Extended Caching and Backjumping for Expressive Description Logics

Andreas Steigmiller[1], Thorsten Liebig[2], and Birte Glimm[1]

[1] Ulm University, Ulm, Germany, <first name>.<last name>@uni-ulm.de
[2] derivo GmbH, Ulm, Germany, liebig@derivo.de

1 Motivation

Due to the wide range of modelling constructs supported by the expressive DL $\mathcal{SROIQ}$, the typically used tableau algorithms in competitive reasoning systems such as FaCT++ [16], HermiT,[3] or Pellet [14] have a very high worst-case complexity. The development of tableau optimisations that help to achieve practical efficiency is, therefore, a long-standing challenge in DL research (see, e.g., [11, 17]). A very effective and widely implemented optimisation technique is "caching", where one caches, for a set of concepts, whether they are known to be, or can safely be assumed to be, satisfiable or unsatisfiable [4]. If the set of concepts appears again in a model abstraction, then a cache-lookup allows for skipping further applications of tableau rules. Unfortunately, with increasing expressivity naively caching become unsound, for instance, due to the possible interaction of inverse roles with universal restrictions [1, Chapter 9].

With this contribution we push the boundary of the caching optimisation to the expressive DL $\mathcal{SROIQ}$. The developed unsatisfiability caching method is based on a sophisticated dependency management, which further enables better informed tableau backtracking and more efficient pruning (Section 3). Our techniques are grounded in the widely implemented tableau calculus for $\mathcal{SROIQ}$ [9], which makes it easy to transfer our results into existing implementations. The optimisations are integrated within a novel reasoning system, called Konclude [13]. Our empirical evaluation shows that the proposed optimisations result in significant performance improvements (Section 4).

2 Preliminaries

Model construction calculi, such as tableau, decide the consistency of a knowledge base $\mathcal{K}$ by trying to construct an abstraction of a model for $\mathcal{K}$, a so-called "completion graph". A completion graph G is a tuple $(V, E, \mathcal{L}, \dot{\neq})$, where each node $x \in V$ represents one or more individuals, and is labelled with a set of concepts, $\mathcal{L}(x)$, which the individuals represented by x are instances of; each edge $\langle x, y \rangle$ represents one or more pairs of individuals, and is labelled with a set of roles, $\mathcal{L}(\langle x, y \rangle)$, which the pairs of individuals represented by $\langle x, y \rangle$ are instances of. The relation $\dot{\neq}$ records inequalities, which must hold between nodes, e.g., due to at-least cardinality restrictions.

[3] http://www.hermit-reasoner.com

The algorithm works by initialising the graph with one node for each Abox individual/nominal in the input KB, and using a set of expansion rules to syntactically decompose concepts in node labels. Each such rule application can add new concepts to node labels and/or new nodes and edges to the completion graph, thereby explicating the structure of a model. The rules are repeatedly applied until either the graph is fully expanded (no more rules are applicable), in which case the graph can be used to construct a model that is a *witness* to the consistency of $\mathcal{K}$, or an obvious contradiction (called a *clash*) is discovered (e.g., both C and $\neg C$ in a node label), proving that the completion graph does not correspond to a model. The input knowledge base $\mathcal{K}$ is *consistent* if the rules (some of which are non-deterministic) can be applied such that they build a fully expanded, clash free completion graph. A cycle detection technique called *blocking* ensures the termination of the algorithm.

2.1 Dependency Tracking

Dependency tracking keeps track of all dependencies that cause the existence of concepts in node labels, roles in edge labels as well as accompanying constrains such as inequalities that must hold between nodes. Dependencies are associated with so-called *facts*, defined as follows:

Definition 1 (Fact) *We say that G contains a* concept fact $C(x)$ *if* $x \in V$ *and* $C \in \mathcal{L}(x)$*, G contains a* role fact $r(x, y)$ *if* $\langle x, y\rangle \in E$ *and* $r \in \mathcal{L}(\langle x, y\rangle)$*, and G contains an* inequality fact $x \neq y$ *if* $x, y \in V$ *and* $(x, y) \in \neq$*. We denote the set of all (concept, role, or inequality) facts in G as* Facts_G*.*

Dependencies now relate facts in a completion graph to the facts that caused their existence. Additionally, we annotate these relations with a running index, called dependency number, and a branching tag to track non-deterministic expansions:

Definition 2 (Dependency) *Let d be a pair in* $\mathsf{Facts}_G \times \mathsf{Facts}_G$*. A* dependency *is of the form* $d^{n,b}$ *with* $n \in \mathbf{N}_0$ *a* dependency number *and* $b \in \mathbf{N}_0$ *a* branching tag.

We inductively define the dependencies Dep_G for *G: If G is an initial completion graph, then* $\mathsf{Dep}_G = \emptyset$*. We initialise the beginning for the next dependency numbers* n_m *with* 1 *if* $\mathsf{Dep}_G = \emptyset$*; otherwise,* $n_m = 1 + max\{n \mid d^{n,b} \in \mathsf{Dep}_G\}$*. Let R be a tableau rule applicable to a completion graph G. If R is non-deterministic, the next non-deterministic branching tag* b_R *for R is* $1 + max\{\{0\} \cup \{b \mid d^{n,b} \in \mathsf{Dep}_G\}\}$*; for R deterministic,* $b_R = 0$*. Let G′ be the completion graph obtained from G by applying R with* $c_0, \ldots, c_k$ *the facts to satisfy the preconditions of R and* $c'_0, \ldots, c'_\ell$ *the newly added facts in G′, then*

$$\mathsf{Dep}_{G'} = \mathsf{Dep}_G \cup \{(c'_j, c_i)^{n,b} \mid 0 \leq i \leq k, 0 \leq j \leq \ell, n = n_m + (j * k) + i, \\ b = max\{\{b_R\} \cup \{b' \mid (c_i, c)^{n',b'} \in \mathsf{Dep}_G\}\}\}.$$

The branching tag indicates which facts were added non-deterministically:

Definition 3 (Non-deterministic Dependency) *For* $d^{n,b} \in \mathsf{Dep}_G$ *with* $d = (c_1, c_2)$*, let* $D_d = \{(c_2, c_3)^{n',b'} \mid (c_2, c_3)^{n',b'} \in \mathsf{Dep}_G\}$*. The dependency* $d^{n,b}$ *is a* non-deterministic dependency *in G if* $b > 0$ *and either* $D_d = \emptyset$ *or* $max\{b' \mid (c, c')^{n',b'} \in D_d\} < b$*.*

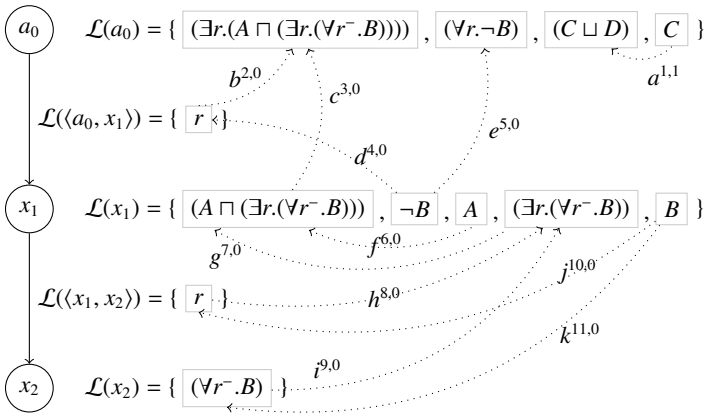

Fig. 1. Tracked dependencies for all facts in the generated completion graph

Figure 1 illustrates a completion graph obtained in the course of testing the consistency of a knowledge base with three concept assertions:

$$a_0 : (\exists r.(A \sqcap (\exists r.(\forall r^-.B)))) \qquad a_0 : (\forall r.\neg B) \qquad a_0 : (C \sqcup D).$$

Thus, the completion graph is initialised with the node a_0, which has the three concepts in its label. Initially, the set of dependencies is empty. For the concepts and roles added by the application of tableau rules, the dependencies are shown with dotted lines, labelled with the dependency. The dependency number increases with every new dependency. The branching tag is only non-zero for the non-deterministic addition of C to the label of a_0 in order to satisfy the disjunction $(C \sqcup D)$. Note the presence of a clash due to B and $\neg B$ in the label of x_1.

3 Extended Caching and Backtracking

In the following we introduce improvements to caching and backjumping by presenting a more informed dependency directed backtracking strategy that also allows for extracting precise unsatisfiability cache entries.

3.1 Dependency Directed Backtracking

Dependency directed backtracking is an optimisation that can effectively prune irrelevant alternatives of non-deterministic branching decisions. If branching points are not involved in clashes, it will not be necessary to compute any more alternatives of these branching points, because the other alternatives cannot eliminate the cause of the clash. To identify involved non-deterministic branching points, all facts in a completion graph are labelled with information about the branching points they depend on. Thus, the united information of all clashed facts can be used to identify involved branching points. A typical realisation of dependency directed backtracking is backjumping [1, 17], where the dependent branching points are collected in the dependency sets for all facts.

3.2 Unsatisfiability Caching

Another widely used technique to increase the performance of a tableau implementation is caching. For *unsatisfiability caching*, one caches sets of concepts that are known to be unsatisfiable. For such a cache entry, it holds that any *superset* is also unsatisfiable. Thus, if, in a future tableau expansion, one encounters a node label that is a superset of a cache entry, one can stop expanding the branch.

Analogously to unsatisfiability caching, one can define satisfiability caching and many systems combine both caches. We focus here, however, on unsatisfiability caching since the two problems are quite different in nature and the required data structures for an efficient cache retrieval can differ significantly. Before we define how and when we create cache entries, we formalise our notion of an unsatisfiability cache.

Definition 4 (Unsatisfiability Cache) *Let $\mathcal{K}$ be a knowledge base and $\mathsf{Con}_\mathcal{K}$ the set of (sub-)concepts that occur in $\mathcal{K}$. An* unsatisfiability cache $\mathsf{UC}_\mathcal{K}$ *for $\mathcal{K}$ is a subset of $2^{\mathsf{Con}_\mathcal{K}}$ such that each* cache entry $S \in \mathsf{UC}_\mathcal{K}$ *is unsatisfiable w.r.t. $\mathcal{K}$. An* unsatisfiability retrieval *for $\mathsf{UC}_\mathcal{K}$ and a completion graph G for $\mathcal{K}$ takes a set of concepts $S \subseteq \mathsf{Con}_\mathcal{K}$ from a node label of G as input. If $\mathsf{UC}_\mathcal{K}$ contains a set $S_\bot \subseteq S$, then $S_\bot$ is returned; otherwise, the empty set is returned.*

Although node labels can have many concepts that are not involved in any clashes, most implementations cache complete node labels. One reason for this might be that the often used backjumping [1, 17] only allows the identification of all branching points involved in a clash, but there is no information about how the clash is exactly caused. We refer to this form of caching as *label caching*. Systems that use label caching typically also only check whether the exact node label from the current tableau is in the cache.

The creation of cache entries rapidly becomes difficult with increasing expressivity of the used DL. Already with blocking for the DL $\mathcal{ALC}$, one can easily generate invalid cache entries [6]. Apart from a node x with a clash in its label, the question is which other node labels are also unsatisfiable. For $\mathcal{ALC}$, this is the case for all labels from x up to the ancestor y with the last non-deterministic expansion. With $\mathcal{ALCI}$, a non-deterministic rule application on a descendant node of x can be involved in the clash, which makes it difficult to determine node labels that can be cached. Nevertheless, caching techniques for $\mathcal{ALCI}$ have been proposed [2, 3, 5], but the difficulty further increases in the presence of nominals and, to the best of our knowledge, the problem of caching with inverses and nominals has not yet been addressed in the literature. In order to avoid unsound results, current systems often deactivate caching in presence of inverse roles or at least in presence of nominals, especially with combined satisfiability and unsatisfiability caching [14, 17].

The extraction of a small still unsatisfiable subset of a node label would yield better cache entries. The use of subset retrieval methods for the cache further increases the number of cache hits. We call such a technique *precise caching*. Although techniques to realise efficient subset retrieval exist [8], unsatisfiability caches that use such subset retrieval are only implemented in very few DL reasoners [7].

Going back to the example in Figure 1, for the node x_1 the set $\{\neg B, (\exists r.(\forall r^-.B))\}$ could be inserted into the cache as well as $\{\neg B, (A \sqcap (\exists r.(\forall r^-.B)))\}$. The number of cache

entries should, however, be kept small, because the performance of the retrieval decreases with an increasing number of entries. Thus, the insertion of concepts for which the rule application is cheap (e.g., concept conjunction) should be avoided. Concepts that require the application of non-deterministic or generating rules are more suitable, because the extra effort of querying the unsatisfiability cache before the rule application can be worth the effort. Optimising cache retrievals for incremental changes further helps to efficiently handle multiple retrievals for the same node with identical or slightly extended concept labels.

The creation of new unsatisfiability cache entries based on dependency tracking can be done during backtracing, which is also coupled with the dependency directed backtracking as described next.

3.3 Dependency Backtracing

The dependency tracking defined in Section 2.1 completely retains all necessary information to exactly trace back the cause of the clash. Thus, this *backtracing* is qualified to identify all involved non-deterministic branching points for the dependency directed backtracking and also to identify small unsatisfiable sets of concepts that can be used to create new unsatisfiability cache entries.

Algorithm 1 performs the backtracing of facts and their tracked dependencies in the presence of inverse roles and nominals. If all facts and their dependencies are collected on the same node while backtracing, an unsatisfiability cache entry with these facts can be generated, assuming all facts are concept facts. As long as no nominal or Abox individual occurs in the backtracing, the unsatisfiability cache entries can also be generated while all concept facts have the same node depth. Thus, an important task of the backtracing algorithm is to hold as many facts as possible within the same node depth to allow for the generation of many cache entries. To realise the backtracing, we introduce the following data structure:

Definition 5 (Fact Dependency Node Tuple) *A* fact dependency node tuple for G *is a triple* $\langle c, d^{n,b}, x\rangle$ *with* $c \in \mathsf{Facts}_G, d^{n,b} \in \mathsf{Dep}_G$ *and* $x \in V$. *As abbreviation we also write* $\langle C, d^{n,b}, x\rangle$ *if* c *is the concept fact* $C(x)$.

If a clash is discovered in the completion graph, a set of fact dependency node tuples is generated for the backtracing. Each tuple consists of a fact involved in the clash, an associated dependency and the node where the clash occurred. The algorithm gets this set T of tuples as input and incrementally traces the facts back from the node with the clash to nodes with depth 0 (Abox individuals or root nodes).

In each loop round (line 3) some tuples of T are exchanged with tuples, whose facts are the cause of the exchanged one. To identify which tuple has to be traced back first, the current minimum node depth (line 4) and the maximum branching tag (line 5) are extracted from the tuples of T. All tuples, whose facts are located on a deeper node and whose dependencies are deterministic, are collected in the set A. Such tuples will be directly traced back until their facts reach the current minimum node depth (line 10-12). If there are no more tuples on deeper nodes with deterministic dependencies, i.e., $A = \emptyset$, the remaining tuples from deeper nodes with non-deterministic dependencies

Algorithm 1 Backtracing Algorithm

Require: A set of fact dependency node tuples T obtained from clashes

procedure DEPENDENCYBACKTRACING(T)
  $pendingUnsatCaching \leftarrow false$
  loop
    $min_D \leftarrow$MINIMUMNODEDEPTH(T)
    $max_B \leftarrow$MAXIMUMBRANCHINGTAG(T)
    $A \leftarrow \{t \in T \mid$ NODEDEPTH(t)$> min_D \wedge$ HASDETERMINISTICDEPENDENCY(t)$\}$
    $C \leftarrow \emptyset$
    if $A \neq \emptyset$ **then**
      $pendingUnsatCaching \leftarrow true$
      for all $t \in A$ **do**
        $T \leftarrow (T \setminus t) \cup$ GETCAUSETUPLESBYDEPENDENCY(t)
      end for
    else
      $B \leftarrow \{t \in T \mid$ NODEDEPTH(t)$> min_D \wedge$ BRANCHINGTAG(t)$= max_B\}$
      if $B = \emptyset$ **then**
        if $pendingUnsatCaching = true$ **then**
          $pendingUnsatCaching \leftarrow$TRYCREATEUNSATCACHEENTRY(T)
        end if
        if HASNODEPENDENCY(t) **for all** $t \in T$ **then**
          $pendingUnsatCaching \leftarrow$TRYCREATEUNSATCACHEENTRY(T)
          return
        end if
        $C \leftarrow \{t \in T \mid$ BRANCHINGTAG(t)$= max_B\}$
      end if
      $t \leftarrow$ANYELEMENT($B \cup C$)
      if HASDETERMINISTICDEPENDENCY(t) **then**
        $T \leftarrow (T \setminus t) \cup$ GETCAUSETUPLESBYDEPENDENCY(t)
      else
        $b \leftarrow$GETNONDETERMINISTICBRANCHINGPOINT(t)
        if ALLALTERNATIVESOFNONDETBRANCHINGPOINTPROCESSED(b) **then**
          $T \leftarrow T \cup$ LOADTUPLESFROMNONDETBRANCHINGPOINT(b)
          $T \leftarrow (T \setminus t) \cup$ GETCAUSETUPLESBYDEPENDENCY(t)
          $T \leftarrow$BACKTRACETUPLESBEFOREBRANCHINGPOINT(T, b)
          $pendingUnsatCaching \leftarrow$TRYCREATEUNSATCACHEENTRY(T)
        else
          $T \leftarrow$BACKTRACETUPLESBEFOREBRANCHINGPOINT(T, b)
          SAVETUPLESTONONDETBRANCHINGPOINT(T, b)
          JUMPBACKTO(max_B)
          return
        end if
      end if
    end if
  end loop
end procedure

and the current branching tag are copied into B (line 14) in the next round. If B is not empty, one of these tuples (line 25) and the corresponding non-deterministic branching point (line 29) are processed. The backtracing is only continued, if all alternatives of the branching point are computed as unsatisfiable. In this case, all tuples, saved from the backtracing of other unsatisfiable alternatives, are added to T (line 31). Moreover, for c the concept fact in the tuple t, t can be replaced with tuples for the fact on which c non-deterministically depends (line 32).

For a possible unsatisfiability cache entry all remaining tuples, which also depend on the non-deterministic branching point, have to be traced back until there are no tuples with facts of some alternatives of this branching point left (line 33). An unsatisfiability cache entry is only generated (line 34), if all facts in T are concept facts for the same node or on the same node depth.

Unprocessed alternatives of a non-deterministic branching point have to be computed before the backtracing can be continued. It is, therefore, ensured that tuples do not consist of facts and dependencies from this alternative, which also allows for releasing memory (line 36). The tuples are saved to the branching point (line 37) and the algorithm jumps back to an unprocessed alternative (line 38).

If B is also empty, but there are still dependencies to previous facts, some tuples based on the current branching tag have to remain on the current minimum node depth. These tuples are collected in the set C (line 23) and are processed separately one per loop round, similar to the tuples of B, because the minimum node depth or maximum branching tag may change. The tuples of C can have deterministic dependencies, which are processed like the tuples of A (line 27). If all tuples have no more dependencies to previous facts, the algorithm terminates (line 21).

Besides the creation of unsatisfiability cache entries after non-deterministic dependencies (line 34), cache entries may also be generated when switching from a deeper node to the current minimum node depth in the backtracing (line 9 and 17) or when the backtracing finishes (line 20). The function that tries to create new unsatisfiability cache entries (line 17, 20, and 34) returns a Boolean flag that indicates whether the attempt has failed, so that the attempt can be repeated later.

For an example, we consider the clash $\{\neg B, B\}$ in the completion graph of Figure 1. The initial set of tuples for the backtracing is T_1 (see Figure 2). Thus, the minimum node depth for T_1 is 1 and the maximum branching tag is 0. Because there are no tuples on a deeper node, the sets A and B are empty for T_1. Since all clashed facts are generated deterministically, the dependencies of the tuples have the current maximum branching tag 0 and are all collected into the set C. The backtracing continues with one tuple t from C, say $t = \langle B, k^{11,0}, x_1 \rangle$. The dependency k of t relates to the fact $(\forall r^-.B)(x_2)$, which is a part of the cause and replaces the backtraced tuple t in T_1. The resulting set T_2 is used in the next loop round. The minimum node depth and the maximum branching tag remain unchanged, but the new tuple has a deeper node depth and is traced back with a higher priority to enable unsatisfiability caching again. Thus, $\langle (\forall r^-.B), i^{9,0}, x_2 \rangle$ is added to the set A and then replaced by its cause, leading to T_3. Additionally, a pending creation of an unsatisfiability cache entry is noted, which is attempted in the third loop round since A and B are empty. The creation of a cache entry is, however, not yet sensible and deferred since T_3 still contains an atomic clash. Let $t = \langle B, j^{10,0}, x_1 \rangle \in C$ be the

$$
\begin{aligned}
T_1 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle \neg B, e^{5,0}, x_1\rangle, \langle B, j^{10,0}, x_1\rangle, \langle B, k^{11,0}, x_1\rangle\}\\
&\downarrow\\
T_2 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle \neg B, e^{5,0}, x_1\rangle, \langle B, j^{10,0}, x_1\rangle, \langle (\forall r^{-}.B), i^{9,0}, x_2\rangle\}\\
&\downarrow\\
T_3 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle \neg B, e^{5,0}, x_1\rangle, \langle B, j^{10,0}, x_1\rangle, \langle (\exists r.(\forall r^{-}.B)), g^{7,0}, x_1\rangle\}\\
&\downarrow\\
T_4 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle \neg B, e^{5,0}, x_1\rangle, \langle r(x_1, x_2), h^{8,0}, x_1\rangle, \langle (\exists r.(\forall r^{-}.B)), g^{7,0}, x_1\rangle\}\\
&\downarrow\\
T_5 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle \neg B, e^{5,0}, x_1\rangle, \langle (\exists r.(\forall r^{-}.B)), g^{7,0}, x_1\rangle\}\\
&\downarrow\\
T_6 &= \{\langle \neg B, d^{4,0}, x_1\rangle, \langle (\forall r.\neg B), -, a_0\rangle, \langle (\exists r.(\forall r^{-}.B)), g^{7,0}, x_1\rangle\}\\
&\downarrow\\
T_7 &= \{\langle r(a_0, x_1), b^{2,0}, x_1\rangle, \langle (\forall r.\neg B), -, a_0\rangle, \langle (A \sqcap (\exists r.(\forall r^{-}.B))), c^{3,0}, x_1\rangle\}\\
&\downarrow\\
T_8 &= \{\langle (\exists r.(A \sqcap (\exists r.(\forall r^{-}.B)))) -, a_0\rangle, \langle (\forall r.\neg B), -, a_0\rangle\}
\end{aligned}
$$

Fig. 2. Backtracing sequence of tuples as triggered by the clash of Figure 1

tuple from T_3 that is traced back next. In the fourth round, the creation of a cache entry is attempted again, but fails because not all facts are concepts facts. The backtracing of $\langle r(x_1, x_2), h^{8,0}, x_1\rangle$ then leads to T_5. In the following round an unsatisfiability cache entry is successfully created for the set $\{\neg B, (\exists r.(\forall r^{-}.B))\}$. Assuming that now the tuple $\langle \neg B, e^{5,0}, x_1\rangle$ is traced back, we obtain T_6, which includes the node a_0. Thus, the minimum node depth changes from 1 to 0. Two more rounds are required until T_8 is reached. Since all remaining facts in T_8 are concept assertions, no further backtracing is possible and an additional cache entry is generated for the set $\{(\exists r.(A \sqcap (\exists r.(\forall r^{-}.B)))), (\forall r.\neg B)\}$.

If a tuple with a dependency to node a_0 had been traced back first, it would have been possible that the first unsatisfiability cache entry for the set $\{\neg B, (\exists r.(\forall r^{-}.B))\}$ was not generated. In general, it is not guaranteed that an unsatisfiability cache entry is generated for the node where the clash is discovered if there is no non-deterministic rule application and if the node is not a root node or an Abox individual. Furthermore, if there are facts that are not concept facts, these can be backtraced with higher priority, analogous to the elements of the set A, to make unsatisfiability cache entries possible again. To reduce the repeated backtracing of identical tuples in different rounds, an additional set can be used to store processed tuples for the alternative for which the backtracing is performed.

The backtracing can also be performed over nominal and Abox individual nodes. However, since Abox and absorbed nominal assertions such as $\{a\} \sqsubseteq C$ have no previous dependencies, this can lead to a distributed backtracing stuck on different nodes. In this case, no unsatisfiability cache entries are possible.

A less precise caching can lead to an adverse interaction with dependency directed backtracking. Consider the example of Figure 3, where the satisfiability of the combination of the concepts $(\exists r.(\exists s.(A \sqcap B)))$, $((C_1 \sqcap \forall r.C) \sqcup (C_2 \sqcap \forall r.C))$, and $((D_1 \sqcap \forall r.(\forall s.\neg A)) \sqcup (D_2 \sqcap \forall r.(\forall s.\neg A)))$ is tested. Note that, in order to keep the figure readable, we no longer show complete dependencies, but only the branching points for non-deterministic decisions. First, the two disjunctions are processed. Assuming that the alternative with the disjuncts $(C_1 \sqcap \forall r.C)$ and $(D_1 \sqcap \forall r.(\forall s.\neg A))$ is considered first (shown on the left-hand side of Figure 3), an r-successor x_1 with label $\{(\exists s.(A \sqcap B)), C^1, (\forall s.\neg A)^2\}$ is generated. The branching points indicate which concepts depend on which non-deterministic

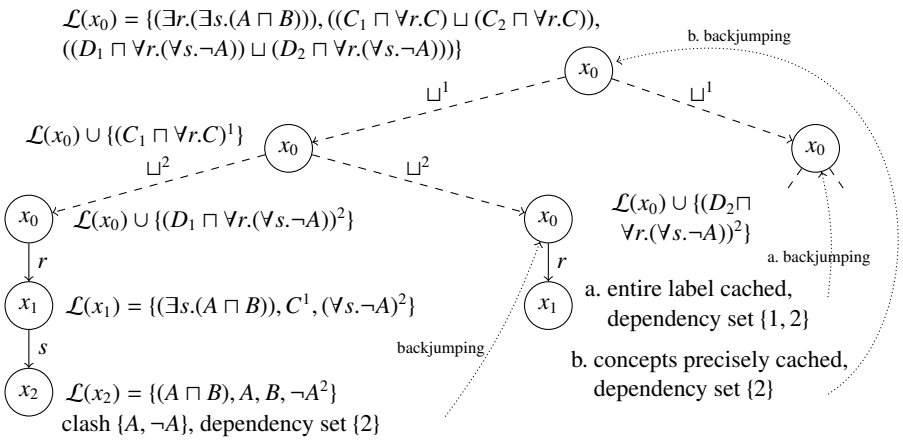

Fig. 3. More pruned alternatives due to dependency directed backtracking and precise caching (case b.) in contrast to label caching (case a.)

decision. For example, C is in $\mathcal{L}(x_1)$ due to the disjunct $(C_1 \sqcap \forall r.C)$ of the first non-deterministic branching decision (illustrated in Figure 3 with the superscript 1). In the further generated s-successor x_2 a clash is discovered. For the only involved non-deterministic branching point 2, we have to compute the second alternative. Thus, an identical r-successor x_1 is generated again for which we can discover the unsatisfiability with a cache retrieval. If the entire label of x_1 was inserted to the cache, the dependent branching points of all concepts in the newly generated node x_1 would have to be considered for further dependency directed backtracking. Thus, the second alternative of the first branching decision also has to be evaluated (c.f. Figure 3, case a., label caching). In contrast, if the caching was more precise and only the combination of the concepts $(\exists s.(A \sqcap B))$ and $(\forall s.\neg A)$ was inserted into the unsatisfiability cache, the cache retrieval for the label of node x_1 would return the inserted subset. Thus, only the dependencies associated to the concepts of the subset could be used for further backjumping, whereby it would not be necessary to evaluate the remaining alternatives (c.f. Figure 3, case b., precise caching).

4 Evaluation

Our Konclude reasoning system implements the enhanced optimisation techniques for $\mathcal{SROIQ}$ described above. We evaluate dependency directed backtracking and unsatisfiability caching with the help of concept satisfiability tests from the well-known DL 98 benchmark suite [10] and spot tests from [12]. An extended evaluation and a comparison of Konclude with other reasoners can be found in the accompanying technical report [15]. From the DL 98 suite we selected satisfiable and unsatisfiable test cases (with _n resp. _p postfixes) and omitted those for which unsatisfiability caching is irrelevant and tests that were too easy to serve as meaningful and reproducible sample.

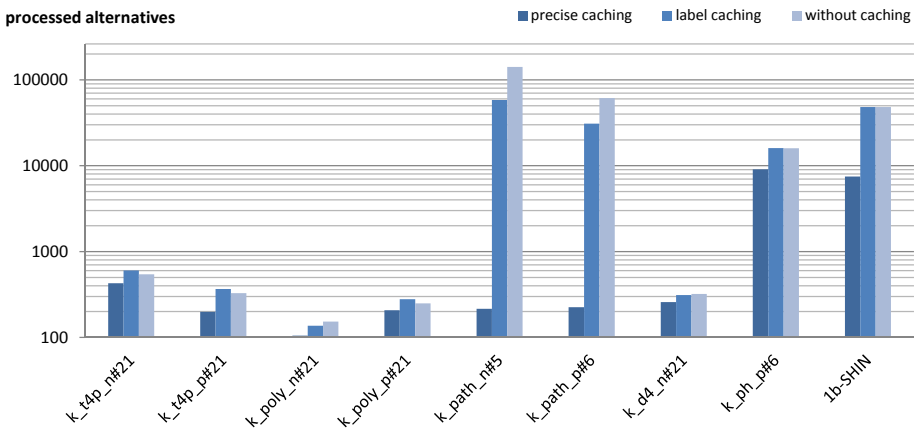

Fig. 4. Log scale comparison of processed alternatives for different caching methods

We distinguish between precise caching and label caching as described in Section 3.2. To recall, precise caching stores precise cache entries consisting of only those backtraced sets of concepts that are explicitly known to cause an unsatisfiability in combination with subset retrieval, while label caching stores and returns only entire node labels.

Konclude implements precise unsatisfiability caching based on hash data structures [8] in order to efficiently facilitate subset cache retrieval. Figure 4 shows the total number of processed non-deterministic alternatives for precise caching, label caching and without caching for a selection of test cases solvable within one minute.

Note that runtime is not a reasonable basis of comparison since the label caching has been implemented (just for the purpose of evaluation) on top of the built-in and computationally more costly precise caching approach. System profiling information, however, strongly indicate that building and querying the precise unsatisfiability cache within Konclude is negligible in terms of execution time compared to the saved processing time for disregarded alternatives. However, we have experienced an increase of memory usage by a worst-case factor of two in case of dependency tracking in comparison to no dependency handling.

Figure 4 reveals that precise caching can, for some test cases, reduce the number of non-deterministic alternatives by two orders of magnitude in comparison to label caching. Particularly the test cases k_path_n/p are practically solvable for Konclude only with precise caching for larger available problem sizes.

5 Conclusions

We have presented an unsatisfiability caching technique that can be used in conjunction with the very expressive DL $\mathcal{SROIQ}$. The presented dependency management allows for more informed backjumping, while also supporting the creation of precise cache unsatisfiability entries. In particular the precise caching approach can reduce the num-

ber of tested non-deterministic branches by up to two orders of magnitude compared to standard caching techniques. The optimisations are well-suited for the integration into existing tableau implementations for $\mathcal{SROIQ}$ and play well with other commonly implemented optimisation techniques.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, second edn. (2007)
2. Ding, Y., Haarslev, V.: Tableau caching for description logics with inverse and transitive roles. In: Proc. 2006 Int. Workshop on Description Logics. pp. 143–149 (2006)
3. Ding, Y., Haarslev, V.: A procedure for description logic $\mathcal{ALCFI}$. In: Proc. 16th European Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'07) (2007)
4. Donini, F.M., Massacci, F.: EXPTIME tableaux for $\mathcal{ALC}$. J. of Artificial Intelligence 124(1), 87–138 (2000)
5. Goré, R., Widmann, F.: Sound global state caching for $\mathcal{ALC}$ with inverse roles. In: Proc. 18th European Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'09). LNCS, vol. 5607, pp. 205–219. Springer (2009)
6. Haarslev, V., Möller, R.: Consistency testing: The RACE experience. In: Proceedings, Automated Reasoning with Analytic. pp. 57–61. Springer-Verlag (2000)
7. Haarslev, V., Möller, R.: High performance reasoning with very large knowledge bases: A practical case study. In: Proc. 17th Int. Joint Conf. on Artificial Intelligence (IJCAI'01). pp. 161–168. Morgan Kaufmann (2001)
8. Hoffmann, J., Koehler, J.: A new method to index and query sets. In: Proc. 16th Int. Conf. on Artificial Intelligence (IJCAI'99). pp. 462–467. Morgan Kaufmann (1999)
9. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06). pp. 57–67. AAAI Press (2006)
10. Horrocks, I., Patel-Schneider, P.F.: DL systems comparison. In: Proc. 1998 Int. Workshop on Description Logics (DL'98). vol. 11, pp. 55–57 (1998)
11. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. J. of Logic and Computation 9(3), 267–293 (1999)
12. Liebig, T.: Reasoning with OWL – system support and insights –. Tech. Rep. TR-2006-04, Ulm University, Ulm, Germany (September 2006)
13. Liebig, T., Steigmiller, A., Noppens, O.: Scalability via parallelization of OWL reasoning. In: Proc. Workshop on New Forms of Reasoning for the Semantic Web: Scalable & Dynamic (NeFoRS'10) (2010)
14. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics 5(2), 51–53 (2007)
15. Steigmiller, A., Liebig, T., Glimm, B.: Extended caching, backjumping and merging for expressive description logics. Tech. Rep. TR-2012-01, Ulm University, Ulm, Germany (2012), available online at http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui/Ulmer_Informatik_Berichte/2012/UIB-2012-01.pdf
16. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR'06). LNCS, vol. 4130, pp. 292–297. Springer (2006)
17. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. of Automated Reasoning 39, 277–316 (2007)