University of Innsbruck
Semantic Technology Institute

Exploring Propositional Local-Search Procedures for Tableau-based Reasoning in the Description Logic $\mathcal{ALC}$

Bachelor Thesis

Adrian Marte

A-6020 Innsbruck
Matrikelnr.: 0315038

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

Innsbruck, June 29, 2008

Abstract

Description Logics are a popular family of formally well-founded and decidable knowledge representation languages used in various areas of computer science. For example, Description Logics are the basis of ontologies used in the context of the Semantic Web. With Description Logics it is possible to automatically reason about the knowledge represented. Many efficient techniques have been developed to achieve this task. The most successful and most widely used technique are tableau-based approaches.

In this thesis we give an overview of a novel tableau-based reasoning procedure for Description Logics, i.e. *SAT-Tableau Calculus* [14], that combines standard tableau reasoning techniques with propositional satisfiability solvers. We then evaluate the viability of local-search-based propositional satisfiability solvers within the *SAT-Tableau Calculus*.

Contents

Chapter 1

Introduction

This chapter provides an introduction to Description Logics and gives an overview of the most important reasoning problems occurring in the context of Description Logics.

1.1 Description Logics

Description Logics (DL) are a family of *knowledge representation* languages which can be used to represent the terminological knowledge of a domain in a structured and formally well-understood way. The important notions of a domain are described by concept descriptions, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors of the particular Description Logic. In general, a concept denotes the set of individuals that belongs to it, and a role denotes a relationship between concepts.

Typical Description Logics provide the Boolean constructors *conjunction* ($\sqcap$), which is interpreted as set intersection, *disjunction* ($\sqcup$), which is interpreted as set union, and *negation* ($\neg$), which is interpreted as set complement, as well as the *existential restriction* constructor ($\exists r.C$), and the

value restriction constructor ($\forall r.C$). In section 2.1 we will give an overview (including syntax and semantics) of the basic Description Logic $\mathcal{ALC}$ which provides all these boolean constructors.

Assume that we want to define the concept of "A parent whose children are all computer scientists." This concept can be described in the Description Logic $\mathcal{ALC}$ with the following concept description:

$$Human \sqcap \forall hasChild.ComputerScientist.$$

A DL knowledge base typically comes in two parts: a terminogical and an assertional one. In the terminogical part, called the TBox, we can describe the relevant notions of an application domain by stating properties of concepts and roles, and relationships between them. In other words, a TBox represent schema-level knowledge which interrelates concepts, i.e. characterstic properties of sets of individuals. A simple TBox allows statements that introduce a name (abbreviation) for a complex description. For instance, the following concept definition introduces HappyParent as a name abbreviation for the previous example:

$$HappyParent \equiv Human \sqcap \forall hasChild.ComputerScientist$$

More expressive TBoxes allow the statement of more general axioms, i.e. statements that constraint the way in which concepts and roles can be interpreted. For instance,

$$\exists hasChild.Human \sqsubseteq Human$$

says that only humans can have human children.

The assertional part of the knowledge base, called the ABox, is used to describe a concrete situation by stating properties and individuals. For

example, the assertions

$$Human(\text{DANIEL}), hasChild(\text{DANIEL}, \text{JOANA})$$

state that DANIEL and JOANA are instances of the concept $Human$ and that JOANA is one of DANIEL's children.

In contrast to ABox statements which describe a specific situation (or state), TBox statements apply to any possible situation or state that is represented by the knowledge base.

In the Semantic Web Description Logics have an important role, since they are the foundation for knowledge representation systems such as ontology languages. For instance, the OWL-DL respectively OWL-Lite sub-languages of the W3C-endorsed Web Ontology Language (OWL) correspond to the Description Logic $\mathcal{SHIN}$ (D) respectively $\mathcal{SHOIN}$ (D). Therefore, the previous example can be expressed in OWL in a straight-forward way (a concept in DL jargon is referred to as a class in OWL, a role in DL jargon is a property in OWL):

```
<owl:Class rdf:id=''HappyParent''>
  <owl:intersectionOf rdf:type=''Collection''>
    <owl:Class rdf:about=''Human''>
    <owl:Restriction>
      <owl:onProperty rdf:resource=''#hasChild'' />
      <owl:allValuesFrom rdf:resource=''#ComputerScientist'' />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

The Description Logic $\mathcal{ALC}$ that is considered in this thesis is a sublanguage of the DLs underlying the OWL ontology languages.

1.2 Reasoning in Description Logics

The main advantage of using formal languages for knowledge representation is that it is possible to automatically reason about the knowledge. In the case of Description Logics, many efficient techniques have been developed to achieve this task, e.g., the tableau algorithm which we will discuss in section 2.2.

In general, weak Description Logics have efficient reasoning algorithms, but are often not expressive enough. Strong Description Logics are expressive, but do not have efficient algorithms. Therefore, there is always a trade-off between the expressive power and the computational complexity of a Description Logic.

The most important inference problems for Description Logics are:

- **Consistency:** Check whether a concept is meaningful: $\neg C \equiv \bot$?
- **Subsumption:** Check whether a concept is more general than another concept: $C \sqsubseteq D$?
- **Equivalence:** Check whether two concepts are equivalent: $C \equiv D$?
- **Membership:** Check whether an individual i is an instance of a concept C: $i \in C$?

For a DL providing all the Boolean operators, such as $\mathcal{ALC}$, all of the above reasoning problems can be reduced to consistency checking. For example, a concept C is subsumed by a concept D, i.e. $C \sqsubseteq D$, if and only if $C \sqcap \neg D$ is not consistent. Therefore, by providing an efficient sound and complete satisfiability checking procedure, all these reasoning problems can be solved efficiently.

1.3 Goal Of The Project

In this thesis we give an overview of a novel Description Logic reasoning procedure, i.e. *SAT-Tableau Calculus*, that combines standard tableau reasoning techniques with propositional satisfiability solvers. We considered the specific case of local search procedures used as the SAT solver component in the SAT-Tableau calculus. For this we had to overcome the problem of dealing with unsatisfiable propositional subproblems that can occur even when dealing with (under DL-semantics) satisfiable input concepts. We implemented a range of SAT solving algorithms that belong to the class of Local Search Procedures and evaluated the viability of using these solvers for DL reasoning within the SAT-Tableau framework.

Chapter 2

Preliminaries

In this chapter we give an overview of the description logic $\mathcal{ALC}$ and briefly introduce the tableau algorithm, a widely used and successful reasoning procedure for Description Logics.

2.1 Description Logic $\mathcal{ALC}$

In this section we give an introduction to the basic Description Logic $\mathcal{ALC}$ including syntax and semantics. The name $\mathcal{ALC}$ stands for "Attributive concept language with Complements." It was first introduced by Schmidt-Schauß and Smolka in [17].

Definition 1 ($\mathcal{ALC}$ syntax)**.** *Let N_C be a non-empty, countable set of concept names and N_R be a countable set of role names such that N_C and N_R are pairwise disjoint. Then, we call $\Sigma = (N_C, N_R)$ a signature. The set of $\mathcal{ALC}$ concept descriptions is the smallest set such that*

- *$\top$, $\bot$, and every concept name $A \in N_C$ is an $\mathcal{ALC}$ concept,*
- *If C and D are $\mathcal{ALC}$ concepts and $r \in N_R$, then $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall r.C$, and $\exists r.C$ are $\mathcal{ALC}$ concepts.*

DLs can be equipped with a set-theoretic semantics. The concept of and interpretation of a signature is introduced to assign meaning to the description language.

Definition 2 ($\mathcal{ALC}$ semantics)**.** *Let $\Sigma = (N_C, N_R)$ be a signature. A Σ-interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$, called the domain of $\mathcal{I}$, and a function $\cdot^{\mathcal{I}}$ that maps every $\mathcal{ALC}$ concept $C \in N_C$ to a subset of $\Delta^{\mathcal{I}}$, and every role name $r \in N_R$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that, for all $\mathcal{ALC}$ concepts C, D and all role names r,*

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}}, \\
\bot^{\mathcal{I}} &= \emptyset, \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}}, \\
(C \cup D) &= C^{\mathcal{I}} \cup D^{\mathcal{I}}, \\
\neg C^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}, \\
(\exists r.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \textit{There is some } y \in \Delta^{\mathcal{I}} \textit{ with } \langle x, y\rangle \in r^{\mathcal{I}} \textit{ and } y \in C^{\mathcal{I}}\}, \\
(\forall r.c)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \textit{For all } y \in \Delta^{\mathcal{I}}, \textit{ if } \langle x, y\rangle \in r^{\mathcal{I}}, \textit{ then } y \in C^{\mathcal{I}}\}.
\end{aligned}
$$

As mentioned in section 1.1 a DL knowledge base (KB) consists of a terminological part (called TBox) and an assertional part (called ABox), each part consisting of a set of axioms. The most general form of TBox axioms are general concept inclusions.

Definition 3. *A general concept inclusion (GCI) is of the form $C \sqsubseteq D$ where C, D are $\mathcal{ALC}$ concepts. A finite set of GCIs is called a TBox. An interpretation $\mathcal{I}$ is a model of a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. $\mathcal{I}$ is a model of TBox $\mathcal{T}$ if it is a model of every GCI in $\mathcal{T}$.*

The ABox can contain two kinds of axioms, one for asserting that an individual is an instance of a given concept, and the other one for asserting that a pair of individuals is an instance of a given role.

Definition 4. *An assertional axiom is of the form $x : C$ or $(x, y) : r$, where C is an $\mathcal{ALC}$ concept, r is an $\mathcal{ALC}$ role and x and y are individual names.*

A finite set of assertional axioms is called an ABox. An interpretation $\mathcal{I}$ is a model of an assertional axiom $x : C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$ and $\mathcal{I}$ is a model of an assertional axiom $(x, y) : r$ if $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$. $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ if it is a model of every axiom in $\mathcal{A}$.

Definition 5. *A knowledge base (KB) is a pair $(\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox. An interpretation $\mathcal{I}$ is a model of a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\mathcal{I}$ is a model of $\mathcal{T}$ and $\mathcal{I}$ is a model of $\mathcal{A}$*

$\mathcal{I} \models \mathcal{K}$ respectively $\mathcal{I} \models \mathcal{T}$, $\mathcal{I} \models \mathcal{A}$, $\mathcal{I} \models a$ denotes that $\mathcal{I}$ is a model of a KB $\mathcal{K}$ respectively TBox $\mathcal{K}$, ABox $\mathcal{A}$, axiom a.

A fundamental problem in Description Logic reasoning is to prove if a concept is satisfiable, i.e. if it is possible for a concept to have instances. This problem is of major importance since other reasoning problems can be reduced to concept satisfiability (consistency), e.g. concept subsumption or concept validity.

Definition 6 (Concept Satisfiability)**.** *Let $\Sigma = (N_C, N_R)$ be a signature, $\mathcal{T}$ be a terminology over Σ and $C \in N_C$ be a concept. C is satisfiable iff there exists a Σ-interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ such that $C^{\mathcal{I}} \neq \emptyset$. C is satisfiable wrt. the terminology $\mathcal{T}$ if and only if there exists a Σ-interpretation such that $C^{\mathcal{I}} \neq \emptyset$ and $\mathcal{I} \models \mathcal{T}$.*

Deciding concept satisfiability wrt. a (general) terminology is an ExpTime-complete problem [5]. It becomes a PSpace-complete problem if background terminologies are not considered at all (i.e. $\mathcal{T} = \emptyset$) or restricted to only contain definitions of concept names without cyclic dependencies [5].

2.2 Tableau-based Reasoning for Description Logics

For Description Logics there exist various different reasoning techniques such as resolution-based approaches, automata-based approaches and structural

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

Figure 2.1: Tableau expansion rules

approaches (for sub-Boolean DLs). However, the most widely used and most successful technique is the tableau-based approach first introduced by Schmidt-Schauß and Smolka in [17]. In this section we give an overview of this technique for the case of the Description Logic $\mathcal{ALC}$.

Given an $\mathcal{ALC}$ concept description C_0, the tableau algorithm tries to construct a finite interpretation $\mathcal{I}$ that satisfies C_0, i.e. contains an element $x_0 \in \Delta^{\mathcal{I}}$ such that $x_0 \in C_0^{\mathcal{I}}$. It is assumed, that all concept descriptions are in *negation normal form* (NNF), i.e. that negation occurs only directly in front of concept names. Using De Morgan's rules and the usual rules for quantifiers, any $\mathcal{ALC}$ concept description can be transformed into an equivalent description in NNF.

The method works on a tree whose nodes are labelled with concept descriptions $\mathcal{L}(x)$ and each edge $\langle x, y\rangle$ is labelled with a set of role names $\mathcal{L}(\langle x, y\rangle)$. If $\langle x, y\rangle$ is an edge in the tree, then x is a predecessor of y (and that y is successor of x). In case $\langle x, y\rangle$ is labelled with a set containing the role name r, then x is an r-predecessor of y (and y is an r-successor of x). The main principle of tableau is to attempt to "break" a complex concept description into smaller ones until complementary pairs of literals are produced or no further expansion is possible. The procedure starts by generating a single-node tree whose root x_0 is labeled with $\mathcal{L}(x_0) = C_0$. The algorithm then applies so-called *expansion rules* (see Fig. 2.1), which syntactically decom-

pose the concepts in node labels, either inferring new constraints for a given node or extending the tree according to these constraints. The expansion rules directly reflect the semantics of the concept constructor which forms the concept expression to be decomposed. For example, if $C_1 \sqcap C_2 \in \mathcal{L}(x)$, and either $C_1 \notin \mathcal{L}(x)$ or $C_2 \notin \mathcal{L}(x)$, then the $\sqcap$-rule adds both C_1 and C_2 to $\mathcal{L}(x)$. If $\exists r.C \in \mathcal{L}(x)$ and x does not yet have an r-successor node y with $C \in \mathcal{L}(y)$, then the $\exists$-rule generates a new r-successor node y of x with $C \in \mathcal{L}(y)$. Note that the $\sqcup$-rule is different from the other rules in that it is non-deterministic: if $C_1 \sqcup C_2 \in \mathcal{L}(x)$ and neither $C_1 \in \mathcal{L}(x)$ nor $C_2 \in \mathcal{L}(x)$, then it adds *either* C_1 *or* C_2 to $\mathcal{L}(x)$. In practice this is the main source of complexity in tableau algorithms, because it may be necessary to explore all possible choices of rule applications.

If the algorithm encounters a clash, i.e. there exists a node x with $C \in \mathcal{L}(x)$ and $\neg C \in \mathcal{L}(x)$, it tries to backtrack by taking another non-deterministic choice of the possible choices of the $\sqcup$-rule. If all of these choices lead to a clash, then the algorithm answers C_0 is unsatisfiable. If the algorithm stops without having encountered a clash, then the obtained tree yields a finite representation of a model for the concept description C_0 and the algorithm answers C_0 is satisfiable.

Chapter 3

The SAT-Tableau Calculus

This chapter provides an overview of a novel procedure for Description Logic reasoning introduced by Keller and Heymans in [14]. The approach combines the power of modern propositional SAT solving techniques with the most flexible and successful inference technique for Description Logic reasoning, i.e. *tableau*-based procedures. For a more detailed description see [14]. The SAT-Tableau calculus allows to use a range of propositional SAT solving algorithms within tableau-based Description Logic reasoning.

3.1 Basic Idea

In logics, an interpretation $\mathcal{I}$ embodies a particular view on a domain under consideration. In DLs this view is formally based on set-theory, i.e. a domain is structured by means of sets and relations (over some universe of objects).

Semantically, given a particular interpretation (or view) $\mathcal{I}$, concept expressions denote sets of individuals $C^{\mathcal{I}}$ that exist and can be denoted (or described) in the domain according to $\mathcal{I}$. Now, if one considers a concept expression C wrt. a particular individual $i \in \Delta^{\mathcal{I}}$ in the domain, then C essentially becomes a *proposition* $C(i)$ about i that either is true (and

holds for i) or not. This way, when considering a concept expression, the boolean concept constructors $\sqcap, \sqcup, \neg$ in $\mathcal{ALC}$ correspond directly to the classical propositional connectives $\wedge, \vee, \neg$ if we fix an individual i for which we consider the concept expression. Hence, a concept expression C constructed from boolean concept constructor only, represents essentially a formula $C(i)$ in Propositional Logic for any individual $i \in \Delta^{\mathcal{I}}$.

However, $\mathcal{ALC}$ (as well as any other DL) extends Propositional Logics by a small set of *modal constructors*. These constructors differ from the boolean ones in the sense that semantically they are not concerned with the very properties of a single individual only, but that they take into account logical properties of *other individuals* in the domain that are related (in some way) to an individual under consideration.

In $\mathcal{ALC}$ there is essentially one such modal constructor: the *existential value restriction* ($\exists R.D$), or the corresponding dual modal constructor called *universal value restriction* ($\forall R.D$). Given an interpretation $\mathcal{I}$, for any individual $i \in \Delta^{\mathcal{I}}$ the expression $C(i) = \exists R.D(i)$ is true if and only if in $\mathcal{I}$ there is an individual $j \in \Delta^{\mathcal{I}}$ that is R-related to i such that $D(j)$ holds. In other words, the truth of C (a statement about individuals i) does not depend on the (most basic) logical properties of i in $\mathcal{I}$, but instead only on the logical properties of (certain) neighbors j in $\mathcal{I}$.

Therefore, we can distinguish two different levels of knowledge representation that are interwoven in DLs: (i) the purely propositional level (i.e. boolean constructors) concerned with properties of single individuals only and does not take into account any other individuals and (ii) the purely modal level (i.e. modal constructors) that is only concerned with how individuals relate to other individuals.

It seems straightforward, that the two different levels of knowledge representation can then be treated by two separate inference procedures: a purely propositional reasoner that that does not take into account the neighborhood of and a purely modal reasoner which essentially takes into account

the interrelation of individuals, given their logical properties.

In fact, we show below that DL reasoning can be done this way. The only difficulty to overcome is the following: DLs intermix both dimensions of knowledge. A SAT solver can not deal with modal constructors (since they are meaningless in propositional logic), whereas a purely modal solver is only concerned with the neighbors of individuals and does not perform inferences about individuals themselves. Hence, we need to coordinate the SAT solver and the purely modal solver in a well-defined way. This can be achieved in a principled way with the following simple observation: Although modal concept expressions can not be interpreted in propositional logic, they are still statements that are true or false. Hence, for a propositional reasoner, they simply constitute *atomic statements* which are not interpreted by the SAT solver in detail. The SAT solver only decides if these modal statements need to be true or false. In a second step, the modal reasoner can then use the constraints on truth value of the modal concept expressions computed by the SAT solver in order to perform the purely modal inferences when interpreting the input concept and its subexpressions. We illustrate the basic idea with an example:

Example 1. *Consider the concept expression* $C = A \sqcap (\exists S.\exists R.\neg B \sqcup \neg A \sqcup (\exists R.D \sqcap \forall R.E))$ *and the TBox* $\mathcal{T} = \{E \sqcap D \sqsubseteq \bot\}$. *To check if* C *is satisfiable wrt.* $\mathcal{T}$, *we assume that there is an individual* i *such that* $C(i)$ *holds.*

$$C = A \sqcap (\exists S.\exists R.\neg B \sqcup \neg A \sqcup (\exists R.D \sqcap \forall R.E))$$
$$\bullet$$
$$i$$

To construct a model, we consider the modal concept subexpressions as (new) atomic concepts $P_1 = \exists R.D$, $P_2 = \forall R.E$, *and* $P_3 = \exists S.\exists R.\neg B \sqcup \neg A$. *Hence, for a SAT solver* C *appears as* $C' = A \sqcap (P_3 \sqcup (P_1 \sqcap P_2))$.

$$C' = A \sqcap (P_3 \sqcup (P_1 \sqcap P_2))$$
$$\bullet$$
$$i$$

We can therefore infer from $C'(i)$ by purely propositional reasoning (i.e. a call to a SAT solver) that the i must have one of the properties $M_1^i = \{A, P_3\}$ or $M_2^i = \{A, P_1, P_2\}$. From a propositional standpoint, we found at least two models for C'. However, from a DL perspective, they do not necessarily constitute a model from a DL perspective: C' is under DL semantics not the same as the original expression C.

Let's consider the case $M_2^i = \{A, P_1, P_2\}$ first, i.e. we assume that the individual i has the elementary (logical) properties A, P_1, and P_2.

$$M_2^i = \{A, P_1, P_2\}$$
$$\bullet$$
$$i$$

If we now interpret the propositional model M_2^i of C' under DL semantics (and hence assign a detailed semantic structure to $P_1 = \exists R.D$ and $P_2 = \forall R.E$), we find that i must not only have the property A, but in order to satisfy by P_1 it must be related to at least one R-related individual j with the property D.

$$M_2^i = \{A, P_1(= \exists R.D), P_2\} \qquad D$$
$$i \bullet \xrightarrow{\qquad R \qquad} \bullet j$$

Further, since i must have the property P_2, j must at the same time have the property E. Now, if it is possible to construct a graph of individuals i, j with the required elementary logical properties, then we in fact found a DL interpretation in which the input concept is satisfiable.

$$M_2^i = \{A, P_1(= \exists R.D), P_2(= \forall R.E)\} \quad \{D, E\}$$
$$\bullet_i \xrightarrow{\qquad R \qquad} \bullet_j$$

However, it is easy to see that this is not possible in our case here, since the TBox $\mathcal{T}$ specifies that $E \sqcap D \sqsubseteq \bot$ (i.e. $E^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$), there can not be

any such object j. Hence, we can conclude that this way (i.e. by assuming i from a propositional point of view has the properties $M_2^i = \{A, P_1, P_2\}$) we cannot *construct a DL model. However, we might still be able to find a model for concept C if we consider the remaining case M_1^i as our logical perspective on the elementary properties of i. Hence, let us assume that i has the properties A and $P_3 = \exists S.P_4$ for $P_4 = \exists R.\neg B \sqcup \neg A$.*

$$M_1^i = \{A, \exists S.\exists R.\neg B\}$$
$$\bullet$$
$$i$$

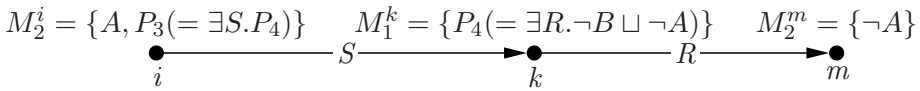
Again, we have modal requirement in M_1^i which is not yet satisfied by our construction: $P_3(i) = \exists S.P_4(i)$ requires that there exist an S-successor k for i that has the property P_4 and no such individual exist in our graph, we extend the graph with a new node k which is known to have the property P_4.

$$M_2^i = \{A, P_3(= \exists S.P_4)\} \qquad\qquad M_1^k = \{P_4\}$$
$$\bullet \xrightarrow{\quad S \quad} \bullet$$
$$i \qquad\qquad\qquad k$$

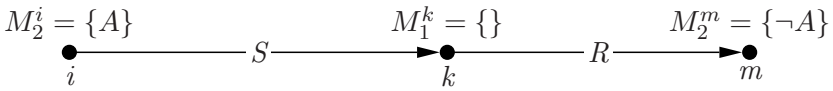
Since P_4 is a modal statement again, we are still not finished, since it is not necessarily satisfied in our current graph structure. By interpreting $P_4 = \exists R.\neg B \sqcup \neg A$ under DL semantics, we can verify easily that we indeed need to perform a last extension of our graph structure:

$$M_2^i = \{A, P_3(= \exists S.P_4)\} \qquad M_1^k = \{P_4(= \exists R.\neg B \sqcup \neg A)\} \qquad \{\neg B \sqcup \neg A\}$$
$$\bullet \xrightarrow{\quad S \quad} \bullet \xrightarrow{\quad R \quad} \bullet$$
$$i \qquad\qquad k \qquad\qquad m$$

For node m we have again two possible propositional models for the concept $P_4 = \exists R.\neg B \sqcup \neg A$, i.e. $M_1^m = \{\neg B\}$ and $M_2^m = \{\neg A\}$. Both are consistent views on the logical properties of the individual m in our domain. We can select either one without having to face any logical inconsistency, e.g. let us decide to use M_2^m. Moreover, no further extensions needed since M_2^m contains no modal statement anymore and therefore can be satisfied by deciding on the the properties of m alone:



Note, that all modal statements about individuals have been satisfied by modifications of the constructed graph. Hence, they are implicitly represented in the interconnection of the individuals and the single selected propositional models for the elements in $\Delta^{\mathcal{I}}$. For this reason, the respective auxiliary propositional symbols that we introduced in the process above are not necessary for our purposes at all and the modal parts of the selected propositional models can be skipped in the end:



Indeed, since there are no further uninterpreted modal statements in our graph, and all individuals i, k, m are assigned a consistent propositional view on their logical properties, we can construct this way immediately a DL model (or Kripke-structure) $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for the input concept C:

$\Delta^{\mathcal{I}}$	$:=$	$\{i, m, k\}$	*and*
$A^{\mathcal{I}}$	$:=$	$\{i\}$	*and*
$N^{\mathcal{I}}$	$:=$	$\emptyset$	*for any other concept name $N \in \mathbf{C}$.*
$S^{\mathcal{I}}$	$:=$	$\{\langle i, k\rangle\}$	*and*
$R^{\mathcal{I}}$	$:=$	$\{\langle k, m\rangle\}$	*and*
$T^{\mathcal{I}}$	$:=$	$\emptyset$	*for any other role name $T \in \mathbf{R}$.*

3.2 The SAT-Tableau Calculus

The SAT-Tableau procedure works on *finite representations* of a *partially constructed tableau* for an input concept and a terminology. This data structure is called a *completion graph*. Completion graphs represent the actual

data structure that represents search states in our model construction algorithm. Before introducing this data structure we need the notion of a SAT-Tableau:

Definition 7 (SAT-Tableau)**.** *Let Σ be a signature, $\mathcal{T}$ be a terminology over Σ, and $C \in \mathcal{C}(\Sigma)$. Let $\mathbf{R}_{C,\mathcal{T}}$ denote the set of role names occurring in C or $\mathcal{T}$ and 2^S denote the powerset of a set S. A* ***SAT-tableau for*** *C* ***wrt.*** *$\mathcal{T}$ is a labeled graph $T = (\mathbf{V}, \mathbf{E}, l, s)$ with a non-empty set of vertices $\mathbf{V}$, a function $\mathbf{E} : \mathbf{R}_{C,\mathcal{T}} \rightarrow 2^{\mathbf{V}\times\mathbf{V}}$ assigning role names occurring in C or $\mathcal{T}$ a set of edges between nodes in $\mathbf{V}$, a function $l : \mathbf{V} \rightarrow 2^{sublit(\{C\}\cup\mathcal{T})}$ assigning to each node a set of concept subexpressions in C or $\mathcal{T}$ or their complement, and a function $s : \mathbf{V} \rightarrow 2^{pliterals(sub(\{C\}\cup\mathcal{T}))}$ assigning to each node a propositional interpretation for $l(n)$ such that for all nodes $n \in \mathbf{V}$ it holds that:*

(P1) *$s(n)$ satisfies $l(n)$ propositionally*

(P2) *for any $\neg\exists R.D \in s(n)$ and any $\langle n, n'\rangle \in \mathbf{E}(R)$: $\neg D \in l(n')$*

(P3) *for any $\exists R.D \in s(n)$ there exists an $\langle n, n'\rangle \in \mathbf{E}(R)$ such that $D \in l(n')$*

(P4) *there exists a node $m \in \mathbf{V}$ such that $C \in l(m)$*

(P5) *for any $D_1 \sqsubseteq D_2 \in \mathcal{T}$: $\neg D_1 \sqcup D_2 \in l(n)$*

Definition 8 (SAT-Completion Graph)**.** *Let $\Sigma = (C, R)$ be a signature and $\mathcal{T}$ be a terminology over Σ. Let $\mathbf{R}_{C,\mathcal{T}}$ denote the set of role names occurring in C or $\mathcal{T}$ and 2^S denote the powerset of a set S. A* ***SAT-completion graph for*** *C* ***wrt.*** *$\mathcal{T}$ is a labeled graph $G = (V, E, p, r, \pi, state)$ with a non-empty but finite set of vertices V representing individuals, a (possibly empty) set of edges $E \subseteq V \times V$ connecting individuals, a node label function $p : V \rightarrow 2^{sublit(\{C\}\cup\mathcal{T})}$ assigning to each node a set of properties (as concept subexpressions in C or $\mathcal{T}$) that are required to hold for the individual, a edge label function $r : E \rightarrow 2^{\mathbf{R}_{C,\mathcal{T}}}$ assigning roles names to edges in G, a function $\pi : V \rightarrow 2^{pliterals(sub(\{C\}\cup\mathcal{T}))}$ assigning to each node a partial propositional truth assignment for $p(n)$, and a function $state : V \rightarrow \{$*NotAssigned,

ASSIGNED, UNSATISFIABLE, REASSIGN} *capturing the processing status of each node in* G. G *must contain a node* $n \in V$ *with* $C \in p(n)$.

For any nodes $n, n' \in V$ *with* $\langle n, n' \rangle \in E$ *and* $r(\langle n, n' \rangle) = R$, *we call* n' *an* R***-successor*** *of* n *in* G. n *is called* ***ancestor*** *of* n' *in* G *if there is a path from* n *to* n' *in* G *whereby the single edges on the path can be labeled arbitrarily.*

A completion graph G *contains a* ***clash*** *if there exists a node* $n \in V$ *such that* $state(n) =$UNSATISFIABLE. *It is called* ***clash-free*** *if it does not contain any clash.*

Essentially, a completion graph is very similar to a SAT-tableau, whereas the variety of properties constraining nodes, edges and interdependencies are not present yet. Therefore, each SAT-tableau corresponds to a completion graph, whereas completion graphs do not necessarily correspond to a SAT-tableau, but allow to represent SAT-tableau partially. Consequently, they are a rather natural data structure for any process that systematically and iteratively tries to construct a SAT-tableau.

In order to construct a tableau (or completion graph), we need to find propositional models for each individual node in the graph. We achieve this by a dedicated algorithm, a *SAT-Solver*, whose internals are not interesting for our matters here. Hence, we need an abstract model for a propositional solver. The following definition captures our understanding formally:

Definition 9 (Propositional Solver)**.** *Let* Σ *be a signature. A* ***propositional solver*** $\mathfrak{B}$ *is an algorithm which takes as input any set* $\mathcal{C} \subseteq \mathcal{C}(\Sigma)$ *of concepts and computes partial truth assignments that are sound for* $\mathcal{C}$, *i.e.* $\mathfrak{B}(\mathcal{C}) = \mathcal{M}$ *for a set* $\mathcal{M} = \{\pi_1, \ldots, \pi_k\}$ *of partial truth assignments that are sound for* $\mathcal{C}$.

A propositional solver $\mathfrak{B}$ *is called* ***complete*** *if and only if it returns for any set of concepts a complete set of partial propositional models, i.e. for any set* $\mathcal{C} \subseteq \mathcal{C}(\Sigma)$, $\mathfrak{B}(\mathcal{C})$ *is a complete set of partial truth assignments for* $\mathcal{C}$.

Let $\mathbb{BS}$ *denote the set of all propositional solvers, and* $\mathbb{CBS}$ *denote the set of*

all complete propositional solvers.

In other words, a (complete) propositional solver is an algorithm that is capable of computing and enumerating compact representations of (all) propositional models of the given set of concepts $\mathcal{C}$. An example for a complete boolean solver is the DPLL procedure [3] (*without* the pure literal rule) when exhaustive backtracking over propositional models is performed (i.e the solver is repeatedly called again in the last search state (after returning propositional model) to compute and enumerate steps-by-step all models).

Constructing a SAT-Tableau. This algorithm for checking concept satisfiability non-deterministically constructs completion graphs for the input concept C by starting with the simplest (or smallest) and in general not fully-expanded completion graph G_0 for C that can be defined as follows:

$$\begin{aligned} G_0 \quad := \quad & (V, E, p, r, \pi, \mathit{state}) \text{ with} \\ & V = \{n_0\}, E = \emptyset, p = \{n_0 \mapsto \{C\}\}, r = \emptyset, \pi = \emptyset, \\ & \mathit{state} = \{n_0 \mapsto \textsc{NotAssigned}\} \end{aligned} \tag{3.1}$$

The algorithm then proceeds by iterative (non-deterministic) application of the completion rules described in Fig. 3.1. Note, that any application of a completion rule from Fig. 3.1 to a completion graph for C wrt. $\mathcal{T}$ results again in a completion graph for C wrt. $\mathcal{T}$.

Definition 10 (Fully-expanded Completion Graph)**.** *A completion graph G is called **fully-expanded** if and only if none of the rules of the completion rules system in Fig. 3.1 is applicable anymore.*

In essence, by any application of any of the completion rules in the inference system we convert a completion graph G into another completion graph G' that either satisfies an increasing number of the semantic constraints (P1) - (P5) that identify a SAT-Tableau (and hence our goal state), or mark the completion graph as containing a clash and therefore being a dead-end

for our completion process. In a sense, we generate increasingly complete (under)approximations of a SAT-tableau (from which we can immediately read of a model for the input concept and the given terminology).

This way, the algorithm eventually (i.e. in the limit of the construction process) must create a fully-expanded completion graph for C wrt. $\mathcal{T}$. This completion graph either contains a clash (in which case we reached a dead-end and where not successful in finding a model) or it is clash-free. In the latter case, we in fact found a (finite representation) of a tableau and therefore a model.

Rule		Description
$\rightarrow_{Select}$	if 1.	n is a node in G, $state(n) \in \{$NotAssigned, ReAssign$\}$
	2.	$\mathfrak{B} \in \mathbb{S}$ is some propositional solver
	3.	π_j is some propositional model in $\mathfrak{B}(p(n) \cup \pi(n))$
	then	set $\pi(n) := \pi_j$ and $state(n) :=$ Assigned
$\rightarrow_{Clash}$	if 1.	n is a node in G, $state(n) \in \{$NotAssigned, ReAssign$\}$
	2.	$\mathfrak{B}(p(n) \cup \pi(n)) = \emptyset$
	3.	$\mathfrak{B} \in \mathbb{S}$ is some complete propositional solver
	then	set $state(n) :=$ Unsatisfiable
$\rightarrow_{\exists}$	if 1.	n is node in G, $state(n) \in \{$Assigned, ReAssign$\}$
	2.	$c_i = \exists R.C \in \pi(n)$ and there does not exist an R-successor n' of n in G such that $C \in p(n')$ and
	then	create a new R-successor n' of n and set $p(n') := \{C\}$ and set $state(n') =$ NotAssigned and set $\pi(n') := \emptyset$
$\rightarrow_{\forall}$	if 1.	n is node in G, $state(n) \in \{$Assigned, ReAssign$\}$
	2.	$c_i = \neg\exists R.C \in \pi(n)$ and n' is an R-successor of n in G such that $\neg C \notin p(n')$
	then	set $p(n') := p(n') \cup \{\neg C\}$ and set $state(n') =$ ReAssign
$\rightarrow_{\mathcal{T}}$	if 1.	n is node in G, $state(n) \neq$ Unsatisfiable
	2.	$C \sqsubseteq D \in \mathcal{T}$ and $\neg C \sqcup D \notin p(n)$
	then	set $p(n) := p(n) \cup \{\neg C \sqcup D\}$ and set $state(n) =$ ReAssign

Figure 3.1: Completion Rules for SAT-Tableau in $\mathcal{ALC}$ (wrt. a class $\mathbb{S} \subseteq \mathbb{BS}$ of propositional solvers)

3.3 Rule System

At any moment in time during the completion process, each node is in precisely one of the following processing states: (1) NotAssigned identifies nodes which have not been considered by the SAT solver yet and therefore need to have assigned a propositional model yet before any kind of modal reasoning step can be performed on that node, (2) Assigned identifies nodes for which the propositional reasoning is completed and which have been assigned a propositional model which can subsequently considered by the modal reasoner to extent and modify the completion graph in the local neighborhood of the node accordingly , (3) Reassign identifies nodes for which some new properties have been required by modal reasoning at neighboring nodes and which have not been taken into account yet (with the currently assigned model), and (4) Unsatisfiable identifies nodes which can not exists as such given their logical properties and therefore dead-ends in our search scheme.

There are essentially three categories of rules in the completion rule system that is given in Fig. 3.1:

- *Propositional Rules:* $\rightarrow_{Select}$ rule selects non-deterministically one of the propositional models for the current set of properties $p(n)$ and the model $\pi(n)$ that has been assigned to the node before. The latter is important for correctness of the procedure in cases where the property set of a node can change *after a propositional model* has already been selected for the same node (and respective modal expansion rules have been used), since this way consistency of the tableau construction with prior decisions is maintained. Typical cases would be the introduction of a TBox axiom by the $\rightarrow_{\mathcal{T}}$ rule whenever a model has already been selected or the propagation of a concept expression with the $\rightarrow_{\forall}$ rule via an inverse role (in DLs that support inverse roles, such as $\mathcal{SHIQ}$). The $\rightarrow_{Clash}$ rule identifies individuals which can not exist according to the given input data.

- *Modal Rules:* When a propositional model has been selected for a node (i.e. the algorithm selected its view on the logical properties of the individual amongst all possible views which are consistent with the input specification and all choices that have been made earlier in the completion process), the modal rules can be used to determine the effect of the selected view of the individual on its neighbors in the current completion graph. The application of modal rules can either insert a new node in the completion graph or extent the property set of a neighboring node by a new concept. No other changes are possible.

- *TBox Rules:* The $\rightarrow_{\mathcal{T}}$ rule ensures that every individual in the completion graph satisfies the selected TBox axiom $C \sqsubseteq D$. This technique is called *internalization* [12] and a standard technique to integrate TBoxes into the concept satisfiability check. Since for every single TBox axiom the inserted concept expression $\neg C \sqcup D$ is a concept disjunction, this rule can be understood as the *major source of non-determinism* that we have to face in the $\rightarrow_{Select}$ when determining concept satisfiability wrt. terminologies. Hence, optimization to this default treatment of concept subsumption statements in terminologies are crucial for the performance of the decision procedure when dealing with terminologies.

3.4 Problems When Using Local-search Procedures

The SAT-tableau procedure works fine with complete SAT solvers which are capable of enumerating all models of a given formula. However, with local-search-based solvers there occurs the following problem: A local-search-based SAT solver usually can not determine unsatisfiable cases. In this case, if one does not set a resource-bound (i.e. a timeout) the method would run forever on a propositionally unsatisfiable sub-problem (i.e. a node in the tableau) and would not be able to construct a fully-expanded and clash-free completion graph (or tableau). But if the input problem in fact is satisfiable

under DL semantics then such a clash-free fully expanded completion graph *must* exist. Hence, there is some way of constructing a completion graph for the input concept such that we never invoke a local-search-based procedure on an unsatisfiable propositional problem. Hence, in each node there is a possibility for finding a suitable model.

However, in order to find such a model, we need the following guarantee for a local-search-based SAT solver: When running the local-search-procedure solver long enough, then it will be able to find every model of the propositional input problem, eventually. This is the case for GSAT, TSAT and WalkSAT because of the restart mechanism (without they would not have the property) and for UnitWalk even without restarting [9].

The key problem now is how to distinguish both situations: on the one hand, if it takes long to find a model, we might have an unsatisfiable problem and need to stop, on the other hand, if, we do not run long enough, we might miss the propositional model which leads us to the fully expanded clash-free completion graph.

In order to solve this dilemma, we use iterative deepening over the time-bound: We first try to construct a fully expanded and clash-free completion graph with all propositional solver using a specific time bound. If a propositional SAT solver does not return a model within the time bound, we assume (incorrectly) that the propositional problem is unsatisfiable and backtrack, i.e. try another way of constructing the completion graph with the same propositional timebound. If in the end we are not able to construct a fully expanded clash-free completion graph this way (i.e. no choices are left), then we (correctly) can not assume that the input problem is unsatisfiable, but we increase the time bound (according to some schedule) and restart the process. Eventually all solvers solving a propositional problem will have enough time to find the required propositional models.

Therefore, using an iterative deepening procedure on top of the resource-bounded tableau construction, we are able to deal with any satisfiable input

problem. For unsatisfiable input problems, we can not use local-search-based SAT solver that (usually) can not detect unsatisfiable problems. The case of unsatisfiable concepts is therefore not considered in this project. Note, however, that there are local-search-based SAT solver, which can detect unsatisfiable problems too [6].

This procedure is essential for incomplete SAT solver, yet, it may have some performance issues. Assume the following scenario: The SAT solver successfully computes models for most nodes in the completion graph within the given time. The prover now recursively creates a lot of successor nodes, which are also successfully proved by the SAT solver. At some node the SAT solver now fails to compute a model within the given time. Therefore the prover tries to backtrack and returns "unsatisfiable" since no other non-deterministic choices can be taken. Now the prover dumps the current completion graph and restarts the procedure with increased time limit $t = t + \Delta t$. Obviously the time limit t and the time limit increase step Δt are important factors for the performance of this extension.

Another performance issue is the following: Since local-search-based SAT solvers do not work systematically, we can not say what models are returned. This means that we might get the same models *again and again* and therefore re-investigate cases that we already checked before and that did not lead in acceptable time to a solution. To solve this problem there are two solutions

- We can avoid the computation of already computed models outside the model search process by explicitly remembering which models have already been computed before for a node in the tableau, or
- We can "learn" a clause which prevents the construction of an already returned model, i.e. construct a clause from a returned model by disjunctively combining the complements of all literals in the model (clause learning).

The latter approach allows to consider the already computed clauses deeply inside the model search process and not only after a model has been found as in the first option. It might therefore be better than the first option. Both alternatives require potentially *exponential* space to represent all previously returned models.

In order to use space more controlled (and at the same time relax the non-redundancy of the model enumeration process) one could keep only a controlled set of known interpretations in memory (say polynomially many) and forget about known interpretations as soon as the restricted "memory" of enumerated models is exhausted. Compared to systematic methods where such a form of systematic memoizing [7] [13] is useful to increase efficiency of the method we might face the following complication: in systematic methods the model enumeration moves rather smoothly over the search space (i.e. the set of all possible interpretations) by flipping variables in a systematic order. Bounded memoizing usually removes "old" learned (partial) interpretations. In contrast with local search-based model enumerations we jump more unsystematically within the search space. The state space traversal is more uncontinuous and non-monotone. Therefore an age-based selection of interpretations to remove from the bounded memory does not promise to be similarly effective since the enumeration process is no longer as continuous and monotone as in the systematic case. Hence, it might be substantially more complicated to define effective strategies for "forgetting" known models when the space-bounds of the memory has been reached.

Chapter 4

Local Search for SAT

SAT, i.e. checking the satisfiability of a boolean formula in conjunctive normal form (CNF), is a canonical NP-complete problem [2]. Moreover, it is a fundamental problem in mathematical logic, automated reasoning, artificial intelligence and various computer science domains.

Local search is a widely used, general approach for solving hard combinatorial search problems. It can be interpreted as performing a random walk in a search space which, for SAT, is given by the set of all truth assignments.

Local-search-based procedures for propositional satisfiability problems can be used to solve hard, randomly generated problems that are an order of magnitude larger than those that can be handled by structured and more traditional approaches, such as the Davis-Putnam procedure [4] or resolution [16]. An empirical study in [11] has shown that local search procedures are superior for Random-3-SAT instances, while more structured instances are often, but not always, more efficiently solved by systematic search algorithms, such as Davis-Putnam. However, local search algorithms for SAT are typically incomplete, i.e., they can not detect the unsatisfiability of a given satisfiability problem. In other words, a complete algorithm gives the correct answer with certainty. If an incomplete algorithm finds a satisfying

assignment, it is guaranteed to be correct. However, if it fails to find a a satisfying assignment, this means that either the input formula is unsatisfiable, or it is satisfiable but appeared too hard for this algorithm. Moreover, local search procedures generally fail to give a non-redundant-enumeration of all existing models for a given formula, i.e. when used as model enumerators, these methods can return previously found models again and again. Both the property of incompleteness and the disability of enumerating all models are obstacles wrt. performance when being used within the SAT-Tableau framework.

4.1 Basic Idea

Given a set of clauses ϕ a local-search-based SAT-solver starts with an initial interpretation for ϕ and iteratively modifies this interpretation by flipping the truth assignment of a some variable. After a preset maximum number of flips (MAX-FLIPS) is reached, the algorithm restarts from a new initial interpretation. If after a preset maximum number of restarts (MAX-TRIES) no solution is found, the algorithm terminates unsuccessfully. Obviously, local search procedures may fail to find a satisfying assignment even if one exists, i.e. local search SAT-solver are incomplete.

Listing 4.1 shows a generalised form of a local search SAT-solver. Local search algorithms differ mainly in the heuristic for choosing the variable to be flipped in each search step (procedure choose-variable) which is significant for the final performance of the algorithm. In literature this procedure is often called *hill-climbing*.

```
procedure LocalSearch(φ, MAX_FLIPS, MAX_TRIES)
  input set of clauses φ, MAX_FLIPS, MAX_TRIES
  output satisfying assignment of φ or ''no solution found''

  begin
    for i := 1 to MAX_TRIES
      T := random truth assignment
```

```
    for j := 1 to MAX–FLIPS
      if T satisfies φ then return T
      else
        p := chooseVariable(T, φ)
        T := T with truth assignment of p flipped
      end if
    end for
  end for
  return ''no solution found''
end
```

Listing 4.1: Outline of a general local search procedure for SAT

4.2 Local-search-based Procedures

In this section we will give an overview of a few prominent local-search-based satisifiability solvers that were used in this thesis for the propositional part of the SAT-Tableau procedure.

4.2.1 GSAT

GSAT is a greedy local search procedure introduced by *Bart Selman et al.* in [19]. The algorithm performs a greedy local search for a satisfying assignment of a set of propositional clauses. The procedure starts with a randomly generated truth assignment. It then flips the assignment of the variable that leads to the largest increase in the total number of satisfied clauses. If multiple such candidates exist, GSAT randomly picks one of them and flips its truth assignment. Such flips are repeated until either a satisfying assignment is found or a preset maximum number of flips (MAX-FLIPS) is reached. This process is repeated as needed up to a preset maximum number of restarts (MAX-TRIES). See listing 4.2. If given an unsatisfiable formula GSAT will return "no solution found" in time directly propositional to (MAX-FLIPS

× MAX-TRIES). Obviously GSAT may fail to find a satisfying assignment even if one exists, i.e. GSAT is incomplete.

```
procedure GSAT(φ, MAX–FLIPS, MAX–TRIES)
  input set of clauses φ, MAX–FLIPS, MAX–TRIES
  output satisfying assignment of φ or ``no solution found''

  begin
    for i := 1 to MAX–TRIES
      T := random truth assignment
      for j := 1 to MAX–FLIPS
        if T satisfies φ then return T
        else
          p := a propositional variable such that a change
            in its truth assignment gives the largest
            increase in the total number of clauses of
            φ that are satisfied by T
          T := T with truth assignment of p flipped
        end if
      end for
    end for
    return ``no solution found''
  end
```

Listing 4.2: The procedure GSAT

There exists a more efficient version of GSAT, i.e. a Random Walk Strategy (in short RWS-GSAT) [18], which tries to escape local minima by performing sideway moves (also called *random walk*). This variant of GSAT selects the variable to be flipped in the following way: it either picks with probability p a variable occurring in some unsatisfied clause or follows, with probability $1-p$, the standard GSAT selection-theme, i.e. makes the best possible local move. However, in this thesis we used the original version of GSAT as listed in listing 4.2.

4.2.2 TSAT

As shown in section 4.2.1, in each step GSAT determines candidates of variables which lead to the largest increase in the total number of clauses that are satisfied by the current truth assignment. If more than one of such candidates exist GSAT randomly picks one of those candidates and flips its truth assignment. Therefore, it may happen that GSAT flips the same variable over and over again. The idea of TSAT [15] is to avoid such recurrent flips by making a systematic use of a tabu list of variables. TSAT keeps a fixed length chronologically ordered FIFO list[1] of flipped variables and prevents any of the variables in the list from being flipped again during a given amount of time (induced by the length of the tabu list).

The main parameter of TSAT which significantly influences the performance of the algorithm is the length of the tabu list. The experimentally obtained optimal length [15] of the tabu list with respect to the number of variables is $0.01875n + 2.8125$ where n is the number of variables. Moreover,

- a slight departure from the optimal length leads to a corresponding degradation of the performance of TSAT.
- these lengths remain optimal for random-generated instances outside the phase transition (the optimal size depends only on the number n of variables).

Extensive experimental comparisons between GSAT and TSAT have been conducted in [15]. TSAT proved very competitive in the resolution of many problems, in particular hard random K-SAT instances. It has been shown, that given a 3-SAT formula at phase transition, i.e. the ratio of clauses and variables is 4.3, TSAT solved more problems than GSAT and most of them have been solved faster than using GSAT, with respect to the number

[1] When an element is added to list it is appended to the end of the list, if the list did not contain the symbol yet. Otherwise it's position is updated, i.e. the element is moved to the end of the list. If the list exceeds it's fixed length, the first element in the list is removed.

of flips. Interestingly, the performance compared to GSAT increases with increasing number of variables.

4.2.3 WalkSAT

WalkSAT is a variation of RWS-GSAT (see end of section 4.2.1) which was first introduced in [18]. WalkSAT makes flips by first randomly picking a clause that is not satisfied by the current truth assignment and then picking (either at random or according to a greedy heuristic) a variable within that clause to flip. Thus, there remains substantial freedom in the choice of heuristic. In this thesis we focus on the selection strategy as given in listing 4.4 which was given the name *SKC*. Here, b_p denotes the break count, i.e. the number of clauses that become unsatisfied if the truth assignment of variable p is flipped.

```
procedure WalkSAT(φ, MAX–FLIPS, MAX–TRIES)
  input set of clauses φ, MAX–FLIPS, MAX–TRIES
  output satisfying assignment of φ or ``no solution found''

  begin
    for i := 1 to MAX–TRIES
      T := random truth assignment
      for j := 1 to MAX–FLIPS
        if T satisfies φ then return T
        else
          p := select–variable(φ, T)
          T := T with truth assignment of p flipped
        end if
      end for
    end for
    return ``no solution found''
  end
```

Listing 4.3: The procedure WalkSAT

```
procedure select–variable(φ, T)
```

```
  input set of clauses φ, truth assignment T
  output variable p

  C := a random unsatisfied clause
  u := a variable S in C with minimum b_S
  if b_S = 0 then
    p := a variable P in C with b_P = 0
  else
    with probability p
      p := a random variable in C
    with probability 1 − p
      p := a variable P in C with minimal b_P
    end
  end if
  return p
end
```

Listing 4.4: The procedure select-variable

Firstly, hill-climbing is done solely on the number of clauses that become unsatisfied if a variable is flipped, and the number of clauses that get satisfied is ignored. Secondly, a random move is never made if it is possible to do a move in which no previously satisfied clause becomes unsatisfied. Although very similar to RWS-GSAT with 100% walk ($p = 1.0$), WalkSAT can give a substantial speed up over GSAT with walk and especially over GSAT without walk. Still WalkSAT is an incomplete local search procedure.

4.2.4 UnitWalk

UnitWalk [9] is a local search procedure that uses unit clause elimination, which is a common technique in complete SAT algorithms. Unit clause elimination works as follows: If a clause is a unit clause, i.e. it contains only a single literal, this clause can only be satisfied by assigning the necessary value to make this literal true. In practice this often leads to deterministic cascades of units, thus avoiding a large part of the naive search space.

The algorithm generates an initial random truth assignment and then modifies it step by step. The main difference to other local search algorithms is that during this modification the algorithm also modifies the input formula, i.e. replaces a formula G by the formula $G[v \leftarrow t]$ for a variable v and a truth value t. A modification starts with choosing a random permutation of variables. At each step, the algorithm substitutes the value of one variable in the current formula. If there are unit clauses then the variable is taken from one of them. If the value of the variable does not satisfy the unit clause and the formula does not contain the opposite unit clause, it is flipped before the substitution. If there are no unit clauses the algorithm substitutes the value of the next variable in the chosen permutation (taking the value of from the current truth assignment). If no variable was flipped yet, the algorithm chooses a variable at random and flips it. The complete procedure is listed in listing 4.5 with corresponding procedure modify-assignment in listing 4.6.

```
procedure UnitWalk(φ, MAX–FLIPS, MAX–TRIES)
  input set of clauses φ containing n variables
    x_1,...,x_n, MAX–FLIPS, MAX–TRIES
  output satisfying assignment of φ or ''no solution found''

  begin
    for i := 1 to MAX–TRIES
      T := random truth assignment
      for j := 1 to MAX–FLIPS
        if T satisfies φ then return T
        else
          T := modify–assignment(φ, T)
        end if
      end for
    end for
    return ''no solution found''
  end
```

Listing 4.5: The procedure UnitWalk

```
procedure modify–assignment(φ, T)
```

```
  input set of clauses φ containing n variables
    x_1,...,x_n, truth assignment T
  output truth assignment T

  π := random permutation of 1 ... n
  G := φ
  f := 0
  for i := 1 to n
    while G contains a unit clause
      pick a unit clause {x_j} or {¬x_j} from G at random
      if this clause is not satisfied by T and G does not contain
        opposite unit clause then flip T[j] and set f := 1
        G := G[x_j ← T[j]]
      end while
      if variable x_π[i] still appears in G then
        G := G[x_π[i] ← T[π[i]]]
    end while
  end for
  if G contains no clauses then output T and exit
  if f = 0 choose j at random from 1...n and flip T[j]
  return T
end
```

Listing 4.6: The procedure modify-assignment

Another important difference to other local search procedures is that this algorithm is probabilistically approximately complete, i.e., if MAX-FLIPS is set to $+\infty$ and MAX-TRIES to 1, then for every satisfiable formula and every initial assignment UnitWalk finds a satisfying assignment with probability 1.

Experiments in [9] showed that UnitWalk is very competitive compared to other local search procedures. For example, UnitWalk makes substantially less flips on average than other algorithms and is capable of solving hard problems other algorithms failed to solve.

Chapter 5

Implementation

YODLR (**Y**et **O**ne-More **D**escription **L**ogics **R**easoner) is a Java framework that provides an abstract interface for formal (logical) languages and implementations of various reasoning techniques such as a standard tableau-based approaches, BDD-based approaches and a simple implementation of a SAT-Tableau-based reasoner. The standard tableau reasoner has been developed by Simon Knoll and the BDD-based reasoner and an implementation of a simple Description Logic was implemented by Markus Ruepp. Adrian Marte implemented the SAT-Tableau-based reasoner. Currently, simple implementations are provided to find models for propositional logic problems and to check concept satisfiability in $\mathcal{ALC}$ without TBoxes and ABoxes.

The goal of YODLR is "to design, implement and evaluate a novel reasoning framework for Description Logics and related Modal Logics" and to "become a competitive DL reasoner for classical reasoning tasks which eventually might outperform existing state-of-the-art inference systems for these logics."[1]

In the course of this thesis we implemented the local search procedures listed

[1] Yet One More Description Logic Reasoner, May 20th 2008, http://www.yodlr-reasoner.net/

in section 4.2, implemented the SAT-tableau algorithm and wrote various parser for various syntaxes which were used in different benchmarks we performed on the SAT-Tableau.

5.1 Architecture Idea

An important principle of *YODLR* is the exchangeability of the components implemented in the framework. Therefore, a lot of interfaces are provided which should be implemented by the specific realizations of these interfaces. To simplify instantiation of these realizations the factory pattern was applied to a number of places of the framework.

The basic architecture of *YODLR* follows an abstract Model-View-Controller pattern consisting of three different layers:

- **Model:** Consists of interfaces and an implementation for an abstract representation of formal languages and for required data structures such as completion graphs for tableau reasoner.
- **Controller:** Consists of procedures and algorithms such as propositional SAT solver or Description Logics reasoner each building upon components of the *Model* layer.
- **View:** Provides user interfaces for various components of both the *Model* and the *Controller* layer such as a user interface for the BDD-based reasoner or the standard tableau-based reasoner.

A layered architecture has also been applied to the controller component of the SAT-Tableau framework, which consists of two layers, the *modal* layer and the *propositional logic* layer. The *modal* layer consists of data structures required for the tableau procedure, e.g. an implementation of a completion graph. The *propositional* layer (see section 5.2) provides interfaces and implementations for the propositional part of the SAT-Tableau procedure such as satisfiability solver and their required data structures.

5.2 Propositional Layer

As already mentioned in section 5.1 the propositional layer is part of the controller layer of the framework. It is an auxiliary base layer within this controller component. This layer provides interfaces and implementations for the propositional part of the SAT tableau procedure. The package is physically not in the controller package, since it is meant to be reusable outside the context of *YODLR*. Here we want to list the most important components of this layer:

- **ICNF:** An interface representing a propositional formula in conjunctive normal form. It provides methods to check if a truth assignment is satisfying the formula. This component is mainly used by the propositional SAT solvers.
- **IPropositionalModelFinder:** An interface that represents propositional SAT solvers. This component computes a model for given CNF formula if the formula is satisfiable. This interface must be implemented by all propositional SAT solvers.
- **IPropositionalSolver:** This component allows to enumerate models for a propositional input problem. Internally it uses some `IPropositionalModelFinder`. Programmatically, the enumeration is represented by an `Iterator` interface.
- **LocalSearch:** An abstract class representing local search SAT solvers. Subclasses of this class such as `WalkSAT` or `UnitWalk` only have to implement the abstract method `modifyAssignment` which corresponds to hill-climbing in the local search procedure. This class implements the `IPropositionalModelFinder` interface.
- **IModelRelevanceTest:** This component is used by propositional SAT solvers in order to check if a given interpretation is relevant or not. This is of major importance for non-systematic SAT solvers since

it allows to prevent the Propositional Solver implementation to return the same models over and over again.

5.3 A Simple SAT-Tableau Implementation

A simple implementation of the SAT tableau procedure for $\mathcal{ALC}$ without ABox and TBox has been implemented. This implementations checks the satisfiability of a concept description C by creating a completion graph with root labeled with C and recursively creating and checking successor nodes by applying the rules of the SAT tableau procedure.

When checking the satisfiability state of a node a new SAT solver is created which checks the satisfiability of the propositional representation of the concept labelling the specific node. For performance reasons, the implementation of the SAT tableau procedure provides a way to dynamically choose the best propositional solver for the given node. The propositional solver tries to compute a model for the concept labeling the given node. If a model is computed successfully, the rules of the SAT tableau procedure are applied and successor nodes are created (if required), which are recursively checked for satisfiability. Otherwise, the node is marked as unsatisfiable and the procedure tries to backtrack.

As already mentioned in section 3.4 this approach only works for complete SAT solver. Therefore the extension discussed in section 3.4 has also been implemented. See listing 5.1 for an outline of this implementation.

```
procedure iterative-deepening-prover(C)
  input concept description C
  output satisfiability state of concept C

  s := ''unsatisfiable''
  round := 1
  do
    t := get-timelimit(round)
```

```
    set SAT solver time limit to t
    s := result of satisfiability check of tableau prover
    round := round + 1
  while s = ``unsatisfiable'' and t ≤ maximum time limit

  return s
end
```

Listing 5.1: Outline of the iterative deepening based tableau prover implementation

Chapter 6

Evaluation

In this section we evaluate the SAT-Tableau procedure using both a simple implementation of the DPLL algorithm and the different local-search-based SAT solver presented in chapter 4. First, we run the benchmarks from the DL'98 Systems Comparison official test suite[1] which show the strength of DPLL solver for solving structured problems. Later the performance of the various SAT solver is tested by running a few benchmarks on random modal K-CNF [10] problems. Since local search procedures are very competitive when solving such random problems, we expect the local search to produce a speed up over the systematic approaches, which is however not always the case.

All experiments were made on a 2.5GHz Intel Core 2 Duo with 4GB RAM running under Windows Vista. When using the DPLL procedure for SAT-Tableau the standard SAT-Tableau implementation has been used, for the local search SAT solvers the iterative deepening based tableau prover implementation has been used. For the local-search-based solvers GSAT, TSAT and WalkSAT the MAX-TRIES respectively MAX-FLIPS values have been fixed to 5 respectively 500, for UnitWalk the MAX-TRIES was set to 1 and

[1] 1998 International Workshop on Description Logics, June 9th 2008, http://dl.kr.org/dl98/comparison/data.html

Problems	DPLL	GSAT	TSAT	WalkSAT	UnitWalk
k_dum_n	21	1	1	3	1
k_lin_n	21	2	3	3	1
k_grz_n	21	1	2	3	3
k_poly_n	21	4	4	4	1

Table 6.1: DL'98 test suite results

MAX-FLIPS to $+\infty$. Our implementations of the local-search-based SAT solvers do not integrate the search history in the local search process in form of learned clauses, as discussed in section 3.4, but use a simple (and inefficient) mechanism which returns a computed model only if it is has not been computed yet.

The DL'98 test suite consists of 4 sets of tests, from which only the tests in the *Concept satisfiability tests* set have been executed. These measure the performance of the tableau prover when testing the satisfiability of large concept expressions without reference to a TBox. There are 9 pair of files, making 18 files in total, with each file containing 21 concept expressions of increasing size. The pairs have names such as k_branch_p.alc and k_branch_n.alc, with the expressions in the _p files being all unsatisfiable while those in the _n files are all satisfiable. For a more detailed description of the problem instances see [8]. Clearly, we only used the satisfiable ones, i.e. the _n files, since unsatisfiable problems can not be dealt with by SAT-Tableau based on incomplete local-search-based SAT solvers.

The idea of the test is to measure for each file the size of the largest expression whose satisfiability the system is able to test in less than 100 seconds. The correctness of the system is also tested by checking that the answer is as expected. For example if the satisfiability of the first expression is tested in 10s, that of the second in 50s and that of the third in 120s, then the result of the test is 2. If the system is able to test the satisfiability of the largest formula in less than 100s than the result is 21. See table 6.1 for the numerical data of the experiments.

It is obvious that DPLL substantially outperforms the local-search-based

solvers. With DPLL it is possible to solve all 21 problems of each file under 100 seconds. GSAT and TSAT have similar performance, but TSAT seems to perform slightly better than GSAT. This corresponds to the performance of TSAT in [15], where it is shown that TSAT outperforms GSAT on problems with a large number of variables. WalkSAT clearly performs best in these benchmarks compared to the other local-search-based algorithms. This also correlates with the fact, that WalkSAT is one of the fastest local-search-based propositional SAT solvers. Interestingly, UnitWalk shows a quite weak performance on these benchmarks. It seems UnitWalk shows good performance when computing a single model for a propositional formula, but seems to be weak in non-redundantly enumerating models, which significantly affects the performance of the SAT-Tableau procedure (see section 3.4). This may be the case, since our UnitWalk implementation does not restart after a preset number of MAX-FLIPS (i.e. generate a new initial truth assignment) like other local-search-based solvers, but infinitely many times modifies a truth assignment and the input formula until a satisfying assignment is found or the the maximum time limit is reached. Overall, local-search-based methods perform significantly worse on structured problems for the given benchmark problem set. They do not lead to a competitive decision procedure (solving at most 4 compared to 21 on this problem set). Moreover, within the local-search-based procedures WalkSAT gives more robust (or uniform) behaviour than any of the other methods. We suspect that the reason for the bad performance in this experiment is a potentially extreme redundancy of the local-search-based model enumeration algorithms on the benchmark examples.

Further, the SAT-Tableau procedure has been benchmarked on random modal K-CNF problems. For modal K-CNF problems there are five parameters: the number of propositional variables N, the number of modalities M, the number of modal subformulae per disjunction K, the number of modal subformulae per conjunction L, the modal degree D, and the probability P. Based on a given choice of parameters random modal K-CNF formulae are defined inductively as follows. A random (modal) atom of degree 0 is a

variable randomly chosen from the set of N propositional variables. A random modal atom of degree D, $D > 0$, is with probability P a random modal atom of degree 0 or an expression of the form $\Box_i \Phi$, otherwise, where $\Box_i$ is a modality randomly chosen form the set of M modalities (e.g. for $\mathcal{ALC}$ these modalities correspond to the existential role restriction $\exists R_i \Phi$ and to the universal role restriction $\forall R_i \Phi$) and Φ is a random modal K-CNF clause of modal degree $D-1$ (defined below). A random modal literal (of degree D) is with probability 0.5 a random modal atom (of degree D) or its negation, otherwise. A random modal K-CNF clause (of degree D) is a disjunction of K random modal literals (of degree D). Now, a random modal K-CNF formula (of degree D) is a conjunction of L random modal K-CNF clauses (of degree D).

First, we benchmark 3-CNF instances of modal degree $D = 0$, i.e. 3-CNF problems where each literals is a propositional literal. Later, we also show the performance of the procedure on 3-CNF instances with modal degree $D = 1$, i.e. 3-CNF problems where each literal is with probability p a propositional literal or with probability $1 - p$ a modal subformula of the form $\exists R.C_l$ where C_l is a clause with 3 propositional literals. For each problem 10 instances were randomly generated and each instance was run 10 times. For each run the time (in milliseconds) was measured and the median was computed. The tables show for each problem the arithmetical mean of these 10 median values. If there is no value listed in the table, then the problem could not be successfully computed by the specific prover. Again, only satisfiable problems have been used.

For the 3-CNF problems with modal degree 0, 10 instances are randomly generated for the variable numbers 10, 50, 100, 150 and corresponding clause numbers 43, 215, 430, 645. It is important to point out, that satisfiability checking of 3-CNF problems with modal degree 0 directly corresponds to propositional satisfiability testing since only propositional symbols and constructors occur in such a formula.

The SAT-Tableau procedure only creates a single-node completion graph

$D = 0$						
N	L	DPLL	GSAT	TSAT	WalkSAT	UnitWalk
10	43	12.1ms	16.7ms	35.3ms	11.2ms	25.6ms
50	215	61.25ms	3514.8ms	3385.25ms	210.95ms	1321.1ms
100	430	11231.45ms	75167.95ms	46684.95ms	2272.55ms	7903.0ms
150	645	-	-	-	3599.95ms	11020.6ms

Table 6.2: Results for 3-CNF problems with degree 0 (in milliseconds)

and tries to find a satisfying truth assignment for the property set labelling this single node using the propositional SAT solver. See table 6.2 for the results of the benchmark. Interestingly, DPLL shows reasonable performance on the smaller problems, but has worse performance on problems with a larger number of variables. As expected, WalkSAT and UnitWalk show best performance on such random problems. They even succeed to compute the largest problem with 150 variables where the other 3 algorithms failed. The relatively bad performance of GSAT and TSAT can be explained by the fact, that - in contrast to WalkSAT and UnitWalk - they do not perform random walks in order to escape local minima. In order to increase performance of GSAT and TSAT we have to increase the value of MAX-TRIES and more importantly the value of MAX-FLIPS so that the algorithms more often modify the initial truth assignment in order to find a satisfying assignment.

Additionally, we generate 3-CNF problems with modal degree 1. We fix all parameters except L, the number of clauses. The number of variables is fixed to $N = 5$, the number of modalities to $M = 1$ and the probability to $P = 0.7$. 10 satisfiable instances are randomly generated for the clause numbers 5, 10, 15, 20, and 25. There are two reason why we only use quite small problems for this benchmark:

1. The implementation of the SAT-Tableau algorithm works in a recursive manner and is not yet optimized, therefore, the procedure runs out of memory pretty fast for larger problems.

2. For small L the generated formulae are most likely to be satisfiable and

$D = 1$						
N	L	DPLL	GSAT	TSAT	WalkSAT	UnitWalk
5	5	9.80ms	25.45ms	25.90ms	21.95ms	3127.30ms
5	10	17.35ms	334.25ms	277.15ms	93.55ms	5851.23ms
5	15	15.16ms	3430.10ms	3014.77ms	2209.60ms	40565.10ms
5	20	13.20ms	-	-	-	-
5	25	22.12ms	-	-	-	-

Table 6.3: Results for 3-CNF problems with degree 1

for larger L the generated formulae are most likely to be unsatisfiable [1]. Since with local-search-based satisfiability solvers we can only prove satisfiable formulae, we have to use a small number of clauses to generate satisfiable K-CNF problems.

See table 6.3 for the results of the benchmark. Surprisingly, DPLL performs far better than the local-search-procedures. Even small problems take the local-search procedures relatively long to compute a result. For the larger problems the local-search-based solvers fail to compute a result since the SAT-Tableau implementation runs out of memory. Again, WalkSAt performs best within the local-search-based procedures, TSAT performs slightly better than GSAT and UnitWalks shows weakest performance on these benchmarks, because of its bad capabilities of enumerating models.

Although the local-search-based solvers relatively fast succeed in finding a model for the input problem, i.e. the property set of the root node, they had problems in proving the property sets of the successor nodes, which we will discuss in the following:

In the first step the SAT-Tableau procedure tries to compute a propositional model for the property set of the root node, which corresponds to the input problem which in this case is a K-CNF formula. The procedure then applies the expansion rules extending or creating new nodes in the completion graph according to the propositional model previously computed by the SAT solver. Since the number of modalities M was fixed to 1 only the existential quantifier was used in the K-CNF formula, therefore, a modal subformula

is either of the form $\exists r.\Phi$ or $\neg\exists r.\Phi \equiv \forall R.\neg\Phi$ where Φ is a modal K-CNF clause. In the first case the SAT-Tableau procedure creates a new successor node n of the root node with property set $p(n) = \{\Phi\}$, in the latter case the property set of all successor nodes n_i of the root node are updated to $p(n_i) = p(n_i) \cup \{\neg\Phi\}$. The procedure tries to construct a clash-free completion graph, backtracking if necessary. Each time the procedure backtracks, a new propositional model has to be computed for the property set of the root node, which has shown to be a major reason of the bad performance of the local-search-procedures.

However, an important main problem here is, that local-search-based solvers only compute complete truth assignments, which has the following two effects:

- Complete truth assignments potentially lead to an increased number of successor nodes, since to all properties a propositional truth value is assigned. This leads to an increased computational effort, since more nodes have to be (recursively) proven and it is more likely that the procedure has to backtrack
- More importantly, complete truth assignments potentially lead to a larger property set of the successor nodes. Because of this, it is more likely that a successor node becomes an unsatisfiable node, since more concept descriptions are added to the property set of the node. This potentially leads to decreased performance of the iterative deepening based SAT-Tableau procedure, since with local-search-based solvers the procedure has to wait the current time limit to identify unsatisfiable problems.

DPLL returns partial truth assignments, therefore, the number of successor nodes and the property set of each successor node have shown to be relatively small. Thus, the previously mentioned problems occur less likely. These problems clearly also apply to the DL'98 benchmarks. We will illustrate this by an example:

Example 2. *Assume we want to prove the following (satisfiable) concept description using the SAT-Tableau procedure with both a local-search-based prover and the DPLL procedure:* $C = A \sqcup \exists R.(B \sqcap C) \sqcup \exists R.(B \sqcup D)$. *The SAT-Tableau calculus creates a root node* x *with property set* $p(x) = \{A \sqcup \exists R.(B \sqcap C) \sqcup \exists R.(B \sqcup D)\}$.

$$\underset{x}{\bullet}\ p(x) = \{A \sqcup \exists R.(B \sqcap C) \sqcup \exists R.(B \sqcup D)\}$$

First we use a local-search-based SAT solver to compute a complete model $\pi_1 = (A, P1, \neg P2)$, *where* $P1 = \exists R.(B \sqcap C)$ *and* $P2 = \exists R.(B \sqcup D)$ *are the propositional correspondences of the modal subformulae. By applying the expansion rules we get the following completion graph:*

$$\underset{y}{\bullet}\ p(x) = \{A \sqcup \exists R.(B \sqcap C) \sqcup \exists R.(B \sqcup D)\} \xrightarrow{R} \underset{y}{\bullet}\ p(y) = \{B \sqcap C, \neg B \sqcap \neg D\}$$

Clearly, there is a clash in the property set of node y. *The iterative deepening based SAT-Tableau procedure nows spends the time defined by the time limit scheduling mechanism trying to compute a model for the property set of* y *and backtracks since no model can be computed.*

When using DPLL this problem potentially not occurs in the first place. Assume, DPLL computes a partial model $\pi_2 = \{A\}$ *for node* x. *Using* π_2 *no other nodes have to be created and the procedure immediately terminates with the result that the given concept is satisfiable.*

Chapter 7

Conclusion

In this thesis we gave an introduction to a novel tableau-based Description Logic reasoning procedure called SAT-Tableau. The approach combines the power of modern propositional SAT solving techniques with the most flexible and successful inference technique for Description Logic reasoning, i.e. tableau-based procedures. The SAT-Tableau calculus allows to use a range of propositional SAT solving algorithms within tableau-based Description Logic reasoning. In this thesis we considered the specific case of local search procedures used as the SAT solver component in the SAT-Tableau calculus. To overcome the problems which occur when using such local-search-based SAT solvers, we had to integrate an iterative deepening strategy over the time-bound of the SAT solver into the SAT-Tableau calculus.

We then investigated the performance of a variety of local-search-based procedures in context of the SAT-Tableau procedure. The benchmarks clearly pointed out the main advantage of structured procedures such as DPLL over the local-search procedures, which is it's strength of solving structured problems, the good performance of enumerating models for the given problems and the ability of computing partial truth assignments. In summary, the bad performance of the local-search-based solvers compared to DPLL may be explained by the following properties:

- Local-search-based SAT solvers are incomplete satisfiability solvers. Therefore they are not able to identify unsatisfiable formulae and can not (implicitly) enumerate all possible models for a formula. A local-search-based solver potentially computes already investigated models which are either returned to the SAT-Tableau procedure or filtered out by a model-relevance test. In the latter case the SAT solver may fail to compute a *new* model within the given time limit and therefore the iterative deepening based SAT-Tableau algorithm assumes that the given node is unsatisfiable and potentially restarts the procedure with increased time limit. This, however, leads to increased time the SAT solver tries to find a model for unsatisfiable formulae.

- The modal rules of the SAT-Tableau calculus either insert a new node in the completion graph or extend the property set of a neighbouring node by a new concept depending on the propositional models the SAT solver computes. Since local-search-based SAT solvers are only capable of computing complete truth assignments this potentially leads to a larger completion graph since more nodes are created or extended, which would maybe not be the case if partial truth assignments would have been computed. This again leads to an increased computational effort since these newly created or extended nodes also have to be proven by the SAT-Tableau procedure. Moreover, the property set of the (successor) nodes potentially get larger, and therefore the nodes are more likely unsatisfiable.

- The performance of the iterative deepening based SAT-Tableau depends heavily on the initial time limit t_0 and the time limit step size Δt. On the one hand, if t_0 or Δt are too low the procedure needs a lot of restarts for more complex concepts, on the other hand, if set too high the algorithm takes pretty long for not so complex problems, since the SAT solver "wastes" a lot of time computing models for unsatisfiable formulae.

In summary, WalkSAT performs best within the local-search-based proce-

dures, TSAT performs slightly better than GSAT and our implementation of UnitWalks shows weakest performance on the benchmarks because of its bad capabilities of enumerating models.

List of Figures

Listings

List of Tables

Bibliography

[1] Building decision procedures for modal logics from propositional decision procedures: Case study of modal k. *Inf. Comput.*, 162(1/2):158–178, 2000.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[3] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[5] Francesco M. Donini. *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter Complexity of Reasoning, pages 96 – 136. Cambridge University Press, 2003.

[6] H. Fang and W. Ruml. Complete local search for propositional satisfiability, 2004.

[7] Georg Gottlob and Toby Walsh, editors. *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Morgan Kaufmann, 2003.

[8] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics k, kt, and s4. Technical Report IAM-96-015, University of Bern, Switzerland, 1996.

[9] E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, 2001. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.

[10] Jan Hladik. A generator for description logic formulas. In Ian Horrocks, Ulrike Sattler, and Frank Wolter, editors, *Description Logics*, volume 147 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[11] Holger H. Hoos and Thomas Stutzle. Systematic vs. local search for SAT. In *KI - Kunstliche Intelligenz*, pages 289–293, 1999.

[12] Ian Horrocks. *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter Implementation and Optimization Techniques, pages 306 – 346. Cambridge University Press, 2003.

[13] ao P. Marques Silva Jo and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Uwe Keller and Stijn Heymans. The sat-tableau calculus: Integrating sat solvers into description logic reasoning. Technical Report 2008-01-08, Semantic Technology Institute (STI), University of Innsbruck, Technikerstrasse 21a, A-6020 Innsbruck, Austria., April 2008. Version 0.2.

[15] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Tabu search for sat. In *AAAI/IAAI*, pages 281–285, 1997.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

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

[18] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson,

editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.

[19] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.