DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

University of Innsbruck
Digital Enterprise Research Institute

BDD-Tableaux Calculus for DL Reasoning: Implementation and Visualization

Bachelor Thesis

Markus Ruepp
am Steinbruch 6
A-6600 Reutte
Matrikelnr.: 0417280

Supervised by Univ. Dr. Elena Simperl
and co-supervised by Uwe Keller

Innsbruck, May 26, 2008

Abstract

Desription Logic (DL) is current a prominent tool to represent knowledge. DL-based languages like $\mathcal{ALC}$ can be used to model fragments of the real world in a machine processable, formal way. DLs provide ways to check the modelled information for validity and satisfiability, for example.
A popular used way to check satisfiablity of a $\mathcal{ALC}$ concept is the tableaux algorithm. Based on a set of socalled completion rules, tableaux builds a tree like model. In this bachelor thesis this popular algorithm has been modified, in a way extended through Binary Decision Diagrams(BDDs). A Bdd is a structure to represent boolean functions, which are part of (almost) every $\mathcal{ALC}$ expression. The resulting algorithm called *Bdd tableau* only has to reason about modal operators, all propositional logic parts are handeled by the bdds. The implementation of the *bdd tableaux* algorithm can be used to perform proofs of $\mathcal{ALC}$ expressions step-by-step in an interactive manner.
The implementation shall help to examine the classic tableaux and the bdd tableaux and understand its behaviour.

Contents

Chapter 1

Introduction

In this part of the bachelor thesis, we give you an overview about the topic and introduce some important aspects of the thesis.

1.1 Description Logics $\mathcal{ALC}$

In this thesis, a major focus is on the concept of a *DL Reasoner*. This word contains two important terms: **DL**, which is the abbreviation for Description Logic, and **Reasoner**.

1.1.1 Introduction

Description Logic(DL) is a common knowledge representation (KR) formalism, which represents knowledge of a domain(the world). A KR consists of concepts of this domain, and properties for objects and individuals, which are specified trough those concepts. Description logics are based on a formal semantics, Moreover all represented knowledge is unambiguous. The knowledge expressed though dls is processable and understandable by machines
The strong emphasis on knowledge in DLs leads us to the support of many inference patterns, that allow you to infer knowledge from knowledge. That means we can structure concepts in a hyrarchical order (subconcept/superconcept), a socalled *Classification*. This allows you to determine infomation about the connection between various concepts, which can be used to speed up inference. The frist realization of a decription logic was KL-ONE[Brachman and Scholze, 1985]. This first approach was taken to overcome ambiguties in semantic frames and networks. Brachman also stated the three basic ideas of a DL-like system, which are:

- Elementary atomic objects are atomic concepts, atomic roles and individuals
- Complex concept can be build by a few concept constructors, in a recursive manner

- Additional knowledge about concepts and individuals affect relationships between those objects and their properties. (TBox, ABox)

A DL-based knowledge representation supports possibilities to set up new knowledge bases, reason about its content, and manipulate the content.

It turned out, that the first DL-languages have been to expressive, which caused an undecideability for subsumption problems. Generally speacking we can state:

> The higher the expressiveness a DL-language offers, the higher is its complexity.

Later it has been shown that subsumption problems are always hard to solve, even if the chosen DL-system is very inexpressive.

In the following sections we will take a closer look to the description language $\mathcal{ALC}$, (=attributive language). $\mathcal{ALC}$ is part of the DL-family $\mathcal{AL}$, which has been introduced as the minimal language of practical use.

1.1.2 $\mathcal{ALC}$ Syntax and Semantics

The description language underlaying $\mathcal{ALC}$ is formed according to the following syntactical rules. $\mathcal{ALC}$ extends $\mathcal{AL}$, a language that does not support a conjunction directly. Moreover in $\mathcal{AL}$ the negation can only be applied to atomic concepts and the existential quantifier only supports the top concept ($\exists R.\top$).

C,D	$\rightarrow$	$A\|$	(atomic concept)
		$\top\|$	(top concept)
		$\bot\|$	(bottom concept)
		$\neg A\|$	(negation)
		$A \sqcap B\|$	(intersection)
		$A \sqcup B\|$	(conjunction)
		$\exists R.A\|$	(limited existential quantifier)
		$\forall R.A$	(value restriction)

The set of operators to create atomic and complex concepts are called constructors. The following paragraph shows the semantical interpretation of $\mathcal{ALC}$ concepts.

$\top$	$\rightarrow$	$(\top)^I$ (every legal instance is a member)
$\bot$	$\rightarrow$	$(\bot)^I = \{\}$
$\neg C$	$\rightarrow$	$(\neg C)^I = \Delta \backslash C^I$
$C \sqcap D$	$\rightarrow$	$(C \sqcap D)^I = C^I \cap D^I$
$C \sqcup D$	$\rightarrow$	$(C \sqcup D)^I = C^I \cup D^I$
$\exists R.C$	$\rightarrow$	$(\exists R.C)^I = \{x \mid (x,y) \in R^I \wedge y \in C^I\}$
$\forall R.C$	$\rightarrow$	$(\forall R.C)^I = \{x \mid (x,y) \in R^I \rightarrow y \in C^I\}$

$\mathcal{ALC}$ concepts can be seen as fragments of first-order predicate logic. According to the interpretation, every role and concept is mapped to an unary or binary relation over Δ^I - C can be translated into a predicate logic formula.

1.1.3 ABox and TBox

A knowlegde base can offer the functionality of a textslterminological box (TBox) and an *assertional box* ABox. The purpose of a TBox is to introduce the *terminology* to components of the application domain, e.g. vocbulary. The ABox contains *assertions* about individuals and the relationships among them. ABox and TBox statements can be written in first-order logic. A DL system usually provides reasoning tasks, e.g. if the set of assertions is consitent.

1.1.4 Inference Problems

A DL based knowledge representation system has beside the purpose of storing information, the task to provide reasoning. An arbitary model T only makes sense, if it is not contradictory. A model T is contradictory, if there does not exist an interpretation that satisfies T. In that case T is called *unsatisfiable*, *unsatisfiable* otherwise. A unsatisfiable model is in most cases useless, because it cannot be interpretated.
KB Satisfiability can be considered as umbrella term for several inference problems, which are listed below.

Satisfiablility A concept C is satisfiable with respect to its model T if there exists a model I of T such that C^I is nonempty.

Subsumption A concept C is *subsumed* by a concept D with respect to T if their intepretations are subsumed for every model I in T.

Equivalence Two model C and D are *equivalent* with respect to T if $C^I = D^I$ for every model I of T.

Disjointness Two model C and D are *disjoint* with respect to T if $C^I \cap D^I \equiv \{\}$ for every model I of T.

Chapter 2

BDD Tableaux

When description logics, in particular $\mathcal{ALC}$, have been invented, algorithms for basic inference problems had to be found. Tableaux, or tableau, was a first succesfull attempt for an algorithm, and is still commonly used.
Bdd tableaux uses structures like Bdds.

2.1 Boolean Function

A boolean function is per definition a function f of n arguments is mapping from $\{0,1\}^n$ to $\{0,1\}$. Several operator help to build a function:

complement	$\neg$	$\neg a$
product	$\wedge$	$a \wedge b$
sum	$\vee$	$a \vee b$
xor	$\oplus$	$a \oplus b$

There are sveral ways to represent a boolean function like as a propositional formula, a truth table, a BDD, a conjunctive normal form or a disjunctive normal form.

2.2 Binary Descision Diagramm

A Binary Descision Diagramm(BDD) represent a boolean function. It is an acyclic graph, consiting of nodes, that represent atomic propositional variables. Like a boolean function, the result of a BDD for every assignement is either true or false(Usally represented through *1* and *0*). Those values are the leaves in the BDD graph and are called terminals.
Edges represent the evaluation of the previous node. Therefore we can state that a single path in the graph represents a variable assignement. BDDs have been commonly used, because test for satisfiability and vailidity are fairly easy

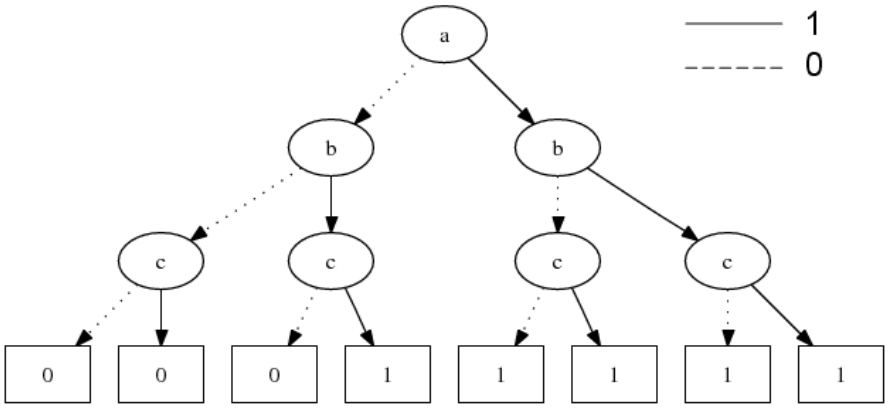


Figure 2.1: Binary Decision Diagramm of the boolean formula $a + b * c$

compared to other representations of boolean function like truth table, CNF or DNF.
An unordered and not reduced BDD, are inefficient.

2.2.1 Ordered BDD

If the sequence of variables in every path of the BDD graph is consistent with a fixed order, the BDD is called ordered (OBDD). OBDDs do not improve the datastructure BDD, cause the size stays the same in both cases.

2.2.2 Reduced BDD

Under application of the socalled Labeling algorithm, the BDD can be reduced(RBDD). The result is probably more efficient, but not unique.

2.2.3 Ordered and Reduced BDD

Working with a Reduced Ordered BDD is efficient, compared to the previously mention representations. Unfortunately this depends on the order of the variables. Finding the optimal, in other words most efficient, BDD is complex. In the implementation the order depends on the first occurence of a variable in the $\mathcal{ALC}$ expression.
The figure 2.2 shows the boolean formula $a + b * c$ as a ROBDD. Compared to the previous figure, it is considerably smaller and clearer.

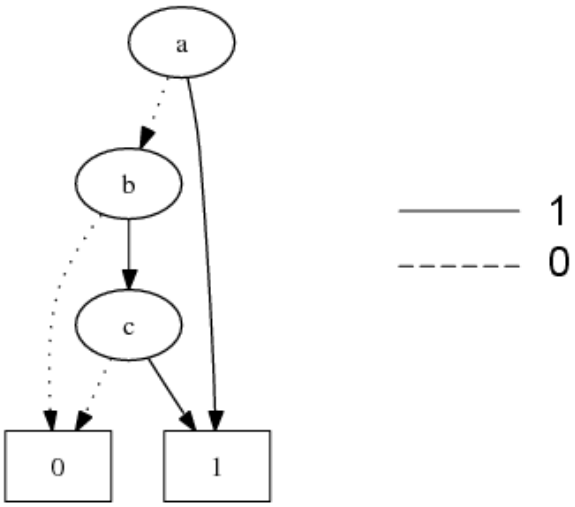


Figure 2.2: Reduced and ordered BDD of the boolean formula $a + b * c$

2.3 Tableaux Algorithms

Tableaux methods are algorithms to test satisfiability of formalisms in variaous logics. Hence, it can also be used for First Order Logic[1]. In the application field of description logics it is usually used to decide concept satisfiability. The architecture of tableaux is very general - special features are not available. Concerning $\mathcal{ALC}$, tableux has to be extended to handle ABox and TBox statements properly. This task is, according to Franz Baader and Werner Nutt, the authors of the book *The description logi handbook*, ëasy:

2.3.1 Algorithm

Tableaux provides a set of rules, which are used to check the satisfiablility of a $\mathcal{ALC}$ expression. Tableaux tries to build a tree-like model I of an input concept C. Subconcepts are transformed into Negation Normal Form(NNF), using De Morgan's rules, e.g.

$$\neg\exists R.C \longrightarrow \forall R.\neg C$$

Tableaux generates a tree T with certain properties

- Nodes are labeled with subconcepts of C
- Edges are labeled with roles

[1] http://en.wikipedia.org/wiki/First-order_logic

In the initialization, T contains only a single mathc, which is labeled only with the concept C itselt. In an iterative manner all completion rules are applied on T, which either extend or modify the tree structure. During the modification and extention socalled *clashes* can occur. This means that the tree T contains a contradiction. The current intepretation of T cannot be a model (unsatisfiable), therefore the clash has to be eliminated, if possible.

Some tableaux rules are nondeterministic (e.g $\sqcup$) and can offer an alternative. If no further Tableaux rules can be applied, T is called *fully expanded*. A concept C is satisfied if a fully expanded tree is contradiction free (*clash*-free). Otherwise it is not satisfiable.

2.3.2 Tableaux Rules

The tableaux rules describe the way to build a tree from an $\mathcal{ALC}$ expressions. There are four rules to derive $\sqcap$, $\sqcup$, $\exists$ and $\forall$.

2.4 BDD Tableaux

The bachelor thesis follows the idea of creating a new Tableaux related algorithm, that merges BDDs and tableaux.

2.4.1 Idea

BDDs are used to represent boolean functions. $\mathcal{ALC}$ concepts contain(in most cases) boolean functions too.
If an $\mathcal{ALC}$ concepts is tested for satisfiability using Tableaux all propositional statements have been processed and included into the tree. This operations can be sourced out to the BDDs. The new algorithm only cares of modal operators (in this thesis only $\exists$ and $\forall$ quantifier occur). A tree that only consists of atomic concepts is satisfiablility-tested per se, because its a BDD. A BDDs is omniscient, it knows all models of the formula it represents. This knowledge can probably be used to improve the old tableaux and get the new *BDD Tableaux*.

2.4.2 Algorithm

Hence, the BDD tableaux has an advance of knowledge, compared to the standard tableaux. To really use this fact another consideration is required.
Like tableaux, bdd tableaux builds a tree, the nodes in the tree contain modal or atomic concepts. Modal concepts are complex and have to be tested itself for satisfiability, therefore they extend or modify the tree when evaluated.
The *Bdd tableau* algorithm has been published in a scientific paper by Uwe Keller(Detailed information can be found in the bibliography).

2.4.3 Backtracking

As already mentioned, some operators, which are considered in this thesis, are non-deterministic. The operator $\sqcup$ in the example concept

$$a \sqcup b$$

means, per definition

$$a == True \vee b == True$$

For testing those operators, we have to choose one of the subconcepts, assuming it evaluates to true. This assumption can disprove. The disprooved model is inconsistent, to verbalize it is unsatisfiable for the current assignement.
BDD tableaux tries to correct models, that contain inconsistencies, using *backtracking*. It tries to undo actions that led to the troubles and find an alternative, which is possibly available(remember the non-determinism).

2.5 Example

This section presents an example proof of the $\mathcal{ALC}$ expression

intersection(some(R0, intersection(some(R1,C1), all(R1,not(C1)))),
union(A1, union(A2, union(A3, union(A4, A5)))))

, that is unsatisfiable and triggers many backtracking steps until the algorithm terminates. Figure 2.3 shows the initial tree of the concepts.

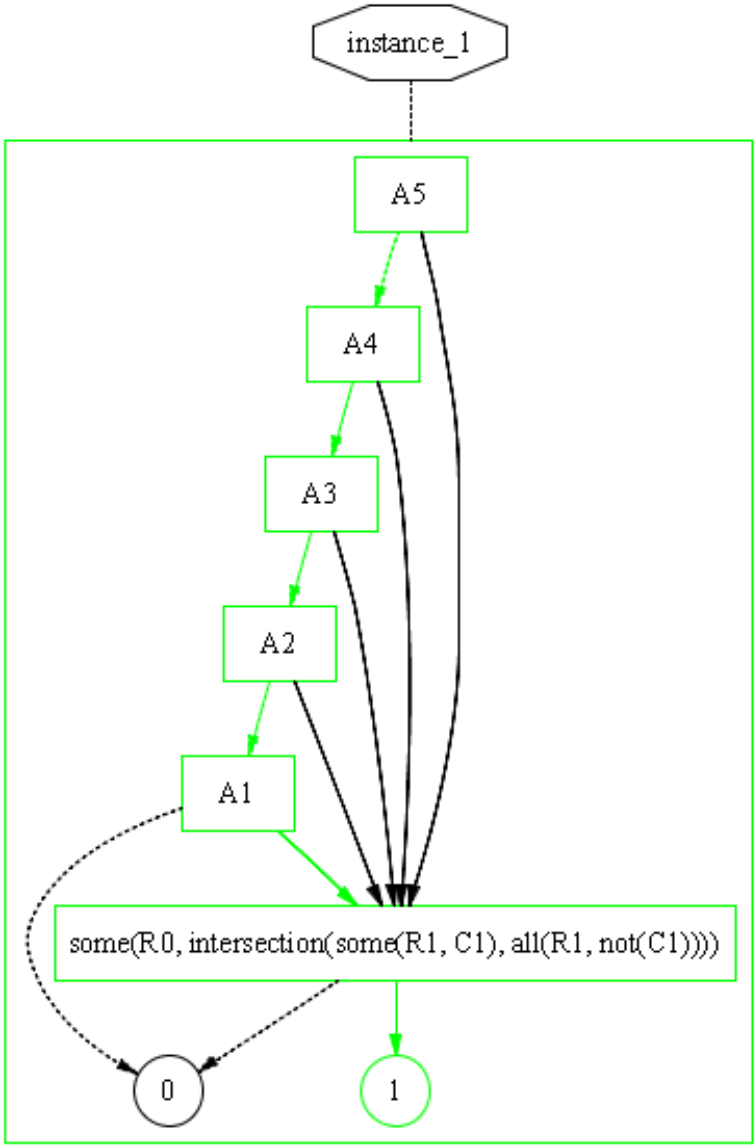


Figure 2.3: Intialization

Figure 2.4 shows the proof after choosing two 1-paths. In Figure 2.3 every 1-path contains the node labeled with the $\mathcal{ALC}$ concept

$$some(R0, intersection(some(R1, C1), all(R1, not(C1))))$$

. This node is followed by a positive edge, and causes, according to the bdd tableaux algorithm, an extension step. Instance 3 consists only of the 0-terminal, that means it is unsatisfiable and causes backtracking. Instance 2 contains only one 1-path - a second backtracking step is triggered.

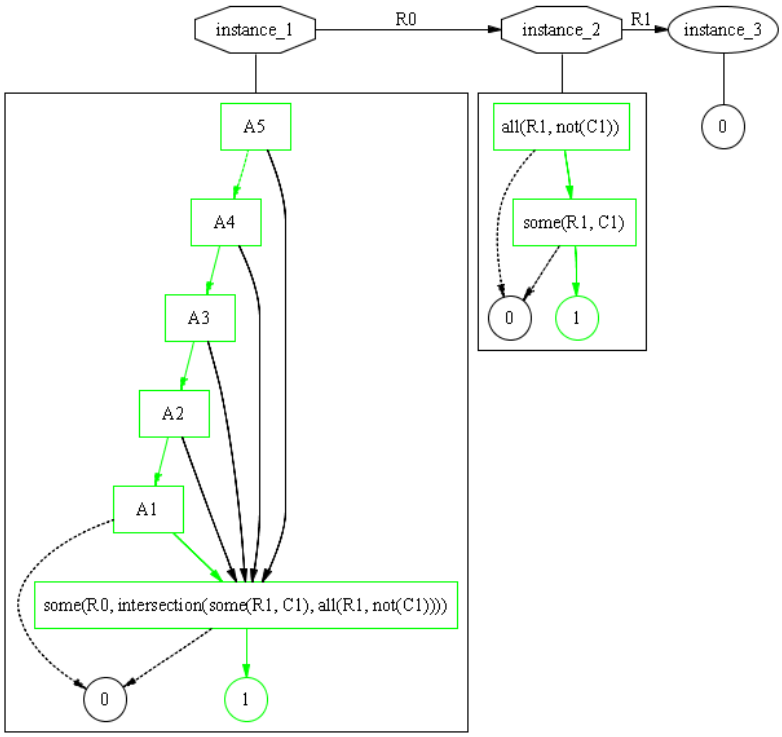


Figure 2.4: Unsatisfiable - backtracking will be triggered.

In the end all paths in instance 1 are unsatisfiable. Figure 2.5 of the example shows the final graph.

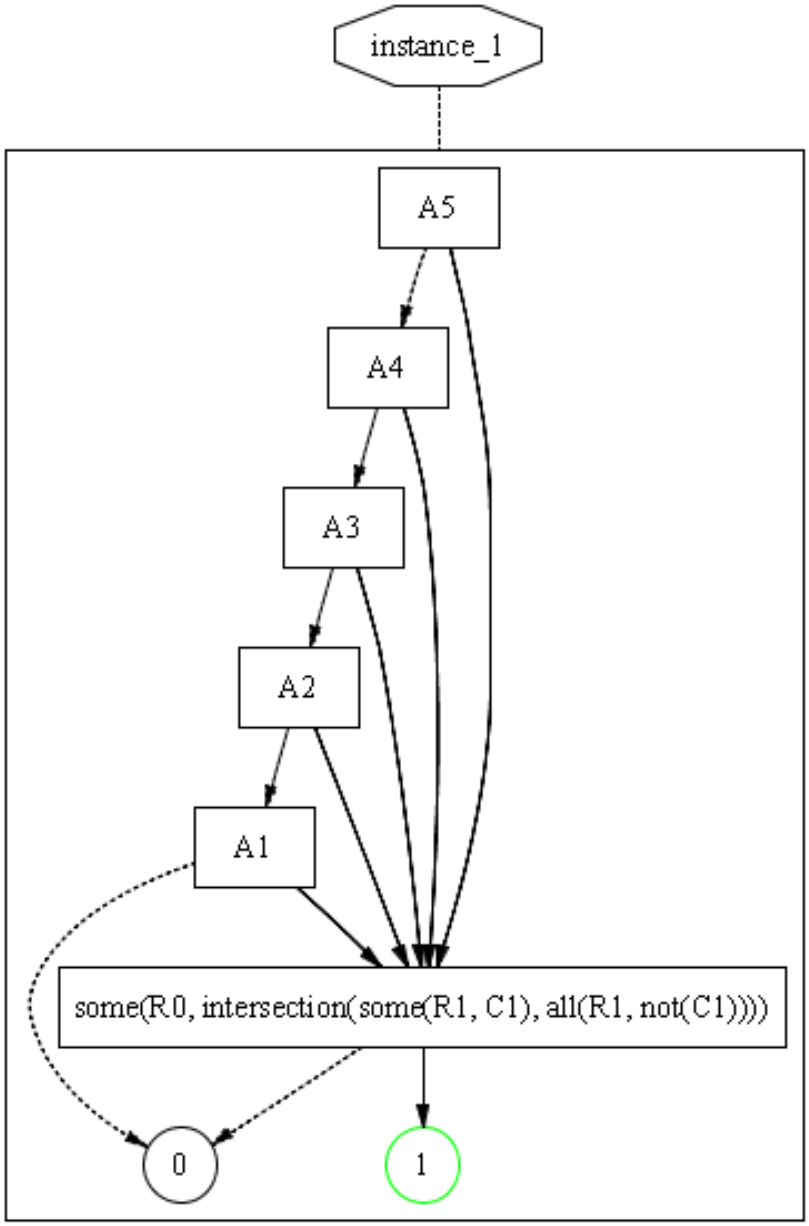


Figure 2.5: Unsatisfiable concept.

Chapter 3

Tool

This chapter deals with the implementation, and describes its user interface and features. Detailed implementation details won't be provided. That topic is handled in the next part, the *implementation part*.

3.1 Graphical User Interface

When the tool is started, the user sees a single window, which offers him the whole functionality, the tool provides. To interact with the tool, named *Reasoner for Description Logic*, the user has to use the GUI. There is no Command Line Interface(CLI).

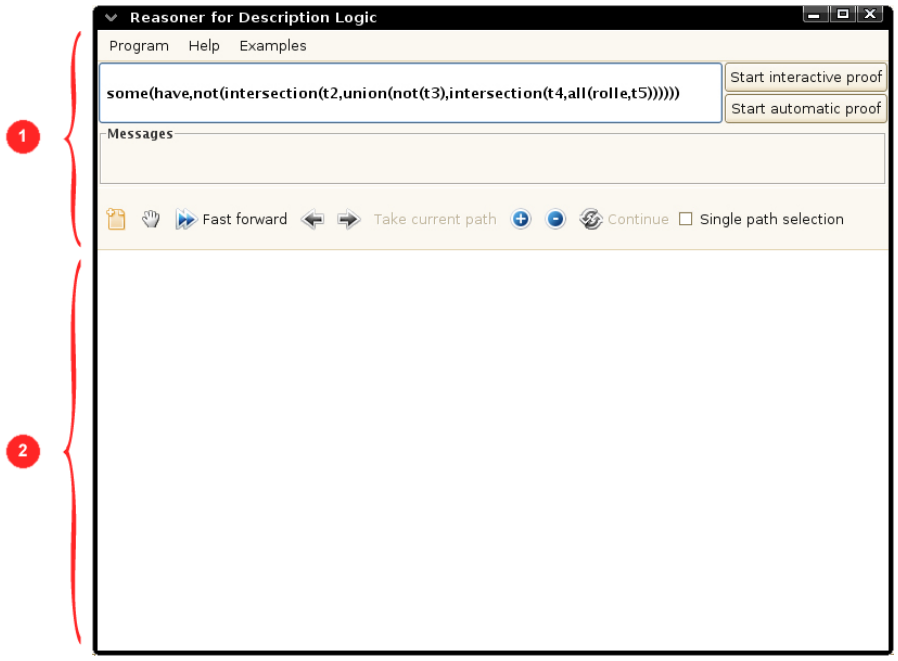


Figure 3.1: Graphical User Interface for the tool

The simple interface proviced by the application, enables a user to watch a proof and manipulate the proof process individually and interactively. Therefore, the user interface can be splitted into two main parts(1 and 2).

Part 1: Operate, Handle

Part one supplies the necessary functionality to interact and influence the proof process. At the beginning of a proof, a arbitary $\mathcal{ALC}$ expression, based on the right Syntax, is entered into the input field.

After entering a expression of your choice, the proof can be started through clicking on one of the buttons *Start interactive proof* or *Start automatic proof*. Information during the proof and advises are shown in the *Messages* frame. This includes prompts, concerning what to do next in the proof.

Figure 3.2: Message frame

Figure 3.3: Proof menu

The proof menu is used during the proof to react on prompts. Some buttons are enabled, some not, depending on the proof state and to simplify the usage.

Part 2: Vizualisation

In this area the representing bdd tableaux is displayed consisting of all nodes, which are important to handle the current proof state.

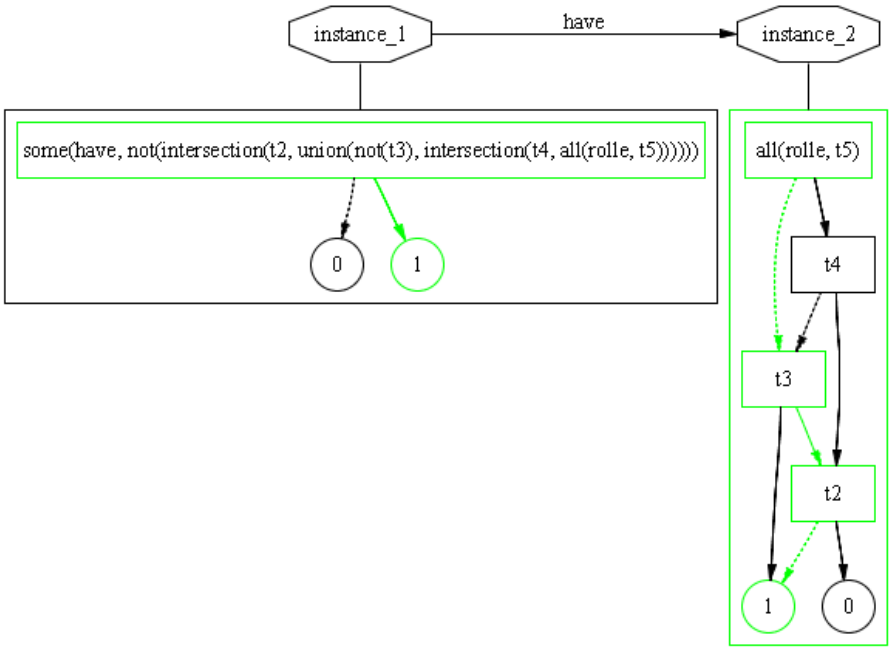


Figure 3.4: Example bdd tableaux with two instances

A bdd tableaux tree consists of at least one instance. All instances are surrounded by a box. The box is marked green if it contains the current bdd instance of interest. Inside the green box the user can switch between the 1-paths through clicking the buttons *path selection*(next chapter).

Selected 1-paths, that contain modal operators, like $\exists$ or $\forall$ quantifier, can trigger extension steps. In that case the a new instance will be generated, which checks the satisfiability for this modal operator.

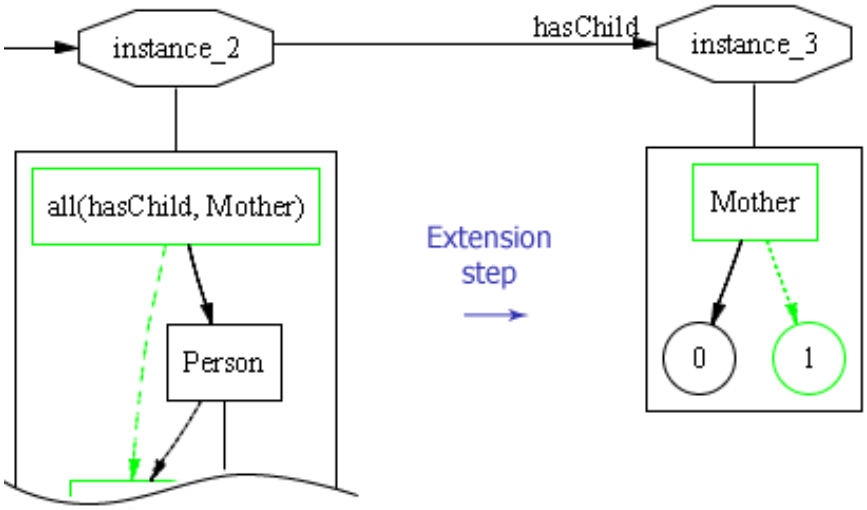


Figure 3.5: Extension step

Unsatisfiable bdds are displayed using the 0-terminal.

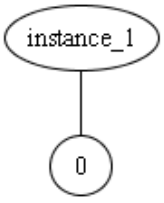


Figure 3.6: Unsatisfiable bdd instance

3.2 Special Features

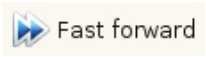
3.2.1 New proof

This button performs a reset of the current proof. All data will be lost. A different way to start a new proof, is simply to enter a new $\mathcal{ALC}$ expression and hit the button *Start* ...

3.2.2 Move

In larger proofs the displayed tableaux bdds can become huge. Using the *hand* tool the user can move the image and focus on the currently important things.

3.2.3 Fast forward



This buttons allows to skip a number of path selection-steps during the proof. Instead of the manual selection, the heuristic of the automated proof mode is used.

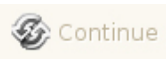
3.2.4 Path selection

Using the two buttons on the left, the user can navigate through the available satisfiable paths of the bdd instance. The button *Take current path* takes the currently selected path(green) and the proof will be continued.

3.2.5 Zoom

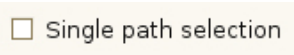
The *Zoom* feature is limited. A user can zoom in(plus) and out(minus).

3.2.6 Continue



In the automated proof mode, the reasoning algorithms can notify the user through messages. When a message appears the proof is frozen. The *Continue* button unfreezes it and shows that the user has recognized it.

3.2.7 Single path selection



A bdd instance that contains only one satisfiable path, does not have to be selected. Anyhow, usually the user has to do that. Checking the *Single path selection* box avoids that.

3.3 $\mathcal{ALC}$ syntax

At the beginning of a proof, there is a $\mathcal{ALC}$ expression. To build a valid $\mathcal{ALC}$ expression, which is well understandable for the tool, the user has to follow a strict syntax, which does not allow the commonly used special chars like $\sqcap$. Those chars are expressed using a plain text. The following table shows the mapping from the common special chars to the valid syntax for the tool.

$\mathcal{ALC}$ special char	Tool's syntax
$\neg a$	$not(a)$
$a \sqcap b$	$union(a, b)$
$a \sqcup b$	$intersection(a, b)$
$\exists R.a$	$some(R, a)$
$\forall R.a$	$all(R, a)$

Table 3.1: $\mathcal{ALC}$ syntax migration

3.3.1 Simple $\mathcal{ALC}$ examples

Satisfiable

1. *not(Human)*

2. *intersection(Human,Female)*

3. *intersection(Human, union(Female, Male))*

Unsatisfiable

4. *intersection(union(not(Female),Human), intersection(union(not(Male),Human), intersection(not(intersection(Male,Female)), not(union(not(union(Female, Male)), Human)))))*

3.3.2 Complex $\mathcal{ALC}$ examples

Satisfiable

5. *intersection(Human, some(father,Human))*

6. *intersection(Human, all(parents,Human))*

7. *intersection(Human, intersection(some(father, intersection(Human,Male)), some(mother,intersection(Human,Female)))*

8. *union(a, intersection(some(r, b), all(r, not(b))))*

Unsatisfiable

9. *union(union(intersection(some(R1,C1), all(R1,not(C1))),intersection(some(R2,C2), all(R2,not(C2)))),union(intersection(some(R3,C3), all(R3,not(C3))),intersection(some(R4,C4), all(R4,not(C4)))))*

10. *some(R1, some(R2, intersection(some(R3,C), all(R3,not(C)))))*

11. *intersection(A1, intersection(some(R, union(C, intersection(A1, intersection(some(R, C),all(R, not(C)))))),all(R, not(C))))*

12. *intersection(some(R0, intersection(some(R1,C1), all(R1,not(C1)))), union(A1, union(A2, union(A3, union(A4, A5)))))*

Chapter 4

Implementation

4.1 Design

Generally speaking, the right use of design patterns makes the code more readable, structured, reuseable, simply more *beautiful*. On this account we looked for an design pattern that provides an appropriate solution for our task, and found model view controller(MVC) pattern. MVC, as the name indicates, encaplulates the software system into three main components. Those parts can be described like the following:

Model contains the data. It is independent from the other parts - view and controller.

View is used for the presentation. The required data is provided by the model.

Controller enpowers to interact between the model and the view. User requests can be handled and restricted in the controller.

In the implementation for the thesis, special data structures are needed - like a structure to store BDDs. Therefore, in a closer view to the architecture, the MVC components in the implementation comprise: the model is representated through a BDD-structure, the view is a simple swing based JPanel and the controller coherends the BDD-tableaux algorithm.

4.1.1 Path Heuristic

A path in the bdd tableaux, that leads to the 1-terminal is called *1-Path*. Those 1-paths represent possible assignements that probable satisfy the input concept C. The advandage of the bdd tableaux algortithm is that all possible 1-paths can be compared and the best is chosen. Thus satisfiability checks can be improved.
The seeking the best path is called heuristic. Finding a good heuristic is certainly difficult. In the implementation, two different heuristics are used.

Heuristic number one is the intelligence of the human user, who chooses the, what he believes, "best" 1-path. In the case the tree is simple, compact and clearly arranged, this mode can possibly provide good results fast.
The second heuristic is used in the *automatic proof mode*. All path are sorted on basis of two parameter, which are

1. **Complexity** stands for the costs for extensionsion and propagation steps
2. **Length** of the path(number of nodes in the path)

The complexity of a path overwheights the length, that means, the implemention always takes the less complex path. In case of the same complexity, it takes the shorter one (length).

4.2 $\mathcal{ALC}$ Parser

Starting a proof, a $\mathcal{ALC}$ expression input string has to be parsed to a computer readable format. Writing a complex parser is hard and may be errorprone. For that reason socalled parser generators can be used, which compile a previously specified parser for the desired language. In our context we used the open source solution JavaCC[1].

4.2.1 JavaCC

JavaCC is a LL(K)-Parser Generator. In a parser specification file, the manner of the parser can be dscribed using a BNF-like syntax. This file (default suffix is jj) contains a mixture of java and javacc code.

```
1  PARSER_BEGIN(MyParser)
2  (..)
   public class MyParser {
4
           (..)
6          this.parse();
   }
8
   PARSER_END(MyParser)
```

Figure 4.1: Plain java part of JavaCC specification file

In the first part of this jj file, surrounded by PARSER_BEGIN and PARSER_END pure java code can be used to initialize the parser(*MyParser*) and invoke the

[1] Java compiler compiler: https://javacc.dev.java.net/

methods to parse the expression (*parse*). This parse method will be generated, and is specified below.

```
1   SKIP: { " " | "\t" }
2
    TOKEN : { < FORALL : "all" > }
4   TOKEN : { < TEXT : ( ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] | "_")+ > }
    (...)
6
    IExpression parse() :
8   {
            Token t;
10          IExpression e;
    }
12  {
            t = <TEXT>
14          {       return new AtomicConcept(t.toString()); }
    |
16          e = Forall()
            { return e; }
18  }

20  (...)
```

Figure 4.2: BNC like description of the grammer

In Figure 4.2 shows a second peace of javacc code, which is used to generate a parser. Tokens describe the main part of a expression. Regular expressions can be used to identify them. The SKIP elements contain expressions that are to be ingnored. Methods, like parse() and Forall(), can be used to realize recursion. The JavaCC parser classes are generated by the provided command javacc

$$javacc < filename.jj >$$

The result is a pure java parser.

4.3 BDD structure

A fundamental part of the software archictecture is the model, in this thesis the model that holds the data in a BDD like structure. BDDs have been popular, for this reason, many free libraries are available. In the implementation we used Java BDD (JBDD), a java port for different C/C++ implementations like

BuDDy and CUDD. The implementation can be chosen at the initialization of the factory. JBDD operates on ordered and reduced bdds. The order of the terminals in the OBDD can be set, but per default it is the order of the occurrence of a terminal.
OBDDs are powerfull in the case of the best order of the terminals. Unfortunately, the effort to find the optimal order is fairly expensive. Java BDD does not provide labeling of terminals, it only uses indexing. Unfortunately node labels play an important role in the tableau algorithm we used in this thesis.

4.4 Visualization

Proofs can be very complex and therefore hard to understand. A proper step-by-step visualization helps you on your way of understanding.
In this implementation the process to visualize a bdd graph, consists of three steps.

- Serialize BDD Graph to DOT format and save in temporary file
- Locate all elements in the graph using *dot* from the GraphViz library
- display the graph in an Grappa Panel from the grappa library

Calculating the position of elements in a graph with an own algorithm, would go beyound the scope of this thesis. The GraphViz[2] library provides several algorithms for this purpose. The provided programs are command line driven and expect graphs in DOT format.

4.4.1 Serialization and Localization

The DOT format is a powerful markup language for directed and undirected graphs. In the implementation a bdd graph constists of several BDDs(handeled by JBDD), which have to be labeled(JBDD only uses indexes) and linked. In the visualization the user only sees one proof state with the current nodes of interest. All other extended nodes and their BDDs are hidden.
The abstract data of a BDD Graph is translated to DOT and written to a temporary file. Using the *dot* programm - provided by the GraphViz package, all elements in the graph will be positioned in a two dimensional cartesian coordinates system. The result of the command line driven programm is again in dot format.

4.4.2 Grappa

Filling the missing link in the visualization draft, Grappa[3], a Java Graph Package, comes into play. It provides a swing panel with the ability to read a dot-encoded text and generate an image. This Grappa Panel offers some more features, linke handeling user requests in the graph.

[2] http://www.graphviz.org/
[3] http://www.research.att.com/ john/Grappa/

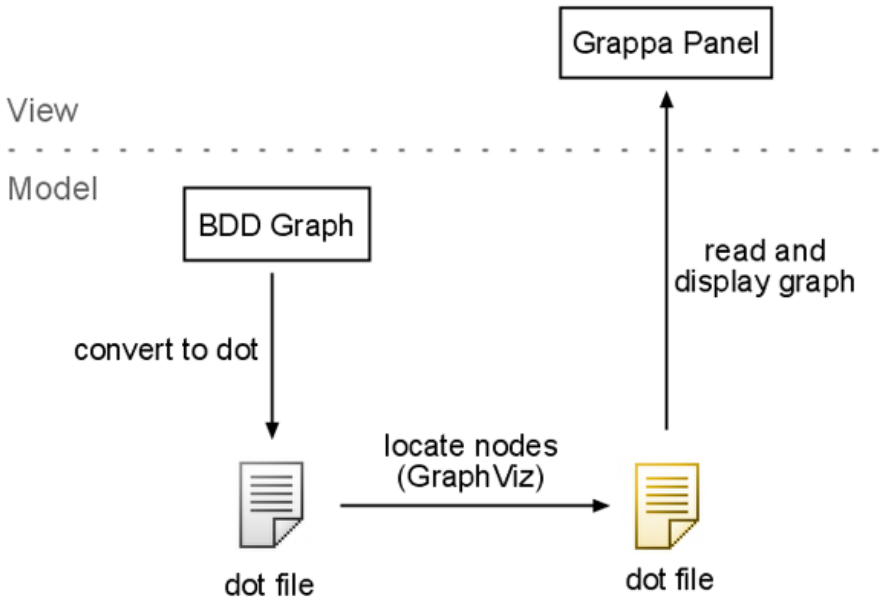


Figure 4.3: Visualize a BDD graph

4.5 Proof Modi

Proving satisfiability of an arbitary concept expression can be done in an manual and an automated way.

4.5.1 Manual Mode

This way of proving is the closest approach of demonstrating a proof with all its facets. For this purpose the user interacts after every proof step and chooses the one-paths, which are significant for the speed of the proof.

4.5.2 Semi-Automatic Mode

Another feature allows you to skip several proof steps and use the heuristic from the automatic mode. The feature is called *Fast Forward* and has already been mentioned in chapter 3. Using Fast Forward, the user can switch from the manual proof mode for a certain number of proof steps to the automated proof mode. This means, that during fast forward, the user does not have to choose paths at all.
Another way to achieve a semi-atomatic proof mode is to activate the *Single path selection*(SPS) checkbox. An activated SPS does not prompt the user to select a 1-path, if there is only one available.

4.5.3 Automatic Mode

According to the heuristics, presented before, the user does not have to interact with the programm. After starting a proof, the resulting graph including the message - satisfiable or not - will flush.

4.6 Dot Format

DOT is a powerfull path description language. Graphs are written in plain text, following the definition of the DOT format[4]. DOT is fairly simple to write. It provides a large amount of different shapes and objects.

[4]http://www.graphviz.org/pub/scm/graphviz2/doc/info/lang.html

Chapter 5

Conclusion

Yet, the implementation for this thesis has some nice features and helps to understand the *BDD Tableaux* and as a matter of course the basic Tableaux.

5.1 Suggestions for Improvement

5.1.1 Input Language

Currently the implementation accepts only simple input $\mathcal{ALC}$ concepts. The following operators are accepted:

- Not $\neg$
- Or $\sqcup$
- And $\sqcap$
- Exists $\exists$
- ForAll $\forall$

To complete the $\mathcal{ALC}$ language *Assertion Box*- and *Terminological Box* statements have to be added. This would enrich the implementation, and new fancy functionality would be available.

5.1.2 Path Heuristic

The second suggestion, I want to state, concerns the path heuristics, which is used in the *automatic proof mode*. Heuristics are used to make a decision. In this case it decides which 1-path is the best to take, to speedup the proof process. There, the underlaying guidlines are pretty simple and therfore not so powerfull.
Improvements of this aspect of the proof procedure could cause an big speed-up.

5.1.3 Graph Generation

Finally I want to mention the probably most important improvement. The implentation generates the graph image using the GraphViz package and the Grappa library. Therefore it creates an external process. GraphViz causes a huge slowdown of the whole programm, especially on the first generation of a graph. GraphViz is used to arrange all elements of the graph on a plane, to verbalize locate the elements.
A JNI[1] or even a plain java solution could solve this troubles. Localization algorithms are available in e.g papers, and just have to be implemented.

5.1.4 Platform Independency

Java has the amazing gift to be platform independent. Unfortunately this property is lost, or at least limited, in the implemention through the C-based library Java BDD. This library is necessarily used to store the BDDs. Though JBDD provides a shared library(so file), which is compiled for *x86* machines.

5.1.5 Drawbacks of Bdds

Bdds can be powerfull, but also weak. This fact depends on the order of the atoms, which cannot be set at the moment. A weak bdd is huge and the proof will durate long and cost a lot heap space. Finding a good order is expensive - nowadays there exist bether structures than bdds.

[1] Java Native Interface

Appendix A

DL Parser - JavaCC Code

```
/*
Parse dl- expressions
operator:
        union, intersection, not, some, all

examples:
        union(t3,t4)
        union(not(t3),t4)
        intersection(t2,union(not(t3),intersection(t4,t5)
            ))
        not(intersection(t2,union(not(t3),intersection(t4
            ,t5))))
        some(have,not(intersection(t2,union(not(t3),
            intersection(t4,t5)))))
        some(have,not(intersection(t2,union(not(t3),
            intersection(t4,all(rolle,t5))))))

*/

options {
        STATIC = false ;
}
PARSER_BEGIN(DLParser)

package expressions.parser.impl;

import java.io.ByteArrayInputStream;
import java.io.InputStream;

import expressions.IConcept;
```

```
import expressions.IExpression;
import expressions.IOperator;
import expressions.IRole;
import expressions.impl.AtomicRole;
import expressions.impl.ConceptComplement;
import expressions.impl.ConceptFactory;
import expressions.impl.ConceptIntersection;
import expressions.impl.ConceptUnion;
import expressions.impl.ExistentialRoleRestriction;
import expressions.impl.RoleFactory;
import expressions.impl.UniversalRoleRestriction;
import expressions.parser.IDLParser;

public class DLParser implements IDLParser {

        //count how many elements
        private int atomicElements = 0;

        //count restrictions like exists and forall
        private int restrictionCount = 0;

        private IConcept rootconcept = null;

        public static void main( String[] args )
                throws ParseException, TokenMgrError {

        String expr = "some(have,not(intersection(t2,
            union(not(t3),intersection(t4,all(rolle,t5))))
            ))";
        InputStream in = new ByteArrayInputStream(expr.
            getBytes());
        DLParser parser = new DLParser( in );
        parser.parse();
        }


        public int atomicElementCount() {
                return atomicElements;
        }

        public IConcept getRootConcept() {
                return rootconcept;
        }

        public int restrictionCount() {
                return restrictionCount;
```

```
        }
71
}

PARSER_END(DLParser)

76 /*
        Defining tokens
*/
SKIP: { " " | "\t" }
TOKEN : { <EOL : "\n" | "\r" | "\r\n" > }
81 TOKEN : { < OPEN : "(" > }
TOKEN : { < CLOSE : ")" > }
TOKEN : { < KOMMA : "," > }
TOKEN : { < UNION : "union" > }
TOKEN : { < COMPLEMENT : "not" > }
86 TOKEN : { < INTERSECTION : "intersection" > }
TOKEN : { < FORSOME : "some" > }
TOKEN : { < FORALL : "all" > }
TOKEN : { < TEXT : ( ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] |
    "_")+ > }

91 IExpression parse() :
        {
                IExpression expr;
        }
        {
96              expr = Expression()
                {
                        rootconcept = (IConcept)expr;
                        return expr;
                }
101     }

IExpression Expression() :
        {
                Token t;
106             IExpression e;
        }
        {
                t = <TEXT>
                {
111                     atomicElements++;
                        return (IExpression)(
                            ConceptFactory.getNamedConcept
                            (t.toString()));
```

```
                }
        |
                e = Union()
116             { return e; }
        |
                e = Complement()
                { return e; }
        |
121             e = Intersection()
                { return e; }
        |
                e = ForAll()
                { return e; }
126     |
                e = ForSome()
                { return e; }

        }
131
    /*
        union(<expression()>,<expression()>)
    */
    IExpression Union() :
136     {
                IExpression t;
                IExpression e1;
        }
        {
141             <UNION>
                <OPEN>
                t = Expression()
                { e1 = (IExpression)t; }
                <KOMMA>
146             t = Expression()
                <CLOSE>
                {
                        IOperator op = new ConceptUnion()
                            ;
                        return ConceptFactory.
                            getComplexConcept(op,(IConcept
                            )e1,(IConcept)t);
151             }
        }

    /*
        not(<expression()>)
```

```
156 */
    IExpression Complement() :
            {
                    IExpression t;
            }
161         {
                    <COMPLEMENT>
                    <OPEN>
                    t = Expression()
                    <CLOSE>
166                 {
                            IOperator op = new
                                ConceptComplement();
                            return ConceptFactory.
                                getComplexConcept(op,(IConcept
                                )t);
                    }
            }
171
    /*
            intersection(<expression()>,<expression()>)
    */
    IExpression Intersection() :
176         {
                    IExpression t;
                    IExpression e1;
            }
            {
181                 <INTERSECTION>
                    <OPEN>
                    t = Expression()
                    { e1 = (IExpression)t; }
                    <KOMMA>
186                 t = Expression()
                    <CLOSE>
                    {
                            IOperator op = new
                                ConceptIntersection();
                            return ConceptFactory.
                                getComplexConcept(op,(IConcept
                                )e1,(IConcept)t);
191                 }
            }

    /*
            all(role,<expression()>)
```

```
196 */
    IExpression ForAll() :
            {
                    Token t;
                    IRole role;
201                 IExpression e;
            }
            {
                    <FORALL>
                    <OPEN>
206                 t = <TEXT>
                    { //role = new AtomicRole(t.toString());
                      role = RoleFactory.getAtomicRole(t.
                          toString());
                    }
                    <KOMMA>
211                 e = Expression()
                    <CLOSE>
                    {
                            restrictionCount++;
                            IOperator op = new
                                UniversalRoleRestriction(role)
                                ;
216                         return ConceptFactory.
                                getComplexConcept(op,(IConcept
                                )e);
                    }
            }

    /*
221         some(role,<expression()>)
    */
    IExpression ForSome() :
            {
                    Token t;
226                 IRole role;
                    IExpression e;
            }
            {
                    <FORSOME>
231                 <OPEN>
                    t = <TEXT>
                    { role = new AtomicRole(t.toString()); }
                    <KOMMA>
                    e = Expression()
236                 <CLOSE>
```

```
                {
                        restrictionCount++;
                        IOperator op = new
                            ExistentialRoleRestriction(
                            role);
                        return ConceptFactory.
                            getComplexConcept(op,(IConcept
                            )e);
241             }
        }
```

Bibliography

[1] Uwe Keller *BDD-Tableau: Combining Rich Semantic Structures and Tableau-based Search for Reasoning in Description Logics* 2007: Digital Enterprise Research Institute, University of Innsbruck, Austria.

[2] Franz Baader, Werner Nutt *The description logic handbook* 2003: Cambridge University Press, New York, USA.

[3] Aart Middeldorp *Logic in Computer Science* 2006: Institute of Computer Science, University of Innsbruck, Austria.

[4] Ying Ding, Ioan Toma *Next Web Generation* 2006: Courtersy DERI Galway, University of Innsbruck, Austria.