An Efficient Tableau Prover using Global Caching for the Description Logic $\mathcal{ALC}$

Linh Anh Nguyen

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

March 2008 (last revised: August 2008)

Abstract. We report on our implementation of a tableau prover for the description logic $\mathcal{ALC}$, which is based on the EXPTIME tableau algorithm using global caching for $\mathcal{ALC}$ that was developed jointly by us and Goré [10]. The prover, called TGC for "Tableaux with Global Caching", checks satisfiability of a set of concepts w.r.t. a set of global assumptions by constructing an and-or graph, using tableau rules for expanding nodes. We have implemented for TGC a special set of optimizations which co-operates very well with global caching and various search strategies. The test results on the test set T98-sat of DL'98 Systems Comparison indicate that TGC is an efficient prover for $\mathcal{ALC}$. This suggests that global caching together with the set of optimizations used for TGC is worth implementing and experimenting also for other modal/description logics.

1 Introduction

Description logics are multimodal logics that represent the domain of interest in terms of concepts, objects, and roles. They are useful for modeling and reasoning with structured knowledge. We can use them, for example, in conceptual modeling or as ontology languages.

The tableau method has been widely applied for description logics for different problems such as: 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 [2] for a survey). The problem of checking satisfiability of a concept w.r.t. a TBox in the basic description logic $\mathcal{ALC}$ is EXPTIME-complete [21]. However, 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 [5] have given the first tableau algorithm for this problem which runs in EXPTIME. Their algorithm 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". As far as we know, the algorithm has not been implemented yet.

Limited kinds of caching were used in most tableau decision procedures of modal/description logics, including different kinds of blocking [1, 16, 5, 14, 17, 2, 19] as well as caching nodes with determined status (satisfiable or unsatisfiable) [15, 5, 14, 4]. Note that traditional tableau provers like FaCT or DLP [15] as well as the tableau

⋆ To appear in Proceedings of CS&P'2008.

algorithm of Donini and Massacci [5] explore a search space of the form "or"-tree, where each node of the tree is an ABox, which in turn is an "and"-structure. In such decision procedures, "blocking" (including "anywhere blocking" [19]) is defined within an ABox, and in those decision procedures, prefixed sets (i.e. nodes in an ABox) with undetermined status *are not cached across ABoxes*.

In [10], together with Goré we have given an optimal (EXPTIME) tableau algorithm for $\mathcal{ALC}$ using *global caching*. The 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. Global caching means that each possible set of concepts/formulas is expanded at most once. It has also been formulated and proved sound by us and Goré for tableaux in other modal/description logics [8, 9, 11, 13]. Global caching is a useful technique for developing optimal tableau algorithms for modal/description logics whose satisfiability problem is EXPTIME-complete.

The problem we are dealing with in this work is whether the tableau algorithm proposed by us and Goré in [10] can be implemented to give an efficient prover for $\mathcal{ALC}$. This problem can be raised as what optimization techniques can be implemented together with global caching to obtain a prover for $\mathcal{ALC}$ that is both optimal (EXPTIME) from the theoretical point of view and efficient in practice. To answer this question we have implemented in C++ a tableau prover called TGC (Tableaux with Global Caching) for $\mathcal{ALC}$. Apart from global caching for nodes of the and-or graph, TGC also uses other optimizations like normalizing formulas, caching formulas using an efficient catalogue, literals elimination, semantic branching, propagation of `unsat` (unsatisfiability) using `unsat`-cores and subset-checking, as well as cutoffs. It also allows various search strategies.

The current version 1.2 of TGC supports only checking satisfiability of a set X of concepts w.r.t. a set Γ of global assumptions, where X and Γ may be defined using an acyclic TBox of concept definitions. Note that a general TBox (with cyclic concept definitions and general concept inclusions) can be translated in polynomial time into an acyclic TBox of concept definitions with a special concept standing for the set of additional global assumptions. Also note that the problem of checking satisfiability of a set X of concepts w.r.t. an acyclic TBox of concept definitions is PSPACE-complete [18], but the problem of checking satisfiability of a set X of concepts w.r.t. a set Γ of global assumptions, where X and Γ are defined using an acyclic TBox of concept definitions, is EXPTIME-complete. The approach using translation is, however, not efficient in practice, and we intend to extend TGC to directly handle also general TBoxes. The source code of the current version of TGC is available on the internet [20].

We are not able yet to compare our prover with the current versions of the other existing provers of $\mathcal{ALC}$ (because we do not know those provers well enough to test them), so we use the data from DL'98 Systems Comparison. In particular, we compare the test results of our prover (which use appropriately lower time limits for tests to reflect new speed of computers nowadays) with the test results of the provers DLP and FaCT from that comparison on the test set T98-sat.

Digression: Here, we would like to justify our approach of comparison. First, even when the source code of a system is available, it needs not to be true that the system can easily be tested. It depends on whether the system is friendly for testing, whether the documentation is available and good enough, and whether the person who wants to test it knows the language in which the system is written well enough. For example, the source C++ code of FaCT++ [22] is available but we found it difficult

	FaCT-98		TGC(1s)		DLP-98		TGC	
	n	p	n	p	n	p	n	p
k_branch	4	6	8	21	12	18	10	21
k_d4	8	21	21	21	21	21	21	21
k_dum	21	21	21	21	21	21	21	21
k_grz	21	21	21	21	21	21	21	21
k_lin	21	21	21	21	21	21	21	21
k_path	6	8	21	21	21	21	21	21
k_ph	7	6	16	6	13	7	21	7
k_poly	21	21	21	15	21	21	21	21
k_t4p	21	21	21	21	21	21	21	21

Table 1. Test results on the test set DL'98 T98-sat. The data in the columns "n" and "p" represent the numbers of solved problems for the corresponding test subsets with name ended respectively by "_n" and "_p". The tests for FaCT-98 were done with the 100s time limit for each test problem on a PC with 200 MHz Pentium II CPU and 64MB RAM. The tests for DLP-98 were done with the 100s time limit for each test problem on a SPARC computer with 150 MHz Ross RT626 CPU and 132MB RAM. The tests for TGC (resp. TGC(1s)) were done with the 7s (resp. 1s) time limit for each test problem on a PC with 2.13GHz Intel$^{(R)}$ Core^{TM}2 6400 CPU and 2GB RAM.

to learn how to test that system. Second, even in DL'98 Systems Comparison, the test results of the systems that took part in the comparison were done on different computers (with different speeds and architectures) and by the authors of the systems themselves. Third, the point of our comparison is not like "this system runs faster than the others" but is that the set of optimizations used for TGC including global caching is good (at least on the used test sets) and worth experimenting.

According to [15] and the test results from DL'98 Systems Comparison, FaCT and DLP outperformed the other systems that took part in the comparison on the test set DL'98 T98-sat, with DLP being a clear winner. DLP also outperformed the other systems that took part in the Tableaux'98 comparison [3]. We will refer to the versions of FaCT and DLP used in DL'98 Systems Comparison respectively as FaCT-98 and DLP-98. Note that both FaCT and DLP may have been improved since that time (1998), and a comparison with FaCT-98 and DLP-98 is probably not a comparison with the current versions of FaCT and DLP. Our comparison is only "relative", but hopefully valuable. (As the difference between the speeds of the computers used for testing is taken into account and our main optimization techniques[1] for TGC are different from the ones used for FaCT-98 and DLP-98, the ten years period does not matter much. Moreover, our system has been implemented and experimented in only a few months and can certainly be further optimized.)

Table 1 contains the test results of FaCT-98, DLP-98 and our prover TGC on the test set T98-sat. The results for FaCT and DLP are taken from the website of DL'98 Systems Comparison. The tests for FaCT-98 were done with the 100s time limit for each test problem on a PC with 200 MHz Pentium II CPU and 64MB RAM. The tests for DLP-98 were done with the 100s time limit for each test problem on a SPARC computer with 150 MHz Ross RT626 CPU and 132MB RAM. We did the tests for TGC on a PC with 2.13GHz Intel$^{(R)}$ Core^{TM}2 6400 CPU, 2GB RAM and operating system Fedora, using the default set of options for TGC that is specified in Section 4.

[1] including global caching, normal form of formulas, literals elimination, propagation of `unsat` for parent nodes and brother nodes, and cutoffs

According to the clock frequencies, the last computer is about 14.2 times faster than the former ones.[2] So, instead of the 100s time limit, we used only 7s for TGC, and for a closer look, we also tested TGC using 1s. We refer to TGC that uses the 1s time limit as TGC(1s). On the mentioned test set, TGC clearly outperforms FaCT-98 and slightly outperforms DLP-98 (it solves 9 problems more than DLP-98). This statement is supported also by the fact that: TGC(1s) already outperforms FaCT-98 (it solves 63 problems more than FaCT-98) and solves only 5 problems less than DLP-98.

In Table 2 we present also the test results of TGC on the test set DL'98 T98-kb (for checking satisfiability of a concept $TEST$ defined by a TBox). TGC performs on T98-kb equally well as on T98-sat.

	No. of solved problems		Total CPU time (s)		Memory use (MB)	
Time limit	7s	1s	7s	1s	7s	1s
k_branch_n	10	8	6.19	1.04	< 120	< 30
k_branch_p	21	21	3.96		< 50	
k_d4_n	21	21	0.89		< 20	
k_d4_p	21	21	0.37		< 20	
k_dum_n	21	21	0.07		< 20	
k_dum_p	21	21	0.06		< 20	
k_grz_n	21	21	0.15		< 20	
k_grz_p	21	21	0.15		< 20	
k_lin_n	21	21	0.56		< 20	
k_lin_p	21	21	0.12		< 20	
k_path_n	21	21	0.83		< 20	
k_path_p	21	21	0.72		< 20	
k_ph_n	$18^{(\star)}$	16	16.14	9.96	$< 440^{(*)}$	$< 220^{(*)}$
k_ph_p	7	6	2.09	0.26	< 80	< 20
k_poly_n	21	21	1.85		< 30	
k_poly_p	21	15	13.48	3.78	< 90	< 40
k_t4p_n	21	21	1.11		< 20	
k_t4p_p	21	21	0.38		< 20	

Table 2. Test results of TGC on the test set DL'98 T98-kb (for checking satisfiability of a concept $TEST$ defined by a TBox). The tests were done on a PC with 2.13GHz Intel$^{(R)}$ Core^{TM}2 6400 CPU and 2GB RAM, using the default options and the time limits 7s and 1s for each test problem. ($\star$): the test set k_ph_n of DL'98 T98-kb contains only 18 problems. ($*$): when testing a single problem, the memory use is significantly smaller.

The rest of this paper is organized as follows. In Section 2, we present the notation and semantics of $\mathcal{ALC}$ and the tableau algorithm using global caching for $\mathcal{ALC}$ [10]. In Section 3, we present the advanced features and optimizations used for TGC. In Section 4, we provide experimental results on usefulness of the optimizations of TGC. Section 5 concludes this work.

[2] TGC does not use the second core of CPU. Clock frequency does not tell the complete story, but using the available information we do not have better measures. As indicated by the last two columns of Table 2, TGC does not use much memory for testing the mentioned test sets DL'98 T98-kb and T98-sat, and the amount of available memory does not affect the essence of the comparison.

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

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

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

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

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

Fig. 1. Interpretation of Complex Concepts

2 Preliminaries

2.1 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:

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

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 terminological axioms/global assumptions) Γ is a finite set of concepts: traditionally, a TBox is defined to be a finite set of terminological axioms of the form $C \doteq D$ or $C \sqsubseteq D$, where C and D are concepts without $\doteq$ and $\sqsubseteq$, 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.

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 terminological axioms/global 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.

2.2 Global Caching

We assume that the reader is familiar with tableau methods [6, 7]. Suppose that L is a normal modal/description logic and CL is a traditional tableau calculus with the analytic subformula property that is sound and complete for checking whether a given formula set X_0 is L-satisfiable w.r.t. a formula set Γ of global assumptions. Then CL can be used with global caching for checking L-satisfiability by constructing an and-or graph for the input pair (X_0, Γ) as follows. (This is only an informal description; see [13, 10] for a formal formulation.) To illustrate the construction we use the tableau calculus $C\mathcal{ALC}$ [10] presented in Figure 2. The graph is initialized with root τ, which is a node with content (label) $X_0 \cup \Gamma$ and status `unexpanded`. During the construction, each node of the graph can have status `unexpanded`, `expanded`, `sat`, or `unsat`, where `sat`

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

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

Fig. 2. The tableau calculus $\mathcal{CALC}$ for $\mathcal{ALC}$ [10]. Here, X and Γ are sets of concepts and Γ plays the role of the set of global assumptions; all of the used concepts (including the ones of Γ) are assumed to be in negation normal form (where $\doteq$ and $\sqsubseteq$ are translated away and $\neg$ occurs only directly before atomic concepts); $(\bot')$ and $(\bot)$ are *inconsistency rules*, $(\sqcap)$ and $(\sqcup)$ are *static rules*, while $(\exists R)$ is a *transitional rule*; all of the rules except $(\sqcup)$ are unary rules.

(resp. `unsat`) means the content of the node is CL-consistent (resp. CL-inconsistent) w.r.t. Γ and therefore L-satisfiable (resp. L-unsatisfiable) w.r.t. Γ. As long as the status of the initial node τ is neither `sat` nor `unsat` and there are still `unexpanded` nodes, the algorithm chooses an `unexpanded` node to expand. Here, different search strategies can be applied; in particular, one can use depth-first search or heuristic search. A node v is expanded as follows. If an inconsistency rule of CL (e.g. $(\bot)$ or $(\bot')$ in the case of $\mathcal{CALC}$) is applicable to the content of v, then v receives status `unsat`, else the algorithm tries to apply a static rule of CL to the content of v, where unary rules (e.g. $(\sqcap)$ in the case of $\mathcal{CALC}$) are tried before non-unary rules (e.g. $(\sqcup)$ in the case of $\mathcal{CALC}$). If none of those rules are applicable, then the algorithm applies every transitional rule of CL (e.g. $(\exists R)$ in the case of $\mathcal{CALC}$) simultaneously in every possible way.[3] Without "caching" and "blocking", each formula set obtained from an application of a tableau rule would form a new child node with status `unexpanded`. With global caching, however, we first check whether an already existing node w has that formula set as content. (This is called *loop-checking*.) If so, then we do not create a new child node, but merely insert an edge from v to w. If a non-unary static rule is applied to v (e.g. $(\sqcup)$ in the case of $\mathcal{CALC}$), then v is an `or-node`, else v is an `and-node`. If no rule of CL is applicable to the content of v, then v receives status `sat`. Next, if the status of v is neither `sat` or `unsat` then it is changed (from `unexpanded`) to `expanded`. When a node receives status `sat` or `unsat`, the status and "related information" are propagated (mostly backward) to the other nodes of the graph. The *basic propagation* is to check if there is enough information to decide that a parent node can receive status `sat` or `unsat` according to the kind of the node (`and-node`/`or-node`) and the status of its children (where `sat` is treated as `true` and `unsat` as `false`). There are also other kinds of propagation, which will be discussed later in this paper. 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).

3 Advanced Features and Optimizations of TGC

We have chosen C++ to implement TGC in order to increase efficiency, portability, and extendibility. Pointers (of C++) are intensively used for the data structures of TGC to reduce memory use and to allow caching objects like formulas, sets of formulas, and nodes of a graph. It is clear that "caching" not only reduces memory use

[3] Assume that the transitional rules of CL are unary rules [13].

but also increases the performance. Advanced data structures of C++ with low complexity like the templates *set*, *map*, *multimap*, *priority_queue* are also intensively used for TGC. We designed and implemented TGC with the intention of extending it for other modal/description logics in a modular way, and an object-oriented programming language like C++ is suitable for the purpose.

In this section we present the advanced features and optimizations used for TGC.

3.1 The Formula Catalogue of TGC

Efficient data structures are very important for provers. In TGC, the first fundamental data type is *formula*. By a formula we mean a concept, a general concept inclusion (of the form $C \sqsubseteq D$), or a conjunction/disjunction/set of formulas. TGC globally caches not only nodes of the constructed and-or graph but also formulas that are created and used during the construction. If two formulas are essentially the same in the sense that they have the same "normal form", then their contents are represented only once in the data structure by the normal form, and *every formula is referred to via a pointer to its normalized representation.*

TGC adopts the following normalization rules:

1. (Normalized) formulas are represented in negation normal form.
2. A conjunction $C_1 \sqcap \ldots \sqcap C_k$ is represented by an "and"-set, denoted by $\sqcap\{C_1, \ldots, C_k\}$, which eliminates repetitions and makes the order of the conjuncts inessential. Using this, the tableau rule $(\sqcap)$ of $\mathcal{CALC}$ is not needed anymore for TGC, as it is implicitly embedded in the normalization process.
3. Nested "and"-sets are flattened.
   That is, $\sqcap\{\sqcap\{C_1, \ldots, C_i\}, C_{i+1}, \ldots, C_k\}$ is flattened to $\sqcap\{C_1, \ldots, C_k\}$.
4. $\forall R. \sqcap \{C_1, \ldots, C_k\}$ is replaced by $\sqcap\{\forall R.C_1, \ldots, \forall R.C_k\}$.
5. $\forall R.\top$ is replaced by $\top$.
6. $\sqcap\{\top, C_1, \ldots, C_k\}$ is replaced by $\sqcap\{C_1, \ldots, C_k\}$.
7. $\sqcap\{\bot, \ldots\}$ is replaced by $\bot$.
8. $\sqcap\{C\}$ is replaced by C.
9. Optionally (according to option *checkSubsumption4NormalForm*), if C_1 is an "or"-set containing C_2, then $\sqcap\{C_1, \ldots, C_k\}$ is replaced by $\sqcap\{C_2, \ldots, C_k\}$.
10. The rules "dual" to the above rules, except the first one.
    For example, dually to the 7th rule, $\sqcup\{\top, \ldots\}$ is replaced by $\top$.

Note that, in contrast to FaCT-98 and DLP-98 [15], normalized formulas of TGC are in negation normal form, and the 4th rule of the above list is opposite to an optional normalization rule mentioned in [15]. Notice also that the 9th rule of the above list is neither adopted nor optional for FaCT-98 and DLP-98.

TGC tags each normalized formula with a hash code and uses a special data structure called the *formula catalogue*, which is a C++ *multimap* that maps the hash code of a normalized formula to the representation of the formula. The hashing function has been chosen so that "collisions" rarely occur and the *multimap* acts nearly like a C++ *map*. When a formula is normalized, the result is included in the formula catalogue. (The catalogue contains only formulas in the TGC normal form.) TGC maintains the principle that subformulas of the currently considered formula are always in the TGC normal form and present in the formula catalogue. With this principle, normalizing a formula (and computing its hash code) can efficiently be done in linear time in the

size of the first two layers of the tree representation of the formula (because the tree representation needs to be "unfolded" only to the second layer).

As TGC does global caching for nodes of the constructed and-or graph, each formula may be the content of at most one node. Each formula keeps a reference to its eventual node by a pointer. When a node is deleted from the graph for some reason (see Section 3.8), its content is permanently kept in the formula catalogue for loop-checking. That is, loop-checking and "global caching" are done via the formula catalogue.

The formula catalogue of TGC is like an acyclic TBox of concept definitions, where each pointer to a formula plays the role of a concept name; and conversely, an acyclic TBox of concept definitions can be fully embedded into the formula catalogue, and in that case the set of global assumptions can be assumed to be empty. In particular, for checking satisfiability of a concept $TEST$ defined by an acyclic TBox $\mathcal{T}$ of concept definitions as in the case of DL'98 T98-kb, the input for TGC is $X_0 = \{TEST\}$ and $\Gamma = \{\top\}$.

3.2 Fast Detection of SAT and UNSAT

By $\overline{C}$ we denote the formula $\neg C$ in negation normal form. Observe that, if a formula C is in the TGC normal form, then $\overline{C}$ is also in the TGC normal form. An (`unsat`) clash of the form $\{A, \neg A\}$ (where A is an atomic concept) is too naive. TGC checks for (more) general clashes: if a formula set contains both C (as an element) and $\overline{C}$ (as an element or a subset), then it contains a clash. TGC does not use the tableau rule $(\bot)$ of $\mathcal{CALC}$ but checks for a general clash immediately after a node is created for the constructed and-or graph. The total time used for checking for general clashes (including time used for computing negations of formulas) is significantly reduced by caching negations of formulas. $\overline{C}$ is computed at most once for any formula C.

An `and-node` (resp. `or-node`) becomes `unsat` (resp. `sat`) when one of its children becomes `unsat` (resp. `sat`). For the remaining case, an `and-node` (resp. `or-node`) becomes `sat` (resp. `unsat`) when the number of its children with status $\neq$ `sat` (resp. $\neq$ `unsat`) reduces to 0. For fast detection of `sat` and `unsat`, TGC keeps and updates such a number for each node of the constructed and-or graph.

TGC does "subset-checking" only for small `unsat` sets. When a node becomes `unsat`, its `unsat`-core is computed, which is a subset of the content of the node that causes the node `unsat` (see Section 3.5 for details). If the `unsat`-core is an "and"-set small enough (decided by parameter *maxLenLimitForKeptUnsat*) or is not an "and"-set, then it is included into a set called *unsatFormulas*, provided that the `unsat`-core is not a super "and"-set of some already existing element of *unsatFormulas*. Thus, *unsatFormulas* contains only "small" minimal `unsat` "and"-sets, which are the `unsat`-core of some `unsat` node of the graph. When a new node is created, if its content is a super "and"-set of some element of *unsatFormulas*, the node becomes `unsat`. Such a check has time complexity $O(k \times h \times \log(l))$, where k is the size of the set *unsatFormulas*, $h = maxLenLimitForKeptUnsat$, and l is the size of the "and"-set being the content of the node. Observe that l is of polynomial rank in the size of the input pair (X_0, Γ), and when *maxLenLimitForKeptUnsat* is high enough, k may be of exponential rank in the size of the input (X_0, Γ). Observe also that, the smaller *maxLenLimitForKeptUnsat* is, the lower k will be. Since a subset-check for a node costs $O(k \times h \times \log(l))$ time units and the graph may contain exponentially many nodes, the parameter *maxLenLimitForKeptUnsat* should be small enough.

The above kind of subset-checking is only an optional heuristic of TGC, which can be set on/off using option *subsetChecking*. For completeness, TGC always does loop-checking first (via the formula catalogue).

Note that one can adopt dual subset-checking, which can be called *superset-checking*, for `sat`. We decided not to adopt this heuristic for $\mathcal{ALC}$ because superset-checking for `sat` is costly (as `sat` formula sets need to be big enough to get benefit from superset-checking).

Recall that at the last step of the tableau algorithm described in Section 1.1, all nodes with status $\neq$ `unsat` are given status `sat`. We call such nodes *looping* `sat` *nodes*. Can such `sat` nodes be detected earlier? If no `unexpanded` node is reachable from an expanded node v, then v is a looping `sat` node. TGC does such detections, but only occasionally, according to some parameters. This optimization is turned off for the test sets DL'98 T98-sat and T98-kb, because no looping `sat` nodes will be created when (the logic is $\mathcal{ALC}$ and) the only global assumption is $\top$.

3.3 Literals Elimination

Pure literal deletion and unit clause elimination are ones of the main techniques of the Davis-Putnam-Logemann-Loveland procedure, which is used in the most efficient SAT solvers for classical propositional calculus. The pure literal deletion rule states that, if an atom A occurs only positively (resp. only negatively) in the considered set of clauses, then every clause containing A (resp. $\neg A$) can be deleted from the set. The unit clause elimination rule states that, if the considered set of clauses contains a unit clause of the form A (resp. $\neg A$), then every clause containing the literal A (resp. $\neg A$) can be deleted from the set as a whole and all occurrences of $\neg A$ (resp. A) can be deleted from the remaining clauses.

Pure literal deletion is mentioned neither in [15] for FaCT-98 and DLP-98 nor in [5] for the framework for $\mathcal{ALC}$ by Donini and Massacci. Unit clause elimination was implemented for FaCT-98 and DLP-98 by the following simplification rule [15]:

$$\frac{X\,;\,\neg C_1\,;\,\dots\,;\,\neg C_n\,;\,C_1\sqcup\dots\sqcup C_n\sqcup D}{X\,;\,\neg C_1\,;\,\dots\,;\,\neg C_n\,;\,D}$$

We adopted for TGC three rules: *pure classical literal deletion*, *classical unit clause elimination* and *modal unit clause elimination*, all in a *generalized form*. By a *classical literal* of a formula C we mean an atomic concept or its negation that occurs at the *object level* (i.e. not under any modal operator) of C. A *pure classical literal* of a formula C is a classical literal of C whose negation does not occur at the object level of C. A *classical unit clause* of an "and"-set C is a classical literal belonging to the set C. The mentioned rules can be stated as follows:

Pure classical literal deletion: If A (resp. $\neg A$) is a pure classical literal of a label then assign A value $\top$ (resp. $\bot$) for the object level and normalize the label.

Classical unit clause elimination: If A (resp. $\neg A$) is a classical unit clause of a label then assign A value $\top$ (resp. $\bot$) for the object level and normalize the label.

Modal unit clause elimination: If an "and"-set C is a label and D is a formula of the form $\forall R.D'$ or $\exists R.D'$ (called a *modal literal*) such that D is an element of the "and"-set C (called a *modal unit clause* of C) and D occurs in the formula C at least twice in the form D or $\overline{D}$ at the object level, then we can change the label to

$C' \sqcap D$, where C' is the formula obtained from C by replacing every occurrence of D (resp. $\overline{D}$) at the object level of C by $\top$ (resp. $\bot$) and doing normalization.

TGC does all the three above mentioned optimizations simultaneously on all classical literals and classical/modal unit clauses of the considered label by one operation. The combined rule is called *literals elimination*. Applying it, for example, to

$$\sqcap\{A, \exists R.C, \sqcup\{\sqcap\{\neg A, A'\}, \sqcap\{A', \exists R.C, \exists S.D\}\}, \sqcup\{\neg A, \exists S.D, \forall R.\overline{C}\}\}$$

results in $\sqcap\{\exists R.C, \exists S.D\}$.

3.4 Semantic Branching

Semantic branching is also a main technique of the Davis-Putnam-Logemann-Loveland procedure for classical propositional calculus. It is applied when no further simplification (e.g. by pure literal deletion and unit clause elimination) can be done. Semantic branching on an atom A creates an or-branching with 2 branches, where A is assigned `true` for the first branch and `false` for the second branch (or vice versa). After assigning an atom value `true` or `false`, the considered set of clauses is appropriately simplified.

In the systems FaCT-98 and DLP-98 [15] and the framework for $\mathcal{ALC}$ by Donini and Massacci [5], semantic-branching is adopted/optional in the following form: choose a disjunct D of an unexpanded disjunction of the label of a node x in the considered ABox $\mathcal{A}$ and do an or-branching by considering two ABoxes which extend $\mathcal{A}$ by adding D or $\neg D$, respectively, to the label of x. If D is a "large" formula, the addition of D or $\neg D$ could result in a significantly larger search space.

TGC does semantic branching only on classical literals and modal literals (i.e. formulas of the form $\forall R.D$ or $\exists R.D$). When the literals elimination rule is not applicable (i.e. not effective), TGC chooses a formula D of the form A or $\forall R.D'$ that occurs most, in the form D or $\overline{D}$, at the object level of the content of the chosen node v, and if D occurs at least twice then TGC does semantic branching on D as follows:

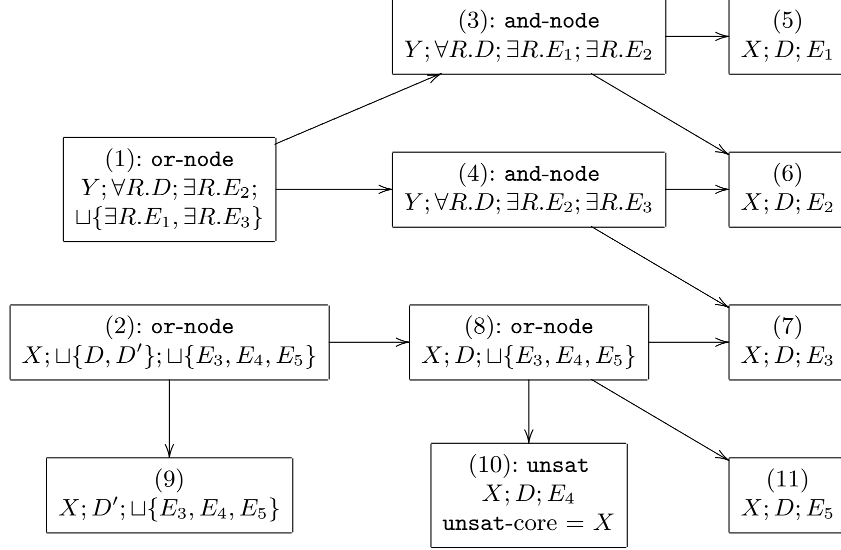
Semantic branching on a classical literal: If D is an atomic concept A then TGC makes v an or-node with successors u and w, where the content of u (resp. w) is obtained from the content of v by assigning A value $\top$ (resp. $\bot$) for the object level and doing normalization.

Semantic branching on a modal literal: If D is a modal literal then TGC makes v an or-node with successors u and w, where the content of u (resp. w) is the normalized form of $C \sqcap D$ (resp. $C \sqcap \overline{D}$) with C obtained from the content of v by replacing every occurrence of D at the object level by $\top$ (resp. $\bot$) and every occurrence of $\overline{D}$ at the object level by $\bot$ (resp. $\top$).

Note that, in TGC, the successors created by semantic branching for a node v have a "smaller" content than v. The two kinds of semantic branching of TGC are optional and controlled, respectively, by boolean options *semanticBranching1* and *semanticBranching2* (together with a threshold parameter).

3.5 Propagations

Recall that when a node receives status `sat` or `unsat`, the status and "related information" are propagated to the other nodes of the graph. The *basic propagation* is to



Here, X denote an "and"-set $\sqcap\{F_1, \ldots, F_k\}$ with $k > \mathit{maxLenLimitForKeptUnsat}$, and $Y = \sqcap\{\forall R.F_1, \ldots, \forall R.F_k\}$. By writing something like $(X; C; D)$ we denote the "and"-set consisting of the elements of X and C, D. Suppose that node (10) becomes `unsat` at some step with `unsat`-core $= X$. Then:

- By propagating the status of (10) for the parent node (8) and the brother nodes (7) and (11), these nodes become `unsat` with `unsat`-core $= X$.
- By propagating the status of (8) for the parent node (2) and the brother node (9), these nodes become `unsat` with `unsat`-core $= X$.
- By a basic propagation from node (7) to node (4), the latter node becomes `unsat` with `unsat`-core $= (Y; \exists R.E_3)$.
- By propagating the `unsat` status of (7) for the brother node (6), this node becomes `unsat` with `unsat`-core $= X$.
- By a basic propagation from node (6) to node (3), the latter node becomes `unsat` with `unsat`-core $= (Y; \exists R.E_2)$.
- By propagating the `unsat` status of (6) for the brother node (5), this node becomes `unsat` with `unsat`-core $= X$.
- By propagating the `unsat` status of (3) for the parent node (1), this node becomes `unsat` with `unsat`-core $= (Y; \exists R.E_2)$.

Fig. 3. An Example about Propagations

check if there is enough information to decide that a parent node can receive status `sat` or `unsat` according to the kind of the node (`and-node`/`or-node`) and the status of its children (where `sat` is treated as `true` and `unsat` as `false`).

TGC also does four other kinds of propagation: "propagation of `unsat`-cores", "propagation of `unsat` for parent nodes and brother nodes", "propagation of `sat` for brother nodes", and "propagation of `unsat` via subset-checking".

When a node becomes `unsat`, TGC computes a subset of the content of the node that causes the node `unsat`. The subset is called the `unsat`-*core* of the node. For example, if a node becomes `unsat` because it contains a clash of C and $\overline{C}$, then the `unsat`-core of the node is $\{C, \overline{C}\}$ if $\overline{C}$ is not an "and"-set, or $\{C\} \cup \overline{C}$ otherwise. Consider another case: Suppose that u is a node with content $\sqcap\{\sqcup\{C, D\}, E_1, \ldots, E_k\}$, which is expanded by the tableau rule $(\sqcup)$ using $\sqcup\{C, D\}$ as the principal formula, resulting in two child nodes v and w whose contents are $\sqcap\{C, E_1, \ldots, E_k\}$ and $\sqcap\{D, E_1, \ldots, E_k\}$, respectively. Notice that u is an `or-node`. Suppose that at some later step, u becomes `unsat` because both v and w are `unsat`. Let the `unsat`-cores of v and w be $\sqcap X$ and $\sqcap Y$, respectively, with $X \subseteq \{C, E_1, \ldots, E_k\}$ and $Y \subseteq \{D, E_1, \ldots, E_k\}$. Then the `unsat`-core of u is $\sqcap Z$, with $Z = (X \setminus \{C\}) \cup (Y \setminus \{D\}) \cup \{\sqcup\{C, D\}\}$. Note that the `unsat`-cores of v and w are propagated backward for computing the `unsat`-core of the parent node u. The `unsat`-core of u in turn is also propagated backward in a similar way. The procedure of computing `unsat`-core is defined also for the remaining rules (i.e. literals elimination, semantic branching, and the transitive rule $(\exists R)$), but we omit the details. (If option *useUnsatCores* is set off, the `unsat`-core of a node is defined to be the whole content of the node.)

Observe that an `unsat`-core is like a formula set returned by the procedure Apply of the DFS algorithm given by Donini and Massacci for $\mathcal{ALC}$ [5, Fig.8].[4] They use such formula sets for subset-checking and backjumping. (FaCT-98 and DLP-98 use a different technique called "dependency sets" for backjumping [15].) We use `unsat`-cores for similar purposes, but technically, in a different way. The reason is that TGC works on an and-or graph with various search strategies, while the DFS algorithm by Donini and Massacci works on a search space of the tree form, and backjumping is formulated for DFS on a search tree.

Recall that if the `unsat`-core of an `unsat` node u is an "and"-set small enough (decided by parameter *maxLenLimitForKeptUnsat*) or is not an "and"-set, it is added into the set *unsatFormulas*, provided that the `unsat`-core is not a super "and"-set of some already existing element of *unsatFormulas*. After executing such an action, TGC propagates `unsat` via subset-checking for the `unsat`-core as follows. For every node v, if the status of v is neither `sat` nor `unsat` and the content of v is a super "and"-set of (or is the same as) the `unsat`-core of u, then v becomes `unsat`. This subset-checking is optional and controlled by boolean option *subsetChecking2*. Notice that it is a supplement to the subset-checking controlled by option *subsetChecking*, which was discussed in Section 3.2.

If the `unsat`-core of an `unsat` node u is not added to the set *unsatFormulas*, then TGC also propagates the status of u via subset-checking but in a local scale, which is called *propagation of* `unsat` *for parent nodes and brother nodes*: for every node v being a parent or a brother of u, if the status of v is neither `sat` nor `unsat` and the content

[4] Note that TGC and the DFS algorithm for $\mathcal{ALC}$ by Donini and Massacci [5] use (different kinds of tableau calculi and) different sets of tableau rules.

of v is a super "and"-set of (or is the same as) the `unsat`-core of u, then v becomes `unsat`. In a dual way, TGC does *propagation of* `sat` *(via superset-checking) for brother nodes*, which is controlled by boolean option *localSupersetChecking*.

Figure 3 presents an example illustrating propagations done by TGC. Motivated by the example, to make `unsat`-cores smaller and to maximize propagation, TGC gives a preference to "propagation of `unsat` for parent nodes", then "propagation of `unsat` for brother nodes", then "basic propagation of `unsat`".

3.6 Cutoffs

It is expected that a node u of the and-or graph is expanded only when it could affect the status of the initial node τ. That is, we should expand a node u only when there is a path from τ to u with no node with status `sat` or `unsat`. If u satisfies that condition then we say that u is *essentially reachable* from τ. As the graph may have exponentially many nodes, computing and updating the exact relation "essentially reachable" is costly.[5] TGC uses a heuristic to overcome this problem. It maintains a tree called an "essential reachability tree" of the graph and a set *reachable* of nodes with the invariants that:

- every node of the set *reachable* has status $\notin \{\texttt{sat}, \texttt{unsat}\}$;
- every node with status $\notin \{\texttt{sat}, \texttt{unsat}\}$ is a node of the "essential reachability tree" (thus, *reachable* is a subset of the set of nodes of the tree);
- if *reachable*(u) holds then u is essentially reachable from τ;
- if u is absent (deleted) from *reachable* then all nodes of the subtree with root u of the "essential reachability tree" are also absent (deleted) from *reachable*.

The cost for maintaining the above invariants is relatively low. TGC uses an expanding queue, which is a stack in the case of DFS and a priority queue is the case of heuristic search. While the initial node τ is neither `sat` nor `unsat` and the expanding queue is not empty: TGC pops a node u from the expanding queue; if u does not belong to *reachable*, then TGC ignores u and continues to pop another node from the expanding queue, else TGC expands u; if the queue is empty then TGC rebuilds it together with a new minimal "essential reachability tree" and a new maximal set *reachable* that satisfy the invariants; if the queue after rebuilding is still empty then TGC exits with result `sat` (for τ).

3.7 Search Strategies

As mentioned in the previous subsection, TGC realizes various search strategies by using an expanding queue, which is a stack in the case of depth-first search (DFS), a priority queue is the case of heuristic search (HS), and a FIFO queue in the case of breadth-first search (BFS).

Expanding a node, TGC gives a preference to the literals elimination rule, then the semantic branching rule, then the (syntactic branching) rule ($\sqcup$), then the transitional rule of $\mathcal{CALC}$. This order does not affect completeness of the prover. The way of choosing an atomic concept to semantically branch on was discussed in Section 3.4. When the syntactic branching rule ($\sqcup$) is chosen (and applicable), TGC chooses a heaviest "or"-set (disjunction) to branch on. The weight of a formula can be set to its

[5] Counting the number of parents of a node does not help when the node is in cycles.

size (i.e. the length of the traditional representation of the formula) or can be manually defined/implemented for TGC.

Using DFS, when a node is chosen and expanded, its successor nodes are considered for expanding in the increasing order of weights, i.e. lighter nodes first, heavier nodes later. (A node u is lighter than a node v if the weight of the content of u is smaller than the weight of the content of v.)

BFS appears to be a weak strategy for TGC, while DFS performs well. We have not found heuristic search strategies for TGC that are better than DFS yet (especially on the test sets DL'98 T98-sat and T98-kb). It is a subject for further investigation. The cost of operations on possible expanding queues for TGC is small in comparison with the other ones. (Using heuristic search instead of DFS, TGC runs only slightly more slowly on the test sets T98-sat and T98-kb.) The real problem for heuristic search is to find good heuristic functions with low complexity.

3.8 Efficient Memory Management

As the constructed and-or graph may be exponentially large, efficient memory management is an important matter of TGC. For this, TGC tries to minimize the graph whenever possible and to reduce the formula catalogue occasionally. Note that keeping the graph small also increases the performance of TGC.

Using the formula catalogue to permanently keep `sat` formulas and their status (`sat`), TGC always deletes `sat` nodes from the graph and do not create nodes with a content known to be `sat`. When boolean option *useUnsatCores* is set off, TGC does the same thing for `unsat` nodes. When the option is set on and a node u becomes `unsat`, TGC computes the `unsat`-core of u and changes the content of u to that `unsat`-core. If the new content of u is the same as the content of a node $v \neq u$, then TGC merges u and v in an appropriate way. When option *useUnsatCores* is set on, `unsat` nodes may be needed to compute `unsat`-cores for their parent nodes and cannot be deleted. But, if an `unsat`-node is not needed for that purpose for the current graph, TGC deletes the node (but keeps its content and the status in the formula catalogue for loop-checking). The same node may be re-created later (with a very small cost) for the purpose of computing and propagating `unsat`-cores.

The last two columns of Table 2 show that TGC manages well with memory.[6] Note also that Moore's law holds not only for CPU speed but also for computer memory. These disprove the stereotype that global caching must use too much memory and is thus useless.

4 Experimental Results on Usefulness of the Optimizations

In this section, we provide test results about the usefulness of the optimizations of TGC on the test set DL'98 T98-kb. The tests were done on a PC with 2.13GHz Intel(R) Core™2 6400 CPU and 2GB RAM, using the 1s time limit for each test problem. We consider only the number of solved problems (but not the used CPU time).

The default set of options for TGC, numbered (1) in Table 3, has the following boolean options set on:

[6] The amount of memory used by TGC can be observed using "System Monitor" of Fedora. It is refreshed after each 0.25s (for "Resources") and the measuring method is thus not exact, but we have loosened the estimations of memory used by TGC.

This table contains test results of TGC on the test set DL'98 T98-kb. The tests were done on a PC with 2.13GHz Intel(R) Core™2 6400 CPU and 2GB RAM, using the 1s time limit for each test problem. Each table cell contains the number of solved problems for the corresponding subset of tests and the corresponding set of options, where the dot symbol denotes 21 (the number of test problems in the subset). Some cells contain a range of numbers because the computer may have performed a little differently over time and TGC is random at a certain level (due to the use of pointers as elements of C++ *set*s and *map*s). The column headers denote the following sets of optimizations:

(1) is the default set of options
(2) is (1) with *cutoff* set off
(3) is (1) with *useUnsatCores* set off
(4) is (1) with *literalElimination*, *semanticBranching1* and *semanticBranching2* set off
(5) is (1) with *semanticBranching1* and *semanticBranching2* set off
(6) is the set with all the boolean options set off except *dfsExpanding*
(7) is (6) with *cutoff* set on
(8) is (6) with *useUnsatCores* set on
(9) is (6) with *literalElimination* set on
(10) is (6) with *cutoff*, *useUnsatCores* and *literalElimination* set on

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
k_branch_n	8	8	8	5	8	2	3	3	8	8
k_branch_p	·	·	·	8 - 9	·	2	2	6 - 7	8	·
k_d4_n	·	·	·	·	·	11	·	12	·	·
k_d4_p	·	·	·	·	·	·	·	·	·	·
k_dum_n	·	·	·	·	·	·	·	·	·	·
k_dum_p	·	·	·	·	·	·	·	·	·	·
k_grz_n	·	·	·	·	·	·	·	·	·	·
k_grz_p	·	·	·	·	·	8	8	·	·	·
k_lin_n	·	·	·	·	·	·	·	·	·	·
k_lin_p	·	·	6	·	·	5	6	·	6	·
k_path_n	·	·	·	·	·	6	·	12 - 14	·	·
k_path_p	·	·	·	·	·	6 - 7	12 - 14	17 - 19	·	·
k_ph_n	16	16	16	13	18	13 - 14	13	13	18	18
k_ph_p	6	6	6	6	6	6	6	6	6	6
k_poly_n	·	·	·	·	·	17 - 20	19 - 21	·	·	·
k_poly_p	15	14	13	13 - 14	16	7 - 9	7 - 9	9 - 10	15	16
k_t4p_n	·	·	·	·	·	·	·	·	·	·
k_t4p_p	·	·	·	·	·	·	·	·	·	·

Table 3. Effects of the Optimizations Used for TGC

– *dfsExpanding* (DFS, see Section 3.7),
– *cutoff* (cutoffs, see Section 3.6),
– *useUnsatCores* (using `unsat`-cores, see Section 3.5),
– *literalElimination* (literals elimination, see Section 3.3),
– *semanticBranching1* (semantic branching on classical literals, see Section 3.4),
– *semanticBranching2* (semantic branching on modal literals, see Section 3.4),
– *subsetChecking* (subset-checking for `unsat`, see Section 3.2),
– *subsetChecking2* (propagation of `unsat` via subset-checking, see Section 3.5),
– *localSupersetChecking* (propagation of `sat` for brother nodes via superset-checking, see Section 3.5).

The default set of options uses parameter *maxLenLimitForKeptUnsat* = 5 for *subsetChecking* and *subsetChecking2*. It sets off the boolean option *checkSubsumption4NormalForm*, because subsumption checking for normal form of formulas significantly slows down the performance of TGC (on the test sets T98-sat and T98-kb).

Another base set of options, numbered (6) in Table 3, has the boolean option *dfsExpanding* set on but all the other mentioned boolean options set off.

Our tests show that, w.r.t. the default set of options, setting off any option among *semanticBranching1*, *semanticBranching2*, *subsetChecking*, *subsetChecking2*, *localSupersetChecking* does not decrease the number of problems solved by TGC for the test set T98-kb. Also, w.r.t. the set (6) of options, setting on any of the mentioned options does not increase the number of problems solved by TGC for the test set T98-kb. In fact, comparing columns (1), (5), (10) of Table 3, it can be seen that *semanticBranching1* and *semanticBranching2* slightly slow down the performance of TGC, while *subsetChecking*, *subsetChecking2* (with *maxLenLimitForKeptUnsat* = 5), and *localSupersetChecking* are neutral for the performance of TGC (on the test set T98-kb). We therefore conclude that semantic branching, global subset-checking (for `unsat`), and local superset-checking (for `sat`) are not significant for TGC on the test set T98-kb.

Table 3 shows that "literals elimination", "propagation of `unsat` using `unsat`-cores and subset-checking for parent nodes and brother nodes" (controlled by the boolean option *useUnsatCores*) and "cutoffs" are very useful optimizations for TGC.

5 Conclusions

We have implemented TGC as a prover for checking satisfiability of a set X of concepts w.r.t. a set Γ of global assumptions, where X and Γ may be defined using an acyclic TBox of concept definitions. Apart from global caching, TGC uses a number of optimization techniques. The main optimizations for TGC are:

Normalization Our normal form of formulas is based on negation normal form and hence different from the ones of FaCT-98, DLP-98 and the framework for $\mathcal{ALC}$ by Donini and Massacci [5]. It essentially improves the normal form proposed by us and Goré in the unpublished manuscript [12].

Caching formulas The formula catalogue of TGC is an efficient data structure that significantly improves the performance of the system. Caching negations of formulas is an important feature of the catalogue.

Global caching for nodes of the and-or graph This technique is new w.r.t. the systems FaCT, DLP and the framework for $\mathcal{ALC}$ by Donini and Massacci [5].

Literals elimination The idea of this technique is natural and has been applied earlier for other systems. What is new here is the adoption itself of pure classical literals deletion for a modal prover as well as the generalized form of modal unit clause elimination.

Propagation of `unsat` for parent nodes and brother nodes An `unsat`-core is like a formula set returned by the procedure Apply of the DFS algorithm given by Donini and Massacci for $\mathcal{ALC}$ [5, Fig.8]. However, using `unsat`-cores to propagate `unsat` in a local scale via subset-checking for parent nodes and brother nodes is first proposed in this work and implemented for a modal prover.

Cutoffs The general idea of cutoffs was mentioned in [12]. We have used, however, a specific heuristic for its implementation in TGC.

Compacting the and-or graph Compacting contents of nodes and merging nodes of the same content to keep the and-or graph small and save memory were proposed by us and Goré in [12] and have been implemented for TGC.

One of the main contributions of this work is that we have established a set of optimizations that co-operates very well with global caching and various search strategies on search spaces of the form and-or graph. Some optimizations like propagation of `unsat` (resp. `sat`) via subset-checking (resp. superset-checking) for brother nodes as well as cutoffs are specific to that kind of search spaces. (The optimizations "dependency directed backtracking" and "backjumping" used in FaCT and DLP are specific to search spaces of the form "or"-tree and are neither suitable nor necessary for TGC.)

Despite we have not done thorough comparisons between TGC and the (current versions of the) other provers of $\mathcal{ALC}$, the comparison with FaCT-98 and DLP-98 presented in Table 1 clearly indicates that TGC is an efficient prover for $\mathcal{ALC}$. Recall also that, in contrast to FaCT and DLP, whose worst complexity for $\mathcal{ALC}$ is at least 2EXPTIME, TGC is an optimal (EXPTIME) prover for $\mathcal{ALC}$. Our conclusion is that global caching is worth implementing and experimenting for modal/description logics.

Of course, there remains lots of work to do with TGC, for example, to directly and efficiently handle general TBoxes and to deal with ABoxes. We intend to extend TGC also for other modal/description logics.

Acknowledgements: This work has been supported by grant N N206 399334 from the Polish Ministry of Science and Higher Education.

References

1. F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. *Artif. Intell.*, 88(1-2):195–213, 1996.
2. F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
3. P. Balsiger and A. Heuerding. Comparison of theorem provers for modal logics - introduction and summary. In H.C.M. de Swart, editor, *Proceedings of TABLEAUX'1998, LNCS 1397*, pages 25–26. Springer, 1998.
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. Fitting. *Proof Methods for Modal and Intuitionistic Logics.* D. Reidel Publishing Co., Dordrecht, 1983.
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. A tableau system with automaton-labelled formulae for regular grammar logics. In B. Beckert, editor, *Proceedings of TABLEAUX 2005, LNAI 3702*, pages 138–152. Springer-Verlag, 2005.
9. R. Goré and L.A. Nguyen. Analytic cut-free tableaux for regular modal logics of agent beliefs. In F. Sadri and K. Satoh, editors, *Proceedings of CLIMA VIII*, pages 274–289, 2007.
10. R. Goré and L.A. Nguyen. EXPTIME tableaux for ALC using sound global caching. In D. Calvanese et al., editor, *Proceedings of Description Logics 2007*, pages 299–306, 2007.
11. R. Goré and L.A. Nguyen. EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In N. Olivetti, editor, *Proc. of TABLEAUX 2007, LNAI 4548*, pages 133–148. Springer-Verlag, 2007.
12. R. Goré and L.A. Nguyen. Optimised EXPTIME tableaux for $\mathcal{ALC}$ using sound global caching, propagation and cutoffs. Manuscript, available at `http://www.mimuw.edu.pl/~nguyen/papers.html`, 2007.
13. R. Goré and L.A. Nguyen. Sound global caching for abstract modal tableaux. Available at `http://www.mimuw.edu.pl/~nguyen/papers.html`, 2007.
14. V. Haarslav and R. Möller. Consistency testing: The RACE experience. In R. Dyckhoff, editor, *Proceedings of TABLEAUX'2000, LNCS 1874*, pages 57–61. Springer, 2000.
15. I. Horrocks and P.F. Patel-Schneider. Optimizing description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.
16. I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies. *J. Log. Comput.*, 9(3):385–410, 1999.
17. I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic SHIQ. In D.A. McAllester, editor, *Proceedings of CADE-17, LNCS 1831*, pages 482–496. Springer, 2000.
18. C. Lutz. Complexity of terminological reasoning revisited. In H. Ganzinger, D.A. McAllester, and A. Voronkov, editors, *Proceedings of LPAR'1999, LNCS 1705*, pages 181–200. Springer, 1999.
19. B. Motik, R. Shearer, and I. Horrocks. Optimized reasoning in description logics using hypertableaux. In F. Pfenning, editor, *Proceedings of CADE-21, LNCS 4603*, pages 67–83, 2007.
20. L.A. Nguyen. The source code of TGC. `http://www.mimuw.edu.pl/~nguyen/systems.html`, 2008.
21. K. Schild. Terminological cycles and the propositional μ-calculus. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proceedings of KR'94, LNCS 4130*, pages 509–520. Morgan Kaufmann, 1994.
22. D. Tsarkov and I. Horrocks. Description logic reasoner: System description. In U. Furbach and N. Shankar, editors, *Proceedings of IJCAR'2006*, pages 292–297. Springer, 2006.