Extended Caching, Backjumping and Merging for Expressive Description Logics

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

[1] Ulm University, Ulm, Germany
firstname.lastname@uni-ulm.de
[2] derivo GmbH, Ulm, Germany
liebig@derivo.de

Abstract. With this contribution we push the boundary of some known optimisations such as caching to the very expressive Description Logic $\mathcal{SROIQ}$. The developed method is based on a sophisticated dependency management and a precise unsatisfiability caching technique, which further enables better informed tableau backtracking and more efficient pruning. Additionally, we optimise the handling of cardinality restrictions, by introducing a strategy called pool-based merging.

We empirically evaluate the proposed optimisations within the novel reasoning system Konclude and show that the proposed optimisations indeed result in significant performance improvements.

1 Motivation

Tableau algorithms are dominantly used in sound and complete reasoning systems, which are able to deal with ontologies specified in the OWL 2 DL ontology language [16]. Such algorithms are usually specified in terms of Description Logics (DLs) [1], which provide the formal basis for OWL, e.g., OWL 2 is based on the DL $\mathcal{SROIQ}$ [11].

To our knowledge, all competitive systems for reasoning with $\mathcal{SROIQ}$ knowledge bases such as FaCT++ [19], HermiT,[1] jFact,[2] or Pellet [17] use a variant of the tableau method – a refutation-based calculus that systematically tries to construct an abstraction of a model for a given query by exhaustive application of so called tableau rules.

Due to the wide range of modelling constructs supported by expressive DLs, the typically used tableau algorithms have a very high worst-case complexity. Developing optimisations to nevertheless allow for highly efficient implementations is, therefore, a long-standing research area in DLs (see, e.g., [13,20]). 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. Caching even allows for implementing worst-case optimal decision procedures for $\mathcal{ALC}$ [6].

Unfortunately, with increasing expressivity some of the widely used optimisations become unsound. For instance, naively caching the satisfiability status of interim

[1] http://www.hermit-reasoner.com
[2] http://jfact.sourceforge.net/

B. Gramlich, D. Miller, and U. Sattler (Eds.): IJCAR 2012, LNAI 7364, pp. 514–529, 2012.

results easily causes unsoundness in the presence of inverse roles due to their possible interactions with universal restrictions [1, Chapter 9]. On the other hand, for features such as cardinality restrictions there are nearly no optimisations yet. An attempt to use algebraic methods [9,5], i.e., by combining a tableau calculus with a procedure to solve systems of linear (in)equations, performs well, but requires significant changes to the calculus and has not (yet) been extended to very expressive DLs such as $\mathcal{SROIQ}$.

Our contribution in this paper is two-fold. We push the boundary of known optimisations, most notably caching, to the expressive DL $\mathcal{SROIQ}$. The developed method is based on a sophisticated dependency management and a precise unsatisfiability caching technique, which further enables better informed tableau backtracking and more efficient pruning (Section 3). In addition we optimise the handling of cardinality restrictions, by introducing a strategy called *pool-based merging* (Section 4). Our techniques are grounded in the widely implemented tableau calculus for $\mathcal{SROIQ}$ [11], which makes it easy to transfer our results into existing tableau implementations. The presented optimisations are implemented within a novel reasoning system, called Konclude [15]. Our empirical evaluation shows that the proposed optimisations result in significant performance improvements (Section 5).

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.

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 $x \dot{\neq} y$ *if $x, y \in V$ and $(x, y) \in \dot{\neq}$. 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 $n \in \mathbb{N}_0$ a* dependency number *and $b \in \mathbb{N}_0$ a* branching tag.

We inductively define the dependencies for G, *written Dep_G. If G is an initial completion graph, then $\mathsf{Dep}_G = \emptyset$. Let R be a tableau rule applicable to a completion graph G with $\{c_0, \ldots, c_k\}$ a minimal set of facts in G that satisfy the preconditions of R. If $\mathsf{Dep}_G = \emptyset$, then $n_m = b_m = 0$, otherwise, let $n_m = max\{n \mid d^{n,b} \in \mathsf{Dep}_G\}$ and $b_m = max\{b \mid d^{n,b} \in \mathsf{Dep}_G\}$. If R is non-deterministic, then $b_R = 1 + b_m$, otherwise $b_R = 0$. Let G' be the completion graph obtained from G by applying R and let $c'_0, \ldots, c'_\ell$ be the newly added facts in G', then*

$$\mathsf{Dep}_{G'} = \mathsf{Dep}_G \cup \{(c'_j, c_i)^{n,b} \mid 0 \leq i \leq k, 0 \leq j \leq \ell, n = n_m + 1 + (j * k) + i,$$
$$b = max\{\{b_R\} \cup \{b' \mid (c_i, c)^{n',b'} \in \mathsf{Dep}_G\}\}\}.$$

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

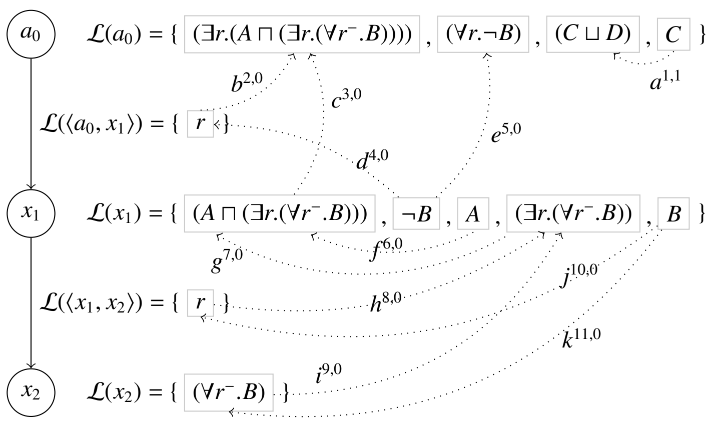

Fig. 1. Tracked dependencies for all facts in the generated completion graph

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

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 [1,20], 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, which comes in two flavours: satisfiability and unsatisfiability caching. For the former, one caches sets of concepts, e.g., from node labels, that are known to be satisfiable. In contrast, for an unsatisfiability cache, we cache sets of concepts that are unsatisfiable. For such a cached set, any *superset* is also unsatisfiable. Thus, one is interested in caching a minimal, unsatisfiable set of concepts. Although the caching of satisfiable and unsatisfiable sets of concepts is often considered together, we focus here on the unsatisfiability caching problem since the two problems are quite different in nature and already the required data structure for an efficient cache retrieval can differ significantly.

Definition 4 (Unsatisfiability Cache). *Let $\mathcal{K}$ be a knowledge base, $\mathsf{Con}_{\mathcal{K}}$ 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 an unsatisfiable set of concepts. 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.*

Deciding when we can safely create a cache entry rapidly becomes difficult with increasing expressivity of the used DL. Already with blocking on tableau-based systems for the DL $\mathcal{ALC}$ care has to be taken to not generate invalid cache entries [7]. There are some approaches for caching with inverse roles [2,3,6], where possible propagations over inverse roles from descendant nodes are taken into account. The difficulty increases further 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 this setting, it is difficult to determine, for a node x with a clash in its label, which nodes (apart from x) are also labelled with unsatisfiable sets of concepts. Without nominals and inverse roles, we can determine the ancestor y of x with the last non-deterministic expansion and consider the labels of all nodes from x up to y as unsatisfiable. With inverse roles, a non-deterministic rule application on a descendant node of x can be involved in the creation of the clash, whereby the node labels that can be cached as unsatisfiable become limited.

In order to demonstrate the difficulties with inverse roles, let us assume that the example in Figure 1 is extended such that $((\forall r^-.B) \sqcup E) \in \mathcal{L}(x_2)$ and that $(\forall r^-.B) \in \mathcal{L}(x_2)$ results from the non-deterministic expansion of the disjunction. For the resulting clash in $\mathcal{L}(x_1)$, it is not longer sufficient to consider only non-deterministic expansions on ancestor nodes. The label of x_2 cannot be cached because some facts ($\neg B$) involved in the clash are located on different nodes (x_1). Furthermore, if trying the disjunct E also leads to a clash, the disjunction $((\forall r^-.B) \sqcup E)$ in $\mathcal{L}(x_2)$ is unsatisfiable in the context of *this* completion graph. Nevertheless, a cache entry cannot be generated because (at least) the first disjunct involves facts of an ancestor node. In order to also handle inverse roles, it would, therefore, be necessary to remember all nodes or at least the minimum node depth involved in the clashes of all alternatives. In the presence of nominals, it further becomes necessary to precisely manage the exact causes of clashes, e.g., via tracking the dependencies as presented in Section 2.1. If such a technique is missing, often the only option is to deactivate caching completely [17,20].

Since node labels can have many concepts that are not involved in any clashes, the precise extraction of a small set of concepts that are in this combination unsatisfiable would yield better entries for the unsatisfiability cache. With an appropriate subset retrieval potentially more similar also unsatisfiable node labels can be found within the cache. We call this technique *precise caching*. Although techniques to realise efficient subset retrieval exist [10], unsatisfiability caches based on this idea are only implemented in very few DL reasoners [8]. Furthermore, the often used backjumping only allows the identification of all branching points involved in a clash, but there is no information about how the clash is exactly caused. As a result, only complete node labels can be saved in the unsatisfiability cache. We refer to this often used form of caching combined with only an equality cache retrieval as *label caching*.

For precise caching, the selection of an as small as possible but still unsatisfiable subset of a label as cache entry should be adjusted to the cache retrieval strategy, i.e., the strategy of when the cache is queried in the tableau algorithm. 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. Basically all facts involved in a clash are backtraced to collect the facts that cause the clash within one node, whereby then an unsatisfiability cache entry can be created.

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 $x \in V$. Abbreviatory 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

Algorithm 1. Backtracing Algorithm

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

```
procedure dependencyBacktracing(T)
    pendingUnsatCaching ← false
    loop
        min_D ← minimumNodeDepth(T)
        max_B ← maximumBranchingTag(T)
        A ← {t ∈ T | nodeDepth(t) > min_D ∧ hasDeterministicDependency(t)}
        C ← ∅
        if A ≠ ∅ then
            pendingUnsatCaching ← true
            for all t ∈ A do
                T ← (T \ t) ∪ getCauseTuplesByDependency(t)
            end for
        else
            B ← {t ∈ T | nodeDepth(t) > min_D ∧ branchingTag(t) = max_B}
            if B = ∅ then
                if pendingUnsatCaching = true then
                    pendingUnsatCaching ← tryCreateUnsatCacheEntry(T)
                end if
                if hasNoDependency(t) for all t ∈ T then
                    pendingUnsatCaching ← tryCreateUnsatCacheEntry(T)
                    return
                end if
                C ← {t ∈ T | branchingTag(t) = max_B}
            end if
            t ← anyElement(B ∪ C)
            if hasDeterministicDependency(t) then
                T ← (T \ t) ∪ getCauseTuplesByDependency(t)
            else
                b ← getNonDeterministicBranchingPoint(t)
                if allAlternativesOfNonDetBranchingPointProcessed(b) then
                    T ← T ∪ loadTuplesFromNonDetBranchingPoint(b)
                    T ← (T \ t) ∪ getCauseTuplesByDependency(t)
                    T ← forceTuplesBeforeBranchingPoint(T, b)
                    pendingUnsatCaching ← tryCreateUnsatCacheEntry(T)
                else
                    T ← forceTuplesBeforeBranchingPoint(T, b)
                    saveTuplesToNonDetBranchingPoint(T, b)
                    jumpBackTo(max_B)
                    return
                end if
            end if
        end if
    end loop
end procedure
```

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 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 fact in 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

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

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

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 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

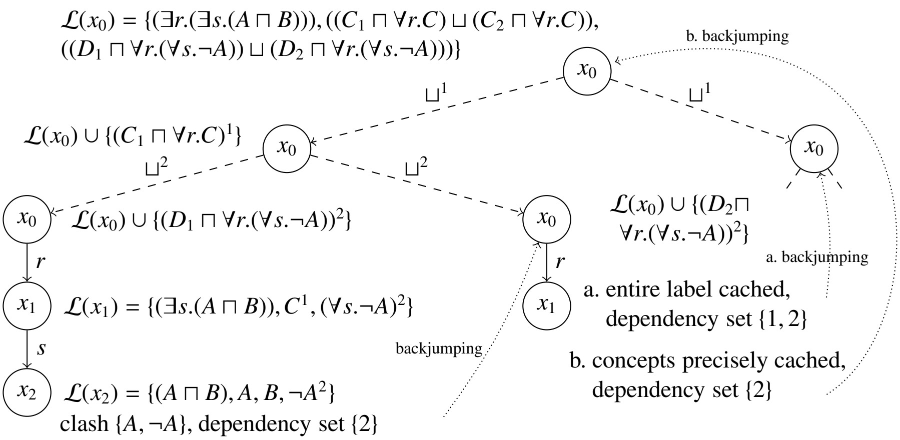

Fig. 3. More pruned alternatives due to dependency directed backtracking and precise caching (b.) in contrast to label caching (a.)

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 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, a.). 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, b.).

4 Optimised Merging

At-most cardinality restrictions require the non-deterministic merging of role neighbours until the cardinality restriction is satisfied. Only for cardinalities of 1, merging is deterministic. The usual merging approach [11], which can still be found in several available reasoner implementations, employs a $\leq$-rule that shrinks the number of role neighbours by one with each rule application. Each such merging step gathers pairs of potentially mergeable neighbouring nodes. For each merging pair a branch is generated in which the merging of the pair is executed. Without optimisations, this approach leads to an inefficient implementation since for merging problems that require more than one merging step, several identical merging combinations have to be evaluated multiple times. Throughout this section, we consider the following example: a node in the completion graph has four r-neighbours w, x, y and z, which have to be merged into two nodes. The naive approach described above leads to eighteen non-deterministic alternatives: in the first of two necessary merging steps there are $\sum_{i=1}^{n-1} i$, i.e., six possible merging pairs. A second merging step is required to reduce the remaining three nodes to two. If the merging rule is applied again without any restrictions, each second merging step generates three more non-deterministic alternatives. However, only seven of these eighteen alternatives overall are really different. For example, the combination wxy, z, where the nodes w, x and y have been merged, can be generated by $merge(merge(w, x), y)$, $merge(merge(w, y), x)$ and $merge(merge(x, y), w)$.

The problem is very similar to the syntactic branching search [1], where unsatisfiable concepts of non-disjoint branches might have to be evaluated multiple times. The semantic branching technique is commonly used to avoid such redundant evaluations and in the merging context an analogous approach can be very beneficial.

In order to apply this technique, all nodes of previously tested merging pairs are set to be pairwise distinct. For example, when merging (w, x) in the first merging step leads to a clash, w and x are set to be distinct because this combination has been tested and should be avoided in future tests. In the second alternative, the nodes w and y are merged, which leads to $wy \dot{\neq} x$. As a result of the inequality, $merge(merge(w, y), x)$ is never tested in the second merging step (Figure 4). If also merging w and y fails, a further inequality $w \dot{\neq} y$ is added. Finally, for the last two alternatives of the first merging step the inequality constraints prevent further merging and show that these alternatives are unsatisfiable. Summing up, with the inequalities the total number of non-deterministic alternatives can be reduced to nine in this example. Unfortunately, similarly sophisticated merging techniques can hardly be found in current reasoners.

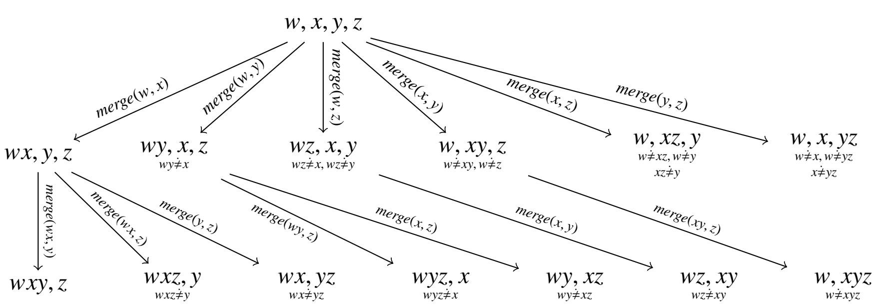

Fig. 4. Non-deterministic merging alternatives with added inequality information

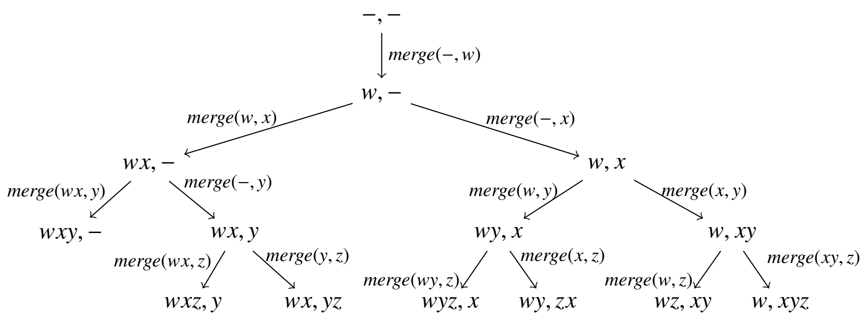

Fig. 5. Pool-based merging approach to avoid redundant evaluation of previous merging attempts

Apart from using the inequality information, the *pool-based merging* method that we propose also prevents the redundant evaluation of previously computed merging attempts. Furthermore it works very well in combination with dependency directed backtracking due to the thin and uniform branching tree.

Regarding the implementation of the pool-based merging method, the nodes that have to be merged are managed in a queue. Each merging step takes the next node from the queue and non-deterministically inserts this node into a so-called *pool*, where the number of pools corresponds to the required cardinality. All pools are considered as distinct and nodes within one pool are merged together. If there are several empty pools, we will only generate one alternative, where the node is inserted in one of these empty pools. If several empty pools were initialised with the same node, once again redundant merging combinations would have to be evaluated. For the example, the generated merging combinations due to the pool based merging procedure are illustrated in Figure 5. At the beginning, all nodes are in the queue and both pools are empty. In the first merging step the node w is taken from the queue and inserted to the first empty pool. In the second step the next node x is non-deterministically inserted into the first pool together with the node w or into another empty pool. This process continues until the cardinality restriction is satisfied. Note that z is not removed from the queue for the alternative shown on the left-hand side since the cardinality is already satisfied. If a clash occurs in an alternative, all relevant merging steps can be identified with the dependency directed backtracking. Different insertion alternatives are, therefore, only tested for nodes that are involved in the clashes. In the worst-case also the pool based merging is systematically testing all possible combinations, but the different generation of these alternatives prevents redundant evaluations. Other tableau expansions rules for $\mathcal{SROIQ}$, such as the *choose*- or the *NN*-rule, are not influenced by the merging method, consequently also qualified cardinality restrictions are supported in combination with the pool based merging.

5 Evaluation

Our Konclude reasoning system implements the enhanced optimisation techniques for $\mathcal{SROIQ}$ described above. In the following, we first compare different caching methods.

Furthermore, we benchmark our pool-based merging technique against the standard pair-based approach that is used in most other systems. A comparison of Konclude with other reasoners can be found in the accompanying technical report [18].

We evaluate dependency directed backtracking and unsatisfiability caching with the help of concept satisfiability tests from the well-known DL 98 benchmark suite [12] and spot tests regarding cardinality restrictions and merging first proposed in [14]. 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.

With respect to caching, 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.

Independent of the caching method, we distinguish between unfiltered and relevant dependencies for further dependency backtracing after a cache hit. *Unfiltered dependency* denotes the backtracing technique that uses all the concept facts and their dependencies within a node label, for which the unsatisfiability has been found in the cache. In contrast, *relevant dependency* uses only those facts and dependencies of a node label for further backtracing that caused the unsatisfiability (as if the unsatisfiability would be found without caching).

Konclude natively maintains relevant dependencies and implements precise unsatisfiability caching based on hash data structures [10] in order to efficiently facilitate subset cache retrieval. Figure 6 shows the total number of processed non-deterministic alternatives for five configurations of caching precision and dependency handling for a selection of test cases solvable within one minute. Note that runtime is not a reasonable basis of comparison since all configuration variants of Figure 6 have 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

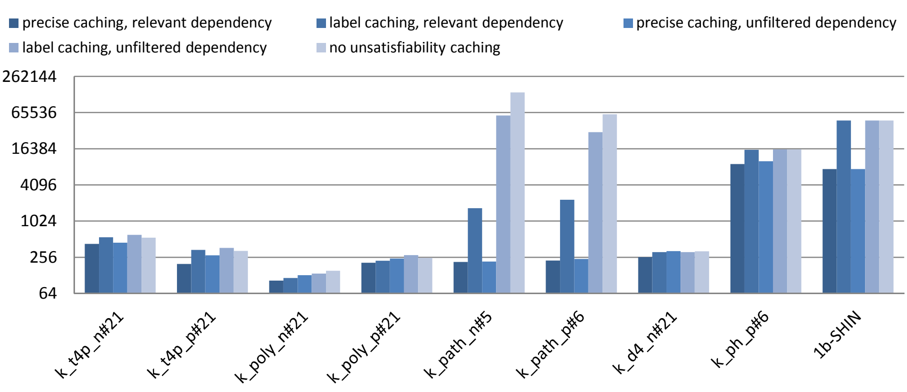

Fig. 6. Log scale comparison of processed alternatives for different caching methods

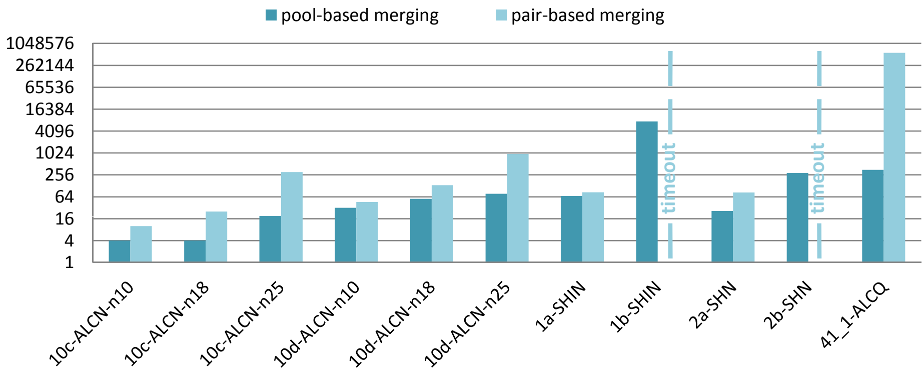

Fig. 7. Processed alternatives (on a logarithmic scale) for different merging methods

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 6 reveals that, amongst the tested configurations, precise caching provides the most effective pruning method. For some test cases it can reduce the number of non-deterministic alternatives by two orders of magnitude in comparison to label caching with unfiltered dependencies. Particularly the test cases k_path_n/p are practically solvable for Konclude only with precise caching at their largest available problem size (#21). The difference between relevant and unfiltered dependencies is less significant at least within our set of test cases.

Figure 7 compares pool-based with pair-based merging in terms of non-deterministic alternatives that have to be processed in order to solve selected test cases from [14]. In addition to the built-in pool-based merging we also added pair-based merging to our Konclude system. The test cases 10c and 10d are variants of the original test case 10a in terms of different problem sizes (10c) as well as more hidden contradictions nested within disjunctions (10d). The pool-based approach introduced in Sec. 4 clearly dominates the naive pair-based merging, especially when dealing with satisfiable problems (1b and 2b) and expressive DLs. Note that the test cases 1b and 2b are only solvable with pool-based merging within a one minute timeout. The required reasoning times strongly correlate to the number of processed alternatives for all test cases of Figure 7.

6 Conclusions

We have presented a range of optimisation techniques 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 number of tested non-deterministic branches by up to two orders of magnitude compared to standard caching techniques. Regarding cardinality constraints, the presented pool-based merging technique also achieves a significant improvement and a number of test cases can only be solved with this optimisation within an acceptable time limit. Both techniques 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, 2nd edn. Cambridge University Press (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 2007 (2007)
4. Donini, F.M., Massacci, F.: EXPTIME tableaux for $\mathcal{ALC}$. J. of Artificial Intelligence 124(1), 87–138 (2000)
5. Faddoul, J., Farsinia, N., Haarslev, V., Möller, R.: A hybrid tableau algorithm for $\mathcal{ALCQ}$. In: Proc 18th European Conf. on Artificial Intelligence (ECAI 2008), pp. 725–726 (2008)
6. Goré, R., Widmann, F.: Sound Global State Caching for *ALC* with Inverse Roles. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 205–219. Springer, Heidelberg (2009)
7. Haarslev, V., Möller, R.: Consistency Testing: The RACE Experience. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS (LNAI), vol. 1847, pp. 57–61. Springer, Heidelberg (2000)
8. 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 2001), pp. 161–168. Morgan Kaufmann (2001)
9. Haarslev, V., Sebastiani, R., Vescovi, M.: Automated Reasoning in $\mathcal{ALCQ}$ via SMT. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 283–298. Springer, Heidelberg (2011)
10. Hoffmann, J., Koehler, J.: A new method to index and query sets. In: Proc. 16th Int. Conf. on Artificial Intelligence (IJCAI 1999), pp. 462–467. Morgan Kaufmann (1999)
11. 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 2006), pp. 57–67. AAAI Press (2006)
12. Horrocks, I., Patel-Schneider, P.F.: DL systems comparison. In: Proc. 1998 Int. Workshop on Description Logics (DL 1998), vol. 11, pp. 55–57 (1998)
13. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. J. of Logic and Computation 9(3), 267–293 (1999)
14. Liebig, T.: Reasoning with OWL – system support and insights –. Tech. Rep. TR-2006-04, Ulm University, Ulm, Germany (September 2006)
15. 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 2010) (2010)
16. OWL Working Group, W.: OWL 2 Web Ontology Language: Document Overview. W3C Recommendation (October 27, 2009), `http://www.w3.org/TR/owl2-overview/`
17. 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)

18. 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), http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui/Ulmer_Informatik_Berichte/2012/UIB-2012-01.pdf
19. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
20. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. of Automated Reasoning 39, 277–316 (2007)