Optimised EXPTIME Tableaux for $\mathcal{ALC}$ Using Sound Global Caching, Propagation and Cutoffs

Rajeev Goré[1]* and Linh Anh Nguyen[2]

[1] The Australian National University and NICTA
Canberra ACT 0200, Australia
Rajeev.Gore@anu.edu.au

[2] Institute of Informatics, University of Warsaw
ul. Banacha 2, 02-097 Warsaw, Poland
nguyen@mimuw.edu.pl

Abstract. We show that global caching can be used with propagation of both satisfiability and unsatisfiability in a sound manner to give an EXPTIME algorithm for checking satisfiability w.r.t. a TBox in the basic description logic $\mathcal{ALC}$. Our algorithm is based on a simple traditional tableau calculus which builds an and-or graph where no two nodes of the graph contain the same formula set. When a duplicate node is about to be created, we use the pre-existing node as a proxy, even if the proxy is from a different branch of the tableau, thereby building global caching into the algorithm from the start. Doing so is important since it allows us to reason explicitly about the correctness of global caching. We then show that propagating both satisfiability and unsatisfiability via the and-or structure of the graph remains sound.

We show that various search strategies can be used with global caching, including heuristics ones. To obtain a more efficient decision procedure for $\mathcal{ALC}$, we apply several optimisation techniques.

Of particular importance are the propagation of **both** satisfiability and unsatisfiability of a given node through the whole graph via subset-checking between node contents. Another is the minimisation of unsatisfiable sets: that is, when the status of a node becomes unsatisfiable, we identify an (ideally minimal) inconsistent subset, and replace the content of the node with this "inconsistency" since the smaller set has a greater likelihood of propagating its unsatisfiability via subset-checking. We also use a dual procedure for maximising satisfiable sets.

We do cutoffs for various search strategies, in particular, backjumping for depth-first search, by propagating "inconsistencies" and unsatisfiability through the and-or graph, and avoiding unnecessary expansions.

By combining global caching, propagation and cutoffs, our framework reduces the search space more significantly than the framework of Donini and Massacci [3]. Also, the freedom to use arbitrary search heuristics significantly increases the application potential of our framework.

* National ICT Australia is funded by the Australian Government's Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

1 Introduction

Description logics are (multi-modal) logics that represent the domain of interest in terms of concepts, objects, and roles. They are useful for modelling and reasoning about structured knowledge. The tableau method has been widely applied for description logics for different problems like: whether a concept is satisfiable; whether an ABox is consistent; whether a concept is satisfiable w.r.t. a TBox; and whether an ABox is consistent w.r.t. a TBox. See [1] for a survey.

The traditional naive tableau method for checking whether a concept C is satisfiable w.r.t. a TBox Γ leads to a double-exponential time algorithm because each tableau branch may have an exponential length, meaning that the whole tableau may have a double-exponential number of nodes. To counter this, Donini and Massacci [3] have given what is apparently the first tableau algorithm for this problem which runs in EXPTIME. Their (non-traditional) tableaux manipulate prefixed formulae of the form $\sigma : \varphi$ where σ is a string that names a particular world in the underlying model built by the tableau. The prefixing is used to distinguish which traditional formulae reside at which worlds. Donini and Massacci also proposed various techniques and optimisations to improve the performance of their tableau method for proof search.

The algorithm given by Donini and Massacci uses depth-first search, permanently caches "*all and only unsatisfiable sets of concepts*", and temporarily caches visited nodes on the current branch, even though this means that *"many potentially satisfiable sets of concepts are discarded when passing from a branch to another branch"*. Donini and Massacci state that the caching optimisation technique for the problem of checking satisfiability of a concept w.r.t. a TBox in $\mathcal{ALC}$ *"prunes heavily the search space but its unrestricted usage may lead to unsoundness [37]. It is conjectured that 'caching' leads to EXPTIME-bounds but this has not been formally proved so far, nor the correctness of caching has been shown."* [3, Page 89]. Later they explain that this situation arises because *"the caching optimisations are left out of the formal descriptions"* [3, Page 126].

Here we show that there is a simple way to use global caching and propagation to achieve an EXPTIME decision procedure for $\mathcal{ALC}$. Our algorithm is based on a simple traditional tableau calculus. It builds an and-or graph, where an or-node reflects the application of an "or" branching rule as in a tableau, while an and-node reflects the choice of a tableau rule and possibly many different applications of that rule to a given node of a tableau. We build caching into the construction of the and-or graph by ensuring that no two nodes of the graph have the same content. The status of a non-end-node is computed from the status of its successors using its kind (and-node/or-node) and treating satisfiability w.r.t. the TBox (i.e. `sat`) as true and unsatisfiability w.r.t. the TBox (i.e. `unsat`) as false. When a node gets status `sat` or `unsat`, the status is propagated to its predecessors in a way appropriate to the graph's and-or structure.

Recall that the problem of checking satisfiability of a concept w.r.t. a TBox in $\mathcal{ALC}$ is EXPTIME-complete. With global caching and the assumption that EXPTIME $\neq$ PSPACE, depth-first search has no advantages over other search strategies for our framework. That is, the naive version of our EXPTIME al-

gorithm can accept any systematic search strategy. We propose to try heuristic strategies and learn good parameters from large sets of tests.

To obtain an efficient decision procedure for $\mathcal{ALC}$, we apply several optimisation techniques. One is to propagate **both** `sat` and `unsat` via subset-checking between node contents. Another is to isolate and propagate the causes of unsatisfiability: that is, when the status of a node becomes `unsat`, we compute an "inconsistent" subset of its content that causes unsatisfiability and change the content of the node to that "inconsistent" subset. The smaller the content of the node becomes, the greater its potential for propagating its `unsat` status via subset-checking to other nodes. We do cutoffs for various search strategies, in particular, backjumping for depth-first search, by propagating "inconsistencies" and propagating `unsat` through the and-or graph. Finally, we avoid unnecessary expansions by expanding a node only if it is accessible from the initial node of the and-or graph via a path which is free of `unsat` nodes and `sat` nodes. By combining global caching, propagation and cutoffs, our framework significantly reduces the search space when compared with the framework of Donini and Massacci. Furthermore, the freedom to use arbitrary search heuristics significantly increases the application potential of our framework.

The rest of this paper is structured as follows. In Section 2, we recall the notation and semantics of $\mathcal{ALC}$. In Section 3, we present our tableau calculus for $\mathcal{ALC}$ and prove its soundness, and in Section 4, we prove its completeness. In Section 5, we present a simple EXPTIME decision procedure for $\mathcal{ALC}$ that is based on the calculus and uses global caching and propagation. In Section 6, we present our optimised decision procedure for $\mathcal{ALC}$. Section 7 contains comparisons with related work and Section 8 contains concluding remarks.

2 Notation and Semantics of $\mathcal{ALC}$

We use A for atomic concepts, use C and D for arbitrary concepts, and use R for a role name. Concepts in $\mathcal{ALC}$ are formed using the following BNF grammar:

$$C, D ::= \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid C \sqsubseteq D \mid C \doteq D \mid \forall R.C \mid \exists R.C$$

An *interpretation* $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ consists of a non-empty set $\Delta^{\mathcal{I}}$, the *domain* of $\mathcal{I}$, and a function $\cdot^{\mathcal{I}}$, the *interpretation function* of $\mathcal{I}$, that maps every atomic concept to a subset of $\Delta^{\mathcal{I}}$ and every role name to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to complex concepts as shown in Figure 1.

An interpretation $\mathcal{I}$ *satisfies* a concept C if $C^{\mathcal{I}} \neq \emptyset$, and *validates* a concept C if $C^{\mathcal{I}} = \Delta^{\mathcal{I}}$. Clearly, $\mathcal{I}$ *validates* a concept C iff it does not *satisfy* $\neg C$.

A TBox (of global axioms/assumptions) Γ is a finite set of concepts: traditionally, a TBox is defined to be a finite set of terminological axioms of the form $C \doteq D$, where C and D are concepts, but the two definitions are equivalent. An interpretation $\mathcal{I}$ is a *model* of Γ if $\mathcal{I}$ validates all concepts in Γ. We also use X, Y to denote finite sets of concepts. We say that $\mathcal{I}$ *satisfies* X if there exists $d \in \Delta^{\mathcal{I}}$ such that $d \in C^{\mathcal{I}}$ for all $C \in X$. Note: satisfaction is defined "locally", and $\mathcal{I}$ satisfies X does not mean that $\mathcal{I}$ is a model of X.

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}} \qquad \bot^{\mathcal{I}} = \emptyset \qquad (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

$$(C \sqsubseteq D)^{\mathcal{I}} = (\neg C \sqcup D)^{\mathcal{I}} \qquad (C \doteq D)^{\mathcal{I}} = ((C \sqsubseteq D) \sqcap (D \sqsubseteq C))^{\mathcal{I}}$$

$$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}}\}$$

$$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \text{ and } \ b \in C^{\mathcal{I}}\}.$$

Fig. 1. Interpretation of Complex Concepts

We say that Γ *entails* C, and write $\Gamma \models C$, if every model of Γ validates C. We say that C is satisfiable w.r.t. Γ if some model of Γ satisfies $\{C\}$. Similarly, X is satisfiable w.r.t. (a TBox of global axioms/assumptions) Γ if there exists a model of Γ that satisfies X. Observe that $\Gamma \models C$ iff $\neg C$ is unsatisfiable w.r.t. Γ.

Note: We now assume that concepts are in *negation normal form*, where $\doteq$ and $\sqsubseteq$ are translated away and $\neg$ occurs only directly before atomic concepts. It is well known that, in $\mathcal{ALC}$, every concept C has a logically equivalent concept C' which is in negation normal form.

3 A Tableau Calculus for $\mathcal{ALC}$

We consider tableaux with a fixed TBox of global axioms/assumptions Γ. A *tableau rule* consists of a TBox Γ, a numerator X, and a (finite) list of denominators $Y_1, Y_2, \ldots, Y_k$, all of which are finite sets of concepts. Such a rule (ρ) is written in the form

$$(\rho)\ \Gamma : \frac{X}{Y_1 \mid \ldots \mid Y_k}$$

The TBox Γ is omitted if it is not essential for the rule: for example, in the "static" rules (defined later). As we shall see later, each rule is read downwards as "if the numerator X is $\mathcal{ALC}$-satisfiable w.r.t. the TBox Γ, then some denominator Y_i is also $\mathcal{ALC}$-satisfiable w.r.t. Γ". The numerator of each tableau rule contains one or more distinguished concepts called the *principal concepts*.

We write $X;Y$ for $X \cup Y$, and $X;C$ for $X \cup \{C\}$.

The tableau calculus $\mathcal{CALC}$ for $\mathcal{ALC}$ consists of the following tableau rules:

$$(\bot)\ \frac{X\ ;\ A\ ;\ \neg A}{\bot} \qquad (\sqcap)\ \frac{X\ ;\ C \sqcap D}{X\ ;\ C\ ;\ D} \qquad (\sqcup)\ \frac{X\ ;\ C \sqcup D}{X\ ;\ C \mid X\ ;\ D}$$

$$(\exists R)\ \Gamma : \frac{X\ ;\ \exists R.C}{\{D : \forall R.D \in X\}\ ;\ C\ ;\ \Gamma}$$

The rules $(\bot)$, $(\sqcap)$, and $(\sqcup)$ are *static* rules, while $(\exists R)$ is a *transitional rule*.

A $\mathcal{CALC}$-tableau (tableau, for short) w.r.t. a TBox Γ for a finite set X of concepts is a tree with root $(\Gamma; X)$ whose nodes carry finite sets of concepts

obtained from their parent nodes by instantiating a $\mathcal{CALC}$-tableau rule with the proviso that: if a child s carries a set Y and no rule is applicable to Y or Y has already appeared on the branch from the root to s then s is an *end node*.

A branch in a tableau is *closed* if its end node carries only $\perp$. A tableau is *closed* if every one of its branches is closed. A tableau is *open* if it is not closed.

A finite set X of concepts is *consistent* w.r.t. a TBox Γ if every tableau w.r.t. Γ for X is open. If some tableau w.r.t. Γ for X is closed then X is *inconsistent* w.r.t. Γ. Calculus $\mathcal{CALC}$ is *sound* if for all finite sets Γ and X of concepts, X is satisfiable w.r.t. Γ implies X is consistent w.r.t. Γ. It is *complete* if for all finite sets Γ and X of concepts, X is consistent w.r.t. Γ implies X is satisfiable w.r.t. Γ.

Example 1. Let $\Gamma = \{A \sqsubseteq B \sqcap C\} \equiv \{\neg A \sqcup (B \sqcap C)\}$ be the TBox. We want to check satisfiability of the following concept w.r.t. Γ :

$$(\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$$

Here is a closed tableau w.r.t. the TBox Γ for the above concept where a superscript $*$ marks the principal concept of every rule application:

$$\dfrac{\neg A \sqcup (B \sqcap C);\ (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup^* (\exists R.A \sqcap \exists R.(A \sqcap \neg C))}{\dfrac{\neg A \sqcup (B \sqcap C); \exists R.A \sqcap^* \exists R.(A \sqcap \neg B)}{\dfrac{\neg A \sqcup (B \sqcap C); \exists R.A; \exists R^*.(A \sqcap \neg B)}{\dfrac{A \sqcap^* \neg B; \neg A \sqcup (B \sqcap C)}{\dfrac{A; \neg B; \neg A \sqcup^* (B \sqcap C)}{\dfrac{A; \neg B; \neg A}{\perp} \;\Big|\; \dfrac{A; \neg B; B \sqcap^* C}{\dfrac{A; \neg B; B; C}{\perp}}}}}} \;\Bigg|\; \dfrac{\neg A \sqcup (B \sqcap C); \exists R.A \sqcap^* \exists R.(A \sqcap \neg C)}{\dfrac{\neg A \sqcup (B \sqcap C); \exists R.A; \exists R^*.(A \sqcap \neg C)}{\dfrac{A \sqcap^* \neg C; \neg A \sqcup (B \sqcap C)}{\dfrac{A; \neg C; \neg A \sqcup^* (B \sqcap C)}{\dfrac{A; \neg C; \neg A}{\perp} \;\Big|\; \dfrac{A; \neg C; B \sqcap^* C}{\dfrac{A; \neg C; B; C}{\perp}}}}}}}$$

As shown below, the calculus $\mathcal{CALC}$ is sound, hence the given concept is unsatisfiable w.r.t. the TBox Γ.

A tableau rule is *sound* if it has the property that if the numerator is $\mathcal{ALC}$-satisfiable w.r.t. the TBox then one of the denominators is $\mathcal{ALC}$-satisfiable w.r.t. the TBox.

Lemma 1. *The calculus $\mathcal{CALC}$ is sound.*

Proof. The calculus $\mathcal{CALC}$ is sound because all rules of $\mathcal{CALC}$ are sound.

Let $Sf(C)$ denote the set of all sub-concepts of C (including itself) and let

$$Sf(X) = \bigcup_{C \in X} Sf(C) \cup \{\perp\}.$$

Observe that if C is a concept appearing in a tableau w.r.t. Γ for X then $C \in Sf(\Gamma \cup X)$. Our calculus $\mathcal{CALC}$ thus has the *analytic subformula property*.

4 Completeness

We prove completeness of our calculus via model graphs following [12, 5, 10].

4.1 Proving Completeness via Model Graphs

A model graph is a tuple $\langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$, where:

Δ is a finite set;
τ is a distinguished element of Δ;
$\mathcal{C}$ is a function that maps each element of Δ to a set of concepts; and
$\mathcal{E}$ is a function that maps each role name to a binary relation on Δ.

A model graph $\langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$ is *saturated* if every $x \in \Delta$ satisfies:

1. if $C \sqcap D \in \mathcal{C}(x)$ then $\{C, D\} \subseteq \mathcal{C}(x)$
2. if $C \sqcup D \in \mathcal{C}(x)$ then $C \in \mathcal{C}(x)$ or $D \in \mathcal{C}(x)$
3. if $\forall R.C \in \mathcal{C}(x)$ and $\mathcal{E}(R)(x, y)$ holds then $C \in \mathcal{C}(y)$
4. if $\exists R.C \in \mathcal{C}(x)$ then there exists $y \in \Delta$ with $\mathcal{E}(R)(x, y)$ and $C \in \mathcal{C}(y)$.

A saturated model graph $\langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$ is *consistent* if no $x \in \Delta$ has a $\mathcal{C}(x)$ containing $\bot$ or containing a pair A, $\neg A$ for some atomic concept A.

Given a model graph $M = \langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$, the *interpretation corresponding to M* is the interpretation $\mathcal{I} = \langle \Delta, \cdot^{\mathcal{I}} \rangle$ where $A^{\mathcal{I}} = \{x \in \Delta \mid A \in \mathcal{C}(x)\}$ for every atomic concept A and $R^{\mathcal{I}} = \mathcal{E}(R)$ for every role name R.

Lemma 2. *If $\mathcal{I}$ is the interpretation corresponding to a consistent saturated model graph $\langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$, then for every $x \in \Delta$ and $C \in \mathcal{C}(x)$ we have $x \in C^{\mathcal{I}}$.*

Proof. By induction on the structure of C.

Given finite sets X and Γ of concepts, where X is consistent w.r.t. Γ, we construct a model of Γ that satisfies X by constructing a consistent saturated model graph $\langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$ with $X \subseteq \mathcal{C}(\tau)$ and $\Gamma \subseteq \mathcal{C}(x)$ for every $x \in \Delta$.

4.2 Saturation

The rules $(\sqcap)$ and $(\sqcup)$ do not carry their principal concept into their denominators. For these rules, let (ρ') be the version that carries the principal concept into each of its denominators. Each new rule is clearly sound for $\mathcal{ALC}$.

For a finite set X of concepts that is consistent w.r.t. a TBox Γ, a set Y of concepts is called a *saturation of* X w.r.t. Γ if Y is a maximal set consistent w.r.t. Γ that is obtainable from X (as a leaf node in a tableau) by applications of the rules $(\sqcap')$ and $(\sqcup')$.

A set X is *closed* w.r.t. a tableau rule if applying that rule to X gives back X as one of the denominators.

Algorithm 1
Input: a TBox Γ and a finite set X of concepts, where X is consistent w.r.t. Γ.
Output: a model graph $M = \langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$.

1. For an arbitrary node name τ, let $\Delta := \{\tau\}$, and $\mathcal{E}(R) := \emptyset$ for every role name R. Let $\mathcal{C}(\tau)$ be a saturation of $\Gamma \cup X$ and mark τ as unexpanded.
2. While Δ contains unexpanded elements, take one, say x, and do:
   (a) For every concept $\exists R.C \in \mathcal{C}(x)$:
      i. Let $Y = \{D \mid \forall R.D \in \mathcal{C}(x)\} \cup \{C\} \cup \Gamma$ be the result of applying rule $(\exists R)$ to $\mathcal{C}(x)$, and let Z be a saturation of Y.
      ii. If there exists a proxy $y \in \Delta$ with $\mathcal{C}(y) = Z$ then add pair (x, y) to $\mathcal{E}(R)$;
      iii. Else add a new element y with $\mathcal{C}(y) := Z$ to Δ, mark y as unexpanded, and add the pair (x, y) to $\mathcal{E}(R)$.
   (b) Mark x as expanded.

Fig. 2. Constructing a Model Graph

Lemma 3. *Let X be a finite set of concepts consistent w.r.t. a TBox Γ, and Y a saturation of X w.r.t. Γ. Then $X \subseteq Y \subseteq Sf(\Gamma \cup X)$ and Y is closed w.r.t. the rules $(\sqcap')$ and $(\sqcup')$. Furthermore, there is an effective procedure that constructs such a set Y from Γ and X.*

Proof. It is clear that $X \subseteq Y \subseteq Sf(\Gamma \cup X)$. Observe that if $(\sqcap')$ or $(\sqcup')$ is applicable to Y, then one of the corresponding instances of the denominators is consistent w.r.t. Γ. Since Y is a saturation of X w.r.t. Γ, it is closed w.r.t. the rules $(\sqcap')$ and $(\sqcup')$.

We construct a saturation of X w.r.t. Γ as follows: let $Y := X$; while $(\sqcap')$ or $(\sqcup')$ is applicable to Y and has a corresponding denominator instance Z which is consistent w.r.t. Γ and strictly contains Y, set $Y := Z$. At each iteration, $Y \subset Z \subseteq Sf(\Gamma \cup X)$, so this process always terminates. Clearly, the resulting set Y is a saturation of X w.r.t. Γ.

4.3 Constructing Model Graphs

Figure 2 contains an algorithm for constructing a model graph. Algorithm 1 assumes that X is consistent w.r.t. Γ and constructs a model of Γ that satisfies X. Algorithm 1 terminates because each $x \in \Delta$ has a unique finite set $\mathcal{C}(x)$, which is a subset of the finite set $Sf(\Gamma \cup X)$, so eventually Step 2(a)ii always finds a proxy. Note that Step 2(a)ii builds caching into the algorithm.

Lemma 4. *Let Γ be a TBox, X be a finite set of concepts consistent w.r.t. Γ, $M = \langle \Delta, \tau, \mathcal{C}, \mathcal{E} \rangle$ be the model graph constructed by Algorithm 1 for Γ and X, and $\mathcal{I}$ be the interpretation corresponding to M. Then $\mathcal{I}$ validates Γ and satisfies X.*

Proof. It is easy to see that M is a consistent saturated model graph. By the construction of M we have $X \subseteq \mathcal{C}(\tau)$ and $\Gamma \subseteq \mathcal{C}(x)$ for every $x \in \Delta$. Hence, by Lemma 2, the corresponding interpretation $\mathcal{I}$ validates Γ and satisfies X.

The following theorem immediately follows from Lemmas 2 and 4.

Theorem 1. *The calculus $\mathcal{CALC}$ is sound and complete.*

4.4 Complexity of Algorithm 1

To estimate the complexity of Algorithm 1, let n be the sum of the size of $\Gamma \cup X$.

The naive decision procedure for $\mathcal{ALC}$ that explores tableaux in the usual way may require double exponential time ($2^{2^{O(n)}}$) because each tableau branch may have an exponential length ($2^{O(n)}$), meaning that the whole tableau (tree) may have $2^{2^{O(n)}}$ nodes. When using such a naive method for checking consistency, computing a saturation may also require double exponential time since we have to check consistency for each node.

Algorithm 1 constructs an and-or graph with at most $2^{O(n)}$ nodes since there are at most 2^n different subsets of $Sf(\Gamma \cup X)$. Finding a proxy at Step 2(a)ii requires $2^{O(n)} \times 2^{O(n)} = 2^{2.O(n)}$ steps: the number of nodes in the graph times the size of the content of a node. Therefore, Algorithm 1 runs in $2^{O(n)} \times (2^{2^{O(n)}} + 2^{2.O(n)})$ steps: the number of nodes in the graph times the sum of the time for creating a saturation at Step 2(a)i and the time for finding a proxy at Step 2(a)ii or for creating a new node at Step 2(a)iii. That is, Algorithm 1 runs in double-exponential time ($2^{2^{O(n)}}$).

5 A Simple EXPTIME Decision Procedure for $\mathcal{ALC}$

By simulating the creation of saturations and checking whether the resulting model graph is consistent, it is easy to alter Algorithm 1 so that it explicitly checks, rather than assumes, that X is consistent with a given TBox Γ. By guessing which or-branch to follow, we create a *candidate* for simulating a saturation nondeterministically in linear time. Checking whether such a candidate contains a pair A and $\neg A$ for some atomic concept A is also done in linear time. Therefore, following the argument in Section 4.4, the resulting algorithm for checking whether X is consistent with a TBox Γ runs nondeterministically in $2^{O(n)} \times (O(n) + 2^{2.O(n)}) = 2^{O(n)}$ steps: that is in NEXPTIME.

In Figure 3 we present an EXPTIME decision procedure for $\mathcal{ALC}$ which directly uses the tableau rules of $\mathcal{CALC}$ to create an and-or graph as follows.

A node in the constructed and-or graph is a record with three attributes:

content: the set of concepts carried by the node
status: {`unexpanded`, `expanded`, `sat`, `unsat`}
kind: {`and-node`, `or-node`}

To check whether a given finite set X is satisfiable w.r.t. the given TBox Γ, the content of the initial node τ with status `unexpanded` is $\Gamma \cup X$. The main while-loop continues processing nodes until the status of τ is determined to be in {`sat`, `unsat`}, or until every node is expanded, whichever happens first.

Algorithm 2

Input: two finite sets of concepts Γ and X
Output: an and-or graph $G = \langle V, E\rangle$ with $\tau \in V$ as the initial node such that $\tau.status = \texttt{sat}$ iff X is satisfiable w.r.t. Γ

1. create a new node τ with $\tau.content := \Gamma \cup X$ and $\tau.status := \texttt{unexpanded}$;
   let $V := \{\tau\}$ and $E := \emptyset$;
2. while $\tau.status \notin \{\texttt{sat}, \texttt{unsat}\}$ and we can choose an unexpanded node $v \in V$ do:
   (a) $\mathcal{D} := \emptyset$;
   (b) if no $\mathcal{CALC}$-tableau rule is applicable to $v.content$ then $v.status := \texttt{sat}$
   (c) else if $(\bot)$ is applicable to $v.content$ then $v.status := \texttt{unsat}$
   (d) else if $(\sqcap)$ is applicable to $v.content$ giving denominator Y then
       $v.kind := \texttt{and-node}$, $\mathcal{D} := \{Y\}$
   (e) else if $(\sqcup)$ is applicable to $v.content$ giving denominators Y_1 and Y_2 then
       $v.kind := \texttt{or-node}$, $\mathcal{D} := \{Y_1, Y_2\}$
   (f) else
       i. $v.kind := \texttt{and-node}$,
       ii. for every $\exists R.C \in v.content$, apply $(\exists R)$ to $v.content$ giving denominator $\{D \mid \forall R.D \in v.content\} \cup \{C\} \cup \Gamma$ and add this denominator to $\mathcal{D}$;
   (g) for every denominator $Y \in \mathcal{D}$ do
       i. if some proxy $w \in V$ has $w.content = Y$ then add edge (v, w) to E
       ii. else let w be a new node, set $w.content := Y$, $w.status := \texttt{unexpanded}$, add w to V, and add edge (v, w) to E;
   (h) if ($v.kind = \texttt{or-node}$ and one of the successors of v has status $\texttt{sat}$)
       or ($v.kind = \texttt{and-node}$ and all the successors of v have status $\texttt{sat}$) then
       $v.status := \texttt{sat}$, $propagate(G, v)$
   (i) else if ($v.kind = \texttt{and-node}$ and one of the successors of v has status $\texttt{unsat}$)
       or ($v.kind = \texttt{or-node}$ and all the successors of v have status $\texttt{unsat}$) then
       $v.status := \texttt{unsat}$, $propagate(G, v)$
   (j) else $v.status := \texttt{expanded}$;
3. if $\tau.status \notin \{\texttt{sat}, \texttt{unsat}\}$ then
   for every node $v \in V$ with $v.status \neq \texttt{unsat}$, set $v.status := \texttt{sat}$;

Fig. 3. A Simple EXPTIME Decision Procedure for $\mathcal{ALC}$

Inside the main loop, Steps (2b) to (2f) try to apply one and only one of the tableau rules in the order $(\bot)$, $(\sqcap)$, $(\sqcup)$, $(\exists R)$ to the current node v. The set $\mathcal{D}$ contains the contents of the resulting denominators of v. If the applied tableau rule is $(\sqcap)$ then v has one denominator in $\mathcal{D}$; if the applied rule is $(\sqcup)$ then v has two denominators in $\mathcal{D}$; otherwise, each concept $\exists R.C \in v.content$ contributes one appropriate denominator to $\mathcal{D}$. At Step (2g), for every denominator in $\mathcal{D}$, we create the required successor in the graph G only if it does not yet exist in the graph: this step merely mimics Algorithm 1 and therefore uses global caching.

In Algorithm 2, a node that contains both A and $\neg A$ for some atomic concept A becomes an end-node with status $\texttt{unsat}$ (i.e. unsatisfiable w.r.t. Γ). A node to which no tableau rule is applicable becomes an end-node with status $\texttt{sat}$ (i.e.

Procedure *propagate*(G, v)
Parameters: an and-or graph $G = \langle V, E \rangle$ and $v \in V$ with $v.status \in \{\texttt{sat}, \texttt{unsat}\}$
Returns: a modified and-or graph $G = \langle V, E \rangle$

1. $queue := \{v\}$;
2. while *queue* is not empty do
3. (a) extract x from *queue*;
   (b) for every $u \in V$ with $(u, x) \in E$ and $u.status = \texttt{expanded}$ do
      i. if ($u.kind$ = `or-node` and one of the successors of u has status `sat`)
         or ($u.kind$ = `and-node` and all the successors of u have status `sat`) then
            $u.status := \texttt{sat}$, $queue := queue \cup \{u\}$
      ii. else if ($u.kind$ = `and-node` and one of the successors of u has status `unsat`)
         or ($u.kind$ = `or-node` and all the successors of u have status `unsat`) then
            $u.status := \texttt{unsat}$, $queue := queue \cup \{u\}$;

Fig. 4. Propagating Satisfiability and Unsatisfiability Through an And-Or Graph

satisfiable w.r.t. Γ). Both conclusions are **irrevocable** because each relies only on classical propositional principles and not on modal principles. That is, we do not need to undo either of these upon backtracking.

On the other hand, an application of ($\sqcup$) to a node v causes v to be an *or-node*, while an application of ($\sqcap$) or ($\exists R$) to a node v causes v to be an *and-node*. Steps (2h) and (2i) try to compute the status of such a non-end-node v using the kind (or-node/and-node) of v and the status of the successors of v, treating `unsat` as irrevocably `false` and `sat` as irrevocably `true`.

If these steps cannot determine the status of v as `sat` or `unsat`, then its status is set to `expanded`. But if these steps do determine the status of a node v to be `sat` or `unsat`, this information is itself propagated to the predecessors of v in the and-or graph G via the routine *propagate*(G, v), explained shortly.

The main loop ends when the status of the initial node τ becomes `sat` or `unsat` or all nodes of the graph have been expanded. In the latter case, all nodes with status $\neq$ `unsat` are given status `sat` (effectively giving the status *open* to tableau branches which loop). Again, caching is present at Step 2(g)i.

The procedure *propagate* used in the above algorithm is specified in Figure 4. As parameters, it accepts an and-or graph G and a node v with (irrevocable) status `sat` or `unsat`. The purpose is to propagate the status of v through the and-or graph and alter G to reflect the new information.

Initially, the queue of nodes to be processed contains only v. Then while the queue is not empty: a node x is extracted; the status of x is propagated to each predecessor u of x; and if the status of a predecessor u becomes (irrevocably) `sat` or `unsat` then u is inserted into the queue for further propagation.

This construction thus uses both caching and propagation techniques.

Proposition 1. *Algorithm 2 runs in EXPTIME.*

Proof. Let $G = \langle V, E \rangle$ be the graph constructed by Algorithm 2 for Γ and X and n be the size of input, i.e. the sum of the lengths of the concepts of $\Gamma \cup X$.

Each $v \in V$ has $v.content \subseteq Sf(\Gamma \cup X)$, hence $v.content$ has size $2^{O(n)}$. For all $v, w \in V$, if $v \neq w$ then $v.content \neq w.content$. Hence V contains $2^{O(n)}$ nodes.

Every $v \in V$ is expanded (by Steps (2a)–(2j)) only once and every expansion takes $2^{O(n)}$ time units not counting the execution time of procedure *propagate* since $v.content$ contains $2^{O(n)}$ concepts and we still have to search for proxies amongst possibly $2^{O(n)}$ nodes in V. When $v.status$ becomes `sat` or `unsat`, the procedure *propagate* executes $2^{O(n)}$ basic steps directly involved with v, so the total time of the executions of *propagate* is of rank $2^{2.O(n)}$. Hence Algorithm 2 runs in exponential time.

Lemma 5. *It is an invariant of Algorithm 2 that for every $v \in V$:*

1. *if $v.status =$ `unsat` then*
   - *$v.content$ contains both A and $\neg A$ for some atomic concept A,*
   - *or $v.kind =$ `and-node` and there exists $(v, w) \in E$ such that $w \neq v$ and $w.status =$ `unsat`,*
   - *or $v.kind =$ `or-node` and for every $(v, w) \in E$, $w.status =$ `unsat`;*
2. *if $v.status =$ `sat` then*
   - *no $\mathcal{CALC}$-tableau rule is applicable to $v.content$,*
   - *or $v.kind =$ `or-node` and there exists $(v, w) \in E$ with $w.status =$ `sat`,*
   - *or $v.kind =$ `and-node` and for every $(v, w) \in E$, $w.status =$ `sat`.*

(If $v.kind =$ `or-node` and $(v, w) \in E$ then $w \neq v$ since $w.content \neq v.content$.)

Proof. Lemma 5(1) clearly holds since these are the only three ways for a node to get status `unsat`. For Lemma 5(2) there is the possibility that the node gets status `sat` via Step 3 of Algorithm 2.

For a contradiction, assume that $v.status$ becomes `sat` because of Step 3 of Algorithm 2 and that all three clauses of the "then" part of Lemma 5(2) fail:

1. first, the rule assumed to be applicable to $v.content$ cannot be the ($\bot$)-rule as this would have put $v.status =$ `unsat`, contradicting our assumption that $v.status =$ `sat`. Thus the rule must be either ($\sqcup$) or ($\sqcap$) or ($\exists R$), meaning that $v.kind =$ `or-node` or $v.kind =$ `and-node` after this rule application.
2. second, if $v.kind =$ `or-node` then v must have two successors created by the ($\sqcup$)-rule since this is the only way for a node to have $v.kind =$ `or-node`. If neither successor has status `sat` then they must both have status `unsat`. But Algorithm 2 and procedure *propagate* always ensure that `unsat` is propagated whenever it is found. As soon as the `unsat` status of the latter of these two children is found, the ensuing call to *propagate* would have ensured that $v.status =$ `unsat`, contradicting our assumption that $v.status =$ `sat`.
3. third, if $v.kind =$ `and-node` then v has at least one successor w (say) with $(v, w) \in E$. If $w.status \neq$ `sat`, then we must have $w.status =$ `unsat`. But again, as soon as w gets status `unsat`, procedure *propagate* would ensure that $v.status =$ `unsat` too, contradicting our assumption that $v.status =$ `sat`.

Lemma 6. *Let $G = \langle V, E \rangle$ be the graph constructed by Algorithm 2 for Γ and X. For every $v \in V$, if $v.status =$ `unsat` then $v.content$ is inconsistent w.r.t. Γ.*

Proof. Using Lemma 5, we can construct a closed tableau w.r.t. Γ for $v.content$ by induction on the way v depends on its successors and by copying nodes to ensure that the resulting structure is a (tree) tableau rather than a graph.

Let $G = \langle V, E \rangle$ be the graph constructed by Algorithm 2 for Γ and X. For $v \in V$ with $v.status = \texttt{sat}$, we say that $v_0 = v, v_1, \ldots, v_k$ with $k \geq 0$ is a *saturation path of* v *in* G if for each $1 \leq i \leq k$, we have $v_i.status = \texttt{sat}$, the edge $E(v_{i-1}, v_i)$ was created by an application of $(\sqcap)$ or $(\sqcup)$, and $v_k.content$ contains no concepts of the form $C \sqcap D$ nor $C \sqcup D$. By Lemma 5, if $v.status = \texttt{sat}$ then there exists a saturation path of v in G.

Lemma 7. *Let* $G = \langle V, E \rangle$ *be the graph constructed by Algorithm 2 for* Γ *and* X*. For all* $v \in V$*, if* $v.status = \texttt{sat}$ *then every tableau w.r.t.* Γ *for* $v.content$ *is open.*

Proof. Choose any $v \in V$ with $v.status = \texttt{sat}$ and let T be an arbitrary tableau (tree) w.r.t. Γ for $v.content$.

We maintain a *current node* cn of T that will follow edges of T to pin-point an open branch of T. Initially we set $cn := v$. We also keep a (finite) saturation path σ of the form $\sigma_0, \ldots, \sigma_k$ for some $\sigma_0 \in V$ and call σ the *current saturation path in* G. At the beginning, set $\sigma_0 := v$, so v is a node of both T and G and let σ be a saturation path for σ_0 in G: we know σ exists since $v.status = \texttt{sat}$.

We maintain the following invariant where $cn.content$ is the set carried by cn:

Invariant: $\forall C \in cn.content. \exists i. 0 \leq i \leq k$, $C \in \sigma_i.content$.

Remark 1. Observe that if $C \in \sigma_i.content$ for some $0 \leq i \leq k$ and C is of the form A, $\neg A$, $\exists R.D$, or $\forall R.D$ then $C \in \sigma_k.content$ since the saturation process does not affect concepts of these forms. By the definition of saturation path, we know that $\sigma_k.status = \texttt{sat}$, hence the $(\bot)$-rule is not applicable to $\sigma_k.content$. Hence, the invariant implies that $cn.content$ does not contain a pair A, $\neg A$ for any atomic concept A, and thus the rule $(\bot)$ is not applicable to cn. Also, note that the universal quantification over C encompasses the existential quantification over i, so each C can have a different σ_i in the invariant.

Clearly, the invariant holds at the beginning with $i = 0$ since $\sigma_0 = v = cn$ is in σ. Depending upon the rule applied to cn in the tableau T, we maintain the invariant by changing the value of the current node cn of T and possibly also the current saturation path σ in G:

$(\sqcap)$: Tableau rule $(\sqcap)$ is applied to cn with principal concept $D_1 \sqcap D_2$, and the successor is $u \in T$.
For every $C \in u.content$, we have that $C \in cn.content$ or $C = D_1$ or $C = D_2$. We know that the invariant holds for all $C \in cn.content$ by assumption. So consider the case $C = D_1$ or $C = D_2$. Since $D_1 \sqcap D_2 \in cn.content$, and there are no $\sqcap$-concepts in σ_k, there must be some σ_j in σ such that $0 \leq j < k$ and $D_1 \sqcap D_2 \in \sigma_j.content$ and $D_1 \sqcap D_2$ was the principal concept of the static rule applied to $\sigma_j.content$. Thus, $\{D_1, D_2\} \subseteq \sigma_{j+1}.content$ and σ_{j+1} is in σ. Setting $cn := u$ therefore maintains the invariant without changing σ.

($\sqcup$): Tableau rule ($\sqcup$) is applied to cn with principal concept $D_1 \sqcup D_2$, and the successors are $u_1 \in T$ and $u_2 \in T$, with $D_1 \in u_1$ and $D_2 \in u_2$.
Since $D_1 \sqcup D_2 \notin \sigma_k$, the invariant gives us a rightmost σ_i in σ such that $0 \leq i < k$ and $D_1 \sqcup D_2 \in \sigma_i.content$. The content of σ_{i+1} in σ must have been created by applying ($\sqcup$) to $\sigma_i.content$ with $D_1 \sqcup D_2$ as the principal concept. Hence $\sigma_{i+1}.content$ must contain D_1 or D_2.
If $C \notin \{D_1, D_2\}$ and $C \in u_1.content$ then $C \in cn.content$ and the invariant holds by assumption. Similarly for u_2. For $C \in \{D_1, D_2\}$, setting $cn := u_1$ if $D_1 \in \sigma_{i+1}.content$ else setting $cn := u_2$ preserves the invariant.

($\exists R$): The tableau rule applied to cn is ($\exists R$) with principal concept $\exists R.D$, and the successor is $u \in T$.
Remark 1 and the invariant tell us that $\{\exists R.D\} \cup \{\forall R.C \mid \forall R.C \in cn.content\} \subseteq \sigma_k.content$. So there must be a node $w \in V$ such that the edge $E(\sigma_k, w)$ was created by the application of ($\exists R$) to $\sigma_k.content$ with $\exists R.D$ as the principal concept. Thus, $u.content \subseteq w.content$ by the form of the ($\exists R$)-rule. Moreover, σ_k is an and-node with $\sigma_k.status = \texttt{sat}$, hence $w.status \neq \texttt{unsat}$, meaning that $w.status = \texttt{sat}$.
Setting $cn := u$ and setting σ to be a saturation path of w in G maintains the invariant with $i = 0$ since the new σ has $\sigma_0 = w$.

By Remark 1, the branch formed by the instances of cn is an open branch of T.

Theorem 2. *Let $G = \langle V, E \rangle$ be the graph constructed by Algorithm 2 for Γ and X, with $\tau \in V$ as the initial node. Then X is satisfiable w.r.t. Γ iff $\tau.status = \texttt{sat}$.*

Proof. By Lemmas 6 and 7, X is consistent w.r.t. Γ iff $\tau.status = \texttt{sat}$ since $\tau.content = \Gamma \cup X$. Since the calculus $\mathcal{CALC}$ is sound and complete, it follows that X is satisfiable w.r.t. Γ iff $\tau.status = \texttt{sat}$.

Corollary 1. *Algorithm 2 is an EXPTIME decision procedure for $\mathcal{ALC}$.*

Proof. The EXPTIME complexity is established by Proposition 1.

6 An Efficient Decision Procedure for $\mathcal{ALC}$

We have proved in the previous section that using global caching and propagation (of satisfiability and unsatisfiability) is sufficient to obtain an EXPTIME decision procedure for $\mathcal{ALC}$. We kept Algorithm 2 as simple as possible. In this section, we adapt some well-known optimisation techniques into our framework to create an efficient algorithm.

6.1 Optimisations

We use the following useful optimisations to speed up our decision procedure.

Normalising Contents of Nodes We normalise a set Y of concepts by applying the following operations until no further change can be made:

Implicit ($\sqcap$)-rule: Replace every concept $C \sqcap D$ by $(C; D)$. By using the implicit ($\sqcap$)-rule, we avoid creating and-nodes due to the ($\sqcap$)-rule.
$\forall\sqcap$-distribution: Replace every concept $\forall R.(C \sqcap D)$ by $\forall R.C; \forall R.D$.
$\sqcup$-flattening: Replace every concept/subconcept of the form $(C_1 \sqcup \ldots \sqcup C_i) \sqcup (C_{i+1} \sqcup \ldots \sqcup C_{i+k})$ by $C_1 \sqcup \ldots \sqcup C_k$. We create only one or-node for a principal concept if it can be flattened.
$\exists\sqcup$-distribution: Replace every concept/subconcept $\exists R.(C_1 \sqcup \ldots \sqcup C_n)$ by $(\exists R.C_1) \sqcup \ldots \sqcup (\exists R.C_n)$.
Subsumption: Delete $C_1 \sqcup \ldots \sqcup C_n$ from Y if Y contains C_i or the set representing the normalisation of C_i for some $1 \leq i \leq n$.
Implicit Modus Ponens: If Y contains both $C \sqcup D$ and the set representing the negation normal form $nnf(\neg C)$ of $\neg C$ or the normalisation of $nnf(\neg C)$, then replace $C \sqcup D$ by D. In particular, if Y contains both $\neg A \sqcup D$ and A then replace $\neg A \sqcup D$ by D.

We write $nf(Y)$ for the set obtained from normalising Y. Then when a node v is created or modified we guarantee that $v.content = nf(v.content)$ by setting $v.content := nf(v.content)$ if necessary. Observe that normalising Y can be done in polynomial time in the size of Y, and the maximum size of Y is $2^{O(n)}$.

Using the Generalised ($\bot$)-Rule Let $\overline{C} := nf(nnf(\neg C))$. Due to the implicit ($\sqcap$)-rule, $\overline{C}$ is a set of concepts. We can use a clash $C; \overline{C}$ to derive `unsat`. That is, we use the following rule ($\bot'$) instead of ($\bot$) :

$$(\bot')\ \frac{X; C; \overline{C}}{\bot}$$

Checking and Propagating `sat` and `unsat` via Subset-Checking
If $Y \subset Z$ then:

1. Z is unsatisfiable w.r.t. a TBox Γ if Y is unsatisfiable w.r.t. Γ
2. Y is satisfiable w.r.t. Γ if Z is satisfiable w.r.t. Γ.

We utilise this fact when checking the status of newly created nodes (using the cache) and also incorporate it into the propagation process.

Propagating Inconsistencies and Keeping the Cache Compact
For each `unsat` node v, we compute a subset $UnSatCore(v)$ of $v.content$ that causes v to become `unsat`. Such an "inconsistent" subset is also unsatisfiable w.r.t. the TBox Γ. The smaller the subset $UnSatCore(v)$ is, the more we can propagate its `unsat` status via subset-checking. In general, there may be a number of minimal subsets of $v.content$ that may cause v to become `unsat`. We try to make $UnSatCore(v)$ as small as possible, but do not guarantee it to be minimal. In particular, we compute $UnSatCore(v)$ using backward propagation as follows:

Procedure $set_content(G, v, Z)$
Parameters: an and-or graph $G = \langle V, E\rangle$, a node $v \in V$, and a set $Z = nf(Z)$ s.t.
($v.status =$ `unsat` and $Z \subset v.content$) or ($v.status =$ `sat` and $Z \supset v.content$).
Purposes: Replace v by any other node with content Z or set the content of v to Z.

1. if some node $u \neq v \in V$ has $u.content = Z$ then replace v by u as follows:
   (a) replace every existing edge $(w, v) \in E$ by a (new) edge $(w, u) \in E$;
   (b) $u.status := v.status$;
   (c) delete the node v from V;
   (d) let v now denote the node u;
2. else $v.content := Z$;

Fig. 5. Procedure To Set the Content of $v \in G$ to Z

1. if $v.content$ contains a clash $C; \overline{C}$ then $UnSatCore_0(v) := \{C\} \cup \overline{C}$;
2. else if $v.kind =$ `or-node` and $C_1 \sqcup \ldots \sqcup C_n$ is the principal concept of $v.content$ then let w_i be the C_i-successor of v, for $1 \leq i \leq n$, and set

$$UnSatCore_0(v) := \{C_1 \sqcup \ldots \sqcup C_n\} \cup \bigcup_{1 \leq i \leq n} (UnSatCore(w_i) - nf(\{C_i\}));$$

3. else if ($v.kind =$ `and-node` and) w is a successor of v with $w.status =$ `unsat` and the edge (v, w) is caused by the concept $\exists R.C \in v.content$ then $UnSatCore_0(v)$ is specified to be a minimal $Z \subseteq v.content$ such that $\exists R.C \in Z$ and $nf(\{D \mid \forall R.D \in Z\} \cup \{C\} \cup \Gamma) \supseteq UnSatCore(w)$;
4. $UnSatCore(v) := UnSatCore_0(v) - nf(\Gamma)$;

When a node v becomes `unsat` we change its content to $UnSatCore(v)$ (assuming that $UnSatCore(v) \neq v.content$). This has the following effects:

– Since the content of v is smaller than before, the propagation of its `unsat` status via subset-checking is likely to affect more nodes.
– Further propagation of this "inconsistency" to predecessors of v can be done.
– The number of nodes with distinct contents may decrease.

Modifying the content of v may have the side effect of making two nodes of the graph have identical contents, which is undesirable, so we just merge the duplicates as shown in Figure 5, thereby keeping the cache compact.

Note that all of these operations require only polynomial time in the size of the cache. Also, the merging of nodes with identical contents does not conflict with the invariant about identical contents in Lemma 5 since we never apply Lemma 5 to Algorithm 3.

Avoiding Redundant Computations This may seem an obvious thing to aim for, but we want to emphasise three points. First, if v is an and-node then one successor of v becoming `unsat` is sufficient to decide that v is `unsat`. Similarly, if v is an or-node then one successor of v becoming `sat` is sufficient to decide

that v is `sat`. Second, if every path from the initial node τ to an unexpanded node v contains an `unsat` node or contains a `sat` node, then v does not affect the status of τ and expansion of v should be avoided. Third, global caching is a very useful technique for avoiding redundant computations.

Allowing Various Search Strategies In Algorithm 2, we did not specify how to choose the next node v to expand. Indeed, we can apply depth-first search (DFS), breadth-first search (BFS), or some heuristic to choose v. But since the problem of checking satisfiability of a concept w.r.t. a TBox is known to be EXPTIME-complete, we cannot hope to develop a PSPACE decision procedure for this problem (assuming that PSPACE $\neq$ EXPTIME).

That is, the ability to reclaim space upon backtracking by using DFS is no longer an important advantage of DFS over BFS when used for $\mathcal{ALC}$. In [3], Donini and Massacci use DFS as one of their main techniques. But that is because they cache only visited nodes on the current branch plus `unsat` nodes. That is, their caching method (for visited nodes) is specifically tailored for DFS.

Since our framework uses global caching, we see no advantages of DFS over other search strategies. We therefore believe that well-designed heuristics could serve just as well for choosing the next node to expand. Indeed, the algorithm should choose to expand the node v whose status becoming either `sat` or `unsat` will most strongly affect the fragment of the and-or graph essential for deciding the status of the root node τ.

Cutoffs Backjumping is a useful technique that is used for DFS to restrict the search space. For example, suppose that after applying the $(\sqcup)$-rule to $X; C \sqcup D$, the resulting C-branch $X; C$ closes, but in a way so that C is not actually used (or essential) for the closure. In that case, we can also close the D-branch $X; D$ without exploring it since the cause of the closure must come from X itself. Thus backjumping is a cutoff technique for DFS. We do cutoffs for various search strategies by propagating "inconsistencies", propagating `unsat` through the and-or graph, and only expanding nodes that are accessible from the initial node τ via a path free of `unsat` nodes and `sat` nodes. For the example above, if the successor $X; C$ of $X; C \sqcup D$ is `unsat` due to an (inconsistent) subset $Y \subset X$, then the `unsat` status of the "inconsistency" Y is propagated via subset-checking to the successor $X; D$ of $X; C \sqcup D$ and the D-branch of $X; C \sqcup D$ is pruned.

6.2 The Improved Decision Procedure for $\mathcal{ALC}$

We incorporate the previously discussed optimisations into Algorithm 2 to obtain Algorithm 3 in Figure 6. Recall that this entails the use of the implicit $(\sqcap)$ rule, meaning that the and-nodes correspond only to applications of the $(\exists R)$ rule. In the new algorithm, each edge of the and-or graph is labelled by a concept:

- if v is an or-node with the principal concept $C_1 \sqcup \ldots \sqcup C_n$ and w_i is the C_i-successor of v then the edge (v, w_i) is labelled by C_i;

– if v is an and-node and w is a successor of v due to $\exists R.C \in v.content$ then the edge (v, w) is labelled by $\exists R.C$.

We also modify the procedure *propagate* into $propagate_2$ of Figure 7 to propagate `sat` and `unsat` also via subset-checking and to propagate "inconsistencies" as well. The boolean $flag$ indicates whether or not to propagate the status of x via subset-checking. That is, if $u.content \subset x.content$ at Step 3(b)i then putting $(u, true)$ into the *queue* is superfluous since any w such that $w.content \subset u.content$ will also be processed via $w.content \subset x.content$ at the current incarnation of Step 3(b)i. Analogously but dually for Step 3(b)ii. Note that by the calls of $set_contents(G, u, x.content)$ in Steps 3(b)i and 3(b)ii, we keep only maximal `sat` nodes and minimal `unsat` nodes.

As an example, in Figure 8 we present the and-or graph created by Algorithm 3 for the concept $(\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$ and the TBox $\Gamma = \{A \sqsubseteq B \sqcap C\} \equiv \{\neg A \sqcup (B \sqcap C)\}$, which were used earlier in Example 1. The node (8) gets status `sat` and is used by 4 predecessors via global caching. The nodes (9) and (10) get status `unsat` and each of them is used by 2 predecessors via global caching. If DFS is used then no propagation of `sat` or `unsat` occurs. If BFS is used, then we have the following propagations of `unsat`: $(9) \to (6)$, $(10) \to (7)$, $(6) + (7) \to (3)$, $(3) + (2) \to (1)$.

It is not appropriate to compare the closed tableau from Example 1 with this and-or graph because the tableau is a result of searching and some branchings are not shown. But note that the and-or graph from Algorithm 3 is a little smaller than that from Algorithm 2 due to the implicit $(\sqcap)$ rule.

Figure 9 shows an application of Algorithm 3 using DFS for the example given by Haarslev and Möller in [7]. They use this example to demonstrate that caching has to be done carefully in their framework, but our algorithm requires no such care even though a number of propagations of `unsat` occur.

Theorem 3. *Algorithm 3 is an EXPTIME decision procedure for $\mathcal{ALC}$.*

Proof. We start from the result that Algorithm 2 is a correct decision procedure for $\mathcal{ALC}$. It is easy to see that all the incorporated optimisations preserve the satisfiability and unsatisfiability of the contents of nodes of the and-or graph. The search strategy used in the loop condition of Step 2 of Algorithm 3 as well as the modification for Step 3 of Algorithm 3 do not affect the final status of the initial node τ. Hence Algorithm 3 is a correct decision procedure for $\mathcal{ALC}$.

The complexity of Algorithm 3 is also EXPTIME because: the number of nodes in the and-or graph is always of rank $2^{O(n)}$; the graph is built incrementally since the changes made by procedure *set_content* only speed up the decision procedure; there is no hidden nondeterminism; and our basic operations on the and-or graph such as $propagate_2$ can be done in polynomial time in its size.

6.3 Further Optimisations

In this subsection, we list some other useful optimisations and discuss some potential optimisation techniques for our framework.

Algorithm 3

Input: two finite sets of concepts Γ and X.

Output: an and-or graph $G = \langle V, E\rangle$ with initial node $\tau \in V$ such that $\tau.status = \texttt{sat}$ iff X is satisfiable w.r.t. Γ

1. create a node τ with $\tau.content := nf(\Gamma \cup X)$ and $\tau.status := \texttt{unexpanded}$; set $V := \{\tau\}$, $E := \emptyset$, and $\Gamma' := nf(\Gamma)$;
2. while $\tau.status \notin \{\texttt{sat}, \texttt{unsat}\}$ and we can *select* (using DFS, BFS, or using some heuristic) an unexpanded node $v \in V$ that is accessible from τ via a path in G that contains no nodes with $status \in \{\texttt{sat}, \texttt{unsat}\}$ do:
   (a) if no $\mathcal{CALC}$-tableau rule is applicable to $v.content$ then $v.status := \texttt{sat}$
   (b) else if $v.content$ contains a clash $C; \overline{C}$ then
   $v.status := \texttt{unsat}$; $set_content(G, v, \{C\} \cup \overline{C} - \Gamma')$
   (c) else if $C_1 \sqcup \ldots \sqcup C_n \in v.content$ then
      i. $v.kind := \texttt{or-node}$; $i := 0$; $Z := \{C_1 \sqcup \ldots \sqcup C_n\}$; $flag := true$;
      ii. repeat
         A. $i := i + 1$;
         B. apply $(\sqcup)$ to $v.content$ with $C_1 \sqcup \ldots \sqcup C_n$ as the principal concept to obtain the i-th denominator Y_i;
         C. if $flag$ and no $w \in V$ has $w.status = \texttt{unsat}$ & $w.content \subseteq nf(Y_i)$ then $flag := false$;
         D. if $flag$ and $\exists w \in V.\ w.status = \texttt{unsat}$ & $w.content \subseteq nf(Y_i)$ then
            $Z := Z \cup (w.content - nf(\{C_i\}))$;
            if $i = n$ then $v.status := \texttt{unsat}$; $set_content(G, v, Z - \Gamma')$
         E. else if $\exists w \in V.\ w.status = \texttt{sat}$ & $w.content \supseteq nf(Y_i)$ then $v.status := \texttt{sat}$
         F. else if some proxy $w_i \in V$ has $w_i.content = nf(Y_i)$ then add edge (v, w_i) with label C_i to E
         G. else let w_i be a new node, set $w_i.content := nf(Y_i)$, $w_i.status := \texttt{unexpanded}$, add w_i to V, and add edge (v, w_i) with label C_i to E;

         until $(v.status = \texttt{sat})$ or $(i = n)$
   (d) else
      i. $v.kind := \texttt{and-node}$; $todo := \{\exists R.C \mid \exists R.C \in v.content\}$;
      ii. while $todo \neq \emptyset$ do
         A. delete some concept $\exists R.C$ from $todo$ and apply $(\exists R)$ to $v.content$ with $\exists R.C$ as principal concept to obtain denominator $Y = \{D \mid \forall R.D \in v.content\} \cup \{C\} \cup \Gamma$;
         B. if $\exists w \in V$ with $w.status = \texttt{sat}$ and $w.content \supseteq nf(Y)$ then skip
         C. else if $\exists w \in V$ with $w.status = \texttt{unsat}$ and $w.content \subseteq nf(Y)$ then
            $v.status := \texttt{unsat}$; let Z be a minimal subset of $v.content$ such that $\exists R.C \in Z$ and $nf(\{D \mid \forall R.D \in Z\} \cup \{C\} \cup \Gamma) \supseteq w.content$; $set_content(G, v, Z - \Gamma')$; $todo := \emptyset$
         D. else if some proxy $w \in V$ has $w.content = nf(Y)$ then add edge (v, w) with label $\exists R.C$ to E
         E. else let w be a new node, set $w.content := nf(Y)$, $w.status := \texttt{unexpanded}$, add w to V, and add edge (v, w) with label $\exists R.C$ to E;
      iii. if $v.status \neq \texttt{unsat}$ and v has no successors then $v.status := \texttt{sat}$;
   (e) if $v.status \in \{\texttt{sat}, \texttt{unsat}\}$ then $propagate_2(G, v)$
   (f) else $v.status := \texttt{expanded}$;
3. if $\tau.status \neq \texttt{unsat}$ then $\tau.status := \texttt{sat}$;

Fig. 6. An Improved Decision Procedure for $\mathcal{ALC}$

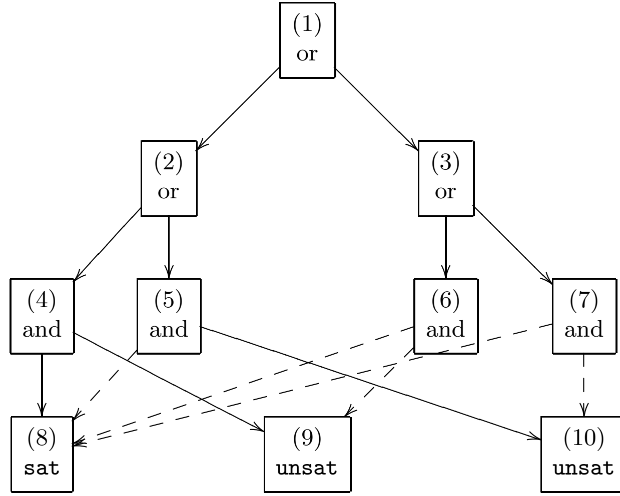
Procedure $propagate_2(G, v)$
Parameters: an and-or graph $G = \langle V, E\rangle$ and $v \in V$ with $v.status \in \{\mathtt{sat}, \mathtt{unsat}\}$
Global variables: a TBox Γ and $\Gamma' = nf(\Gamma)$
Returns: a modified and-or graph $G = \langle V, E\rangle$

1. $queue := \{(v, true)\}$;
2. while $queue$ is not empty do
3. (a) extract a pair $(x, flag)$ from $queue$;
   (b) if $flag$ then
      for every $u \in V$ with $u.status \notin \{\mathtt{sat}, \mathtt{unsat}\}$ do
      i. if $(u.content \subset x.content)$ and $(x.status = \mathtt{sat})$ then
         $u.status := \mathtt{sat}$, $queue := queue \cup \{(u, false)\}$,
         $set_content(G, u, x.content)$;
      ii. if $(x.content \subset u.content)$ and $(x.status = \mathtt{unsat})$ then
         $u.status := \mathtt{unsat}$, $queue := queue \cup \{(u, false)\}$,
         $set_content(G, u, x.content)$;
   (c) for all $u \in V$ with $(u, x) \in E$, $u.kind = \mathtt{or\text{-}node}$, and $u.status = \mathtt{expanded}$ do
      i. if $x.status = \mathtt{sat}$ then $u.status := \mathtt{sat}$, $queue := queue \cup \{(u, true)\}$
      ii. else if all successors of u have status $\mathtt{unsat}$ then
         A. $u.status := \mathtt{unsat}$, $queue := queue \cup \{(u, true)\}$;
         B. let $w_1, \ldots, w_n$ be the successors of u, and C_i be the label of the edge (u, w_i) for $1 \leq i \leq n$;
         C. $Z := \{C_1 \sqcup \ldots \sqcup C_n\} \cup \bigcup_{1 \leq i \leq n}(w_i.content - nf(\{C_i\}))$;
         D. $set_content(G, u, Z - \Gamma')$;
   (d) for all $u \in V$ with $(u, x) \in E$, $u.kind = \mathtt{and\text{-}node}$, and $u.status = \mathtt{expanded}$ do
      i. if $x.status = \mathtt{unsat}$ then
         A. $u.status := \mathtt{unsat}$, $queue := queue \cup \{(u, true)\}$;
         B. let $\exists R.C$ be the label of the edge (u, x);
         C. let Z be a minimal subset of $u.content$ such that $\exists R.C \in Z$ and $nf(\{D \mid \forall R.D \in Z\} \cup \{C\} \cup \Gamma) \supseteq x.content$;
         D. $set_content(G, u, Z - \Gamma')$
      ii. else if all the successors of u have status $\mathtt{sat}$ then
         $u.status := \mathtt{sat}$, $queue := queue \cup \{(u, true)\}$;

Fig. 7. Improved Propagation Through an And-Or Graph

The two optimisation techniques given below certainly speed up our decision procedure, but they are different in nature from the previous optimisations since they are more at the implementation level than the conceptual level:

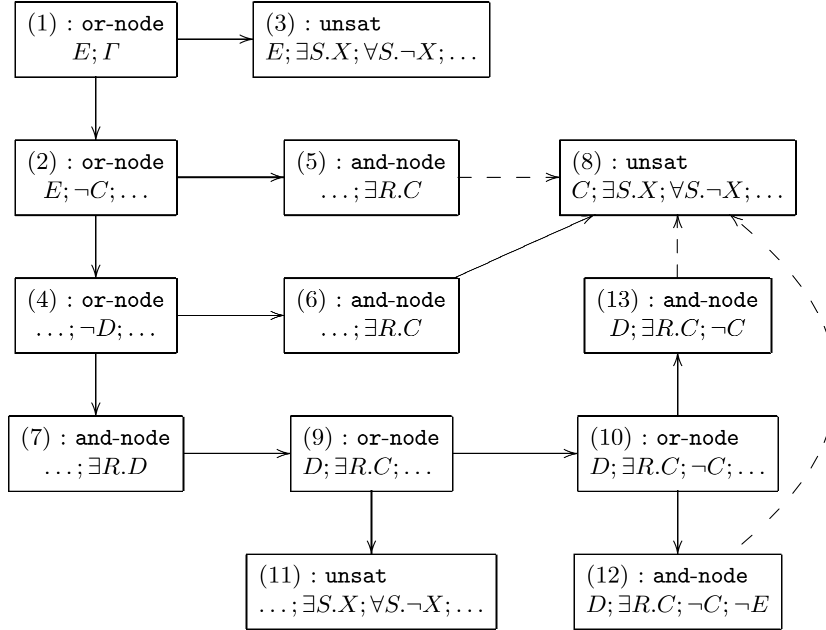
Fast Detection of SAT and UNSAT Suppose that an and-node v has n successors. We can avoid checking the status of all successors of v each time we wish to compute the status of v by maintaining an extra field in the node v which tracks the number of successors with status $\notin \{\mathtt{sat}, \mathtt{unsat}\}$ that may affect the status of v. When this number becomes 0, if node v does not yet have status `unsat` then we can set the status of v to `sat`. We can also apply this technique (in the dual way) for or-nodes.



Node	Content
(1)	$\neg A \sqcup (B \sqcap C); (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(2)	$\neg A; (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(3)	$B; C; (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(4)	$\neg A; \exists R.A; \exists R.(A \sqcap \neg B)$
(5)	$\neg A; \exists R.A; \exists R.(A \sqcap \neg C)$
(6)	$B; C; \exists R.A; \exists R.(A \sqcap \neg B)$
(7)	$B; C; \exists R.A; \exists R.(A \sqcap \neg C)$
(8)	$A; B; C$
(9)	$A; \neg B; B; C$
(10)	$A; \neg C; B; C$

Fig. 8. An and-or graph created by Algorithm 3 for the concept $(\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$ and the TBox $\Gamma = \{A \sqsubseteq B \sqcap C\} = \{\neg A \sqcup (B \sqcap C)\}$. Nodes are numbered using BFS. Dashed arrows are cache hits. The answer is "unsatisfiable".

TBox $\Gamma = \{C \sqsubseteq (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A);\ D \sqsubseteq \exists R.C;\ E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)\}$



Node	Content
(1)	$E; C \sqsubseteq^* (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A); D \sqsubseteq \exists R.C; (\exists R.C) \sqcup (\exists R.D)$
(2)	$E; \neg C; D \sqsubseteq^* \exists R.C; (\exists R.C) \sqcup (\exists R.D)$
(3)	$E; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; D \sqsubseteq \exists R.C$
(4)	$E; \neg C; \neg D; (\exists R.C) \sqcup^* (\exists R.D)$
(5)	$E; \neg C; \exists R^*.C$
(6)	$E; \neg C; \neg D; \exists R^*.C$
(7)	$E; \neg C; \neg D; \exists R^*.D$
(8)	$C; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; D \sqsubseteq \exists R.C; E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(9)	$D; \exists R.C; C \sqsubseteq^* (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A); E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(10)	$D; \exists R.C; \neg C; E \sqsubseteq^* (\exists R.C) \sqcup (\exists R.D)$
(11)	$D; \exists R.C; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(12)	$D; \exists R^*.C; \neg C; \neg E$
(13)	$D; \exists R^*.C; \neg C$

Fig. 9. An and-or graph created by Algorithm 3 for concept E and TBox Γ. Principal concepts are marked with superscript $*$. Nodes are numbered when created but explored using DFS: 1:(2,3), 2:(4,5), 4:(6,7), 6:8, 8, 7:9, 9:(10,11), 10:(12,13), 12:8, 13:8, 11, 5:8, 3. Dashed arrows are cache hits. Result is "unsatisfiable".

Efficient Loop-Checking and Subset-Checking Since the content of each node is a subset of $Sf(\Gamma \cup X)$, we can use a vector of bits to represent the content of a node, where each bit is used to include or exclude one concept. Then loop-checking reduces to checking whether the two vectors are equivalent and subset-checking reduces to checking the inclusion of the two vectors. Indeed, loop-checking can be further optimised by using a hash-table. We do not know a similar technique for subset-checking, but since this operation is used very often, it deserves further investigation.

To speed up subset-checking, as mentioned earlier, we keep only maximal `sat` nodes and minimal `unsat` nodes.

Note that both loop- and subset-checking are in $O(n.m)$, where n is the size of $\Gamma \cup X$ and m is the size of the corresponding cache. When m dominates n, this complexity is essentially linear in the size of the cache.

We now mention some potential optimisation techniques, which should be evaluated on a large scale to learn good parameters.

Semantic Branching To adopt semantic branching one can use the following rule instead of $(\sqcup)$:

$$(\sqcup^*) : \frac{X; C \sqcup D}{X; C \mid X; \overline{C}; D} \quad (\text{if } nf(\{C\}) \not\subseteq X \text{ and } nf(\{D\}) \not\subseteq X)$$

The rule $(\sqcup^*)$ does not always speed up the search (for the case when $X; C$ is `unsat` and $X; D$ is `sat`). However, we believe that, in general this rule is useful: if $X; D$ is `unsat` then it is easier to show that $X; \overline{C}; D$ is `unsat`; if $X; \overline{C}; D$ is `sat` then this may cause more propagation of `sat` via subset-checking (than does $X; D$). Note that semantic branching is one of the main techniques of efficient SAT solvers for classical propositional logic.

Heuristics for Expanding the And-Or Graph As stated earlier, an efficient implementation of our decision procedure should choose a node v for expansion so that the fragment of the and-or graph that is essential for deciding the status of τ will be strongly affected by v becoming either `sat` or `unsat`. We develop here one possible heuristic strategy for choosing such a node v. We first give some definitions.

A node x is said to be *essential* (for τ) if it may affect the satisfiability or unsatisfiability of τ*.content* w.r.t. Γ: that is, if there exists a path from τ to x which is free of `unsat` nodes and free of `sat` nodes.

The `sat` (resp. `unsat`) *propagation degree* of a node x is defined to be the number of nodes of the graph with status $\notin \{\mathtt{sat}, \mathtt{unsat}\}$ that are essential for τ and which will become `sat` (resp. `unsat`) as the result of giving x status `sat` (resp. `unsat`) and doing propagation. The *propagation degree* of a node x is defined to be the sum of the `sat` propagation degree and the `unsat` propagation degree of x.

The *propagation performance* of a period in an execution of our decision procedure is defined to be the number of nodes that have become `sat` or `unsat` during that period.

Normally, we would like to always expand an unexpanded node that has the highest propagation degree. But the problem is that computing propagation degrees of (all) nodes of the graph is costly. As a possible solution, one can try to apply some heuristic to compute and update approximations of the propagation degrees of nodes of the graph.
Here we propose another approach which is based on computing the propagation degrees of nodes of the graph only occasionally. Our algorithm for steering the process of expanding the and-or graph is as follows:

While $\tau.status \notin \{\texttt{sat}, \texttt{unsat}\}$ and there are essential nodes do:

1. Compute the propagation degrees of all essential nodes of the current graph and sort those nodes with a propagation degree greater than some parameter `PropaDegThreshold` into a queue `Q` in descending propagation degree;
2. Repeat
   (a) If `Q` is not empty then extract a node from `Q` for expansion (else expansion will be continued from the current node);
   (b) Repeat
   Execute the next `DFSPeriodLength` expansions using DFS, where `DFSPeriodLength` is a parameter, and compute the propagation performance `PropaPerf` of this period;
   Until `PropaPerf` is less than `PropaPerfThreshold`;

   Until `Q` is empty;

This steering algorithm tries to use best-first search, while reducing the number of calls of the evaluation function. It should be applied only when the graph has become large enough and the parameters should be learnt from large test suites.

Finally, we want to mention that the optimisations called "shallow reduction rules", "deep reduction rules", and "absorption" by Donini and Massacci [3] are not essential for our framework (when our optimisations are used).

Shallow reduction rules correspond to the static rule shown below left which can be improved to the rule shown below right:

$$\frac{X; \overline{C}; C \sqcup D}{X; \overline{C}; C \sqcup D; D} \qquad\qquad \frac{X; \overline{C}; C \sqcup D}{X; \overline{C}; D}$$

But this latter rule is part of our node content normalisation process.

Deep reduction rules are typified by

$$\frac{X; \forall R.C; \forall R.(C \sqcap D)}{X; \forall R.C; \forall R.D}$$

which is also included in our node content normalisation process.

Absorption corresponds to the following two rules:

$$\frac{X; \neg A \sqcup C; A \sqcup \overline{C}; A}{X; A; C} \qquad\qquad \frac{X; \neg A \sqcup C; A \sqcup \overline{C}; \neg A}{X; \neg A; \overline{C}}$$

but these speed up the search by only one step when our node content normalisation is in place.

7 Comparisons With Other Work

In [9], Horrocks and Patel-Schneider analysed optimisation techniques for description logics. The main optimisation techniques used for the advanced implemented systems FaCT [8] and DLP [11] are lexical normalisation, semantic branching, simplification, dependency directed backtracking (i.e. backjumping), heuristic guidance for search, and caching. Heuristic guidance is used to choose a branch to explore first between the (remaining) branches generated by a disjunction in DFS. In particular, the authors discuss variants of the MOMS (Maximum number of Occurences in disjunctions of Minimum Size) heuristic [4]. Recall that our framework has the freedom to use any search heuristics, including these. For the remaining optimisation techniques, lexical normalisation and simplification are part of our normalisation process while semantic branching, backjumping, and caching have been all considered in our framework. Note that both FaCT and DLP use depth-first search. Propagation of `sat` and `unsat` is not mentioned in [9]. Soundness of caching is not studied in [9] either.

In [2], Ding and Haarslev studied tableau caching for description logics with inverse and transitive roles. As expected, caching improves performance also for such description logics. The authors gave some sufficient conditions that guarantee soundness of caching. When used for $\mathcal{ALC}$, those conditions are too restrictive, while we do not require any condition for soundness of caching. Our global caching method can be adapted in a sound way for description logics with inverse and transitive roles. This will be addressed in our next work on efficient decision procedures for expressive description logics.

In the rest of this section, we compare our framework with the framework given by Donini and Massacci [3]. The main points are summarised below and analysed in more detail later:

1. Prefixes (also called labels) are superfluous for $\mathcal{ALC}$.
2. All of the specific techniques and optimisations proposed by Donini and Massacci have been considered for our framework and all the essential ones have been incorporated in our decision procedure.
3. Our framework uses optimisations not considered by Donini and Massacci.
4. By combining global caching, propagation and cutoffs, our framework explores a smaller search space than the framework of Donini and Massacci.
5. The freedom to use an arbitrary search heuristic increases the application potential of our framework over the DFS-based framework of [3].

Here are justifications for the above claims:

1. Prefixed (or labelled) tableau systems are useful for symmetric modal logics like converse PDL or $\mathcal{ALC}$ with inverse roles. However, prefixes are superfluous for $\mathcal{ALC}$. This is demonstrated by our simple traditional tableau calculus for $\mathcal{ALC}$. Furthermore, one can simplify and optimise the algorithm of Donini and Massacci by using a traditional tableau calculus for $\mathcal{ALC}$.

   More specifically, consider Techniques 4, 5, 6 used by Donini and Massacci:

Technique 4: For every prefixed set $e : \mathcal{C}$, apply rules α, pos, KB, and $\nu(R)$ before other rules, and apply rule β before rule $\pi(R)$.

Technique 5: The new prefix $e.R.n$ generated by rule $\pi(R)$ must be such that $n > m$ for every other integer m already present in the tableau.

Technique 6: Apply a rule to a prefixed set $e : \mathcal{C}$ only if there is no prefixed set $e' : \mathcal{D}$, with $e' < e$, to which a rule can be applied.

Informally, Technique 6 and Technique 5 jointly say that a prefixed set (a node in our terminology) is "resolved" before considering other prefixed sets. Additionally, in our terminology, Technique 4 says that the ($\sqcup$)-rule has the lowest priority amongst the static rules and that static rules should be applied to saturate a node (whose status is not known to be `unsat`) before using the transitional rule. Hence, the algorithm given by Donini and Massacci can be reformulated to utilise a traditional tableau calculus.

Such a reformulation simplifies the tableau calculus and also speeds up the performance of the algorithm. For example, the following fragment of [3, Page 98] implies that a number of applications of rule $\nu(R)$ must be combined with an application of rule $\pi(R)$ to simulate our single ($\exists R$)-rule: *"The combination of Technique 4 and 6 force the application of a $\nu(R)$ rule just after an application of a $\pi(R)$ rule. That is, just after rule $\pi(R)$ introduces a new prefix, all additional concepts ν_0 imposed by universal formulae $\nu(R)$ are transferred to the newly generated prefix by the application of $\nu(R)$-rule."*

2. Of the 7 techniques proposed by Donini and Massacci: Techniques 1–4 are already a part of our framework; Technique 5 is irrelevant for our framework as it concerns prefixes; Technique 7 (using DFS) is one of the options of our framework; and Technique 6 is a part of our framework when DFS is used. Furthermore, all the specific optimisations proposed by Donini and Massacci have been considered in our framework. As mentioned earlier, "absorption" is not essential for our framework as it speeds up the search by only one step. "Shallow reduction rules", "deep reduction rules", and "unit subsumption" are done by normalising the contents of nodes. "Backjumping" is done by our cutoff technique. "Semantic branching" is an optional optimisation that can be adopted for our framework. "Membership testing" is done by our loop-checking and "subset checking" is done by our subset-checking and propagation of "inconsistencies".
3. Our framework also uses other optimisations like: normalising node contents; a special representation of node contents for efficient loop-checking and subset-checking; and propagation of both satisfiability and unsatisfiability via subset-checking. Also observe that $UnSatCore(v)$ of an `unsat` node v is computed more tightly by our algorithm than by the algorithm of Donini and Massacci: compare Step 2(d)iiC of our Algorithm 3 and Step 3(d)i of our procedure $propagate_2$ with the case $\pi(R)$ of Fig. 8 of [3, Page 109].
4. Recall that the algorithm given by Donini and Massacci permanently caches "*all and only unsatisfiable sets of concepts*" and temporarily caches visited nodes on the current branch, even though this means that *"many potentially satisfiable sets of concepts are discarded when passing from a branch to another branch"* [3, Page 126]. Our algorithm caches the contents of all

nodes and never discards any information. With our propagation and cutoff techniques it reduces the search space significantly more than the algorithm of Donini and Massacci.
To illustrate this, consider the TBox $\Gamma = \{A \sqsubseteq C^\top\}$ and suppose $D_1^\bot$, ..., $D_k^\bot$, $D^\top$, $C^\top$ are complex concepts not containing A which are independent of each other. Suppose it is easy to show that each of $D_1^\bot$, ..., $D_k^\bot$ is unsatisfiable w.r.t. the TBox Γ, and also easy to show that $D^\top$ is satisfiable w.r.t. Γ, but that it is very costly to show that $C^\top$ is satisfiable w.r.t. Γ. Now suppose we have to check the satisfiability w.r.t. Γ of the concept below:

$$(\exists R.A \sqcap \exists R.D_1^\bot) \sqcup \ldots \sqcup (\exists R.A \sqcap \exists R.D_k^\bot) \sqcup D^\top$$

A node with content $\{A, C^\top\}$ is explored (and declared to be `sat`) by our algorithm only once, while it is explored by the algorithm of Donini and Massacci k times. Since this node is assumed to be very costly compared to the other nodes, our algorithm runs nearly k times faster than the algorithm by Donini and Massacci for this example.
We next show that we can instantiate $C^\top$ so that our algorithm runs nearly k^2 times faster than the algorithm of Donini and Massacci for the resulting example. Take $\Gamma' = \{A \sqsubseteq (\exists R.A' \sqcap \exists R.D_1'^\bot) \sqcup \ldots \sqcup (\exists R.A' \sqcap \exists R.D_k'^\bot) \sqcup D'^\top, A' \sqsubseteq C'^\top\}$ and assume that: $D_1'^\bot$, ..., $D_k'^\bot$, $D'^\top$, $C'^\top$ do not contain A and are independent from each other and independent from A'; and it is easy to show that each of $D_1'^\bot$, ..., $D_k'^\bot$ are unsatisfiable and $D'^\top$ is satisfiable w.r.t. Γ'; but it is very costly to show that $C'^\top$ is satisfiable w.r.t. Γ'.
Indeed, we can instantiate $C^\top$ to more and more complex scenarios which make the speed-up of our framework arbitrarily large.
5. The freedom to use arbitrary search heuristics increases the application potential of our framework in comparison with the framework based on DFS by Donini and Massacci. To see this, reconsider the example above. It is easy to see that BFS runs faster than DFS for the example because $D^\top$ is satisfiable. A heuristic search may perform even better than BFS.

8 Conclusions

We have shown that global caching can indeed be formalised in the description of tableaux algorithms in a sound manner to give an EXPTIME algorithm for checking satisfiability w.r.t. a TBox in $\mathcal{ALC}$. Further work is required to see if our framework can also formalise other caching methods.

Global caching severely prunes the search space by avoiding multiple expansions of the same node, by enabling greater propagation of `sat` and `unsat` via subset-checking, and hence allowing more cutoffs to be done. Furthermore, various search strategies can be applied together with global caching.

We have developed an advanced EXPTIME decision procedure for checking satisfiability of a concept w.r.t. a TBox in $\mathcal{ALC}$. Our framework not only uses global caching and allows advanced search strategies but also exploits well-known optimisation techniques. In particular, all essential techniques and optimisations

proposed by Donini and Massacci [3] are incorporated in our decision procedure. The combination of global caching, propagation, and cutoffs allows our framework to cut down the search space much more than the framework by Donini and Massacci [3]. The freedom to use arbitrary search heuristics also increases the application potential of our framework.

Our method extends easily to tableau calculi for many other logics [6]. It can also be extended for checking consistency of an ABox w.r.t. a TBox in $\mathcal{ALC}$.

References

1. F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
2. Y. Ding and V. Haarslev. Tableau caching for description logics with inverse and transitive roles. In *Proc. DL-2006: International Workshop on Description Logics*, pages 143–149, 2006.
3. F. Donini and F. Massacci. EXPTIME tableaux for $\mathcal{ALC}$. *Artificial Intelligence*, 124:87–138, 2000.
4. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, 1995.
5. R. Goré. Tableau methods for modal and temporal logics. In D'Agostino et al, editor, *Handbook of Tableau Methods*, pages 297–396. Kluwer, 1999.
6. R. Goré and L.A. Nguyen. Tableaux for regular grammar logics of agents using automaton-modal formulae. Manuscript, 2006.
7. V. Haarslav and R. Möller. Consistency testing: The RACE experience. In R. Dyckhoff, editor, *TABLEAUX-2000: Proceedings of the International Conference on Theorem Proving with Analytic Tableaux and Related Methods*, volume LNCS 1847, pages 57–61. Springer, 2000.
8. I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics.* PhD thesis, University of Manchester, 1997.
9. I. Horrocks and P.F. Patel-Schneider. Optimizing description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.
10. L.A. Nguyen. Analytic tableau systems and interpolation for the modal logics KB, KDB, K5, KD5. *Studia Logica*, 69(1):41–57, 2001.
11. P.F. Patel-Schneider. DLP system description. In *Proceedings of Description Logics*, 1998.
12. W. Rautenberg. Modal tableau calculi and interpolation. *JPL*, 12:403–423, 1983.