Global Caching, Inverse Roles and Fixpoint Logics

Rajeev Goré

Logic and Computation Group
College of Engineering and Computer Science
The Australian National University
Canberra ACT 0200, Australia

Abstract. I will begin by explaining an optimal tableau-based algorithm for checking ALC-satisfiability which uses "global caching" and which appears to work well in practice. The algorithm settles a conjecture that "global caching can lead to optimality". I will then explain how "global caching" can be extended to "global state caching" for inverse roles, thereby extending the result to ALCI-satisfiability, and converse roles in general. Finally, I will explain how "global caching" can be used to give optimal "on the fly" tableau decision procedures for some fixpoint logics. Finally, some open questions. The talk is intended to be expository so it will be at a fairly high level.

1 Introduction

This overview is based on joint work with Linh Anh Nguyen [8, 9], Linda Postniece [10] and Florian Widmann [12, 11].

The tableau method is a very general method for automated reasoning and has been widely applied for modal logics [7] and description logics [2]. Tableau methods usually come in two flavours as we explain shortly. Both methods build a rooted tree with some leaves duplicating ancestors, thereby giving cycles. Because the same node may be explored on multiple branches, tableau algorithms are typically suboptimal w.r.t. the known theoretical bounds for many logics. For example, the traditional tableau method for $\mathcal{ALC}$ can require double-exponential time even though the decision problem is known to be EXPTIME-complete.

For fixpoint logics like PLTL, CTL and PDL, optimal tableau methods are possible if we proceed in stages with the first stage building a cyclic graph, and subsequent stages pruning nodes from the graph until no further pruning is possible or until the root node is pruned [14]. Optimality can also be obtained if we construct the set of all subsets of the Fischer-Ladner closure of the given initial formula [6]. But these methods can easily require exponential time even when it is not necessary. Indeed, the method of Fischer and Ladner will always require exponential time since it must first construct the set of all subsets of a set whose size is usually linear in the size of the given formula.

Thus a long-standing open problem in tableau methods for modal, description, and fixpoint logics has been to find an optimal and "on the fly" method

for checking satisfiability which only requires exponential time when it is really necessary. We describe such tableau methods for each of the logics ALC, $ALCI$, and PDL, but before doing so, we clarify some terminology and ask the expert reader to bear with us. In particular, we leave the actual search strategy unspecified even though depth-first search is often used in practice.

A tableau is a tree of nodes where the children of a node are created by applying a tableau rule to the parent and where each node contains a finite set of formulae. We refer to these formulae as the "contents" of a node, noting that the term "label" is also used to mean the same thing. Thus a label is *not* a name for a Kripke world as in some formulations of "labelled tableaux". The ancestors of a node are simply the nodes on the unique path from the root to that node.

A leaf node is "closed" when it can be deemed to be unsatisfiable, usually because it contains an obvious contradiction like p and $\neg p$. A leaf is "open" when it can be deemed to be satisfiable, usually when no rule is applicable to it, but also when further rule applications are guaranteed to give an infinite branch. A branch is closed/open if its leaf is closed/open. The aim of course is to use these classifications to determine whether the root node is satisfiable or unsatisfiable. But the tableau used in modal logics and those used in description logics are dual in a sense which is explained next.

Traditional Beth Tableaux are Or-trees. Traditional modal tableaux a là Beth [3] are **or-trees** in that branches are caused by disjunctions only. Each "existential/diamond" formula in a node causes the creation of a "successor world", fulfilling that formula. But such successors of a given node are created and explored one at a time, using backtracking, until one of them is closed, meaning that there is no explicit trace of previously explored "open" successors in any single tableau. This or-tree perspective makes sense when the goal is to find a closed tableau since we must close both disjuncts of a disjunction but need close only one "existential/diamond"-successor. A closed tableau implies that the root node is unsatisfiable, but an open tableau does not imply satisfiability since a different existential/diamond choice may give a closed tableau. It is only after all such "and-choices" have been shown to be open that we can assert satisfiability. We can summarise this view by writing the associated rules as below where we use | for or-branching and use ";" for set union:

$$(\vee)\ :\ \frac{X\ ;\ C_1 \vee C_2}{X\ ;\ C_1 \mid X\ ;\ C_2} \qquad\qquad (\exists)\ :\ \frac{X\ ;\ \exists R.C}{C\ ;\ \{D : \forall R.D \in X\}}$$

Traditional Description Logic Tableaux are And-trees. The tableaux used in description logics are usually **and-trees** in that branches are caused by existential/diamond formulae only. Each disjunctive formula causes the creation of a child, one at a time, using backtracking, until one child is open, meaning that there is no explicit trace of previously explored "closed" or-children in any single tableau. This and-tree perspective makes sense when the goal is to find a Kripke model that satisfies the given formula set since we must satisfy every existential/diamond formula in a node but need to satisfy only one disjunct of

a disjunction. A particular and-tree is usually called a "run" and the different or-choices give rise to multiple runs. The task is to find one single run in which all (and-)branches are open since this implies that the root is satisfiable. But a run in which some (and-)branch is closed does not imply unsatisfiability since a different or-choice may give an open run. It is only after all or-choices (i.e. runs) have been shown to be closed that we can assert unsatisfiability. We can summarise this view by writing the associated rules as below where we deliberately use $||$ to flag and-branching and put $X_i = \{D : \forall R_i.D \in X\}$ for $1 \leq i \leq n$, to save some horizontal space:

$$(\vee) \ : \frac{X \ ; \ C_1 \vee C_2}{X \ ; \ C_i} i \in \{1,2\} \qquad\qquad (\exists) \ : \frac{X \ ; \ \exists R_1.C_1 \ ; \ \cdots \ ; \ \exists R_n.C_n}{C_1 \ ; \ X_1 \ || \cdots || \ C_n \ ; \ X_n}$$

Summary Thus, in both types of tableaux, the overall search space is really an and-or tree: traditional modal (Beth) tableaux display only the or-related branches and explore the and-related branches using backtracking while description logic tableaux do the reverse. The key to our main result is to unify these two views by taking a global view which considers tableaux as and-or trees rather than as or-tree or and-trees. We can summarise this view by writing the associated rules as below using both or-branching and and-branching:

$$(\vee) \ : \frac{X \ ; \ C_1 \vee C_2}{X \ ; \ C_1 \mid X \ ; \ C_2} \qquad\qquad (\exists) \ : \frac{X \ ; \ \exists R_1.C_1 \ ; \ \cdots \ ; \ \exists R_n.C_n}{C_1 \ ; \ X_1 \ || \cdots || \ C_n \ ; \ X_n}$$

Termination via Blocking/Loop-checking. Under certain circumstances, both types of tableau can contain infinite branches because the same node appears again and again on the same branch. To obtain termination, most tableau methods employ a "loop check" or "blocking technique" [13, 2]. The simplest is called "ancestor equality blocking" where we stop expansion of a branch when a node duplicates an ancestor (on the same branch). A variation called "ancestor subset blocking" is to stop expansion if a node is a subset of an ancestor (on the same branch). Note that blocking is merely a device for termination: the *a priori* logical status of the blocked node is "unknown" rather than satisfiable or unsatisfiable. Nevertheless, in certain logics, we can classify such nodes as satisfiable, as explained next.

For most (non fix-point) modal and description logics, ancestor cycles are "good" in that a branch ending with a node blocked by an ancestor can be soundly deemed to be satisfiable. In modal tableau, such a branch will cause backtracking to a higher node where a different and-choice can be made in the hope of finding a choice that closes its branch. In description logic tableau, such a branch will cause no backtracking. In description logic tableaux, where branches are all and-branches, a further refinement called "anywhere blocking" [1] is also possible where a node can be blocked by a node which lies on a different (previously created) and-branch of the same run, although extra conditions are usually required to ensure soundness. Thus in both types of tableaux for simple logics, a blocked node is immediately classified as open even though its logical status is "unknown" rather than satisfiable.

Caching. Even with all of these blocking refinements, the naive (description logic or modal) tableau method for checking whether a concept/formula C is satisfiable w.r.t. a TBox $\mathcal{T}$ leads to a double-exponential time (2EXPTIME) algorithm because each tableau branch may have an exponential length, meaning that the method may explore a double-exponential number of nodes. To counter this, some authors have investigated the idea of remembering ("caching") the satisfiable or unsatisfiable status of previously seen nodes in a look-up table [5, 4]. Then, when a new tableau node is created, we first check whether this node already has a status of satisfiable or unsatisfiable in the cache, and attach that same status to the new node. For most logics we can refine this to the following, assuming that the current node has content X: if the cache contains a node with content $Y \supseteq X$ (respectively $X \subseteq Y$) and Y has status satisfiable (unsatisfiable) then the current node is given the status satisfiable (unsatisfiable).

Differences Between Blocking and Caching. Note the difference between blocking and caching: the status of a blocking node must be either satisfiable or unknown/open, but cannot be unsatisfiable/closed, while the status of a cached node must be known as either unsatisfiable or as satisfiable, but cannot be unknown/open. Moreover, a blocking node must be in the same and-tree while a cached node is stored in an external data structure which sits outside the tableau under construction. As a consequence, it is much easier to prove the soundness of blocking than of caching.

Global Caching. Now consider the following notion of "global caching" where each tableau node contains (is labelled with) a unique set of formulae:

Global caching: each tableau node is processed at most once in the search space.

Our notion of global caching is an immediate foundation for an EXPTIME procedure if the search space contains at most an exponential number of different nodes and as long as the "processing" at each node requires at most exponential time, both with respect to the size of the given problem. Moreover, it can replace all of the previously mentioned notions of equality-blocking and caching simultaneously because it does not rely on knowing the satisfiability or unsatisfiability status of the cached nodes viz:

Ancestor equality blocking occurs if the ancestor (with unknown status) is the first occurrence of the repeating node in the whole search space, otherwise, even the ancestor would not have been re-created but would have been replaced by a cache-hit to its first occurrence (with status unknown);

Anywhere equality blocking occurs without the extra conditions required by the existing methods for the same reasons;

Caching occurs automatically since a previously seen node whose status is known as unsatisfiable or satisfiable must have been processed, so it will never be processed again.

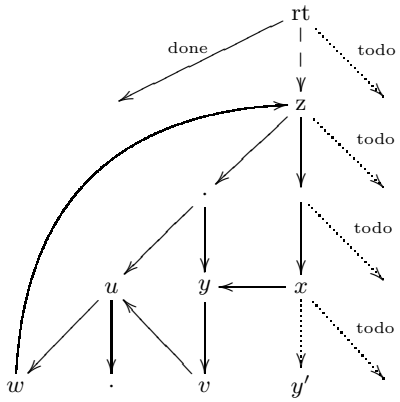

Fig. 1. Graph constructed by our algorithm for *ALC* using global caching

2 Overview for ALC

Given a TBox $\mathcal{T}$ and a concept/formula D, both in negation normal form, our method searches for an model which satisfies D w.r.t. $\mathcal{T}$ by building an and-or graph G with root node τ containing $\mathcal{T} \cup \{D\}$.

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

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

The root node has initial status `unexpanded` and our method constructs the and-or graph using a traditional strategy explained shortly. But we interleave this generation strategy with a propagation phase which propagates the status of a node throughout the graph. We explain each in turn.

Our strategy for building the and-or graph applies the rules for decomposing $\sqcap$ and $\sqcup$ repeatedly until they are no longer applicable to give a "saturated" node x, and then applies the $\exists$-rule which creates a child node for x containing $\mathcal{T} \cup \{C\} \cup \{D \mid \forall r.D \in x\}$ for each $\exists r.C \in x$. The addition of the TBox $\mathcal{T}$ to such a child is a naive way to handle TBoxes but suffices for our needs. We now saturate any such child to obtain a saturated node y, then apply the $\exists$-rule to y, and so on, until we find a contradiction, or find a repeated node, or find a saturated node which contains no $\exists$-formulae. For uniformity with our method for the extensions to inverse roles and PDL, we explore/expand children in a left to right depth-first manner, although any search strategy can be used for *ALC* [8]. All nodes are initially given a status of `unexpanded`.

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`. Notice that our method uses the $(\vee)$ and $(\exists)$ rules which use both or-branching and and-branching as summarised in Section 1. The crucial difference from traditional tableau methods

is that we create the required child in the graph G only if it does not yet exist in the graph: this step therefore uses global caching. Notice that the required child need not be an ancestor but can exist on any previous branch of the tableau. For example, as shown in Figure 1, suppose the current node is x and that the rule applied to x generates a node y' which duplicates y. The node y' is not put into G, but y becomes the child of x instead. Thus, G is really a rooted and-or tree with cross-branch edges to nodes on previously created branches like that from x to y or from v to u, or edges to ancestors like that from w to z. The problem of course is to show that this remains sound.

The propagation phase begins whenever we determine the status of a node as either `unsat` or `sat` as explained next.

A generated node that contains both A and $\neg A$ for some atomic formula A becomes a `leaf-node` with status `unsat` (i.e. unsatisfiable w.r.t. $\mathcal{T}$). A generated node to which no tableau rule is applicable becomes a `leaf-node` with status `sat` (i.e. satisfiable w.r.t. $\mathcal{T}$). Both conclusions are **irrevocable** because each relies only on classical propositional principles and not on modal principles. We therefore propagate this information to the parent node v using the kind (`or-node`/`and-node`) of v and the status of the children of v, treating `unsat` as irrevocably `false` and `sat` as irrevocably `true`. That is, an `or-node` gets status `sat` as soon as one of its children gets status `sat`, and gets status `unsat` when all of its children get status `unsat`. Dually for `and-node`s. In particular, it does not matter whether the parent-child edge is a cross-branch edge or whether it is a traditional top-down edge. If these steps cannot determine the status as `sat` or `unsat`, then the rule application sets the status to `expanded` and we return to the generation phase.

The main loop ends when the status of the initial node τ becomes `sat` or `unsat`, or when no node has status `unexpanded`. In the last case, all nodes with status $\neq$ `unsat` are given status `sat` (effectively giving the status "open" to tableau branches which loop to an ancestor) and this status is propagated through the graph to obtain the status of the root node as either `unsat` or `sat`.

This algorithms thus uses both caching and propagation techniques and runs in EXPTIME [8].

3 Overview for Inverse Roles: ALCI

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, $\forall$-formulae and $\exists$-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 $\exists$-rule which creates a node containing $\mathcal{T} \cup \{C\} \cup \{D \mid \forall r.D \in x\}$ for each $\exists r.C \in x$. The standard strategy will now saturate any such child to obtain a state y, then apply the $\exists$-rule to y, and so on, until we find a contradiction, or find a repeated node, or find a state which contains no $\exists$-formulae. Let us call x the parent state of y since all intervening nodes are not states.

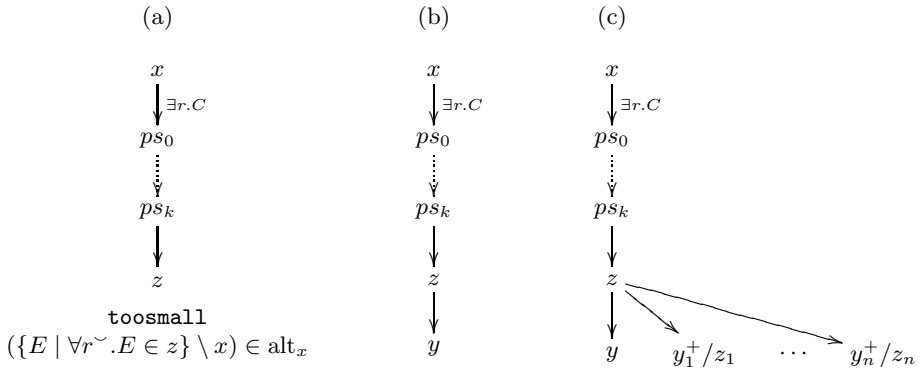

Fig. 2. The use of special node z to handle in/compatibility between states x and y. Scenario (a) occurs when x and y are incompatible. Scenario (b) occurs when x and y are compatible. Scenario (c) occurs when x and y are compatible, but y is/becomes `toosmall`.

When inverse roles are present, we require that $\{E \mid \forall r^{\smile}.E \in y\} \subseteq x$, since y is then compatible with being an r-successor of x in the putative model under construction. If some $\forall r^{\smile}.E \in y$ 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. 2, 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 $\{E \mid \forall r^{\smile}.E \in z\} \subseteq x$. If z is not compatible then we mark z as `toosmall`, and add $\{E \mid \forall r^{\smile}.E \in z\} \setminus x$ to the set of alternative sets contained in alt_x, without creating y, as shown in Fig. 2(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. 2(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

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 2(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.

Global State Caching. The complexities introduced by alternative nodes makes it difficult to use global caching so instead we use "global state caching": that is, the saturation phase is allowed to re-create prestates that occur on previous branches, but states cannot be duplicated so we must use cross-branch edges to their previous incarnations.

The resulting algorithm runs in EXPTIME [11].

4 Overview for PDL

Our algorithm starts at a root containing a given formula ϕ and builds an and-or tree in a depth-first and left to right manner to try to build a model for ϕ. The rules are based on the semantics of PDL and either add formulae to the current world using Smullyan's α/β rules from Table 1, or create a new world in the underlying model and add the appropriate formulae to it. For a node x, the attribute Γ_x carries this set of formulae.

The strategy for rule applications is the usual one where we "saturate" a node using the α/β-rules until they are no longer applicable, giving a "state" node s, and then, for each $\langle a \rangle \xi$ in s, we create an a-successor node containing $\{\xi\} \cup \Delta$, where $\Delta = \{\psi \mid [a]\psi \in s\}$. These successors are saturated to produce new states using the α/β-rules, and we create the successors of these new states, and so on.

Our strategy can produce infinite branches as the same node can be created repeatedly on the same branch. We therefore "block" a node from being created if this node exists already on any previous branch, thereby using global caching again, but now nodes are required to contain "focused sets of formulae" [11]. For example, in Fig. 3, if the node y' already exists in the tree, say as node y, then we create a "backward" edge from x to y (as shown) and do not create y'. If y' does not duplicate an existing node then we create y' and add a "forward" edge from x to y'. The distinction between "forward" and "backward" edges is important for the proofs. Thus our tableau is a tree of forward edges, with backward edges that either point upwards from a node to a "forward-ancestor", or point leftwards from one branch to another. Cycles can arise only via backward edges to a forward-ancestor.

Our tableau must "fulfil" every formula of the form $\langle \delta \rangle \varphi$ in a node but only eventualities, defined as those where δ contains $*$-connectives, cause problems. If $\langle \delta \rangle \varphi$ is not an eventuality, the α/β-rules reduce the size of the principal formula, ensuring fulfilment. If $\langle \delta \rangle \varphi$ is an eventuality, the main problem is the

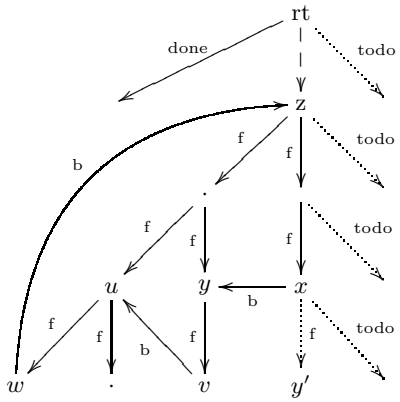

Fig. 3. Tree constructed by our algorithm using forward (f) and backward edges (b)

β-rule for formulae of the form $\langle\gamma*\rangle\varphi$. Its left child reduces $\langle\gamma*\rangle\varphi$ to a strict subformula φ, but the right child "reduces" it to $\langle\gamma\rangle\langle\gamma*\rangle\varphi$. If the left child is always inconsistent, this rule can "procrastinate" an eventuality $\langle\gamma*\rangle\varphi$ indefinitely and never find a world which makes φ true. This non-local property must be checked globally by tracking eventualities.

Consider Fig. 3, and suppose the current node x contains an eventuality e_x. We distinguish three cases. The first is that some path from x fulfils e_x in the existing tree. Else, the second case is that some path from x always procrastinates the fulfilment of e_x and hits a forward-ancestor of x on the current branch: *e.g.* the path x, y, v, u, w, z. The forward-ancestor z contains some "reduction" e_z of e_x. The path from the root to the current node x contains the only currently existing nodes which may need further expansion, and may allow z to fulfil e_z at a later stage, and hence fulfil e_x. We call the pair (z, e_z) a "potential rescuer" of e_x in Γ_x. The only remaining case is that $e_x \in \Gamma_x$ is unfulfilled, has no potential rescuers, and hence can never become fulfilled later, so x can be "closed". The machinery to distinguish these three cases and compute, if needed, all currently existing potential rescuers of every eventuality in Γ_x is described next.

A tableau node x also contains a status sts_x. The value of sts_x is the constant `closed` if the node x is closed. Otherwise, the node is "open" and sts_x contains a

Table 1. Smullyan's α- and β-notation to classify formulae

α	$\varphi \wedge \psi$	$[\gamma \cup \delta]\varphi$	$[\gamma*]\varphi$	$\langle\psi?\rangle\varphi$	$\langle\gamma;\delta\rangle\varphi$	$[\gamma;\delta]\varphi$
α_1	φ	$[\gamma]\varphi$	φ	φ	$\langle\gamma\rangle\langle\delta\rangle\varphi$	$[\gamma][\delta]\varphi$
α_2	ψ	$[\delta]\varphi$	$[\gamma][\gamma*]\varphi$	ψ		

β	$\varphi \vee \psi$	$\langle\gamma \cup \delta\rangle\varphi$	$\langle\gamma*\rangle\varphi$	$[\psi?]\varphi$
β_1	φ	$\langle\gamma\rangle\varphi$	φ	φ
β_2	ψ	$\langle\delta\rangle\varphi$	$\langle\gamma\rangle\langle\gamma*\rangle\varphi$	$\sim\psi$

function prs which maps each eventuality $e_x \in \Gamma_x$ to $\bot$ or to a set of pairs (v, e) where v is a forward-ancestor of x and e is an eventuality. The status of a node is determined from those of its children once they have all been processed. A closed child's status is propagated as usual, but the propagation of the function prs from open children is more complicated. The intuition is that we must preserve the following invariant for each eventuality $e_x \in \Gamma_x$:

> if e_x is fulfilled in the tree to the left of the path from the root to the node x then $\text{prs}_x(e_x) := \bot$, else $\text{prs}_x(e_x)$ is exactly the set of all potential rescuers of e_x in the current tableau.

An eventuality $e_x \in \Gamma_x$ whose $\text{prs}_x(e_x)$ becomes the empty set can never become fulfilled later, so $\text{sts}_x := \texttt{closed}$, thus covering the three cases as desired.

Whenever a node n gets a status `closed`, we interrupt the depth-first and left-to-right traversal and invoke a separate procedure which explicitly propagates this status transitively throughout the and-or graph rooted at n. For example, if z gets closed then so will its backward-parent w, which may also close u and so on. This propagation (update) may break the invariant for some eventuality e in this subgraph by interrupting the path from e to a node that fulfils e or to a potential rescuer of e. We must therefore ensure that the propagation (update) procedure re-establishes the invariant in these cases by changing the appropriate prs entries. At the end of the propagation (update) procedure, we resume the usual depth-first and left-to-right traversal of the tree by returning the status of n to its forward-parent. This "on-the-fly" nature guarantees that unfulfilled eventualities are detected as early as possible.

Our algorithm terminates, runs in EXPTIME, and formula ϕ is satisfiable iff the root is open [11].

5 Conclusions and Further Work

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 $\mathcal{ALC}$ [8]. Our experimental results show that global caching is competitive with mixed caching for $\mathcal{ALC}$ [10]. We have extended this method using global state caching to give an EXPTIME algorithm for checking satisfiability w.r.t. a TBox in $\mathcal{ALCI}$ [12]. We have also extended this method to give an EXPTIME algorithm for checking satisfiability in PDL [11]. Both methods have been implemented and experimental results show that they are feasible for non-trivial sized formulae. Further work is required to add optimisations to make these methods practical on large examples and to extend them to more expressive logics like SHOIQ, CTL and the modal mu-calculus.

References

1. F Baader, M Buchheit, and B Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1-2):195–213, 1996.

2. F Baader, D Calvanese, D L McGuinness, D Nardi, and P Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, England, 2003.
3. E W Beth. On Padoa's method in the theory of definition. *Indag. Math.*, 15:330–339, 1953.
4. 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.
5. F. Donini and F. Massacci. EXPTIME tableaux for $\mathcal{ALC}$. *Artificial Intelligence*, 124:87–138, 2000.
6. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Science*, 18:194–211, 1979.
7. R. Goré. Tableau methods for modal and temporal logics. In D'Agostino et al, editor, *Handbook of Tableau Methods*, pages 297–396. Kluwer, 1999.
8. R Goré and L A Nguyen. EXPTIME tableaux for ALC using sound global caching. In *Proc. DL-07*, 2007.
9. R Goré and L A Nguyen. EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In *Proc. TABLEAUX-07*, volume 4548 of *LNCS*, pages 133–148. Springer, 2007.
10. R Goré and L Postniece. An experimental evaluation of global caching for ALC (system description). In P Baumgartner, editor, *Proc. IJCAR-2008*, volume LNCS 5195, pages 299–305. Springer, 2008.
11. R Goré and F Widmann. An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In *Proc. CADE-2009*. Springer, 2009. to appear.
12. R Goré and F Widmann. Sound global state caching for ALC with inverse roles. In *Proc. TABLEAUX-2009*. Springer, 2009. to appear.
13. A Heuerding, M Seyfried, and H Zimmermann. Efficient loop-check for backward proof search in some non-classical logics. In *Tableaux 96: Proceedings of Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 1071:210-225. Springer, 1996.
14. Vaughan R. Pratt. A near-optimal method for reasoning about action. *Journal of Computer and System Sciences*, 20(2):231–254, 1980.