaces (such as DIG) and built-in functions (such as KRSS). The tests were done by running the RacerPro on input file using the command

```
    time(./RacerPro -f family.racer)
```

6.4 Evaluation and Results

In the following, we present the results of testing the prototype inference engine along with their interpretation.

6.4.1 Memory vs. Performance

XSB system has an important tradeoff between memory and the computation time in form of tabling mechanism. By using tabling, the results of the predicates can be stored in the physical memory so that the results to any successive call are returned instantaneously from the memory, thus avoiding re-computation. But this increases the memory footprint of the program and sometimes even leads to the out-of-memory error. So, deciding whether a predicate should be tabled or not is an important decision. However, XSB system provides a compiler directive `auto_table/0`, which can table the predicates based on its program analysis. But we have found that the decision taken by it is far from optimal, since most of the time it tables most of the program’s predicates resulting in a memory usage explosion. The other way is to manually state which predicates need to be tabled using the compiler directive `table p/n`. An informed decision can be made regarding the tabling of the predicates by running the profiler on the program. The profiling session will reveal the predicates, which
Table 6.6: Satisfiability Results of Queries without TBox

<table>
<thead>
<tr>
<th>Query</th>
<th>Satisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1.P</td>
<td>No</td>
</tr>
<tr>
<td>q2.P</td>
<td>Yes</td>
</tr>
<tr>
<td>q3.P</td>
<td>Yes</td>
</tr>
<tr>
<td>q4.P</td>
<td>No</td>
</tr>
<tr>
<td>q5.P</td>
<td>No</td>
</tr>
<tr>
<td>q6.P</td>
<td>Yes</td>
</tr>
<tr>
<td>q7.P</td>
<td>Yes</td>
</tr>
</tbody>
</table>

if tabled will increase the performance of the program without considerably increasing the memory used by the program. E.g., the query $atq_{9}$ takes about 104 seconds of computation time while using about 70 percent of the memory. If the predicate $transition_{NLTO}/1$ is not tabled then the same query make take around 118 seconds of CPU time, but this time the memory utilization decreases to around 50 percent. This clearly indicates that the choice of tabled predicates affects the performance of the system. Based on several profiling sessions, seven predicates $transition_{ATAO}/5$, $alphNodRes/7$, $alphNodRes1/6$, $alphNodRes1/4$, $alphULTO/6$, $transition_{NLTO}/3$, $search/6$ are tabled in the current implementation.

6.4.2 Config. 1 Results

Figure 6.1 shows the performance of our automata-based inference engine (henceforth ISA engine) compared to that of tableux.pl. The X-axis simply represents the query number shown in table 6.6. The Y-axis represents the CPU time taken for evaluation in milliseconds (ms).

For queries numbered q1 to q6, both ISA engine and tableux.pl show relatively equivalent performance. This behavior can be explained by the fact that the tested queries are relatively simple, so the corresponding automata is also small in size. However, in q7 there is a considerable increase in the time taken by ISA engine. The reason is that the query q7 is satisfiable and the ISA engine proceeds in an incremental fashion and checks all the possible conjuncts for the satisfiability. This case shows a limitation of the incremental approach followed by the ISA engine. In such cases the ISA engine is bound to have inferior performance than the tableau based approach.

6.4.3 Config. 2 Results

The performance results for the queries with acyclic TBox are shown in figure 6.2 (see Appendix A.2 for queries). The overall performance of ISA engine is compared to that of tableaux.pl. The X-axis represents the query number shown in table 6.7. The Y-axis represents the log_{10} value of the CPU time(ms) spent in the execution of the query.

The performance results of the ISA engine have a considerable variance as compared to that of the tableaux.pl whose performance is almost constant. This observation highlights the strengths and weaknesses of the incremental satisfaction strategy. E.g., consider the queries $atq_{18}.P$ and $atq_{20}.P$, both the queries are unsatisfiable. In these, the ISA engine performs better than the tableaux.pl since it finds out the contradiction early thus checking only 2
<table>
<thead>
<tr>
<th>Query</th>
<th>Satisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>atq1.P</td>
<td>No</td>
</tr>
<tr>
<td>atq2.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq3.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq4.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq5.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq6.P</td>
<td>No</td>
</tr>
<tr>
<td>atq7.P</td>
<td>No</td>
</tr>
<tr>
<td>atq8.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq9.P</td>
<td>No</td>
</tr>
<tr>
<td>atq10.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq11.P</td>
<td>No</td>
</tr>
<tr>
<td>atq12.P</td>
<td>No</td>
</tr>
<tr>
<td>atq13.P</td>
<td>No</td>
</tr>
<tr>
<td>atq14.P</td>
<td>No</td>
</tr>
<tr>
<td>atq15.P</td>
<td>No</td>
</tr>
<tr>
<td>atq16.P</td>
<td>No</td>
</tr>
<tr>
<td>atq17.P</td>
<td>No</td>
</tr>
<tr>
<td>atq18.P</td>
<td>No</td>
</tr>
<tr>
<td>atq19.P</td>
<td>No</td>
</tr>
<tr>
<td>atq20.P</td>
<td>No</td>
</tr>
<tr>
<td>atq21.P</td>
<td>No</td>
</tr>
<tr>
<td>atq22.P</td>
<td>Yes</td>
</tr>
<tr>
<td>atq23.P</td>
<td>No</td>
</tr>
<tr>
<td>atq24.P</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.7: Satisfiability Results of Queries with Acyclic TBox
out of the 8 subgoals in case of atq18.P. This behavior exactly shows the strength of the ISA algorithm on this type of queries. However, atq11.P shows an important pitfall for the ISA algorithm and in general, the automata-based techniques. In this case, the ISA engine is quite slow as compared to the tableux.pl since it reaches the contradiction at the very end, thus constructing the huge automata and checking it for emptiness. But this might be helpful in debugging the TBox since ISA engine starting in a bottom-up fashion shows exactly which sub-concepts are satisfiable and outputs the first unsatisfiable sub-concept. The TBox developer can then check that axiom only without caring about the previous ones. From the implementation point of view, one of the reason for relatively slow performance of the ISA engine is the use of member/2 predicate in its generative form for computing the transitions of intermediate ULT automata transition. Due to use of generating capabilities of member/2, considerable amount of backtracking takes place for finding all the possible alternatives. The other reason is the use of setof/3 predicate to find all possible answers
and collect them in the form of a list. This predicate internally calls the \texttt{sort/2} predicate on the resulting list, which increases the computational overhead significantly.

### 6.4.4 Config. 3 Results

The third configuration involves the handling of cyclic axioms in the input TBox. Since, we are using descriptive semantics, the cyclic axioms are handled by ensuring that \( \neg C \sqcup D \) holds for each axiom \( C \sqsubseteq D \) in the TBox with the input concept tested for satisfiability. So, we are dealing with the TBox axioms in the transition condition of the automaton by adding a special concept \( C_T \) representing TBox axioms to the label of each node of the resulting automata. Figure 6.3 shows the results of the test queries. The X-axis again represents the number of the test case while the Y-axis represents the natural log value of CPU time(ms). But this approach leads to non-deterministic expansion and search, the main cause of empirical intractability. Due to the huge expansion in search space we are not able to get the answers for the queries ctq_4.P and ctq_11.P, before the system runs out of physical memory.

In order to contain this kind of search space explosion we have splitted the TBox into two components, an acyclic part \( T_{acy} \) and a cyclic part \( T_{cy} \) such that \( T_{cy} = T \setminus T_{acy} \). Now the \( T_{acy} \) is handled by the unfolding, while the \( T_{cy} \) is handled by adding \( C_T \) to each node of the automata. Figure 6.3 shows that this led to a considerable improvement in the performance since the number of axioms handled directly in the automaton transitions have now reduced to just one. However, if any TBox contains significant number of cyclic axioms even after splitting it, then the cyclic axioms must be handled in an another way instead of introducing new disjunctions.

<table>
<thead>
<tr>
<th>Query</th>
<th>Satisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctq_1.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_2.P</td>
<td>Yes</td>
</tr>
<tr>
<td>ctq_3.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_4.P</td>
<td>Yes</td>
</tr>
<tr>
<td>ctq_5.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_6.P</td>
<td>Yes</td>
</tr>
<tr>
<td>ctq_7.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_8.P</td>
<td>Yes</td>
</tr>
<tr>
<td>ctq_9.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_10.P</td>
<td>No</td>
</tr>
<tr>
<td>ctq_11.P</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.8: Satisfiability Results of Queries with Cyclic TBox

Test case ctq_9.P highlights an important notion that should be kept in mind while checking the satisfiability over a splitted TBox. If the concepts in query refer to both cyclic as well as acyclic parts, then the axioms that belong to \( T_{acy} \) and are referred in the satisfiability query, should be handled in the same manner as the cyclic axioms, in order to get the correct answer.
The performance plot also shows the ISA engine against the commercial system Racer-Pro. However, the ISA engine was tested against a commercial reasoner for the sole motive of verifying the correctness of the answers of test queries and identifying the pitfall queries for future optimization purposes. For queries ctq_{1-10}.P, the ISA engine results are comparable to that of RacerPro. But for the query ctq_{11}.P, the ISA engine takes considerably longer time to answer whereas RacerPro has almost constant performance. This query shows that at current stage, scaling up in terms of complexity of queries is a problem for the ISA engine. Also, currently the ISA engine doesn’t perform any preprocessing on queries or TBox to minimize the number of inferences required or for simplification of input queries, since our aim was to demonstrate the potential of incremental satisfaction algorithm rather coming up with a production implementation. Nonetheless, these observations point towards the future course of action.

Figure 6.3: Config. 3 Performance Results
Conclusion

The goal of this work was to realize the incremental satisfaction algorithm for the DL $\text{ALC}$ satisfiability problem by giving the definitions required for the construction of intermediate automata and developing an inference engine based on the proposed algorithm in order to highlight its strengths and shortcomings. The ideas presented for $\mu$-calculus in Kupferman and Vardi [2005] and Unel and Toman [2007a] served as the starting point for this work. Taking them into account we have proposed

- definitions for the alternating looping tree automaton (ALT), the universal looping tree automaton (ULT), and the nondeterministic tree automaton (NLT),
- the translations $\text{ALT} \rightarrow \text{ULT} \rightarrow \text{NLT},$
- the incremental satisfaction algorithm for DL $\text{ALC},$
- a prototype implementation of ISA in Prolog.

These notions have been put to work in the Prolog implementation of the ISA engine. Comparison with the traditional tableau based inference engine revealed a comparable performance in the unsatisfiable conjunctive queries that can be splitted into subgoals and the contradiction is detected at an early stage. Moreover, the reference system, tableaux.pl is currently unable to handle the TBoxes containing cycles since the sophisticated optimizations like blocking will be required in tableau-based approach for handling the cycle. But the ISA engine has the capability of handling general TBoxes due to the fact that automata-based algorithms do not require any special techniques for termination. However, the tableaux.pl performs better on arbitrary queries, since sometimes it is not possible to break complex concepts into simpler conjuncts which leads to the construction of a larger automaton and resulting in slower performance. The satisfiable queries also took comparatively longer execution times since the ISA engine has to satisfy all the sub-concepts first before testing the satisfiability of input concept. But this overhead can be minimized to a greater extent by utilizing the previously computed automata in a more efficient manner. The comparison of ISA engine with the commercial DL reasoner RacerPro revealed that ISA engine has reasonable performance in case of simple queries. But the response time increases with the complexity of queries, i.e., currently ISA engine doesn’t scale well with the increasing complexity of queries.
The prototype implementation of the incremental automata-based satisfaction algorithm has shown promising results for the unsatisfiable concepts, that can be reduced to the simpler and smaller sub-goals detecting an early contradiction. However, the performance over arbitrary queries is not that satisfying as compared to tableau-based approach. For improving the performance over the arbitrary queries new optimizations are required, e.g., limiting the non-deterministic choices of transitions from a given state, efficient memory representation of the computed automata transitions etc. The BDDs can be used for the efficient internal representation of the automata. Currently, the approach adopted for handling of cyclic and general axioms leads to the empirical intractability due to the non-deterministic expansion. A new optimization technique is required to keep the search space small for a consistent performance over a large TBox.

7.1 Outlook

The presented work leaves room for a lot of extensions along various dimensions.

7.1.1 Optimizations

The current implementation needs to be fine-tuned for better performance. Prolog platform was chosen for the prototype implementation to reduce the implementation time, by taking the advantage of its existing backtracking mechanism. However, an imperative language like Java or perhaps a functional language like Scala will be more suitable from the performance point of view. Also, the new optimizations have to be devised specific to the incremental algorithm. An interesting starting point could be the existing tableau algorithm optimizations.

7.1.2 DL Language Extensions

The presented implementation only covers the most basic DL $\mathcal{ALC}$. A natural and desirable way to extend it would to support a more powerful language, for e.g., $\mathcal{ALCN}$ i.e., $\mathcal{ALC}$ with cardinality restrictions in unqualified form or $\mathcal{ALCQ}$, i.e., $\mathcal{ALC}$ extended with qualified number restrictions. Also, extending $\mathcal{ALC}$ with union and transitive reflexive closure role-forming operator while having the same worst case complexity for concept satisfiability, will provide another way to handle the cyclic axioms by using internalization technique.

7.1.3 Ontological Representations

The system could be enriched with more parsers for various ontological representations and formats. This would allow a greater variety of ontologies to be read in and reasoned with.

7.1.4 Proof Explanation

The current implementation provides an option to output the basic proof constructed in the search of finding a witness. It also outputs the last satisfiable concept in case of an unsatisfiable input concept. But these can be further extended to expose the crucial steps of the transformations which can be helpful to the user.
Below are the contents of the test files containing the respective prolog terms. The tests files are broadly divided according to the three test configurations. For each section the file name precedes its contents. In each section, first the query is presented then the list of subconcepts generated for the incremental satisfiability checking and finally the incremental output obtained from the inference engine. The incremental output will not be included for the larger queries in order to conserve space. Also, the subconcepts list can provide the tested subconcepts and the final results are mentioned in section 6.4.

### A.1 Queries without TBox

**q1.P**

query(exist(r,b), and(exist(r,b), forall(r, ∼b))).

Subconcepts List: [forall(r, ∼b), and(exist(r,b), forall(r, ∼b))]

Concept Tested: forall(r, ∼b) Concept Satisfiable: forall(r, ∼b)

Concept Tested: and(exist(r,b), forall(r, ∼b))

**q2.P**

query(and(exist(r,b), forall(r, or(c, ∼b))))).

Subconcepts List: [forall(r,or(c,∼b)), and(exist(r,b),forall(r,or(c,∼b)))]

Concept Tested: forall(r,or(c,∼b))

Concept Satisfiable: forall(r,or(c,∼b))

Concept Tested: and(exist(r,b),forall(r,or(c,∼b)))

Concept Satisfiable: and(exist(r,b),forall(r,or(c,∼b)))

**q3.P**

query(or(and(or(a,b), or(c,d)), and(or(a,c), or(b,d))))).

Subconcepts List: [or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))]
APPENDIX A. TEST QUERIES

Concept Tested: or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))
Concept Satisfiable: or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))

q4.P
query(and(and(or(a,b), or(c,d)), and(or(a,c), or(b,d))))).
Subconcepts List: [or(b,d),and(or(a,c),or(b,d)),and(and(or(a,b),or(c,d)),
and(or(a,c),or(b,d)))]
Concept Tested: or(b,d)
Concept Satisfiable: or(b,d)
Concept Tested: and(or(a,c),or(b,d))
Concept Satisfiable: and(or(a,c),or(b,d))
Concept Tested: and(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))
Concept Satisfiable: and(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))

q5.P
query(and(forall(r, and(a,b)), and(exist(r,a), exist(r, ∼a))))).
Subconcepts List: [exist(r,∼a),and(exist(r,a),exist(r,∼a)),
and(forall(r,and(a,b)),and(exist(r,a),exist(r,∼a)))]
Concept Tested: exist(r,∼a) Concept Satisfiable: exist(r,∼a)
Concept Tested: and(exist(r,a),exist(r,∼a))

q6.P
query(or(and(or(a,b), or(c, d)), and(or(a, c), or(b, d))))).
Subconcepts List: [or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))]
Concept Tested: or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))
Concept Satisfiable: or(and(or(a,b),or(c,d)),and(or(a,c),or(b,d)))

q7.P
query(and(a, and(b, and(forall(p,c), and(forall(p, ~c), exist(r, d)))))).
Subconcepts List: [exist(r,d),and(forall(p,∼c),exist(r,d)),and(forall(p,c),
and(forall(p,∼c),exist(r,d))),and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d)))),
and(a,and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d)))))]
Concept Tested: exist(r,d) Concept Satisfiable: exist(r,d)
Concept Tested: and(forall(p,∼c),exist(r,d))
Concept Satisfiable: and(forall(p,∼c),exist(r,d))
Concept Tested: and(forall(p,c),and(forall(p,∼c),exist(r,d)))
Concept Satisfiable: and(forall(p,c),and(forall(p,∼c),exist(r,d)))
Concept Tested: and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d))))
Concept Satisfiable: and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d))))
Concept Tested: and(a,and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d)))))
Concept Satisfiable: and(a,and(b,and(forall(p,c),and(forall(p,∼c),exist(r,d))))
A.2 Queries with Acyclic TBox

atq1.P

tbox(equiv(a, and(h, and(i, ∼d)))).
tbox(equiv(j, ∼k)).
tbox(equiv(b, ∼g)).
tbox(equiv(d, forall(q, j))).
tbox(equiv(g, ∼e)).

query(and(exist(p, a), and(exist(p, b), and(and(c, d), ∼exist(p, ∼and(∼e, f)))))).
Subconcepts List: [forall(p,and(∼e, f)),and(and(c,forall(q,∼k)),forall(p,and(∼e, f))),
and(exist(p,e),and(and(h,exist(q,k)),forall(p,and(∼e, f)))),
and(exist(p, and(h,exist(i,q,k))),and(exist(p,e),and(and(c,forall(q,∼k)),
forall(p,and(∼e, f)))))]

Concept Tested:forall(p,and(∼e, f)) Concept Satisfiable: forall(p,and(∼e, f))
Concept Tested:and(and(c,forall(q,∼k)),forall(p,and(∼e, f)))
Concept Satisfiable:and(and(c,forall(q,∼k)),forall(p,and(∼e, f)))
Concept Tested:and(exist(p,e),and(and(c,forall(q,∼k)),forall(p,and(∼e, f))))

atq2.P

tbox(equiv(wierdopizza, and(pizza, exist(hastopping, alien)))).
tbox(equiv(veggiepizza, and(pizza, forall(hastopping, ∼meat)))).
tbox(equiv(meatpizza, and(pizza, forall(hastopping, ∼veggie)))).
tbox(equiv(veggie, or(mushroom, olive))).
tbox(equiv(alien, anchovy)).
tbox(equiv(meat, or(pepperoni, sausage))).

query(and(veggiepizza, meatpizza)).
Subconcepts List: [forall(hastopping,and(∼mushroom,∼olive)),and(pizza,forall(hastopping, 
and(∼mushroom,∼olive))),and((pizza,forall(hastopping,(∼pepperoni,∼sausage))),
and((pizza,forall(hastopping, (∼mushroom,∼olive))))]

Concept Tested:forall(hastopping,and(∼mushroom,∼olive))
Concept Satisfiable:forall(hastopping,and(∼mushroom,∼olive))
Concept Tested:and(pizza,forall(hastopping,and(∼mushroom,∼olive)))
Concept Satisfiable:and(pizza,forall(hastopping,and(∼mushroom,∼olive)))
Concept Tested:and(and(pizza,forall(hastopping,and(∼pepperoni,∼sausage))),
and((pizza,forall(hastopping, (∼mushroom,∼olive)))))
Concept Satisfiable: and(and(pizza,forall(hastopping,and(∼pepperoni,∼sausage))),
and((pizza,forall(hastopping, (∼mushroom,∼olive)))))

atq3.P

tbox(equiv(wierdopizza, and(pizza, exist(hastopping, alien)))).
tbox(equiv(veggiepizza, and(pizza, forall(hastopping, ∼meat)))).
tbox(equiv(meatpizza, and(pizza, forall(hastopping, ∼veggie)))).
tbox(equiv(veggie, or(mushroom, olive))).
APPENDIX A. TEST QUERIES

tbox(equiv(alien, anchovy)).  
tbox(equiv(meat, or(pepperoni, sausage))).

query(or(~wierdopizza, veggiepizza)).  
Subconcepts List: [or(or(~pizza,forall(hastopping,~anchovy)),and(pizza,forall(hastopping,and(~pepperoni,~sausage)))]

Concept Tested: or(or(~pizza,forall(hastopping,~anchovy)),and(pizza,forall(hastopping, and(~pepperoni,~sausage))))  
Concept Satisfiable: or(or(~pizza,forall(hastopping,~anchovy)),and(pizza,forall(hastopping, and(~pepperoni,~sausage))))

atq4.P
This query can be used to check disjointness of two concepts, for this we can negate the final proof i.e., disjoint(brother, sister) ≡ not(isa(and(brother, sister), Sol)).  
tbox(equiv(man, and(person, male))).  
tbox(equiv(woman, and(person,~man))).

tbox(equiv(mother, and(woman, exist(hasChild, person)))).  
tbox(equiv(father, and(man, exist(hasChild, person)))).  
tbox(equiv(parent, exist(hasChild, person))).

tbox(equiv(grandfather, and(man, exist(hasChild, father)))).

tbox(equiv(brother, and(man, exist(hasSibling, person)))).

tbox(equiv(sister, and(person, and(~brother, exist(hasSibling, person)))).  
tbox(equiv(luckyBrother, and(man, forall(hasSibling, sister)))).

disjoint(brother, sister).
Subconcepts List: [exist(hasSibling, person),and(or(or(~person,~male),forall(hasSibling,~person))),exist(hasSibling, person)],and(person,and(or(or(~person,~male),forall(hasSibling,~person))),exist(hasSibling, person)),and(person, and(or(or(~person,~male),forall(hasSibling,~person))),exist(hasSibling, person))]

Concept Tested: exist(hasSibling, person) Concept Satisfiable:exist(hasSibling, person)  
Concept Tested: and(or(or(~person,~male),forall(hasSibling,~person)),exist(hasSibling, person))  
Concept Satisfiable:and(or(or(~person,~male),forall(hasSibling,~person)),exist(hasSibling, person))  
Concept Tested:and(person, and(or(or(~person,~male),forall(hasSibling,~person))),exist(hasSibling, person))  
True (Since the last concept is unsatisfiable, its negation holds).

atq5.P

disjoint(man, woman)\(^1\)  
Subconcepts List: [or(~person,~male),and(person,or(~person,~male)),and(person,~male),and(person,or(~person,~male))]

Concept Tested:or(~person,~male) Concept Satisfiable:or(~person,~male) 
Concept Tested:and(person,or(~person,~male))  
Concept Satisfiable:and(person,or(~person,~male))  
Concept Tested:and(and(person,~male),and(person,or(~person,~male)))  
True

\(^1\)The queries from atq5 to atq10 will use the same TBox as that of atq4.
APPENDIX A. TEST QUERIES

atq6.P

query(and(man, and(mother, and(man, person)))).

Subconcepts List: [person, and(and(person, male), person), and(and(person, or(~person, ~male)), exist(hasChild, person)), and(and(person, male), person), and(and(person, or(~person, ~male)), exist(hasChild, person)), and(and(person, male), person)]

Concept Tested: person
Concept Satisfiable: person
Concept Tested: and(and(person, male), person)
Concept Satisfiable: and(and(person, male), person)

This query can be used to check subsumption of two concepts, for this we can negate the final proof i.e., subsume(A, D) ≡ not(isa(and(A, ~D), Sol)) and then the final answer should be false.

subsume(luckyBrother, brother).

Subconcepts List: [or(or(~person, ~male), forall(hasSibling, ~person)), and(and(person, male), forall(hasSibling, and(person, or(~person, ~male), forall(hasSibling, ~person)), exist(hasSibling, person)))], or(or(~person, ~male), forall(hasSibling, ~person))]

Concept Tested: or(or(~person, ~male), forall(hasSibling, ~person))
Concept Satisfiable: or(or(~person, ~male), forall(hasSibling, ~person))
Concept Tested: and(and(and(person, male), forall(hasSibling, and(person, or(~person, ~male), forall(hasSibling, ~person)), exist(hasSibling, person)))), or(or(~person, ~male), forall(hasSibling, ~person))]

Concept Satisfiable: and(and(and(person, male), forall(hasSibling, and(person, or(~person, ~male), forall(hasSibling, ~person)), exist(hasSibling, person)))), or(or(~person, ~male), forall(hasSibling, ~person))]

False (The input is satisfiable so its negation should be unsatisfiable)

atq8.P

subsume(grandfather, father).

Subconcepts List: [or(or(~person, ~male), forall(hasChild, ~person)), and(and(person, male), exist(hasChild, and(and(person, male), exist(hasChild, person)))), or(or(~person, ~male), forall(hasChild, ~person))]

Concept Tested: or(or(~person, ~male), forall(hasChild, ~person))
Concept Satisfiable: or(or(~person, ~male), forall(hasChild, ~person))
Concept Tested: and(and(and(person, male), exist(hasChild, and(and(person, male), exist(hasChild, person)))), or(or(~person, ~male), forall(hasChild, ~person))]

Concept Satisfiable: and(and(and(person, male), exist(hasChild, and(and(person, male), exist(hasChild, person)))), or(or(~person, ~male), forall(hasChild, ~person))]

True

atq9.P

disjoint(grandfather, father).

Subconcepts List: [exist(hasChild, person), and(and(person, male), exist(hasChild, person)), and(and(and(person, male), exist(hasChild, and(and(person, male), exist(hasChild, person)))), and(and(person, male), exist(hasChild, person))]

Concept Tested: disjoint(grandfather, father)
APPENDIX A. TEST QUERIES

Concept Tested: exist(hasChild,person) Concept Satisfiable: exist(hasChild,person)
Concept Tested: and(and(person,male),exist(hasChild,person))
Concept Satisfiable: and(and(person,male),exist(hasChild,person))
Concept Tested: and(and(and(person,male),exist(hasChild,and(and(person,male), exist(hasChild,person)))), and(and(person,male),exist(hasChild,person)))
Concept Satisfiable: and(and(and(person,male),exist(hasChild,and(and(person,male), exist(hasChild,person)))), and(and(person,male),exist(hasChild,person)))
False

atq10.P
disjoint(grandfather, sister).
Subconcepts List: [exist(hasSibling,person), and(or(or(~person,~male),
forall(hasSibling,~person)), exist(hasSibling,person)),
and(person, and(or(or(~person,~male),forall(hasSibling,~person)),
exist(hasChild, and(and(person,male), exist(hasChild,person)))),
and(person, and(or(or(~person,~male),forall(hasSibling,~person)),
exist(hasSibling,person)))]
Concept Tested: exist(hasSibling,person) Concept Satisfiable: exist(hasSibling,person)
Concept Tested: and(or(or(~person,~male),forall(hasSibling,~person)), exist(hasSibling,person))
Concept Satisfiable: and(or(or(~person,~male),forall(hasSibling,~person)), exist(hasSibling,person))
Concept Tested: and(person, and(or(or(~person,~male),forall(hasSibling,~person)),
exist(hasSibling,person)))
True

atq11.P
tbox(equiv(flyingdragon, and(dragon, exist(transportMode, flying)))).
tbox(equiv(walkingdragon, and(dragon, exist(transportMode, walking)))).
tbox(equiv(slitheringdragon, and(dragon, forall(transportMode, ~or(flying, walking))))).
tbox(equiv(westerndragon, and(flyingdragon, and(forall(elemental, or(earth, water)),
exist(disposition, foe))))).
tbox(equiv(orientaldragon, and(walkingdragon, and(exist(elemental, water), forall(disposition,
friend))))).
tbox(equiv(drake, and(walkingdragon, and(exist(elemental, or(water, fire)), forall(disposition,
foe))))).
tbox(equiv(icedrake, and(drake, and(forall(elemental, water), exist(disposition, foe))))).
tbox(equiv(firedrake, and(drake, and(forall(elemental, fire), exist(disposition, foe))))).
tbox(equiv(wyrm, and(slitheringdragon, exist(elemental, water)))).
tbox(equiv(hydra, and(or(slitheringdragon, flyingdragon), exist(disposition, foe)))).
tbox(equiv(dragonet, and(forall(disposition, foe), and(or(walkingdragon, flyingdragon),
forall(elemental, ~or(earth, water))))).

query(and(dragonet, ~and(forall(disposition, foe), and(or(walkingdragon,
hydra), forall(elemental, ~or(earth, water))))).)
Subconcepts List: [or(exist(disposition,~foe),or(and(or(~dragon, forall(transportMode,~walking))),
or(~dragon,forall(transportMode,~flying))),exist(elemental,or(earth,water))],
and(and(forall(disposition,foe),and(or(and(dragon,exist(transportMode,walking))),
dragon, exist((transportMode,flying))),forall(elemental, and(~earth,~water))),or(exist(disposition,~foe),
or(and(or(~dragon,forall(transportMode,~walking))),or(~dragon,forall(transportMode,~flying))),
APPENDIX A. TEST QUERIES

```
exist(elemental, or(earth, water)))
```

### atq12.P

```
tbox(equiv(beer, and(drink, and(exist(has-ingr, water), and(exist(has-ingr, hops), and(exist(has-ingr, malt), forall(has-ingr, or(water, or(hops, malt)))))))).
tbox(equiv(grapes, and(∼hops, and(∼malt, ∼water)))).
tbox(equiv(wine, and(drink, exist(has-ingr, grapes)))).
```

query(and(wine, beer)).

Subconcepts List: [forall(has-ingr, or(water, or(hops, malt))), and(exist(has-ingr, malt), forall(has-ingr, or(water, or(hops, malt)))), and(exist(has-ingr, hops), and(exist(has-ingr, malt), forall(has-ingr, or(water, or(hops, malt))))), and(drink, and(exist(has-ingr, water), and(exist(has-ingr, hops), and(exist(has-ingr, malt), forall(has-ingr, or(water, or(hops, malt)))))))]

### atq13.P

```
tbox(equiv(a, ∼b)).
```

query(and(exist(r, b), forall(r, or(a, ∼b)))).

Subconcepts List: [forall(r, or(∼b, ∼b)), and(exist(r, b), forall(r, or(∼b, ∼b)))]

### atq14.P

```
tbox(equiv(werewolf, and(animal, and(exist(has-power, magical), forall(speaks, language))))).
tbox(equiv(acramantula, and(beast, and(or(male, female), exist(has-power, magical))))).
tbox(equiv(wizard, and(male, and(human, exist(has-power, magical))))).
tbox(equiv(centaur, and(animal, and(∼human, and(or(male, female), and(exist(has-power, magical), forall(speaks, language))))))).
tbox(equiv(vampire, and(beast, and(or(male, female), forall(has-power, magical))))).
tbox(equiv(beast, and(animal, ∼human))).
tbox(equiv(muggle, and(human, and(or(male, female), forall(has-power, ∼magical))))).
tbox(equiv(witch, and(female, and(human, exist(has-power, magical))))).
tbox(equiv(human, and(animal, exist(speaks, language))))).
tbox(equiv(male, and(animal, ∼female))).
tbox(equiv(merpeople, and(animal, and(∼human, and(forall(has-power, magical), forall(speaks, language)))))).
```

query(and(werewolf, human)).

Subconcepts List: [exist(speaks, language), and(animal, exist(speaks, language)), and(animal, exist(has-power, magical), forall(speaks, language)), and(animal, exist(speaks, language))]
APPENDIX A. TEST QUERIES

atq15.P

tbox(equiv(salto, and(cartwheel, and( ~exist(hasposition, handsonfloor), exist(hasposition, twist))))).

tbox(equiv(fronttuck, and(∼cartwheel, forall(hasposition, tuck))))).

tbox(equiv(roundoff, and(cartwheel, and(handstand, forall(hasposition, pike))))).

tbox(equiv(backwalkover, and(exist(hasposition, bridge), exist(hasposition, handstand))))).

tbox(equiv(forwardroll, or(exist(hasposition, pike), or(exist(hasposition, straddle), exist(hasposition, tuck))))).

tbox(equiv(backhandspring, exist(hasposition, bridge))).

tbox(equiv(handstand, ∼forall(hasposition, pike))).

query(and(roundoff, ∼backhandspring)).
Subconcepts List: [forall(hasposition, ∼bridge), and(and(cartwheel, and(exist(hasposition, ∼pike), forall(hasposition, pike))), forall(hasposition, ∼bridge)])

atq16.P

tbox(equiv(werewolf, and(beast, and(exist(haspower, magical), forall(speaks, language))))).

tbox(equiv(acramantula, and(beast, and(or(male, female), exist(haspower, magical))))).

tbox(equiv(wizard, and(male, and(human, exist(haspower, magical))))).

tbox(equiv(centaur, and(animal, and(∼human, and(or(male, female), and(exist(haspower, magical), forall(speaks, language)))))))).

tbox(equiv(vampire, and(beast, and(or(male, female), and(exist(haspower, magical), forall(speaks, language))))))).

tbox(equiv(beast, and(animal, ∼human))).

tbox(equiv(muggle, and(human, and(or(male, female), forall(haspower, ∼magical))))).

tbox(equiv(witch, and(female, and(human, exist(haspower, magical))))).

tbox(equiv(human, and(animal, exist(speaks, language))))).

tbox(equiv(male, and(animal, ∼female))).

tbox(equiv(merpeople, and(animal, and(∼human, and(forall(haspower, magical), forall(speaks, language))))))).

query(and(werewolf, human)).
Subconcepts List: [exist(speaks, language), and(animal, exist(speaks, language)), and(and(animal, or(∼animal, forall(speaks, ∼language))), and(exist(haspower, magical), forall(speaks, language))), and(animal, exist(speaks, language))]]

atq17.P

tbox(equiv(academicfreenulllicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(∼mustdistributemods, (∼licenseisviral, ∼cannotredistribute))))))).

tbox(equiv(commonpubliclicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, (∼licenseisviral, ∼cannotredistribute))))))).

tbox(equiv(publicdomainlicense, and(freesoftwarelicense, exist(hasrestriction, (∼mustkeepdisclaimer, or(mustdistributemods, ∼licenseisviral, ∼cannotredistribute))))))).

tbox(equiv(lessergnupubliclicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, (∼licenseisviral, ∼cannotredistribute))))))).

tbox(equiv(commercialsoftwarelicense, exist(hasrestriction, cannotredistribute))).

tbox(equiv(opensoftwarelicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, ∼licenseisviral, ∼cannotredistribute))))))).

tbox(equiv(softwarerelease, or(freesoftwarelicense, commercialsoftwarelicense))).

tbox(equiv(opensoftwarelicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, ∼licenseisviral, ∼cannotredistribute))))))).

query(and(opensoftwarelicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, ∼licenseisviral, ∼cannotredistribute))))))).
APPENDIX A. TEST QUERIES

and(mustdistributemods, and(licenseisviral, ¬cannotredistribute))).
tbox(equiv(gnugeneralpubliclicense, and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, and(licenseisviral, ¬cannotredistribute)))))),
query(and(lessergnupubliclicense, forall(hasrestriction, cannotredistribute))).
Subconcepts List: [forall(hasrestriction, cannotredistribute), and(and(freesoftwarelicense, exist(hasrestriction, and(mustkeepdisclaimer, and(mustdistributemods, and(licenseisviral, ¬cannotredistribute))))), forall(hasrestriction, cannotredistribute)]

atq18.P
This query particularly highlights the working of incremental technique. The subconcept list has large number of elements but actually only few are required for deciding satisfiability of the query.

tbox(equiv(parentdog, or(papadog, mamadog))).
tbox(equiv(husbanddog, and(maledog, exist(haswife, femaledog))).
tbox(equiv(maledog, and(dog, ¬female))).
tbox(equiv(papadog, and(maledog, exist(haschild, dog))).
tbox(equiv(femaledog, and(dog, female))).
tbox(equiv(wifedog, and(femaledog, exist(has丈夫, maledog))).
tbox(equiv(puppy, and(or(maledog, femaledog), and(exist(hasmother, mamadog), and(exist(hasfather, papadog), ¬exist(haschild, dog))))).
tbox(equiv(mamadog, and(femaledog, exist(haschild, dog))).
query(and(puppy, and(mamadog, wifedog))).
Subconcepts List: [exist(has丈夫, and(dog, ¬female)), and(and(dog, female), exist(has丈夫, and(dog, ¬female))), and(and(dog, female), exist(haschild, dog))), and(and(or(dog, ¬female), and(dog, female)), and(exist(hasmother, and(and(dog, female), exist(haschild, dog))), and(exist(hasfather, and(and(dog, ¬female), exist(haschild, dog))), forall(haschild, ¬dog)))), and(and(dog, female), exist(hasfather, and(dog, female)), exist(haschild, dog))), and(and(dog, female), exist(has husbands, and(dog, ¬female)))]
Concept Tested:exist(has丈夫, and(dog, ¬female))
Concept Satisfiable:exist(has丈夫, and(dog, ¬female))
Concept Tested:and(and(dog, female), exist(has丈夫, and(dog, ¬female)))

atq19.P

tbox(equiv(slitheringdragon, and(dragon, forall(transportmode, ¬or(flying, walking))))).
tbox(equiv(walkingdragon, and(dragon, exist(transportmode, walking)))).
tbox(equiv(firedrake, and(drake, and(forall(elemental, fire), exist(disposition, foe))))).
tbox(equiv(icedrake, and(drake, and(forall(elemental, water), exist(disposition, foe))))).
tbox(equiv(orientaldragon, and(walkingdragon, and(exist(elemental, water), forall(disposition, friend))))).
tbox(equiv(drake, and(walkingdragon, and(exist(elemental, or(water, fire)), forall(disposition, foe))))).
tbox(equiv(hydra, and(or(slitheringdragon, flyingdragon), exist(disposition, foe))))).
tbox(equiv(westerndragon, and(flyingdragon, and(forall(elemental, or(earth, water)), exist(disposition, foe))))).
APPENDIX A. TEST QUERIES

62

tbox(equiv(wyrm, and(slitheringdragon, exist(elemental, water))))).
tbox(equiv(flyingdragon, and(dragon, exist(transportmode, flying))))).
tbox(equiv(dragonet, and(forall(disposition, foe), and(or(walkingdragon, flyingdragon), forall(elemental, ~or(earth, water)))))).

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire)))))]

atq20.P
This query highlights the working of incremental technique.

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, or(w,x))))))))))))).
Subconcepts List: [or(w,x), and(~v1, or(w,x)), and(~u, and(~v1, or(w,x))), and(~t, and(~u, and(~v1, or(w,x)))), and(~r, and(~s, and(~t, and(~u, and(~v1, or(w,x)))))), and(~o, and(~p, and(~q, and(~r, and(~s, and(~t, and(~u, and(~v1, or(w,x)))))), and(~m, and(~n, and(~o, and(~p, and(~q, and(~r, and(~s, and(~t, and(~u, and(~v1, or(w,x)))))))))), and(~m, and(~n, and(~o, and(~p, and(~q, and(~r, and(~s, and(~t, and(~u, and(~v1, or(w,x)))))))))))), and(~m, and(~n, and(~o, and(~p, and(~q, and(~r, and(~s, and(~t, and(~u, and(~v1, or(w,x)))))))))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq21.P

query(and(a, or(b, c))).
tbox(equiv(b, or(m, n))).
tbox(equiv(e, or(q, r))).
tbox(equiv(d, or(e, f))).
tbox(equiv(g, or(h, i))).
tbox(equiv(f, or(s, t))).
tbox(equiv(i, or(w, x))).
tbox(equiv(h, or(u, v1))).

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, or(w,x))))))))))))).

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq20.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

atq21.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq20.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq21.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq20.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq21.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq20.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))

atq21.P

query(and(hydra, and(dragonet, exist(elemental, fire)))).
Subconcepts List: [exist(elemental, fire), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), forall(elemental, and(~earth, ~water)))), exist(elemental, fire)), and(and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(disposition, foe)), and(and(forall(disposition, foe), and(or(and(dragon, exist(transportmode, walking)), and(dragon, exist(transportmode, flying))), exist(elemental, fire))))]

Concept Tested: or(w,x) Concept Satisfiable: or(w,x)

Concept Tested: and(~v1, or(w,x)) Concept Satisfiable: and(~v1, or(w,x))

Concept Tested: and(~u, and(~v1, or(w,x)))
APPENDIX A. TEST QUERIES

tbox(equiv(maledog,and(dog, ∼female))).
tbox(equiv(papadog,and(maledog, exist(haschild, dog)))).
tbox(equiv(femaledog,and(dog, female))).
tbox(equiv(wifedog,and((femaledog, exist(haschild, maledog))))).
tbox(equiv(puppy, and(or( maledog, femaledog), and(exist(hasmother, mamadog), and(exist(hasfather, papadog), ∼exist(haschild, dog))))).
tbox(equiv(mamadog,and(femaledog, exist(haschild, dog))))).

query(and(wifedog, husbanddog)).
Subconcepts List: [exist(haswife,and(dog,female)),and(and(dog,∼female),exist(haswife,and(dog,female))),
and(and(and(dog,female),exist(hashusband,and(dog,∼female))),and(and(dog,∼female),exist(haswife,and(dog,female)))))]

Concept Tested: exist(haswife,and(dog,female)) Concept Satisfiable: exist(haswife,and(dog,female))
Concept Tested: and(and(dog,∼female),exist(haswife,and(dog,female)))

atq22.P

tbox(equiv(b, and(∼a, a))).
tbox(equiv(f, exist(p, and(g, h))))).

query(and(exist(p, or(a, or(b, c))), and(exist(q, or(d, or(e, f))), forall(p, and(c, d))))).
Subconcepts List: [forall(p,and(c,d)),and(exist(q,or(d,or(e,exist(p,and(g,h))))),forall(p,and(c,d))),
and(exist(p,or(a,or(and(∼a,a),c))),and(exist(q,or(d,or(e,exist(p,and(g,h))))),forall(p,and(c,d))))]

atq23.P

tbox(equiv(map,and(exist(contains, words), ∼or(pictures, chapters)))).
tbox(equiv(nonfiction,and(∼fiction, and(exist(contains, chapters),forall(contains, words)))))
tbox(equiv(childrensbook, forall(contains, and(pictures, ∼words)))).
tbox(equiv(fiction,or(map, and(chapters, words))))).

query(and(nonfiction, childrensbook)).
Subconcepts List: [forall(contains,and(pictures,∼words)),and(and(or(forall(contains,∼words),or(pictures,chapters)),or(∼chapters,∼words)),and(exist(contains,chapters),forall(contains,words))),
forall(contains,and(pictures,∼words))]

atq24.P

tbox(equiv(parentdog,or( papadog,mamadog))).
tbox(equiv(husbanddog,and( maledog, exist(haswife, femaledog)))).
tbox(equiv(maledog,and( dog, ∼female)))).
tbox(equiv(papadog,and( maledog, exist(haschild, dog)))).
tbox(equiv(femaledog,and( dog, female))).
tbox(equiv(wifedog,and( femaledog, exist(haschild, maledog)))).
tbox(equiv(puppy, and(or( maledog, femaledog), and(exist(hasmother, mamadog), and(exist(hasfather, papadog), ∼exist(haschild, dog)))))).
tbox(equiv(mamadog,and( femaledog, exist(haschild, dog))))).
query(and(maledog, femaledog)).
Subconcepts List: [female,and(dog,female),and(and(dog,∼female),and(dog,female))]
A.3 Queries with Cyclic TBox

ctq1.P

tbox(equiv(man, and(person, male))).
tbox(equiv(woman, and(person, ~man))).
tbox(equiv(mother, and(woman, exist(hasChild, person)))).
tbox(equiv(father, and(man, exist(hasChild, person)))).
tbox(equiv(parent, exist(hasChild, person))).
tbox(equiv(grandfather, and(man, exist(hasChild, father)))).
tbox(equiv(brother, and(man, exist(hasSibling, person)))).
tbox(equiv(sister, and(person, and(brother, exist(hasSibling, person))))).
tbox(equiv(luckyBrother, and(man, forall(hasSibling, sister)))).
tbox(equiv(uncle, and(man, exist(hasSibling, parent)))).
tbox(equiv(aunt, and(person, and(uncle, exist(hasSibling, parent))))).
tbox(equiv(momd, and(man, forall(hasChild, momd)))).

and(brother, sister).
Subconcepts List: [exist(hasSibling, person), and(or(or(~person, ~male), forall(hasSibling, ~person))),
exist(hasSibling, person), and(person, and(or(or(~person, ~male), forall(hasSibling, ~person))), exist(hasSibling, person)),
and(or(or(~person, ~male), forall(hasSibling, ~person))),
exist(hasSibling, person), and(person, and(or(or(~person, ~male), forall(hasSibling, ~person))), exist(hasSibling, person)),
exist(hasSibling, person)]

ctq2.P

and(brother, ~uncle).
Subgoals List: [or(or(~person, ~male), forall(hasSibling, forall(hasChild, ~person))],
and(and(and(person, male), exist(hasSibling, person)), or(or(~person, ~male), forall(hasSibling, forall(hasChild, ~person))))]

ctq3.P

and(man, woman)²
Subconcepts List: [or(~person, ~male), and(person, or(~person, ~male)), and(and(person, male), and(person, or(~person, ~male)))]

ctq4.P

and(luckyBrother, ~brother).
Subconcepts List: [or(or(~person, ~male), forall(hasSibling, ~person)), and(and(and(person, male), forall(hasSibling, and(person, or(or(~person, ~male), forall(hasSibling, ~person)))), exist(hasSibling, person))), or(or(~person, ~male), forall(hasSibling, ~person))]

ctq5.P

and(grandfather, ~father).
Subconcepts List: [or(or(~person, ~male), forall(hasChild, ~person)), and(and(and(person, male), exist(hasChild, and(and(person, male), exist(hasChild, person)))), or(or(~person, ~male), forall(hasChild, ~person)))]

²The queries from ctq3 to ctq11 will use the TBox of ctq2
APPENDIX A. TEST QUERIES

ctq6.P
and(grandfather, father).
Subconcepts List: [exist(hasChild,person), and(person,male), exist(hasChild,person)], and(and(person,male), exist(hasChild, and(person,male), exist(hasChild,person))), and(person,male), exist(hasChild,person))]

ctq7.P
and(grandfather, sister).
Subconcepts List: [exist(hasSibling,person), and(or(person,~male), forall(hasSibling,~person)), exist(hasSibling,person)], and(person, and(or(person,~male), forall(hasSibling,~person)), exist(hasSibling,person)), and(and(person,male), exist(hasChild, and(person,male), exist(hasChild,person))), and(person, and(or(person,~male), forall(hasSibling,~person)), exist(hasSibling,person))]

ctq8.P
and(momd, ~father).
Subgoals List: [or(or(person,~male), forall(hasChild,~person)), and(momd, or(person, ~male), forall(hasChild,~person))]

ctq9.P
and(momd, exist(hasChild, woman)).
Subgoals List: [exist(hasChild, and(person,~male)), and(momd, exist(hasChild, and(person,~male))))]

ctq10.P
and(momd, father, ~parent).
Subgoals List: [forall(hasChild,~person), and(momd, and(person,male), exist(hasChild,person)), forall(hasChild,~person)]

ctq11.P
and(momd, and(grandmother, exist(hasChild, aunt))).
Subgoals List: [exist(hasChild, and(person, and(or(person,~male), forall(hasSibling, forall(hasChild,~person)))), exist(hasSibling, exist(hasChild,person)))), and(person, and(or(person,~male), exist(hasChild, and(person,male), exist(hasChild,person)))), exist(hasChild, and(person, and(or(person,~male), forall(hasSibling, forall(hasChild,~person)))), exist(hasSibling, exist(hasChild,person)))), and(momd, and(person,male), exist(hasChild, and(person,male), exist(hasChild,person)))), exist(hasChild, and(person, and(or(person,~male), forall(hasSibling, forall(hasChild,~person)))), exist(hasSibling, exist(hasChild,person))))]}


BRICS. The mona project. [http://www.brics.dk/mona/index.html](http://www.brics.dk/mona/index.html).


