Tableau-based Reasoning

Ralf Möller[1] and Volker Haarslev[2]

[1] Hamburg University of Technology, Germany
r.f.moeller@tu-harburg.de
[2] Concordia University, Montreal, Canada
haarslev@cse.concordia.ca

1 Introduction

As part of the infrastructure for working with ontologies, reasoning systems are required. Reasoning is used at ontology development or maintenance time as well as at the time ontologies are used for solving application problems. In this section we will review so-called tableau-based decision procedures for inference problems arising in both contexts. We start with the satisfiability problem for a set of logical formulae. Speaking about ontologies, we focus on description logics , which provide the basis for standardized practical ontology languages. In this context, the set of formulae mentioned above is usually divided into a Tbox and an Abox for the intensional and extensional part of the ontology, respectively (see below for details). We are aware of the fact that ontology processing systems based on description logics also support some form of rules as well as means for specifying constraints among attributes of different individuals [5]. For introductory purposes, here we focus on satisfiability checking in basic description logics, however.

The main idea of tableau-based methods for satisfiability checking is to systematically construct a representation for a model of the input formulae. If all representations that are considered by the procedure turn out to contain an obvious contradiction (clash), it is concluded that the set of formulae is unsatisfiable. In early publications on tableau-based proof procedures, in particular for first-order logics, the notation for the model representations was done using tables (tableaux in French). In recent approaches these tables are better described as graph structures. The name tableau is retained for historical reasons, however.

Initially, tableau-based methods for description logics have been developed for decidability proofs, and due to this fact, they are highly nondeterministic for expressive description logics. It turned out, however, that they can indeed be efficiently implemented using appropriate search strategies and index structures such that for typical-case inputs, acceptable runtimes can be expected even though the worst-case complexity is high. In practical systems,

tableau structures are efficiently maintained during branch and bound (or backtracking) with the result that tableau-based methods have been successfully employed in ontology reasoning systems such as FaCT++, Pellet, or RacerPro (cf. [26] for an overview about description logic systems).

Although, in practical contexts, tableau-based methods are often applied in a refutation-based way (i.e., they are used to show unsatisfiability of a set of formulae), the graph structures computed for solving the ontology satisfiability problem can be reused for efficiently implementing higher-level reasoning services such as instance retrieval requests. In other words, in practical systems, tableau-based methods are not just used for satisfiability checking but are also used to compute index structures for subsequent calls to other reasoning services.

In this chapter, tableau-based reasoning methods are formally introduced. We start with a nondeterministic basic version which subsequently will be extended with optimization techniques in order to demonstrate how practical systems can be built. We also demonstrate how computed tableau structures can be exploited in an ontology reasoning system. In order to make this chapter self-contained, we shortly introduce the syntax and semantics of the description logic $\mathcal{ALC}$ and introduce Tboxes and Aboxes.

An overview on tableau algorithms for description logics can also be found in [7] as well as in [6]. In this chapter, we also consider optimization issues, and the presentation is oriented towards implementation aspects in order to complement the presentations in [7, 6].

1.1 Syntax and Semantics of $\mathcal{ALC}$

For a given application problem one chooses a set of elementary descriptions (or *atomic descriptions*) for *concepts* and *roles* representing unary and binary predicates, respectively. A set of *individuals* is fixed to denote specific objects of a certain domain. We use letters A and R for atomic concepts and roles, respectively. In addition, let $\{i, j, \ldots\}$ be the set of individuals. In $\mathcal{ALC}$ (Attributive Language with full Complement), descriptions for *complex concepts* C or D can be inductively built using the following grammar:

$$
\begin{array}{rll}
C, D \longrightarrow & A & \mid \text{atomic concept} \\
& C \sqcap D & \mid \text{conjunction} \\
& C \sqcup D & \mid \text{disjunction} \\
& \neg C & \mid \text{negated concept} \\
& \exists R.C & \mid \text{existential quantification} \\
& \forall R.C & \mid \text{value restriction}
\end{array}
$$

We introduce the concept descriptions $\top$ and $\bot$ as abbreviations for $A \sqcup \neg A$ and $A \sqcap \neg A$, respectively. Concept descriptions may be written in parentheses in order to avoid scoping ambiguities.

For defining the semantics of concept and role descriptions we consider *interpretations* $\mathcal{I}$ that consist of a non-empty set $\Delta^{\mathcal{I}}$, the domain, and an interpretation function $\cdot^{\mathcal{I}}$, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a set $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. For complex concept descriptions the interpretation function is extended as follows:

$$
\begin{aligned}
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} &= \{x \mid \exists y.(x,y) \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \\
(\forall R.C)^{\mathcal{I}} &= \{x \mid \forall y. \text{ if } (x,y) \in R^{\mathcal{I}} \text{ then } y \in C^{\mathcal{I}}\}
\end{aligned}
$$

The semantics of description logics is based on the notion of satisfiability. An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *satisfies* a concept description C if $C^{\mathcal{I}} \neq \emptyset$. In this case, $\mathcal{I}$ is called a *model* for C.

A *Tbox* is a set of so-called *generalized concept inclusions* $C \sqsubseteq D$. For brevity the elements of a Tbox are called *GCIs*. An interpretation $\mathcal{I}$ satisfies a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. An interpretation is a *model* of a Tbox if it satisfies all GCIs in the TBox. A concept description C *is subsumed by* a concept description D w.r.t. a Tbox if the GCI $C \sqsubseteq D$ is satisfied in all models of the Tbox. In this case, we also say that D *subsumes* C.

An *Abox* is a set of *assertions* of the form $C(i)$ or $R(i,j)$ where C is a concept description, R is a role description, and i, j are individuals. A concept assertion $C(i)$ is satisfied w.r.t. a Tbox $\mathcal{T}$ if for all models $\mathcal{I}$ of $\mathcal{T}$ it holds that $i^{\mathcal{I}} \in C^{\mathcal{I}}$. A role assertion $R(i,j)$ is satisfied w.r.t. a Tbox $\mathcal{T}$ if $(i^{\mathcal{I}}, j^{\mathcal{I}}) \in R^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$. An interpretation satisfying all assertions in an Abox $\mathcal{A}$ is called a model for $\mathcal{A}$. An Abox $\mathcal{A}$ is called *consistent* if such a model exists, it is called *inconsistent* otherwise.

1.2 Decision Problems and their Reductions

The definitions given in the previous section can be paraphrased as decision problems.

The *concept satisfiability* problem is to check whether a model for a concept exists. The *Tbox satisfiability problem* is to check whether a model for the Tbox exists. The *concept subsumption* problem (w.r.t. a Tbox) is to check whether $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds (in all models of the Tbox).

The *Abox consistency problem* for an Abox $\mathcal{A}$ (w.r.t. a Tbox) is the problem to determine whether there exists a model of $\mathcal{A}$ (that is also a model of the Tbox). Another problem is to test whether an individual i is an instance of a concept description C w.r.t. a Tbox and an Abox (*instance test* or *instance problem*). The *instance retrieval* problem w.r.t. a query concept description C is to find all individuals i mentioned in the assertions of an Abox such that i is an instance of C.

The latter problem is a retrieval problem but, in theory, it can be reduced to several instance problems. Furthermore, the satisfiability problem for a concept C can be reduced to the consistency problem of the Abox $\{C(i)\}$. In order to solve the instance problem for an individual i and a concept description C one can check if the Abox $\{\neg C(i)\}$ is inconsistent. The concept subsumption problem can be reduced to an Abox consistency problem as well. If the Abox $\{C(i), \neg D(i)\}$ is not consistent, C is subsumed by D [6].

Thus, in theory, all problems introduced above can be reduced to the Abox consistency problem. Note that in practical systems, specific algorithms might be used to decide a certain problem.

2 Deciding the Consistency Problem for $\mathcal{ALC}$ Aboxes

A decision procedure for the $\mathcal{ALC}$ Abox consistency problem is described in this section using a so-called tableau-based algorithm. In order to simplify the presentation, in this section, we do not consider Abox consistency with respect to Tboxes. For Tboxes, among other extensions, additional machinery is required to ensure termination (see Section 3 for details).

As indicated in the introduction, the main idea of the Abox consistency algorithm is to systematically generate a representation for a model. In this process which searches for a model, some representations are generated which contain an obvious contradiction (*clash*), i.e., for an individual i we have $C(i)$ and $\neg C(i)$ in an Abox. The assertions $C(i)$ and $\neg C(i)$ are called the *culprits* for the clash. In case of a clash, the generated representations turn out to not to describe a model.

From a theoretical point of view, the algorithm described below is sound and complete but nondeterministic. In a practical implementation, indeterminism must be handled with systematic search techniques, and various heuristics have been described in the literature to guide the search process. If we see a tableau-based algorithm not as a theoretical vehicle for proving decidability of a logic, but as a practical way to solve the Abox consistency problem, then it becomes clear that it is important to be able to detect clashes as early as possible while the model representations are built. A representation with a clash no longer needs to be considered. Thus, we should be able to identify clashes not only for assertions with atomic concepts but also for assertions with complex concepts. Therefore, in contrast to other presentations of tableau algorithms we will not transform a concept into a form that makes the presentation (and analysis) of the tableau algorithm easier (negation normal form), but directly use a form that is efficient for detecting clashes in typical-case inputs (encoded normal form). The $\mathcal{ALC}$ Abox consistency algorithm described below checks whether there exists a model for the input Abox.

The algorithm operates on a set of Aboxes $\mathfrak{A}$. Each Abox represents an alternative to be investigated in the exhaustive model generation process. We

also call such an internal Abox, i.e., an element of $\mathfrak{A}$, a *tableau* (but also use the term Abox in the following). Initially, the algorithm starts with a set $\mathfrak{A} = \{\mathcal{A}\}$ containing the input Abox $\mathcal{A}$. A set of rules is applied to an Abox from this set until no more rule applications are possible or no more rule applications are needed to determine the result. The rules are introduced below. If a rule is applied to an Abox, the Abox is replaced by one or more Aboxes. If it is replaced by more than one Abox (the so-called *successor Aboxes*), we say that the rule introduces a so-called *choice point*. A rule introducing a choice point is called a *nondeterministic* rule. All other rules are called *deterministic*. In any case, the Abox to which a rule is applied is replaced with new Aboxes that are "copies" of the original one plus some additional assertions.

If a tableau (Abox) is found to contain a clash, the tableau is called *closed*, otherwise it is called *open*. A tableau to which no rule can be applied is called *complete*. For a complete tableau the synonym *completion* is also used. If there exists a completion, i.e., an open tableau to which no more rules can be applied, the algorithm returns "yes" (indicating consistency). If all tableaux that could be generated by applying rules are closed, the algorithm returns "no" (inconsistency).

2.1 Concept Normalization and Encoding

In order to speed up the clash test, concepts are normalized using several transformation steps. First, double negations are eliminated, i.e., $\neg\neg C$ is replaced with C. Then, maximal sequences of conjunctions (possibly with nested parentheses) are flattened and represented with an n-ary conjunction term $\bigwedge\{C_1, C_2, \ldots, C_n\}$ (written as a prefix operator to the set of arguments). Corresponding representations $\bigvee\{C1, C2, \ldots, C_n\}$ are built for disjunctions. The interpretation function is extended in the obvious way

$$(\bigwedge\{C_1, C_2, \ldots, C_n\})^{\mathcal{I}} = (C_1)^{\mathcal{I}} \cap (C_1)^{\mathcal{I}} \cap \ldots \cap (C_n)^{\mathcal{I}}$$

$$(\bigvee\{C_1, C_2, \ldots, C_n\})^{\mathcal{I}} = (C_1)^{\mathcal{I}} \cup (C_1)^{\mathcal{I}} \cup \ldots \cup (C_n)^{\mathcal{I}}$$

If there are two concepts C and $\neg C$ mentioned in a conjunction (disjunction) or $\bot$ ($\top$), the whole term $\bigwedge\{C_1, C_2, \ldots, C_n\}$ ($\bigvee\{C_1, C_2, \ldots, C_n\}$) is replaced with $\bot$ ($\top$).

Afterwards, in an encoding process every concept description C and its negation $\neg C$ is inductively associated with a unique identifier. For instance, one could use numbers as unique identifiers and store concepts as records in an array, or it is possible use pointers to records (or objects) as unique identifiers. The fact that conjunctions (or disjunctions) are represented as sets enables the assignment of the same unique identifier to syntactically different but semantically equivalent conjunctive and disjunctive concept descriptions. The assignment of unique identifier to a concept is known as *encoding* a concept. If we use a concept description in the following text, we assume that we refer to its unique identifier.

The function $neg(C)$ is used to find the negation of a concept. The implementation of this function should require constant time (i.e., be as efficient as possible).[3]

Next, an internal representation of the input Abox with encoded concepts is built in a preprocessing step. We assume normalized concepts are used from now on. For readability issues, however, in the presentation below, we still use concept descriptions as introduced above.

2.2 Tableau rules

The tableau rules are applied to an Abox $\mathcal{A}$ as part of the set of Aboxes $\mathfrak{A}$ on which the algorithm operates. A rule can be *applied* whenever the precondition is satisfied and $\mathcal{A}$ is not in the set of closed Aboxes (this is an implicit condition). Initially, the set of closed Aboxes is empty. Saying that $\mathcal{A}$ is *replaced* by an Abox or a sequence of Aboxes we mean that $\mathcal{A}$ is removed from $\mathfrak{A}$ and the Aboxes generated by the rule are added to $\mathfrak{A}$. If a rule is applied to an assertion, we say the assertion is *expanded*.

- **Conjunction rule**: If $(\bigwedge\{C_1, \ldots, C_n\})(x) \in \mathcal{A}$ and $\{C_1(x), \ldots, C_n(x)\} \not\subseteq \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C_1(x), \ldots, C_n(x)\}$.
- **Disjunction rule**: If $(\bigvee\{C_1, \ldots, C_n\})(x) \in \mathcal{A}$ and for all $i \in \{1 \ldots n\}$ it holds that $C_i(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with a sequence of Aboxes $A_1, \ldots A_n$ where $\mathcal{A}_1 = \mathcal{A} \cup \{C_1(x)\}, \ldots, \mathcal{A}_n = \mathcal{A} \cup \{C_n(x)\}$.
- **Existential quantification rule**: If $(\exists R.C)(x) \in \mathcal{A}$ but there is no individual name y such that $\{C(y), R(x, y)\} \subseteq \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(z), R(x, z)\}$ such that z is a fresh individual (i.e., an individual not occurring in $\mathcal{A}$).
- **Value restriction rule**: If $\{(\forall R.C)(x), R(x, y)\} \subseteq \mathcal{A}$ but $C(y) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(y)\}$.
- **Negation rule**: If $\neg C(x) \in \mathcal{A}$ but $neg(C)(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $(\mathcal{A} \cup \{ neg(C)(x)\}$.
- **Clash rule**: If $\{C(x), neg(C)(x)\} \subseteq \mathcal{A}$, then add $\mathcal{A}$ to the set of closed Aboxes.

The algorithm runs in a loop and applies a rule if its precondition is satisfied. A precondition of a rule is satisfied if there exists a substitution for the variables x or y with individuals such that the condition is satisfied. As indicated before, if the precondition is satisfied, a rule is applied to an Abox in

[3] For instance, if numbers are chosen for the unique identifier, the unique identifier of the negation of a (non-negated) concept with number n could be $n + 1$. The encoding process must assign numbers accordingly. If (pointers to) objects are used for representing concepts, a field with a pointer to the negated concept provides for a fast implementation of *neg* at the cost of memory requirements probably being a little bit higher.

$\mathfrak{A}$. Applying a rule means to execute the then-part applying the variable substitution computed from the if-part of the rule. The loop ends if a completion is found or if no rule is applicable.

The algorithm returns "yes" if there exists a completion and "no" otherwise. In principle, the rules defined above can be applied in any order. Later we will see that a rule application strategy might impose restrictions on the order of rule applications. Restrictions are introduced to find completions "early", i.e., the strategy is used for optimization purposes. If the expressivity of the language is extended, a particular rule application strategy might also be necessary to ensure termination or soundness and completeness.

A few additional definitions are appropriate for the analysis of the algorithm in the next subsection. If an Abox is replaced with one new Abox, the new Abox contains strictly more assertions. We call this process *and-branching*. Applying a nondeterministic rule (for the time being, the disjunction rule) might introduce several new Aboxes. We call this process *or-branching*.

The individuals mentioned in the original Abox are called *old* individuals, all other individuals are called *fresh*. A sequence of role assertions $R_1(x_1, x_2), R_2(x_2, x_3), \ldots R_{n-1}(x_{n-1}, x_n), R_n(x_n, x_{n+1})$ is called a *path* (of length n) from x_1 to x_{n+1}. In a path of length 1, x_2 is called the (direct) *successor* and x_1 is called *predecessor* of x_2 (for a role R). The individuals x_i with $i \in \{2, n+1\}$ are called indirect successors of x_1.

Formal Properties

The formal properties of the algorithm are analyzed in three steps. We first show termination, and afterwards we prove soundness and completeness.

The procedure *terminates*: First, no rule can be applied twice to the same Abox with the same bindings for the variables x, y due to the preconditions (no infinite and-branching). Second, although new individuals are introduced by applying the existential quantification rule, the quantification concept is of a smaller size than the original concept. Hence, there can be no infinite applications of the existential quantification rule. The length of the longest Abox is bounded by the size of the input Abox. Third, no rule deletes an assertion, and therefore, Aboxes can only grow (i.e., no so-called yo-yo effects can occur [24, p. 547]).

The algorithm is sound: If the algorithm returns "yes" there exists a model satisfying all assertions of the input Abox. This is shown as follows. If the algorithm returns "yes" there exists a completion. From the completion $\mathcal{A}$ a so-called *canonical model* $\mathcal{I}_{\mathcal{A}} = (\Delta^{\mathcal{I}}_{\mathcal{A}}, \cdot^{\mathcal{I}}_{\mathcal{A}})$ can be constructed (cf. [6]).

1. Let $\Delta^{\mathcal{I}}_{\mathcal{A}}$ be the set of all individuals mentioned in $\mathcal{A}$.
2. For all atomic concept descriptions A let $A^{\mathcal{I}}_{\mathcal{A}} = \{x \mid A(x) \in \mathcal{A}\}$.
3. For all role descriptions R let $R^{\mathcal{I}}_{\mathcal{A}} = \{(x, y) \mid R(x, y) \in \mathcal{A}\}$.

By definition all role assertions are satisfied by $\mathcal{I}_\mathcal{A}$. Now, using induction on the structure of concepts, it is easy to show that $\mathcal{I}_\mathcal{A}$ also satisfies all concept assertions in $\mathcal{A}$ (see [6] for details).

The algorithm is complete: If there exists a model for the input Abox, then the algorithms returns "yes". Or, by contraposition, it holds that if the algorithm returns "no", then there does not exist a model. If the algorithm returns "no" all tableaux are closed, i.e., there is a clash in each tableau. Under the assumption that there exists a model for an Abox to which a rule is applied, it is shown that a model for at least one of the generated Aboxes can be constructed by examining every rule (and hence, no alternative to be investigated is forgotten, for details see [6] again). Now since there is a clash in every tableau in case the algorithm returns "no", there cannot exist a model for the input Abox.

The $\mathcal{ALC}$ Abox consistency problem is PSPACE-complete, cf. [29]. The algorithm needs an exponential number of steps in the worst case. As we will see in the next subsection, there exists a rule application strategy such that intermediate tableaux can be discarded such that the algorithm runs in polynomial space in order to be worst-case-optimal.

2.3 Towards an Optimized Implementation

The tableau rules refer to assertions for specific individuals or check for a clash w.r.t. a specific individual. Thus, rather than using an arbitrary set data structure for representing a tableau, in a concrete implementation of the tableau algorithm, the set of assertions in an Abox is partitioned w.r.t. the individuals the assertions refer to (for $C(i)$ and $R(i, j)$ the assertion refers to i). We call such a partition an *individual partition* P_i. The access to the partition of an individual i should require almost constant time. If there is an assertion $R(i, j) \in P_i$, then there will also be a partition P_j for j (possibly empty). We say P_j *depends on* P_i.

Furthermore, looking at the preconditions of the rules, it is revealed that for each individual, the preconditions refer to specific concept constructors (conjunctions, disjunctions, existential quantifications, or value restrictions). Thus, for each individual partition, the set of conjunctions, disjunctions, existential quantifications, and value restrictions must be efficiently identifiable. For the latter two subsets, a further index over different roles might be considered.

The selection of a particular rule to apply is nondeterministic in the algorithm above. Various kinds of heuristics have been investigated to reduce the number of rule applications for typical-case inputs. First, in a practical implementation, best results have been achieved if the clash rule is applied with highest priority. Since no rules are applied to closed tableaux by definition, the number of applicability tests for rules is reduced if clashes are found early. The overhead for the clash rule must be kept at a minimum, however. Usually, in concrete implementations, the clash rule is (implicitly) applied

whenever a concept assertion $C(x)$ is to be added to an Abox. Note that for $\mathcal{ALC}$, role assertions do not directly involved in a clash test – a condition that is no longer true for more expressive logics. Checking whether the assertion $neg(C)(x)$ is already an element of the tableau to which $C(x)$ is to be added is a frequently executed operation in a practical implementation, and has to be implemented very efficiently. As part of this so-called *clash test*, in a practical implementation it might become apparent that $C(x)$ is also already contained in the Abox. So, there is no maintenance effort for the Abox to which $C(x)$ is added.

The conjunction rule is applied with second-highest priority. Although the number of assertions to be handled in a partition is increased, the chance that a clash is detected early is also increased. Since in many contexts the Abox will indeed be consistent, conjunctions have to be "expanded" anyway. So, it is a good heuristic to prefer the conjunction rule over other rules.

In order to reduce the number of Aboxes to be handled as parts of $\mathfrak{A}$, in practical systems, deterministic rules are preferred over nondeterministic ones. In order to reduce memory requirements (and to meet the complexity class of the Abox consistency problem) the so-called *trace technique* has been developed. Employing the trace technique, the disjunction rule is applied before the deterministic value restriction rule. Then, for each existential quantification assertion $(\exists R.C)(x)$, it is ensured that all potentially applicable value restrictions are indeed available in the Abox. Thus, the existential quantification rule can be combined with the value restriction rule. Rather than only adding a concept assertion based on the quantification concept as indicated in the existential quantification rule for a role R, additionally for every value restriction $(\forall R.D_i)(x)$ the assertion $D_i(y)$ is added, with y being the fresh individual introduced by the exists quantification rule. Then all assertions $\{C(y), D_1(y), \ldots D_n(y)\}$ can be treated in isolation. If they turn out not to lead to a clash, the assertions, and all those derived from them, can be removed (and $(\exists R.C)(x)$ must somehow be marked to avoid repetitive rule applications).

In a practical implementation, the trace technique might not be adopted for various reasons. For instance, the removal of assertions might interfere with the idea to reuse of previous computation results, in particular if Tboxes are involved (see below). Or the strategy is to avoid the expansion of disjunctions but check the satisfiability of existential quantifications first.

2.4 Dealing with Indeterminism in a Tableau Algorithm

In the description above, a nondeterministic rule (in $\mathcal{ALC}$ only the disjunction rule) generates a sequence of new Aboxes. If Aboxes are created in a naive way, this can hardly be efficient. Thus, a practical implementation must find a way to implement a structure-sharing strategy for copies of Aboxes in order to avoid structures to be copied repeatedly. Copying complete structures is memory-extensive as well as time-consuming. Even with a structure-sharing

approach, the naive generation of successor Aboxes should be avoided due to the memory-management overhead involved.

Obviously, the disjunction rule does not need to generate successors that immediately lead to a clash. If the disjunction rule would be applicable to some disjunct C_i in an assertion $(\bigvee\{C_1, \ldots, C_n\})(x)$ and $neg(C_i)(x) \in \mathcal{A}$ the corresponding successor Abox does not need to be generated (it will be closed according to the clash rule immediately). The detection of those situations requires some additional machinery in the implementation (boolean constraint propagation, BCP) [9]. Boolean constraint propagation is implemented in all major contemporary tableau-based reasoners. The challenge is to most efficiently determine which disjunctions can be virtually "shrunk" or "eliminated" in this way. Note that a disjunction becomes deterministic if only one disjunct "remains" after BCP, and it can be treated as a conjunction in this case (also w.r.t. priorities in rule applications). It is easy to see that termination and correctness is still fulfilled if boolean constraint propagation is employed.

Further optimizations that also have no impact on the correctness of the algorithm but provide for improved performance for typical-case inputs are possible. For instance, the disjunction rule requires as a precondition that the disjunct $C_i(x)$ is not already in $\mathcal{A}$. Thus, looking for a completion, in a concrete implementation it is advantageous to first apply the disjunction rule to those concept assertions $(\bigvee\{C_1, \ldots, C_n\})(x)$ with disjuncts C_i such that C_i is also mentioned in many other disjunctive assertions for the individual x. The application of the disjunction rule for the other disjunctions for x involving C_i is then "avoided" (due to the precondition of the disjunction rule). Efficiently finding those concepts C_i such that the number of occurrences in all disjunctions applying to an individual x is maximized (or large) is non-trivial, however. There is a tradeoff between the time spent in search for occurrences of a concept assertion $C_i(x)$, management of index structures for speeding up this search process, and the gain of this in terms of or-branching reduction.

Reusing previous results can help finding clashes early. If $\mathcal{A}$ is an Abox, then all Aboxes derived from $\mathcal{A}$ by applying a tableau rule are called *sibling* Aboxes. Information acquired for one successor Abox $\mathcal{A}'$ of $\mathcal{A}$ can be propagated to sibling Aboxes of $\mathcal{A}'$. Let us consider the successor Aboxes of an application of the disjunction rule to an Abox $\mathcal{A}$ again. If it turns out that one of the successor Aboxes $\mathcal{A}'$ with $C_i(x)$ being added contains a clash, then, $neg(C_i)(x)$ can be added to all (open) sibling Aboxes of $\mathcal{A}'$. Again, if $neg(C_i)(x)$ is explicitly present in a sibling Abox, an application of the disjunction rule to the sibling Abox might be prevented and a clash might be revealed earlier. On the negative side it has to be mentioned that applying rules to $neg(C_i)(x)$ in a tableau also causes some overhead. In a practical implementation one might avoid the applications of the rules to assertions added this way (without impact on soundness and completeness).

Up to now, we have considered ways to find a completion earlier (i.e., we attempt to reduce or-branching). This heuristic is useful because the algorithm

terminates if a completion is found. However, it is also possible to find ways to close tableaux early. Consider the following example Abox, which is obviously inconsistent (adapted from [17]).

$$\mathcal{A} = \{(\bigvee\{C_1, D_1\})(i), \ldots (\bigvee\{C_n, D_n\})(i), (\exists R.(\bigwedge\{A, B\}))(i), (\forall R.\neg A)(i)\}$$

We assume that the tableau algorithm applies the disjunction rule to the first disjunction in $\mathcal{A}$. We get two new Aboxes $\mathcal{A}_1$ and $\mathcal{A}_2$. Both Aboxes are supersets of $\mathcal{A}$. In Figure 1, Aboxes are indicated as circles. A set inclusion relation is indicated with a solid arrow (pointing from the superset to the subset). The assertions being added to an Abox w.r.t. its predecessor Abox are written next to the circle used for indicating the Abox. Aboxes that initially are not closed are presented with a bold outline. As indicated in Figure 1, we assume that tableaux are represented using a kind of trie data structure.

In Figure 1 we assume further rule applications to the Aboxes $\mathcal{A}_1, \mathcal{A}_3, \ldots, \mathcal{A}_{2n-1}$, and then $\mathcal{A}_{2n+1}, \mathcal{A}_{2n+2}$ and finally $\mathcal{A}_{2n+3}$. Initially, $\mathcal{A}_{2n+3}$ is assumed to be open. Now, a clash is found and $\mathcal{A}_{2n+3}$ is marked as closed by the clash rule (due to $\neg A(j)$ and $A(j)$ being an element of $\mathcal{A}_{2n+3}$).

The dashed lines (curved) indicate the dependencies of the assertions that are added to the respective Aboxes (not all dependencies are shown for readability reasons). Looking at Figure 1 it should be apparent that the Aboxes $\mathcal{A}_2, \mathcal{A}_4, \ldots, \mathcal{A}_{2n}$, which are still open, will inevitably lead to the same clash.

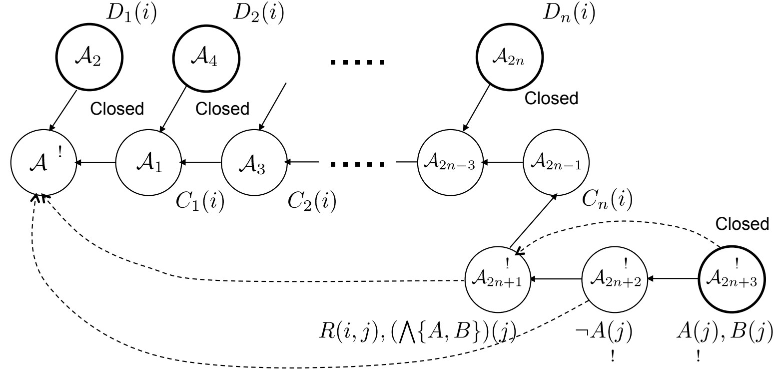

Fig. 1. A clash occurs in tableau, and exploring other open tableaux will not resolve it. The corresponding tableau can be closed even without a clash (see text).

The idea to avoid repeatedly detecting the same clash over and over again is to find a way to close the Aboxes $\mathcal{A}_2, \mathcal{A}_4, \ldots, \mathcal{A}_{2n}$ in advance.

This can be achieved as follows. The clash occurs in $\mathcal{A}_{2n+3}$. The culprit in $\mathcal{A}_{2n+3}$ is $A(j)$. The other culprit $\neg A(j)$ is in $\mathcal{A}_{2n+2}$. Culprit assertions are

indicated with an exclamation mark. Starting with the culprits and following the dashed lines the Aboxes are marked with exclamation markers. See Figure 1 for the final marking in our example. Then, starting from the Abox with the "rightmost" culprit (in our case $\mathcal{A}_{2n+3}$) and following the solid lines in the direction of the arrows, the reachable Aboxes are visited. If an Abox $\mathcal{A}'$ marked with "!" pointing to an Abox $\mathcal{B}$ (with more than one incoming link) is reached, and the leaves of the other Aboxes pointing to the predecessor of $\mathcal{A}'$ are not closed, then the process stops. Then, all leaf Aboxes reachable in the inverse direction of the solid lines from the Abox $\mathcal{B}$ are marked as closed.

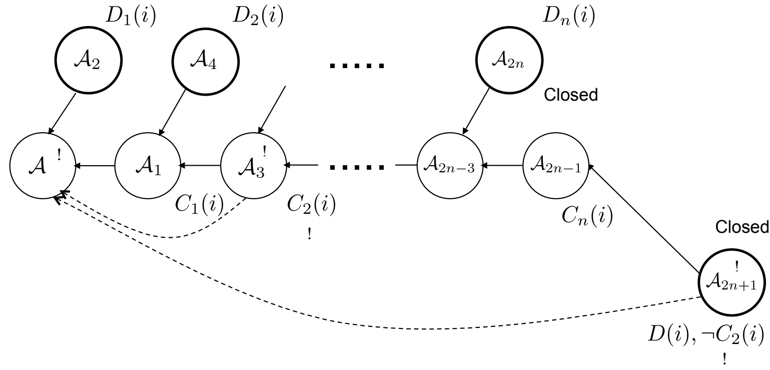

Fig. 2. Examining the dependencies reveals that $\mathcal{A}_4$ must not be automatically closed to retain completeness (see text).

In the example shown in Figure 1 the process stops at $\mathcal{B} = \mathcal{A}$. The closed Aboxes found by following the solid lines in the inverse direction are indicated with a corresponding label "Closed" in Figure 1. Hence, futile rule applications to $\mathcal{A}_2, \mathcal{A}_4, \ldots, \mathcal{A}_{2n}$ are avoided.

A slightly modified example illustrates that the process does not necessarily close all Aboxes but only those which, due to the given culprits, do not lead to completions. The example is given as follows:

$$\mathcal{A} = \{(\bigvee\{C_1, D_1\})(i), \ldots (\bigvee\{C_n, D_n\})(i), (\exists R.(\bigwedge\{\neg C_2, D\}))(i)\}$$

In Figure 2, the situation is shown after a few rule applications. Culprit markers indicate the assertions on which the clash depends. Walking from the rightmost culprit along the solid lines stops at $\mathcal{B} = \mathcal{A}_3$. All leaves reachable from $\mathcal{A}_3$ in the inverse direction of the solid lines are closed. In this example, $\mathcal{A}_2$ and $\mathcal{A}_4$ remain open, which is necessary not to miss a possible completion to be constructed. If all successor Aboxes of an Abox are marked with a clash,

their assertions are also seen as clash culprits and exclamation markers are propagated via the curved dependency links as introduced before [9].

In the literature, the technique described here is known as *backjumping*, which is a restricted form of dependency-directed backtracking [17]. We have presented it here using mathematical structures as part of the branch-and-bound strategy of a tableau algorithm. Note that Aboxes in the trie might indeed be reused, and thus, full dependency-directed backtracking might be achieved (with previous computations being maximally reused).

Usually, a partition for an individual is seen as a graph node with role assertions "pointing" to other graph nodes, and concept assertions defining the so-called "label" of a "node" (see below for a more formal introduction of a label). Hence, from an implementation point of view a tableau is seen as a graph with nodes and edges, both associated with a *label*. For nodes, the label is a set of concepts, and for edges it is a set of roles. To every path in the Abox trie from the root to a leaf (see Figure 1 or Figure 2) there corresponds a particular graph. For understanding tableau algorithms one can switch between the graph view and the tableau (or Abox) view. The graph corresponds to a model (see the construction of the canonical model mentioned above).

3 Dealing with Tboxes

The tableau algorithm introduced above must be extended to work with non-empty Tboxes. In theory, it is possible to transform all GCIs of the Tbox into a single GCI of the form $\top \sqsubseteq M$. The transformation is very simple. Instead of writing a GCI as $C_i \sqsubseteq D_i$ one could write $\top \sqsubseteq \neg C_i \sqcup D_i$, and thus, M is the conjunction $\bigwedge_i \{M_i\}$ of all $M_i = \neg C_i \sqcup D_i$ stemming from the GCIs in the Tbox (M and M_i are called *global constraints*). With the Tbox transformed into $\top \sqsubseteq \bigwedge\{M_1, \ldots M_n\}$ we can see that the restriction M on the righthand side applies to all domain objects. The transformation is called *internalization* in the literature [8]. The tableau algorithm is extended with a new rule which adds $M(x)$ if there is an individual x mentioned in a tableau in which $M(x)$ is not already present.

- **GCI rule**: If $C(x) \in \mathcal{A}$ or $R(x, y) \in \mathcal{A}$ or $R(y, x) \in \mathcal{A}$ and $M(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{M(x)\}$.

A problem with this rule is that the algorithm then does not terminate. M might contain existential quantifications which cause new individuals y to be created, for which $M(y)$ is added and so on (infinite and-branching occurs). Some form of blocking must be enforced (see e.g., [8]).

The tableau algorithm can be slightly changed to exploit these insights. We give a definition for the label of a partition. The *label* of a partition for an individual x is defined as $\{C \mid C(x) \in P_x\}$. Now, if there exists an individual partition P_k and there is no rule applicable to P_k nor to all partitions that

depend on P_k, then no rule need to be applied to an assertion in a partition P_l if $label(P_l) \subseteq label(P_k)$ and k, l are fresh variables (otherwise, partitions for old individuals might block each other). We say P_l is *blocked* by the *witness* P_k. The witness must be a fresh individual. Note that if there is a clash detected for k the whole tableau is closed and no rule is applied to l anyway. The condition $label(P_l) \subseteq label(P_k)$ might become false if, due to other rule applications, new assertions are added for the individuals k or l in P_k and P_l, respectively. So, blocking conditions must be dynamically checked.

In order to show soundness in case of blocked partitions, one constructs a canonical interpretation for an individual i for which there exists a blocked partition P_i by defining tuples $(i, x) \in R^{\mathcal{I}}$ for every assertion $R(w, x)$ related to the witness w of i (for details see, e.g, [8]).[4]

The precondition that no rule is applicable to P_k and all partitions that depend on P_k could be dropped. But then one must make sure that P_l is a strict subset of P_k. Otherwise, to both partitions no rule would be applied and the algorithm would become incomplete. For more expressive logics, more expressive blocking conditions have to be defined (e.g., [21]). In the literature it has been shown that the extended algorithm is sound and complete for arbitrary $\mathcal{ALC}$ Tboxes.

One drawback from a practical point of view is that now a possibly large set of disjunctions are introduced for every individual mentioned in a tableau, since $M_i = \neg C_i \sqcup D_i$. Keeping in mind that, e.g., boolean constraint propagation is employed to deal with disjunctions in a practical system, it becomes clear that disjunctions always involve "heavy-weight" methods in a practical implementation of the tableau algorithm. For specific forms of GCIs, the disjunctions do not have to be explicitly generated, however. This is explained in the next subsection.

3.1 Lazy unfolding

Let us assume, there is a global constraint of the form $L_i = \neg A \sqcup C$ in M such that A is an atomic concept description. Rather than adding this global constraint to every individual x, the idea is to only add $C(x)$ if adding $\neg A(x)$ would lead to a clash. For those global constraints, one can implicitly assume that individuals are instances of $\neg A$ "if not stated otherwise" (see the construction of the canonical interpretation). The global constraint L_i can be handled "in a lazy way" by a new rule which "unfolds" an assertion $A(x)$ in a tableau (cf. [3]).

We need some definitions for specifying the exact conditions under which soundness and completeness can be guaranteed. An atomic concept description A directly refers to an atomic description B if there exists a GCI $A \sqsubseteq C$

[4] We use a way to construct the canonical interpretation that already considers additional concept constructors such as, say, number restrictions. In case of $\mathcal{ALC}$ it would be possible to map i to its witness w in the canonical interpretation.

such that B is mentioned in C but not in the scope of an exist quantification or value restriction. A refers to B if it directly refers to B or there exists an atomic concept description B' such that A refers to B' and B' refers to B.

A global constraint of the form $L_i = \neg A \sqcup C$ need not be handled as a disjunction if A is an atomic concept description and C does not refer to A. Let us assume that global constraints that satisfy the conditions introduced above for L_i are collected not in M but into a set L. There must be no other global constraint in L with a disjunct $\neg A$ or disjunct A. Then, the following rule is used to deal with global constraints in L [3, 21].

- **Lazy unfolding rule 1**: If $A(x) \in \mathcal{A}$ and $(\neg A \sqcup C) \in L$ and $C(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(x)\}$.

In other words, only if there is an assertion $A(x)$ in a tableau, then $C(x)$ must be added because assuming $\neg A(x)$ would lead to a clash.

In case we also collect assertions of the form $A \sqcup C$ into the set of concepts L (and not into M) another rule must be added. Corresponding restrictions as for $\neg A \sqcup C$ apply.

- **Lazy unfolding rule 2**: If $\neg A(x) \in \mathcal{A}$, $(A \sqcup C) \in L$ and $C(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(x)\}$.

Lazy unfolding exploits the fact that one can safely assume that a domain object which is not explicitly enforced to be in $A^{\mathcal{I}}$ (or $(\neg A)^{\mathcal{I}}$ in the second case) is an element of $(\neg A)^{\mathcal{I}}$ (or $A^{\mathcal{I}}$ in the second case). See [22] for details.

3.2 GCI Absorption

Global constraints in L are handled more effectively. If, initially, the global constraints are not of the form that they can be put into L but must be stored in M, the goal is to transform them in a way that a maximum number of global constraints can be put into L, and possibly none must be kept in M, without changing the semantics of the Tbox. This transformation process is known as GCI absorption (see [15, 16, 22] for details).

In some cases, still some global constraints remain in M even if GCIs are transformed as describe above, unfortunately. For instance, this happens if there are two GCIs of the form $A \sqsubseteq (\exists R.A) \sqcap C$ and $\exists R.A \sqsubseteq A$ in a Tbox. The latter kind of GCI is only relevant for an individual x if there exists an assertion $R(x, y)$ in tableau.

For $\exists R \top. \sqsubseteq C$ an effective treatment is possible. The same holds for range restrictions $\top \sqsubseteq \forall R.C$. Let $domain(R)$ and $range(R)$ denote sets of concepts (initially empty). We assume that all $(\forall R.\bot) \sqcup C$ are removed from M, and for each $(\forall R.\bot) \sqcup C$ removed, C is added to $domain(R)$. In addition, all $\bot \sqcup \forall R.C$ are removed from M, and for each $\bot \cup \forall R.C$ removed, there is C added to $range(R)$.

- **Domain restriction rule**: If $R(x, y) \in \mathcal{A}$ and $C \in domain(R)$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(x)\}$.
- **Range restriction rule**: If $R(x, y) \in \mathcal{A}$ and $C \in range(R)$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(y)\}$.

See [10] for a description of the initial idea and [31] for an analysis of this technique. However, for $\exists R.A \sqsubseteq A$ a disjunction has to be added (to any individual for which there exists a role successor), which still could cause a combinatorial explosion if the wrong choice is made in a practical system. For instance, this could happen for the following Abox.

$$\mathcal{A} = \{\neg A(x_0), R(x_0, x_1), R(x_1, x_2), \ldots R(x_{n-1}, x_n), A(x_n)\}$$

For all $x_i, i \in \{0, \ldots n-1\}$, a disjunction $(\forall R.\neg A) \sqcup A$ would be asserted, and choice points are set up. We assume that rules are first applied to the tableaux created for $\forall R.\neg A$. After several rule applications, a clash w.r.t. $\neg A(x_n)$ and $A(x_n)$ would be detected. The situation could be even worse, if there was a GCI $B \sqcap \exists R.A \sqsubseteq A$ with B being an atomic concept for which there exists a GCI $B \sqsubseteq D$. Thus, there is no way to absorb $B \sqcap \exists R.A \sqsubseteq A$ into L using the above-mentioned techniques.

In [23], a new transformation called *binary absorption* has been introduced to tackle this problem. Applying this transformation requires a new rule to be added to the tableau algorithm. This is discussed in the next subsection.

3.3 Binary Absorption

A GCI $B \sqcap \exists R.A \sqsubseteq A$ should not be transformed into a disjunction to be placed in M. It can be transformed into

$$\exists R^{-1}.\top \sqsubseteq A_1$$

$$A_1 \sqcap A \sqsubseteq \forall R^{-1}.A_2$$

$$A_2 \sqcap B \sqsubseteq A$$

where R^{-1} denote the inverse of role R. The idea is to introduce a marker $A_1(y)$ for every y for which there is an assertion $R(x, y)$. The first GCI of the list above can be handled effectively by absorbing it into $domain(R)$ as described before. A_2 is a marker indicating an instance of $\exists R.A$ (see the second GCI). The third GCI now enforces y to be an instance of A in the tableau. In order to deal with a conjunction of two atomic concepts on the lefthand side of a GCI the lazy unfolding rules can be extended as follows [23].

- **Lazy unfolding rule 3**: If $\{A_1(x), A_2(x)\} \subseteq \mathcal{A}$, $(\not A_1 \sqcap A_2) \sqcup C) \in L$ and $C(x) \notin \mathcal{A}$, then replace $\mathcal{A}$ with $\mathcal{A} \cup \{C(x)\}$.

No disjunctions of this particular type have to be handled after the transformation is applied. Soundness and completeness of this approach have been shown in [23]. A disadvantage is that the tableau algorithm now must also handle inverse roles (denoted as R^{-1} in the GCIs above). Extensions to $\mathcal{A}$ are briefly discussed in the next section.

4 Tableau Structures for Subsumption Problems

Tableau-based reasoning can be exploited for solving other reasoning problems as well. First, we consider the subsumption problem, then we turn to the instance problem and the retrieval problem.

Subsumption problems occur very frequently if the so-called taxonomy of a Tbox is computed. This is usually done at ontology development time in order to check for modeling errors (unsatisfiable atomic concept descriptions, unwanted subsumption relationships etc.). The taxonomy of a Tbox is a graph where the nodes are the atomic concept descriptions mentioned in the Tbox (including $\top$ and $\bot$), and the edges indicate whether a node is a most-specific subsumer of another node.

For expressive languages such as $\mathcal{ALC}$ the subsumption problem $A \sqsubseteq_? B$ can be reduced to the Abox consistency problem $\{(A \sqcap \neg B)(i)\}$ for some individual i. If the Abox is inconsistent the subsumption relation holds, otherwise it does not hold. For practical Tboxes, GCIs usually involve conjunctions on the righthand side. Thus, if B is negated, disjunctions have to be handled by the tableau algoithm, which might lead to "unfocused" applications of the rules. For computing the taxonomy, many similar subsumption problems of the form $A_i \sqsubseteq_? B$ have to be solved, and hence, many Abox consistency problems $\{(A_i \sqcap \neg B)(i)\}$ are the consequence. In almost all cases the subsumption relation between A and B does not hold, and hence, the Abox is likely to be shown to be consistent. Quite some number of applications of tableau rules might be required, however (with large or-branching, and for realisistic ontology considerable and-branching as well).

Therefore, in [15] the following technique was developed. Since $\neg B$ is used many times, its satisfiability is tested in isolation. Usually, $\neg B$ is satisfiable in practical contexts (otherwise, B would a synonym to $\top$). The test whether $\{\neg B(i)\}$ is consistent leads to a consistent Abox such that a label $\mathcal{L}_1$ is defined for i. We call this label a *pseudo model* (for an atomic concept description). The same is done now for A_i (let the label be called $\mathcal{L}_i$).

In [15] a process called *model merging* is defined. The idea of this process is to show non-subsumption by comparing the labels. Four conditions must be satisfied in order to conclude non-subsumption.

- For every $A \in \mathcal{L}_1$ there does not exist an $\neg A \in \mathcal{L}_i$
- For every $\neg A \in \mathcal{L}_1$ there does not exist an $A \in \mathcal{L}_i$
- For every $\exists R.C \in \mathcal{L}_1$ there does not exist an $\forall R.D \in \mathcal{L}_i$
- For every $\forall R.C \in \mathcal{L}_1$ there does not exist an $\exists R.D \in \mathcal{L}_i$

If non-subsumption cannot be concluded, the "full" test whether $\{(C \sqcup \neg D)(i)\}$ is consistent is performed. However, practical experiments have shown that this is not often required [15, 16], so there is hardly any overhead introduced by the model merging process. There is almost no search involved in comparing the labels in the way defined above if the assertions in the labels are indexed appropriately.

5 Conclusion

The tableau algorithm introduced in this section can be extended to deal with additional concept and role constructors. With the addition of new constructors, the rule application strategy becomes important for termination and correctness, not only for optimization. Furthermore, the blocking condition might become more complex. For instance, the following constructs have been investigated in the literature and tableau algorithms have bee specified:

- concrete domains (with feature composition) [4],
- qualifying number restrictions [14],
- number restrictions plus role conjunctions and GCIs [8],
- transitive roles [27],
- transitive roles, role hierarchies, GCIs, and features [16],
- transitive roles, role hierarchies, GCIs, plus number restrictions and Aboxes [12],
- transitive roles, role hierarchies, GCIs, number restrictions and Aboxes plus concrete domains without feature composition [11],
- transitive roles, role hierarchies, GCIs, Aboxes, plus qualifying number restrictions and inverse roles [21],
- nominals [28, 1, 30, 20],
- role axioms [18],
- concrete domains with role composition for description logics with GCIs [25].

For almost all of the language features in this list, efficient implementations based on tableau algorithms are available.

Tableau-based reasoning methods are very effective for concept satisfiability checking [19] as well as for Tbox-based reasoning tasks [16, 13, 32]. Even for some specific Tboxes for which it was assumed that resolution-based reasoning methods show better behavior, new techniques such as binary absorption have shown that tableau-based methods can exploit similar structures. Tableau-based Tbox reasoners such as FaCT++, Pellet or RacerPro are the fastest systems for expressive description logics for a wide range of expressive Tboxes that regularly occur in practice.

Acknowledgments

We would like to thank Sebastian Wandelt and Michael Wessel for comments on a draft of this chapter.

References

1. Franz Baader, Martin Buchheit, and Bernhard Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1–2):195–213, 1996.

2. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
3. Franz Baader, Enrico Franconi, Bernhard Hollunder, Bernhard Nebel, and Hans-Jürgen Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.
4. Franz Baader and Philipp Hanschke. A schema for integrating concrete domains into concept languages. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91)*, pages 452–457, 1991.
5. Franz Baader, Ralf Küsters, and Frank Wolter. Extensions to description logics. In *[2]*, pages 219–261. 2003.
6. Franz Baader and Werner Nutt. Basic description logics. In *[2]*, pages 43–95. 2003.
7. Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
8. Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. of Artificial Intelligence Research*, 1:109–138, 1993.
9. J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
10. V. Haarslev and R. Möller. Practical reasoning in racer with a concrete domain for linear inequations. In *Proceedings of the International Workshop on Description Logics (DL-2002), Toulouse, France, April 19-21*, pages 91–98, 2002.
11. V. Haarslev, R. Möller, and M. Wessel. The description logic alcnhr+ extended with concrete domains. Technical Report FBI-HH-M-290/00, University of Hamburg, Computer Science Department, 2000.
12. Volker Haarslev and Ralf Möller. Expressive abox reasoning with number restrictions, role hierarchies, and transitively closed roles. In *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 273–284. Morgan Kaufmann, 2000.
13. Volker Haarslev and Ralf Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, pages 161–168, 2001.
14. Bernhard Hollunder and Franz Baader. Qualifying number restrictions in concept languages. In *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pages 335–346, 1991.
15. Ian Horrocks. Optimisation techniques for expressive description logics. Technical Report UMCS-97-2-1, University of Manchester, Department of Computer Science, 1997.
16. Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
17. Ian Horrocks. Implementation and optimization techniques. In *[2]*, pages 306–346. 2003.
18. Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

19. Ian Horrocks and Peter F. Patel-Schneider. Optimizing description logic subsumption. *J. of Logic and Computation*, 9(3):267–293, 1999.
20. Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 448–453, 2005.
21. Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David McAllester, editor, *Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 482–496. Springer, 2000.
22. Ian Horrocks and Stephan Tobies. Reasoning with axioms: Theory and practice. In *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 285–296, 2000.
23. Alexander. K. Hudek and Grant Weddell. Binary absorption in tableaux-based reasoning for description logics. In *Proc. of the 2006 Description Logic Workshop (DL 2006)*. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-189/`, 2006.
24. C. Lutz. PSPACE reasoning with the description logic $\mathcal{ALCF}(\mathcal{D})$. *Logic Journal of the IGPL*, 10(5):535–568, 2002.
25. C. Lutz and M. Milicic. A tableau algorithm for description logics with concrete domains and general tboxes. *Journal of Automated Reasoning*, 2006. To appear.
26. Ralf Möller and Volker Haarslev. Description logic systems. In *[2]*, pages 282–305. 2003.
27. Ulrike Sattler. A concept language extended with different kinds of transitive roles. In Günter Görz and Steffen Hölldobler, editors, *Proc. of the 20th German Annual Conf. on Artificial Intelligence (KI'96)*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 333–345. Springer, 1996.
28. Andrea Schaerf. Reasoning with individuals in concept languages. In *Proc. of the 3rd Conf. of the Ital. Assoc. for Artificial Intelligence (AI*IA'93)*, Lecture Notes in Artificial Intelligence. Springer, 1993.
29. Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.
30. Stephan Tobies. The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. *J. of Artificial Intelligence Research*, 12:199–217, 2000.
31. Dmitry Tsarkov and Ian Horrocks. Efficient reasoning with range and domain constraints. In *Proc. of the 2004 Description Logic Workshop (DL 2004)*, pages 41–50, 2004.
32. Dmitry Tsarkov, Ian Horrocks, and Peter F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *J. of Automated Reasoning*, 39(3):277–316, 2007.