Optimising Terminological Reasoning for Expressive Description Logics

Dmitry Tsarkov, Ian Horrocks and Peter F. Patel-Schneider
School of Computer Science
University of Manchester, UK,
and
Bell Labs Research
Lucent Technologies, USA

March 5, 2007

Abstract. Tableau algorithms are currently the most widely-used and empirically the fastest algorithms for reasoning in expressive Description Logics, including the important Description Logics $\mathcal{SHIQ}$ and $\mathcal{SHOIQ}$. Achieving a high level of performance on terminological reasoning in expressive Description Logics when using tableau-based algorithms requires the incorporation of a wide variety of optimisations. The Description Logic system FaCT++ implements a wide variety of such optimisations, some present in other reasoners and some novel or refined in FaCT++.

Keywords: Description Logic, Reasoning Systems, Optimisations

1. Introduction

Description Logics (DLs) are a family of logic based knowledge representation formalisms. Although they have a range of applications (e.g., configuration (McGuinness and Wright, 1998) and reasoning with database schemas and queries (Calvanese et al., 1998b; Calvanese et al., 1998a)), they are perhaps best known as the basis for widely used ontology languages such as OIL (Fensel et al., 2001), DAML+OIL (Horrocks et al., 2002) and OWL (Horrocks et al., 2003). As well as DLs providing the formal underpinnings for these ontology languages (by means of the declarative semantics of DLs), DL systems are also used to provide computational services for ontology tools and applications (Knublauch et al., 2004; Rector, 2003). Such services typically include (at least) computing the subsumption hierarchy and checking the consistency of named concepts in an ontology.

Most modern DL systems are based on tableau algorithms. Such algorithms were first introduced in a seminal paper by Schmidt-Schauß and Smolka (Schmidt-Schauß and Smolka, 1991), and subsequently extended to deal with a wide range of different logics (Baader et al., 2003). Many systems now implement the $\mathcal{SHIQ}$ DL, for which a tableau algorithm was first presented by Horrocks, Sattler, and Tobies (Horrocks et al., 1999), or its extension $\mathcal{SHOIQ}$, whose tableau algorithm was recently described by Horrocks and Sattler (Horrocks and Sattler, 2005). These two DLs are very expressive, and correspond closely to the OWL ontology language. (Reasoning in OWL Lite can be reduced to reasoning in $\mathcal{SHIQ}$ and reasoning in OWL DL can be reduced to reasoning in $\mathcal{SHOIQ}$ (Horrocks and Patel-Schneider, 2003).) In spite of the high worst case complexity of the satisfiability/subsumption problem for this logic (ExpTime-complete for $\mathcal{SHIQ}$ and NExpTime-complete for $\mathcal{SHOIQ}$), highly optimised implementations of the tableau algorithms for $\mathcal{SHIQ}$ and $\mathcal{SHOIQ}$ have been shown to work well in many realistic (ontology) applications (Horrocks, 1998; Pan, 2005; Stevens et al., 2002; Wolstencroft et al., 2005).

Optimisation is crucial to the viability of systems that employ tableau-based algorithms for DL reasoning: in experiments using both artificial test data and application ontologies, (relatively) unoptimised systems performed very badly, often being (at least) several orders of magnitude slower than optimised systems; in many cases, hours of processing time (in some cases even hundreds of hours) proved insufficient for unoptimised systems to solve problems that took only a few milliseconds for an optimised system (Massacci, 1999; Horrocks and Patel-Schneider, 1998).

The optimisation of tableau-based systems has been the subject of extensive study over the course of the last fifteen years (Baader et al., 1994; Horrocks, 1998; Haarslev and Möller, 2001a; Haarslev and Möller, 2001b; Haarslev et al., 2001b; Horrocks, 2003; Horrocks and Sattler, 2002; Tsarkov and Horrocks, 2005b; Sirin et al., 2005a; Haarslev et al., 2005; Sirin et al., 2006), and modern systems typically employ a wide range of optimisations, including (at least) those described by Baader *et al* (Baader et al., 1994) and Horrocks and Patel-Schneider (Horrocks and Patel-Schneider, 1999).

In this paper we describe the optimisation techniques employed in our FaCT++ reasoner (Tsarkov and Horrocks, 2006). While focusing mainly on novel techniques and significant refinements and extensions of previously known techniques, we have also described the "standard" optimisations used in most implemented systems, as well as some simple optimisations that are widely employed but often not reported in the literature. In this way we hope to provide both a self contained report on the novel techniques developed in the FaCT++ system and a reasonably comprehensive survey of the optimisation techniques employed in state of the art DL reasoning systems.

Many of the techniques we will describe could be applied to a wide range of DLs. We will, however, focus on $\mathcal{SHOIQ}$, because

– the expressive power of $\mathcal{SHOIQ}$ subsumes that of most of the DLs discussed in the literature or implemented in DL reasoners;

– $\mathcal{SHOIQ}$ is the logic implemented in our FaCT++ system, and other state-of-the-art DL reasoners such as Pellet (Sirin et al., 2005b) and Racer (Haarslev and Möller, 2003) implement DLs very close to $\mathcal{SHOIQ}$; and

– as the basis of the W3C OWL web ontology language, $\mathcal{SHOIQ}$ is now very widely used in practice.

DLs usually distinguish between the terminological part of a knowledge base (called the TBox), which describes the structure of the domain of discourse in terms of concepts and roles, and the assertional part (called the ABox), which describes some particular situation in terms of instances of concepts and roles. FaCT++ is primarily designed to support TBox reasoning, as this kind of reasoning is widely used, e.g., in ontology design and maintenance tools such as Protégé (Protégé, 2003) and Swoop (Kalyanpur et al., 2005). However, the expressive power of $\mathcal{SHOIQ}$ is such that it blurs the usual distinction between TBox and ABox, and FaCT++ exploits this to support ABox reasoning. This simple approach would, however, clearly not scale to very large ABoxes, and for a discussion of implementation and optimisation techniques designed to address the problems of reasoning with large ABoxes, the reader is referred to work on optimising ABox reasoning in the Racer system (Haarslev and Möller, 2000; Haarslev and Möller, 2001a; Haarslev and Möller, 2004; Chen et al., 2005).

2. Preliminaries

We present here a brief introduction to DL syntax and semantics (in particular the syntax and semantics of $\mathcal{SHOIQ}$), and to tableau algorithms for DLs (in particular for $\mathcal{SHOIQ}$). We will not attempt to give a comprehensive description of the algorithms, or present any proofs; rather we will provide just such details of the basic, unoptimised method as will be necessary in order to understand the subsequent sections on optimisation techniques. The interested reader is referred to other work on tableau algorithms (such as (Baader and Sattler, 2001)) for further details on tableau algorithms in general, and the introduction of the $\mathcal{SHOIQ}$ tableau algorithm (Horrocks and Sattler, 2005) for further details on the $\mathcal{SHOIQ}$ algorithm.

2.1. $\mathcal{SHOIQ}$ Syntax and semantics

$\mathcal{SHOIQ}$ is a very expressive DL that, in addition to the standard Boolean connectives, allows for transitive roles, a hierarchy of sub- and super-roles, inverse (sometimes called converse) roles, qualified cardinality constraints, and nominals (i.e., the ability to refer to individuals in concept expressions). This last feature leads to a blurring of the usual DL distinction between TBox (a set of axioms concerning classes and roles) and ABox (a set of axioms concerning individuals); we will return to this point when discussing the syntax and semantics of $\mathcal{SHOIQ}$ knowledge bases.

DEFINITION 1. *Let* $\mathbf{R}$ *be a set of* role names *with transitive role names* $\mathbf{R}_+ \subseteq \mathbf{R}$*. The set of* $\mathcal{SHOIQ}$-roles *(or* roles *for short) is* $\mathbf{R} \cup \{R^- \mid R \in \mathbf{R}\}$*. A* role inclusion axiom *is of the form* $R \sqsubseteq S$*, for two roles* R *and* S*. A* role hierarchy *is a finite set of role inclusion axioms.*

An interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ *consists of a non-empty set* $\Delta^\mathcal{I}$*, the* domain *of* $\mathcal{I}$*, and a function* $\cdot^\mathcal{I}$ *which maps every role to a subset of* $\Delta^\mathcal{I} \times \Delta^\mathcal{I}$ *such that, for* $P \in \mathbf{R}$ *and* $R \in \mathbf{R}_+$*,*

$$\langle x, y\rangle \in P^\mathcal{I} \text{ iff } \langle y, x\rangle \in P^{-\mathcal{I}}, \text{ and}$$
$$\text{if } \langle x, y\rangle \in R^\mathcal{I} \text{ and } \langle y, z\rangle \in R^\mathcal{I} \text{ then } \langle x, z\rangle \in R^\mathcal{I}.$$

An interpretation $\mathcal{I}$ satisfies a role hierarchy $\mathcal{R}$ *if* $R^\mathcal{I} \subseteq S^\mathcal{I}$ *for each* $R \sqsubseteq S \in \mathcal{R}$*; such an interpretation is called a* model *of* $\mathcal{R}$*.*

We introduce some notation to make the following considerations easier.

1. The inverse relation on roles is symmetric, so to avoid considering roles such as R^{--}, we define a function Inv which returns the inverse of a role. For $R \in \mathbf{R}$, $\mathsf{Inv}(R) := R^-$ and $\mathsf{Inv}(R^-) := R$.

2. Since set inclusion is transitive and $R^\mathcal{I} \subseteq S^\mathcal{I}$ implies $\mathsf{Inv}(R)^\mathcal{I} \subseteq \mathsf{Inv}(S)^\mathcal{I}$, for a role hierarchy $\mathcal{R}$, we introduce $\overset{*}{\sqsubseteq}_\mathcal{R}$ as the transitive-reflexive closure of $\sqsubseteq$ on $\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$. We use $R \equiv_\mathcal{R} S$ as an abbreviation for $R \overset{*}{\sqsubseteq}_\mathcal{R} S$ and $S \overset{*}{\sqsubseteq}_\mathcal{R} R$.

3. Obviously, a role R is transitive if and only if its inverse $\mathsf{Inv}(R)$ is transitive. However, in cyclic cases such as $R \equiv_\mathcal{R} S$, S is transitive if R or $\mathsf{Inv}(R)$ is a transitive role name. In order to avoid these case distinctions, the function Trans returns true iff R is a transitive role—regardless whether it is a role name, the inverse of a role name, or equivalent to a transitive role name (or its inverse): $\mathsf{Trans}(S, \mathcal{R}) := \mathrm{true}$ if, for some R with $R \equiv S$, $R \in \mathbf{R}_+$ or $\mathsf{Inv}(R) \in \mathbf{R}_+$; $\mathsf{Trans}(S, \mathcal{R}) := \mathrm{false}$ otherwise.

4. A role R is called *simple* w.r.t. $\mathcal{R}$ iff $\mathsf{Trans}(S, \mathcal{R}) = \mathrm{false}$ for all $S \overset{*}{\sqsubseteq}_\mathcal{R} R$.

5. In the following, if $\mathcal{R}$ is clear from the context, then we may abuse our notation and use $\overset{*}{\sqsubseteq}$ and $\mathsf{Trans}(S)$ instead of $\overset{*}{\sqsubseteq}_\mathcal{R}$ and $\mathsf{Trans}(S, \mathcal{R})$.

DEFINITION 2. *Let* N_C *be a set of* concept names *with a subset* $N_I \subseteq N_C$ *of* nominals*. The set of* $\mathcal{SHOIQ}$-concepts *(or* concepts *for short) is the smallest set such that*

1. *every concept name* $C \in N_C$ *is a concept,*

2. *if* C *and* D *are concepts and* R *is a role, then* $(C \sqcap D)$*,* $(C \sqcup D)$*,* $(\neg C)$*,* $(\forall R.C)$*, and* $(\exists R.C)$ *are also concepts (the last two are called universal and existential restrictions, resp.), and*

3. *if C is a concept, R is a simple role*[1] *and $n \in \mathbb{Z}^+$, then $(\leqslant nR.C)$ and $(\geqslant nR.C)$ are also concepts (called atmost and atleast restrictions, resp.).*

Concepts that are not concept names are referred to as complex *concepts. The interpretation function $\cdot^{\mathcal{I}}$ of an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ maps, additionally, every concept to a subset of $\Delta^{\mathcal{I}}$ such that*

$$
\begin{aligned}
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}}, \quad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}},\\
\neg C^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}, \qquad\qquad \sharp o^{\mathcal{I}} = 1 \textit{ for all } o \in N_I,\\
(\exists R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(x, C) \neq \emptyset\},\\
(\forall R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(x, \neg C) = \emptyset\},\\
(\leqslant nR.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \sharp R^{\mathcal{I}}(x, C) \leqslant n\}, \textit{ and}\\
(\geqslant nR.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \sharp R^{\mathcal{I}}(x, C) \geqslant n\},
\end{aligned}
$$

where $\sharp M$ is the cardinality of a set M and $R^{\mathcal{I}}(x, C)$ is defined as $\{y \mid \langle x, y\rangle \in R^{\mathcal{I}}$ and $y \in C^{\mathcal{I}}\}$.

DEFINITION 3. *A $\mathcal{SHOIQ}$ knowledge base (KB) is a pair $\langle \mathcal{T}, \mathcal{A}\rangle$, where*

– *$\mathcal{T}$ (the TBox) is a set of* general concept inclusion *(GCI) axioms of the form $C \sqsubseteq D$, where C and D are (possibly complex) concepts, and*

– *$\mathcal{A}$ (the ABox) is a set of axioms of the form $i : C$ and $(i, j) : R$, where C is a (possibly complex) concept, R is a role, and $\{i, j\} \subseteq N_I$ are nominals.*

An interpretation $\mathcal{I}$ satisfies *a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it satisfies an axiom $i : C$ if $i^{\mathcal{I}} \subseteq C^{\mathcal{I}}$; and it satisfies an axiom $(i, j) : R$ if for some $\langle x, y\rangle \in R^{\mathcal{I}}$, $i^{\mathcal{I}} = \{x\}$ and $j^{\mathcal{I}} = \{y\}$. An interpretation $\mathcal{I}$* satisfies *a TBox $\mathcal{T}$ (resp. an ABox $\mathcal{A}$) if it satisfies each axiom in $\mathcal{T}$ (resp. $\mathcal{A}$), and it satisfies a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A}\rangle$ if it satisfies both $\mathcal{T}$ and $\mathcal{A}$; such an interpretation is called a* model of *$\mathcal{T}$ (resp. $\mathcal{A}$, $\mathcal{K}$).*

Note that the use of nominals instead of individuals in ABox axioms leads to the slightly non-standard semantics, but is otherwise insignificant.[2]

As mentioned above, nominals blur the distinction between TBox and ABox. ABox axioms can be transformed into TBox axioms: it is easy to see that an interpretation $\mathcal{I}$ satisfies an axiom $i : C$ iff it satisfies $i \sqsubseteq C$, and it satisfies an axiom $(i, j) : R$ iff it satisfies $i \sqsubseteq \exists R.j$. It is, therefore, possible (and convenient) to think of a $\mathcal{SHOIQ}$ KB as consisting only of a TBox.

[1] Restricting number restrictions to simple roles is required in order to yield a decidable logic (Horrocks et al., 1999).

[2] In the standard DL semantics individuals are the named elements of the interpretation domain.

DEFINITION 4. *A TBox $\mathcal{T}$ is* satisfiable w.r.t. a role hierarchy $\mathcal{R}$ *if there is a model $\mathcal{I}$ of $\mathcal{T}$ and $\mathcal{R}$. A concept C is* satisfiable w.r.t. a role hierarchy $\mathcal{R}$ and a TBox $\mathcal{T}$ *if there is a model $\mathcal{I}$ of $\mathcal{R}$ and $\mathcal{T}$ with $C^{\mathcal{I}} \neq \emptyset$. Such an interpretation is called a* model of C *w.r.t. $\mathcal{R}$ and $\mathcal{T}$. A concept D* subsumes *a concept C w.r.t. $\mathcal{R}$ and $\mathcal{T}$ (written $C \sqsubseteq_{\mathcal{R},\mathcal{T}} D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds in every model $\mathcal{I}$ of $\mathcal{R}$ and $\mathcal{T}$. Two concepts C, D are* equivalent *w.r.t. $\mathcal{R}$ and $\mathcal{T}$ (written $C \equiv_{\mathcal{R},\mathcal{T}} D$) if they are mutually subsuming w.r.t. $\mathcal{R}$ and $\mathcal{T}$.*

Note that, as usual, subsumption and satisfiability can be reduced to each other: $C \sqsubseteq_{\mathcal{R},\mathcal{T}} D$ if $(C \sqcap \neg D)$ is not satisfiable w.r.t. $\mathcal{R}$ and $\mathcal{T}$; and C is not satisfiable w.r.t. $\mathcal{R}$ and $\mathcal{T}$ if $C \sqsubseteq_{\mathcal{R},\mathcal{T}} \bot$. Moreover, the satisfiability of a concept w.r.t. a role hierarchy and TBox can be reduced to the satisfiability of a TBox w.r.t. a role hierarchy: C is satisfiable w.r.t. $\mathcal{R}$ and $\mathcal{T}$ if $\mathcal{T} \cup \{o \sqsubseteq C\}$ is satisfiable w.r.t. $\mathcal{R}$, where o is a "fresh" nominal, i.e., one that does not occur in $\mathcal{T}$. When $\mathcal{R}$ is obvious from the context (or assumed to be empty) we will talk about TBox satisfiability; when both $\mathcal{R}$ and $\mathcal{T}$ are obvious from the context (or assumed to be empty) we will talk about concept satisfiabiliy and subsumption (written $C \sqsubseteq D$).

2.2. A Tableau Algorithm for $\mathcal{SHOIQ}$

The basic idea behind the tableau algorithm for $\mathcal{SHOIQ}$ TBox satisfiability is to take as input a TBox $\mathcal{T}$ and role hierarchy $\mathcal{R}$, and to try to prove the satisfiability of $\mathcal{T}$ w.r.t. $\mathcal{R}$ by demonstrating the existence of a model $\mathcal{I}$ of $\mathcal{T}$ w.r.t. $\mathcal{R}$. This is done by syntactically decomposing $\mathcal{T}$ so as to derive constraints on the structure of such a model. For example, if a nominal o occurs in $\mathcal{T}$, then any model of $\mathcal{T}$ must, by definition, contain some individual x such that $o^{\mathcal{I}} = \{x\}$, and if $o \sqsubseteq \exists R.D$ is an axiom in $\mathcal{T}$, then the model must also contain an individual y such that $\langle x, y\rangle \in R^{\mathcal{I}}$ and y is an element of $D^{\mathcal{I}}$; if D is non-atomic, then continuing with the decomposition of D would lead to additional constraints. The process fails if the constraints include a *clash* (an obvious contradiction), e.g., if some individual z must be an element of both C and $\neg C$ for some concept C. The algorithm is designed so that it is guaranteed to terminate, and guaranteed to construct a model if one exists; such an algorithm is clearly a decision procedure for $\mathcal{SHOIQ}$ TBox satisfiability.

In practice, the algorithm works on a labelled graph, called a *completion graph*, that has a close correspondence to a model; this is because $\mathcal{SHOIQ}$ models may be infinite (although finitely representable), and because it is convenient not to explicate edges that may be implied by transitivity (of roles).

For ease of presentation we will, as usual, assume all concepts to be in *negation normal form* (NNF). A concept can be transformed into an equivalent one in NNF by pushing negation inwards, making use of de Morgan's laws and the duality between existential and universal restrictions, and between atmost and atleast number restrictions (Horrocks et al., 2000). For a concept C, we use $\dot{\neg}C$ to denote the NNF of $\neg C$, and we use $\mathsf{sub}(C)$ to denote the set of all subconcepts of C (including C). We will also abuse our notation by saying that a TBox $\mathcal{T}$ is in NNF if all the concepts occurring in $\mathcal{T}$ are in NNF. For a TBox $\mathcal{T}$ in NNF and a role hierarchy $\mathcal{R}$, we define the set of "relevant sub-concepts" $\mathsf{cl}(\mathcal{T},\mathcal{R})$ as follows:

$$\begin{aligned}
\mathsf{cl}(\mathcal{T},\mathcal{R}) &:= \bigcup_{C \sqsubseteq D \in \mathcal{T}} (\mathsf{cl}(C,\mathcal{R}) \cup \mathsf{cl}(D,\mathcal{R})) \quad \text{where} \\
\mathsf{cl}(E,\mathcal{R}) &:= \mathsf{sub}(E) \cup \{\dot{\neg}C \mid C \in \mathsf{sub}(E)\} \cup \\
&\quad \{\forall S.C \mid \ \forall R.C \in \mathsf{sub}(E) \text{ or } \dot{\neg}\forall R.C \in \mathsf{sub}(E), \\
&\qquad\qquad \text{and } S \text{ occurs in } \mathcal{T} \text{ or } \mathcal{R}\} \cup \\
&\quad \{(\leqslant m R.C) \mid \ (\leqslant n R.C) \in \mathsf{sub}(E) \text{ or} \\
&\qquad\qquad \dot{\neg}(\leqslant n R.C) \in \mathsf{sub}(E), \text{ and } m \leq n\}
\end{aligned}$$

When $\mathcal{R}$ is clear from the context, we use $\mathsf{cl}(\mathcal{T})$ instead of $\mathsf{cl}(\mathcal{T},\mathcal{R})$.

DEFINITION 5. *Let $\mathcal{R}$ be a role hierarchy and $\mathcal{T}$ a $\mathcal{SHOIQ}$ TBox in NNF. A* completion graph *for $\mathcal{T}$ with respect to $\mathcal{R}$ is a directed graph $\mathbf{G} = (V, E, \mathfrak{L}, \dot{\neq})$ where each node $x \in V$ is labelled with a set $\mathfrak{L}(x) \subseteq \mathsf{cl}(\mathcal{T}) \cup N_I$ and each edge $\langle x, y\rangle \in E$ is labelled with a set of roles $\mathfrak{L}(\langle x, y\rangle)$ containing (possibly inverse) roles occurring in $\mathcal{T}$ or $\mathcal{R}$. Additionally, we keep track of inequalities between nodes of the graph with a symmetric binary relation $\dot{\neq}$ between the nodes of* $\mathbf{G}$.

If $\langle x, y\rangle \in E$, then y is called a successor *of x and x is called a* predecessor *of y.* Ancestor *is the transitive closure of predecessor, and* descendant *is the transitive closure of successor. A node y is called an R-successor of a node x if, for some R' with $R' \sqsubseteq\!\!\!\!*\, R$, $R' \in \mathfrak{L}(\langle x, y\rangle)$; x is called an R-predecessor of y if y is an R-successor of x. A node y is called a* neighbour *(*R-neighbour*) of a node x if y is a successor (R-successor) of x or if x is a successor (R^--successor) of y.*

For a role S and a node x in $\mathbf{G}$, we define the set of x's S-neighbours with C in their label, $S^{\mathbf{G}}(x, C)$, as follows:

$$S^{\mathbf{G}}(x, C) := \{y \mid y \text{ is an } S\text{-neighbour of } x \text{ and } C \in \mathfrak{L}(y)\}.$$

$\mathbf{G}$ *is said to contain a* clash *if*

1. *for some $A \in N_C$ and node x of $\mathbf{G}$, $\{A, \neg A\} \subseteq \mathfrak{L}(x)$,*

2. *for some node x of $\mathbf{G}$, $(\leqslant n S.C) \in \mathfrak{L}(x)$ and there are $n + 1$ S-neighbours $y_0, \ldots, y_n$ of x with $C \in \mathfrak{L}(y_i)$ for each $0 \leq i \leq n$ and $y_i \dot{\neq} y_j$ for each $0 \leq i < j \leq n$,* or

3. *for some $o \in N_I$, there are two nodes x, y of $\mathbf{G}$ with $x \not\doteq y$ and $o \in \mathcal{L}(x) \cap \mathcal{L}(y)$.*

If $o_1, \ldots, o_\ell$ are all the nominals occurring in $\mathcal{T}$, then the tableau algorithm for checking satisfiability of a concept D starts with the completion graph $\mathbf{G} = (\{r_0, r_1 \ldots, r_\ell\}, \emptyset, \mathcal{L}, \emptyset)$ where $\mathcal{L}(r_0) = \{D\}$ and $\mathcal{L}(r_i) = \{o_i\}$ for $1 \leq i \leq \ell$. $\mathbf{G}$ is then expanded by repeatedly applying the expansion rules given in Figures 1 and 2, stopping if a clash occurs.

The expansion rules use some auxiliary terms and operations, which we will now define:

Nominal Nodes and Blockable Nodes We distinguish two types of nodes in $\mathbf{G}$: *nominal* nodes and *blockable* nodes. A node x is a nominal node if $\mathcal{L}(x)$ contains a nominal. A node that is not a nominal node is a blockable node. A nominal $o \in N_I$ is said to be *new in* $\mathbf{G}$ if no node in $\mathbf{G}$ has o in its label.

Blocking A node x is *label blocked* if it has ancestors x', y and y' such that

1. x is a successor of x' and y is a successor of y',
2. y, x and all nodes on the path from y to x are blockable,
3. $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x') = \mathcal{L}(y')$, *and*
4. $\mathcal{L}(\langle x', x\rangle) = \mathcal{L}(\langle y', y\rangle)$.

In this case, we say that *y blocks x*. A node is *blocked* if either it is label blocked or it is blockable and its predecessor is blocked.

Note that blocking helps to ensure termination by preventing the tableau algorithm from attempting to generate infinite branches. When the sequence of nodes and edges in a branch of the completion graph starts to repeat itself, a block is established, and no further generating rules (see below) are applied to the blocked nodes. In contrast, non-generating rules are still applicable to blocked nodes: in the presence of inverse roles, information from these nodes can be propagated back to predecessor nodes, possibly leading to clashes or the unblocking of previously blocked nodes (Horrocks and Sattler, 1999).

Generating and Shrinking Rules and Safe Neighbours The $\geqslant$-, $\exists$-, and $\mathsf{o}?$-rules are called *generating rules*, and the $\leqslant$-, $\leqslant_o$- and o-rules are called *shrinking rules*. An R-neighbour y of a node x is *safe* if (i) x is blockable or if (ii) x is a nominal node and y is not blocked.

Note that generating rules generate only safe nodes: the $\mathsf{o}?$-rule generates nominal nodes, which can't be blocked. If a blockable successor y of a node x is generated by the $\geqslant$- or $\exists$-rule, then x is the only predecessor of y. If x is a blockable node, then its successor y is safe, and if x is a nominal node, then it cannot block y, and y is again safe.

Pruning When a node y is *merged* into a node x, we "prune" the completion graph by removing y and, recursively, all blockable successors of y. More precisely, pruning a node y (written $\mathsf{Prune}(y)$) in $\mathbf{G} = (V, E, \mathcal{L}, \not\doteq)$ yields a graph that is obtained from $\mathbf{G}$ as follows:

1. for all successors z of y, remove $\langle y, z\rangle$ from E and, if z is blockable, $\mathsf{Prune}(z)$;
2. remove y from V.

Merging merging a node y into a node x (written $\mathsf{Merge}(y, x)$) in $\mathbf{G} = (V, E, \mathcal{L}, \not\doteq)$ yields a graph that is obtained from $\mathbf{G}$ as follows:

1. for all nodes z such that $\langle z, y\rangle \in E$
   a) if $\{\langle x, z\rangle, \langle z, x\rangle\} \cap E = \emptyset$, then add $\langle z, x\rangle$ to E and set $\mathcal{L}(\langle z, x\rangle) = \mathcal{L}(\langle z, y\rangle)$,
   b) if $\langle z, x\rangle \in E$, then set $\mathcal{L}(\langle z, x\rangle) = \mathcal{L}(\langle z, x\rangle) \cup \mathcal{L}(\langle z, y\rangle)$,
   c) if $\langle x, z\rangle \in E$, then set $\mathcal{L}(\langle x, z\rangle) = \mathcal{L}(\langle x, z\rangle) \cup \{\mathsf{Inv}(S) \mid S \in \mathcal{L}(\langle z, y\rangle)\}$, and
   d) remove $\langle z, y\rangle$ from E;
2. for all nominal nodes z such that $\langle y, z\rangle \in E$
   a) if $\{\langle x, z\rangle, \langle z, x\rangle\} \cap E = \emptyset$, then add $\langle x, z\rangle$ to E and set $\mathcal{L}(\langle x, z\rangle) = \mathcal{L}(\langle y, z\rangle)$,
   b) if $\langle x, z\rangle \in E$, then set $\mathcal{L}(\langle x, z\rangle) = \mathcal{L}(\langle x, z\rangle) \cup \mathcal{L}(\langle y, z\rangle)$,
   c) if $\langle z, x\rangle \in E$, then set $\mathcal{L}(\langle z, x\rangle) = \mathcal{L}(\langle z, x\rangle) \cup \{\mathsf{Inv}(S) \mid S \in \mathcal{L}(\langle y, z\rangle)\}$, and
   d) remove $\langle y, z\rangle$ from E;
3. set $\mathcal{L}(x) = \mathcal{L}(x) \cup \mathcal{L}(y)$;
4. add $x \not\doteq z$ for all z such that $y \not\doteq z$; and
5. $\mathsf{Prune}(y)$.

Rule applications are prioritised as follows (Horrocks and Sattler, 2005): the o-rule is applied with highest priority; the $\leqslant_o$- and the $\mathsf{o?}$-rule are applied next, and they are applied first to nominal nodes with lower levels; finally, all other rules are applied with a lower priority. The level of a nominal node is defined inductively as follows:

– each (nominal) node x with an $o_i \in \mathcal{L}(x)$, $1 \leq i \leq \ell$, is of level 0, and

$\sqcap$-rule: if 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, and
2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$

$\sqcup$-rule: if 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$ and
2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$

$\exists$-rule: if 1. $\exists S.C \in \mathcal{L}(x)$, x is not blocked, and
2. x has no safe S-neighbour y with $C \in \mathcal{L}(y)$,
then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{S\}$
and $\mathcal{L}(y) = \{C\}$

$\forall$-rule: if 1. $\forall S.C \in \mathcal{L}(x)$, and
2. there is an S-neighbour y of x with $C \notin \mathcal{L}(y)$
then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$

$\forall_+$-rule: if 1. $\forall S.C \in \mathcal{L}(x)$, and
2. there is some R with $\mathsf{Trans}(R)$ and $R \mathrel{\underline{\boxast}} S$,
3. there is an R-neighbour y of x with $\forall R.C \notin \mathcal{L}(y)$
then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{\forall R.C\}$

?-rule: if 1. $(\leqslant n S.C) \in \mathcal{L}(x)$, and
2. there is an S-neighbour y of x with $\{C, \dot{\neg}C\} \cap \mathcal{L}(y) = \emptyset$
then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{E\}$ for some $E \in \{C, \dot{\neg}C\}$

$\geqslant$-rule: if 1. $(\geqslant n S.C) \in \mathcal{L}(x)$, x is not blocked, and
2. there are not n safe S-neighbours $y_1, \ldots, y_n$ of x with
$C \in \mathcal{L}(y_i)$ and $y_i \dot{\neq} y_j$ for $1 \leq i < j \leq n$
then create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_i\rangle) = \{S\}$,
$\mathcal{L}(y_i) = \{C\}$, and $y_i \dot{\neq} y_j$ for $1 \leq i < j \leq n$

$\leqslant$-rule: if 1. $(\leqslant n S.C) \in \mathcal{L}(z)$, and
2. $\sharp S^{\mathbf{G}}(z, C) > n$ and there are two S-neighbours
x, y of z with $C \in \mathcal{L}(x) \cap \mathcal{L}(y)$, and not $x \dot{\neq} y$
then 1. if x is a nominal node, then $\mathsf{Merge}(y, x)$
2. else if y is a nominal node or an ancestor of x,
then $\mathsf{Merge}(x, y)$, else $\mathsf{Merge}(y, x)$

$\sqsubseteq$-rule: if 1. $C_1 \sqsubseteq C_2 \in \mathcal{T}$, and
2. $\{\dot{\neg}C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{\dot{\neg}C_1, C_2\}$

Figure 1. Basic tableau expansion rules for $\mathcal{SHOIQ}$

- a nominal node x is of level i if x is not of some level $j < i$ and x has a neighbour that is of level $i - 1$.

The combination of the $\leqslant_o$- and the o?-rules, and the priority with which they are applied, is necessary in order to ensure termination. Without the o?-rule, non-termination could result from repeated generation and merging of predecessors of a nominal node, and without the $\leqslant_o$-rule, non-termination could result from the repeated generation and pruning of a path leading to a nominal node.

Two kinds of rule will be of particular interest in the following discussion: *non-deterministic* rules, such as the $\sqcup$-rule and $\sqsubseteq$-rule, and *generating* rules, such as the $\exists$-rule and $\geqslant$-rule. In practice, non-deterministic rules are dealt with by using backtracking search to investigate the completion graphs resulting from each possible expansion (see Section 4.1). Applying such rules is, therefore, likely to be more "costly", as they either increase the size of the graph or increase the size of the search space. It is worth noting that, although the o?-rule adds a new source of nondeterminism, it will only be applicable in case nominals, number restrictions and inverse roles are used in a way that seems rather unnatural, and that has not been observed (so far) in realistic KBs (Horrocks and Sattler, 2005).

3. Preprocessing and simplifications

The first group of optimisations in FaCT++ is performed directly on the syntax of the input. These optimisations serve to preprocess and simplify the input into a form more amenable to later processing. As well as simplifications such as tautology elimination and obvious clash detection, preprocessing optimisations (like absorption, Section 3.2.5) can also lead to significant speedup of the subsequent reasoning process.

Note that satisfiability checking for expressive DLs requires, in the worst case, time that is at least exponential in the size of the input, whereas most preprocessing optimisations have polynomial, or even linear worst case complexity.

3.1. Lexical normalisation and simplification

Descriptions of DL tableau algorithms, such as the one given in Section 2, typically assume that the input is in negation normal form (NNF); this simplifies the (description of the) algorithm, but it means that the first (and most common) kind of clash, i.e., $\{C, \neg C\} \subseteq \mathcal{L}(x)$ for some node x in the completion graph, will only be detected when C is a named concept. For example, when testing the satisfiability of the concept $(A \sqcap B) \sqcap \neg(A \sqcap B)$,

```
o-rule:  if   for some $o \in N_I$ there are 2 nodes $x, y$ with $o \in \mathcal{L}(x) \cap \mathcal{L}(y)$
              and not $x \stackrel{.}{\neq} y$
         then Merge$(x, y)$

o?-rule: if 1. $(\leqslant nS.C) \in \mathcal{L}(x)$, $x$ is a nominal node, and there is a blockable
               $S$-neighbour $y$ of $x$ such that $C \in \mathcal{L}(y)$ and
               $x$ is a successor of $y$, and
            2. there is no $(\leqslant mS.C) \in \mathcal{L}(x)$ such that $m \leqslant n$,
               and there exist $m$ nominal $S$-neighbours $z_1, \ldots, z_m$ of $x$
               with $C \in \mathcal{L}(z_i)$ and $z_i \stackrel{.}{\neq} z_j$ for all $1 \leq i < j \leq m$.
         then 1. guess $m$ with $1 \leqslant m \leqslant n$ and set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{(\leqslant mS.C)\}$
              2. create $m$ new nodes $z_1, \ldots, z_m$ with $\mathcal{L}(\langle x, z_i \rangle) = \{S\}$,
                 $\mathcal{L}(z_i) = \{C, o_i\}$ with $o_i \in N_I$ new in $\mathbf{G}$,
                 and $z_i \stackrel{.}{\neq} z_j$ for $1 \leq i < j \leq m$.

$\leqslant_o$-rule: if 1. $(\leqslant mS.C) \in \mathcal{L}(x)$, $x$ is a nominal node, and there is a blockable
               $S$-neighbour $y$ of $x$ such that $C \in \mathcal{L}(y)$,
            2. there exist $m$ nominal $S$-neighbours $z_1, \ldots, z_m$ of $x$
               with $C \in \mathcal{L}(z_i)$ and $z_i \stackrel{.}{\neq} z_j$ for all $1 \leq i < j \leq m$, and
            3. there is a nominal $S$-neighbour $z$ of $x$ with $C \in \mathcal{L}(z)$, and not $y \stackrel{.}{\neq} z$
         then Merge$(y, z)$
```

Figure 2. Expansion rules dealing with nominals

the transformation into NNF would give $(A \sqcap B) \sqcap (\neg A \sqcup \neg B)$; in practice this means that, in spite of the "obvious" contradiction, backtracking search will be performed in order to determine that the concept is unsatisfiable (see Section 4.1).

For this reason, practical algorithms do not transform the input into NNF, but include a $\neg$-rule that performs a single (negation) normalisation step (e.g., applying the $\neg$-rule to $\neg(A \sqcap B) \in \mathcal{L}(x)$ would cause $\neg A \sqcup \neg B$ to be added to $\mathcal{L}(x)$), and the completion graph is said to contain a clash if it contains a node x with $\{C, \neg C\} \subseteq \mathcal{L}(x)$ for an arbitrary concept C. Moreover, in order to facilitate the detection of such clashes, the input is *normalised* and *simplified* so that logically equivalent concepts are more often syntactically equivalent. This is achieved by (recursively) applying a set of rewrite rules to concept expressions, and by ordering conjuncts w.r.t. some total ordering (Horrocks, 2003). For example, we re-write $\sqcup$- and $\exists$-concepts as negated $\sqcap$- and $\forall$-concepts, respectively; we merge conjunctions wherever possible; we order conjuncts; and we use logical equivalences and semantics preserving transformations in order to simplify concepts. These equivalences and transformations can be split into three groups: *constant elimination*, *syntactic equivalences* and *semantic transformations*.

The constant elimination group includes the following equivalences: $(C \sqcap \top) \equiv C$, $(C \sqcap \bot) \equiv \bot$, $\forall R.\top \equiv \top$.

The syntactic equivalence group includes the following equivalences: $(C \sqcap C) \equiv C$, $\neg\neg C \equiv C$, $C \sqcap \neg C \equiv \bot$, $\geq 1 R.C \equiv \exists R.C$

The semantic transformation group exploits semantic information already gathered during the preprocessing phase to simplify concept expressions. For example, if the TBox contains an axiom $A \sqsubseteq B$ for named concepts A and B, then the concept expression $A \sqcap B$ can be simplified to A. The same is true for arbitrary concept expressions.

In FaCT++, rewrite rules are separated into two classes: "cheap" ones and "expensive" ones. Cheap ones (i.e., $C \sqcap \bot \rightarrow \bot$) are easily applied and almost always give a positive effect. They are, therefore, applied to every concept expression appearing in the TBox.

Expensive rules require more effort to recognise when they are applicable (i.e., $A \sqcap B \rightarrow A$ when $A \sqsubseteq B$), and may not give a positive effect at all. Indeed, a well designed TBox is likely to include few such constructions, so applying such rule to the whole (possibly huge) TBox will be a waste of time. However, there are special cases in which it is crucial to have concept expression be as simple as possible. Such cases include axioms in $\mathcal{T}_g$, absorbed concept expressions and role ranges and domains (all of which will be explained below). In these cases, all simplification rules (including the expensive ones) are applied.

3.2. Dealing with axioms

If dealt with naively, TBox axioms can lead to a serious degradation in reasoning performance, as each such axiom would cause a disjunction to be added to every node of the completion graph, leading to potentially enormous amounts on nondeterministic expansion and backtracking search.

For example, given a TBox $\mathcal{T}$, if $\top \sqsubseteq A \in \mathcal{T}$, with $A = ((C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n))$, and testing the satisfiability of $\mathcal{T}$ leads to the construction of a completion graph containing k nodes, then there are 2^{kn} different ways to apply the $\sqcap$- and $\sqcup$-rules to the resulting k copies of A. This explosion in the size of the search space can easily lead to a catastrophic degradation in performance, even when optimisations such as backjumping (see Section 4.3) and caching (see Section 4.7) are employed (Horrocks, 1997).

Fortunately, optimisations known as *lazy unfolding* and *absorption* have proved to be very effective in reducing the size of the search space.

3.2.1. *Lazy Unfolding*

TBox axioms are often (restricted to be) of the form $A \sqsubseteq C$ or $A \equiv C$ for some concept name A (where $A \equiv C$ is an abbreviation for the pair of GCIs, $A \sqsubseteq C$ and $\neg A \sqsubseteq \neg C$). Such axioms are often called *definitional*, as they can be thought of as defining the meaning of A. A TBox

$$\mathcal{T} = \{A_1 \equiv C_1, \ldots, A_\ell \equiv C_\ell, A_{\ell+1} \sqsubseteq C_{\ell+1}, \ldots, A_{\ell+m} \sqsubseteq C_{\ell+m}\}$$

is said to be *unfoldable*, if it satisfies the following conditions.

– All axioms in $\mathcal{T}$ are definitional.

– Axioms in $\mathcal{T}$ are *unique*. That is, for each concept name A, $\mathcal{T}$ contains at most one axiom of the form $A \equiv C$ (i.e., $A_i \neq A_j$ for $1 \leqslant i < j \leqslant \ell$), and if it contains an axiom of the form $A \equiv C$, then it does not contain any axiom of the form $A \sqsubseteq C$. (Note that an arbitrary set of axioms $\{A \sqsubseteq C_1, \ldots, A \sqsubseteq C_n\}$ can be combined into a single axiom $A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$.) We will call a named concept A *non-primitive* if $\mathcal{T}$ contains an axiom $A \equiv C$, which is called a *non-primitive definition*. In the other case we will call A a *primitive concept*, and an axiom $A \sqsubseteq C$ a *primitive definition*.

– $\mathcal{T}$ is *acyclic*. That is, there is no axiom $A_i \equiv C_i \in \mathcal{T}$ such that A_i occurs in C_i. A concept name A *occurs* in a concept expression C if either A occurs syntactically in C, or there is a concept name A' such that A' occurs syntactically in C, and there is an axiom $A' \equiv C' \in \mathcal{T}$ such that A occurs in C'.

The idea of lazy unfolding was initially introduced in (Baader et al., 1994) for unfoldable TBoxes and refined for general TBoxes in (Horrocks, 1998). Instead of being dealt with using the $\sqsubseteq$-rule, such a set of axioms can be lazily *unfolded* during the tableau expansion. That is, for an axiom $A_1 \sqsubseteq C_1 \in \mathcal{T}$, if A_i is added to $\mathcal{L}(x)$ for some node x, then C_i is also added to $\mathcal{L}(x)$, and for an axiom $A_j \equiv C_j \in \mathcal{T}$, if A_j (resp. $\neg A_j$) is added to $\mathcal{L}(x)$ for some node x, then C_j (resp. $\neg C_j$) is also added to $\mathcal{L}(x)$.

It is obvious that an arbitrary TBox $\mathcal{T}$ can be divided into an unfoldable part $\mathcal{T}_u$ and a general part $\mathcal{T}_g$ such that $\mathcal{T}_u \cup \mathcal{T}_g = \mathcal{T}$ and $\mathcal{T}_u \cap \mathcal{T}_g = \emptyset$. The unfoldable part $\mathcal{T}_u$ can then be dealt with using lazy unfolding while the general part $\mathcal{T}_g$ is dealt with using the $\sqsubseteq$-rule. In fact it has been shown that the definition of an unfoldable TBox can be extended somewhat while still allowing the use of the above lazy unfolding technique. In particular, the concept expressions occurring on the left hand side of primitive- and non-primitive definitions can also be negated named concepts, and the acyclicity condition can be relaxed by distinguishing positive and negative occurrences of named concepts in a stratified theory (Horrocks and Tobies, 2000b; Lutz, 1999).

$\sqsubseteq_u$-rule: if 1. $C_1 \in \mathcal{L}(x), C_1 \sqsubseteq C_2 \in \mathcal{T}_u$, and
2. $\{C_2\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_2\}$

$\sqsubseteq_g$-rule: if 1. $C_1 \sqsubseteq C_2 \in \mathcal{T}_g$, and
2. $\{\dot{\neg} C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{\dot{\neg} C_1, C_2\}$

Figure 3. Tableau expansion rules for unfoldable and general axioms

Lazy unfolding can be viewed as a modification of the tableau expansion rules, replacing the $\sqsubseteq$-rule with two rules, one for unfoldable axioms and one for general axioms. These two rules, the $\sqsubseteq_u$-rule and the $\sqsubseteq_g$-rule are given in Figure 3.

In FaCT++, a form of lazy unfolding is also used to deal more efficiently with so called *range* and *domain* constraints. These often arise in TBoxes derived from ontologies, where it is common to state, e.g., that the role "drives" has domain "adult" and range "vehicle", where adult and vehicle are concepts, and where the intuitive meaning is that only adults can drive, and that only vehicles can be driven. This can easily be expressed as $\mathcal{SHOIQ}$ axioms of the form $\top \sqsubseteq \forall R^-.C$ (to state that the domain of R is C, e.g., $\top \sqsubseteq \forall\text{drives}^-.\text{adult}$), and $\top \sqsubseteq \forall R.C$ (to state that the range of R is C, e.g., $\top \sqsubseteq \forall\text{drives.vehicle}$).

Such axioms are not unfoldable, and will therefore be dealt with by the $\sqsubseteq$-rule. For an axiom $\top \sqsubseteq \forall R.C$, this would lead to $\neg\top \sqcup \forall R.C$ being added to the label of each node. This can clearly be simplified to $\forall R.C$, so there is no need for non-deterministic expansion. Even so, when there are very large numbers of range and domain axioms (which is the case in some TBoxes derived from ontologies, where it is common practice to specify the range and domain of *every* role), this may lead to a significant degradation of performance simply due to the large size of node labels (and using complex data structures for node labels is also problematical due to the saving and restoring that is needed during backtracking search). Extending the tableau algorithm to lazily unfold range and domain axioms not only deals with this problem, but also provides additional opportunities for applying the important *absorption* optimisation (see Section 3.2.5).

An arbitrary TBox $\mathcal{T}$ can now be divided into three parts: an unfoldable part $\mathcal{T}_u$, a range and domain part $\mathcal{T}_r$, and a general part $\mathcal{T}_g$, such that $\mathcal{T} = \mathcal{T}_u \cup \mathcal{T}_r \cup \mathcal{T}_g$, and $\mathcal{T}_u$, $\mathcal{T}_r$ and $\mathcal{T}_g$ are pairwise disjoint. The unfoldable part $\mathcal{T}_u$ is as before, the range and domain part $\mathcal{T}_r$ consists of all the axioms of the form $\top \sqsubseteq \forall R.C$ that would formerly have been in $\mathcal{T}_g$ (note that R can be an

$$\begin{array}{l}\mathbf{R}\text{-rule: if } 1.\ \top \sqsubseteq \forall S.C \in \mathcal{T}_r, \text{ and} \\ \qquad 2.\ \text{there is an } S\text{-neighbour } y \text{ of } x \text{ with } C \notin \mathcal{L}(y) \\ \quad \text{then } \mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}\end{array}$$

Figure 4. Tableau expansion rule for domain and range axioms

inverse role), and $\mathcal{T}_g$ consists of the remaining general axioms. A new tableau expansion rule, the $\mathbf{R}$-rule, is added in order to deal with $\mathcal{T}_r$; this rule is given in Figure 4.

3.2.2. *Taxonomic Encoding and Synonym Replacement*

If $\mathcal{T}_u$ contains a definition of the form $A \equiv C$, then replacing occurrences of C with A may allow further simplifications to be performed. For example, given the definition $A \equiv \forall R.C$, the concept $\neg A \sqcup \forall R.C$ can be rewritten as $\neg A \sqcup A$ and then simplified to give $\top$. The usefulness of this technique can be further enhanced by introducing new names for any unnamed concept expressions occurring in the TBox, a technique known as *taxonomic encoding* (Horrocks, 2003); in practice, this is often implemented by using pointers and structure sharing rather than by introducing new definitions.

If the TBox includes an axiom of the form $A \equiv B$, where both A and B are concept names, then A and B are obviously synonyms (two names for the same concept). This kind of axiom may introduce “fake” cycles and dependencies into the TBox, and interfere with the application of some optimisations. For example, if the TBox includes the axioms $B \equiv \forall R.C$ and $B \equiv A$, then one of these two axioms would have to be treated as a pair of inclusion axioms in $\mathcal{T}_g$, leading to additional (and unnecessary) nondeterminism in tableau expansion. Moreover, it may no longer be possible (in general) to simplify the concept $\neg A \sqcup \forall R.C$ to give $\top$.

In FaCT++, a technique called *synonym replacement* is used in order to mitigate such problems. If the TBox includes an axiom of the form $A \equiv B$, for A and B concept names, then A is called a *synonym* of B, B is called the *representative concept* of A, and the axiom $A \equiv B$ is called the *synonym definition* of A. During synonym replacement, each synonym definition is removed from the TBox, and all occurrences of the synonym are replaced with its representative concept. After this, other simplifications, such as those described in Section 3.1, can be applied. For example, if the TBox includes the axioms $B \equiv \forall R.C$ and $B \equiv A$, as above, then we can discard $B \equiv A$ and replace B with A, so that $B \equiv \forall R.C$ becomes $A \equiv \forall R.C$; the concept $\neg A \sqcup \forall R.C$ can then be simplified to $\top$.

Note that care must be taken to ensure that $\mathcal{T}_u$ is still unfoldable after an application of synonym replacement. For example, if $\mathcal{T}_u$ includes the axioms

$A \equiv C$ and $B \equiv D$, applying synonym replacement to the axiom $B \equiv A$ would leave two definitions of A in $\mathcal{T}_u$; one of these must moved to $\mathcal{T}_g$ (in the form of a pair of inclusion axioms).

3.2.3. *Told Cycle Elimination*

The idea of a told subsumer is widely used in DL reasoning, especially in classification algorithms (see Section 5 below). Informally, a named concept C has a *told subsumer* D if $C \sqsubseteq D$ is "obvious" from the syntactic structure of the TBox. This information is useful in classification, because it provides some initial information about subsumption relations (Horrocks, 2003).

More formally, a concept D *appears in the expression* C, if either $C = D$, or $C = C_1 \sqcap \ldots \sqcap C_n$ and D appears in the expression C_i for some $1 \leq i \leq n$. A named concept B is called an *immediate told subsumer* (ITS) of a named concept A, written $B \in ITS(A)$, iff:

– $A \sqsubseteq C \in \mathcal{T}$ or $A \equiv C \in \mathcal{T}$ and B appears in the expression C;

– $A \sqsubseteq C \in \mathcal{T}$ or $A \equiv C \in \mathcal{T}$, $\exists R.D$ appears in the expression C, $\top \sqsubseteq \forall \mathsf{Inv}(R).E \in \mathcal{T}_r$ and B appears in the expression E;

– $A \sqsubseteq C \in \mathcal{T}$ or $A \equiv C \in \mathcal{T}$, $\geq nR.D$ appears in the expression C, $\top \sqsubseteq \forall \mathsf{Inv}(R).E \in \mathcal{T}_r$ and B appears in the expression E.

Told subsumer is the transitive closure of ITS.

FaCT++ includes a simple but useful optimisation that tries to eliminate cyclical told subsumptions. A TBox $\mathcal{T}$ has a *told cycle* if, for some $A \in N_C$, A is a told subsumer of itself. The presence of told cycles in the TBox can lead to several problems, and in particular can cause problems for algorithms that exploit the told subsumer hierarchy (i.e., the concept hierarchy that is implied by told subsumer relations). Told cycles also make some GCI simplifications inapplicable. These cycles are, however, often quite easy to eliminate. We assume that most modern reasoners include such an optimisation, although we know of no reference to it in the literature.

Assume $A_1 \ldots A_n$ are named concepts, such that $A_{i+1} \in ITS(A_i)$ for all $1 \leq i < n$, and $A_1 = A_n$. In this case, all A_i are equivalent. We can chose A_1 as a representative concept, make $A_2, \ldots, A_{n-1}$ synonyms of A_1 by adding axioms $A_i \equiv A_1$ to $\mathcal{T}_u$, and run the synonym replacement algorithm on $\mathcal{T}$.

3.2.4. *Redundant subsumption elimination*

After applying told cycle elimination, some axioms might include redundant information due to synonym replacement. For example, if $A \sqsubseteq B \sqcap C \in \mathcal{T}$, and after told cycle elimination $B \equiv A$ was added to $\mathcal{T}_u$, then the axiom would be transformed into $A \sqsubseteq A \sqcap C$, where the A in $A \sqcap C$ is clearly redundant. *Redundant subsumption elimination* is a simple optimisation used in FaCT++ to remove this kind of redundancy.

A concept A *occurs at the top level* of a concept expression C if A syntactically occurs in C, but not in the scope of role quantifiers. During optimisation, for every axiom $A \sqsubseteq C$, any top-level occurrences of A in C are replaced with $\top$; the usual syntactic simplifications are then applied to the axiom.

Note that this optimisation is different from told cycle elimination (Section 3.2.3), although there are some axioms that can be simplified using either optimisation; the axiom $A \sqsubseteq A \sqcap \neg B$ would, for example, be simplified to $A \sqsubseteq \neg B$ by either optimisation. Told cycle elimination cannot, however, deal with axioms like $A \sqsubseteq \neg A \sqcup C$, and redundant subsumption elimination is not applicable if several axioms are involved in a cycle.

3.2.5. *Absorption*

Given the effectiveness of lazy unfolding in dealing with the unfoldable part of a TBox $\mathcal{T}_u$ and the range and domain axioms in $\mathcal{T}_r$, it makes sense to try to rewrite the axioms in $\mathcal{T}$ so that the size of $\mathcal{T}_g$ can be reduced.

Absorption is such a rewriting optimisation that tries to eliminate GCIs in $\mathcal{T}_g$ by absorbing them into concept definitions in $\mathcal{T}_u$ (*concept absorption*) or domain axioms in $\mathcal{T}_r$ (*role absorption*). This is usually done by rewriting general axioms into an equivalent form suitable for one of these absorptions: for concept absorption, the axiom should be of the form $CN \sqsubseteq D$, where CN is a named primitive concept and D is an arbitrary concept expression (Horrocks, 2003); for role absorption, the axiom should be of the form $\exists R.\top \sqsubseteq D$, where D is an arbitrary concept expression (Tsarkov and Horrocks, 2004). In addition, a special form of concept absorption, called nominal absorption, can be employed when an axiom has form $o_1 \sqcup \ldots \sqcup o_n \sqsubseteq D$, or $\exists R.o \sqsubseteq D$, where $o, o_1, \ldots, o_n$ are nominals and D is an arbitrary concept expression (Sirin et al., 2006).

Given a TBox $\mathcal{T}$, absorption proceeds as follows. First, $\mathcal{T}_u$, $\mathcal{T}_r$ and $\mathcal{T}_g$ are initialised such that $\mathcal{T}_r = \emptyset$, $\mathcal{T}_u \cup \mathcal{T}_g = \mathcal{T}$ and $\mathcal{T}_u$ is unfoldable. This can be trivially achieved by setting $\mathcal{T}_g = \mathcal{T}$ and $\mathcal{T}_u = \mathcal{T}_r = \emptyset$, but it is usually best to try to maximise the number of definitional axioms in $\mathcal{T}_u$, and in particular to maximise the number of definitional axioms of the form $A \equiv D$ in $\mathcal{T}_u$ (Horrocks and Tobies, 2000b). Due to the uniqueness and acyclicity restrictions, however, there may be no unique maximal $\mathcal{T}_u$.

Next, the axioms in $\mathcal{T}_g$ are normalised by rewriting them as semantically equivalent axioms of the form $\top \sqsubseteq C$, i.e., $A \sqsubseteq B$ is rewritten as $\top \sqsubseteq C$, where $C = (B \sqcup \neg A)$. The concepts occurring in such axioms can be simplified using the techniques described in Section 3.1, and trivial axioms can be dealt with as follows:

– $\top \sqsubseteq \top$ is trivially satisfiable and can be removed from the TBox.

– $\top \sqsubseteq \bot$ is trivially unsatisfiable and leads directly to TBox unsatisfiability.

Finally, a range of rewriting rules can be applied to axioms in $\mathcal{T}_g$ in order to transform them into a suitable form, and then add them to either $\mathcal{T}_u$ or $\mathcal{T}_r$. These rules are repeatedly applied until either $\mathcal{T}_g$ is empty or no further rules are applicable. Note that it is important to first eliminate told cycles, as described in Section 3.2.3, otherwise application of the rewriting rules may not terminate.

Axiom transformation rules:

– $\top \sqsubseteq B \sqcup C$, where B is a named concept with $B \equiv D \in \mathcal{T}_u$, can be rewritten as $\top \sqsubseteq D \sqcup C$.
– $\top \sqsubseteq \neg B \sqcup C$, where B is a named concept with $B \equiv D \in \mathcal{T}_u$, can be rewritten as $\top \sqsubseteq \neg D \sqcup C$.
– $\top \sqsubseteq (D_1 \sqcap D_2) \sqcup C$ can be rewritten as two axioms $\top \sqsubseteq D_1 \sqcup C$ and $\top \sqsubseteq D_2 \sqcup C$.

Concept absorption:

– $\top \sqsubseteq \neg A \sqcup C$, where A is a named concept with $A \sqsubseteq D \in \mathcal{T}_u$, can be absorbed into $\mathcal{T}_u$ by removing $\top \sqsubseteq \neg A \sqcup C$ from $\mathcal{T}_g$ and adding $A \sqsubseteq C$ to $\mathcal{T}_u$.

Role absorption:

– $\top \sqsubseteq (\forall R.C) \sqcup D$ can be absorbed into $\mathcal{T}_r$ by removing it from $\mathcal{T}_g$ and adding $\top \sqsubseteq \forall \mathsf{Inv}(R).((\forall R.C) \sqcup D)$ to $\mathcal{T}_r$.
– $\top \sqsubseteq (\leq nR.C) \sqcup D$ can be absorbed into $\mathcal{T}_r$ by removing it from $\mathcal{T}_g$ and adding $\top \sqsubseteq \forall \mathsf{Inv}(R).((\leq nR.C) \sqcup D)$ to $\mathcal{T}_r$.

Nominal absorption:

– $\top \sqsubseteq C$, where $C = \neg(o_1 \sqcup \ldots \sqcup o_n) \sqcup D$, can be absorbed into $\mathcal{T}_u$ by removing $\top \sqsubseteq C$ from $\mathcal{T}_g$ and adding $o_i \sqsubseteq D$ to $\mathcal{T}_u$ for all $1 \leq i \leq n$.
– $\top \sqsubseteq C$, where $C = \neg(\exists R.(o_1 \sqcup \ldots \sqcup o_n)) \sqcup D$, can be absorbed into $\mathcal{T}_u$ by removing $\top \sqsubseteq C$ from $\mathcal{T}_g$ and adding $o_i \sqsubseteq \forall \mathsf{Inv}(R).D$ to $\mathcal{T}_u$ for all $1 \leq i \leq n$.

Note that there may be many different ways to apply these rewriting rules, some of which may eventually lead to different absorptions. Defining what constitutes a "good" absorption is still an open problem, but an absorption that leaves $\mathcal{T}_g$ empty is invariably better in practice than one that does not (Horrocks and Tobies, 2000a; Tsarkov and Horrocks, 2004).

Finally, recent work has suggested a possible generalisation of absorption, called *binary absorption*, that could extend the range of situations in which absorption and lazy unfolding can be applied (Hudek and Weddell, 2006). The basic idea is to extend lazy unfolding so as to deal with axioms of the form $A \sqcap B \sqsubseteq C$, and to extend absorption so as to be able to rewrite axioms into this form. It remains to be seen, however, if this technique will be useful in practice.

4. Optimisations in Core Satisfiability Testing

The core of FaCT++ is its TBox satisfiability algorithm, which implements a highly optimised version of the tableau algorithm described in Section 2.2. Before introducing the optimisations used in FaCT++, we briefly show how such an algorithm is implemented in practice.

4.1. A "Standard" Satisfiability Algorithm

The presentation of the algorithm in Section 2.2 is designed to facilitate correctness proofs rather than implementation; it does not, for example, say anything about the order in which rules of equal priority are applied, or describe how non-deterministic rules would be dealt with in practice. We will, therefore, describe a "standard" implementation of a satisfiability testing algorithm which is based on the theoretical description from Section 2.2.

The first divergence from theory to practice is that concept expressions are *not* usually in NNF. Instead, as we described in Section 3.1, practical algorithms use a simplified normal form together with an additional $\neg$-rule, which allows algorithms to find clashes faster.

Secondly, practical algorithms also usually keep track of changes in the completion graph (e.g., which concepts were added to node labels, or the creation of new edges in a completion graph), and maintain some ordering of the concepts that have to be further expanded. As several expansion rules might be applicable at the same time, some order of rule application must be chosen in a deterministic implementation.

Thirdly, whenever a non-deterministic rule is applied, a deterministic algorithm must create a *branching point* in which the algorithm saves the state of the reasoning process (including, e.g., the completion graph). The algorithm then tries (in some order) possible applications of the rule, restoring the state after each try, until one leads to a complete and clash free completion graph or all have been shown to lead to a clash. In the former case, the algorithm terminates and returns "satisfiable", in the later case, it *backtracks* to the previous branching point if there is one, or terminates and returns "unsatisfiable" otherwise.

It is easy to see that, in moving from theory to practice, our standard implementation introduces two orderings: the order in which to deal with applicable expansion rules, and the order in which to investigate different possible expansions of non-deterministic rules. The chosen orderings can have an enormous effect on the time and space used by the algorithm, and many optimisations are targeted towards selecting "good" orderings for each of these two cases.

4.2. Ordering of Expansion Rules

Most systems use a (modified) *top-down* approach for ordering expansion rules. In this approach generating rules (like the $\geq$-rule and the $\exists$-rule) are applied after all other rules. This approach is largely carried over from when implemented DLs were weaker, and this top-down approach was both easy to implement and generally effective. For some subsets of $\mathcal{SHOIQ}$, like $\mathcal{SHF}$ (a DL with simplified number restrictions and without inverse roles and nominals), such a strategy can be used to create algorithms that only use polynomial storage (by employing the so-called *trace technique* (Schmidt-Schauß and Smolka, 1991)). In this technique the algorithm retains only the current branch of the completion tree (for such logics the completion graph is tree-shaped), fully expands it, and then discards the fully expanded branch in order to reuse the space when investigating other branches.

Note that this trace technique cannot be directly used in the presence of inverse roles, in particular because subtrees may interact via a common ancestor. However, most practical algorithms still use a modified version of the trace technique.

The FaCT++ system, on the other hand, was designed with the intention of implementing DLs that include inverse roles and other constructs that interfere with trace techniques, as well as investigating new optimisation techniques, including new ordering heuristics. Instead of a top-down approach, FaCT++ uses a *ToDo list* to control the application of the expansion rules. The basic idea behind this approach is that rules may become applicable whenever a concept is added to a node label. When this happens, a note of the node/concept pair is added to the ToDo list. The ToDo list sorts all these entries according to some order, and gives access to the "first" element in the list. A tableau algorithm based on a ToDo list proceeds by taking an entry from the ToDo list and processing it according to the expansion rule(s) relevant to the entry (if any). During the expansion process, new concepts may be added to node labels, and hence entries may be added to the ToDo list. The process continues until either a clash occurs or the ToDo list becomes empty.

In FaCT++ the ToDo list is implemented as a priority queue. It is possible to set a priority for each rule type, and whenever an entry is added to the queue, it is inserted after the already existing entries with the same or higher priorities. This means that if the $\exists$-rule (the generating rule that expands existential restrictions) has a low priority (say 0), and all other rules have the same priority (say 1), then the expansion will be (modulo inverse roles) top-down and breadth first.

The ToDo list approach has a number of advantages when compared to the top-down approach. Firstly, it is applicable to a much wider range of logics, including the expressive logics implemented in modern systems, because it makes no assumptions about the structure of the graph (in particular, whether it is tree-shaped or not) or the order in which the graph will be constructed. Secondly, the ToDo list approach allows for the use of more powerful heuristics that try to improve typical-case performance by varying the global order in which different syntactic structures are decomposed; in a top-down construction, such heuristics can only operate on a local region of the graph—typically a single node.

Empirical analysis (Tsarkov and Horrocks, 2005b) shows that the following ordering would be optimal for the majority of realistic ontologies: the $\sqcup$-rule has the lowest priority, the generating rules have the second-lowest priority, all other rules except for the $\leqslant_o$-rule, the o-rule and the $\mathsf{o?}$-rule have the second-highest priority, and the $\leqslant_o$-rule, the o-rule and the $\mathsf{o?}$-rule have the highest priority. (Recall that giving the $\leqslant_o$-rule, the o-rule and the $\mathsf{o?}$-rule the highest priority is needed to ensure that the algorithm always terminates.)

4.3. Dependency-directed backtracking (Backjumping)

Consider, for example, the concept C in the form

$$(C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists R.(A \sqcap B) \sqcap \forall R.\neg A.$$

In a classic top-down architecture, the disjunctions in this concept would be expanded before the existential. A tableau algorithm would thus first expand n disjunctions and then find a clash due to the $\exists$-rule.[3] In the case of normal (unoptimised) backtracking, 2^n choices of different disjunction expansions would be tried before the concept would be determined to be unsatisfiable.

To avoid this sort of exponential behaviour when checking the satisfiability of C and similar concepts, a more sophisticated solution is required, and can be found by adapting a form of dependency directed backtracking called *backjumping* (Horrocks, 1998), which has also been used, e.g., in solving constraint satisfiability problems (Baker, 1995) and (in a slightly different form) in the HARP theorem prover (Oppacher and Suen, 1988).

[3] If the ToDo list algorithm is used with the priority of the $\exists$-rule higher than that of the $\sqcup$-rule, then the clash would be found after expansions of the $\exists$- and $\forall$-rules. The exponential behaviour would return, however, if the existential is "hidden" inside a disjunction, e.g., $(\exists R.(A \sqcap B) \sqcup \exists R.(A \sqcap B'))$.

Intuitively, backjumping works by labelling each concept C in the label of a node x with a dependency set $Dep_C(x)$ indicating the branching points (i.e., applications of a non-deterministic rule) on which it depends. In case the completion graph contains some node x with $\{C, \neg C\} \in \mathcal{L}(x)$, we use $Dep_C(x)$ and $Dep_{\neg C}(x)$ to identify the most recent branching point b on which either C or $\neg C$ depends. The algorithm can then *jump* back to b over intervening branching points *without* exploring any alternative branches (non-deterministic choices), and make a different non-deterministic choice which might not lead to the same clash condition being encountered. In case no such b exists, the clash did not depend on any non-deterministic choice, and the algorithm stops, reporting the unsatisfiability of the TBox.

A more detailed explanation of a backjumping in tableau algorithms can be found in (Horrocks et al., 2006).

4.4. Optimisations of Disjunction Processing

4.4.1. *Boolean constraint propagation (BCP)*

As well as the standard tableau expansion rules described in Section 2, additional inference rules can be applied to concepts occurring in a node label, usually with the objective of simplifying them and reducing the number of $\sqcup$-rule applications. The most commonly used simplification, often called *Boolean Constraint Propagation* (BCP) (Freeman, 1995), is derived from SAT solvers, where it is usually used in conjunction with the Davis-Putnam procedure. The basic idea is to identify a disjunction $C_1 \sqcup \ldots \sqcup C_n \in \mathcal{L}(x)$ such that the negations of all but one of the C_j are already elements of $\mathcal{L}(x)$; when this is the case, the disjunction can be deterministically expanded by adding the relevant C_j to $\mathcal{L}(x)$. This amounts to applying the following inference rule

$$\frac{\neg C_1, \ldots, \neg C_n, C_1 \sqcup \ldots \sqcup C_n \sqcup C}{C}$$

to the concept in a node label.

Note that, as with the more sophisticated search techniques described below, careful consideration needs to be given to the dependencies of concepts added by such inference rules if they are to be used together with backjumping. In general, if a new inference rule which adds C to the label of node x is introduced, then the dependency set $Dep_C(x)$ will be the union of the dependencies of *all* the concepts and completion graph edges that are mentioned in the inference rule. In the case of the BCP inference rule, for example, this would be:

$$Dep_C(x) = Dep_{C_1 \sqcup \ldots \sqcup C_n \sqcup C}(x) \cup \bigcup_{i=1}^{n} Dep_{C_i}(x).$$

4.4.2. *Semantic Branching*

Consider the expansion of the following concept (for simplicity, assume that disjunctions will be expanded before existentials, and C would be the first choice in all applications of the $\sqcup$-rule):

$$(C \sqcup D_1) \sqcap \ldots \sqcap (C \sqcup D_n) \sqcap \exists R.\forall R^{-}.\neg C.$$

At the first branching point, C would be added to the label of the node. All the other disjunctions become moot (no expansion is needed, because one of the disjuncts, namely C, is already present in the label of the node). Further expansion leads to a clash, because $\neg C$ would be added to the node label after applications of $\exists$- and $\forall$-rules. After backtracking, D_1 would be chosen from the first disjunction. But then C would be chosen from the second disjunction, and the story would continue. Thus $3n$ expansion rules would be applied in order to generate a clash-free completion graph.

Semantic Branching (Giunchiglia and Sebastiani, 1996) allows a tableau algorithm to avoid such a situation. The idea is to try to avoid repeating "failed" choices when expanding disjunctions. Formally, it (implicitly) transforms a disjunction of the form $(C_1 \sqcup C_2)$ into the semantically equivalent form $(C_1 \sqcup (\neg C_1 \sqcap C_2))$. After expanding the disjunction by adding C_1 to $\mathcal{L}(x)$ and obtaining a clash, choosing the other disjunct and applying the $\sqcap$-rule would cause $\neg C_1$ and C_2 to be added to $\mathcal{L}(x)$. The former concept can be used to trigger BCP rule applications, thus preventing subsequent attempts to add the concept C_1 to the label of x (each of which would inevitably lead to a clash). Note that this is very similar to the Davis-Putnam-Logemann-Loveland procedure (DPLL), commonly used to solve propositional satisfiability problems (Freeman, 1995; Davis and Putnam, 1960; Davis et al., 1962).

In the example given at the beginning of this section, after the clash caused by adding C, semantic branching would cause $\neg C$ and D_1 to be added to the node label, and all other disjunctions would then be expanded deterministically using BCP. In this case, only $n + 4$ expansion rules would be applied in order to generate a clash-free completion graph. Note that this approach may, however, also lead to an additional overhead, because further (possibly non-deterministic) expansion of the concepts $\neg C_i$ added to $\mathcal{L}(x)$ may be required.

4.4.3. *Heuristics for Choosing Expansion Ordering*

It is well known that different orders of expanding non-deterministic rules can result in huge (up to several orders of magnitude) differences in reasoning performance (Tsarkov and Horrocks, 2005b). Heuristics can be used to choose a "good" order in which to try the different possible expansions of such rules. In practise, this usually means using heuristics to select the way in which the $\sqcup$-rule is applied to the disjunctions in a node label; a heuristic function is used to compute the relative “goodness” of candidate concept.

In DPLL, the well known MOMS heuristic (Freeman, 1995) is often used to select the propositional variable on which to branch; by choosing a variable that occurs frequently in small disjunctions (i.e., the variable that has the Maximum number of Occurrences in disjunctions of Minimum Size), it tries to maximise the effect of BCP and so minimise the number of non-deterministic choices needed in order to find a solution (or determine that there is no solution). This technique can easily be adapted to tableau algorithms: the MOMS value for a candidate concept C can be computed by simply counting the number of times C or its negation occur in minimally sized disjunctions (Horrocks, 2003). There is little evidence, however, that this heuristic is effective with concepts, and even some evidence to suggest that interference with the backjumping optimisation makes it counter-productive (Horrocks, 2003).

An alternative heuristic, whose design was prompted by this observation, tries to maximise the effect of backjumping by preferentially selecting concepts which depend on earlier branching points (Horrocks, 2003; Hladik, 2002). This heuristic has the added advantage that it can also be used to select the order in which successor nodes are expanded.

Most of these heuristics were, however, developed for the top-down reasoning approach in order to deal with its limitations. For the ToDo list approach, where non-deterministic expansion rules may have lower priority than generating ones, new classes of heuristics may be used. In FaCT++ we have concentrated on heuristics that take into account the syntactic characteristics of disjuncts (Tsarkov and Horrocks, 2005b).

FaCT++ allows several possible expansion-ordering heuristics to be used to choose the order in which to explore the different expansion choices offered by the non-deterministic $\sqcup$-rule. This ordering can be on the basis of the size, maximum quantifier depth, or frequency of usage of each of the concepts in the disjunction, and the order can be either ascending (smallest size, minimum depth and lowest frequency first) or descending. In order to avoid the cost of repeatedly computing such values, all the relevant statistics for each concept can be gathered as the knowledge base is loaded.

Another approach, proposed by Sirin *et al* (Sirin et al., 2005a), is to use a learning-based technique to select disjuncts. Initially, all disjuncts are given the same “penalty” value. When expanding disjunctions, disjuncts are ordered according to their penalties, so that those with the lowest penalties are tried first. If a clash occurs, disjuncts that are identified as being a cause of the clash have their penalty values increased, resulting in their being chosen with lower priority if the same disjunction is expanded again. This approach should be particularly effective for disjunctions on nominal nodes, because they are often expanded many times in a similar manner, and thus a poor statically-chosen ordering on them can lead to a significant slowdown.

4.5. Save/restore optimisations

As we have seen, in order to deal with non-deterministic rules, an implementation of a tableau algorithm must save its internal state before trying one of the expansion choices, and restore it when backtracking (after a clash has been detected). Amongst other things, the state of the completion graph (including node labels, edge labels, etc.) must be saved.

A naive approach would be to save everything (including the whole completion graph) before each such expansion choice, and restore everything on each backtrack. In realistic applications, however, the completion graph may contain hundreds or even thousands of nodes, and only a few of them might be changed as a result of any given expansion choice. This is true even when using the ToDo List approach: a sequence of rule applications, which may include non-deterministic rules, often changes only a small part of the completion graph. A naive approach to saving and restoring is, therefore, often highly inefficient. In order to address this problem, FaCT++ uses lazy approach to state saving.

4.5.1. *Lazy saving*

The goal of the *lazy saving* optimisation is to minimise the saving of completion graph nodes: instead of saving the whole completion graph at each branching point, lazy saving implements a "save-on-demand" approach.

In this approach, branching points are numbered according to the order in which they are created, and each node of the completion graph contains an additional field that records its *level*, i.e., the number of the most recently created branching point when the node was last saved. Whenever a change is going to be made to a node (including, e.g., additions to the node label, adding new incoming or outgoing edges, merging with another node or changes in blocking status), its level is checked: if it is lower than the most recent branching point, then the node's state is saved, and its level is set to the most recent branching point, before the change is made.

In addition to the obvious gains in terms of state saving, there are additional savings when restoring the completion graph: when backtracking from level n to level $n-1$, only those nodes whose level is equal to n need to be restored; when combined with the backjumping optimisation, backtracking from level n directly back to level m still only requires the restoration of those nodes whose level is greater than m.

In large completion graphs, it is usually the case that only a small number of nodes are changed at a given branching level. This is especially true in cases where non-deterministic rules are given the lowest priority. Avoiding unnecessary saves for large numbers of nodes saves processing time and memory space, but does introduce some additional overhead due to the need to check a node's level prior to any update. In theory (e.g., for TBoxes with a small number of branching concepts) this check may consume more time than is gained as a result of the optimisation, but in practice this is unlikely to be a serious problem as the check is very cheap (just an integer comparison).

4.5.2. *Lazy restoring*

As for lazy saving, lazy restoring implements a "restore-on-demand" approach. Instead of restoring when backtracking, it is possible to check before accessing nodes (i.e., before either examining or changing them) whether it is necessary to restore the state of the node. If the node appears to be "out of date", then it is necessary to restore it before accessing it.

This technique is, however, less useful than lazy saving for the following reasons:

– Lazy restoring is a check-on-access instead of a check-on-write technique. As there are many more reads than writes to a completion graph ($\forall$-rule application, blocking checks etc.), this checking will be much more costly.

– In lazy saving, after backtracking from level n to level m, new branching points can "reuse" levels greater than m. This is not possible when using lazy restoring, because there may still be nodes with levels greater than m that need to be restored before being accessed. In some cases, it might be necessary to keep and (probably) maintain the whole tree of branching decisions in order to determine when nodes need to be restored before being accessed.

– The number of restores is usually just a small fraction of the number of saves. In our experiments on realistic TBoxes, restores were between 0.5 and 5% as frequent as saves. This indicates that lazy restoring would provide at best only 5% of the benefit of lazy saving.

FaCT++ does not, therefore, use lazy restoring.

4.6. COMBINED GENERATING RULES

After creating or changing an edge label $\mathcal{L}(\langle x, y\rangle)$, the $\forall$- and/or $\forall_+$-rules may become applicable to any or all of the universal restriction concepts in $\mathcal{L}(x)$ or $\mathcal{L}(y)$. It is usually better to immediately check for and expand any relevant concepts, rather than to defer this work (for example, by putting entries in the ToDo list).

Assume, for example, that a concept $\exists R.C \in \mathcal{L}(x)$ was expanded, leading to the creation of a node y with $C \in \mathcal{L}(y)$ and $R \in \mathcal{L}(\langle x, y\rangle)$. Usually, $\exists$-rules are expanded later than $\forall$-rules (applying generating rules with a low priority is a commonly used heuristic), and in this case universal restrictions in $\mathcal{L}(x)$ will only be applicable to the new R-neighbour y of x. To minimise the amount of work that needs to be (re-)done, it is possible to immediately check for the applicability of the $\forall$- and/or $\forall_+$-rules with respect to the new or modified edge label; moreover, it is only necessary to apply the expansion rules to concepts of the form $\forall S.D \in \mathcal{L}(x)$ or $\forall \mathsf{Inv}(S).D \in \mathcal{L}(y)$ such that $R \,\underline{\circledast}\, S$.

Combined generating and universal restriction rules of this kind are probably implemented in most tableau-based reasoners.

4.7. Caching SAT status

Suppose that no inverse roles and no nominals appear in the TBox, and that a top-down technique is used, i.e., applications of the expansion rules are ordered by applying generating rules with the lowest priority. In this case, all information from predecessors is added to a node label before it is processed. This means that when a given node has been fully expanded (i.e., the expansion rules have been exhaustively applied to it), a successor node y with $\mathcal{L}(y) = \{C_1, \ldots, C_n\}$ can be treated as an independent problem, equivalent to testing the satisfiability of $C_1 \sqcap \ldots \sqcap C_n$. Please note that this is the basis of the trace technique, discussed in Section 4.2.

A completion graph may contain many such nodes, and the labels of nodes tend to be quite similar. For some concepts, this may result in the same sub-problem being solved again and again. In order to avoid this, it is possible to cache and re-use the results of such sub-problems. The usual technique is to use a hash table to store the satisfiability status of node labels (i.e., sets of concepts treated as a conjunction). Before applying any expansion rules to a new node x, the cache is interrogated to determine if the satisfiability status of $\mathcal{L}(x)$ is already known. If it is known, then the result can be used without further expansion, i.e., $\mathcal{L}(x)$ can be treated as though it were either $\{\bot\}$ (for unsatisfiable) or $\{\top\}$ (for satisfiable). If the satisfiability status of $\mathcal{L}(x)$ is not known, then $\mathcal{L}(x)$ is added to the cache; if we subsequently prove that the subproblem represented by $\mathcal{L}(x)$ is satisfiable, then its cache status set to satisfiable, and to unsatisfiable otherwise.

Note that when the tableau algorithm uses blocking in order to ensure termination (see Section 2.2), care must be taken when determining the satisfiability status of $\mathcal{L}(x)$. For example, if x has a successor y, y has a successor z, and x blocks z, then the satisfiability status of $\mathcal{L}(y)$ depends on $\mathcal{L}(x)$ also being satisfiable (because the block represents a cyclical or periodically repeating model in which z is “replaced by” x) (Möller, 2001).

Since the satisfiability of a set of concepts L implies the satisfiability of each subset of L, and the unsatisfiability of a set of concepts L implies the unsatisfiability of each superset of L, this basic idea can be extended to check for satisfiable supersets of $\mathcal{L}(x)$ and unsatisfiable subsets of $\mathcal{L}(x)$. However, this requires a considerably more sophisticated data structure if cache operations are to be efficient (Hoffmann and Koehler, 1999; Giunchiglia and Tacchella, 2000).

Apart from the problem of the storage required for the cache, another more subtle disadvantage of caching is that in the case where the cache returns "unsatisfiable" for $\mathcal{L}(x)$ there is no information about the cause of the unsatisfiability that can be used to derive the dependency information required for backjumping. Backjumping can still be performed by combining the dependency sets of all of the concepts in $\mathcal{L}(x)$, but this is likely to overestimate the set of branching points on which the unsatisfiability depends.

Another problem with caching is that in the presence of inverse roles or nominals, it cannot be directly used. In particular, nominals can be referred to from different nodes of a completion graph, so new concepts may be propagated to the label of an already cached node. In the case of inverse roles, information may be propagated from the node label to its parent, so just checking the SAT status of the child is not enough. Recent publications (Ding and Haarslev, 2005; Ding and Haarslev, 2006) suggest how caching can still be used in the presence of inverse roles, although so far only for a weaker logic than $\mathcal{SHOIQ}$ ($\mathcal{SHI}$).

4.8. Optimised Blocking

In the tableau decision procedure for $\mathcal{SHOIQ}$, blocking is used to ensure the termination of the algorithm, but its definition (in Section 2.2) serves theoretical purposes such as facilitating proofs more than practical ones such as efficient reasoning. For example, using the given definition of blocking, the root node cannot block any other node. This may lead to later recognition of repeating parts of completion graphs, the construction of larger graphs, and a decrease in the performance of reasoning.

This conservative approach to blocking is designed to ensure that a block in the completion graph always corresponds to an infinitely repeating pattern in the model represented by the graph. Such a model can be defined by infinitely "unravelling" the cycle that would result if the blocked node were identified with the blocking node. The blocking condition ensures that interactions between concepts that appear in the label of the blocking node and the predecessor of the blocked node are just those that are required, i.e., neither undesirable nor insufficient. FaCT++ employs a more fine-grained approach, as proposed by Horrocks and Sattler (Horrocks and Sattler, 2002), that allows blocks to be identified sooner, and significantly increases reasoning performance. This happens because earlier blocking leads to smaller completion graphs, which in turn leads to fewer applications of expansion rules, some of which may have been non-deterministic ones.

The idea of this optimised approach is to explicitly check all possible interactions between concepts in the blocking node and the predecessor of the blocked node. This makes it possible to weaken the blocking condition, and in particular to identify situations where a so-called "cyclic block" (c-block) can be used. Such a block corresponds to the case where the blocked and blocking nodes are mapped to the same element in the model represented by the completion graph, and this model thus contains a cycle. In the case of $\mathcal{SHIQ}$ and $\mathcal{SHOIQ}$, atmost restrictions in the label of the blocking node may be applicable to inverse roles, and this may rule out cyclical models. The stricter blocking condition described in the $\mathcal{SHIQ}$ and $\mathcal{SHOIQ}$ algorithms ensures that a block can always be "unravelled" to produce an infinite model where such interactions do not present any problem; this kind of block is called an "acyclic block" (a-block). The optimised approach does not prevent the identification of a-blocks, but allows the possible earlier identification of c-blocks.

Even with respect to a-blocks, the original blocking condition is unnecessarily conservative: it calls for two pairs of node labels and the intervening edge labels to be identical. In the optimised approach, this condition is weakened by only comparing relevant parts of the node and edge labels.

In the formal definition of optimised blocking, as presented in (Horrocks and Sattler, 2002), a node is label blocked if it is either *a-blocked* or *c-blocked*. A node w is *a-blocked* if none of its blockable ancestors are blocked, it is not c-blocked, and it has ancestors v and w' such that w is a successor of v, and

B1 $\mathcal{L}(w) \subseteq \mathcal{L}(w')$,

B2 if w is an $\mathsf{Inv}(S)$-successor of v and $\forall S.C \in \mathcal{L}(w')$ then

1. $C \in \mathcal{L}(v)$, and
2. if there is some R with $\mathsf{Trans}(R)$ and $R \sqsubseteq\!\!\!\!\ast\; S$ such that w is an $\mathsf{Inv}(R)$-successor of v, then $\forall R.C \in \mathcal{L}(v)$,

B3 if $(\leqslant nT.E) \in \mathcal{L}(w')$, then

1. w is not an $\mathsf{Inv}(T)$-successor of v or
2. w is an $\mathsf{Inv}(T)$-successor of v and $\dot{\neg}E \in \mathcal{L}(v)$ or
3. w is an $\mathsf{Inv}(T)$-successor of v, $E \in \mathcal{L}(v)$ and w' has at most $n-1$ S-successors z with $E \in \mathcal{L}(z)$, and

B4 if $(\geqslant mU.F) \in \mathcal{L}(w')$ (resp. $\exists U.F \in \mathcal{L}(w')$), then $\dot{\neg}F \in \mathcal{L}(w)$.

1. w' has at least m (resp. at least one) U-successors z with $F \in \mathcal{L}(z)$ or
2. w is an $\mathsf{Inv}(U)$-successor of v and $F \in \mathcal{L}(v)$.

A node w is *c-blocked* if none of its blockable ancestors are blocked, it has ancestors v and w' such that w is a successor of v, it satisfies **B1** and **B2** above, and

B5 if $(\leqslant nT.E) \in \mathcal{L}(w')$, then w is not an $\mathsf{Inv}(T)$-successor of v or $\dot{\neg}E \in \mathcal{L}(v)$, and

B6 if w is a U-successor of v and $(\geqslant mU.F) \in \mathcal{L}(v)$, then $\dot{\neg}F \in \mathcal{L}(w)$.

As proved in (Horrocks and Sattler, 2002), the correctness of the tableau algorithm is preserved when optimised blocking is used in place of standard blocking.

5. Optimisations in Classification

Classification is the process of establishing the partial order $\prec$ on the set of named concepts in a TBox with respect to the subsumption relation, i.e. $C \preceq D \iff C \sqsubseteq D$. This order is often referred to as the *taxonomy*, or *hierarchy*, of concepts. Classification is an important operation for reasoners because its results can be used to cheaply answer subsequent subsumption queries. It is also important in ontology engineering applications, where ontology developers often want to compare the computed taxonomy with their own intuition.

Traditionally, the classification partial order is built iteratively. The order is initialised with the trivial relation $\bot \prec \top$ (unless the TBox is unsatisfiable, in which case all concepts are equivalent, and the whole taxonomy collapses into a single node), and on every iteration one new concept name C is added. For each concept name C to be added to the taxonomy, the set of *parents* (i.e., already classified immediate subsumers) and the set of *children* (i.e., already classified immediate subsumees) is determined. These two sets uniquely identify the place of C in the current taxonomy. Note that if, for some concept D, it is determined that D is both a parent and a child of C, then $C \equiv D$, and C and D share the same place in the taxonomy.

The set of parents (resp. children) is defined by a procedure called *top-down* (resp. *bottom-up*) search. These two procedures are very similar, so we describe only one of them in detail.

The top-down search phase for concept C involves breadth-first search for the subsumers of C starting from $\top$. Once it is determined that $C \sqsubseteq D$ the algorithm searches for subsumers among the children of D. Concepts that are subsumers of C, but where none of their children are subsumers of C, become parents of C. This algorithm was first described by Baader *et al* (Baader et al., 1994), along with other optimisations that significantly improve the performance of classification. In the following section we are going to touch on some of these, along with some newer techniques.

5.1. Taxonomy Creation Order

The number of subsumption tests required for classification can be affected by the order in which concepts are added to the taxonomy. A well-known optimisation is to add concepts in *definition order*.

The concept named C *directly uses* the concept named D if D occurs syntactically in the definition of C. The relation *uses* is the transitive closure of directly uses. If C uses D then D comes before C in definition order. If no named concept uses itself then a TBox can be classified in definition order, i.e., a named concept is not classified until all the named concepts it uses are classified. Baader *et al* (Baader et al., 1994) showed that, for a purely definitional TBox (i.e. $\mathcal{T}_g = \emptyset$), when classifying in definition order, the bottom-up search phase can be omitted for primitive concepts (as the only child for such concepts would be $\bot$).

If $\mathcal{T}_g \neq \emptyset$, however, the bottom-up search phase cannot be omitted for primitive concepts. Consider the TBox $\mathcal{T} = \mathcal{T}_u \cup \mathcal{T}_g$ with

$$\mathcal{T}_u = \{C_1 \sqsubseteq D, C_2 \sqsubseteq D\}, \mathcal{T}_g = \{\top \sqsubseteq C_1 \sqcap C_2\}.$$

In this TBox concept D should be classified as equal to $\top$, i.e., have $\top$ both as a parent and child. If the bottom-up phase is omitted, however, the algorithm would be unable to find the subsumption $\top \sqsubseteq D$, and hence would give the wrong result.

Haarslev and Möller (Haarslev and Möller, 2001a) modified the basic approach in order to allow the bottom-up phase to be omitted in more cases. They utilized a *directly refers to* relation, which is similar to directly uses, but where references occurring in the scope of role quantifiers are not considered. Again, *refers to* is the transitive closure of directly refers to, and this relation induces a partial order relation on sets of concept names. A total order compatible with this relation is called a *quasi-definition order*.

Usage of a quasi-definition order instead of definition order presupposes that the subsumption relation between primitive concepts cannot be derived from information inside role quantifiers. This is not true in the presence of inverse roles. For example, in the TBox

$$C \sqsubseteq D, D \sqsubseteq \exists R.\forall R^{-}.C$$

using a quasi-definition order will result in a classification taxonomy where $D \prec C$, while semantically $C \equiv D$.

5.2. Told Subsumers and Disjoints

Various optimisations can be used in order to minimise the number of subsumption tests needed in classification. For example, when adding a concept C to the taxonomy, a top-down breadth first traversal can be used that only checks if D subsumes C when it has already been determined that C is subsumed by all the concepts in the taxonomy that subsume D (Baader et al., 1994). The structure of TBox axioms can also be also used to compute a set of *told subsumers* of C (i.e., trivially obvious subsumers). (See Section 3.2.3 for the formal definition of a told subsumer.) As subsumption is immediate for told subsumers, no tests need to be performed w.r.t. these concepts. In order to maximise the benefit of this optimisation, all of the told subsumers of a concept C should be classified *before* C itself is classified.

The told subsumer optimisation can be used to approximate the position of C in the taxonomy: all of its told subsumers can be marked as subsumers of C. The most specific concepts in this set of marked concepts are then candidates to be parents of C. In the standard algorithm, however, it is necessary to (recursively) check if the children of these concepts are also subsumers of C. This can be costly in the case where one of the told subsumers has a very large number of children. When it has been determined for some subsumer D of C that none of the children of D subsume C, then we know that D is a parent of C.

At the end of the top-down phase, the set of parents of C has been computed: all of the concepts in this set, along with all their super-concepts, are subsumers of C; all other concepts are non-subsumers of C. The next step is to determine the set of children of C (as mentioned above, this step can sometimes be omitted for a primitive concept when concepts have been classified in definitional order). This bottom-up phase is very similar to (the reverse of) the top-down one, and we won't describe it here—interested readers can refer to (Baader et al., 2003) for full details. It is, however, worth mentioning that, prior to the bottom-up phase, non-subsumption results can be propagated down the classified hierarchy (Baader et al., 1994): If, when classifying C, it is determined that D is not a subsumer of C, then clearly no concept D' subsumed by D can be a subsumer of C, i.e., $C \not\sqsubseteq D$ implies $C \not\sqsubseteq D'$ for all D' such that $D' \sqsubseteq D$. Thus, all descendents of D in the classified hierarchy can be marked as non-subsumers of C.

Finally, in addition to using told subsumers to propagate positive subsumption information, it is possible to use *told disjoints* to propagate negative subsumption information (Haarslev and Möller, 2001a). A concept D is *told disjoint* with concept C if the definition of C looks like $C \sqsubseteq \neg D \sqcap D'$. Having told disjoint information, it is possible to immediately determine non-subsumption (and propagate this information if necessary).

5.3. Using Simple but Incomplete [non-]Subsumption Tests

In some cases, it is possible to replace expensive subsumption tests with other (incomplete, but cheap) tests.

If one has the completion graph $\mathbf{G}_C$ for C, it is possible to save (partial) information about $\mathbf{G}_C$ in a structure usually called a *pseudo-model*. The pseudo-model contains information about concepts that appeared in the root node label of $\mathbf{G}_C$, in particular information about named concepts and about complex concepts involving roles. In order to check whether two concepts can label the same node (i.e., if the intersection of the two concepts is satisfiable), it is possible to check if the relevant pseudo models can be *merged*. Informally, two pseudo-models can be merged if a) the union of their named concept sets does not contain a clash, and b) no complex concept involving a role (e.g., $\exists R.C$) in one pseudo model interacts with a complex concept involving a role (e.g., $\forall R.D$) in the other pseudo model. If the TBox does not contain nominals, then a successfully merged pseudo model implies the existence of a "real" model for the intersection of the two concepts (i.e., one produced by merging suitable models for the individual concepts). For a more detailed description of this procedure, see (Horrocks, 2003; Haarslev et al., 2001a).

This technique can be used as a cheap way to check for non-subsumption. When checking the subsumption $C \sqsubseteq D$, for example, cached pseudo-models for C and $\neg D$ can be used (such pseudo-models can be generated and cached on the fly if necessary). If these pseudo-models can be merged, and the TBox is nominal-free (or at least the parts of it reachable from C and D are nominal-free), then the concept $C \sqcap \neg D$ is satisfiable and the subsumption $C \sqsubseteq D$ does not hold.

The pseudo-model merging technique relies on the fact that, if the completion graphs from which the pseudo-models are derived were merged, then they could only interact via their (common) root node. In the presence of nominals, however, things become more complicated, as interactions between merged graphs might also occur via nominal nodes. For example, if one graph includes a nominal node a labelled with C, and another one has the same node labelled with $\neg C$, then the combined graph will contain a clash. It is not difficult to imagine other examples in which more subtle interactions occurring via one or more nominals could also lead to a clash. In view of these problems, FaCT++ takes a conservative approach, and simply gives up if more than one of the pseudo-models to be merged was derived from a completion graph containing nominal nodes.

5.4. Clustering

In the case of wide (and shallow) taxonomies, one taxonomy node (representing a concept D) may have tens or hundreds of children (representing concepts $D_1, \ldots, D_n$). If, when classifying a concept C, it is determined that $C \sqsubseteq D$, then n subsumption tests will be needed in order to determine if $C \sqsubseteq D_i$ for each of $D_1, \ldots, D_n$. Typically, at most one of these tests will succeed (i.e., $C \sqsubseteq D_i$ for at most one of the D_i), as such taxonomies are usually tree-like.

In order to reduce this large number of failed subsumption tests, Haarslev and Möller (Haarslev and Möller, 2001a) propose a so-called *clustering technique*. For concepts $D_i \ldots D_j$, a new "virtual concept" $D_{ij} \equiv D_i \sqcup \ldots \sqcup D_j$ is inserted into the taxonomy. Instead of checking $C \sqsubseteq D_k$ for $i \leqslant k \leqslant j$, the subsumption $C \sqsubseteq D_{ij}$ is first checked, and if $C \not\sqsubseteq D_{ij}$, then it immediately follows that $C \not\sqsubseteq D_k$ for $i \leqslant k \leqslant j$. Moreover, when $D_i, \ldots, D_j$ are primitive concepts, this check can be done cheaply using the pseudo-model merging technique, because the label of the root node of the pseudo-model for $\neg D_{ij}$ is just $\{\neg D_i, \ldots, \neg D_j\}$.

The children of D can be gathered into clusters using several such virtual concepts. Typically, at most one of these clusters will need to be fully explored, with the rest being marked as non-subsumers as a result of a failed subsumption test w.r.t. the relevant virtual concept.

5.5. Completely Defined Concepts

In FaCT++, the notion of completely defined concepts is used in order to reduce the number of subsumption tests needed for classification. Given a TBox $\mathcal{T}$, a primitive concept C is said to be *completely defined* (CD) w.r.t. $\mathcal{T}$ when, for the definition $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$ in $\mathcal{T}$, it holds that:

1. For all $1 \leqslant i \leqslant n$, C_i is a primitive concept (*primitivity*).
2. There exist no i, j such that $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$ and C_j is a told subsumer of C_i (*minimality*).

When the TBox is obvious from the context we will talk about completely defined concepts without reference to a TBox.

If we assume a cycle-free TBox containing only CD concepts and no GCIs, then the classification process is very simple. In fact, we don't need to perform any subsumption tests at all: the position of every concept in the taxonomy is *completely defined* by its told subsumers. If concepts are processed in definitional order, then when a concept C is classified, where C is defined by the axiom $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$, the parents of C are $C_1, \ldots, C_n$, and the only child of C is $\bot$ (Tsarkov and Horrocks, 2005a). Note that every concept in such a taxonomy is satisfiable, because there is no use of negation.

This fact is, however, of very little practical value due to the very stringent conditions on the structure of the TBox. In the following, we will show how the basic technique can be made more useful by weakening some of these conditions.

5.5.1. *Primitivity*

In general, a CD concept should not have non-primitive concepts in its definition. This is because when the taxonomy already includes non-primitive concepts (which will be the case given definitional order classification) the bottom-up phase can not be omitted, and the CD method could therefore lead to incorrect results. Assume, e.g., a TBox

$$\{C \sqsubseteq C_1 \sqcap C_2 \sqcap C_3, \quad C' \equiv C_1 \sqcap C_2\}. \tag{1}$$

Using the CD classification approach, C will be classified under C_1, C_2 and C_3, whereas it should be classified under C' and C_3.

One case in which this condition can be weakened is for *primitive synonyms*. A non-primitive concept C is a primitive synonym if its definition is of the form $C \equiv D$, where D is a primitive concept. Synonyms (primitive or otherwise) may come from an application domain, or occur as a result of different preprocessing optimisations. It is easy to see that primitive synonyms don't require special classification: once D is classified, C will take the same place in the hierarchy, so adding primitive synonyms to the CD-only TBox still allows application of the CD approach.

5.5.2. *Minimality*

Non-minimal concepts may occur as a result, e.g., of badly designed TBoxes or the absorption of GCIs. The minimality check may, however, be removed from the definition of CD concepts provided that we check for any non-minimal concepts at classification time, i.e., we check that each C_i in a definition $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$ is really a parent of C (i.e., has no children that are subsumers of C). This check is relatively cheap, and is already implemented as part of the standard classification algorithm (see Section 5.5.5), where it is used to check which of the told subsumers of a concept C are possible parents of C.

5.5.3. *Non-CD Concepts*

CD-classification would be of limited interest if its applicability were limited to TBoxes consisting entirely of CD concepts. This is because most "interesting" TBoxes, including most TBoxes designed using DL based ontology languages, will contain concept constructors other than conjunction, and this will lead to some concepts being non-CD; in its current form the CD approach would, therefore, be of very limited use.

On the other hand, almost all TBoxes will contain *some* CD concepts. In this case, it may be possible to split the TBox into two parts—a CD part (i.e., containing only CD concepts) and a non-CD part—and use the CD algorithm only for the CD part.

Note that such a split will not introduce any problems if the CD part of the classification is performed first—in fact the classification of the CD part is independent of the non-CD part of the TBox, because the definitions of CD concepts only refer to other CD concepts.

In the TBox (1) above, for example, concepts C_1, C_2, C_3 and C will be in the CD part, and C' in the non-CD part. After CD-classification C will have three parents, and the standard algorithm will then insert C' with parents C_1, C_2 and child C.

In the case of definitional cycles, we can distinguish two kinds. The first (and simplest) is a cycle via concept names, as in the TBox $K = \{A \sqsubseteq B, B \sqsubseteq A \sqcap C\}$; cycles of this kind are eliminated during preprocessing (see Section 3.2.3). Any other kind of terminological cycle must involve non-CD concepts, and so must occur in the non-CD part of the TBox; in this case it will be dealt with in the normal way by the standard classification algorithm.

5.5.4. *General Axioms*

It is easy to see that in the general case the CD approach cannot be used in the presence of GCIs. Consider, for example, a TBox $\mathcal{T} = \mathcal{T}_u \cup \mathcal{T}_g$ with

$$\mathcal{T}_u = \{C \sqsubseteq \top, A \sqsubseteq D, B \sqsubseteq D\}, \mathcal{T}_g = \{\top \sqsubseteq A \sqcup B\}.$$

In this case, the CD algorithm classifies C under $\top$, whereas it should be classified under D. The CD approach is, therefore, applicable only if all GCIs in the TBox are absorbed (i.e., $T_g = \emptyset$) using the techniques described in Section 3.2.5.

5.5.5. *A Two-stage Approach for Completely Defined Concepts*

A two-stage CD classification algorithm has been implemented in FaCT++. After preprocessing, as described in Section 3.1, the TBox is classified using the following procedure:

1. If the TBox does not contain any GCIs, mark some concepts as CD. Namely, $\top$ is marked as CD; a primitive concept C is marked as CD if it has the definition $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$ and every C_i is marked CD; a non-primitive concept D is marked CD if it has the definition $D \equiv C$ and C is marked CD.

2. If the TBox contains concepts marked as CD, then run the CD-classifier. The CD classifier works only on those concepts that are marked CD, processing them in definitional order. For each such concept C, the steps it performs are as follows:

   a) If C is a synonym of some already classified concept D, then insert C at the same place as D.

   b) If C has the definition $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$, the concepts $C_1, \ldots, C_n$ are candidates to be parents of C.

   c) For every candidate C_i, check whether it is redundant, i.e. whether C_i has a child that is an ancestor of C. This can be done by labelling all ancestors of candidate concepts: labelled candidates will be redundant. Remove redundant candidates from the list of candidates.

   d) Insert C into the taxonomy with the remaining candidates as parents and $\perp$ as the only child.

3. Classify the remaining concepts using the standard classification algorithm.

5.5.6. *Empirical Evaluation*

We have tested our implementation using several large TBoxes derived from application ontologies. NCI is the US National Cancer Institute "thesaurus", an ontology containing the working vocabulary used in NCI data systems (see `http://ncicb.nci.nih.gov/NCICB/core/EVS/`); SNOMED is a "universal health care terminology" developed by the College of American Pathologists (see `http://www.snomed.org/`); GO is the Gene Ontology from the Gene Ontology Consortium; GALEN is the anatomical part of the well-known medical terminology ontology (Rogers et al., 2001). The structural characteristics of these TBoxes is summarised in the beginning of Table I, where `nCD` is the number of completely defined concepts, and `nNonCD` is the number of the other named concepts. All experiments used v.0.99.6 of FaCT++ running under Linux on a Pentium4 2.2GHz machine with 512Mb of memory.

The results of the classification tests are given in the rest of Table I, where `time` is the time taken to classify the TBox (in seconds), `nOps` is the number of tableau expansion rule applications during the classification process, and `nCache` is the number of subsumptions that were computed using cached pseudo-models (see Section 5.3).

As might be expected, the effectiveness of CD-classification depends largely on the structure of the TBox. For NCI and GO, both of which have many CD concepts and few non-primitive concepts, the CD optimisation reduces classification time by factors of approximately 20 and 2 respectively. It is interesting to note that, for NCI, all subsumption tests are solved cheaply using cached models. Without CD, however, more than ten million tests are performed; using CD reduces this number to less than one million. Classifying GO does require some satisfiability tests to be performed, but the great majority of subsumption tests are again solved cheaply using cached models. Like NCI, GO has a simple structure, with a very broad and shallow taxonomy, so CD still gives a very significant reduction in the number of these cache based tests that need to be performed.

Table I. Classification with- and without completely defined optimisation

TBox	nCD	nNonCD	CD	time	nOps	nCache
NCI	15,195	12,457	no	49.27	1,614,903	10,311,475
			yes	2.06	1,496,621	766,054
SNOMED	200,295	179,396	no	5,816	1,887,490,334	947,755,276
			yes	4,523	1,883,296,987	19,613,458
GO	11,718	2,211	no	4.06	744,497	5,184,061
			yes	1.48	729,666	625,879
GALEN	546	2,201	no	102.44	69,644,624	74,558
			yes	101.56	70,150,537	38,010

For SNOMED, which has quite a large number of non-primitive (and hence non-CD) concepts, the reduction factor is only about 1.3. This is because, although there is still a dramatic reduction in the number of cache based tests, the classification time is dominated by the relatively large number of satisfiability tests that need to be performed. Moreover, more than half of the total time is consumed in loading and preprocessing of the TBox, and this is not affected by the optimisation. For GALEN, the complex structure of the TBox means that the proportion of cache based tests is much smaller, and less than half of these can be avoided by CD. The classification time is again dominated by the relatively large number of satisfiability tests that need to be performed, and the benefit of CD is negligible. It is interesting to note that, in this case, CD actually leads to a small increase in the number of satisfiability tests being performed—this is due to the change in classification ordering.

In general, it is easy to see that CD optimisation can significantly reduce the number of subsumption tests, but mainly by eliminating those tests that can be performed using pseudo-model merging (i.e., "obvious" non-subsumptions). If such tests make up a significant proportion of all tests, then the impact can be quite large (more than an order of magnitude in the case of NCI). In other cases the impact is usually quite small, but it is unlikely that it will have a detrimental effect: the CD check is a syntactic one, and so is always likely to be cheaper than pseudo-model merging.

6. Discussion

6.1. Summary and Overview of Optimisation Techniques

An evaluation of the effectiveness of all of the presented techniques is beyond the scope of this paper; the interested reader will find evaluations of most of these techniques in the papers where they were first described. Instead we will summarise the various techniques, noting important interactions and indicating their relative importance in influencing the overall performance of a tableau-based reasoner.

6.1.1. *Preprocessing Optimisations*

As mentioned in Section 3, the preprocessing optimisations described in this paper take at most polynomial time to run, and will hardly ever have a detrimental effect on reasoning performance. It is thus sensible to use them in almost every case. Moreover, these optimisations are not specific to tableau-based reasoners, and may also be useful with other kinds of DL reasoners.

Lexical Normalisation is particularly effective because it helps to improve the performance of the core satisfiability test by facilitating early clash detection. Lexical normalization also works well with the lazy unfolding optimisation. Similarly, most *lexical simplifications* are so simple that they take very little time to apply. Some of the simplifications defined in Section 3.1, though, are more costly and give positive effects only in very special cases; it is important, therefore, to be more circumspect when using such simplifications. *Lazy unfolding* is almost always very beneficial, and can never increase the number of rule applications, so there is no reason not to use it. It doesn't do any good in case no early clash occurs, but these cases are very rare in realistic ontologies.

Synonym replacement has linear complexity and works well together with *told cycle elimination* which, while not very useful by itself, is also very cheap and, if applicable, facilitates other optimisations (such as absorption and definition-order classification). These techniques can be effective with TBoxes obtained by merging several smaller TBoxes together; in this case synonyms and told cycles are much more likely to arise. *Redundant subsumption elimination* is also unlikely to be of much use with a typical TBox, but it is useful after told cycle elimination (and in general, in the presence of cycles of length 1). Eliminating such cycles allows definition order to be used, and so may allow the bottom-up phase of classification to be omitted.

Finally, *absorption* is well known to be a crucial optimisation for tableau-based reasoners. Although it is difficult to find an optimal absorption, or even to define what optimal might mean w.r.t. absorption, empirical evidence suggests that almost *any* absorption in which $\mathcal{T}_g = \emptyset$ will lead to better performance than one in which $\mathcal{T}_g \neq \emptyset$. Moreover, when $\mathcal{T}_g \neq \emptyset$, it is impossible to use other optimisations such as definition order and completely defined classification.

6.1.2. *Core Satisfiability Optimisations*

Reasoning with expressive DLs like $\mathcal{SHOIQ}$ is highly intractable, and in the worst case, the time required for a single satisfiability test will increase at least exponentially w.r.t. the size of the input TBox. It is, therefore, crucial to employ optimisations that try to improve the performance of the core satisfiability tester in *typical cases*.

Using a *ToDo List architecture* makes the reasoner more flexible, and allows for a wider range of heuristic tuning optimisations. In (Tsarkov and Horrocks, 2005a) we show that, in most cases, it is best to give branching rules the lowest priority (as opposed to the standard top-down approach that gives generating rules the lowest priority). Moreover, decision procedures for $\mathcal{SHOIQ}$ require some ordering on expansion rule application in order to ensure termination (Horrocks and Sattler, 2005), a requirement that is easily incorporated into the ToDo list priority scheme. Note that, unlike a reasoner based on the top-down approach, a ToDo list-based reasoner cannot use the trace technique to stay in polynomial space. This is irrelevant when reasoning with $\mathcal{SHOIQ}$, however, as the problem is known to require exponential space in the worst case.

Backjumping is another optimisation that is crucial for effective tableau-based reasoning. Although there is a cost in building and carefully maintaining dependency sets, this is vastly outweighed (in all realistic cases) by the benefits that are obtained from this optimisation.

As mentioned in (Horrocks and Patel-Schneider, 1999), using *semantic branching* gives significant increases in performance on artificial tests (where the emphasis is often on propositional reasoning). With real applications, however, our experiments show that this optimisation produces only very small (although almost always positive) effects. *BCP* has the ability to turn non-deterministic operations into deterministic ones, and can thus provide significant benefits (although mainly when used in conjunction with semantic branching, and on artificial tests). In any case, BCP does not have any detrimental effect, as it simply prevents the reasoner from choosing disjuncts that immediately lead to a clash.

Tsarkov and Horrocks (Tsarkov and Horrocks, 2005a) have shown that the choice of *disjunction expansion heuristics* can have a very large effect on the performance of a reasoner (the difference in some cases was more than three orders in magnitude). Unfortunately, there is no currently known way to consistently choose the best (or even a good) set of heuristics, so most systems opt for a set of heuristics that usually behave reasonably, and rarely exhibit very bad behaviour. Developing better heuristics, and selecting good heuristics, e.g., on the basis of the input TBox, is an area for future research.

Note that some heuristics (like MOMS), interact badly with backjumping (which is one of the most important optimisation techniques in satisfiability testing). This sort of interaction means that care must be taken when choosing heuristics.

Caching optimisations can be highly effective, and can produce significant improvements in the speed of reasoning (Horrocks and Patel-Schneider, 1999). Unfortunately, caching optimisations are difficult to apply in the presence of inverse roles (although recent work has shown how this problem can be partly overcome (Ding and Haarslev, 2005; Ding and Haarslev, 2006)).

Finally, "low level" optimisations can also lead to significant performance gains. For example, *lazy saving* is very important with complex TBoxes that lead to the generation of large completion graphs; it is also very important if the TBox contains a large number of nominals, as the completion graph will usually include many or all of them (Sirin et al., 2005a). *Optimised Blocking* is also important with more complex TBoxes, in particular those with TBox cycles, where blocking is often required in order to ensure termination. Horrocks and Sattler (Horrocks and Sattler, 2002) have shown that, in such cases, optimised blocking can improve reasoning performance by as much as 2-3 orders of magnitude with respect to standard blocking.

6.1.3. *Classification Optimisations*

Note that, while all classification optimisations save (at most) a number of subsumption tests that is quadratic in the number of named concepts in the TBox, this can still lead to significant improvements in performance, particularly if very expensive subsumption tests can be avoided. Moreover, such optimisations rarely have any significant negative effect on performance, so it makes sense to apply them in all cases.

Taxonomy Creation Order, *Told Subsumers* and *Caching Subsumptions* all work well when applicable, reducing the number of performed subsumption tests at the price of (at most) polynomial time computations. Several empirical evaluations (Baader et al., 1994; Haarslev and Möller, 2001a) have shown that they are of benefit with both artificial and application TBoxes.

Tsarkov and Horrocks (Tsarkov and Horrocks, 2005b) have shown that the *completely defined* classification optimisation gives the best results on large TBoxes with very simple structure. At the same time, it incurs only a very small additional cost, and does not adversely affect performance on general TBoxes, as most TBoxes contain at least a (large or small) part that can be dealt with using this technique.

6.1.4. *Tuning*

In practice, not all optimisations can be (usefully) applied to a given TBox: most of optimisations have clearly defined areas of application, and it would be possible to tune the (choice of) optimisations to suit a given TBox, or even a given satisfiability test. Such tuning might be performed on the basis of a (syntactic) analysis of the TBox, or in response to the run-time behaviour of the algorithm. Currently, however, FaCT++ simply uses a fixed setting that has been found to work well for a wide range of realistic TBoxes.

6.2. Non-Tableau DL Reasoners

In spite of the success of the tableau approach to building DL reasoners, there exist a number of reasoners that are based on alternative algorithmic techniques.

6.2.1. *CEL*

CEL (Suntisrivaraporn et al., 2006) is a reasoner for the description logic EL+ (Baader et al., 2005), supporting as its main reasoning task the computation of the subsumption hierarchy induced by EL+ ontologies. A distinguishing feature of CEL is that, unlike the other reasoners mentioned here, it implements a polynomial-time algorithm for a sub-Boolean DL.

The supported description logic EL+ offers a selected set of expressive means that are tailored towards the formulation of medical and biological ontologies, and CEL has been shown to work well with large ontologies of this kind. If the expressive power of EL+ is sufficient for a given application, then CEL is likely to be a good choice.

6.2.2. *Hoolet*

Hoolet (Tsarkov et al., 2004; Horrocks et al., 2005) is an implementation of a $\mathcal{SHOIQ}$ DL reasoner that uses a first order prover. In order to test TBox satisfiability, the TBox is translated into a first order theory (in the obvious way, based on the DL semantics), and this theory is given to a first order prover for consistency checking.

Hoolet is intended as a proof of concept rather than as an effective DL reasoner, and the approach used means that it is not guaranteed to terminate on all $\mathcal{SHOIQ}$ reasoning problems. However, in spite of this, it provides a useful tool for use on small illustrative examples and allows for easy experimentation with more expressive languages.

6.2.3. *KAON2*

KAON2 is “an infrastructure for managing OWL-DL, SWRL and F-Logic ontologies” (see `http://kaon2.semanticweb.org/`). The reasoning part of KAON2 consists of a $\mathcal{SHIQ}$ reasoner, which uses novel algorithms to reduce a $\mathcal{SHIQ}$ KB to a disjunctive datalog program (Hustadt et al., 2004). This allows for the application of well-known deductive database techniques, such as magic sets and join-order optimizations, to DL reasoning. Recent work has shown how to extend the resolution-based approach to the logic $\mathcal{SHOIQ}$ (Kazakov and Motik, 2006).

KAON2 is still quite new, and there is as yet relatively little empirical evidence to show how it will perform in comparison to existing tableau-based reasoners. Early results suggest, however, that it is likely to be very effective when the TBox is relatively simple, but there are very large numbers of individuals; it is less clear how effective it will be with large and/or complex TBoxes.

6.2.4. *MSPASS*

MSPASS (Hustadt and Schmidt, 2000; Hustadt et al., 1999) is a fully automated theorem prover that can handle a wide range of logics. Being an extension of SPASS, it is fundamentally a resolution prover for first-order logic (with equality). However, by using sophisticated translation techniques allied to particular resolution strategies, it can also decide satisfiability problems for numerous description logics, including $\mathcal{SHI}$.

Although MSPASS has been shown to be quite effective with artificial test problems, there is little evidence as to how it would behave as a DL classifier in realistic applications: it is not easy to perform such tests as MSPASS has no infrastructure for handling TBoxes or for classifying them.

6.3. Future Work

The FaCT++ reasoner employs a wide variety of optimisation techniques that are needed for acceptable performance when performing TBox reasoning in the $\mathcal{SHOIQ}$ Description Logic. Some of these are well-known, some are modifications of existing optimisations, and some are new to FaCT++.

Much more work remains to be done in devising and evaluating optimisations for expressive Description Logics like $\mathcal{SHOIQ}$. Among other areas, better heuristics are needed to guide the choices needed during tableau reasoning, better use of cached information is expected to provide significant benefits, and more opportunistic optimisations (noticing that a part of the TBox does not use constructs that make optimisations from simpler Description Logics invalid) will often provide benefits, particularly in realistic ontologies. Devising an automated strategy for selecting and tuning optimisations is also a promising direction for future work. We expect to continue to improve FaCT++ in all of these areas, as well as to add both existing and new optimisations directed towards ABox reasoning, an area in which FaCT++ is still relatively weak.

References

Baader, F., S. Brandt, and C. Lutz: 2005, 'Pushing the $\mathcal{EL}$ Envelope'. In: *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*.

Baader, F., D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.): 2003, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Baader, F., E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich: 1994, '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.

Baader, F. and U. Sattler: 2001, 'An Overview of Tableau Algorithms for Description Logics'. *Studia Logica* **69**(1), 5–40.

Baker, A. B.: 1995, 'Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results'. Ph.D. thesis, University of Oregon.

Calvanese, D., G. De Giacomo, and M. Lenzerini: 1998a, 'On the Decidability of Query Containment under Constraints'. In: *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*. pp. 149–158.

Calvanese, D., G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati: 1998b, 'Description Logic Framework for Information Integration'. In: *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*. pp. 2–13.

Chen, C., V. Haarslev, and J. Wang: 2005, 'LAS: Extending Racer by a Large Abox Store'. In: *Proceedings of the 2005 International Workshop on Description Logics (DL-2005), Edinburgh, Scotland, UK, July 26-28*. pp. 200–207.

Davis, M., G. Logemann, and D. Loveland: 1962, 'A machine program for theorem proving'. *Communications of the ACM* **5**, 394–397.

Davis, M. and H. Putnam: 1960, 'A Computing Procedure for Quantification Theory'. *J. of the ACM* **7**(3), 201–215.

Ding, Y. and V. Haarslev: 2005, 'Towards Efficient Reasoning for Description Logics with Inverse Roles'. In: *Proceedings of the 2005 International Workshop on Description Logics (DL-2005), Edinburgh, Scotland, UK, July 26-28*. pp. 208–215.

Ding, Y. and V. Haarslev: 2006, 'Tableau Caching for Description Logics with Inverse and Transitive Roles'. In: *Proceedings of the 2006 International Workshop on Description Logics (DL-2006)*.

Fensel, D., F. van Harmelen, I. Horrocks, D. McGuinness, and P. F. Patel-Schneider: 2001, 'OIL: An Ontology Infrastructure for the Semantic Web'. *IEEE Intelligent Systems* **16**(2), 38–45.

Freeman, J. W.: 1995, 'Improvements to Propositional Satisfiability Search Algorithms'. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Giunchiglia, E. and A. Tacchella: 2000, 'A Subset-Matching Size-Bounded Cache for Satisfiability in Modal Logics'. In: *Proc. of the 4th Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX 2000)*. pp. 237–251, Springer.

Giunchiglia, F. and R. Sebastiani: 1996, 'Building Decision Procedures for Modal Logics from Propositional Decision Procedures—the Case Study of Modal K'. In: M. A. McRobbie and J. K. Slaney (eds.): *Proc. of the 13th Int. Conf. on Automated Deduction (CADE'96)*, Vol. 1104 of *Lecture Notes in Artificial Intelligence*. pp. 583–597, Springer.

Haarslev, V. and R. Möller: 2000, 'Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles'. In: *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*. pp. 273–284.

Haarslev, V. and R. Möller: 2001a, '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)*. pp. 161–168.

Haarslev, V. and R. Möller: 2001b, 'Optimizing Reasoning in Description Logics with Qualified Number Restrictions'. In: *Proc. of the 2001 Description Logic Workshop*

(DL 2001). pp. 142–151, CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-49/`.

Haarslev, V. and R. Möller: 2003, 'Racer: A Core Inference Engine for the Semantic Web'. pp. 27–36.

Haarslev, V. and R. Möller: 2004, 'Optimization Techniques for Retrieving Resources Described in OWL/RDF Documents: First Results'. In: *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*. pp. 163–173.

Haarslev, V., R. Möller, and A. Turhan: 2001a, 'Exploiting Pseudo Models for TBox and ABox Reasoning in Expressive Description Logics'. In: R. Goré, A. Leitsch, and T. Nipkow (eds.): *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*. pp. 29–44, Springer-Verlag.

Haarslev, V., R. Möller, and M. Wessel: 2005, 'Description Logic Inference Technology: Lessons Learned in the Trenches'. In: I. Horrocks, U. Sattler, and F. Wolter (eds.): *Proc. International Workshop on Description Logics*.

Haarslev, V., M. Timmann, and R. Möller: 2001b, 'Combining Tableaux and Algebraic Methods for Reasoning with Qualified Number Restrictions'. In: *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*. pp. 152–161.

Hladik, J.: 2002, 'Implementation and Optimisation of a Tableau Algorithm for the Guarded Fragment'. In: U. Egly and C. G. Fermüller (eds.): *Proceedings of the International Conference on Automated Reasoning with Tableaux and Related Methods (Tableaux 2002)*, Vol. 2381 of *Lecture Notes in Artificial Intelligence*. Springer.

Hoffmann, J. and J. Koehler: 1999, 'A New Method to Index and Query Sets'. In: *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*. pp. 462–467.

Horrocks, I.: 1997, 'Optimising Tableaux Decision Procedures for Description Logics'. Ph.D. thesis, University of Manchester.

Horrocks, I.: 1998, 'Using an Expressive Description Logic: FaCT or Fiction?'. In: *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*. pp. 636–647.

Horrocks, I.: 2003, 'Implementation and Optimisation Techniques'. In: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, Chapt. 9, pp. 306–346.

Horrocks, I., U. Hustadt, U. Sattler, and R. Schmidt: 2006, 'Computational modal logic'. In: P. Blackburn, J. van Benthem, and F. Wolter (eds.): *Handbook of Modal Logic*. Elsevier.

Horrocks, I. and P. F. Patel-Schneider: 1998, 'DL Systems Comparison'. In: *Proc. of the 1998 Description Logic Workshop (DL'98)*, Vol. 11 of *CEUR* (`http://ceur-ws.org/`). pp. 55–57.

Horrocks, I. and P. F. Patel-Schneider: 1999, 'Optimizing Description Logic Subsumption'. *J. of Logic and Computation* **9**(3), 267–293.

Horrocks, I. and P. F. Patel-Schneider: 2003, 'Reducing OWL Entailment to Description Logic Satisfiability'. In: D. Fensel, K. Sycara, and J. Mylopoulos (eds.): *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*. pp. 17–29, Springer.

Horrocks, I., P. F. Patel-Schneider, S. Bechhofer, and D. Tsarkov: 2005, 'OWL Rules: A Proposal and Prototype Implementation'. *J. of Web Semantics* **3**(1), 23–40.

Horrocks, I., P. F. Patel-Schneider, and F. van Harmelen: 2002, 'Reviewing the Design of DAML+OIL: An Ontology Language for the Semantic Web'. In: *Proc. of the 18th Nat. Conf. on Artificial Intelligence (AAAI 2002)*. pp. 792–797, AAAI Press.

Horrocks, I., P. F. Patel-Schneider, and F. van Harmelen: 2003, 'From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language'. *J. of Web Semantics* **1**(1), 7–26.

Horrocks, I. and U. Sattler: 1999, 'A Description Logic with Transitive and Inverse Roles and Role Hierarchies'. *J. of Logic and Computation* **9**(3), 385–410.

Horrocks, I. and U. Sattler: 2002, 'Optimised Reasoning for $\mathcal{SHIQ}$'. In: *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*. pp. 277–281.

Horrocks, I. and U. Sattler: 2005, 'A Tableaux Decision Procedure for $\mathcal{SHOIQ}$'. In: *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. pp. 448–453.

Horrocks, I., U. Sattler, and S. Tobies: 1999, 'Practical Reasoning for Expressive Description Logics'. In: H. Ganzinger, D. McAllester, and A. Voronkov (eds.): *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*. pp. 161–180, Springer.

Horrocks, I., U. Sattler, and S. Tobies: 2000, 'Reasoning with Individuals for the Description Logic $\mathcal{SHIQ}$'. In: D. McAllester (ed.): *Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000)*, Vol. 1831 of *Lecture Notes in Computer Science*. pp. 482–496, Springer.

Horrocks, I. and S. Tobies: 2000a, 'Optimisation of Terminological Reasoning'. In: *Proc. of the 2000 Description Logic Workshop (DL 2000)*. pp. 183–192.

Horrocks, I. and S. Tobies: 2000b, 'Reasoning with Axioms: Theory and Practice'. In: *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*. pp. 285–296.

Hudek, A. K. and G. Weddell: 2006, 'Binary Absorption in Tableaux-Based Reasoning for Description Logics'. In: *Proc. of the 2006 Description Logic Workshop (DL 2006)*, Vol. 189. CEUR (`http://ceur-ws.org/`).

Hustadt, U., B. Motik, and U. Sattler: 2004, 'Reducing SHIQ-Description Logic to Disjunctive Datalog Programs'. In: *Proc. of the 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2004)*. pp. 152–162.

Hustadt, U. and R. A. Schmidt: 2000, 'MSPASS: Modal Reasoning by Translation and First-Order Resolution'. In: R. Dyckhoff (ed.): *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, Vol. 1847 of *Lecture Notes in Artificial Intelligence*. pp. 67–71, Springer.

Hustadt, U., R. A. Schmidt, and C. Weidenbach: 1999, 'MSPASS: Subsumption Testing with SPASS'. In: P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider (eds.): *Proc. of Intern. Workshop on Description Logics'99*. pp. 136–137, Linköping University.

Kalyanpur, A., B. Parsia, and J. Hendler: 2005, 'A Tool for Working with Web Ontologies'. *International Journal on Semantic Web and Information Systems* **1**(1), 36–49.

Kazakov, Y. and B. Motik: 2006, 'A Resolution-Based Decision Procedure for $\mathcal{SHOIQ}$'. In: *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, Vol. 4130 of *Lecture Notes in Artificial Intelligence*. pp. 662–677, Springer.

Knublauch, H., R. Fergerson, N. Noy, and M. Musen: 2004, 'The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications'. In: S. A. McIlraith, D. Plexousakis, and F. van Harmelen (eds.): *Proc. of the 2004 International Semantic Web Conference (ISWC 2004)*. pp. 229–243, Springer.

Lutz, C.: 1999, 'Complexity of Terminological Reasoning Revisited'. In: *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, Vol. 1705 of *Lecture Notes in Artificial Intelligence*. pp. 181–200, Springer.

Massacci, F.: 1999, 'TANCS Non Classical System Comparison'. In: *Proc. of the 3rd Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX'99)*, Vol. 1617 of *Lecture Notes in Artificial Intelligence*.

McGuinness, D. L. and J. R. Wright: 1998, 'An Industrial Strength Description Logic-based Configuration Platform'. *IEEE Intelligent Systems* pp. 69–77.

Möller, R.: 2001, 'Expressive Description Logics: Foundations for Practical Applications'. Habilitation Thesis, University of Hamburg, Computer Science Department.

Oppacher, F. and E. Suen: 1988, 'HARP: A Tableau-Based Theorem Prover'. *J. of Automated Reasoning* **4**, 69–100.

Pan, Z.: 2005, 'Benchmarking DL Reasoners Using Realistic Ontologies'. In: *Proc. of the First OWL Experiences and Directions Workshop*.

Protégé: 2003, '`http://protege.stanford.edu/`'.

Rector, A.: 2003, 'Medical Informatics'. In: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, Chapt. 13, pp. 415–435.

Rogers, J. E., A. Roberts, W. D. Solomon, E. van der Haring, C. J. Wroe, P. E. Zanstra, and A. L. Rector: 2001, 'GALEN Ten Years On: Tasks and Supporting tools'. In: *Proc. of MEDINFO2001*. pp. 256–260.

Schmidt-Schauß, M. and G. Smolka: 1991, 'Attributive Concept Descriptions with Complements'. *Artificial Intelligence* **48**(1), 1–26.

Sirin, E., B. C. Grau, and B. Parsia: 2005a, 'Optimizing Description Logic Reasoning for Nominals: First Results'. Technical report, University of Maryland Institute for Advanced Computes Studies (UMIACS), 2005-64. Available online at http://www.mindswap.org/papers/OptimizeReport.pdf.

Sirin, E., B. C. Grau, and B. Parsia: 2006, 'From Wine to Water: Optimizing Description Logic Reasoning for Nominals'. In: *International Conference on the Principles of Knowledge Representation and Reasoning (KR-2006)*.

Sirin, E., B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz: 2005b, 'Pellet: A Practical OWL-DL Reasoner'. To appear.

Stevens, R., C. Goble, I. Horrocks, and S. Bechhofer: 2002, 'Building a bioinformatics ontology using OIL'. *IEEE Transactions on Information Technology in Biomedicine* **6**(2), 135–141.

Suntisrivaraporn, B., F. Baader, and C. Lutz: 2006, 'CEL—A Practical Reasoner for Life Science Ontologies'. In: *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*.

Tsarkov, D. and I. Horrocks: 2004, 'Efficient Reasoning with Range and Domain Constraints'. In: *Proc. of the 2004 Description Logic Workshop (DL 2004)*. pp. 41–50.

Tsarkov, D. and I. Horrocks: 2005a, 'Optimised Classification for Taxonomic Knowledge Bases'. In: *Proc. of the 2005 Description Logic Workshop (DL 2005)*, Vol. 147 of *CEUR (`http://ceur-ws.org/`)*.

Tsarkov, D. and I. Horrocks: 2005b, 'Ordering Heuristics for Description Logic Reasoning'. In: *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. pp. 609–614.

Tsarkov, D. and I. Horrocks: 2006, 'FaCT++ Description Logic Reasoner: System Description'. In: *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, Vol. 4130 of *Lecture Notes in Artificial Intelligence*. pp. 292–297, Springer.

Tsarkov, D., A. Riazanov, S. Bechhofer, and I. Horrocks: 2004, 'Using Vampire to Reason with OWL'. In: S. A. McIlraith, D. Plexousakis, and F. van Harmelen (eds.): *Proc. of the 2004 International Semantic Web Conference (ISWC 2004)*. pp. 471–485, Springer.

Wolstencroft, K., A. Brass, I. Horrocks, P. Lord, U. Sattler, R. Stevens, and D. Turi: 2005, 'A Little Semantic Web Goes a Long Way in Biology'. In: *Proc. of the 2005 International Semantic Web Conference (ISWC 2005)*. pp. 786–800, Springer.