University of Innsbruck
Semantic Technology Institute

Implementation of an ALC Tableau Reasoner

Bachelor Thesis

Simon Knoll
Karl-Innerebner-Strasse 76
A-6020 Innsbruck
Matrikelnr.: 0415801

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

Innsbruck, January 15, 2009

Chapter 1

Acknowledgements

First of all i want to thank Uwe Keller who supervised my project and had always time for me if i got stuck with my project.
Further i want to thank all the people who supported me doing this thesis.

Contents

Chapter 2

Introduction

Description Logics (DL) are a family of class-based knowledge representation formalisms characterized by the use of various constructors to build complex classes from simpler ones, and by an emphasis on the provision of sound, complete and (empirically) tractable reasoning services. They have a range of applications, but are most widely known as the basis for ontology languages such as OWL.

A specifically simple Description Logic is the logic $\mathcal{ALC}$. $\mathcal{ALC}$ can be seen as a syntactic variant of the (multi-)modal logic K.

The goal of the project is to implement the standard tableau-based decision procedure for checking concept satisfiability for the simple Description Logic $\mathcal{ALC}$ as well as well-known standard optimization techniques. Optional is the integration of additional language features (leading to more expressive DL) .

The implementation should be done in Java [2]. The system should be configurable in the sense that is easily possible to switch on / off the available optimizations.

The goal of the project is (a) to learn about DL and (for simple DL) about some classical proof procedures, (b) to learn how to implement such a (non-deterministic) procedure and (c) to see how proof search can be optimized and integrate a few well-known optimizations. An evaluation of the implemented procedure (and the effect of optimizations) could complete the project.[7]

Chapter 3

Description Logics

Overview Since the WWW was called to live by Tim Berners Lee the importance of knowledge representation is increasing. To use knowledge bases in an efficient way there is a need for a well defined semantic witch can be understand by humans and machines. Ontologies(sets of definitions and concepts) accomplish this requirements. Now to use ontologies for knowledge based systems we have to introduce a ontology language witch is expressive enough to describe relevant concepts at the one hand and at the other hand not too expressive, so that reasoning keeps achievable. And there DL comes into play.

3.1 Syntax

Description-Logics DL are a family of knowledge representing languages, most of them are a subset of *Predicate Logics* but are decidable on the contrary. This gives the possibility to conclude about a DL and to achieve new knowledge.
The most expressive means of DL are the so-called concept descriptions which describe sets of individuals or objects. Formally they are described by a set of concept constructors, concept names and role names, where the available concept constructors define the expressiveness of the DL. In the DL $\mathcal{ALC}$ (**A**ttributive **L**anguage with **C**omplements) the concept constructors are defined by negation, conjunction, disjunction, existential restriction and universal restriction.It was first introduced by Schmidt-Schau and Smolka in [9]. Basically the syntax of DL $\mathcal{ALC}$ is described by:

- atomic concepts (A,B)
- roles (R)
- defining new concepts by applying concept constructors ($\sqcup, \sqcap, \forall, \exists$)

A	atomic concept
$\top$	top
$\bot$	bottom
$\neg$ C	negation
$C_1 \sqcup C_2$	conjunction
$C_1 \sqcap C_2$	disjunction
$\exists R.C$	existential restriction
$\forall R.C$	universal restriction

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

Example: $happyParent \equiv person \sqcap \forall hasChild.computer - scientist$ All the children of a happy parent are computer scientists.

3.2 Semantics

The semantics of description logics is defined by interpreting concepts as sets of individuals and roles as sets of pairs of individuals. Those individuals are typically assumed from a given domain. The semantics of non atomic concepts and roles is then defined in terms of atomic concepts and roles. This is done by using a recursive definition similar to the syntax. Then let:

- $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be the interpretation
- $\Delta^{\mathcal{I}}$ the domain
- $\cdot^{\mathcal{I}}$ the interpretation function
- for all atomic concepts A: $A \subseteq \Delta^{\mathcal{I}}$
- for all roles R: $R \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

constructor	**syntax**	**semantic**
negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
disjunction	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y : (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
universal restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y : (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$

Table 3.2: concept constructors and semantic of DL $\mathcal{ALC}$

top	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
bottom	$\bot^{\mathcal{I}} = \emptyset$
equivalency	$X = Y <> X^{\mathcal{I}} \equiv Y^{\mathcal{I}}$
inclusion	$X \sqsubseteq Y <> X^{\mathcal{I}} \subseteq Y^{\mathcal{I}}$, where X,Y can be concepts or roles

Table 3.3: additional semantic information

Example:
$\Delta^{\mathcal{I}} = John, Peter, Jim, Sarah, Mike$
$(male)^{\mathcal{I}} = John, Peter, Jim, Mike$
$(hasChild)^{\mathcal{I}} = (John, Sarah), (Jim, Peter), (Sarah, Mike)$

Definition of a complex concept: $Father \equiv male \sqcap hasChild.\top$
$(Father)^{\mathcal{I}} = (male \sqcap hasChild.\top)^{\mathcal{I}} = (male)^{\mathcal{I}} \cap (\exists.hasChild.Person)^{\mathcal{I}} =$
$= \{John, Peter, Jim, Mike\} \cap \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in (hasChild)^{\mathcal{I}}\} =$
$= \{John, Peter, Jim, Mike\} \cap \{John, Jim, Sarah\} = \{John, Jim\}$

Chapter 4

Tableau-based Decision Procedures

4.1 Introduction

A Tableaux-calculus (also tree-calculus), is an algorithm to prove logical statements. It tries to prove satisfiability of a concept expression $\mathcal{D}$ by searching a model for $\mathcal{D}$. The Algorithm works on a tree T representing a model $\mathcal{I}$ of a concept $\mathcal{D}$ A model would be an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\mathcal{D}^{\mathcal{I}} \neq \emptyset$. The Tableaux algorithm starts with a single individual $\mathcal{D}$ and tries to construct a complete model by inferring additional individuals and constraints. More specifically, a tableau calculus consists of a finite collection of rules with each rule specifying how to break down one logical connective into its constituent parts. The rules typically are expressed in terms of finite sets of formulae. $\mathcal{D}$ is assumed to be an unfolded conceptexpression. Important for the calculus is, that all the concepts must be represented in *negation normal form*, which means that negations apply only to concept names and not to complex concept terms (such as intersection, union, etc.). If a concept isn't in the *nnf*, it can be simply transformed by applying DeMorgan's laws
$\neg(\exists R.C \sqcap \forall S.B) \longrightarrow \neg\exists R.C \sqcup \neg\forall S.B$
$\neg\exists R.C \sqcup \neg\forall S.B \longrightarrow \forall R.\neg C \sqcup \exists S.\neg B$
In $\mathcal{ALC}$ every concept expression can be transformed (in linear time) into an equivalent description in *nnf*. In the field of description logics, the most reasoner systems ar working with tableau calculi (such as Fact, Racer and Pellet) The following informations about tableau calculi where mostly taken from *An Overview of Tableau Algorithms for Description Logics* of *Franz Baader* and *Ulrike Sattler*[7]

4.1.1 Tableaux Algorithm for $\mathcal{ALC}$

To check the satisfiability of an $\mathcal{ALC}$ expression D a tree T is initialized with a single node labeled with the expression to prove. In this tree every node x represents an individual and is labeled with a set $\mathcal{L}(x)$ of $\mathcal{ALC}$ expressions and each edge $\langle x, y\rangle$ in the tree represents a pair of individuals and is labeled with a role name. The satisfiability will be proved by applying the following rules to an initialized tree T with one node x_0 labeled with $\mathcal{L}(x_0) = C$, where C is the expression to prove.

$x \circ \{C_1 \sqcap C_2\}$	$\rightarrow \sqcap - rule$	$x \circ \{C_1 \sqcap C_2, C_1, C_2, \dots\}$
$x \circ \{C_1 \sqcup C_2\}$	$\rightarrow \sqcup - rule$	$x \circ \{C_1 \sqcup C_2, C, \dots\}$ **for some** $C \in \{C_1, C_2\}$
$x \circ \{\exists R.C, \dots\}$	$\rightarrow \exists - rule$	$x \circ \{\exists R.C, \dots\}$ $R\downarrow$ $y \circ \{C\}$
$x \circ \{\forall R.C, \dots\}$ $R\downarrow$ $y \circ \{\dots\}$	$\rightarrow \forall - rule$	$x \circ \{\forall R.C, \dots\}$ $R\downarrow$ $y \circ \{C\}$

Table 4.1: short overview about Tableaux expansion rules

A tree T is fully expanded if none of the expansion rules are applicable. A tree T contains a contradiction if the for some node x yields that $C, \neg C \subseteq \mathcal{L}(x)$ or $\bot \in \mathcal{L}(x)$.

To guarantee termination of the Tableaux calculus we have to add some restrictions to the expansion rules in table 3.1

$\sqcap$-rule ***(intersection rule)*** if

1. $(C_1 \sqcap C_2) \in \mathcal{L}(x)$
2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$

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

$\sqcup$-rule ***(union rule)*** if

1. $(C_1 \sqcup C_2) \in \mathcal{L}(x)$
2. $(C_1, C_2) \cap \mathcal{L}(x) = \emptyset$ then
   a save T
   b try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1\}$
   If that leads to a clash, then restore T and
   c try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_2\}$

$\exists$-rule *(exists rule)* if

1. $\exists R.C \in \mathcal{L}(x)$
2. there is no y s.t. $\mathcal{L}(\langle x, y\rangle) = R \; and \; C \in \mathcal{L}(y)$

then create a new node y and edge $\langle x, y\rangle$
with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}(\langle x, y\rangle) = R$

$\forall$-rule *(forall rule)* if

1. $\forall R.C \in \mathcal{L}(x)$
2. there is some y s.t. $\mathcal{L}(\langle x, y\rangle) = R \; and \; C \notin \mathcal{L}(y)$

then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Due to this additional restrictions the algorithm will always terminate for expressions in $\mathcal{ALC}$ .It avoids multiple expansions of the same expression. In case of $\{\sqcup, \sqcap$ and $\exists\}$ expressions, the corresponding rules can be applied only once and $\forall$ expressions only once for every corresponding edge.

Because of the non-determinism of the $\sqcup - rule$ there is a need to handle it different.If there is a branching point *(disjunction)* a non-deterministic decision is made. In case of a clash, the decision must be undone as shown in the upper description. The rule terminates if every expansion leads to a clash, or a fully and clashfree expansion was found.

Example

$$Z \equiv H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C))$$

1. Initialize tree T with a single node labeled by the concept to prove
   - $x_0 \; \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C))\}$
2. Apply the *intersection* expansion rule on node x_0
   - $x_0 \; \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C))\}$
   - $x_0 \; \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
3. As next apply the *existential* expansion on node x_0
   - $x_0 \; \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $x_0 \; \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y\rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C\}$

4. Apply the *forall* expansion rule on node x_0
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C\}$
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C)\}$
5. Apply a non-deterministic decision on node $x_1 \rightarrow save\mathrm{T}$
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C)\}$
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C), \neg C\}\lightning$
6. A clash was identified, now restore T and mark already chosen branch
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C)\}$
7. Choose other branch
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C), A\}$
8. Now the Tableaux is fully expanded and the expression results as satisfiable
   - $\circ\ x_0\ \mathcal{L}(x_0) = \{H \sqcap (\exists R.C) \sqcap (\forall R.(A \sqcup \neg C)), H, (\exists R.C), (\forall R.(A \sqcup \neg C))\}$
   - $\downarrow \mathcal{L}(\langle x, y \rangle) = R$
   - $\circ x_1 \mathcal{L}(x_0) = \{C, (A \sqcup \neg C), A\}\mathcal{OK}$

4.2 Optimizations

If a reasoning algorithm should be useful in practice and fulfill real requirements, it should handle expressive logics at the one side an do fast reasoning at the other side. But practice [8] has shown performance as a big problem, even with limited expressiveness. Therefor an implementation of some wellknown optimizations is required. The different optimization techniques where taken from *chapter 9* of *The Description Logic Handbook: Theory, Implementation, and Applications*, written by *Ian Horrocks*[8]

4.2.1 Normalization and Simplification

The standard *unfolding process* for the *tableau calculus* wouldn't discover a concept expression and its negation as a contradiction. eg. $(\neg A \sqcup \neg B) and (A \sqcap B)$. This would very helpful, an can easily reached by transforming concept expressions into a syntactic normal form. Then it is possible to locate contradictions of composed concept expressions such as with atomic concepts. Redundancy is another problem which can be eliminated by *normalizing* concept expressions. In $\mathcal{ALC}$ the simplest way is to transform due to the *De'Morgans Law* all discuntions in to conjunctions and all *existential restrictions* in to into *universal restrictions*. Important is that within the normalization process conjunctions are considered as sets, which simplifies the elimination of redundant conjuncts and the identification of tautologies. So in this case a conjunction $(C_1 \sqcap \cdots \sqcap C_n)$ is representated as $\sqcap\{C_1, \ldots, C_n\}$. The *normalization* and *simplification* is done by a set of recursive functions, applied to the concept expression to check.

Function		Return Value
$Norm(A)$	$=$	A
$Norm(\neg A)$	$=$	$Simp(\neg(Norm(A)))$
$Norm(C_1 \sqcap \cdots \sqcap C_n)$	$=$	$Simp(\sqcap\{Norm(C_1)\} \cup \cdots \cup \{Norm(C_n)\})$
$Norm(C_1 \sqcup \cdots \sqcup C_n)$	$=$	$Norm(\neg(\neg C_1 \sqcap \cdots \sqcap \neg C_n))$
$Norm(\forall R.C)$	$=$	$Simp(\forall R.Norm(C))$
$Norm(\exists R.C)$	$=$	$Norm(\neg \forall R.\neg C)$

Figure 4.1: Normalization Rules

Function		Return Value
$Simp(A)$	$=$	A
$Simp(\neg C)$	$=$	$\begin{cases} \bot, & \text{if } C = \top \\ \top, & \text{if } C = \bot \\ Simp(D) & \text{if } C = \neg C \\ \neg C & otherwise \end{cases}$
$Simp(\sqcap \mathrm{S})$	$=$	$\begin{cases} \bot, & \text{if } \bot \in \mathbf{S} \\ \bot, & \text{if } \{C, \neg C\} \subseteq \mathbf{S} \\ \top, & \text{if } \mathbf{S} = \emptyset \\ Simp(\mathbf{S} \setminus \top), & \text{if } \top \in \mathbf{S} \\ Simp(\sqcap \mathbf{S} \cup \mathbf{P} \setminus \{\sqcap\ \mathbf{P}\}), & \text{if } \sqcap\{\mathbf{P}\} \in \mathbf{S} \end{cases}$
$Simp(\forall R.C)$	$=$	$\begin{cases} \top, & \text{if } C = \top \\ \forall R.C, & otherwise \end{cases}$

Figure 4.2: Simplification Functions

4.2.2 Semantic Branching

The standard syntactic branching function choose a unfolded disjunction $(C_1 \dots C_n)$ and search the different models by adding each of the disjuncts. As the alternative branches are not disjoint, the recurrence of an unsatisfiable disjunct in different branches can occur. This can lead to a lot of wasted expansions as shown in the following example.

assume a node x with label $\mathcal{L}(x) = \{(A \sqcup B), (A \sqcup C)\}$ where A leads to a clash

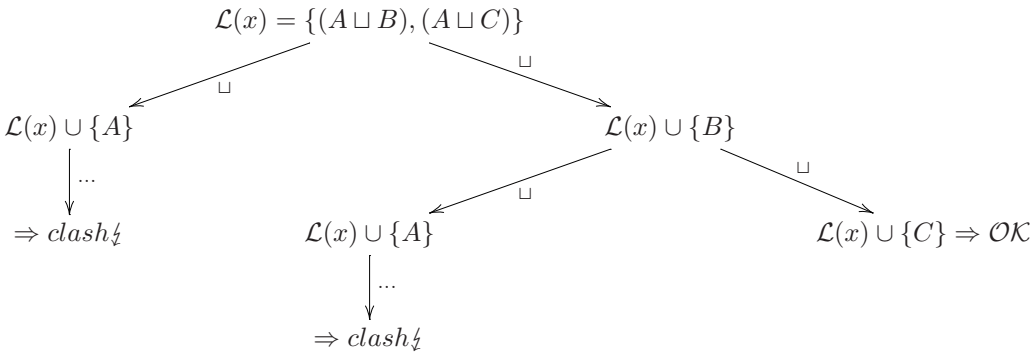

Figure 4.3: Syntactic Branching with wasted expansion

As supposed before, with syntactic branching we've got in this example a

wasted expansion. There, the advantage of the *semantic branching* comes into play.

With semantic branching a single disjunct D is chosen from a unexpanded disjunction, where the possible subtrees obtained by adding D or $\neg D$. Now we have to disjoint subtrees and the possibility of wasted expansions such as in syntactic branching is banned.

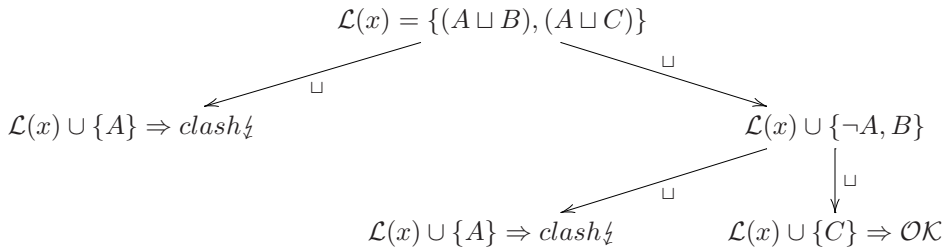

Figure 4.4: Sematic Branching

With *Semantic Branching* only one exploration of the expression A was needed. With *Syntactic Branching* instead, there where two explorations needed. The resulting search space can be reduced if semantic branching is used in collaboration with Local Simplification.

4.2.3 Local Simplification

Local Simplification is a technique to reduce the amount of branching in the expansions of node labels. This optimization method deterministically expands disjunctions in $\mathcal{L}(x)$ where only one expansion possibility is given. If a expansion is detected where no expansion is possible it detects the clash automatically. Local simplification is also known as *boolean constraint propagation*. The effect of *Local Simplification* is gained by applying two simple rules on conjunctive concepts.

$$\frac{\neg C_1,\ldots,\neg C_n,\neg C_1 \sqcup \ldots, C_n \sqcup D}{D} \text{ and } \frac{C_1,\ldots,C_n,\neg C_1 \sqcup \ldots,\neg C_n \sqcup D}{D}$$

Figure 4.5: Local Simplification inference rules

In the following example we see the effect of *Local Simplification* in combination with *Semantic Branching*

The advantage of Local Simplification is that it can be applied to many algorithms and logics. Also it never increases the search space of a Tableaux.

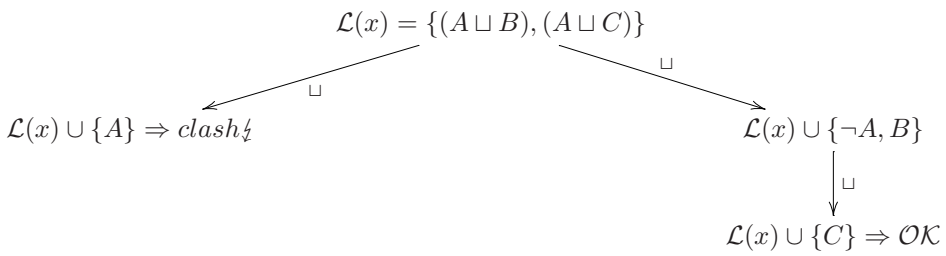

Figure 4.6: Local Simplification and Semantic Branching

4.2.4 Dependency Directed Backtracking

Branching expansions generate exponentially many branches. If an entry leads to a clash, can only be detected if the entry is expanded. For example unfolding following concept $\mathcal{L}(x) = \{(C_1 \sqcup (D_1), \ldots, (C_n \sqcup (D_n), \exists R.(A \sqcup B), \forall R.\neg A\}$ could lead to the fruitless exploration of 2^n possible R successors of x until the inherent unsatisfiability is discovered. The problem within is, to locate the cause of clashes and using this to downsize the search space. So the main idea is to prune away dispensable branches and by jumping back to the last branching point which contribute to the conflict.

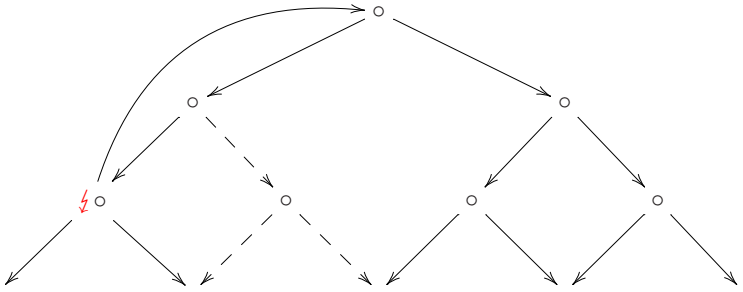

Figure 4.7: Pruning away unnecessary branches

To achieve pruning and backjumping, every branching point has to know the branch where it was generated. So there must be an order relation for the branching points.

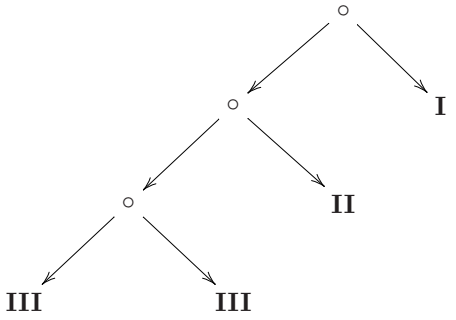

Figure 4.8: Order relation

Every new concept added to a node by applying the Tableaux rules inherits the dependencies from the concepts it was generated by. If the new added concept results by applying a non deterministic Tableaux rule, also a dependency from the new branching point is added.

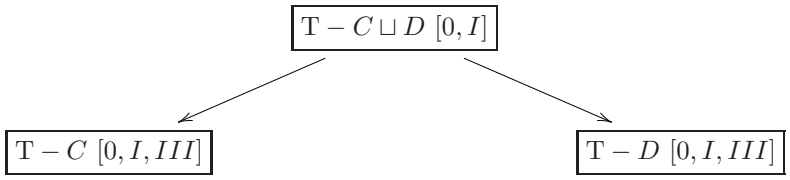

Figure 4.9: Dependencies

Now if a clash is identificated the dependency of the conflict is the union of the dependencies of the involved concepts

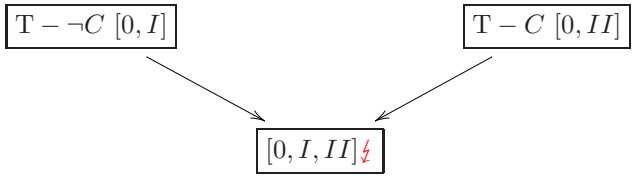

Figure 4.10: Clash dependencies

In this small example the clash depends on branchingpoints $0, I$ and II. Other branching points which are not in the dependency list can be leaved out (pruned)

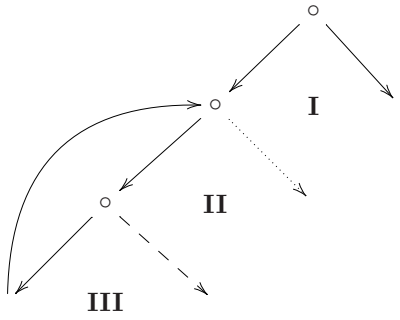

Figure 4.11: Pruning

In the figure over here the dashed branch can leaved out and the algorithm jumps directly back to the branchingpoint II and the dotted branch can be taken.

Now if we go back to the example, which we initially specified $\mathcal{L}(x) = \{(C_1 \sqcup (D_1), \ldots, (C_n \sqcup (D_n), \exists R.(A \sqcup B), \forall R.\neg A\}$ the branching tree would look like

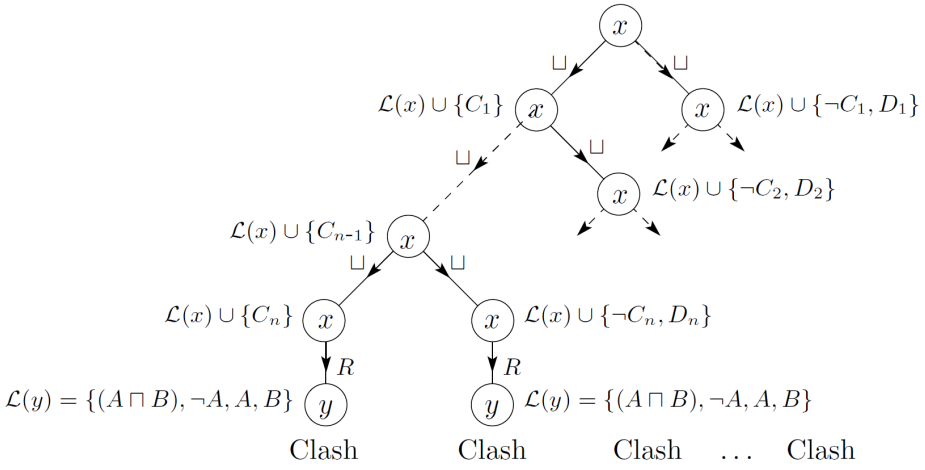

Figure 4.12: Trashing with normal backtracking

By applying the DDBT technique the search space is scaled down dramatically which can be seen in the following figure.

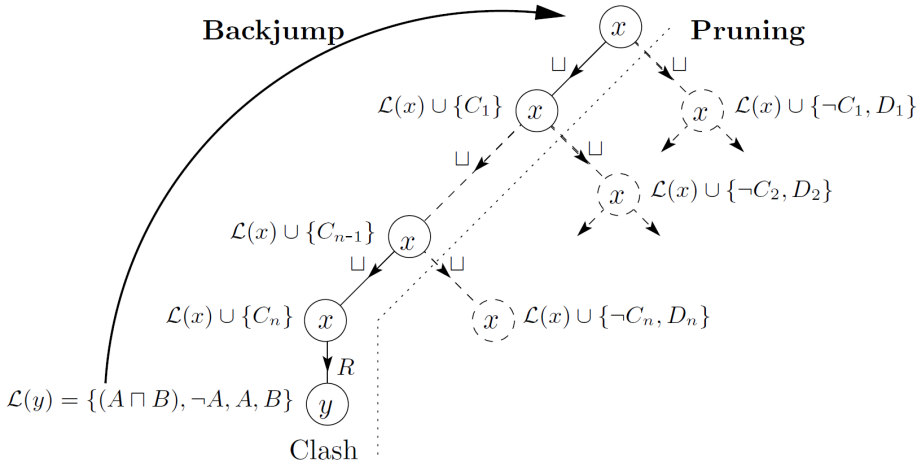

Figure 4.13: Same example with DDBT

Chapter 5

Implementation

5.1 Architecture

For the implementation the *Model View Controller (MVC)* pattern seemed the most adequate. Because it fulfills the claim to the project, to be flexible and extensible. This is covered by dividing a software project in three different Layers.

Model The model layer contains the representing data and is independent from the other Layers

View The view is the presentation layer. It represents the data and interacts with the user *(User Interface)*. Therefor it knows about the model and controller. But the *view* does not process the data, it will be informed by the controller if data is changed and the reload it from the model.

Controller The controller layer manages the presentations and the user interactions. The controller itself does not modify the data, but it transcript the changes on the model done in the presentation layer.

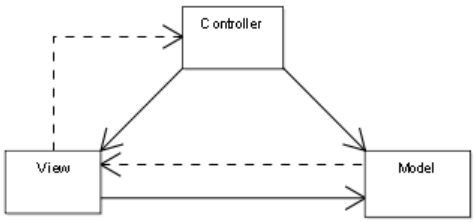

Figure 5.1: Model View Controller

5.2 Layer Description

The whole project is divided in five Packages, where obviously three of them are the *model, view and controller* packages. The other two are a package named *util*, containing some utility classes and the *heuristic* package, containing the classes implementing the heuristic and backtrack strategies.

model

This package contains all the classes which represent a *Tableaux Tree*

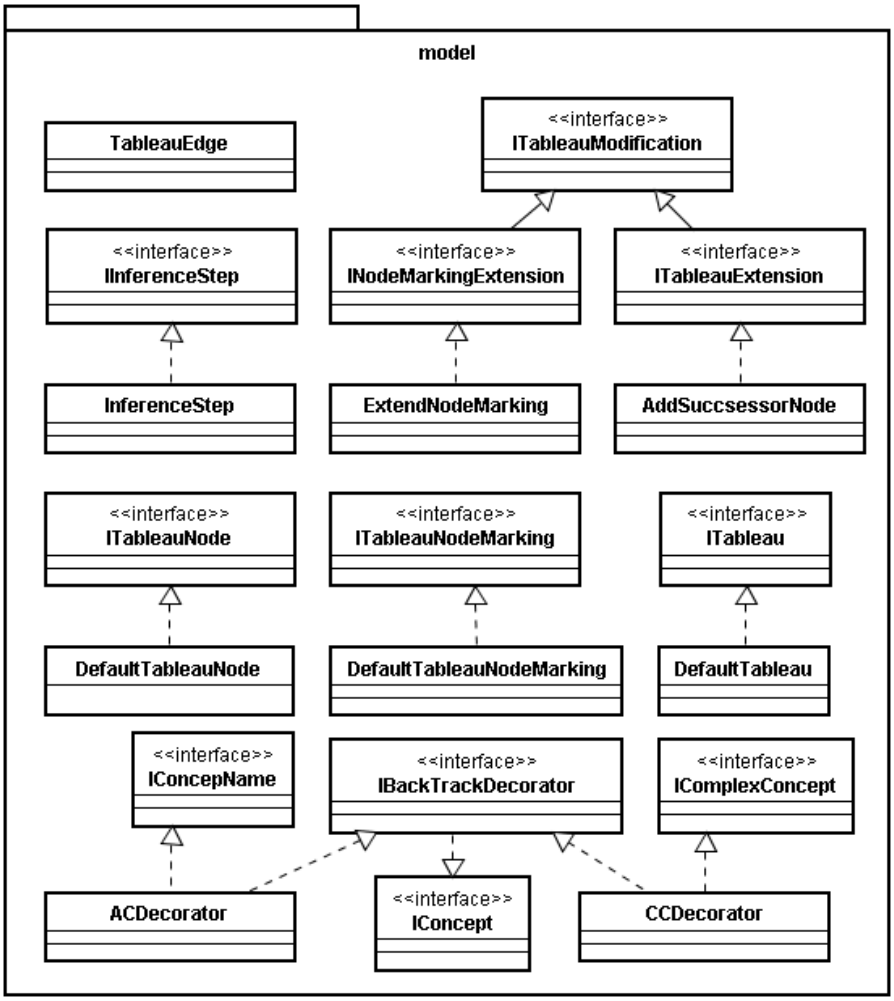

Figure 5.2: the model package

The *DefaultTableau* class is the basic class which represents a tableau tree **T**. Therefore it contains basically a set of nodes and a set of edges

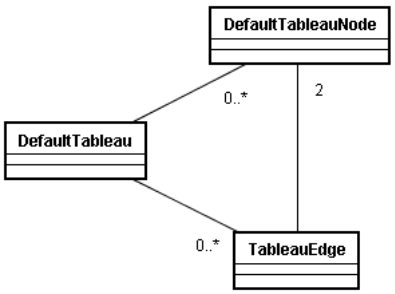

Figure 5.3: DefaultTableau class diagram

The *DefaultTableauNode* class, as a node is defined form the Tableau calculus has a Label, in this case the label is a class Named DefaultTableauMarking. Also it contains two List containing applicable inference steps and steps which the algorithm shouldn't take any more, named The DefaultTableauMarking consist of a set of Concepts implementing the IConcept Interface *(concept structure used from the Bachelor Thesis of Markus Ruepp) inferenceList* and *neverAgain*

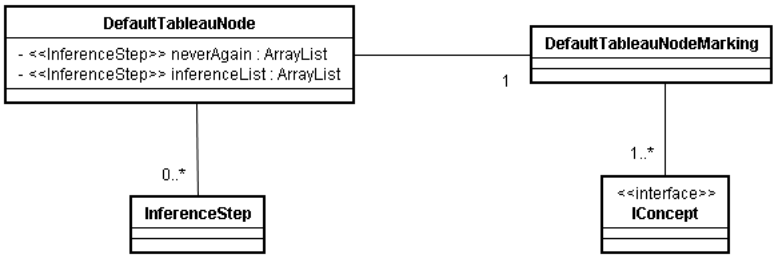

Figure 5.4: DefaultTableauNode class diagram

An InferenceStep represents the possible modifications on a Tableau tree caused by an inferencerule of the tableau calculus. It contains a targetnode where the changes ar performed and a set of modifcations where each one describes one single Modification *(eg. for every affected node of an* $\forall$ *restriction there is a single instance of ExtendNodeMarking).* With the enum *InferenceRule* the type of the inference step is set *(union, intersection, universal-, existential role restriction)*

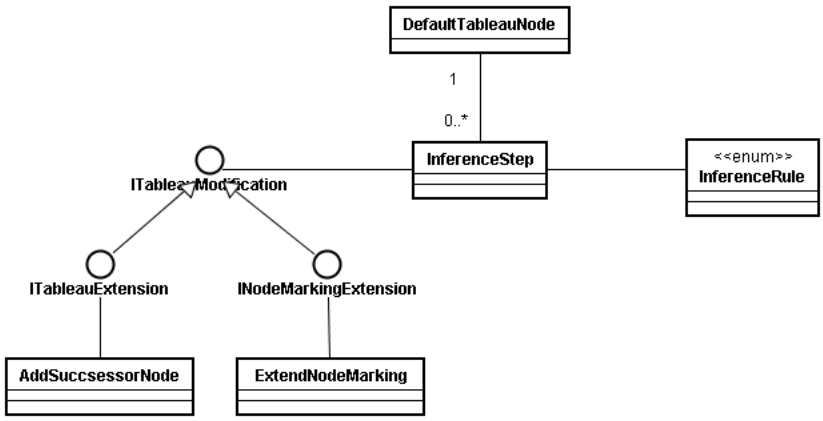

Figure 5.5: InferenceStep class diagram

view

The view Package contains the responsible classes for the visualization of the project. All graphical elements where programmed in SWT/Jface. As shown in

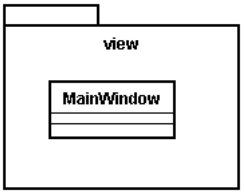

Figure 5.6: the view package

the package diagram the view consists only of one class, the *MainWindow* The

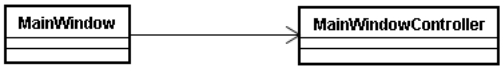

Figure 5.7: MainWindow class diagramm

controlling functions for the MainWindow where done in the MainWindowController, which will be described later in the controller section. The MainWindow consist of one frame with an in- and outputfield. In the upper textfield the concepts to prove where inserted. In the lower textfield the results such as *needed steps, needed undoings, tree* $\boldsymbol{T}$ *and calculation time* are shown. Also the optimisations are act- end deactivable through the MainWindow and it offers the

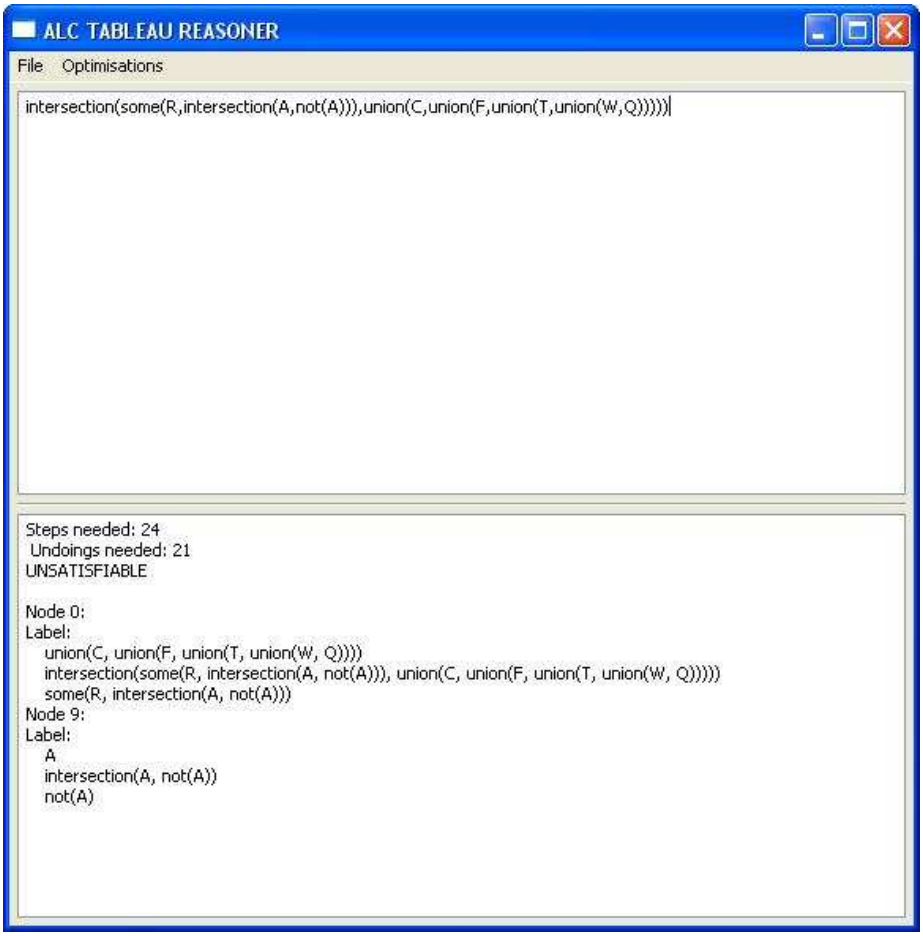

Figure 5.8: The MainWindow

possibility to load concepts from textfiles and to export a found Tableau Model into a xml file.

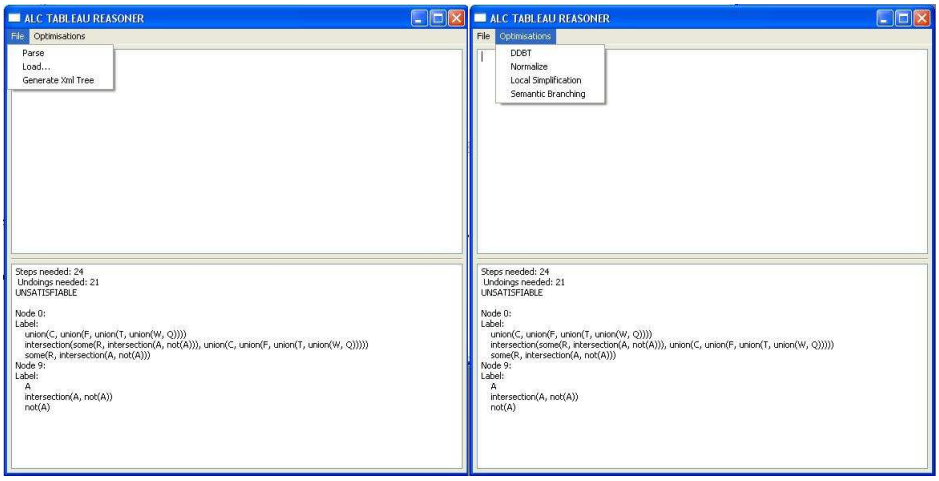

Figure 5.9: Functions activable through the MainWindow

controller

The *controller* package consists of the *MainWindow, TableauProverAdapter* and *Prover* class. The TableauProverAdapter implements an Interface which is also used by other projects and it guaranties that all provers and sat-solvers can be used in the same way. The *Prover* class contains the tableau calculus functions and manipulates a tableau tree, represented in a DefaultTableau.

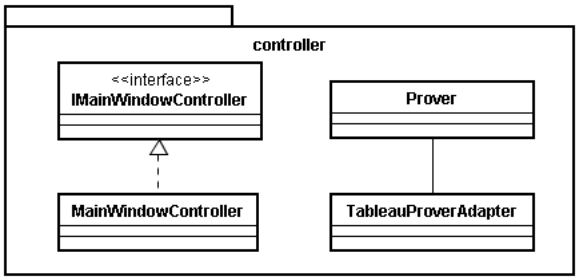

Figure 5.10: the controller package

In the following figure it is evident, that the *TableauProverAdapter* simply follows the Adapter-Pattern and wraps the Prover class. So that the satisfiability testing can be done by one method call. The *MainWindowController* performs the actions triggered in the view. Also the *MainWindowController* contains a Prover instance. The single settings which are selectable in the *MainWindow* The *MainWindowController* sets in on the Prover.

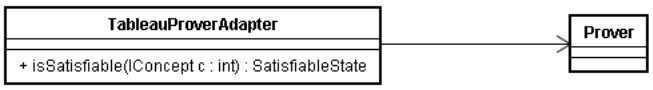

Figure 5.11: The Adapter

heuristic

In the heuristic package, the different heuristic and backtrack strategies are implemented. Every heuristic or backtrack startegy has to extend the *AbstractHeuristic* or *AbstractBacktrack* class to guarantee extensibility. Both the

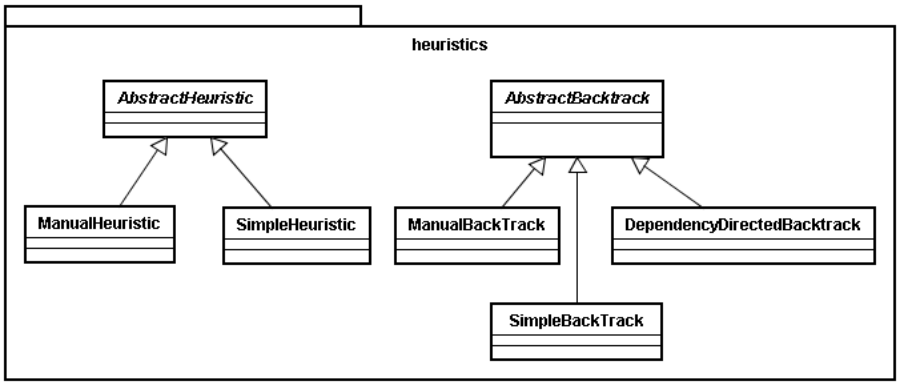

Figure 5.12: the controller package

abstract Heuristic and Backtrack classes are getting the needed data through the Prover class. The Heuristic needs the applicable inferencesteps and the BackTrack implementation needs the history of already executed inferencesteps, which all is provided by the prover.

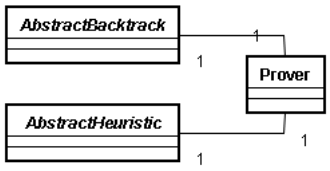

Figure 5.13: the controller package

5.3 External libraries

eclipse SWT The Standard Widget Toolkit (SWT) [5]is a graphical widget toolkit for use with the Java platform. In opposite to AWT [1] and Swing it uses the native GUI libraries of the running operating system. All GUI's in this project where implemented in SWT.

jGraphT JGraphT [4] is a free Java graph library that provides mathematical graph-theory objects and algorithms. JGraphT supports various types of graphs including:

- directed and undirected graphs.
- graphs with weighted / unweighted / labeled or any user-defined edges.
- various edge multiplicity options, including: simple-graphs, multi-graphs, pseudographs.
- unmodifiable graphs - allow modules to provide ẗread-onlyäccess to internal graphs.
- listenable graphs - allow external listeners to track modification events.
- subgraphs graphs that are auto-updating subgraph views on other graphs.
- all compositions of above graphs.

For the project a *DirectedGraph* from the jGraphT library was used to represent a tree containing the *TableauEdges* and *TableauNodes*

5.4 Optimizations

In this section the way of implement the different optimizations will be explained

5.4.1 Local Simplification

For this optimization each concept-union of a *TableauNodeMarking* is checked by the LocalSimplification Method. If an union contains a disjunct which complement contained also in the marking, the LocalSimplification builds an InferenceStep containing the disjunct. This InferenceStep is added to the neverAgain list, which contains all the disjuncts which the Tableau calculus shouldn't take any more. So there is no possibility that the standard Tableau calculus takes this branch who obviously would lead to a clash.

5.4.2 Semantic Branching

The *Semantic Branching* is realized by adding the Complement of a single disjunct to the *TableauNodeMarking* that leads to a clash. So if the next time the same Concept as within the disjunct is added to the TableauNodeMarking there is no need to unfold it anymore to identificate a clash

5.4.3 Normalization an Simplification

The *Normalization an Simplification* functions are implemented as recursive functions such as shown in fig. 3.1 and 3.2. This set of functions is applied on the initial concept expression during the initialisation of the tableau *(before the tableau algorithm starts)*. Then the calculus starts with the already normalized and simplificated concept expression

5.4.4 Dependency Directed Backtracking

For the *DDB* there was a need to add to a concept some more information, so the concepts had to be decorated using the *Decorator Pattern*

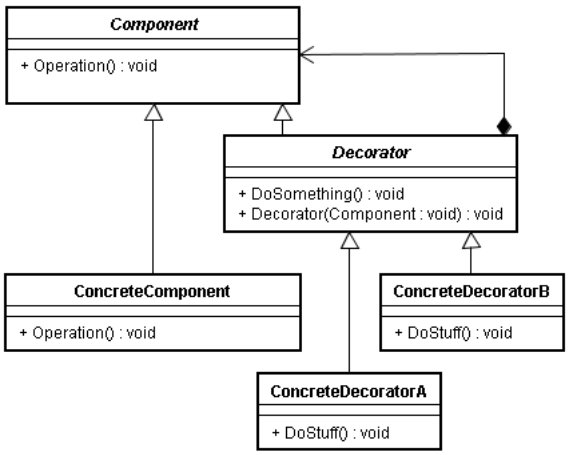

Figure 5.14: The Decorator Pattern

With this pattern it is possible now to add dependencies to a concept without changing the tableau algorithm. The dependencies of a concept initially where set to $\emptyset$. Now during the exploration of *InferenceSteps* every concept which is added to a *TableauModification* will be decorated with the dependencies of its generating concept. If a new disjunction is discovered as InferenceStep in addition to the dependencies of the generating concept, the branching point is added to the dependency set. Now if a clash is discovered it is possible to retrieve the two clashing concepts and the union of their dependencies. So the BackTrack algorithm can track back through the dependencies returned by the clashing concepts.

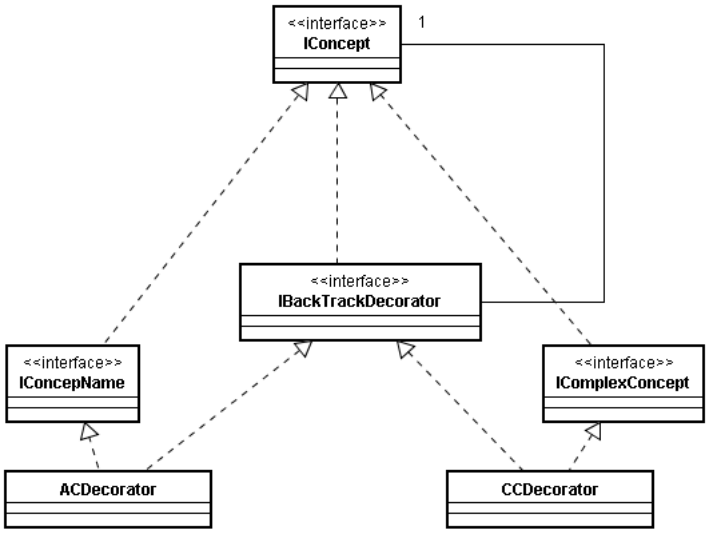

Figure 5.15: The implemented decorator pattern in the project

Chapter 6

Evaluation

6.1 Problem Generator with automated evaluation

Evaluating a reasoner would be a huge amount of work, if all combinations of optimizations with all problem sizes would be done by hand. Therefore, the idea of an automated problem generator with integrated testing / evaluation function for the implemented tableau reasoner was obvious.

So the ProblemGenerator class takes over three parts of the evaluation work

- Generating Problems with the same pattern and different sizes
- Feeding the reasoner with the generated problems and solving them with the different variations of enabled or disabled optimizations.
- Assembling all the results in a human readable form such as HTML or LaTeX

6.2 Results of Evaluation

In this section the results of the evaluation will be displayed in tables and diagrams, but first some abbreviations need to be cleared

DDBT Dependency Directed Backtracking

Local Simp Local Simplification

Sem Bra Semantic Branching

Config. 1 All optimizations disabled

Config. 2 Normalizing enabled

Config. 3 Semantic Branching and Local Simplification enabled

Config. 4 Normalizing, Semantic Branching and Local Simplification enabled

Config. 5 Dependency Directed Backtracking enabled

Config. 6 Dependency Directed Backtracking, Semantic Branching and Local Simplification enabled

Config. 7 All optimizations enabled

6.2.1 Satisfiable Problems

Basically the problems looks like:

union(intersection(union(some(R, intersection(A, not(A))), union(C, union(F, union(T, union(W, Q))))), intersection(Sat, Sat)), intersection(intersection(A, A), intersection(A, not(A))))

There is one satisfiable disjunct and the other disjuncts are all unsatisfiable. By increasing the problem size, the problem is unified with another problem created by the pattern.

Size	DDBT	Normalize	Local Simp	Sem Bra	Result	Time[sec]	Steps	Backtracks
30	no	no	no	no	Sat	3.11	97	33
35	no	no	no	no	Sat	2.155	95	38
40	no	no	no	no	Sat	4.12	119	43
45	no	no	no	no	Sat	6.696	136	48
50	no	no	no	no	Sat	8.513	137	53
55	no	no	no	no	Sat	12.464	151	58
60	no	no	no	no	Sat	20.507	187	63
30	no	yes	no	no	Sat	0.712	66	2
35	no	yes	no	no	Sat	1.225	71	2
40	no	yes	no	no	Sat	1.871	80	2
45	no	yes	no	no	Sat	2.452	67	2
50	no	yes	no	no	Sat	4.111	93	2
55	no	yes	no	no	Sat	5.512	95	2
60	no	yes	no	no	Sat	7.044	99	2
30	no	no	yes	yes	Sat	1.338	80	33
35	no	no	yes	yes	Sat	1.596	81	38
40	no	no	yes	yes	Sat	3.219	100	43
45	no	no	yes	yes	Sat	8.191	138	48
50	no	no	yes	yes	Sat	12.456	156	53
55	no	no	yes	yes	Sat	8.551	124	58
60	no	no	yes	yes	Sat	16.482	153	63
30	no	yes	yes	yes	Sat	0.632	48	2
35	no	yes	yes	yes	Sat	1.23	64	2
40	no	yes	yes	yes	Sat	1.539	52	2
45	no	yes	yes	yes	Sat	3.099	81	2
50	no	yes	yes	yes	Sat	4.699	91	2
55	no	yes	yes	yes	Sat	6.527	109	2
60	no	yes	yes	yes	Sat	7.624	93	2
30	yes	no	no	no	Sat	1.1	74	33
35	yes	no	no	no	Sat	2.866	114	38
40	yes	no	no	no	Sat	4.996	128	43
45	yes	no	no	no	Sat	5.554	120	48
50	yes	no	no	no	Sat	9.506	143	53
55	yes	no	no	no	Sat	14.804	165	58
60	yes	no	no	no	Sat	19.227	172	63
30	yes	no	yes	yes	Sat	2.017	94	33
35	yes	no	yes	yes	Sat	1.916	85	38
40	yes	no	yes	yes	Sat	4.772	112	43
45	yes	no	yes	yes	Sat	8.885	139	48
50	yes	no	yes	yes	Sat	11.004	142	53
55	yes	no	yes	yes	Sat	13.601	145	58
60	yes	no	yes	yes	Sat	17.385	153	63
30	yes	yes	yes	yes	Sat	0.814	59	2
35	yes	yes	yes	yes	Sat	1.158	57	2
40	yes	yes	yes	yes	Sat	2.2	76	2
45	yes	yes	yes	yes	Sat	3.441	92	2
50	yes	yes	yes	yes	Sat	4.806	96	2
55	yes	yes	yes	yes	Sat	6.201	95	2
60	yes	yes	yes	yes	Sat	6.767	71	2

Table 6.1: Satisfiable reasoning results with mixed optimizing configurations

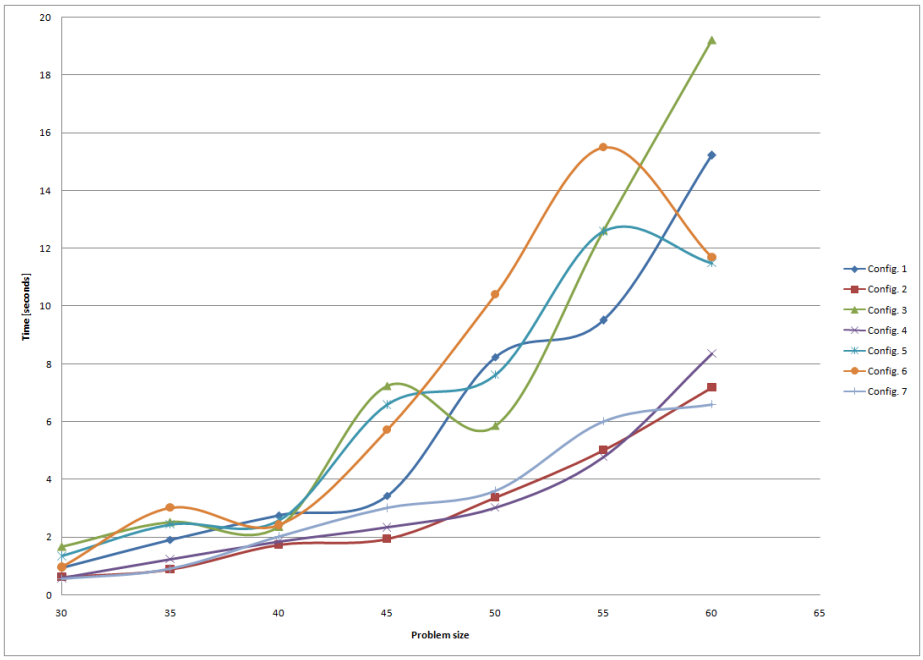

Figure 6.1: Calculation time - Problem size diagram

As in figure 6.1 shown, we receive the best performances in this optimization configurations, where normalizing is enabled (Config. 2, Config. 4 and Config. 7). The normalizing process, if we take this part from the basic problem $some(R, intersection(A, not(A)))$ and remember the normalizing rules of figure 4.1 and 4.2 on page 11 and 12 we see that the normalizing function can declare the whole function as $\perp$, which brings the effort that at this point thanks to the normalizing function, no additionally node that later on has to be backtracked will be added. So this optimization helps ternary (Steps, Backtrackings, and space because no extension of the tableau needed). Config. 3 achieves the highest computation time for the biggest problem size of all configurations. This results from the fact that the Local Simplification and the Semantic Branching bring no effort for the basic example, so these two optimizations in this case just produce computing overhead.

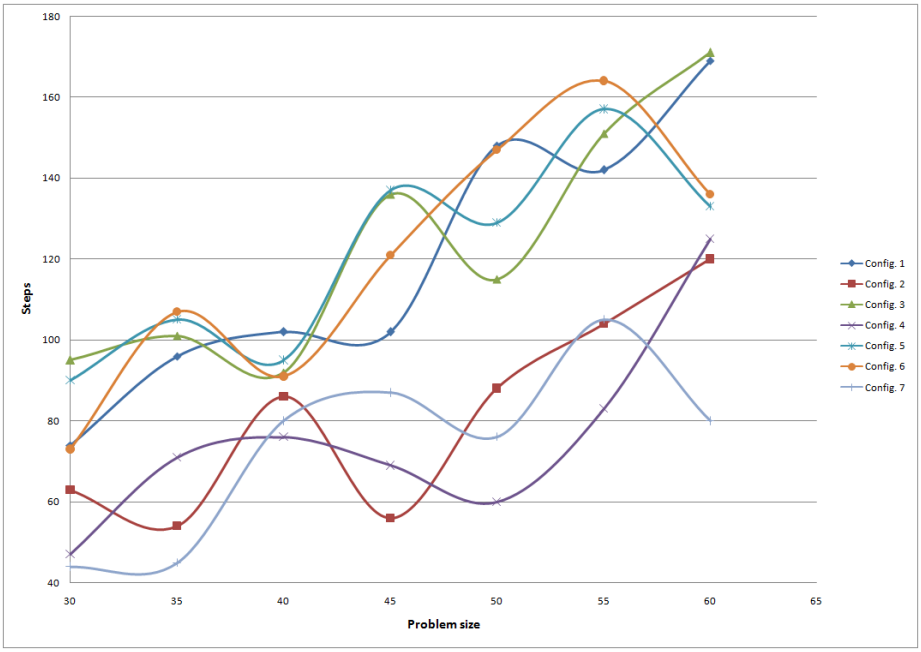

Figure 6.2: Steps - Problem size diagram

In figure 6.2 we see how the amount of executed steps increases by increasing the problem size. There are, like in figure 6.1, seven combinations of optimization configurations. Also in this diagram are two groups noticeable and again these with enabled normalizing have got the best effort and in this case the least amount of steps.

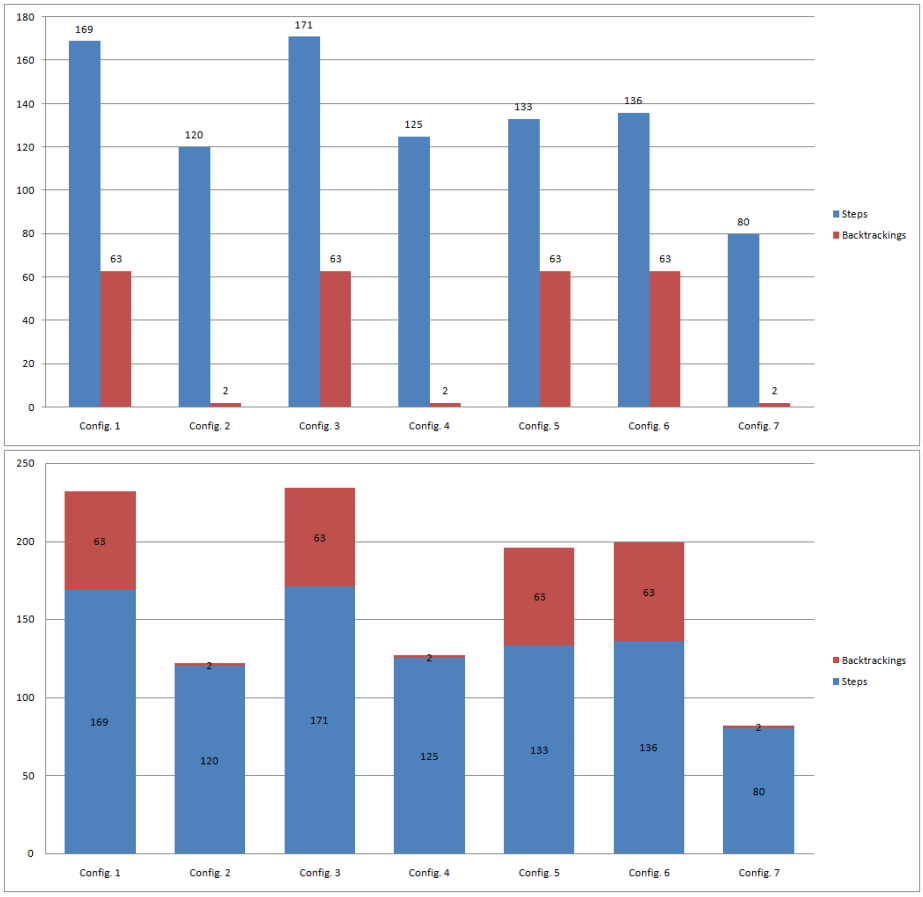

Figure 6.3: Backtrackings in comparison to steps with problem size 60

Figure 6.3 confirms that for this kind of problems the normalizing got the best effort, because as we see also the amount of backtrackings is decreased on the configurations with normalizing enabled.

6.2.2 Unsatisfiable Problems

Basically the problems looks like:

union(intersection(A, not(A)), intersection(intersection(A, not(A)), some(R0, intersection(intersection(union(A, not(B)), intersection(union(union(intersection(not(A), not(B)), intersection(C, not(D))), not(B)), union(union(not(C), D), not(A)))), B))))

There is a union where all disjuncts are unsatisfiable, by increasing the problem size, the problem is unified by another problem created by the pattern.

Size	DDBT	Normalize	Local Simp	Sem Bra	Result	Time[sec]	Steps	Backtracks
30	no	no	no	no	UnSat	0.517	121	121
35	no	no	no	no	UnSat	0.769	141	141
40	no	no	no	no	UnSat	1.267	161	161
45	no	no	no	no	UnSat	1.887	181	181
50	no	no	no	no	UnSat	2.771	201	201
55	no	no	no	no	UnSat	3.716	221	221
60	no	no	no	no	UnSat	5.001	241	241
30	no	yes	no	no	UnSat	0.048	31	2
35	no	yes	no	no	UnSat	0.066	36	2
40	no	yes	no	no	UnSat	0.116	41	2
45	no	yes	no	no	UnSat	0.17	46	2
50	no	yes	no	no	UnSat	0.228	51	2
55	no	yes	no	no	UnSat	0.33	56	2
60	no	yes	no	no	UnSat	0.441	61	2
30	no	no	yes	yes	UnSat	0.801	121	121
35	no	no	yes	yes	UnSat	1.293	141	141
40	no	no	yes	yes	UnSat	2.028	161	161
45	no	no	yes	yes	UnSat	2.911	181	181
50	no	no	yes	yes	UnSat	4.039	201	201
55	no	no	yes	yes	UnSat	5.402	221	221
60	no	no	yes	yes	UnSat	7.232	241	241
30	no	yes	yes	yes	UnSat	0.181	31	31
35	no	yes	yes	yes	UnSat	0.3	36	36
40	no	yes	yes	yes	UnSat	0.455	41	41
45	no	yes	yes	yes	UnSat	0.694	46	46
50	no	yes	yes	yes	UnSat	0.975	51	51
55	no	yes	yes	yes	UnSat	1.416	56	56
60	no	yes	yes	yes	UnSat	1.894	61	61
30	yes	no	no	no	UnSat	0.0040	7	7
35	yes	no	no	no	UnSat	0.0030	7	7
40	yes	no	no	no	UnSat	0.0030	7	7
45	yes	no	no	no	UnSat	0.0040	7	7
50	yes	no	no	no	UnSat	0.0050	7	7
55	yes	no	no	no	UnSat	0.0050	7	7
60	yes	no	no	no	UnSat	0.0060	7	7
30	yes	no	yes	yes	UnSat	0.0040	7	7
35	yes	no	yes	yes	UnSat	0.0030	7	7
40	yes	no	yes	yes	UnSat	0.0030	7	7
45	yes	no	yes	yes	UnSat	0.0050	7	7
50	yes	no	yes	yes	UnSat	0.0060	7	7
55	yes	no	yes	yes	UnSat	0.0090	7	7
60	yes	no	yes	yes	UnSat	0.0090	7	7
30	yes	yes	yes	yes	UnSat	0.065	31	2
35	yes	yes	yes	yes	UnSat	0.108	36	2
40	yes	yes	yes	yes	UnSat	0.151	41	2
45	yes	yes	yes	yes	UnSat	0.231	46	2
50	yes	yes	yes	yes	UnSat	0.32	51	2
55	yes	yes	yes	yes	UnSat	0.44	56	2
60	yes	yes	yes	yes	UnSat	0.602	61	2

Table 6.2: Unsatisfiable reasoning results with mixed optimizing configurations

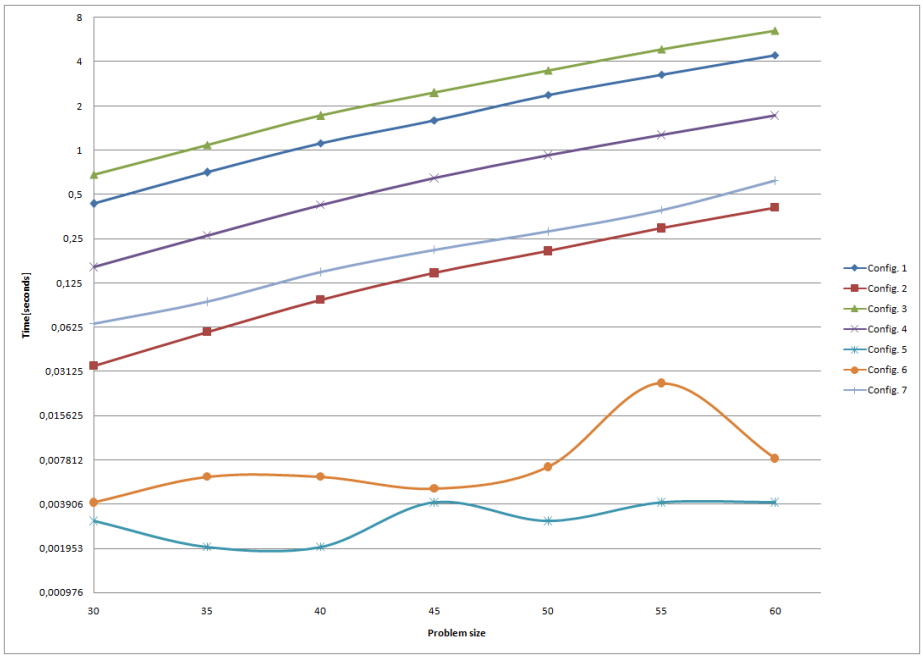

Figure 6.4: Calculation time - Problem size diagram

As in figure 6.4 noticeable, the least time consuming configuration is Config. 5 (only DDBT enabled) followed by Config. 6 (DDBT and Semantic Branching with Local Simplification enabled) then comes Config. 2(only Normalizing enabled) but with a calculation time increased by the factor of 10. Semantic Branching with Local Simplification needs the most calculation time because it brings no effort in this case, so it produces only a calculation overhead and needs even more calculation time as with disabled optimizations.

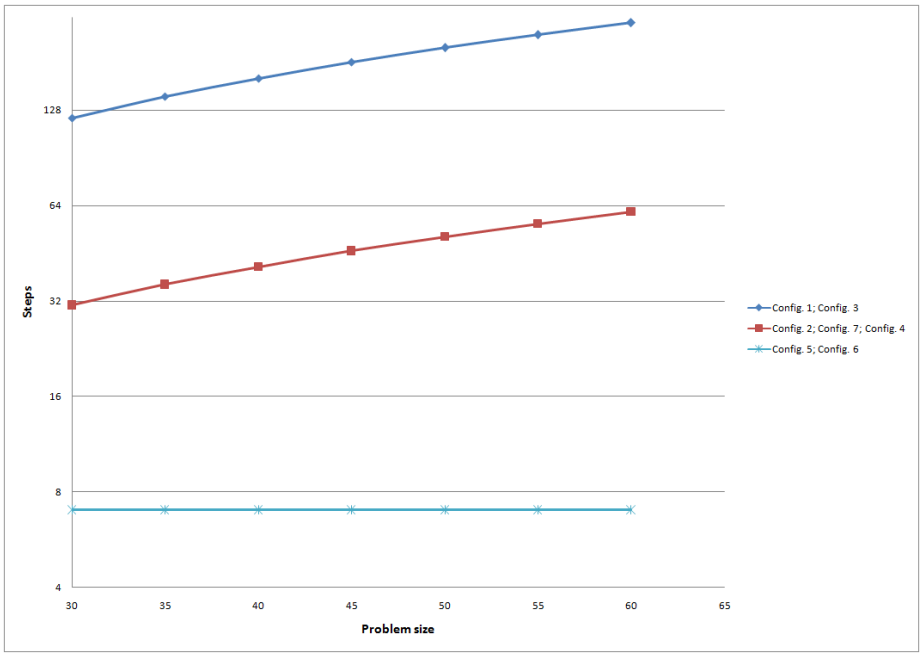

Figure 6.5: Steps - Problem size diagram

Figure 6.5 confirms the conclusions made for figure 6.4. If DDBT is enabled there are uniformly 7 steps needed to solve the problem, so Config. 5 needs the least amount of steps, also Config. 6 needs uniformly 7 step for solving the problem. On second place there are all combinations with enabled normalization but it needs four to eight times the steps which DDBT needs. Again far behind there is Config. 1(no optimizations) and 3(Local Simplification with Semantic Branching) with 16 to nearly 32 times of needed steps in comparative to Config. 5 and 6.

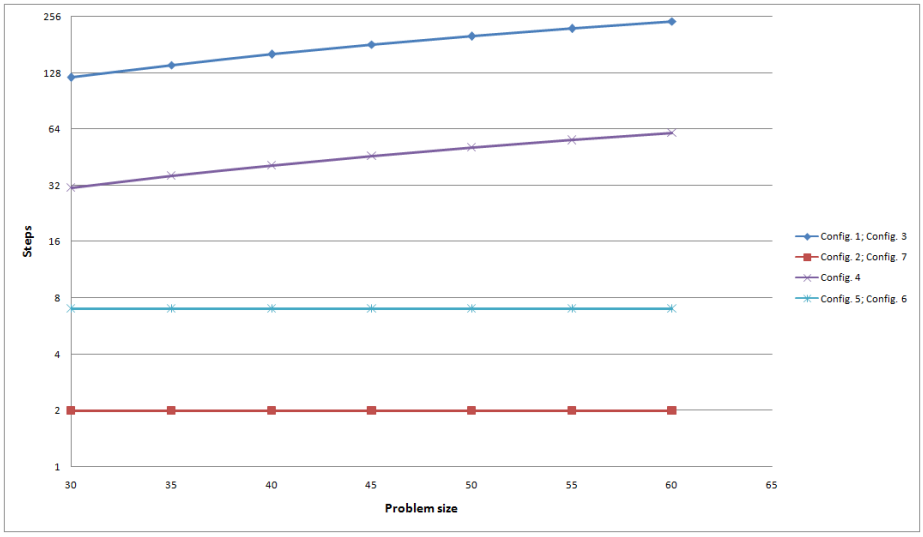

Figure 6.6: Backtrackings - Problem size diagram

Figure 6.6 shows, that Normalizing brings the least amount of backtracking steps with it, Followed by DDBT. Also here Semantic Branching with Local Simplification did not gathered any improvement.

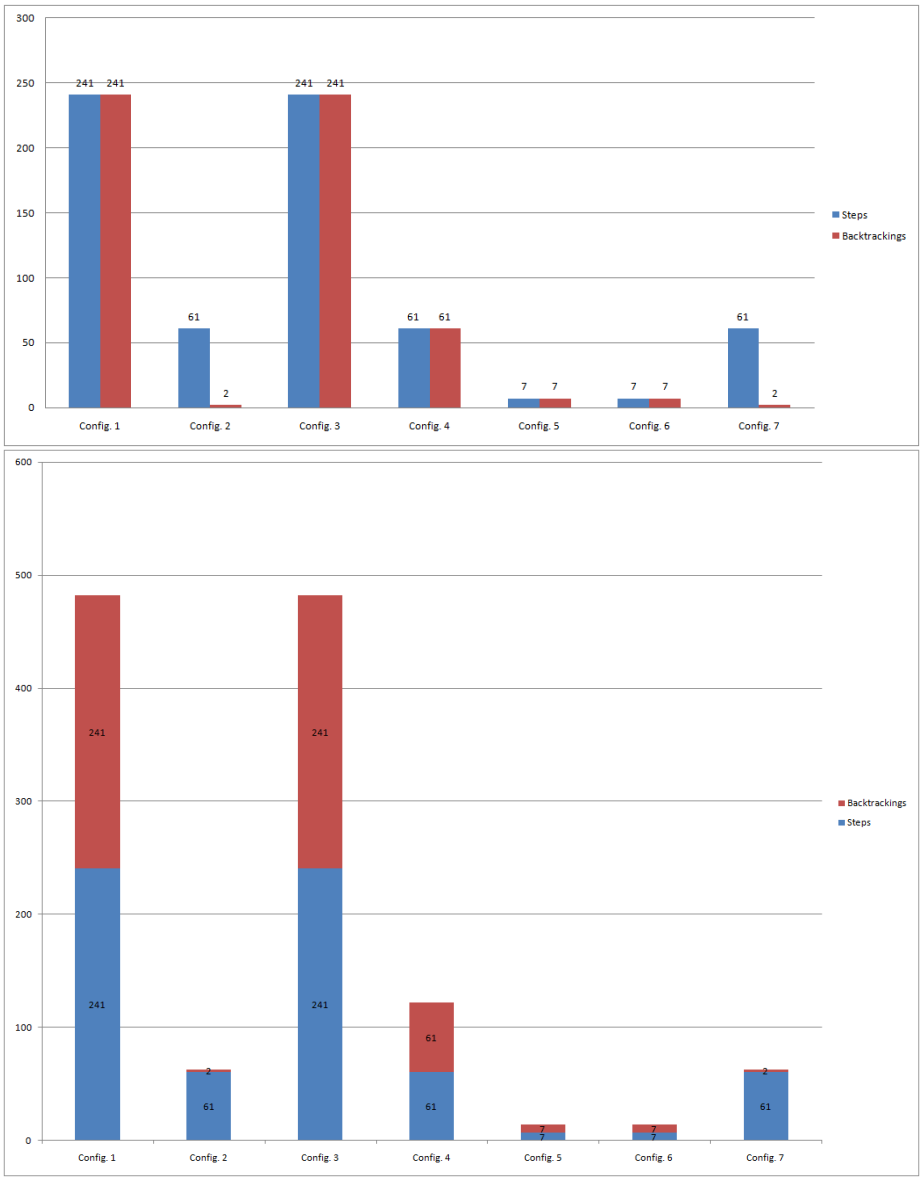

Figure 6.7: Backtrackings in comparison to steps with problem size 60

Looking at figure 6.7 shows why DDBT is the fastest optimization for this kind of problems. It got more backtrackings than normalization, but a lot less needed steps. Which makes DDBT to the winning strategy for the treated problems.

6.3 Conclusion

After the evaluation two optimizations brought the best performances Normalizing and Dependency Directed Backtracking. Local Simplification and Semantic Branching were not so helpful for this kind of problems, which does not mean that these two optimizations are generally not helpful. But during the evaluation it became evident, that even if Local Simplification and Semantic Branching bring no effort they've still got a calculation overhead. So finding the "right" optimization configuration isn't easy as it sounds at first, like there is no "perfect optimization" for every problem. So it depends on the situation, or more on the composition of the problems which optimization or optimization configuration is the best. As seen in the evaluation DDBT and Normalization haven't got such a calculation overhead like Semantic Branching with Local Simplification, but always a speed improvement. Therefore it seems reasonable to handle all problems with Normalizing and DDBT and to decide case by case if Local Simplification and Semantic Branching should be applied or not.

Chapter 7

Conclusion

7.1 Potential extensions

A potential extension for this project would be a visualization of found models. There exists already an implementation which takes as input JGraphT [4] models, its name is JGraph [3]. Also a possible approach could be the XML [6] output of the implemented reasoner, which gives out the found model in a xml structure. Here a little sample output how the xml structure looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Tableau>
    <Nodes>
        <Node>
            <Name>1</Name>
            <Marking>
            [union(C, union(F, union(T, union(W, Q)))),
            intersection(some(R, intersection(A, not(A))),
            union(C, union(F, union(T, union(W, Q))))),
            some(R, intersection(A, not(A)))]
            </Marking>
        </Node>
        <Node>
            <Name>10</Name>
            <Marking>[A, intersection(A, not(A)), not(A)]</Marking>
        </Node>
    </Nodes>
    <Edges>
        <Edge>
            <Label>R</Label>
            <Source>1</Source>
            <Drain>10</Drain>
        </Edge>
```

```
    </Edges>
</Tableau>
```

Both approaches have their advantages. The faster one would be to use JGraph, because it takes as input the already existent graph structure of JGraphT and builds upon it the visualization. Developing an own visualization upon the xml output has the beneficial to get an independent and personalized solution, but would be connected with great effort.

List of Figures

List of Tables

Bibliography

[1] *AWT - http://java.sun.com/products/jdk/awt/*, January 2009.

[2] *JAVA - http://java.sun.com/*, January 2009.

[3] *JGraph - http://www.jgraph.com/*, January 2009.

[4] *JGraphT - http://jgrapht.sourceforge.net/visualizations.html*, January 2009.

[5] *SWT - http://www.eclipse.org/swt/*, January 2009.

[6] *XML - http://www.w3.org/XML/*, January 2009.

[7] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. 2000.

[8] I. Horrocks. Implementation and optimisation techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter 9, pages 306–346. Cambridge University Press, 2003.

[9] Manfred Schmidt-Schaubßand Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.