Sound Global State Caching for ALC with Inverse Roles

Rajeev Goré[1] and Florian Widmann[2]

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

[2] Logic and Computation Group and NICTA*, The Australian National University
Canberra, ACT 0200, Australia, Florian.Widmann@anu.edu.au

Abstract. We give an optimal (EXPTIME), sound and complete tableau-based algorithm for deciding satisfiability with respect to a TBox in the logic $ALCI$ using global state caching. Global state caching guarantees optimality and termination without dynamic blocking, but in the presence of inverse roles, the proofs of soundness and completeness become significantly harder. We have implemented the algorithm in OCaml, and our initial comparison with FaCT++ indicates that it is a promising method for checking satisfiability with respect to a TBox.

1 Introduction

Description logics are classical multi-modal logics with applications in knowledge representation and reasoning [1]. Most applications can be reduced to the problem of deciding whether a given concept is satisfiable with respect to a finite set of concepts called a TBox. This problem is known to be EXPTIME-complete for the most basic expressive description logic ALC (normal multi-modal logic K_n), and known to be NEXPTIME-complete for more expressive logics like $SHOIQ$ [2]

The known optimal algorithms [1] for these decision problems are rarely used by practitioners because they are difficult to implement. Practitioners have instead implemented sub-optimal, typically tableau-based, algorithms which exhibit good average-case behaviour by utilising a vast array of optimisations like "back-jumping" and "lazy unfolding" to reduce the tableau search space [3].

Tableau calculi for description logics typically build and-trees of nodes where each node can be viewed as a set of concepts, and where the or-branching caused by disjunctions is conceptually handled by splitting one and-tree into several. An important optimisation is to "cache" previously seen tableau nodes when their status is either known to be, or can safely be assumed to be, satisfiable or unsatisfiable [4]. If the same node appears again then a (hopefully fast) "cache hit" gives us the answer without having to explore the node's subtree again. Caching

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

unsatisfiable nodes is sound across different and-trees, but caching satisfiable nodes can only be done within the same and-tree.

Goré and Nguyen have recently given an optimal, sound and complete algorithm for deciding ALC-satisfiability with respect to a TBox which globally caches all nodes, regardless of their status [5]. Their "global caching" algorithm never explores the same node twice, immediately giving an optimal and terminating procedure. The main difficulty is to prove that the method is sound and complete. Recent experimental work of Goré and Postniece [6] has shown that the method is competitive with the existing caching methods. Goré and Nguyen have extended their method to several extensions of ALC by using an analytic cut-rule [7]. It is doubtful if these extended methods will lead to practical implementations because blind use of analytic cut is considered "impractical" by practitioners in this field. It is therefore important to find direct methods for these extension which utilise global caching without recourse to analytic cut.

One such extension is $ALCI$ which extends ALC with inverse roles (converse modalities). Inverse roles cause problems because a concept like $\langle r\rangle([r^-]\varphi_1 \sqcup [r^-]\varphi_2)$ in a node w causes the creation of an r-successor node v containing $[r^-]\varphi_1 \sqcup [r^-]\varphi_2$, which then demands that the parent node w contains φ_1 if we expand the first disjunct in v but demands that the parent node w contains φ_2 if we expand the second disjunct. Assuming we take the first disjunct, if the node w does not contain φ_1 then it is "incompatible" with its r-child v, so, in principle, we have to add φ_1 to w and re-process the new φ_1 in w. But the re-processing may well make w contain a new concept $[r^-]\psi$ which we then have to pass back to the parent of w, and so on. Moreover, if choosing the first disjunct leads to an inconsistency, we have to undo all additions caused by the insertion of φ_1 into w so that we can explore the second disjunct in its original context. Conceptually, this can be done by creating a copy of the and-tree for each choice-point. Practical algorithms use many optimisations to minimise the copying required to maintain such choice-points as well as "dynamic equality blocking" to avoid infinite loops caused by TBoxes and inverse roles [8].

In summary, these methods can only globally cache unsatisfiable nodes, must cleverly manage choice-points and require blocking to be dynamic. Actual implementations are often sub-optimal in terms of their worst-case complexity. Polynomial reductions from $ALCI$ to ALC are also known [9].

Here, we give a sound, complete and cut-free method using "global state caching" for deciding satisfiability with respect to a TBox for $ALCI$. As opposed to global caching, which guarantees that the same node is never explored twice, our method globally caches only state nodes, but this is sufficient to guarantee worst-case optimality. This restriction can be safely relaxed to globally cache certain non-state nodes as well, and we return to this issue in Section 5. Since our underlying data structure is a cyclic graph, the main technical difficulty is to prove soundness and completeness (but the proofs are omitted for lack of space).

We present our method as pseudo code rather than as traditional tableau rules because the treatment of special nodes and the procedure `update` gives our algorithm a non-local flavour. Thus a set of traditional local tableau "completion

rules" would be cluttered by side-conditions to enforce the non-local aspects or would require a complicated strategy of rule applications.

Comparison of our OCaml implementation with FaCT++ shows that our method is a promising method for checking $ALCI$-satisfiability w.r.t. a TBox.

Section 2 contains the syntax and semantics of $ALCI$. Section 3 contains an overview of the algorithm, the detailed algorithm itself, and statements of the theorems on soundness, completeness and optimal complexity. Section 4 contains a fully worked example. Section 5 contains a brief description of the implementation and our initial experimental results, and concludes.

2 Syntax and Semantics

Definition 1. *Let $\mathcal{AR}$ and $\mathcal{AC}$ be disjoint and countably infinite sets of* role names *and* concept names*, respectively. The set $\mathcal{R}$ of all* role descriptions *and the set $\mathcal{C}$ of all* concept descriptions *are inductively defined as follows: $\mathcal{AR} \subseteq \mathcal{R}$; if r is in $\mathcal{R}$ then so is r^-; $\mathcal{AC} \subseteq \mathcal{C}$; if C and D are in $\mathcal{C}$ then so are $\neg C$, $C \sqcap D$, and $C \sqcup D$; if C is in $\mathcal{C}$ then so are $[r]C$ and $\langle r\rangle C$ for every $r \in \mathcal{R}$. A concept of the form $\langle r\rangle C$ and $[r]C$ is called a $\langle\cdot\rangle$- and $[\cdot]$-concept, respectively.*

Definition 2. *An* interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *is a pair where $\Delta^{\mathcal{I}}$ is a non-empty set, the* domain *of $\mathcal{I}$, and $\cdot^{\mathcal{I}}$ is an* interpretation function *mapping every $A \in \mathcal{AC}$ to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and every $r \in \mathcal{AR}$ to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. An interpretation function is inductively extended to concepts and roles as follows:*

$$
\begin{array}{lcl}
(\neg C)^{\mathcal{I}} & := & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & := & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} & := & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
([r]C)^{\mathcal{I}} & := & \{d \in \Delta^{\mathcal{I}} \mid \forall e.(d,e) \in r^{\mathcal{I}} \;\Rightarrow\; e \in C^{\mathcal{I}}\} \\
(\langle r\rangle C)^{\mathcal{I}} & := & \{d \in \Delta^{\mathcal{I}} \mid \exists e.(d,e) \in r^{\mathcal{I}} \;\&\; e \in C^{\mathcal{I}}\} \\
(r^-)^{\mathcal{I}} & := & \{(e,d) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (d,e) \in r^{\mathcal{I}}\} \ .
\end{array}
$$

Definition 3. *An interpretation $\mathcal{I}$* satisfies *a (not necessarily finite) set of concepts $X \subseteq \mathcal{C}$ iff $\bigcap_{C\in X} C^{\mathcal{I}} \neq \emptyset$, and* validates *$X$ iff $\bigcap_{C\in X} C^{\mathcal{I}} = \Delta^{\mathcal{I}}$. A* TBox *$\mathcal{T} \subseteq \mathcal{C}$ is a finite set of concepts. A set $X \subseteq \mathcal{C}$ is* satisfiable *with respect to $\mathcal{T}$ iff there exists an interpretation which validates $\mathcal{T}$ and satisfies X.*

We extend the definitions to single concepts by interpreting them as singleton sets. Clearly $\mathcal{I}$ validates C iff it does not satisfy $\neg C$. Traditionally, a TBox is defined to be a finite set of terminological axioms of the form $C \sqsubseteq D$, where C and D are concepts, but the two definitions are equivalent.

Definition 4. *For a role $r \in \mathcal{R}$ we define $r^\smile$ as s if r is of the form s^-, and as r^- otherwise. A concept $C \in \mathcal{C}$ is in* negation normal form *if $\neg$ appears only directly before concept names and if all roles appearing in C are in $\mathcal{AR} \cup \{r^- \mid r \in \mathcal{AR}\}$. It is well known that, in $ALCI$, every concept C has a logically equivalent concept $\mathrm{nnf}(C)$ which is in negation normal form. A TBox $\mathcal{T}$ is in negation normal form if all concepts in $\mathcal{T}$ are in negation normal form.*

3 Algorithm, Soundness, Completeness and Termination

Given a TBox $\mathcal{T}$ and a concept D, both in negation normal form, our method searches for an interpretation which validates $\mathcal{T}$ and satisfies D by building an and-or graph. We start with a high level description of our algorithm.

3.1 Overview of the Algorithm

Recall that the standard strategy for rule applications in tableau algorithms is to apply the rules for decomposing $\sqcap$ and $\sqcup$ repeatedly until they are no longer applicable, giving a "saturated" node which contains only atoms, negated atoms, $\langle\cdot\rangle$-formulae and $[\cdot]$-formulae. Let us call such a "saturated" node a *state* and call the other nodes *prestates*. Thus the only rule applicable to a state x is the $\langle\cdot\rangle$-rule which creates a node containing $\{C\} \cup \{D \mid [r]D \in x\}$ for each $\langle r\rangle C \in x$. The standard strategy will now saturate any such child to obtain a state y, then apply the $\langle\cdot\rangle$-rule to y, and so on, until we find a contradiction, or find a repeated node, or find a state which contains no $\langle\cdot\rangle$-formulae. Let us call x the parent state of y since all intervening nodes are not states.

When inverse roles are present, we require that $\{E \mid [r^\smile]E \in y\} \subseteq x$, since y is then compatible with being an r-successor of x in the putative interpretation under construction. If some $[r^\smile]E \in y$ has $E \notin x$ then x is "too small", and must be enlarged into an alternative node x^+ by adding all such E. If any such E is a complex formula then the alternative node x^+ is not "saturated", and hence not a state. So we must saturate it using the $\sqcap/\sqcup$-rules until we reach a state. That is, a state x may conceptually be "replaced" by an alternative prestate x^+ which is an enlargement of x, and which may have to be saturated further in order to reach a state.

Our algorithm handles these "alternatives" by introducing a new type of node called a *special node*, introducing a new type of status called `toosmall`, allowing states to contain a field alt for storing these alternatives, and ensuring that a state always has a special node as its parent. When we need to replace a state x by its alternatives, the special node above x extracts these alternatives from the alt_x field and creates the required alternative nodes as explained next.

Referring to Fig. 1, suppose state x has an r-successor prestate ps_0, and further saturation of ps_0 leads to prestate ps_k, and an application of an $\sqcap/\sqcup$-rule to p_k will give a state y. Instead of directly creating y, we create a special node z which carries the same set of formulae as would y, and make z a child of ps_k. We now check whether z is compatible with its parent state x by checking whether $\{E \mid [r^\smile]E \in z\} \subseteq x$. If z is not compatible then we mark z as `toosmall`, and add $\{E \mid [r^\smile]E \in z\} \setminus x$ to the set of alternative sets contained in alt_x, without creating y, as shown in Fig. 1(a). If z is compatible with x, we create a state y if it does not already exist, and make the new/old y a child of z, as in Fig. 1(b).

Suppose that y is compatible with x and that either y is already `toosmall` or becomes so later because of some descendant state w of y. In either case, the attribute alt_y then contains a number of sets $y_1, y_2, \ldots, y_n$ (say), and the `toosmall` status of y is propagated to the special node z. In response, z will

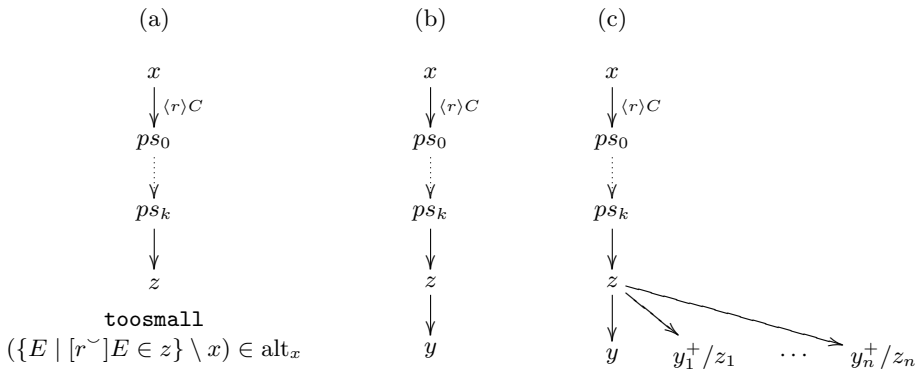

Fig. 1. The use of special node z to handle in/compatibility between states x and y. Scenario (a) occurs when x and y are incompatible. Scenario (b) occurs when x and y are compatible. Scenario (c) occurs when x and y are compatible, but y is `toosmall`.

create the alternatives $y_1^+, y_2^+, \ldots, y_n^+$ for y with $y_i^+ := y \cup y_i$. If y_i^+ is a state then our algorithm will create a special node z_i below z, and if z_i is compatible with x then y_i^+ will be created or retrieved and will become the child of z_i as in (b) else y_i^+ will not be created and z_i will be marked as `toosmall` as in (a). If y_i^+ is not a state then it will be created as a direct prestate child of z. Figure 1(c) captures this by using y_i^+/z_i to stand for either y_i^+ or z_i. Each of these new non-special nodes will eventually be expanded by our algorithm but now the "lapsed" special node z will be treated as a $\sqcup$-node.

3.2 The Algorithm

Our algorithm builds a graph G consisting of nodes and directed edges. We first explain the structure of G in more detail. In the rest of the paper, we use the notation $\mathscr{P}(Y)$ for the power set of Y and $x \in Y^?$ ($x \subseteq Y^?$) to indicate that x is either an element (a subset) of Y or undefined ("$\perp$").

Definition 5. *Each node $x \in G$ has six attributes belonging to it: $\Gamma_x \subseteq \mathcal{C}$, $\mathrm{alt}_x \subseteq \mathscr{P}(\mathcal{C})^?$, $\mathrm{pst}_x \in G^?$, $\mathrm{prl}_x \in \mathcal{R}^?$, $\mathrm{spl}_x \in (G \cup \{\mathrm{lsn}\})^?$, and $\mathrm{sts}_x \in \mathfrak{S}^?$ where $\mathfrak{S} := \{\mathtt{unsat}, \mathtt{sat}, \mathtt{toosmall}, \mathtt{open}\}$ and* lsn *is just a constant.*

Some attributes of a node $x \in G$ may be undefined initially. Once an attribute is defined in x, however, it will never become undefined again.

The attribute Γ_x of a node $x \in G$ contains the concepts that are assigned to x. It is set at the creation of x and is not changed afterwards. There may exist several nodes having the same set of concepts.

The attribute alt_x is defined (at the creation of x) if and only if x is a state. If defined it contains a set of sets of concepts. Each set of concepts can be seen as a way to extend Γ_x to form an *alternative node* for x. The set alt_x is initially empty but can grow as the algorithm proceeds.

The attributes pst_x and prl_x are defined for all nodes excepts states. They are set at the creation of x and are never changed. The attribute pst_x identifies the, as we will ensure, unique ancestor $p \in G$ of x such that p is a state and there is no other state between p and x in G. We call p the *parent state* of x. The creation of the child of p which lies between p and x was caused by a $\langle\cdot\rangle$-concept $\langle r\rangle C$ in Γ_p. The role r which we call the *parent role* of x is stored in prl_x.

The attribute spl_x is defined if and only if x is a *special node*. If defined, its value is either lsn or is the state that is the child of the special node. As explained in the overview, the special nature of special nodes can eventually lapse, after which they are treated as $\sqcup$-nodes: thus lsn stands for "lapsed special node".

The last attribute sts_x describes the *status* of x. It is initially undefined but becomes defined eventually during the algorithm. Its value may be modified several times. The value `unsat` indicates that the node is unsatisfiable. The value `sat` indicates that the node is satisfiable. The value `toosmall` indicates that the node is "useless" for building an interpretation because it does not contain some concepts that are required by inverse roles and $[\cdot]$-concepts. Hence, it is treated similarly to `unsat`. Finally, the value `open` indicates that it is currently not known whether or not the node is satisfiable.

Definition 6. *Let $x \in G$ be a node. We call x* unsat *iff it has $\text{sts}_x =$* `unsat`*,* sat *iff it has $\text{sts}_x =$* `sat`*,* too small *iff it has $\text{sts}_x =$* `toosmall`*, and* open *iff it has $\text{sts}_x =$* `open`*. A* path *π in G is a finite or infinite sequence $x_0, x_1, x_2, \ldots$ of nodes in G such that x_{i+1} is a child of x_i for all x_i which have a successor in π.*

Next we comment on all procedures given in pseudocode.

Procedure `is-sat`$(D, \mathcal{T})$ is the main procedure which determines whether a concept $D \in \mathcal{C}$ is satisfiable w.r.t. a TBox $\mathcal{T}$, both in negation normal form. It first initialises G to the empty graph. We consider G as a global variable, so the other procedures have access to it. Then we create a dummy state which we call the *root node* and insert it in G. If we create a node, all attributes which are not explicitly given are undefined. The root node is inserted for technical reasons so that each node that is not a state has a parent state.

While there exists a node $x \in G$ whose status is undefined, we expand x as explained next. Since special nodes and nodes which contain a contradiction get their status in the invocation of `insert-node` which creates them, the following classifications do not contain such nodes.

If Γ_x contains a $\sqcap$-concept C whose immediate subconcepts are not in Γ_x, we call x a $\sqcap$-*node*, so we create a new set Γ' by adding C_1 and C_2 to Γ_x. Note $\Gamma' \supsetneq \Gamma_x$. We then invoke `insert-node` which creates a node with Γ' assigned to it and adds an edge from x to that node. Note that pst_x and prl_x are defined as x is not a state. After that we determine and set the status of x.

If x is not a $\sqcap$-node and Γ_x contains a $\sqcup$-concept C none of whose immediate subconcepts is in Γ_x, we call x a $\sqcup$-*node*. For each decomposition C_i we do the following: We create a new set Γ_i by adding C_i to Γ_x. Thus Γ_i is a strict superset of Γ_x. Then we invoke `insert-node` which creates a node with Γ_i assigned to it and add an edge from x to that node. Note that pst_x and prl_x must be defined as x is not a state. Finally, we determine and set the status of x.

Procedure `is-sat`$(D, \mathcal{T})$ for testing whether D is satisfiable w.r.t. $\mathcal{T}$

Input: a concept $D \in \mathcal{C}$ and a TBox $\mathcal{T}$, both in negation normal form
Output: true iff D is satisfiable w.r.t. $\mathcal{T}$

$G :=$ a new empty graph
let $s \in \mathcal{AR}$ be a dummy role name which does not occur in D or $\mathcal{T}$
create new node rt with $\Gamma_{\text{rt}} := \{\langle s \rangle D\}$ and $\text{alt}_{\text{rt}} := \emptyset$
insert rt in G
while $\exists x \in G.\, \text{sts}_x = \bot$ **do** (* x is not expanded yet *)
  if $\exists C \in \Gamma_x.\, C = C_1 \sqcap C_2$ & $\{C_1, C_2\} \not\subseteq \Gamma_x$ **then** (* x is a $\sqcap$-node *)
    $\Gamma' := \Gamma_x \cup \{C_1, C_2\}$
    `insert-node`$(\Gamma', x, \text{pst}_x, \text{prl}_x)$
    $\text{sts}_x :=$ `det-sts-or`(x)
  else if $\exists C \in \Gamma_x.\, C = C_1 \sqcup C_2$ & $\{C_1, C_2\} \cap \Gamma_x = \emptyset$ **then** (* x is a $\sqcup$-node *)
    for $i \longleftarrow 1$ **to** 2 **do**
      $\Gamma_i := \Gamma_x \cup \{C_i\}$
      `insert-node`$(\Gamma_i, x, \text{pst}_x, \text{prl}_x)$
    $\text{sts}_x :=$ `det-sts-or`(x)
  else (* x is a state *)
    let $\langle r_1 \rangle C_1, \cdots, \langle r_k \rangle C_k$ be all of the $\langle \cdot \rangle$-concepts in Γ_x
    for $i \longleftarrow 1$ **to** k **do**
      $\Gamma_i := \{C_i\} \cup \{E \mid [r_i]E \in \Gamma_x\} \cup \mathcal{T}$
      `insert-node`(Γ_i, x, x, r_i)
    $\text{sts}_x :=$ `det-sts-state`(x)
  let $y_1, \ldots, y_k$ be all the parents of x
  for $i \longleftarrow 1$ **to** k **do** `update`(y_i)
return $\text{sts}_{\text{rt}} \in \{\texttt{sat}, \texttt{open}\}$

If x is neither a $\sqcap$-node nor $\sqcup$-node, it must be fully saturated and hence a state. For each $\langle \cdot \rangle$-concept $\langle r_i \rangle C_i$ we do the following: We create a new set Γ_i containing C_i, all concepts in $\mathcal{T}$, and all concepts E such that $[r_i]E \in \Gamma_x$. Then we invoke `insert-node` which creates a node with Γ_i assigned to it and adds an edge from x to that node. We call this node the *successor* of $\langle r_i \rangle C_i$. Finally, we determine and set the status of x.

At the end of the while loop, we update the status of all parent nodes of x.

The procedure stops if all nodes in G have a defined status. It returns "satisfiable" iff the root node is either sat or open.

Procedure `insert-node`(Γ, x, p, r) nominally creates a node with Γ assigned to it and inserts an edge from x to that node. Due to the issues with inverse roles, however, the details are more complicated. We start by explaining the arguments of `insert-node` in more detail.

The node $x \in G$ invokes `insert-node` because it requires the existence of a node which has $\Gamma \subseteq \mathcal{C}$ assigned to it. The arguments $p \in G$ and $r \in \mathcal{R}$ are the parent state and the parent role of the new node, respectively. By inspecting the three invocations of `insert-node` in `is-sat`, it should not be hard to see that p and r are given the "right" values: if x is a $\sqcap$- or $\sqcup$-node then pst_x and prl_x

Procedure `insert-node`(Γ, x, p, r) for inserting a node into the graph

Input: a set $\Gamma \subseteq \mathcal{C}$ containing the concepts of the new node; a node $x \in G$ which invoked this procedure; the parent state $p \in G$ (in particular $\text{alt}_p \neq \bot$); and the parent role $r \in \mathcal{R}$

if $\exists C \in \mathcal{C}.\, \{C, \text{nnf}(\neg C)\} \subseteq \Gamma$ **then** (* contradiction found *)
  create new node y with $\Gamma_y := \Gamma$, $\text{pst}_y := p$, $\text{prl}_y := r$, and $\text{sts}_y :=$ `unsat`
  insert y and edge (x, y) in G
else if $\exists C \in \Gamma.\, C = C_1 \sqcap C_2$ *or* $C = C_1 \sqcup C_2$ **then**
  create new ($\sqcap$- or $\sqcup$-)node y with $\Gamma_y := \Gamma$, $\text{pst}_y := p$, and $\text{prl}_y := r$
  insert y and edge (x, y) in G
else (* Γ is fully saturated *)
  create new (special) node z with $\Gamma_z := \Gamma$, $\text{pst}_z := p$, and $\text{prl}_z := r$
  $\Gamma_{\text{alt}} := \{C \mid [r^\smile]C \in \Gamma\} \setminus \Gamma_p$
  if $\Gamma_{\text{alt}} = \emptyset$ **then** (* Γ is compatible with p *)
    if $\exists y \in G.\, \text{alt}_y \neq \bot$ & $\Gamma_y = \Gamma$ **then** (* state already exists in G *)
      insert edge (z, y) in G
      $\text{spl}_z := y$
    else (* state is not in G yet *)
      create new (state) node y with $\Gamma_y := \Gamma$ and $\text{alt}_y := \emptyset$
      insert y and edge (z, y) in G
      $\text{spl}_z := y$
    $\text{sts}_z :=$ `det-sts-spl`(z)
  else (* Γ is not compatible with p *)
    if $\text{sts}_p \in \{\bot,$ `open`$\}$ **then** $\text{alt}_p := \text{alt}_p \cup \{\Gamma_{\text{alt}}\}$
    $\text{spl}_z := \text{lsn}$
    $\text{sts}_z =$ `toosmall`
  insert z and edge (x, z) in G

are just passed on; if x is a state then p is x itself and r is the role from the $\langle\cdot\rangle$-concept in x which requires the existence of the said node.

If Γ contains an immediate contradiction, we create a new node y which immediately becomes unsat and insert an edge from x to y. For the other cases, we assume implicitly that Γ does not contain an immediate contradiction.

If Γ contains a $\sqcap$- or $\sqcup$-concept which still has to be decomposed, we create a new $\sqcap$- or $\sqcup$-node y and insert an edge from x to y. Note that we create a new node even if there already exists a node in G which has Γ assigned to it; otherwise the parent state and the parent role of a node would not be unique.

If Γ is fully saturated, things become more interesting. In this case we first create a *special node* z, not because of the usual tableau rules, but to handle the "special" issue arising from inverse roles, as explained in the overview. Like $\sqcap$- and $\sqcup$-nodes, special nodes have a unique parent state and a unique parent role.

Next we determine the set Γ_{alt} of all concepts C such that $[r^\smile]C$ is in Γ but C is *not* in p. If there is no such concept we say that Γ is *compatible* with p. Note that incompatibilities can only arise because of inverse roles.

If Γ is compatible with p, we check whether some state y in G has Γ assigned to it. If such a state y already exists in G, we insert an edge from z to y; otherwise we create a new state y first and then insert an edge from z to y. Consequently, there is at most one state in G for every set of concepts explaining the term "global state caching" of the title. In both cases we flag z as special by defining $\mathrm{spl}_z := y$. Then we determine and set the status of z.

If Γ is not compatible with p, we cannot connect p to a state with Γ assigned to it as explained in the overview. Hence the intermediary z flags this by becoming too small. A node which is too small is treated similarly to an unsat node as both are useless for building an interpretation. That does not, however, mean that p is unsatisfiable; maybe it is just missing some concepts. We cannot extend Γ_p directly as this may have side-effects elsewhere; but to give p a clue as to what went wrong, we add Γ_{alt} to alt_p if p is still open or has an undefined status. The meaning is that if we create an alternative node for p by adding the concepts in Γ_{alt}, we might be more successful in building an interpretation. We flag z as special by defining $\mathrm{spl}_z := \mathrm{lsn}$, which in this case is just a dummy value which is not needed later.

Finally we put an edge from x to the special node z.

Note that if a special node z requires a state y which already exists in G *and is already known to be too small* then we must insert the alternative extensions of y immediately via `det-sts-spl`(z) to determine the status of z. Since such an alternative may itself be a special node, `insert-node` may recurse via `det-sts-spl`. This is why special nodes are the only nodes which get their status in the procedure `insert-node`, rather than in the outer procedure `is-sat`.

Procedure `det-sts-spl`(x) computes the status of a special node $x \in G$. By definition of a special node, the attribute spl_x is defined.

If spl_x is a node $y \in G$ then it must be a state and we do the following: If y is unsat or sat, the status is just propagated to x. If y is open or its status is not defined then x becomes open. The interesting case arises if y is too small, meaning that y is unsuitable for building an interpretation. Its attribute alt_y contains information on how to extend Γ_y in order to potentially fix the problem. So we do the following for every set $\Gamma \in \mathrm{alt}_y$: We create a new set by adding the concepts in Γ to Γ_y and then insert this alternative node of y in G and add an edge from x to it. It is easy to see that the new set is a strict superset of Γ_y. The alternative node does not have to be a state since it may contain $\sqcap$- and $\sqcup$-concepts, so it may require further saturation. Moreover, it is possible that it contains a contradiction or that its saturation leads to sets of concepts which are not compatible with the parent state of x. Hence we have to use `insert-node` to insert the alternative nodes. If one of the alternative nodes turns out to be "useful" for building an interpretation, it "replaces" the discarded y. Hence the special nature of x has "lapsed" and it behaves like a $\sqcup$-node from now on, so we set spl_z to lsn and determine its status by invoking `det-sts-or`.

If $\mathrm{spl}_x = \mathrm{lsn}$ then we know that x has already created the alternative nodes of the corresponding state and that it should behave like a $\sqcup$-node. Hence we invoke `det-sts-or` and pass on the result.

Procedure `det-sts-or`(x) computes the status of a $\sqcap$- or $\sqcup$-node $x \in G$. Note that for this task, a $\sqcap$-node can be seen as a $\sqcup$-node with exactly one child. If some child is sat then x is also sat. Otherwise, if there is at least one child that is open or has an undefined status, then x is still open. If none of the two cases apply, all children are unsat or too small. If there exists a child which is too small then x is too small. If all children are unsat then so is x. Note that, apart from the "extra" value `toosmall`, which is conceptually treated as `unsat`, the procedure captures the standard behaviour for tableaux.
Procedure `det-sts-state`(x) computes the status of a state $x \in G$. If one of the children of x is unsat then x must also be unsat. If some child y of x is too small then x must also be too small as y cannot be used for building an interpretation. If none of the children is unsat or too small, but some child is open or has an undefined status, then x is still open. If none of the other cases apply, all children must be sat, so x must be sat too. Again, apart from the "extra" value `toosmall`, which is conceptually treated as `unsat`, the procedure captures the standard behaviour for tableaux.
Procedure `update`(x) propagates status changes through G. It takes a node $x \in G$ and recomputes its status if the node is still open. If this new status differs from its old status stored in sts_x, it updates sts_x and invokes `update` recursively on all nodes whose status might be affected by this change. Note that a node that is open is either a special node or a $\sqcap/\sqcup$-node or a state.

We now list some facts which are useful in understanding the algorithm and which are needed in the (omitted) proofs of Theorems 8-10. These facts can be verified by inspection of the procedures in a rather straightforward way.

Proposition 7. *Let $x, y, z \in G$ be nodes.*

(i) if `update`(x) is invoked then the status of x is defined;
(ii) if x and y are states with $\Gamma_x = \Gamma_y$ then $x = y$;
(iii) if x is a state then its parents are exactly the special nodes y with $\Gamma_y = \Gamma_x$;
(iv) if x is a state and $\Gamma \in \mathrm{alt}_x$ then $\Gamma \neq \emptyset$ and $\Gamma \cap \Gamma_x = \emptyset$;
(v) if x is a special node, it has at least one child iff $\{C \mid [\mathrm{prl}_x^\smile]C \in \Gamma_x\} \subseteq \Gamma_{\mathrm{pst}_x}$. In this case, one of its children is the state y with $\Gamma_y = \Gamma_x$.
(vi) if y is a child of x and neither of them are states then $\mathrm{pst}_x = \mathrm{pst}_y$ and $\mathrm{pst}_x = \mathrm{pst}_y$ and $\Gamma_x \subsetneqq \Gamma_y$;

3.3 Soundness, Completeness, and Complexity

Let $D \in \mathcal{C}$ be a concept and $\mathcal{T}$ a TBox, both in negation normal form. Furthermore let G be the final graph with root node rt that was created by invoking `is-sat`$(D, \mathcal{T})$. Note that all nodes in G have a defined status. We define the size of a concept $C \in \mathcal{C}$ as the number of symbols in C. Let n be the sum of the sizes of all concepts in $X := \{D\} \cup \mathcal{T}$.

Theorem 8. *The algorithm terminates and runs in* EXPTIME *in n.*

Theorem 9. *If root node* rt *is sat or open then D is satisfiable w.r.t. $\mathcal{T}$.*

Theorem 10. *If* rt *is unsat or too small then D is not satisfiable w.r.t. $\mathcal{T}$.*

Procedure `det-sts-spl`(x) for determining the status of a special node

Input: a special node $x \in G$ (*i.e.* $\mathrm{spl}_x \neq \bot$) with at least one child in G
Output: the new status of x

if $\mathrm{spl}_x \neq \mathrm{lsn}$ **then** (∗ spl_x is a child of x ∗)
  $y := \mathrm{spl}_x$
  if $\mathrm{sts}_y =$ `unsat` **then return** `unsat`
  else if $\mathrm{sts}_y =$ `sat` **then return** `sat`
  else if $\mathrm{sts}_y \in \{\bot, \texttt{open}\}$ **then return** `open`
  else (∗ y must be too small so create its alternatives ∗)
    foreach $\Gamma \in \mathrm{alt}_y$ **do** `insert-node`$(\Gamma_y \cup \Gamma, x, \mathrm{pst}_x, \mathrm{prl}_x)$
    $\mathrm{spl}_x := \mathrm{lsn}$
    return `det-sts-or`(x)
else return `det-sts-or`(x)

Procedure `det-sts-or`(x) for determining the status of a $\sqcap$- or $\sqcup$-node

Input: a $\sqcap$- or $\sqcup$-node $x \in G$
Output: the new status of x

let $y_1, \ldots, y_k \in G$ be all the children of x
if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `sat` **then return** `sat`
else if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} \in \{\bot, \texttt{open}\}$ **then return** `open`
else if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `toosmall` **then return** `toosmall`
else return `unsat`; (∗ all children are unsatisfiable ∗)

Procedure `det-sts-state`(x) for determining the status of a state

Input: a state $x \in G$
Output: the new status of x

let $y_1, \ldots, y_k \in G$ be all the children of x
if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `unsat` **then return** `unsat`
else if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `toosmall` **then return** `toosmall`
else if $\exists i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} \in \{\bot, \texttt{open}\}$ **then return** `open`
else return `sat`; (∗ all children are satisfiable ∗)

Procedure `update`(x) for propagating the status of nodes

Input: a node $x \in G$ that has a defined status

if $\mathrm{sts}_x =$ `open` **then** (∗ otherwise the status cannot change ∗)
  sts := **if** $\mathrm{spl}_x \neq \bot$ **then** `det-sts-spl`(x)
    else if $\mathrm{alt}_x \neq \bot$ **then** `det-sts-state`(x) **else** `det-sts-or`(x)
  if $\mathrm{sts}_x \neq$ sts **then**
    $\mathrm{sts}_x :=$ sts
    let $z_1, \ldots, z_k$ be all the parents of x
    for $i \longleftarrow 1$ **to** k **do** `update`(z_i)

4 A Fully Worked Example

Given the TBox $\mathcal{T} := \{\neg A\}$, the concept $C := \neg A \sqcap \langle r\rangle\langle r\rangle[r^-][r^-](A \sqcup B)$ is satisfiable w.r.t. $\mathcal{T}$, so our algorithm should say so. Note that $\mathcal{T}$ does not play a prominent role in this example and is only included for demonstration purposes.

Figure 2 and 3 show the graph just before processing node (17). The root is node (1) and Fig. 3 shows the subgraph rooted at node (14). The nodes are labelled in the order in which they are created. The bottom left corner of a node x contains sts_x. The bottom right corner contains alt_x if x is a state, and pst_x and prl_x if x is a $\sqcap/\sqcup$- or special node. If x is a special node then spl_x is given in the middle of the bottom line. We have not marked the principal concept which is decomposed in a node, since it is obvious. Arrows leaving states are labelled with a role r if an $\langle r\rangle$-concept caused the creation of the child. When a special node creates the alternative nodes, the edges that are created during this process are labelled with alt. The labelling of arrows leaving $\sqcap$- and $\sqcup$-nodes is obvious.

The creation and processing of nodes (1) and (2) is straightforward. As node (3) is compatible with state (1), state (4) is created and inserted as a child of node (3). Special node (5) and state (6) are handled similarly. Special node (7), however, is incompatible with state (6). Hence node (7) immediately becomes too small, and the set $\{[r^-](A \sqcup B)\}$ is added to the set of alternatives of state (6). Then state (6) becomes too small via `det-sts-state`. Its status change is propagated to special node (5) which now creates the alternative node (8) via `det-sts-spl`. Since node (8) is incompatible with state (4), it immediately becomes too small. Moreover the set $\{A \sqcup B\}$ is added to the set of alternatives of state (4). Because all children of node (5) are too small, it becomes too small as well via `det-sts-spl` and `det-sts-or`. This result is propagated to node (4) and (3). Similar to node (5), node (3) creates the alternative node (9).

The first child of node (9) contains a contradiction and becomes unsat immediately. Special node (11) and state (12) are handled similarly to node (3) and (4). Special node (13) is compatible with state (12) which contains no $[r^-]$-concepts. The state it requires is already in the graph as node (6), but it is too small, so node (13) immediately creates the alternative node (14). Nodes (8) and (14) are similar to each other but unlike node (8), node (14) *is* compatible with its parent state (12). So node (15) is created and inserted as its child. Node (16) is similar to node (7), but unlike node (7), it is compatible with its parent state (15). So state (17) is created and inserted as a child of node (16).

This is the moment captured by Fig. 2 and 3. Only node (17) remains unprocessed. Since it lacks $\langle\cdot\rangle$-concepts, it becomes sat via `det-sts-state` immediately. The result is propagated to all open nodes, including the root, which becomes sat. The algorithm returns that C is satisfiable w.r.t. $\mathcal{T}$.

5 Implementation, Experimental Results and Conclusion

Our algorithm is fairly detailed, so a naive implementation should be straightforward. But a well-engineered and sophisticated implementation requires more work so we address some aspects that do not show up in the algorithm.

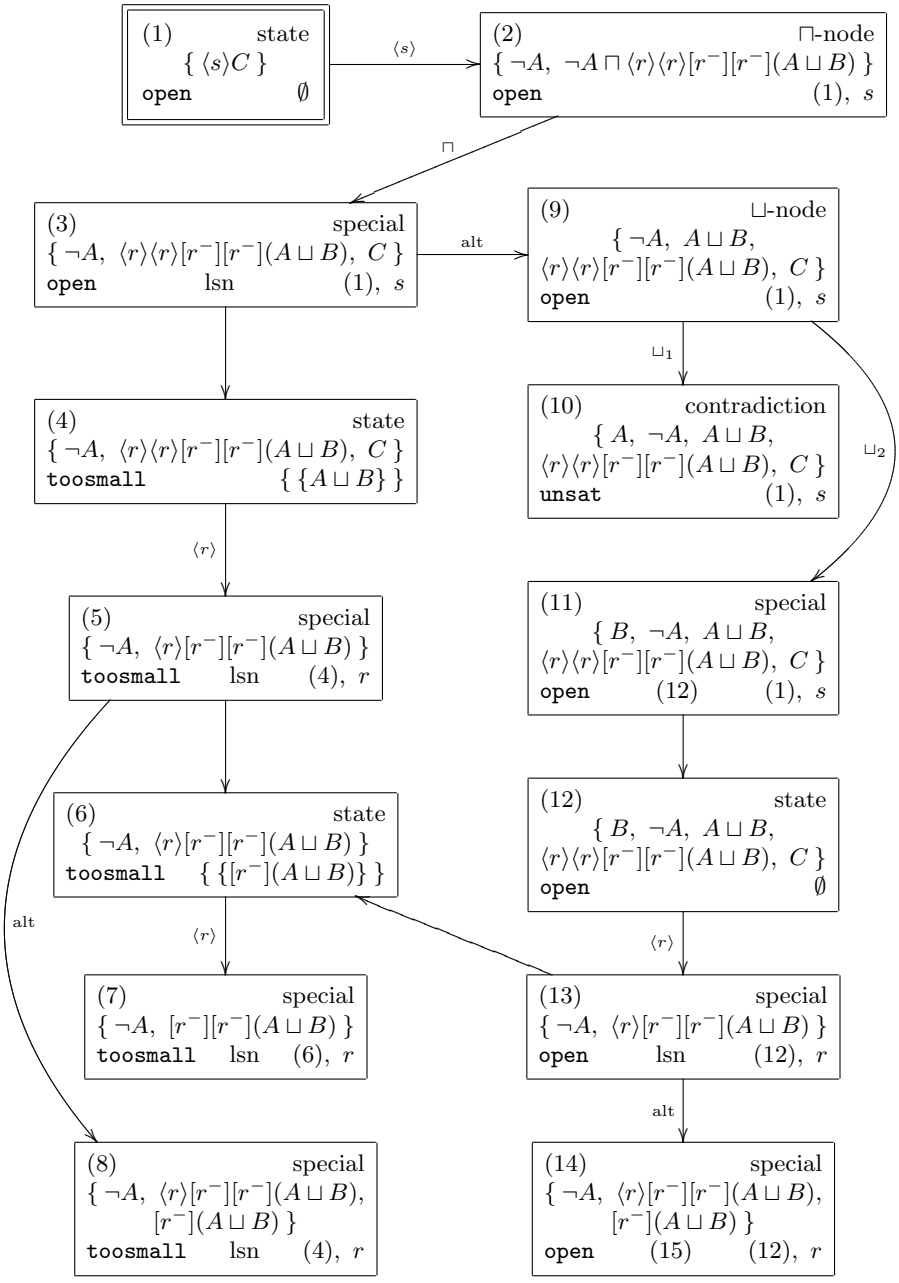

Fig. 2. An example: The graph just before processing node (17) (*cf.* Fig. 3)

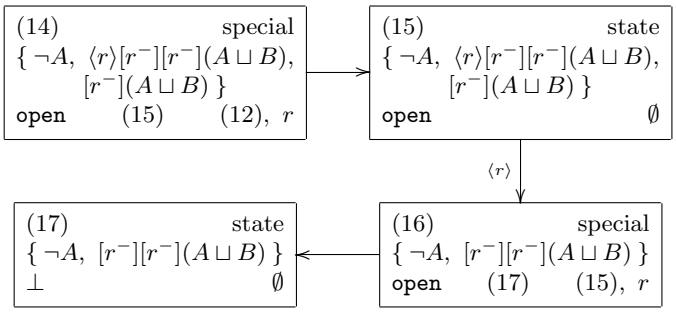

Fig. 3. An example: The graph just before processing node (17) (cont.)

There are some obvious optimisations. For example, the procedure can stop as soon as the root becomes sat, unsat or too small since we know that its status will not change again. Also we do not need to expand a node which has only parents who are all already sat, unsat or too small. Of course, if a new parent which is still open is linked to the node we might have to expand it after all.

We are free to choose any node to expand, but some nodes are obviously more "promising" than others. For example, as long as a $\sqcup$-node has an open child, it is not necessary to expand its other children. They require consideration only if this open child becomes unsat or too small. This can be implemented efficiently by having a decentralised queue which is distributed in the nodes.

One major issue is that our algorithm as given in this paper does the saturation phase for every state independently. That is, if two states differ only slightly, say one state has an additional concept name $A \in \mathcal{AC}$, the "same" saturation phase is done twice which seems unnecessary. However, we cannot unconditionally use the same saturation tree for both states; for example a concept $[r^-]\neg A$ in some node of the saturation tree might affect one state but not the other. Having said that, some improvements are possible. For example, we can cache and reuse unsat nodes in the saturation phase, but this is not possible for the other nodes. We can, however, reuse the same saturation tree for several states. That is, if we are about to recreate (parts of) a saturation tree, we do not create new nodes but allow the existing nodes to have several status flags, one for each state in whose saturation tree they appear. Caching unsat nodes in saturation trees is easy to implement, but the second improvement makes the algorithm significantly more complicated.

We have a fairly sophisticated implementation of our algorithm in OCaml. It uses some of the important optimisations like back-jumping, semantic branching, and local simplification [3], but does not implement many "at a world" optimisations and heuristics from SAT like reordering subconcepts of a $\sqcup$-concept.

We compared our implementation with FaCT++. Since FaCT++ handles $SROIQ$ which is much more expressive than $ALCI$, it is possible that this additional complexity slows FaCT++ down on $ALCI$. On the other hand, FaCT++ is highly optimised. Our intention was not to make a thorough system compar-

ison, but only to check whether our method is feasible. For the same reason, we do not give actual runtimes but just explain the qualitative results.

First, good benchmarks for $ALCI$ do not seem to exist, and most knowledge bases use additional features like number restrictions, so we found it hard to compile a good set of test problems. We therefore used the `T98-sat` and `T98-kb` problems from the DL98 benchmarks (http://dl.kr.org/dl98/comparison). The only problems where FaCT++ did significantly better were the "pigeon hole" problems, which is not surprising since our prover does not include any SAT optimisations. In five problems, our prover showed significantly better results. For the remaining thirty problems, both provers were on par. We see these results only as a sanity check since these problems do not use inverse roles.

Second, we tested the provers on randomly generated $ALCI$ concepts with varying sizes of $ALCI$ TBoxes. For empty TBoxes, the concepts were not very difficult as both provers showed a linear increase in the size of the concepts and gave similar runtimes. As the TBoxes grew bigger, the performance of both provers started to degrade in a similar manner. Although randomly generated concepts are not good benchmarks, they sometimes serve a second purpose. We found that FaCT++ returned some incorrect results as confirmed by its authors.

Our method for $ALCI$ is definitely promising and should be extended to more expressive logics to test whether it remains feasible. The extension to role hierarchies and transitive roles should not present difficulties, but the extension to include nominals and qualified number restrictions is not obvious to us.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge, England (2003)
2. Horrocks, I., Sattler, U.: A tableau decision procedure for SHOIQ. Journal of Automated Reasoning **39**(3) (2007) 249–276
3. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation **9**(3) (1999) 267–293
4. Donini, F.M., Massacci, F.: EXPTIME tableaux for ALC. Artificial Intelligence **124**(1) (2000) 87–138
5. Goré, R., Nguyen, L.A.: EXPTIME tableaux for ALC using sound global caching. In: Proc. DL-07. Volume 250 of CEUR Workshop., CEUR-WS.org (2007)
6. Goré, R., Postniece, L.: An experimental evaluation of global caching for ALC (system description). In: Proc. IJCAR-08. Volume 5195 of LNCS., Springer (2008) 299–305
7. Goré, R., Nguyen, L.A.: EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In: Proc. TABLEAUX-07. Volume 4548 of LNCS., Springer (2007) 133–148
8. Blackburn, P., Wolter, F., van Benthem, J., eds.: Handbook of Modal Logic. Elsevier (2006)
9. Ding, Y., Haarslev, V., Wu, J.: A new mapping from ALCI to ALC. In: Proc. DL-07. Volume 250 of CEUR Workshop Proceedings., CEUR-WS.org (2007)