

An Experimental Evaluation of Global Caching for \mathcal{ALC} (System Description)

Rajeev Goré and Linda Postniece

Computer Sciences Laboratory
The Australian National University
Canberra ACT 0200, Australia
[Rajeev.Gore|Linda.Postniece]@anu.edu.au

Abstract. Goré and Nguyen have recently given the first optimal and sound method for global caching for the description (modal) logic \mathcal{ALC} , and various extensions. We report on an experimental evaluation for \mathcal{ALC} plus its reflexive and reflexive-transitive extensions which compares global caching, mixed caching, unsat caching and no caching, all in a single common framework implementing a depth-first search strategy. We also evaluated a version of global caching using an unrestricted search strategy which necessarily sits outside the common framework. We conclude that global caching is an improvement over the other methods in most but not all cases.

Introduction. Description logics are useful in knowledge representation and reasoning in artificial intelligence and in the semantic web. The description logic \mathcal{ALC} is a multi-modal version of the normal modal logic K and forms the basis of many useful extensions. Many applications of automated reasoning in \mathcal{ALC} can be reduced to the following EXPTIME-complete \mathcal{ALC} -satisfiability problem: given a finite TBox Γ of “global assumptions” and a single formula φ , decide whether there is an \mathcal{ALC} -model which makes φ true at one world, and which makes each member of Γ true at all worlds. We assume the reader is familiar with the terminology used in practical reasoning in (modal) description logics.

Although numerous optimisations are essential for practical (automated) reasoning in description logics, caching has remained an “external” optimisation which is often “bolted on” in an ad-hoc way. Recently, Goré and Nguyen [2] have given the first optimal (EXPTIME) and sound method for global caching for \mathcal{ALC} , and various extensions. We report on an experimental evaluation of various caching methods for \mathcal{ALC} , including two versions of global caching.

Some existing caching methods are sound only when using depth-first search (DFS), so we built a generic implementation with a fixed DFS search strategy and incorporated all methods into this framework. The global caching method of Goré and Nguyen does not rely on DFS, so we built another implementation which uses only global caching, but with an unrestricted search strategy.

While our prover is only a prototype, to retain practicality, we included the following optimisations into both basic (DFS and unrestricted) frameworks:

Negation normal form: at the start, all input formulae are put into an implication-free form such that negations appear only in front of atomic formulae;

Semantic branching for atoms: an (\sqcup) -rule application with principal formula $p \sqcup \psi$ creates a left denominator for p as usual, but creates a right denominator containing $\{\psi, \neg p\}$ instead of one containing just ψ ;

Normalisation: there are three aspects to this optimisation:

- Modus ponens: an extra step is included at each node so that a node containing φ and $\psi \sqcup \neg\varphi$ has ψ added to it immediately;
- Subsumption: if a node contains $\{\psi, \psi \sqcup \varphi\}$, then $\psi \sqcup \varphi$ is deleted;
- Implicit “and”: conjunctions are flattened recursively into their conjuncts;

Backjumping: or “use-check” whereby a clash on the left branch for $X; \varphi$ of an (\sqcup) -rule application on $X; \varphi \sqcup \psi$ can allow us to close the right branch $X; \psi$ without further expansion because extra stored information allows us to trace the source of the “clash” in the left branch to X , meaning that the right branch is guaranteed to close in the same way. It is well-known [4] that caching can interact with certain optimisations like backjumping.

Lazy unfolding: include certain TBox axioms in a demand driven way [5].

The source code of our prototype is available at <http://users.rsise.anu.edu.au/~linda/CWB.html> via the world wide web.

Acknowledgements. We thank Florian Widmann for many useful suggestions on numerous aspects of this work, and Linh Anh Nguyen for many clarifications.

Programming Language and Basic Data Structures. Our prototype is written in C++ using the STL data structures.

Given a finite TBox Γ and an initial formula φ , the satisfiability problem has exponential complexity w.r.t. the length n of $X_0 = \Gamma \cup \{\varphi\}$. The set $\text{Sf}(X_0)$ of subformulae has cardinality at most n , the number of subformulae and their negations is $2n$ and the search space contains at most 2^{2n} different nodes.

Our nodes (formula sets) consist of plain formulae and each node has a unique identifier, which is just an integer. Thus there are no world/individual “names” or “labels”, meaning that we cannot handle ABoxes.

We let \leq be the order in which the formulae are encountered during parsing, so “the i^{th} formula” is unique. Each tableau node carries a bit-string of length $2n$ with the i^{th} bit set to 1/0 if the i^{th} formula is present/absent from this node. The nodes are stored in a red-black tree using `std::map`, so finding a particular node requires $\log_2(2^{2n}) = 2n$ (i.e. polynomial) time.

A rule is applied to a node only if applying that rule produces at least one formula new to the node. Before adding a new node to the tableau, the contents of the node are normalised using the normalisation steps outlined above.

Various Caching Methods. By “caching” we mean the storing of nodes and their status (`sat/unsat`) so that future computations of the nodes are avoided. By “global caching”, we mean the method outlined in [2] in which no node is explored more than once, even if its status is “unknown”. We assume familiarity with these notions but explain the various caching methods briefly:

- No caching DFS (NC): there is no additional caching method, thus we need an explicit ancestor equality-blocking (loop-check) facility for termination;
- Unsat caching DFS (UC): we maintain an explicit separate data-structure which stores every **unsat** node ever found. The procedure searches this cache before it creates a child node. If the required child is in the cache, then it must be **unsat** so the procedure backtracks, else the procedure creates the child and continues with the usual DFS procedure. A node is added to the **unsat**-cache once its status becomes **unsat**, thus this cache grows monotonically. There is a separate ancestor equality-blocking (loop check) to guarantee termination;
- Mixed caching DFS (MC): we maintain an **unsat**-cache as described above but we also keep a local **sat**-cache. The **sat** cache grows monotonically as long as the DFS procedure remains in the same “and-structure” but it is emptied every time we move from the left branch of an (\sqcup)-rule to the right branch since unsoundness can result if we do not do so [1]. There is a separate ancestor equality-blocking (loop check) to guarantee termination. We have implemented the algorithm given by Donini and Massacci [1] in a naive way in that we explicitly copy the **sat**-cache when required. Thus there are possibilities for optimisation here such as lazy saving [5].
- Global caching DFS with propagation (GC-DFS): we use DFS to explore the search-space but do not reclaim space upon backtracking as in all previously described methods. Instead, we keep a graph $\langle V, E \rangle$ of vertices whose status can be either **unexpanded**, **expanded**, **unsat** or **sat**, and propagate the status of **unsat/sat** nodes through the graph as described in detail in [2]. There is no extra ancestor equality-blocking loop check as global caching guarantees termination. Propagation is essential as it is easy to construct examples where omitting propagation can lead to unsoundness.
- Unrestricted global caching and propagation (GC-NonDFS): instead of DFS, we explore the graph using a DFS-like strategy, see [2, Algorithm 2]. Our strategy differs from DFS in that at every node x where the applied rule gives k denominators y_1, \dots, y_k , we create (or lookup) each successor y_i immediately, and place it in the queue for expansion, rather than creating only the first successor y_1 and exploring it as in DFS. Moreover, if at any time we encounter a cache hit to an unexpanded node z (which must already be in the queue), we bring z to the front of the queue, since we know that there are now at least two nodes that rely on the status of z , indicating that z should be processed sooner rather than later.

Problem Sets, Experiments and Results. We ran our experiments on a PC with 2.40GHz Intel(R) Core(TM)2 CPU 6600 and 2GB RAM. We tested our implementation on the K, KT and S4 problem sets from the Logics Work Bench (LWB) benchmarks [3] and the K TBox problem set from the DL98 benchmarks <http://dl.kr.org/dl98/comparison/data.html>. The LWB is a sequent-based prover so a formula φ is “provable” if $\{\neg\varphi\}$ is unsatisfiable, and φ is “not provable” otherwise: hence the appellation “p” or “n” on the problem sets. The LWB problems are actually for *monomodal* K, KT and S4 but are adequate for testing the *multimodal* logic ALC because the caching methods

never inspect the value of the role name (accessibility relation). However, the LWB problems do not contain TBox axioms, therefore we also used the DL98 TBox problem set. We also tested our implementation on the galen1.tkb from the DL98 data set, which is a significantly scaled down version of the Galen ontology, containing TBox axioms and multiple roles (modalities).

Each table to follow contains a column for each of the five different caching methods, with these five methods classified into two classes. The “Unsophisticated” ones comprising of the NC and UC methods and the “Sophisticated” ones comprising of the MC and GC-DFS and GC-NonDFS methods.

For each caching method, there are five entries corresponding to time-outs of 1 second, 2 seconds, 4 seconds, 8 seconds and 16 seconds. Thus a shift of one place to the right in any single sequence of five entries corresponds to a doubling of the time-out allowed. Each such entry is an integer between 1 and 21 indicating the most complex problem that could be solved in the given time-out. Since the time-outs increase monotonically from left to right, we have replaced consecutive occurrences of “21” by blanks. Thus, within a particular caching method, the more blanks the better since this indicates that that method could solve the most complex problem using the smallest time-out. The bottom-most row of each table shows the number of “blanks” for that caching method over all problem instances: a rather crude measure of overall performance.

Finally, we note that differences of 1 integer value are not significant as we noticed such differences over two identical runs at different times, particularly for the 1 second, 2 second and 4 second time-outs.

Results for K. Given the ± 1 fluctuations, Table 1 shows the following trends. Sophisticated caching is better than Unsophisticated caching. Within the Sophisticated caching methods, the best of GC-DFS and GC-NonDFS is usually better than MC. We also see that the GC-NonDFS method significantly outperforms MC for problems k.d4.n and k.t4p.n. There appears to be no clear winner between the two “global caching” methods GC-DFS and GC-NonDFS, although GC-NonDFS has the highest blank-count (due to k.d4.n).

Results for KT. The general trend continues in that the Sophisticated methods outperform the Unsophisticated methods for most problems, but for many problems, the differences are not great. The best of the two global caching methods (GC-DFS and GC-NonDFS) continues to outperform MC. Specifically, for kt.t4p.n, GC-NonDFS significantly outperforms MC, and for kt.t4p.p, GC-DFS outperforms MC.

Results for S4. As for K and KT, most of the results get better as we move from the Unsophisticated to the Sophisticated methods. However, something strange happens for s4.grz.n, where we see that global caching performs significantly worse in both of its incarnations. Another strange result is s4.t4p.n, where mixed caching significantly outperforms all others. This time, the MC method has the highest blank-count, but this is mostly due to its better performance on s4.grz.n.

K Tests: highest problem complexity solved in 1 2 4 8 16 seconds						
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFS GC	NonDFS GC
k.branch_n	7 9 9 10 11	7 9 9 10 11	8 8 9 10 11	8 8 9 10 11	8 8 9 10 11	8 8 9 10 11
k.branch_p	16 19 21	16 19 21	16 19 21	17 20 21	17 20 21	17 20 21
k.d4_n	5 6 6 7 8	6 6 7 7 8	12 16 21	8 10 14 19 21	16 21	16 21
k.d4_p	7 7 7 8 9	6 6 7 7 8	21	21	21	21
k.dum_n	21	21	21	21	21	21
k.dum_p	21	21	21	21	21	21
k.grz_n	21	21	21	21	21	21
k.grz_p	21	21	21	21	21	21
k.lin_n	10 13 19 21	12 13 18 21	10 14 18 21	11 14 19 21	11 15 19 21	11 15 19 21
k.lin_p	21	21	21	21	21	21
k.path_n	3 3 3 3 3	4 4 5 5 6	6 8 10 13 17	8 11 13 18 21	8 11 13 17 21	8 11 13 17 21
k.path_p	3 3 4 4 4	5 6 6 7 8	6 8 10 13 16	9 10 13 18 21	8 11 14 18 21	8 11 14 18 21
k.ph_n	6 6 7 8 8	6 6 7 8 8	6 6 7 8 8	6 6 7 8 8	6 6 7 7 8	6 6 7 7 8
k.ph_p	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6
k.poly_n	7 8 10 12 15	7 8 10 12 15	10 14 17 21	12 14 17 21	12 14 17 21	12 14 17 21
k.poly_p	12 16 19 21	13 16 19 21	13 16 19 21	11 16 18 21	11 16 18 21	11 16 18 21
k.t4p_n	4 7 9 12 16	5 7 9 13 18	10 14 20 21	12 17 21	14 18 21	14 18 21
k.t4p_p	6 8 12 16 20	9 16 21	9 19 21	13 20 21	13 19 21	13 19 21
blank count	24	26	34	33	36	36

Table 1. Results for K Tests

Results for K TBox. The LWB benchmarks for K, KT and S4 do not contain TBox axioms, so they do not display the EXPTIME behaviour of ALC. We therefore used the TBox problem set from the DL98 benchmarks as shown in Table 4. For many problems, there is no significant difference between the Unsophisticated and Sophisticated methods. For k.d4_p, both methods of global caching significantly outperform mixed caching, and for k.lin_n, GC-DFS beats MC. For the other problems, the best of the two global caching methods is on par with mixed caching.

Galen Test. We evaluated mixed caching and global caching on the “galen1.tkb” problem from the DL’98 benchmarks which contains multiple modalities. We found that MC and both types of global caching took around 20 seconds. Moreover, MC accessed about 3 times as many nodes as GC-NonDFS.

Conclusions and Further Work. It is clear that global caching is a promising optimisation which deserves further investigation since it is on par or better than the other caching methods on all our tests except s4.grz_n and s4.t4p_n. In particular, it is vital to investigate good heuristics for GC-NonDFS, since it is not tied to the DFS framework.

We observed that the best global caching method beats MC in most cases. This may be because MC spends more time per node since it must look up both the `unsat`- and `sat`-caches for every node. As noted earlier, further work

KT Tests: highest problem complexity solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFSGC
kt_45_n	6 6 7 8 9	6 6 7 8 9	21	19 21	21
kt_45_p	4 5 5 6 6	4 5 5 6 6	21	21	21
kt_branch_n	7 8 9 10 11	7 8 9 10 11	8 8 9 10 11	7 8 9 10 11	8 8 9 10 11
kt_branch_p	21	21	21	21	21
kt_dum_n	9 11 11 13 15	9 11 11 13 14	21	21	21
kt_dum_p	21 21 21 21 21	14 15 15 17 17	21	21	21
kt_grz_n	21	21	21	21	21
kt_grz_p	21	21	21	21	21
kt_md_n	4 4 4 4 4	4 4 4 4 4	6 6 6 6 6	6 6 6 6 7	6 6 6 6 7
kt_md_p	3 3 4 4 4	3 4 4 4 4	4 4 4 4 4	4 4 4 4 4	4 4 4 4 4
kt_path_n	2 2 2 2 2	2 2 2 3 3	2 2 3 3 3	2 2 3 3 3	2 3 3 3 4
kt_path_p	3 3 3 3 3	3 3 3 3 3	3 3 3 3 3	3 3 3 3 4	3 3 4 4 4
kt_ph_n	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7
kt_ph_p	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6
kt_poly_n	1 2 2 2 2	1 2 2 2 2	3 3 4 4 5	4 4 4 6 6	3 4 5 6 7
kt_poly_p	7 8 10 12 15	7 8 11 13 15	7 8 11 13 15	7 8 10 13 15	7 8 10 13 15
kt_t4p_n	1 1 2 2 2	1 1 2 2 2	2 3 4 5 8	3 4 6 8 12	4 5 7 10 14
kt_t4p_p	2 2 3 4 4	3 3 4 5 5	7 9 13 19 21	9 12 18 21	7 9 14 19 21
blank count	12	12	28	28	28

Table 2. Results for KT Tests

is required to make the `sat`-cache of MC more efficient when descending into (\sqcup)-branches, by implementing lazy saving for example.

References

1. F Donini and F Massacci. EXPTIME tableaux for \mathcal{ALC} . *AIJ*, 124:87–138, 2000.
2. R Goré and L A Nguyen. Exptime tableaux for ALC using sound global caching. In *Proc. DL2007*,. <http://dl.kr.org/dl2007/>, June 2007.
3. A Heuerding and S Schwendimann. A benchmark method for the propositional logics K, KT, S4. Technical report, Universitaet Bern, Switzerland, 1996.
4. I Horrocks and P F Patel-Schneider. Optimizing description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.
5. D Tsarkov, I Horrocks, and P F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *JAR*, 39(3):277–316, 2007.

S4 Tests: highest problem complexity solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFS GC
s4_45_n	12 14 18 21	11 14 18 21	13 17 21	11 14 19 21	12 15 19 21
s4_45_p	11 12 15 20 21	11 12 15 19 21	14 18 21	15 19 21	14 19 21
s4_branch_n	7 8 9 10 10	7 8 9 10 10	7 8 9 10 10	7 8 9 10 10	8 8 9 10 10
s4_branch_p	21	21	21	21	21
s4_grz_n	21	21	21	8 9 11 13 14	7 8 10 11 13
s4_grz_p	21	21	21	21	21
s4_ipc_n	4 5 5 5 5	4 5 5 5 5	7 7 7 8 8	6 6 7 8 8	6 6 7 8 8
s4_ipc_p	18 21	18 21	18 21	19 21	17 21
s4_md_n	6 6 6 6 6	6 6 6 6 7	7 8 8 9 9	7 8 8 9 9	7 8 8 9 9
s4_md_p	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8
s4_path_n	1 1 1 1 1	1 1 1 1 1	2 2 2 3 3	1 2 2 2 2	1 2 2 2 2
s4_path_p	2 2 2 2 2	2 2 2 2 2	3 3 3 3 3	2 2 3 3 3	2 3 3 3 3
s4_ph_n	4 4 4 4 4	4 4 4 4 4	5 5 5 5 6	3 3 3 3 3	3 3 3 3 3
s4_ph_p	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6
s4_s5_n	3 3 3 4 5	3 3 3 4 5	4 4 5 6 6	3 4 5 5 6	4 4 5 6 7
s4_s5_p	21	21	21	21	21
s4_t4p_n	1 2 2 3 4	2 3 5 6 8	4 6 9 12 15	2 4 5 7 10	2 4 5 7 9
s4_t4p_p	3 4 5 6 7	7 9 12 15 20	9 13 17 21	10 13 18 21	9 13 17 21
blank count	20	20	24	19	19

Table 3. Results for S4 Tests

K TBox Tests: highest complexity problem solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFS GC
k_branch_n	3 3 4 5 5	3 4 4 5 5	3 4 4 5 5	3 4 4 5 5	3 4 4 5 5
k_branch_p	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7
k_d4_n	2 3 3 3 4	2 3 3 3 4	3 3 3 4 4	3 3 3 4 5	2 3 3 3 4
k_d4_p	6 6 6 7 7	4 5 6 8 10	4 5 6 8 10	16 18 21	14 19 21
k_dum_n	7 8 9 9 10	7 8 9 10 10	17 21	18 21	16 21
k_dum_p	21	21	21	21	21
k_grz_n	9 12 15 20 21	11 12 15 20 21	11 14 17 21	11 13 17 21	10 13 17 21
k_grz_p	3 6 6 9 11	4 9 11 13 15	4 9 11 13 15	7 9 11 14 15	6 7 10 10 14
k_lin_n	5 7 9 12 16	5 5 6 8 10	5 5 6 8 10	5 6 8 12 16	4 4 5 5 6
k_lin_p	3 5 5 6 6	3 4 5 6 6	3 4 5 6 6	5 5 5 8 8	5 5 5 6 8
k_path_n	2 3 3 3 4	3 3 4 4 5	5 6 6 8 12	6 7 8 11 13	5 6 8 10 13
k_path_p	4 4 4 4 6	4 4 6 6 7	4 6 6 7 9	5 7 8 10 13	5 7 8 10 13
k_ph_n	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7	5 5 6 6 7
k_ph_p	3 3 3 3 3	3 3 4 4 4	3 3 4 4 4	3 3 4 4 4	3 3 4 4 4
k_poly_n	6 8 8 11 13	6 8 9 10 13	11 15 19 21	11 14 18 21	12 14 18 21
k_poly_p	16 19 21	13 19 21	16 19 21	16 19 21	16 19 21
k_t4p_n	2 2 3 5 7	2 3 4 5 7	5 7 10 13 19	4 7 9 13 18	5 6 9 13 19
k_t4p_p	5 7 10 13 21	2 4 7 8 12	5 6 10 15 21	7 8 9 11 17	5 5 7 12 17
blank count	6	6	11	13	13

Table 4. Results for K TBox Tests