Technische Universität Dresden
Fakultät Informatik

Master's Thesis on

Reasoning with Boolean ABoxes

by

Jigneshkumar Ramjibhai Viradia

Overseeing Professor : **Prof. Dr. Franz Baader**
Advisor : **Hongkai Liu**

Dresden, December 2008

Technische Universität Dresden

Author	:	**Jignshkumar Ramjibhai Viradia**
Registeration number	:	**3357602**
Title	:	**Reasoning with Boolean ABoxes**
Degree	:	**Master of Science**
Date of submission	:	**4 December, 2008**

Declaration

Hereby I certify that the thesis has been written by me. Any help that I have received in my research work has been acknowledged. In addition, I certify that I have not used any auxiliary sources and literature except those I cited in the thesis.

Jigneshkumar Ramjibhai Viradia

Abstract

ABoxes in Description Logics are used to represent factual knowledge of the world. In many situations an ABox is not enough to represent the knowledge of the world under consideration, as there is just an implicit boolean AND ($\wedge$) operator between each ABox assertion in the ABox. A Boolean ABox allows us to state Boolean combination of ABox assertions. The present work deals with the consistency problem of Boolean ABoxes in Description Logics (DLs).

Five years ago, a reduction algorithm to check consistency of $\mathcal{ALC}$ Boolean ABoxes has been introduced, in which an $\mathcal{ALC}$ Boolean ABox is reduced to a satisfiability preserving knowledge base, possibly in a more expressive description logic. We also argue that the reduction algorithm can be extended to check consistency of $\mathcal{ALCO}$ Boolean ABoxes. Motivated by ongoing research in Satisfiability Modulo Theory (SMT), we applied the DPLL(T) approach of SMT to check consistency of Boolean ABoxes. We have implemented a SAT solver based on DPLL procedure with state-of-the-art performance enhancing techniques like backjumping and conflict-driven-lemma-learning. By interconnecting the SAT solver with a DL reasoner using a special communication strategy, we developed a DPLL(T) solver to check consistency of $\mathcal{ALCO}$ Boolean ABoxes.

We have also implemented a reasoner to check consistency of $\mathcal{ALCO}$ Boolean ABoxes based on the reduction algorithm on top of an existing DL reasoner having a reasoning service to check consistency of $\mathcal{ALCO}$ ABoxes. In both of the implementations for checking consistency of Boolean ABoxes, Java is used as the programming language. Both of the implementations are tested, compared, and analyzed on the input stemming from the ABox update problem.

List of Figures

Contents

Chapter 1

Introduction

Description Logic is a major field of research within knowledge representation and reasoning (KRR), which is a subfield of Artificial Intelligence (AI). Research in the field of KRR is mainly focused on methods for providing high-level descriptions of the world that can be effectively used to build intelligent applications. In this context, intelligent refers to the ability of a system to find implicit consequences of its explicitly represented knowledge. Such systems are known as knowledge-based systems [13]. The knowledge-based systems should provide their users an inference tool that can deduce implicit knowledge. The main requirement of such inference tools is as shown below :

Inferred results should depend on semantics of the knowledge representation languages, but not on the syntactic representation, on each semantically equivalent knowledge, there should be similar inferred results.

Years ago, using the knowledge representation languages like *semantic networks* [19] and *frames* [12], the knowledge was represented by means of some ad hoc data structures, and reasoning was done by similarly ad hoc procedures. As both of them were lacking precise semantics, many systems behaved differently from the others, in many cases despite virtually identical-looking components and even identical relationship names. Considering this potential problem, one important step in this direction was the recognition that frames could be given a semantics by relying on first-order logic [10]. The further research in the field has shown that semantic networks and frames could be seen as fragments of first-order logic [6]. Later KL-ONE system [7] became the origin for an interesting field of research called Description Logic.

Description Logics (DLs)[1] [2] are members of a very successful set of logic-based knowledge representation formalisms which can be used to represent

[1]Also known as terminological representation languages, concept languages, term subsumption languages, and KL-ONE-based knowledge representation languages.

the conceptual knowledge of an application domain in a structured and formally well-understood way [4]. The name *description logic* is chosen by the fact that, on the one hand, the important notions of the domain are described by *concept descriptions*, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL, on the other hand, DLs are equipped with a formal, logic-based semantics, which the predecessors of the DLs such as semantic networks and frames were lacking.

DLs are used (or could be used) in various areas, such as natural language processing, databases, and reasoning about Web Services [3], but their most notable success so far is the adoption as logical foundation of the Web Ontology Languages OWL[2] and OWL 2[3].

Any Description Logic knowledge base may have two components namely an ABox and a TBox. The ABox is used to state known facts about the domain under consideration. Anyone is interested in building consistent knowledge base because from logical point of view arbitrary conclusions could be drawn from the given inconsistent knowledge base. In many situations the ABox is not enough to represent the factual knowledge as there is just one implicit Boolean AND ($\wedge$) operator present between each of the ABox assertions in the ABox. A Boolean ABox which allows to use the Boolean operators like Boolean AND ($\wedge$), Boolean OR ($\vee$), and Boolean NOT ($\neg$), is needed to represent the knowledge of many domains. An important reasoning service to check consistency of Boolean ABoxes should be investigated.

Five years ago, in [1] the consistency problem to deal with $\mathcal{ALC}$ Boolean ABoxes has been described. They have developed an algorithm which reduces the $\mathcal{ALC}$ Boolean ABox to a satisfiability preserving knowledge base, but possibly in a more expressive DL $\mathcal{ALCHQ}$. We have extended the algorithm to check consistency of $\mathcal{ALCO}$ Boolean ABoxes, in which $\mathcal{ALCO}$ Boolean ABoxes are reduced to a satisfiability preserving knowledge base, possibly in a more expressive DL $\mathcal{ALCHOQ}$. We have also implemented the extended reduction algorithm in Java. Once an $\mathcal{ALCO}$ Boolean ABox is translated to a satisfiability preserving knowledge base (KB), the consistency of the KB is checked using a DL reasoner. We use Pellet 1.5.1 as the DL reasoner.

The Boolean satisfiability problem (SAT) is to decide satisfiability of a propositional logic formula. The Satisfiability Modulo Theory (SMT) problem is to decide satisfiability of a (Boolean combination of atoms in) more expressive theory than propositional logic. Most of the modern SAT solvers are based on Davis-Putnam-Logemann-Loveland (DPLL) procedure together

[2]Web Ontology Language, look at http://www.w3.org/TR/owl-ref/

[3]http://www.w3.org/TR/owl2-semantics/

with many performance enhancing techniques. A DPLL(T) is one of the approach to solve the SMT problem. The core idea behind the DPLL(T) [15] approach is to take advantage of many years of research in SAT problem and well established theory solvers, which are able to check consistency of conjunction of literals of the theory T under consideration. In the DPLL(T) approach, a general DPLL(X) engine is implemented, which is independent of any of the theory T, such that the DPLL(X) engine can be interconnected with any theory solver for the theory T under consideration. The minimal requirement of a theory (T) solver is to check consistency of conjunction of theory literals. The main task of the theory solver is to check consistency of the (partial) model (which contains theory literals), which is built by the DPLL(X) engine. If the partial model is consistent, then the operation of the DPLL(X) engine is not interupted, but if it is not consistent, then an appropriate lemma is constructed and learned. The interconnection between the general DPLL(X) engine and a theory solver for theory T can be done using different type of communication strategies.

We have created and implemented a DPLL(T) approach where T is $\mathcal{ALCO}$ Boolean ABox. As DPLL(X) engine works on a formula in conjunctive normal form (CNF), we first transform an $\mathcal{ALCO}$ Boolean ABox into a CNF ABox. State-of-the-art DL reasoners can not check consistency of negative ABox literals (negated ABox assertions), so we convert negative ABox literal to equivalent positive ABox literals (ABox assetions). We have implemented the DPLL based SAT solver (DPLL(X) engine) with performance enhancing techniques like backjumping and conflict-driven-lemma-learning. We choose Pellet 1.5.1 as DL reasoner. Thereafter, we combine the DPLL(X) engine and Pellet 1.5.1 using a special strategy of communication, consequently we get the DPLL(T) solver to check consistency of $\mathcal{ALCO}$ Boolean ABoxes. In our DPLL(T) solver, every ABox assertion is treated as propositional atom. The goal of the DPLL(T) is to find a (propositional) model using the general DPLL(X) engine such that the obtained model is also consistent according to the semantics of the $\mathcal{ALCO}$ ABox, if such a model exists.

The rest of the thesis is organized as follow:

In chapter 2, at the beginning we introduce concept name, role name and individual name, then we give syntax and semantics of DL $\mathcal{ALC}$. Thereafter, we tell about extensions of $\mathcal{ALC}$, in particular, *nominal* and *qualified number restriction*. Afterwards, the introduction about ABox, role hierarchy, TBox, knowledge base, Boolean ABox and CNF ABox is given. The chapter 2 is closed by giving the introduction about explanation in case of ABox inconsistency, and naming of some of the extensions of $\mathcal{ALC}$.

In chapter 3, we discuss about two different approaches to check consistency of Boolean ABoxes. The first one is the DPLL(T) based approach, and the second one is the reduction based approach. In the DPLL(T) approach we give detail about DPLL procedure, performance enhancing techniques, and the DPLL(T) algorithm.

In chapter 4, we give detail about how we implemented both approaches, why we choose Pellet [20] as DL reasoner, the KRSS syntax, the techniques we used to deal with the negative ABox literals which are not supported by Pellet reasoner.

The chapter 5 is about the testing data, comparison and analysis of the both approaches based on the results obtained from the testing data stemming from the ABox update problem.

In chapter 6, we draw some conclusions and put light on the related and future research.

Chapter 2

Description Logics

In this chapter we describe the basic notions in Description Logics (DLs) [2]. We also define a Boolean ABox and explanations for the ABoxes.

In DL, any domain of interest can be described using three main building blocks :

- concept names: Person, Student, Male, ...
- role names: teaches, supervised-by, is-student-of, ...
- individual names: Albert, Mary, ...

Using the above building blocks, we can build concept descriptions (and sometimes role descriptions) with the help of certain constructors provided by the particular DL. Each DL has different expressive power depending on the constructors used by it.

In this thesis we will use notation N_C, N_R, and N_I for disjoint sets of concept names, role names and individual names respectively.

Now we will introduce syntax and semantics of DL $\mathcal{ALC}$. $\mathcal{ALC}$ has concept constructors namely conjunction, disjunction, negation, existential restriction and value restriction.

Definition 2.0.1 ($\mathcal{ALC}$ Syntax)**.** *The set of* $\mathcal{ALC}$ concept descriptions *(over* N_C *and* N_R*) is inductively defined as shown below:*

- *Every concept name* $A \in N_C$ *is an* $\mathcal{ALC}$ concept description.
- $\top$ *(top concept) and* $\bot$ *(bottom concept) are* $\mathcal{ALC}$ concept descriptions.
- *if* C *and* D *are* $\mathcal{ALC}$ *concept descriptions and* $r \in N_R$*, then the following are* $\mathcal{ALC}$ concept descriptions *:*

Name	Syntax	Semantics
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Conjunction	$\mathrm{C} \sqcap \mathrm{D}$	$\mathrm{C}^{\mathcal{I}} \cap \mathrm{D}^{\mathcal{I}}$
Disjunction	$\mathrm{C} \sqcup \mathrm{D}$	$\mathrm{C}^{\mathcal{I}} \cup \mathrm{D}^{\mathcal{I}}$
Negation	$\neg \mathrm{C}$	$\Delta^{\mathcal{I}} \setminus \mathrm{C}^{\mathcal{I}}$
Existential restriction	$\exists r.\mathrm{C}$	$\{\mathrm{x} \in \Delta^{\mathcal{I}} \mid \exists \mathrm{y} \in \Delta^{\mathcal{I}} \colon\ (\mathrm{x}, \mathrm{y}) \in r^{\mathcal{I}} \wedge\ \mathrm{y} \in \mathrm{C}^{\mathcal{I}}\}$
Value restriction	$\forall r.\mathrm{C}$	$\{\mathrm{x} \in \Delta^{\mathcal{I}} \mid \forall \mathrm{y} \in \Delta^{\mathcal{I}} \colon\ (\mathrm{x}, \mathrm{y}) \in r^{\mathcal{I}} \rightarrow \mathrm{y} \in C^{\mathcal{I}}\}$

Figure 2.1: Semantics of concept constructors in DL $\mathcal{ALC}$

- $C \sqcap D$
- $C \sqcup D$
- $\neg C$
- $\exists r.C$
- $\forall r.C$

The semantics of Description Logics is given in terms of *interpretations*. An *interpretation* consists of an *interpretation domain*, which is a non-empty set, and an *interpretation function*, which assigns extensions to concept, role and individual names, i.e.,

- a subset of the interpretation domain to a concept name,
- a binary relation on elements of interpretation domain to a role name,
- an element of the interpretation domain to an individual name.

Definition 2.0.2 ($\mathcal{ALC}$ Semantics)**.** *An* interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *consists of a non-empty set* $\Delta^{\mathcal{I}}$*, the domain of* $\mathcal{I}$*, and an* interpretation function $\cdot^{\mathcal{I}}$ *which assigns each concept name* A *a set* $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$*, each role name* r *a binary relation* $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$*, and each individual name* b *an element* $b^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$*.*

The interpretation function is extended to concept descriptions as shown in Figure 2.1.

One is free to choose *unique name assumption (UNA)* or not depending on the particular application under consideration. Under *unique name assumption (UNA)*, we assume that different individual names denote different objects. Any two different individual names will not be mapped to a common object under any interpretation $\mathcal{I}$.

Definition 2.0.3 (Unique name assumption)**.** *For all interpretations $\mathcal{I}$, if a and b are distinct individual names, i.e., $a \neq b$, then $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.*

In all of the examples given below in this thesis, we will adapt the following criteria for the names of concepts, roles, and individuals:

- to represent a concept name the first letter of the name will be in uppercase, but the other letters will be in lowercase,
- to represent a role name all the letters will be in lowercase,
- to represent an individual name all the letters will be in uppercase.

A *nominal* constructor allows us to use individual names also within concept descriptions : if b is an individual name, then $\{b\}$ is a concept description, called nominal, which is interpreted as a singleton set. Using the individual name INDIA, we can describe the people who have visited INDIA by $\mathsf{People} \sqcap \exists \mathsf{visited}.\{\mathsf{INDIA}\}$.

For an interpretation $\mathcal{I}$, $\{b\}^{\mathcal{I}} = \{b^{\mathcal{I}}\}$.

Now we will define *qualified number restrictions*, which allow to state counting constraints on the number of domain elements that are related via a particular role and belong to a certain concept description.

Definition 2.0.4 (Qualified number restrictions)**.** *If n is a non-negative integer, r is a role name, and C is a concept description, then the following are* qualified number restrictions*:*

- $(\leq n\ r\ C)$
- $(\geq n\ r\ C)$

For the interpretation $\mathcal{I}$,

$(\leq n\ r\ C)^{\mathcal{I}} = \{\ d \in \Delta^{\mathcal{I}} \mid \sharp\{\ e \mid (d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\} \leq n\ \}$

$(\geq n\ r\ C)^{\mathcal{I}} = \{\ d \in \Delta^{\mathcal{I}} \mid \sharp\{\ e \mid (d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\} \geq n\ \}$

$\sharp N$ denotes the cardinality of the set N.

For example, "students who are taking at most three advanced courses" can be described using qualified number restriction as shown below:

$$\mathsf{Student} \sqcap (\leq 3 \text{ takes } \mathsf{Advanced\_course})$$

Definition 2.0.5 (Concept assertion)**.** *If r is a role name, and C is a concept description, then an assertion of the form $C(a)$ is called a* concept assertion*.*

A concept assertion asserts individual name as an instance of a concept description. Using a concept assertion $C(a)$, one states that a belongs to C.

For example,"Mary is a professor and also teaches a graduate course" can be represented using the concept assertion shown below :

$$\mathsf{Professor} \sqcap \exists \mathsf{teaches.Graduatecourse}(\mathsf{MARY})$$

An interpretation $\mathcal{I}$ *satisfies* the concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

Definition 2.0.6 (Role assertion)**.** *If r is a role name, and a, b are individual names, then an expression of the form $r(a, b)$ is called a* role assertion*.*

The role assertion describes relationship between two individual names. Using a role assertion $r(a, b)$, one states that individual name a is related to individual name b via role name r. For example, "John likes Mary" can be represented using the role assertion as shown below:

$$\mathsf{likes}(\mathsf{JOHN}, \mathsf{MARY})$$

The interpretation $\mathcal{I}$ *satisfies* the role assertion $r(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$.

We also recognize a concept assertion or a role assertion as an *ABox assertion.*

Definition 2.0.7 (ABox)**.** *An* ABox *is a finite set of ABox assertions.*

An ABox is used to give the description of the application domain (world) using conceptual membership of individuals and role names relating individuals. An ABox describes a given application domain in incomplete way, i.e., from absence of information we do not conclude that it does not hold (i.e., its negation holds). This is in contrast to databases. The ABoxes are interpreted with the "open world assumption".

An interpretation $\mathcal{I}$ *satisfies* the ABox $\mathcal{A}$ if it satisfies each ABox assertion in $\mathcal{A}$. In this case interpretation $\mathcal{I}$ is called a *model* of the ABox $\mathcal{A}$.

For example, the ABox shown in Figure 2.2 states that "Eric is a student and has hobby sports", and "Eric has friend Kapil".

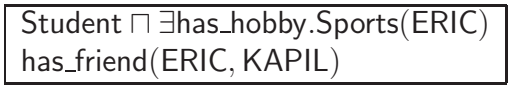

Figure 2.2: An example of an ABox

In many situations an ABox is not enough to express the knowledge of a given application domain for knowledge engineers. It may be convenient for knowledge engineers if they are provided a facility to use arbitrary boolean combination of ABox assertions, rather than only implicit boolean AND ($\wedge$), which is used in case of the ABox.

Definition 2.0.8 (ABox literal, ABox clause, unit-positive-clause)**.** *An* ABox literal *is an ABox assertion al or its negation $\neg al$, where al is called a* positive ABox literal*, and $\neg al$ is called a* negative ABox literal*. An* ABox clause *is a finite disjunction of ABox literals $al_1 \vee \ldots \vee al_n$. A* unit-positive-clause *is a clause containing only one positive ABox literal al.*

The definition of a Boolean ABox is given below.

Definition 2.0.9 (Boolean ABox)**.** *A* Boolean ABox *can be inductively defined as shown below :*

- *Every ABox assertion is a* Boolean ABox*.*
- *If $\mathcal{BA}$ is a Boolean ABox, then $\neg\mathcal{BA}$ is a* Boolean ABox*.*
- *If $\mathcal{BA}_1$ and $\mathcal{BA}_2$ are Boolean ABoxes, then the following are* Boolean ABoxes *:*
  - $\mathcal{BA}_1 \vee \mathcal{BA}_2$
  - $\mathcal{BA}_1 \wedge \mathcal{BA}_2$

The semantics of a Boolean ABox can be given based on interpretation.

Definition 2.0.10 (Boolean ABox Semantics)**.** *Let $\mathcal{I}$ be an interpretation. Semantics of Boolean ABox can be inductively defined as shown below :*

- *If a Boolean ABox $\mathcal{BA}$ is of the form al, where al is an ABox assertion, then $\mathcal{I}$* satisfies *$\mathcal{BA}$ if $\mathcal{I}$ satisfies al.*
- *If a Boolean ABox $\mathcal{BA}$ is of the form $\neg\mathcal{BA}_0$, where $\mathcal{BA}_0$ is a Boolean ABox, then $\mathcal{I}$* satisfies *$\mathcal{BA}$ if $\mathcal{I}$ does not satisfy $\mathcal{BA}_0$.*

- *If a Boolean ABox* $\mathcal{BA}$ *is of the form* $\mathcal{BA}_1 \vee \mathcal{BA}_2$*, where* $\mathcal{BA}_1$ *and* $\mathcal{BA}_2$ *are Boolean ABoxes, then* $\mathcal{I}$ satisfies $\mathcal{BA}$ *if* $\mathcal{I}$ *satisfies at least one of* $\mathcal{BA}_1$ *and* $\mathcal{BA}_2$*.*

- *If a Boolean ABox* $\mathcal{BA}$ *is of the form* $\mathcal{BA}_1 \wedge \mathcal{BA}_2$*, where* $\mathcal{BA}_1$ *and* $\mathcal{BA}_2$ *are Boolean ABoxes, then* $\mathcal{I}$ satisfies $\mathcal{BA}$ *if* $\mathcal{I}$ *satisfies both* $\mathcal{BA}_1$ *and* $\mathcal{BA}_2$*.*

If the DL under consideration is $\mathcal{ALC}$ then a Boolean ABox is called an $\mathcal{ALC}$ Boolean ABox. One can observe that in case of an ABox there is just an implicit boolean AND operator ($\wedge$) between every ABox assertion present in the ABox.

Any two Boolean ABoxes are *equivalent* if they have a same set of models.

Suppose that one has knowledge of a particular domain as shown below:

- John is a student or he does not like Mary
- Mary is a student
- John likes Susane
- Peter is not a teacher or has a sister Lara.

The above mentioned knowledge can be represented by the Boolean ABox as shown in Figure 2.3.

Definition 2.0.11 (CNF ABox)**.** *A* CNF ABox *is a finite conjunction of ABox clauses.*

A *CNF ABox* is written as $AC_1 \wedge \ldots \wedge AC_n$ or AC_1, ..., AC_n, where every AC_i is an ABox clause.

Every Boolean ABox can be converted to a CNF ABox. A Boolean ABox and its corresponding transformation into a CNF ABox are equivalent.

A CNF ABox (an ABox) is a special case of a Boolean ABox.

Definition 2.0.12 (General concept inclusion)**.** *A* general concept inclusion (GCI) *is an expression of the form* $C \sqsubseteq D$*, where* C *and* D *are concept descriptions.*

For example, every male is human can be represented using the following GCI :

$$\mathsf{Male} \sqsubseteq \mathsf{Human}$$

An interpretation $\mathcal{I}$ *satisfies* the GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Definition 2.0.13 (Role inclusion)**.** *A* role inclusion *is an expression of the form $r \sqsubseteq s$, where r and s are role names.*

To represent "each daughter of someone is also child of him/her, and each son of someone is also child of him/her" can be represented using two role inclusions shown below:

$$\begin{aligned} \mathsf{is\_daughter\_of} &\sqsubseteq \mathsf{is\_child\_of} \\ \mathsf{is\_son\_of} &\sqsubseteq \mathsf{is\_child\_of} \end{aligned}$$

An interpretation $\mathcal{I}$ *satisfies* the role inclusion $r \sqsubseteq s$ if $r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$.

Definition 2.0.14 (Role hierarchy)**.** *A* role hierarchy [5] *is a finite set of role inclusions.*

A knowledge engineer has generally limited knowledge about how one concept description is related to the other or how one role name is related to the other. All the known relationships between concept descriptions (or role names) are collectively build a *TBox*.

We will recognize a general concept inclusion or a role inclusion as an *inclusion*.

Definition 2.0.15 (TBox)**.** *A* TBox *is a finite set of inclusions.*

An example of a TBox is shown in Figure 2.4. The first two inclusions are GCIs, and the last two inclusions are role inclusions.

An interpretation $\mathcal{I}$ *satisfies* the TBox $\mathcal{T}$ if $\mathcal{I}$ satisfies every inclusion in $\mathcal{T}$. In this case $\mathcal{I}$ is called a *model* of $\mathcal{T}$.

Definition 2.0.16 (Knowledge base)**.** *A* knowledge base (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *contains two components, a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$.*

$$\begin{array}{|l|}\hline ((\mathsf{Student}(\mathsf{JOHN}) \vee \neg(\mathsf{likes}(\mathsf{JOHN}, \mathsf{MARY})))\wedge \\ (\mathsf{Student}(\mathsf{MARY}))\wedge \\ (\mathsf{likes}(\mathsf{JOHN}, \mathsf{SUSANE}))\wedge \\ (\neg(\mathsf{Teacher}(\mathsf{PETER})) \vee \mathsf{has\_sister}(\mathsf{PETER}, \mathsf{LARA}))) \\ \hline \end{array}$$

Figure 2.3: An example of a Boolean ABox

Male	$\sqsubseteq$	Human
Female	$\sqsubseteq$	Human
is_daughter_of	$\sqsubseteq$	is_child_of
is_son_of	$\sqsubseteq$	is_child_of

Figure 2.4: An example of a TBox

An interpretation $\mathcal{I}$ is a *model* of knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\mathcal{I}$ is a common model of $\mathcal{T}$ and $\mathcal{A}$.

Reasoning allows us to derive implicit knowledge from the explicitly represented one. Subsumption, equivalence, satisfiability, consistency and instance are the basic (standard) reasoning services. In particular, we are interested in consistency of an ABox (a Boolean ABox).

An ABox $\mathcal{A}$ is *consistent* with respect to a TBox $\mathcal{T}$, if there is an interpretation that is a model of both $\mathcal{A}$ and $\mathcal{T}$, i.e., if there is a common model for $\mathcal{A}$ and $\mathcal{T}$.

It is hard to avoid *errors* for knowledge engineers during designing and maintaining large ABoxes. By the term *errors* we meant logical inconsistency. After receiving information that the ABox under construction is inconsistent, to know an explanation, i.e., exactly which ABox assertions are responsible for inconsistency is helpful for knowledge engineers to revise the ABox.

In the following we give a definition of an explanation for inconsistent ABox.

Definition 2.0.17 (explanation)**.** *Let $\mathcal{A}$ be an inconsistent ABox. Let $\mathcal{A}'$ be a subset of $\mathcal{A}$, and $\mathcal{A}'$ is inconsistent, then $\mathcal{A}'$ is called an* explanation*, which is an ABox. Let $\mathcal{A}''$ (be an ABox) obtained by removing an arbitrary ABox assertion from an explanation $\mathcal{A}'$, and if $\mathcal{A}''$ is consistent, then $\mathcal{A}'$ is called a* minimal explanation*.*

If the $\mathcal{ALC}$ is extended by *nominal*, then the new Description Logic is called $\mathcal{ALCO}$. If the $\mathcal{ALC}$ is extended by *nominal*, and *qualified number restriction*, then the new Description Logic is called $\mathcal{ALCOQ}$. If the $\mathcal{ALC}$ is extended by *nominal*, *qualified number restriction*, and *role hierarchies* then the new Description Logic is called $\mathcal{ALCHOQ}$. For more detailed information about naming scheme for Description Logics, reader may look at [2].

Chapter 3

Reasoning with Boolean ABoxes

In this chapter we will describe two different approaches to check consistency of $\mathcal{ALCO}$ Boolean ABoxes. The first approach is based on DPLL(T) [15], and the second approach is based on reduction of Boolean $\mathcal{ALCO}$ ABox to a satisfiability preserving knowledge base, but possibly in a more expressive DL $\mathcal{ALCHOQ}$ [1].

The Satisfiability Modulo Theory (SMT) [14] problem is to decide satisfiability of a (Boolean combination of atoms in) more expressive theory T than propositional logic. Given a formula F in theory T, to know whether F is T-satisfiable, i.e., whether there exists a model of T that is also a model of F. The DPLL(T) is one of the approaches to solve SMT problem for theory T.

3.1 The DPLL(T) Approach

A SAT solver, based on Davis-Putnam-Logemann-Loveland (DPLL) procedure is developed, which has the general goal to know whether a given propositional logic formula in CNF is satisfiable or not. A SAT solver based on DPLL can be developed in such a way that it is easy to integrate with any theory solver without much modification, such a SAT solver could be recognized as the DPLL(X) engine. In the DPLL(T) approach, the variable X of a DPLL(X) engine is instantiated with a T-solver, a theory solver for theory T under consideration. The T-solver should have ability to check consistency of conjunction of theory (T) literals. We developed a DPLL(T) solver, where T is an $\mathcal{ALCO}$ Boolean ABox.

3.1.1 The Boolean Satisfiability Problem

The problem of deciding satisfiability of a propositional (logic) formula is known as the Boolean satisfiability problem (SAT) [9], i.e., given a propositional logic formula F, does there exist a model that makes F true?

Most of the SAT solvers are based on Davis-Putnam-Logemann-Loveland (DPLL) procedure [14]. The performance of the SAT solvers has been increased dramatically due to state-of-the-art performance enhancing techniques called *backjumping*, which is aimed at reducing the amount of the search space, and *conflict-driven-lemma-learning*, which is used to prevent the occurrence of the similar conflict again [15]. Due to rapid increase in performance of the DPLL based SAT-solvers, adoption of DPLL has been started to more expressive Logics than propositional logic. In this thesis, we have shown the first experiment of the DPLL(T) approach to check consistency of $\mathcal{ALCO}$ Boolean ABoxes.

3.1.2 The DPLL Framework

In this section we will represent some formal preliminaries on propositional logic and transition systems. We also describe the DPLL system with backtrack.

Let P be a fixed finite set of the propositional symbols. If $p \in P$, then p is called an *atom*. If $p \in P$, then p and $\neg p$ are *literals* of P. The negation of a *positive literal* l will be written as $\neg l$, and the negation of a *negative literal* $\neg l$ will be written as l. A *clause* is a finite disjunction of literals $l_1 \vee \ldots \vee l_n$. A *unit clause* is a clause consisting of a single literal.

A Boolean combination of atoms is called a *formula*.

A *formula in CNF* is a finite conjunction of clauses $C_1 \wedge \ldots \wedge C_n$. We also write $C_1 \wedge \ldots \wedge C_n$ as $C_1, \ldots, C_n$.

A (*partial truth*) *assignment* M is an ordered set of literals such that $\{p, \neg p\} \not\subseteq M$ for all p *in* P. A (partial truth) assignment is also called *(partial) assignment*. A literal l is *true* in M if $l \in M$, it is *false* in M if $\neg l \in M$. A literal l is *undefined* in M, if it is neither true nor false in M. M is called *total* over P, if every literal of P is false or true in M.

A clause C is *true* in M if at least one of its literals is true in M. A clause C is *false* in M, if all of its literals are false in M (written as $M \models \neg C$). A formula in CNF is *true* in M, if all of its clauses are true in M. If F, a formula in CNF is true in M, we call F is satisfied by M, and is denoted by $M \models F$. M is a *model* of *formula* F, if F is true in (satisfied by) M.

The DPLL procedure is modeled by the transition relation over the *states* [14]. The DPLL procedure is used to find out a model for F, if such a model

exists.

If F_1 and F_2 are formulas in CNF, then we will write $F_1 \models F_2$, if F_2 is true in all the models of F_1. It is read as "F_2 is *entailed by* F_1", or "F_2 is a *logical consequence* of F_1".

A *state* is either FailState, a special state in DPLL procedure or a pair of the form $M \parallel F$, where F is a formula in CNF and M is a (partial truth) assignment.

The *decision literals* are the literals added to partial assignment by the Decide rule given in definition 3.1.1. Each literal in (partial truth) assignment may have an *annotation*, a status that shows either it is decision literal or not. If a literal l present in the partial assignment is a decision literal, then it will be denoted as l^d. Empty assignment will be denoted as $\emptyset$.

A clause C is called a *conflicting clause* (*falsified clause*) in a *state* $M \parallel F$, C if $M \models \neg C$, where M is an assignment, F is a finite conjunction of clauses.

The DPLL procedure is modeled by means of a set of states together with a binary relation $\Longrightarrow$ over the states, called a *transition relation*. Transition from a state S_1 to S_2, which is denoted by $S_1 \Longrightarrow S_2$, is done by applying any of the rules given in definition 3.1.1. Any sequence of transitions of the form $S_0 \Longrightarrow S_1$, $S_1 \Longrightarrow S_2$, ... is called a *derivation*, and it is denoted by $S_0 \Longrightarrow S_1 \Longrightarrow S_2$, ... Any subsequence of a derivation is called a *subderivation*.

Let an assignment M has the form $M_0\ l_1^d\ M_1 \ldots l_k^d\ M_k$, where every l_i^d is a decision literal in M. Then the *decision level* of state $M \parallel F$ is k, and the *decision level* of all the literals in each $l_i\ M_i$ is i.

Definition 3.1.1 (DPLL system with backtrack)**.** *The* DPLL system with backtrack *[14] consists of the four rules -* UnitPropagate*,* Decide*,* Fail *and* Backtrack *as shown below:*

UnitPropagate*:*

$M \parallel F, C \vee l \Longrightarrow M\ l \parallel F, C \vee l$ *if:*

- $M \models \neg C$,
- l *is undefined in* M.

The clause $C \vee l$ *is called* unit clause *in the* UnitPropagate *rule shown above.*

Decide*:*

$M \parallel F \Longrightarrow M\ l^d \parallel F$ *if:*

- l *or* $\neg l$ *occurs in a clause of* F*,*
- l *is undefined in* M*.*

Fail*:*

$M \parallel F, C \Longrightarrow FailState$ *if:*

- $M \models \neg C$*,*
- M *contains no decision literals.*

Backtrack*:*

$M\ l^d\ N \parallel F, C \Longrightarrow M\ \neg l \parallel F, C$ *if:*

- $M\ l^d\ N \models \neg C$*,*
- N *contains no decision literals.*

One can use the rules given in definition 3.1.1 for deciding satisfiability of arbitrary formula in CNF F by generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow \ldots \Longrightarrow S_n$, where S_n is the *final state*, a state where there is not any rule given in definition 3.1.1 is applicable. Such derivations are always finite [14], and
(i) F is *unsatisfiable* if, and only if, the final state S_n is $FailState$, and
(ii) if S_n is of the form $M \parallel F$, then M is a *model* of F.

In the DPLL system with backtrack the second component F of a state $M \parallel F$ remains unchanged during application of any of the rules presented in definition 3.1.1. A brief description of each of the rule mentioned above is given below :

- UnitPropagate : In order to satisfy a formula in CNF, all of its clauses must become *true*. Hence, if a clause of F contains a literal which is undefined, and the rest of the literals in that clause are *false*, then M should be extended to make l *true*.

- Decide : The Decide rule represents a case split. An undefined literal l is selected from F, and added to M. This added literal l is marked as *decision literal*, i.e., l^d. If $M\ l^d$ can not be extended to a model of F, then the alternative choice $M\ \neg l$ must be explored (by the application of the Backtrack rule).

- Fail : The Fail rule detects a *conflicting clause* C in F and produces *FailState* whenever M contains no decision literal.

- Backtrack : A *conflicting clause* C is detected, and if there exist a *decision literal* in the assignment, then the Backtrack rule backtracks one *decision level*, by replacing the most recent *decision literal* l^d in the (partial truth) assignment M by $\neg l$ and removing any subsequent literals from M, but $\neg l$ will be annotated as non-decision literal.

Example 3.1.2. *Let F be a formula in CNF as shown below. We denote atoms by natural numbers, and negation of atoms by overlining it.*

$\overline{1} \vee \overline{2}, \quad \overline{2} \vee 3, \quad 4, \quad \overline{2} \vee \overline{3}.$

Using the DPLL System with backtrack as mentioned in definition 3.1.1, and giving the UnitPropagate *rule higher priority than the* Decide *rule, we got the derivation as shown in Figure 3.1.*

$\emptyset$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(UnitPropagate)
4	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(Decide)
$4\ \overline{1}^d$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(Decide)
$4\ \overline{1}^d\ 2^d$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(UnitPropagate)
$4\ \overline{1}^d\ 2^d\ 3$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(Backtrack)
$4\ \overline{1}^d\ \overline{2}$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$	$\Longrightarrow$	(Decide)
$4\ \overline{1}^d\ \overline{2}\ \overline{3}^d$	$\parallel$	$\overline{1} \vee \overline{2},$	$\overline{2} \vee 3,$	$4,$	$\overline{2} \vee \overline{3}$		

Figure 3.1: A derivation

The last state of the derivation given in Figure 3.1 is *final state*, as any of the rule from the definition 3.1.1 could not be applied to the state. The (total) assignment (i.e., $4\ \overline{1}^d\ \overline{2}\ \overline{3}^d$) is a model of F. Thus, F is satisfiable.

The Modern DPLL Procedures

The state-of-the-art DPLL-based SAT solvers do not implement the DPLL system described in definition 3.1.1 above. A backjumping, a more general and more powerful backtracking, and *conflict-driven-lemma-learning* are used.

Definition 3.1.3 (Backjump)**.** *The* Backjump *rule is shown below:*

$M\ l^d\ N \parallel F, C \Longrightarrow M\ \ l' \parallel F, C$ *if:*

- $M\ l^d\ N \models \neg C$, *and*
- *there exists some clause* $C' \vee l'$ *such that :* $F, C \models C' \vee l'$ *and* $M \models \neg C'$, *and* l' *is undefined in* M, *and* l' *or* $\neg l'$ *occurs in* F *or in* $M\ l^d\ N$.

Clause C *in the* Backjump *rule is a conflicting clause. The clause* $C' \vee l'$ *is called a* backjump clause, *where* l' *is the literal that can be unit propagated in presence of the backjump clause* $C' \vee l'$.

The Backtrack rule always undoes the last decision literal l^d (in the partial assignment), going back to the previous level, and adding $\neg l$ to it. $(\mathsf{Conflict-driven-})\mathsf{Backjumping}$ generally backtracks more than one level by analyzing the reasons for the production of the conflicting clause. Backjumping jumps over the levels that are irrelevant to the conflict.

We will use abbreviated names UP, D, F, BT and BJ for the UnitPropagate, Decide, Fail, Backtrack and Backjump respectively in some of the examples given below.

Example 3.1.4 (Backjump)**.** *Let* F *be a formula in CNF as shown below:*

$$\overline{1} \vee 2, \quad \overline{3} \vee 4, \quad \overline{5} \vee 6, \quad \overline{7} \vee 8, \quad \overline{1} \vee \overline{8} \vee \overline{7}$$

Using the Backjump *rule, and giving the* UnitPropagate *rule higher priority than the* Decide *rule, we got the subderivation as shown in Figure 3.2.*

One can observe from the subderivation given in Figure 3.2 that before the application of the BJ *rule, the clause* $\overline{1} \vee \overline{8} \vee \overline{7}$ *was conflicting clause, i.e., it was* false *in the partial assignment* $1^d\ 2\ 3^d\ 4\ 5^d\ 6\ 7^d\ 8$. *The occurred conflict was consequence of the unit propagation of* 8 *of the decision* 7^d *together with the decision* 1^d . *Therefore, one can infer that the decision* 1^d *is incompatible with the decision* 7^d. F *entails clause* $\overline{1} \vee \overline{7}$. *Presence of such entailed clause*

would have allowed unit propagation at earlier decision level. That is why the Backjump *rule is more powerful, i.e., given a* backjump clause*, it can jump back more than one level (or at least one level), consequently the search space is reduced dramatically. In this example, the* BJ *jumps over the decision* 5^d*, its consequence* 6*, the decision* 3^d*, and its consequence* 4*, because any of them are not responsible for making the clause* $\overline{1} \vee \overline{8} \vee \overline{7}$ *a conflicting clause. Thus, after the application of* BJ*, the state* $1^d\ 2\ \overline{7} \parallel F$ *is reached.*

$\emptyset$	$\parallel$	$F \Longrightarrow$	(D)
1^d	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3^d$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3^d\ 4$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3^d\ 4\ 5^d$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3^d\ 4\ 5^d\ 6$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3^d\ 4\ 5^d\ 6\ 7^d$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3^d\ 4\ 5^d\ 6\ 7^d\ 8$	$\parallel$	$F \Longrightarrow$	(BJ)
$1^d\ 2\ \overline{7}$	$\parallel$	F	

Figure 3.2: An example of the Backjump rule

As said in [15], the construction of a *backjump clause* can be done by taking negation of all decision literals of the partial assignment, but in this case, the Backjump rule simulates the Backtrack rule. For example, one can construct *backjump clause* $\overline{1} \vee \overline{3} \vee \overline{5} \vee \overline{7}$ from the decision literals 1^d, 3^d, 5^d and 7^d present in the first component of the second last state of the subderivation shown in Figure 3.2. If one applies the Backjump rule, taking $\overline{1} \vee \overline{3} \vee \overline{5} \vee \overline{7}$ as backjump clause, then a state $1^d\ 2\ 3^d\ 4\ 5^d\ 6\ \overline{7} \parallel F$ is reached, which would have been reached if one would have applied the Backtrack rule instead.

We have adopted a *backjump clause* construction technique based on resolution as mentioned in [15]. The construction of the backjump clause can be seen as a derivation in the resolution calculus. A backjump clause is constructed according to the *backwards conflict resolution* process [15].

In Figure 3.3 we provide pseudo code for the construction of a backjump clause. First, we record the literals of the current level. We obtain the list LP of all propagated literals of current level, but in reverse order. We also record a conflicting clause, and assign it to a backjump clause. We start resolving the conflicting clause with the clause which is responsible for the

```
Get the literals l_n M_n of the current level of the partial assignment.
    Make a list of propagated literals M_n of the current level literals l_n M_n,
    but in reverse order of their occurrences. We will call such a list LP.
    SET flag := true
    SET backjump clause := conflicting clause
    SET i := 1
    WHILE flag
        SET backjump clause := a clause obtained by resolving the backjump clause
        with a unit clause which was responsible for the propagation of the
        i^th literal in LP.
        IF there is only one literal of l_n M_n is in the backjump clause
            SET flag := false
        END-IF
        SET i := i + 1
    END-WHILE
```

Figure 3.3: Pseudo code of the construction of a backjump clause

occurrence of the the first propagated literal in LP. After resolution we get a clause C_1 and check if there is just one literal of the current level inside the clause C_1, if not then we do the resolution process between clause C_1 and the the clause which is responsible for the propagation of the second propagated literal in LP, and as a result we get a clause C_2. This process goes on until we get only one literal of the current level inside the clause, in such a case the obtained clause is the backjump clause.

Definition 3.1.5. *The* Learn *rule is shown below:*

$M \parallel F \quad \Longrightarrow M \parallel F, C$ *if*

- all atoms of C occur in F,
- $F \models C$.

After each application of the Backjump rule, one may learn the obtained *backjump clause*, i.e., add the *backjump clause* to the given formula in CNF in order to prevent occurrence of similar conflict again and again.

Definition 3.1.6 (Conflict-driven-lemma-learning)**.** *The backjump clause can be added to a formula in CNF as the learned clause, also called lemma. This technique is called* conflict-driven-lemma-learning.

Conflict-driven-lemma-learning is also called *conflict-driven-clause-learning* or *conflict-driven-learning*. Conflict-driven-lemma-learning has played a critical role in the success of the modern SAT solvers [9]. Learning a smaller lemma reduces more search space than learning a larger lemma.

We show how a backjump clause could be constructed using an example shown below:

Example 3.1.7 (Construction of a backjump clause)**.** *Let F be a formula in CNF as shown below:*

$$1 \vee 2,\ \overline{1} \vee 2,\ \overline{2} \vee 3,\ 4 \vee 5,\ 6 \vee 7,\ 6 \vee \overline{7},\ 8 \vee 9,\ \overline{8} \vee 9,\ \overline{3} \vee \overline{8} \vee \overline{9}$$

$\emptyset$	$\parallel$	$F \Longrightarrow$	(D)
1^d	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3\ 4^d$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3\ 4^d\ 5^d$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3\ 4^d\ 5^d\ 6^d$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3\ 4^d\ 5^d\ 6^d\ \overline{7}$	$\parallel$	$F \Longrightarrow$	(D)
$1^d\ 2\ 3\ 4^d\ 5^d\ 6^d\ \overline{7}\ 8^d$	$\parallel$	$F \Longrightarrow$	(UP)
$1^d\ 2\ 3\ 4^d\ 5^d\ 6^d\ \overline{7}\ 8^d\ 9$	$\parallel$	$F \Longrightarrow$	(BJ)
$1^d\ 2\ 3\ \overline{8}$	$\parallel$	F	

Figure 3.4: An example of the construction of a backjump clause

As shown in the subderivation given in Figure 3.4, after application of the 4^{th} UnitPropagate (UP) rule, we find a *conflicting clause* $\overline{3} \vee \overline{8} \vee \overline{9}$, i.e., the clause $\overline{3} \vee \overline{8} \vee \overline{9}$ is *false* in the partial assignment $1^d\ 2\ 3\ 4^d\ 5^d\ 6^d\ \overline{7}\ 8^d\ 9$.

In order to calculate a *backjump clause*, the ordered sequence of propagated literals together with the clauses which are responsible for the propagation, i.e., unit clauses are stored. For the subderivation shown in Figure 3.4, the propagated literals and the corresponding unit clauses are shown in Figure 3.5.

One can observe that there are five decision literals present in the partial assignment $1^d\ 2\ 3\ 4^d\ 5^d\ 6^d\ \overline{7}\ 8^d\ 9$ at the state before application of the BJ in the subderivation shown in Figure 3.4, and the clause $\overline{3} \vee \overline{8} \vee \overline{9}$ is conflicting clause. The conditions for application of the Backjump (BJ) are satisfied, so we can find a *backjump clause* and apply the Backjump rule.

propagated literal	unit clause
2	$\overline{1} \vee 2$
3	$\overline{2} \vee 3$
$\overline{7}$	$6 \vee \overline{7}$
9	$\overline{8} \vee 9$

Figure 3.5: Propagated literals and corresponding unit clauses for the subderivation given in Figure 3.4

In the following resolution process, at the first step the resolution process between *conflicting clause* $\overline{3} \vee \overline{8} \vee \overline{9}$ and a clause $\overline{8} \vee 9$, which is responsible for the last propagation (i.e., literal 9) is done, so we get the clause $\overline{3} \vee \overline{8}$ as a result, and in the clause $\overline{3} \vee \overline{8}$, there is only one literal (i.e., $\overline{8}$) of the literals of the current level (i.e., 8^d 9). Thus, the resolution process terminates and the clause $\overline{3} \vee \overline{8}$ is the *backjump clause*.

$$\dfrac{\dfrac{\overline{3} \vee \overline{8} \vee \overline{9} \qquad \overline{8} \vee 9}{\overline{3} \vee \overline{8}}}{}$$

Figure 3.6: Backwards conflict resolution

3.1.3 The DPLL(T) System

In this section we will discuss about the DPLL(T) [15] system.

Any DPLL(T) system has two components, i.e., a DPLL(X) engine, which is independent of any theory, and a theory (T) solver for theory T under consideration. The theory solver should be able to check consistency of conjunction of theory literals. To check satisfiability of theory T under consideration (a Boolean combination of literals in theory T) using DPLL(T), every literal in theory T will be treated as propositional literal by DPLL(X) engine. In the Satisfiability Modulo $\mathcal{ALCO}$ Boolean ABoxes, each $\mathcal{ALCO}$ ABox literal (i.e., $\mathcal{ALCO}$ ABox assertion or negated $\mathcal{ALCO}$ ABox assertion) occurring in $\mathcal{ALCO}$ Boolean ABoxes will be treated as propositional literal by DPLL(X) engine. The theory solver is used to check consistency of constructed (partial) assignment, and also informs about the consistency status of the (partial) assignment to DPLL(X) engine. The theory solver also computes an explanation when the (partial) assignment is inconsistent.

By negating each literal from the explanation, we can construct a T-conflicting clause of smaller or equal size than the size of T-conflicting clause obtained by negating each literal from the inconsistent partial assignment (ABox). Usually from a smaller T-conflicting clause, a smaller T-backjump clause can be constructed. We add the T-backjump clause to the CNF ABox as a learned lemma in order to prevent similar type of conflict again and again. Learning a smaller lemma can prune more search space than learning a large lemma [21]. It is widely understood that small explanations behave better in practice [14].

The DPLL(X) engine and the theory solver should be interconnected according a particular strategy.

In the following we give definitions of T-consistent and T-inconsistent partial assignment.

Definition 3.1.8 (T-consistent, T-inconsistent)**.** *A partial assignment M is called* T-consistent *if the conjunction of each element (literal in a Theory T) in M is consistent in theory T, otherwise M is called* T-inconsistent*.*

In the following we define logical consequence with respect to theory T under consideration.

Definition 3.1.9 (Logical consequence with respect to theory)**.** *Let F be a formula in theory T. A clause C is called a* logical consequence *of F* with respect to theory *T, if $\mathcal{I}$ is a model of F (according to the semantics of theory T), then $\mathcal{I}$ is a model of C (according to the semantics of theory T), can be written as $F \models_T C$.*

If the partial assignment is T-inconsistent, following three rules shown in the definition 3.1.10 could be employed.

Definition 3.1.10 (T-Backjump, T-Learn, T-Fail)**.** *The* $\mathsf{T-Backjump}$*,* $\mathsf{T-Learn}$ *and* $\mathsf{T-Fail}$ *rules are defined as shown below :*

$\mathsf{T-Backjump}$*:*

$M\ l^d\ N \parallel F, C \quad \Longrightarrow M\ l' \parallel F, C$ *if*

- $M\ l^d\ N \models \neg C$*, and*
- *there exists some clause $C' \vee l'$ such that : $F, C \models_T C' \vee l'$ and $M \models \neg C'$, and l' is undefined in M, and l' or $\neg l'$ occurs in F or in $M\ l^d\ N$.*

In the $\mathsf{T}-\mathsf{backjump}$ *rule given above, the clause* $C' \vee l'$ *is called* T-backjump clause.

$\mathsf{T}-\mathsf{Learn}$*:*

$M \parallel F \quad \Longrightarrow M \parallel F, C$ *if*

- *each atom of* C *occur in* F *or in* M*,*
- $F \models_T C$*.*

$\mathsf{T}-\mathsf{Fail}$*:*

$M \parallel F, C \Longrightarrow FailState$ *if :*

- $M \models_T \neg C$*,*
- M *contains no decision literals.*

The pseudo code to calculate a T-backjump clause is in Figure 3.8.

DPLL(T) solver communication strategy

Depending on how DPLL(X) engine communicates with T solver, there could be different variants of DPLL(T) solvers [14]. We implement a *DPLL(T) Solver* for Boolean $\mathcal{ALCO}$ ABoxes.

The pseudo code of our DPLL(T) Solver is in Figure 3.7, where rule application strategy of the DPLL(X) engine is same as that of the MiniSat [8].

We believe that our DPLL(T) Solver may save large amount of useless work in most of the cases. The DPLL(T) Solver employs the functionality mentioned below:

- The DPLL(X) engine communicates to theory solver whenever there is some change in the partial assignment M (i.e., due to the application of any of the $\mathsf{UnitPropagate}$ rule, Decide rule, $\mathsf{Backjump}$ rule, $\mathsf{T}-\mathsf{Backjump}$ rule), theory solver will answer whether M is T-consistent or not. If T-inconsistent, theory solver returns an explanation why inconsistency has been occurred, i.e., a minimal subset $\{l_1,\ldots,l_n\}$ of M that becomes T-inconsistent by the change of M. Then the DPLL(X) will handle
  $\neg l_1 \vee \ldots \vee \neg l_n$ as a *T-conflicting clause*, and will apply the $\mathsf{T}-\mathsf{Backjump}$ rule or the $\mathsf{T}-\mathsf{Fail}$ rule.

- After each application of the Backjump and the $\mathsf{T}-$Backjump rule, backjump clause and T-backjump clause is learned respectively.
- The DPLL(X) does exhaustive application of the UnitPropagate rule and gives the UnitPropagate rule higher priority than the Decide rule.
- The DPLL(T) algorithm terminates when a T-consistent model is found or there is a top-level conflict, i.e., there exist a conflicting clause (or T-conflicting clause) and there is no decision literal in the partial assignment.

3.2 The Reduction Approach

In this section we will discuss about checking consistency of $\mathcal{ALCO}$ Boolean ABoxes using the reduction based approach. A Boolean ABox is reduced to a satisfiability preserving knowledge base (possibly in a more expressive DL $\mathcal{ALCHOQ}$ [1]).

3.2.1 The Reduction Algorithm

In order to reduce an $\mathcal{ALCO}$ Boolean ABox to a satisfiability preserving knowledge base, we convert the $\mathcal{ALCO}$ Boolean ABox into an $\mathcal{ALCO}$ CNF ABox. The reduction approach aims to convert the $\mathcal{ALCO}$ CNF ABox into unit-positive-clauses, possibly with role inclusions by reductions. We can extend the reduction based algorithm [1] to check consistency of $\mathcal{ALCO}$ Boolean ABox from $\mathcal{ALC}$ Boolean ABox trivially because the nominal and its negation can be treated analogously by the step 2 of the reduction algorithm given below, consequently a satisfiability preserving knowledge base in a more expressive DL $\mathcal{ALCHOQ}$ could be obtained. The following is the reduction algorithm for transforming an $\mathcal{ALCO}$ CNF ABox into a satisfiability preserving knowledge base.

Input: An $\mathcal{ALCO}$ CNF ABox $\mathcal{A}$.

1. If $r(a, b)$ occurs negatively or in non unit-positive-clauses, then uniformly replace $r(a, b)$ in $\mathcal{A}$ by $(\exists r.IamB)$, and add a role inclusion and 3 ABox assertions given below to the knowledge base (using new symbols each time):

- $r \sqsubseteq Fake\_r$
- $Fake\_r(a, b)$
- $IamB(b)$
- $(\leq 1\ Fake\_r\ IamB)(a)$

Intermediate result: A Boolean ABox (possibly together with TBox $\mathcal{T}'$).

2. Push negations inside using the validity $\neg\ ((C(a)) \equiv (\neg C)(a)$.

   Let $C(a)$ is a concept assertion al, using $\neg\ ((C(a))$ we want to represent negated concept assertion, i.e., $\neg al$.

   Intermediate result: A Boolean ABox without negative ABox literals (possibly together with TBox $\mathcal{T}'$).

3. Collect disjunctions concerning the same individual names using the validity $C(a) \vee D(a) \equiv (C \sqcup D)(a)$.

   Intermediate result: A Boolean ABox in which every ABox clause contains at most one ABox literal $C(a)$ for each individual name a (possibly together with TBox $\mathcal{T}'$).

4. Repeatedly transform occurrences of disjunctions of the form $C(a) \vee D(b)$ in any of the ABox clause by $(C \sqcup \forall r.D)(a)$, where r is new role name.

Output: A knowledge base possibly in a more expressive DL $\mathcal{ALCHOQ}$.

After reduction of an $\mathcal{ALCO}$ Boolean ABox is done to a satisfiability preserving knowledge base (KB), one can check consistency of the KB using a Description Logic reasoner. If Description Logic reasoner reports that the KB is consistent, then the $\mathcal{ALCO}$ Boolean ABox under consideration is consistent, otherwise not.

For more information about the reduction approach, reader is directed to [1].

```
WHILE
  WHILE unit propagate rule is applicable
    Apply unit propagate rule
    SET con-status by the output of pseudo-code given in Figure 4.5
    IF not con-status
      RETURN inconsistent Boolean ABox
    END-IF
  ENDWHILE
  IF not conflict (at propositional level)
   IF all the variables (ABox assertions) are assigned
    RETURN consistent Boolean ABox
   ELSE
    Apply decide rule
    SET con-status by the output of pseudo-code given in Figure 4.5
    IF not con-status
      RETURN inconsistent Boolean ABox
    END-IF
   END-IF
  ELSE
   IF top-level conflict
    RETURN inconsistent Boolean ABox
   ELSE
    Apply backjump rule
    SET con-status by the output of pseudo-code given in Figure 4.5
    IF not con-status
      RETURN inconsistent Boolean ABox
    END-IF
   END-IF
  END-IF
ENDWHILE
```

Figure 3.7: Pseudo code of the DPLL(T) Solver

```
Get the literals l_n M_n of the current level of the partial assignment.
Let x := |M_n|
IF x == 0
   SET T-backjump clause := T-conflicting clause
ELSE
   Make a list of propagated literals M_n of the current level literals l_n M_n,
   but in reverse order of their occurrences. We will call such a list LP.
   SET flag := true
   SET T-backjump clause := T-conflicting clause
   SET i := 1
   WHILE flag
      SET T-backjump clause := a clause obtained by resolving the
      T-backjump clause with a unit clause which was responsible for
      the propagation of the i^th literal in LP.
      IF there is only one literal of l_n M_n is in the T-backjump clause
         SET flag := false
      END-IF
      SET i := i + 1
   END-WHILE
END-IF
```

Figure 3.8: Pseudo code of the construction of a T-backjump clause

Chapter 4

Implementation Details

In this chapter we will tell about some important implementation details for the DPLL(T) based approach as well as the reduction based approach. We also give information about KRSS syntax [18].

The Pellet 1.5.1 [20] is selected as the DL reasoner (theory solver/T-solver) because:

- Pellet computes an *explanation* [17] in case of inconsistent ABox. Importance of the explanation is explained in section 3.1.3. It is very useful for the DPLL(T) approach.
- Pellet is an open source OWL DL[1] reasoner.
- In the reduction based approach to check consistency on $\mathcal{ALCO}$ Boolean ABoxes, after reduction of an $\mathcal{ALCO}$ Boolean ABox, we may get a knowledge base in a more expressive DL $\mathcal{ALCHOQ}$. Pellet can also check consistency of DL $\mathcal{ALCHOQ}$.
- Pellet 1.5.1 supports KRSS syntax[2] [18].

Example 4.0.1 shows that Pellet 1.5.1 computes just one explanation, but there might be more than one explanations.

Example 4.0.1. *Figure 4.1 shows an (inconsistent) ABox. Load this ABox (keeping the order of the assertions as it is) to Pellet 1.5.1, which will compute an explanation (having 3 ABox assertions) as shown in Figure 4.2. There could be other explanation (having 2 ABox assertions) as shown in Figure 4.3.*

[1] see, http://www.w3.org/TR/owl-features/

[2] At least for $\mathcal{ALCO}$ ABox assertions, role hierarchies, and qualified number restrictions, which are relevant for this thesis.

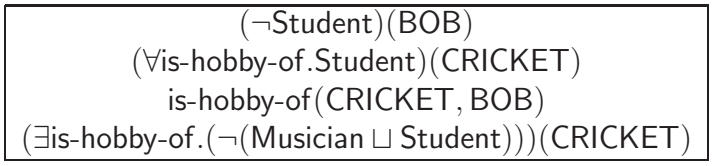

Figure 4.1: An (inconsistent) ABox

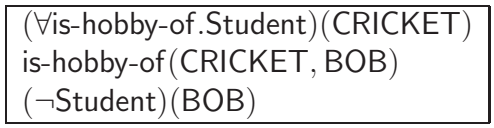

Figure 4.2: Explanation given by Pellet 1.5.1 for the ABox shown in Figure 4.1

One can convert an arbitrary Boolean ABox into CNF ABox. Both of the implementations (i.e., DPLL(T) approach and reduction approach) assume that one has converted a Boolean ABox into a CNF ABox.

Figure 4.4 shows how the concept constructors in $\mathcal{ALCO}$ are represented in KRSS syntax [18], Pellet 1.5.1 accepts KRSS syntax for $\mathcal{ALCO}$ concept descriptions. The *role inclusion* $r_1 \sqsubseteq r_2$ can be represented as (DEFINE-PRIMITIVE-ROLE r_1 r_2), and the *qualified number restriction* $(\leq n\ r\ C)$ can be represented as (AT-MOST n r C) in KRSS syntax, Pellet 1.5.1 recognizes this as well. To represent a concept assertion and a role assertion, Pellet uses the same syntax as of KRSS syntax given in [18].

The concept assertion $C(a)$ can be represented as shown below:

(INSTANCE A C)

The role assertion $r(a, b)$ would be represented as shown below:

(RELATED A B R)

Our testing data, CNF ABoxes are stored in text files. We used the follwing syntax to represent CNF ABoxes:

CNF ABOX := $(AC_1\ AC_2 \ldots AC_n)$
$AC_i := (al_1\ al_2 \ldots al_m)$
$al_i := as \mid (\text{NOT } as)$
Here as is arbitrary $\mathcal{ALCO}$ ABox assertion represented in KRSS syntax.

$(\forall \mathsf{is\text{-}hobby\text{-}of}.\mathsf{Student})(\mathsf{CRICKET})$
$(\exists \mathsf{is\text{-}hobby\text{-}of}.(\neg(\mathsf{Musician} \sqcup \mathsf{Student})))(\mathsf{CRICKET})$

Figure 4.3: An explanation for the ABox shown in Figure 4.1

Concept	**KRSS**
$\top$	TOP
$\bot$	BOTTOM
$\neg C$	(NOT C)
$C_1 \sqcap \ldots \sqcap C_n$	(AND C_1 ... C_n)
$C_1 \sqcup \ldots \sqcup C_n$	(OR C_1 ... C_n)
$\exists r.C$	(SOME r C)
$\forall r.C$	(ALL r C)
{a}	(ONE-OF a)

Figure 4.4: KRSS syntax of $\mathcal{ALCO}$ concept constructors

We have implemented a Java class called *CNF_parsing* which is used in both implementations - the reduction based approach and the DPLL(T) based approach.

In the class *CNF_parsing* there is a method *get_theory_CNF*, which takes a text file having a CNF ABox as an argument and parses the file. Method *get_theory_CNF* returns a list of ABox clauses, where each ABox clause is a list of ABox literals.

We have tested the reduction based approach as well as the DPLL(T) approach on the testing data stemming from the ABox update [11] problem. We have adopted unique name assumption (UNA) in both of the implementations as that is the requirement for the ABox update problem.

4.1 The DPLL(T) Approach

In this section we provide implementation details about the DPLL(T) approach to check consistency of $\mathcal{ALCO}$ Boolean ABoxes.

To represent the model elements we construct a *Model_element* class which has two fields, one is of type *String* which is used to represent content (i.e. the ABox literal) of the model element, and the other is of type *boolean* which is used to mark whether the model element (i.e., the ABox literal) is decision literal or not. If the literal is decision literal then the *boolean* field

will be set to *true*, otherwise it will be set to *false*. For each literal to be put into the partial assignment, we will create an object of type *Model_element*. A list of *Model_element* is used to represent the (partial) assignment.

Important data structures used in the DPLL(T) based implementation are shown below. We also give a brief description about each of them:

- *cnf*
  cnf is used to store each ABox clause.

- *model*
  model is used to store the model elements (i.e., the literals to be collected in a partial assignment).

- *propagate_literal_list*
  propagate_literal_list is used to store literals which are occurring in the *model*, except the literals introduced by the Decide rule.

- *reason_map*
  reason_map is used to store all the literals except the decision literals together with corresponding unit clauses. The *reason_map* is used in construction of a *backjump clause*.

- *decide_literal_list*
  decide_literal_list is a list of the literals introduced by applications of the Decide rule.

- *positive_literals_set*
  positive_literals_set is a set of ABox assertions having only even number of $\neg$ in front of concept description.

- *negative_literals_set*
  negative_literals_set is a set of concept assertions having odd number of $\neg$ in front of concept description.

- *atoms_set*
  atoms_set is union of *positive_literals_set* and only those ABox literals from *negative_literals_set*, whose negation does not occur in *positive_literals_set*. When size of the *model* becomes equal to size of the *atoms_set*, it is an indication that the model has been found in the DPLL(T) implementation.

In the DPLL(T) approach, the negative $\mathcal{ALCO}$ ABox literals occurring in the CNF ABox should be transformed to equivalent positive $\mathcal{ALCO}$ ABox

literals, as the DL reasoner Pellet 1.5.1 can not check consistency of negative $\mathcal{ALCO}$ ABox literals, but only of positive $\mathcal{ALCO}$ ABox literals (i.e., an ABox).

In the following we explain how we deal with negative $\mathcal{ALCO}$ ABox literals:

- The negative $\mathcal{ALCO}$ ABox literals of the form $\neg al$, where al is an arbitrary $\mathcal{ALCO}$ concept assertion $C(a)$. The $\neg al$ can be represented as $\neg(C(a))$. The $\neg(C(a))$ could be converted into equivalent positive $\mathcal{ALCO}$ ABox literal $(\neg C)(a)$.

- The negative $\mathcal{ALCO}$ ABox literals of the form $\neg al$, where al is an arbitrary $\mathcal{ALCO}$ role assertion $r(a, b)$. The $\neg al$ can be represented as $\neg r(a, b)$. The $\neg r(a, b)$ could be converted into equivalent positive $\mathcal{ALCO}$ ABox literal $(\neg(\exists r.\{b\}))(a)$.

We replace every negative $\mathcal{ALCO}$ ABox literals occurring in $\mathcal{ALCO}$ CNF ABoxes by equivalent positive $\mathcal{ALCO}$ ABox literals. After such replacement $\mathcal{ALCO}$ CNF ABoxes has only positive $\mathcal{ALCO}$ ABox literals.

Now we will show the details about some of the methods in the DPLL(T) approach:

- Related to an ABox generation in KRSS.

The method *generate_KRSSfile_from_whole_partialmodel* is implemented in order to put the content of a (partial) assignment into a separate file. We can load the file to Pellet reasoner and check the consistency of the ABox.

- Related to the UnitPropagate rule.

The method *is_unitpropagate_applicable()* is used to check whether the UnitPropagate rule is applicable or not, if it is applicable (i.e., just in case when there exist a clause $C' \vee l'$, where literal l' is undefined in the partial assignment and C' is false in the partial assignment), then the literal l' will be added to *propagate_literal_list*. We also put pair $(l', C' \vee l')$ in the *reason_map* because *reason_map* will be useful in construction of a *backjump clause* later on.

Once we know that the UnitPropagate rule is applicable, then in order to apply the UnitPropagate rule, i.e., to add a model element (i.e., an ABox literal) to the partial assignment, we have implemented a method *unit_propagate()*. This method takes the most recently added literal from the *propagate_literal_list* and creates a model element from it and adds the model element to the partial assignment.

- Related to the Decide rule.

To check if the Decide rule is applicable or not a method *is_decide_applicable()* is used. If some literal l occurring in any of the ABox clauses in the CNF ABox is undefined in partial assignment, then one can guess *true* or *false* value for l, but in our implementation we always guess (decide) *true* value for the undefined literal l, and the literal l will be added to the *decide_literal_list*.

To apply the Decide rule, method *decide()* is created. The most recently added literal from the *decide_literal_list* will be chosen, and a new *Model_element* will be created by setting *true* status for the *decision_var_or_not* field of the *Model_element*. The model element will be added to the partial assignment.

- Related to the Backjump rule, T − Backjump rule, Learn rule, and T − Learn rule.

Calculation of a backjump clause: To construct a *backjump clause* we use a method *get_backjump_clause()* based on the pseudo code given in Figure 3.3.

Calculation of a T-backjump clause: One should note that calculation of a *T-backjump clause* is a bit different from the calculation of *backjump clause*. A method *get_T_backjump_clause* is used to get a *T-backjump clause* which is based on the pseudo-code given in Figure 3.8. Once we found DL-inconsistency on the model elements (the ABox) after calling Pellet 1.5.1, we also get an *explanation*, the ABox assertion(s) which is (are) responsible for generating inconsistency. We can construct T-conflicting clause by negating every literals occurring in the explanation.

The application of the Backjump rule and the Learn rule are implemented togethere in a method *backjump()*. The implementation of the T − Backjump rule as well as T − Learn rule are done using a method *backjump_T_side()*.

- Related to the Fail and T − Fail rule.

We did not implement a method for the Fail rule. Once we know that there exist a conflicting clause and there is not any decision literal present in the partial assignment (i.e., size of the *decide_literal_list* is equal to zero), then the Fail rule can be applied, once conditions for application of the Fail rule are met, we immediately come out of DPLL(T) algorithm and return "INCONSISTENT Boolean ABox". The logic behind the T − Fail rule is the same as that of the Fail rule except in this case we have T-conflict.

- Related to *negating an assertion.*

A method *negate_Assertion* is used to negate a concept assertion. If the concept assertion is of the form C(a), then $(\neg\ C)(a)$ will be returned, and if the concept assertion is of the form $(\neg\ C)(a)$ then $C(a)$ will be returned. Method *negate_Assertion* is used whenever we need to negate an ABox literal in the DPLL(T) approach, as well as in the reduction approach.

- Related to converting the explanation given by Pellet 1.5.1 to KRSS syntax.

Pellet 1.5.1 computes the explanation in case of the inconsistent ABoxes, but the explanation returned by Pellet is not in KRSS form. Thus, in order to recognize which ABox assertion(s) of the inconsistent ABox are in the explanation, we convert the explanation into KRSS form by a method *convert_pellet_explanation_to_KRSS.*

- For *calling a DL reasoner (i.e. Pellet 1.5.1).*

multiple_consistency_calls_backjump_approach_from_T_side method is very important in the implementation of the DPLL(T) approach. The pseudo code of the method is given in Figure 4.5. The task of the method is to check consistency of the (partial) assignment after each application of the UnitPropagate, Decide, Backjump and $\mathsf{T}-$Backjump rule. It also applies the $\mathsf{T}-$Backjump rule and the $\mathsf{T}-$Learn rule exhaustively until a T-consistent partial assignment is reached or a T-inconsistent partial assignment having not any decision literal is reached.

4.2 The Reduction Approach

In this section we give implementation details about the reduction approach [1] to check consistency of the $\mathcal{ALCO}$ Boolean ABoxes.

An $\mathcal{ALCO}$ CNF ABox written in a text file is parsed using a method *get_theory_CNF*, a list of ABox clauses is returned. Methods *rule1*, *rule2*, *rule3*, *rule4* and *post_processing* are called respectively. Thereafter, we get a KB (TBox $\mathcal{T}'$, ABox $\mathcal{A}'$) written in KRSS syntax in a single text file (ReductionApproachoutput_KB.txt). The ReductionApproachoutput_KB.txt file is loaded to Pellet 1.5.1 reasoner to check consistency. If Pellet 1.5.1 reasoner reports that the KB is consistent, then the Boolean ABox is consistent, otherwise not.

```
SET status := DL-consistency of the partial assignment elements (the ABox)
IF status
    RETURN consistent
ELSE
 WHILE not-status AND exists at least one
                          decision literal in the partial assignment.
     get T-backjump clause
     apply T − Backjump rule
     apply T − Learn rule
    SET status := DL-consistency of the partial assignment elements
    IF status
       RETURN consistent
    END-IF
 ENDWHILE
END-IF
RETURN not-consistent
```

Figure 4.5: multiple_consistency_calls_backjump_approach_from_T_side - pseudo code

Chapter 5

Testing Data, Comparison and Analysis

In this chapter, first we give information about the testing environment, and the testing data. Then we compare and analyze the performance of the both algorithms on 6000 Boolean ABoxes stemming from ABox update [11] problem. We also tell about a strategy to choose a positive literal or a negative literal as a decision literal in the Decide rule, which may have considerable impact on the performance of the DPLL(T) approach.

5.1 Testing Environment Details

Our testing experiments were conducted on a notebook (TOSHIBA Satellite M40X) having Intel(R) Pentium(R) M processor 1.73GHz CPU, with 512 MB of system memory, maximum heap size of 1024 MB. The operating system was Ubuntu 7.04 (Linux).

5.2 The Testing Data

In this section we provide information about the testing data used to test both of the approaches.

6000 Boolean ABoxes were taken to test both of the algorithms to check consistency of Boolean ABoxes, compare and analyze the results. The testing data varies depending on number of symbols, number of ABox assertions, and number of ABox clauses in Boolean ABoxes. Some of the details about testing data are shown in Figure 5.2.

Number of symbols occurring in a Boolean ABox is counted as shown below :

Property	Value range
Size of file	172 Bytes to 3 MB
Number of symbols	24 to 216362
Number of assertions	7 to 1257
Number of clauses	5 to 267

Figure 5.1: Testing data details

- 1 for each occurrence of a concept name, a role name, and an individual name,
- 1 for each occurrence of a concept constructor,
- 1 for each occurrence of an ABox assertion.

Figure 5.2 shows a Boolean ABox written in KRSS syntax.

```
(((INSTANCE A STUDENT) (INSTANCE A (NOT STUDENT)))
((RELATED A B FRIEND-OF))
((INSTANCE B (AND DOCTOR HONEST)))
((INSTANCE D (OR ENGINEER HONEST))))
```

Figure 5.2: A Boolean ABox written in KRSS syntax

Number of symbols in the Boolean ABox given in Figure 5.2 is 21.

5.3 Comparison and Analysis

In this section we compare, and analyze the results obtained by testing of the both approaches on available testing data.

Out of 6000 Boolean ABoxes, we found 610 inconsistent Boolean ABoxes after running both of the algorithms on the testing data. Figure 5.3 shows the minimum and maximum runtime (in milliseconds) taken by each algorithm for inconsistent Boolean ABoxes.

There are total 5390 consistent Boolean ABoxes in the testing data. Figure 5.4 shows the minimum and maximum runtime (in milliseconds) taken by each approach in case of consistent Boolean ABoxes.

There are three different graphs drawn below to show the performance of the reduction based approach as well as of the DPLL(T) based approach. The x-axis of all the graphs indicates the total number of symbols in the

	Reduction Approach	**DPLL(T) Approach**
Minimum Time (ms)	3	1
Maximum Time (ms)	72089	497789

Figure 5.3: Minimum and Maximum runtime taken for inconsistent Boolean ABoxes

	Reduction Approach	**DPLL(T) Approach**
Minimum Time (ms)	1	2
Maximum Time (ms)	120126	1152681

Figure 5.4: Minimum and Maximum runtime taken for consistent Boolean ABoxes

Boolean ABox, and the y-axis represents the total time (in milliseconds) taken starting from reading a Boolean ABox to know whether the Boolean ABox is consistent or not. The first graph shown in Figure 5.5 is for all the Boolean ABoxes together (i.e. consistent and inconsistent Boolean ABoxes), the graph shown in Figure 5.6 is for only consistent Boolean ABoxes, and the graph shown in Figure 5.7 is for inconsistent Boolean ABoxes. Green line shows the performance of the implementation based on the reduction approach, and the red line shows the performance of the DPLL(T) based approach.

We analyzed performance of the both approaches and found that in case of consistent Boolean ABoxes, the reduction based approach does better than the DPLL(T) based approach, i.e., in most of the cases the reduction based approach takes less runtime compare to the DPLL(T) based approach. We can observe from Figure 5.8 that the reduction based approach does better than DPLL(T) based approach 3358 times, but the DPLL(T) approach does better than the reduction approach only 1908 times. On 124 cases, both of the approaches take same runtime. Thus, approximately 2:1 cases the performance of the reduction approach is better than the DPLL(T) approach for consistent Boolean ABoxes.

Figure 5.9 shows that the DPLL(T) based approach behaves better than the reduction based approach in case of inconsistent Boolean ABoxes. Out of 610 inconsistent Boolean ABoxes, on 449 cases the DPLL(T) behaves better than the reduction based approach, but the reduction based approach behaves better than the DPLL(T) based approach on 149 cases. On 12 cases both the approaches take same runtime to give answers. Thus, approxi-

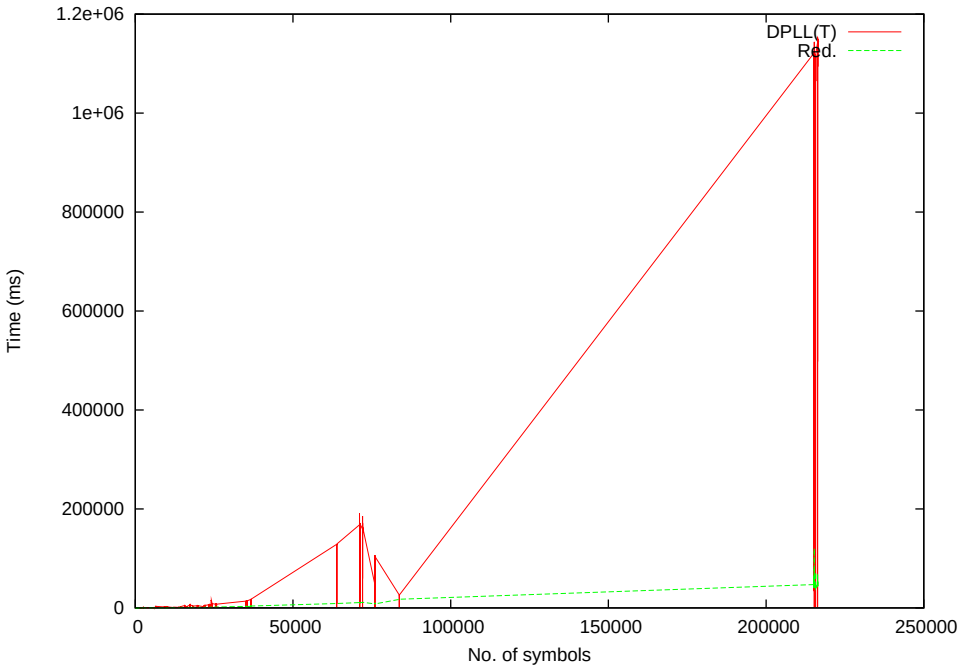

Figure 5.5: The performance of the reduction approach vs the DPLL(T) approach for 6000 Boolean ABoxes

mately 3:1 cases the performance of the DPLL(T) approach is better than the reduction approach for inconsistent Boolean ABoxes.

After running the DPLL(T) approach on 6000 testing files, we found that maximum level backjumped by the $\mathsf{T}-\mathsf{Backjump}$ rule was 2.

The Effect of the Decision Literal Selection

The selection of a decision literal, i.e., a positive literal or a negative literal during the Decide rule has considerable impact on the performance of the DPLL(T) approach on the testing data. *Positive decision literal selection* (*PDLS*) means if the Decide rule is applicable, and the literal l is an undefined literal to be decided then l will be put into the partial assignment. *Negative decision literal selection* (*NDLS*) means if the Decide rule is applicable, and the literal l is an undefined literal to be decided then $\neg l$ will be put into the partial assignment. The graph in Figure 5.10 shows the performance of PDLS vs NDLS for the DPLL(T) approach. After analyzing the 6000 results, we found that PDLS takes less time than NDLS on 4385 testing data, NDLS takes less time than PDLS on 1367 testing data, and PDLS and

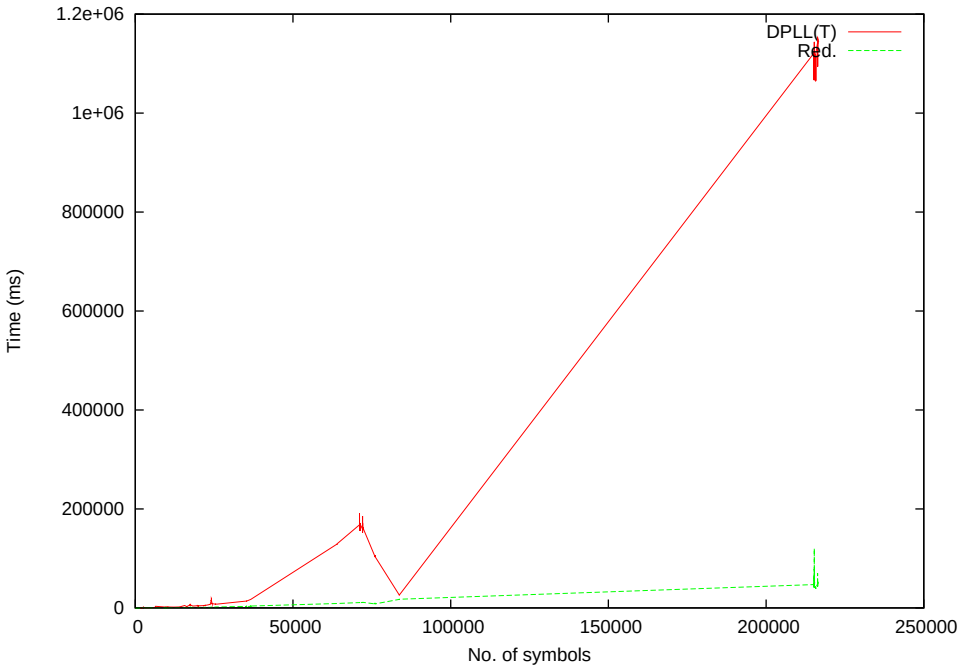

Figure 5.6: The performance of the reduction approach vs the DPLL(T) approach for consistent Boolean ABoxes

NDLS take same time for 248 testing data. Times are in milliseconds. The performance difference is due to number of rule applications needed to solve the SMT problem by choosing PDLS and NDLS are diffrent.

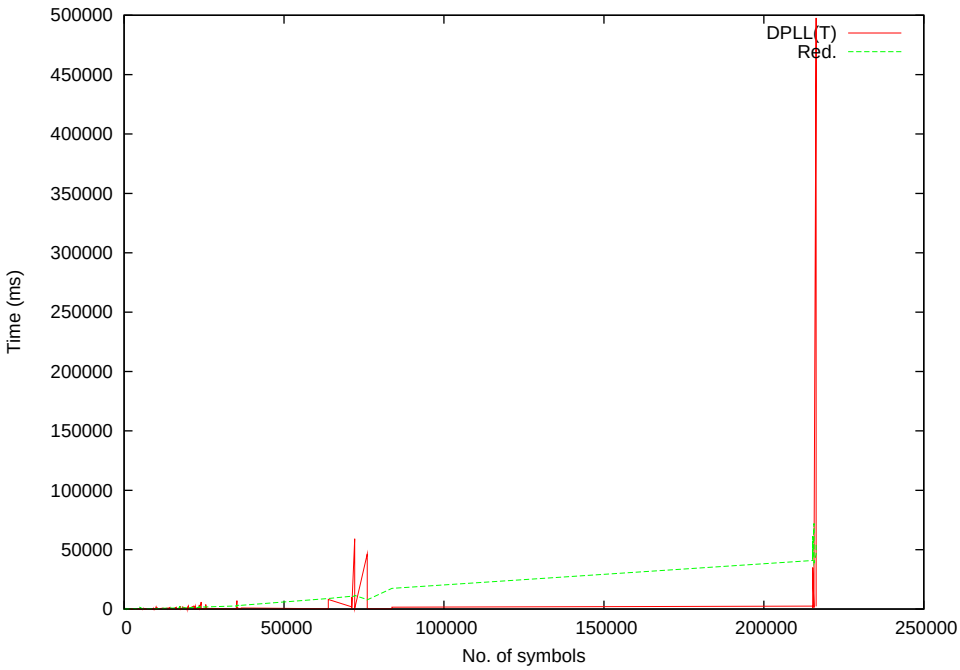

Figure 5.7: The performance of the reduction approach vs the DPLL(T) approach for inconsistent Boolean ABoxes

DPLL(T) < red.	**red. < than DPLL(T)**	**red. == DPLL(T)**
1908	3358	124

Figure 5.8: The performance of the both approaches respect to each other for consistent Boolean ABoxes

DPLL(T) < red.	**red. < than DPLL(T)**	**red. == DPLL(T)**
449	149	12

Figure 5.9: The performance of the both approaches respect to each other for inconsistent Boolean ABoxes

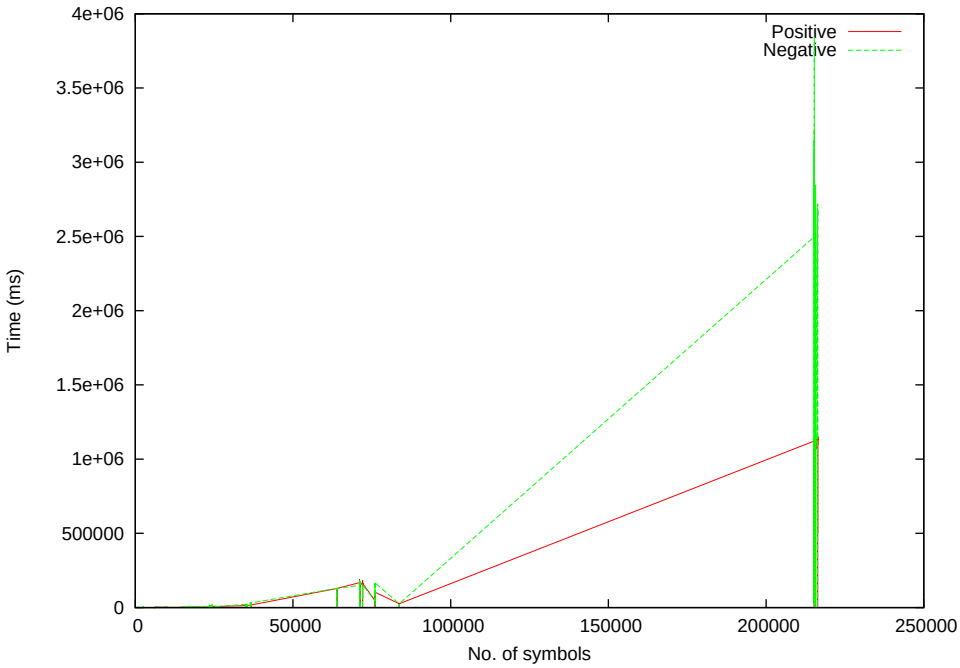

Figure 5.10: The effect of positive decision literal selection vs negative decision literal selection on the DPLL(T) approach

Chapter 6

Conclusions, Related and Future Research

In this thesis we have created and implemented a reasoning service for checking consistency of $\mathcal{ALCO}$ Boolean ABoxes based on the DPLL(T) approach of the Satisfiability Modulo Theories (SMT). We have extended an existing reduction based approach to check consistency of $\mathcal{ALC}$ Boolean ABoxes to $\mathcal{ALCO}$ Boolean ABoxes. We also implemented the reduction algorithm to check consistency of $\mathcal{ALCO}$ Boolean ABoxes. Analyzing the performance of the both implementations on the available testing data from the ABox update problem, we could not reach a conclusion on a single side. The data we have tested shows that the reduction based approach performs better than the DPLL(T) based approach on consistent Boolean ABoxes, but the performance of the DPLL(T) based approach is better than the reduction based approach on inconsistent Boolean ABoxes.

If provided robust *explanation* and *incremental reasoning*[1] by Description Logic reasoners, the DPLL(T) based approach could be optimized further.

Futher research in the direction of less expressive and more expressive DLs than $\mathcal{ALCO}$ could be done as well. It will be interesting to check the consistency of Boolean TBoxes, Boolean knowledge bases using the DPLL(T) approach. The *explanation* reasoning service could be extended to Boolean ABoxes, in this case knowledge engineers may expect to know which ABox clauses are responsible to make the Boolean ABox inconsistent. The further research could also be done in the direction to check the impact of the various (communication) strategies for DPLL(T) based approach, i.e., to call DL reasoner to check consistency of the (partial) assignment/ABox only when a new decision has been made by DPLL based SAT solver. It would be

[1]For incremental reasoning see, [16]

interesting to analyze the effect of *TheoryPropagate*[2] rule in the DPLL(T) based approach.

[2]For more information about TheoryPropagate rule, look at [15]

Bibliography

[1] C. Areces, P. Blackburn, B. Martinez Hernandez, and M. Marx. Handling boolean aboxes. In D. Calvanese, G. De Giacomo, and E. Franconi, editors, *Proceedings of the 2003 International Workshop on Description Logics (DL2003)*, volume 81 of *CEUR - Workshop Proceedings*, Rome, Italy, September 2003.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge, 2003.

[3] F. Baader, C. Lutz, M. Milicic, U. Sattler, and F. Wolter. A description logic based approach to reasoning about web services. In *Proceedings of the WWW 2005 Workshop on Web Service Semantics (WSS2005)*, Chiba City, Japan, 2005.

[4] Franz Baader. Preface to special issue on reasoning in description logics. *J. Autom. Reasoning*, 39(3):245–247, 2007.

[5] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 228–248. Springer, 2005.

[6] Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, San Mateo, CA, 1985.

[7] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[8] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[9] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2007.

[10] P. J. Hayes. The logic of frames. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter and Co., Berlin, Germany, 1979.

[11] H. Liu, C. Lutz, M. Milicic, and F. Wolter. Updating description logic ABoxes. In Patrick Doherty, John Mylopoulos, and Christopher Welty, editors, *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 46–56. AAAI Press, 2006.

[12] M. Minsky. A framework for representing knowledge. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 245–262. Kaufmann, Los Altos, CA, 1985.

[13] Daniele Nardi and Ronald J. Brachman. An introduction to description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook. Theory, Implementation and Applications*, pages 1–40. Cambridge University Press, 2003.

[14] Robert Nieuwenhuis, Albert Oliveras, Enric Rodrguez-Carbonell, and Albert Rubio. Challenges in satisfiability modulo theories. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[15] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, nov 2006.

[16] Bijan Parsia, Christian Halaschek-wiener, and Evren Sirin. E.s.: Towards incremental reasoning through updates. In *in OWL DL. In: Proc. WWW-2006.*, 2006.

[17] Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging owl ontologies. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 633–640, New York, NY, USA, 2005. ACM.

[18] Peter F. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification, NOV 1993.

[19] Ross Quillian. Semantic memory. In *Semantic Information Processing*, pages 216–270. MIT Press, 1968.

[20] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[21] Yinlei Yu and Sharad Malik. Lemma learning in smt on linear constraints. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 2006.