ELK Reasoner: Architecture and Evaluation

Yevgeny Kazakov[1], Markus Krötzsch[2], and František Simančík[2]

[1] Institute of Artificial Intelligence, Ulm University, Germany
[2] Department of Computer Science, University of Oxford, UK

Abstract. ELK is a specialized reasoner for the lightweight ontology language OWL EL. The practical utility of ELK is in its combination of high performance and comprehensive support for language features. At its core, ELK employs a consequence-based reasoning engine that can take advantage of multi-core and multi-processor systems. A modular architecture allows ELK to be used as a stand-alone application, Protégé plug-in, or programming library (either with or without the OWL API). This system description presents the current state of ELK and experimental results with some difficult OWL EL ontologies.

1 The System Overview

The logic-based ontology language OWL is becoming increasingly popular in application areas, such as Biology and Medicine, which require dealing with a large number of technical terms. For example, medical ontology SNOMED CT provides formal description of over 300,000 medical terms covering various topics such as diseases, anatomy, and clinical procedures. Terminological reasoning, such as automatic classification of terms according to subclass (a.k.a. 'is-a') relations, plays one of the central roles in applications of biomedical ontologies. To effectively deal with large ontologies, several profiles of the W3C standard OWL 2 have been defined [11]. Among them, the OWL EL profile aims to provide tractable terminological reasoning. Specialized OWL EL reasoners, such as CEL [1], Snorocket [9], and jCEL [10], can offer a significant performance improvement over general-purpose OWL reasoners.

This paper describes the ELK system.[3] ELK is developed to provide high performance reasoning support for OWL EL ontologies. The main focus of the system is (i) extensive coverage of the OWL EL features, (ii) high performance of reasoning, and (iii) easy extensibility and use. In these regards, ELK can already offer advantages over other OWL EL reasoning systems mentioned above. For example, as of today, ELK is the only system that can utilize multiple processors/cores to speed up the reasoning process, which makes it possible to classify SNOMED CT in less than 10 seconds on a commodity hardware [4]. This paper presents an overview of the implementation techniques used in ELK to achieve high performance of classification, and provides an experimental evaluation of classification using ELK and related reasoners on some of the largest available OWL EL ontologies.

[3] http://elk-reasoner.googlecode.com/

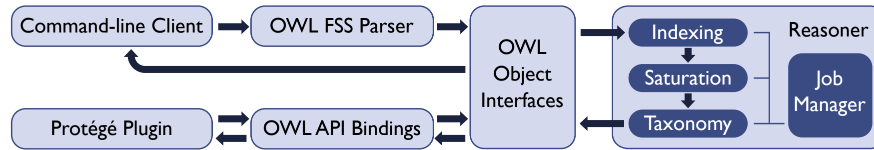


Fig. 1. Main software modules of ELK and information flow during classification

ELK is a flexible system that can be used in a variety of configurations. This is supported by a modular program structure that is organized using the Apache Maven build manager for Java. Maven can be used to automatically download, configure, and build ELK and its dependencies, but there are also pre-built packages for the most common configurations. The modular structure also separates the consequence-based reasoning engine from the remaining components, which facilitates extension of the system with new language features. The latest stable release ELK 0.2.0 supports conjunction (ObjectIntersectionOf), existential restriction (ObjectSomeValuesFrom), the top class (owl:Thing), complex role inclusions (property chains), and syntactic datatype matching. Support for disjointness axioms, ABox facts (assertions), and datatypes is under development.

The main software modules of ELK are shown in Fig. 1. The arrows illustrate the information flow during classification. The two independent entry points are the command-line client and the Protégé plug-in to the left. The former extracts OWL ontologies from files in OWL Functional-Style Syntax (FSS), while the latter uses ELK's bindings to the OWL API[4] to get this data from Protégé.[5] All further processing is based on ELK's own representation of OWL objects (axioms and expressions) that does not depend on the (more heavyweight) OWL API. The core of ELK is its reasoning module, which will be discussed in detail.

Useful combinations of ELK's modules are distributed in three pre-built packages, each of which includes the ELK reasoner. The *standalone client* includes the command-line client and the FSS parser for reading OWL ontologies. The *Protégé plugin* allows ELK to be used as a reasoner in Protégé and compatible tools such as *Snow Owl*.[6] The *OWL API bindings* package allows ELK to be used as a software library that is controlled via the OWL API interfaces.

2 The Reasoning Algorithm of ELK

The ELK reasoning component works by deriving consequences of ontological axioms under inference rules. The improvement and extension of these rules is an important part of the ongoing development of ELK [4, 7, 5]. To simplify the presentation, in this paper we focus on inference rules for a small yet non-trivial fragment of OWL EL, which is sufficient to illustrate the work of the main reasoning component of the ELK system.

[4] http://owlapi.sourceforge.net/
[5] http://protege.stanford.edu/
[6] http://www.b2international.com/portal/snow-owl

$$\mathbf{R_0}\ \frac{\mathsf{init}(C)}{\overline{C} \sqsubseteq C} \qquad \mathbf{R}^+_\top\ \frac{\mathsf{init}(C)}{\overline{C} \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O} \qquad \mathbf{R}_\sqsubseteq\ \frac{\overline{C} \sqsubseteq D}{\overline{C} \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O}$$

$$\mathbf{R}^-_\sqcap\ \frac{\overline{C} \sqsubseteq D_1 \sqcap D_2}{\overline{C} \sqsubseteq D_1 \quad \overline{C} \sqsubseteq D_2} \qquad \mathbf{R}^+_\sqcap\ \frac{\overline{C} \sqsubseteq D_1 \quad \overline{C} \sqsubseteq D_2}{\overline{C} \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occurs negatively in } \mathcal{O}$$

$$\mathbf{R}^-_\exists\ \frac{\overline{C} \sqsubseteq \exists R.D}{\mathsf{init}(D) \quad C \sqsubseteq \exists R.\overline{D}} \qquad \mathbf{R}^+_\exists\ \frac{D \sqsubseteq \exists R.\overline{C} \quad \overline{C} \sqsubseteq E}{\overline{D} \sqsubseteq \exists R.E} : \exists R.E \text{ occurs negatively in } \mathcal{O}$$

Fig. 2. Inference rules for reasoning in $\mathcal{EL}$

We use the more concise description logic (DL) syntax to represent OWL axioms (see, e.g., [2] for details on the relationship of OWL and DL, and [8] for DL syntax and semantics). The DL used here is $\mathcal{EL}$, which supports concept inclusion axioms (TBoxes) but no assertions (ABoxes). $\mathcal{EL}$ concepts are either atomic concepts or of the form $\top$ (top), $C \sqcap D$ (conjunction), and $\exists R.C$ (existential restriction), where C and D are concepts and R is a role. An $\mathcal{EL}$ ontology $\mathcal{O}$ is a set of axioms of the form $C \sqsubseteq D$ (subsumption) where C and D are $\mathcal{EL}$ concepts. We say that a concept C *occurs negatively* (resp. *positively*) in an ontology $\mathcal{O}$ if C is a syntactic subexpression of D (resp. E) for some axiom $D \sqsubseteq E \in \mathcal{O}$.

Inference rules for $\mathcal{EL}$ are shown in Fig. 2. They can be seen as a restriction of the rules for $\mathcal{ELH}$ [4]. The rules operate with expressions of the form $\mathsf{init}(C)$ and subsumptions of the form $\overline{C} \sqsubseteq D$ and $D \sqsubseteq \exists R.\overline{C}$. The bars in $\overline{C}$ and $\overline{D}$ have no effect on the logical meaning of the axioms; they are used to control the application of rules, which will be explained in detail in Section 4. The expression $\mathsf{init}(C)$ is used to initialize the derivation of superconcepts for C. The rules are sound, i.e., the conclusion subsumptions follow from the premise subsumptions and $\mathcal{O}$. The rules are complete for classification in the sense that, for each $\mathcal{EL}$ concept C and each atomic concept A occurring in $\mathcal{O}$, if $\mathcal{O}$ entails $C \sqsubseteq A$, then $\overline{C} \sqsubseteq A$ is derivable from $\mathsf{init}(C)$. Note that the axioms in $\mathcal{O}$ are never used as premises of the rules, but only as side-conditions of the rule $\mathbf{R}_\sqsubseteq$.

Example 1. Consider the ontology $\mathcal{O}$ consisting of the following axioms:

$$A \sqsubseteq \exists R.(B \sqcap C), \tag{1}$$

$$A \sqcap \exists R.B \sqsubseteq C. \tag{2}$$

To compute all atomic superconcepts of A, we start with the goal $\mathsf{init}(A)$ and compute all conclusions under the inference rules in Fig. 2.

$\mathsf{init}(A)$	initial goal	(3)
$\overline{A} \sqsubseteq A$	by $\mathbf{R_0}$ to (3)	(4)
$\overline{A} \sqsubseteq \exists R.(B \sqcap C)$	by $\mathbf{R}_\sqsubseteq$ to (4) using (1)	(5)
$\mathsf{init}(B \sqcap C)$	by $\mathbf{R}^-_\exists$ to (5)	(6)

$$
\begin{aligned}
A &\sqsubseteq \exists R.(\overline{B \sqcap C}) && \text{by } \mathsf{R}^-_\exists \text{ to (5)} && (7)\\
\overline{B \sqcap C} &\sqsubseteq B \sqcap C && \text{by } \mathsf{R_0} \text{ to (6)} && (8)\\
\overline{B \sqcap C} &\sqsubseteq B && \text{by } \mathsf{R}^-_\sqcap \text{ to (8)} && (9)\\
\overline{B \sqcap C} &\sqsubseteq C && \text{by } \mathsf{R}^-_\sqcap \text{ to (8)} && (10)\\
\overline{A} &\sqsubseteq \exists R.B && \text{by } \mathsf{R}^+_\exists \text{ to (7) and (9)} && (11)\\
\overline{A} &\sqsubseteq A \sqcap \exists R.B && \text{by } \mathsf{R}^+_\sqcap \text{ to (4) and (11)} && (12)\\
&\mathsf{init}(B) && \text{by } \mathsf{R}^-_\exists \text{ to (11)} && (13)\\
A &\sqsubseteq \exists R.\overline{B} && \text{by } \mathsf{R}^-_\exists \text{ to (11)} && (14)\\
\overline{A} &\sqsubseteq C && \text{by } \mathsf{R}_\sqsubseteq \text{ to (12) using (2)} && (15)\\
\overline{B} &\sqsubseteq B && \text{by } \mathsf{R_0} \text{ to (13)} && (16)
\end{aligned}
$$

Since $\overline{A} \sqsubseteq C$ has been derived but not, say, $\overline{A} \sqsubseteq B$, we conclude that C is a superconcept of A but B is not. The application of rules $\mathsf{R}^+_\exists$ and $\mathsf{R}^+_\sqcap$ in lines (11) and (12) uses the fact that the concepts $\exists R.B$ and $A \sqcap \exists R.B$ occur negatively in (2). Intuitively, these rules are used to "build up" the subsumption $\overline{A} \sqsubseteq A \sqcap \exists R.B$, so that rule $\mathsf{R}_\sqsubseteq$ with side condition (2) can be applied to derive $\overline{A} \sqsubseteq C$.

In order to classify an ontology $\mathcal{O}$, it is sufficient to compute the deductive closure of $\mathsf{init}(A)$ for every atomic concept A occurring in $\mathcal{O}$ using the rules in Fig. 2. Note that in this case the rules can derive only subsumptions of the form $\overline{C} \sqsubseteq D$ and $D \sqsubseteq \exists R.\overline{C}$ where C and D occur in $\mathcal{O}$. Therefore, the deductive closure can be computed in polynomial time.

In the following three sections we give details of the indexing, saturation, and taxonomy construction phases, which are the main components of the core reasoning algorithm implemented in ELK (see Fig. 1).

3 Indexing

The indexing phase is used to build datastructures that can be used to effectively check the side conditions of the rules in Fig. 2. Specifically, given an ontology $\mathcal{O}$, the index assigns to every (potentially complex) concept C and every role R occurring in $\mathcal{O}$ the following attributes.

$$
\begin{aligned}
C.\mathsf{toldSups} &= \{D \mid C \sqsubseteq D \in \mathcal{O}\}\\
C.\mathsf{negConj} &= \{\langle D, C \sqcap D\rangle \mid C \sqcap D \text{ occurs negatively in } \mathcal{O}\} \cup\\
&\quad\ \{\langle D, D \sqcap C\rangle \mid D \sqcap C \text{ occurs negatively in } \mathcal{O}\}\\
C.\mathsf{negExis} &= \{\langle R, \exists R.C\rangle \mid \exists R.C \text{ occurs negatively in } \mathcal{O}\}\\
R.\mathsf{negExis} &= \{\langle C, \exists R.C\rangle \mid \exists R.C \text{ occurs negatively in } \mathcal{O}\}
\end{aligned}
$$

The sets $C.\mathsf{negConj}$, $C.\mathsf{negExis}$, and $R.\mathsf{negExis}$ consisting of pairs of elements are represented as key-value (multi-) maps from the first element to the second.

Example 2. Consider the ontology $\mathcal{O}$ from Example 1. The following attributes in the index of $\mathcal{O}$ are nonempty.

$$A.\mathsf{toldSups} = \{\exists R.(B \sqcap C)\} \qquad (A \sqcap \exists R.B).\mathsf{toldSups} = \{C\}$$
$$A.\mathsf{negConj} = \{\langle \exists R.B, A \sqcap \exists R.B \rangle\} \qquad (\exists R.B).\mathsf{negConj} = \{\langle A, A \sqcap \exists R.B \rangle\}$$
$$B.\mathsf{negExis} = \{\langle R, \exists R.B \rangle\} \qquad R.\mathsf{negExis} = \{\langle B, \exists R.B \rangle\}$$

Indexing is a lightweight task that can be performed by a single recursive traversal through the structure of each axiom in the ontology. Since it can consider one axiom at a time, it can be started even before the whole ontology is known to the reasoner. In ELK, indexing is executed in a second thread in parallel to loading of axioms. In addition, ELK keeps track of the exact counts of negative and positive occurrences of concepts in order to enable fast incremental updates of the index structure without having to reload the whole ontology.

4 Saturation

The saturation phase computes the deductive closure of the input axioms under the inference rules in Fig. 2. This is where most time is spent in typical cases, and the optimization of this phase is key to overall efficiency.

The saturation algorithm is closely related to the "given clause" algorithm for saturation-based theorem proving and semi-naive (bottom-up) evaluation of logic programs. The algorithm maintains two collections of axioms: the set of *processed axioms* between which the rules have been already applied (initially empty) and the *to-do queue* of the remaining axioms (initially containing the input axioms). The algorithm repeatedly polls an axiom from the to-do queue; if the axiom is not yet in the processed set, it is moved there and the conclusions of all inferences involving this axiom and the processed axioms are added at the end of the to-do queue (regardless of whether they have been already derived).

Example 3. The derivation in Example 1 already presents the axioms in the order they are processed by the saturation algorithm. For example, after processing axiom (8), the processed set contains axioms (3)–(8), and the to-do queue contains axioms (9) and (10). The algorithm then polls axiom (9) from the queue, adds it to the processed set, and applies all inferences involving (9) and the previously processed axioms (3)–(8). In particular, to apply rule $\mathbf{R}^+_\exists$ with (9) as the second premise, the algorithm iterates over $B.\mathsf{negExis}$ to find possible ways of satisfying the side condition. Since $B.\mathsf{negExis}$ contains $\langle R, \exists R.B \rangle$, the algorithm looks for processed axioms of the form $D \sqsubseteq \exists R.(\overline{B \sqcap C})$ for some D, which can be used as the first premise of $\mathbf{R}^+_\exists$. Axiom (7) is of this form, so conclusion (11) is added to the to-do queue. Note that (a pointer to) the concept $\exists R.B$ used in the conclusion (11) can be taken directly from the pair $\langle R, \exists R.B \rangle$ in $B.\mathsf{negExis}$, so the concept does not have to be reconstructed (and reindexed) during the saturation phase. This illustrates that conclusions of the inference rules can be constructed by simply following the pointers in the index.

Algorithm 1: Processing of to-do axioms

```
process(D ⊑ ∃R.C̄):
if C.predecessors.add(⟨R, D⟩) then        // the axiom was not processed
    // the axiom can only be used as the first premise of R∃+
    for E ∈ (R.negExis.keySet() ∩ C.superConcepts) do
        F ← R.negExis.get(E);
        todo.add(D̄ ⊑ F);

process(C̄ ⊑ E):
if C.superConcepts.add(E) then            // the axiom was not processed
    // use the axiom as the second premise of R∃+
    for R ∈ (E.negExis.keySet() ∩ C.predecessors.keySet()) do
        F ← E.negExis.get(R);
        for D ∈ C.predecessors.get(R) do
            todo.add(D̄ ⊑ F);
    // use the axiom as premises of other rules
```

To speed up the search for matching premises of binary rules, there is not just one global set of processed axioms in ELK. Instead, axioms are assigned to different *contexts*, one context per each initialized concept C (one for which $\mathsf{init}(C)$ has been derived). The bar over C in the presentation of inference rules in Fig. 2 indicates that the axiom is assigned to the context of C. For example, $\overline{C} \sqsubseteq \exists R.D$ is assigned to the context of C and $C \sqsubseteq \exists R.\overline{D}$ is assigned to the context of D, even though the two axioms have the same logical meaning. Our assignment of contexts ensures that the two premises of each binary rule belong to the same context. Thus, when processing an axiom in some context, it is possible to restrict the search for relevant premises to this context.

In Example 3, the premises of the form $D \sqsubseteq \exists R.(\overline{B \sqcap C})$ can only occur in the context for $B \sqcap C$. Thus, one only needs to inspect axioms (7)–(10). Yet iterating over all processed axioms of a context may still be inefficient. To optimize the search even further, we save information about the (two types of) processed axioms within each context C in the following sets:

$$C.\mathsf{superConcepts} = \{D \mid \overline{C} \sqsubseteq D \text{ is processed}\},$$
$$C.\mathsf{predecessors} = \{\langle R, D\rangle \mid D \sqsubseteq \exists R.\overline{C} \text{ is processed}\}.$$

The latter set is implemented as a key-value multimap from R to D. Thus, to find all axioms of the form $D \sqsubseteq \exists R.(\overline{B \sqcap C})$ with the given R in the context $(B \sqcap C)$, it is sufficient to retrieve all values D for the key R in $(B \sqcap C).\mathsf{predecessors}$. Algorithm 1 demonstrates how these sets are used for processing of axioms.

The separation of axioms into contexts also helps in parallelizing the saturation phase because multiple workers can independently process axioms in different contexts at the same time [4]. To ensure that no two workers are concurrently processing axioms in the same context, the to-do queue in ELK is split into a two-level hierarchy of queues: each context of C maintains a local queue

C.todo of to-do axioms that are assigned to the context of C, and there is a global queue of *active contexts* whose to-do queues are nonempty. Using concurrency techniques, such as Boolean flags with atomic compare-and-set operations, the queue of active contexts is kept duplicate free. Each worker then repeatedly polls an active context C from the queue and processes all axioms in C.todo.

5 Taxonomy Construction

The saturation phase computes the full transitively closed subsumption relation. However, the expected output of classification is a *taxonomy* which only contains direct subsumptions between nodes representing equivalence classes of atomic concepts (if the taxonomy contains $A \sqsubseteq B$ and $B \sqsubseteq C$ then it should not contain $A \sqsubseteq C$, unless some of these concepts are equivalent). Therefore, the computed subsumptions between atomic concepts must be transitively reduced.

In the first step, we discard all subsumptions derived by the saturation algorithm that involve non-atomic concepts. Thus, in the remainder of this section, we can assume that all concepts are atomic.

A naive solution for computing the direct superconcepts of A is shown in Algorithm 2. The algorithm iterates over all superconcepts C of A, and for each of them checks if another superconcept B of A exists with $A \sqsubseteq B \sqsubseteq C$. If no such B exists, then C is a direct superconcept of A. This algorithm is inefficient because it performs two nested iterations over the superconcepts of A (it also does not work correctly in the presence of equivalent concepts). In realistic ontologies, the number of all superconcepts of A can be sizeable, while the number of direct superconcepts is usually much smaller, often just one. A more efficient algorithm would take advantage of this and perform the inner iteration only over the set of direct superconcepts of A that have been found so far, as shown in Algorithm 3. Given A, the algorithm computes two sets A.equivalentConcepts and A.directSuperConcepts. The first set contains all concepts that are equivalent to A, including A itself. The second set contains exactly one element from each equivalence class of direct superconcepts of A. Note that it is safe to execute Algorithm 3 in parallel for multiple concepts A.

Having computed A.equivalentConcepts and A.directSuperConcepts for each A, the construction of the taxonomy is straightforward. We introduce one tax-

Algorithm 2: Naive Transitive Reduction

```
for C ∈ A.superConcepts do
    isDirect ← true;
    for B ∈ A.superConcepts do
        if B ≠ A and B ≠ C and C ∈ B.superConcepts then
            isDirect ← false;
    if isDirect and C ≠ A then
        A.directSuperConcepts.add(C)
```

Algorithm 3: Better Transitive Reduction

```
for C ∈ A.superConcepts do
    if A ∈ C.superConcepts then
        A.equivalentConcepts.add(C);
    else
        isDirect ← true ;      // so far C is a direct superconcept of A
        for B ∈ A.directSuperConcepts do
            if C ∈ B.superConcepts then
                isDirect ← false ; // C is not a direct superconcept of A
                break;
            if B ∈ C.superConcepts then
                // B is not a direct superconcept of A
                A.directSuperConcepts.remove(B);
        if isDirect then
            A.directSuperConcepts.add(C);
```

onomy node for each distinct class of equivalent concepts, and connect the nodes according to the direct superconcepts relation. Finally, we put the top and the bottom node in the proper positions, even if $\top$ or $\bot$ do not occur in the ontology.

6 Evaluation

In this section we evaluate the performance of ELK for classification of large existing ontologies, and compare it to other commonly used DL reasoners.

Our test ontology suite contains the SNOMED CT ontology obtained from the official January 2012 international release by converting from the native syntax (RF2) to OWL functional syntax using the supplied converter. We also include the $\mathcal{EL}$ version of GALEN which is obtained from the version 7 of OpenGALEN[7] by removing all inverse role, functional role, and role chain axioms. Both ontologies have been used extensively in the past for evaluating $\mathcal{EL}$ reasoners. To obtain additional test data, we selected some of the largest ontologies listed at the OBO Foundry [14] and the Ontobee [16] websites that were in OWL EL but were not just plain taxonomies, i.e., included some non-atomic concepts. This gave us the Foundational Model of Anatomy (FMA), the e-Mouse Atlas Project (EMAP), Chemical Entities of Biological Interest (ChEBI), the Molecule Role ontology, and the Fly Anatomy. We also used two versions of the Gene Ontology which we call GO1 and GO2. The older GO1, published in 2006, has been used in many performance experiments. GO2 is the version of Mar 23 2012 and uses significantly more features than GO1, including negative occurrences of conjunctions and existential restrictions, and even disjointness axioms. Table 1

[7] http://www.opengalen.org/sources/sources.html

Table 1. Ontology metrics

Ontology	Atomic concepts	Atomic roles	Axioms
SNOMED CT	294,469	62	294,479
GALEN	23,136	950	36,489
GO1	20,465	1	28,896
GO2	36,215	7	139,485
FMA	80,469	15	126,547
ChEBI	31,470	9	68,149
EMAP	13,731	1	13,730
Molecule Role	9,217	4	9,627
Fly Anatomy	7,797	40	19,208

shows the number of concepts, roles, and axioms in each of these ontologies. Links to their sources can be found on the ELK website.[8] We plan to maintain and extend this list with further interesting $\mathcal{EL}$ ontologies in the future.

We compared the performance of the public development version of ELK (r577) to the specialized $\mathcal{EL}$ reasoners jcel 0.17.0 [10] and Snorocket 1.3.4.alpha4 [9], and to general OWL 2 reasoners FaCT++ 1.5.3 [15], HermiT 1.3.6 [12], and Pellet 2.3.0 [13]. We accessed all reasoners uniformly through the OWL API 3.2.4 [3] in their default settings. All experiments were executed on a laptop (Intel Core i7-2630QM 2GHz quad core CPU; 6GB RAM; Java 1.6; Microsoft Windows 7). On this architecture, ELK defaults to using 8 concurrent workers in the saturation phase; the other reasoners run in a single thread. We set time-out to 30 minutes and allowed Java to use 4GB of heap space. All figures reported here were obtained as the average over 5 runs of the respective experiments.

We loaded the ontologies using the OWL API and, in our first experiment, we measured the wall-clock time each reasoner spent executing the classification method `precomputeInferences(InferenceType.CLASS_HIERARCHY)`. The results are shown in Table 2. In all the 5 runs of the experiment, Pellet threw a `ConcurrentModificationException` on ChEBI. The measured classification times for Snorocket were 0 in all the test cases: the reasoner appears to trigger classification automatically after loading the ontology without waiting for the above method call. Therefore, for a more meaningful comparison of Snorocket with the remaining reasoners, in our second experiment we measured the overall time for loading and classification. These results are shown in Table 3.

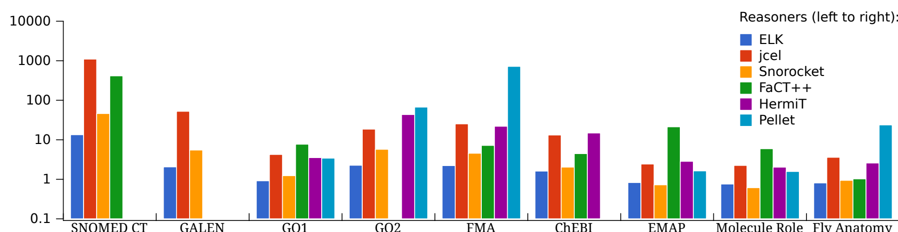
The results show that, on all tested ontologies, ELK and Snorocket far outperform all the remaining reasoners. On the smaller ontologies (GO1, ChEBI, EMAP, Molecule Role, and Fly Anatomy), ELK and Snorocket show similar performance, while on the larger ontologies (SNOMED CT, GALEN, GO2, and FMA) ELK is 2–3 times faster than Snorocket. In particular, ELK can load and classify SNOMED CT in under 15 seconds. Since ELK can update its index structure incrementally without having to reload the whole ontology, subsequent

[8] http://code.google.com/p/elk-reasoner/wiki/Test_Ontologies

Table 2. Classification times in seconds

	ELK	jcel	Snorocket	FaCT++	HermiT	Pellet
SNOMED CT	6.2	1041.6	0	408.9	time-out	mem-out
GALEN	1.3	48.2	0	time-out	mem-out	mem-out
GO1	0.4	2.6	0	7.2	2.5	2.3
GO2	1.0	12.8	0	time-out	41.0	63.5
FMA	0.9	19.4	0	5.8	19.6	714.9
ChEBI	0.9	8.5	0	3.8	13.5	exception
EMAP	0.4	1.1	0	20.9	1.9	0.9
Molecule Role	0.3	1.0	0	5.6	1.3	0.9
Fly Anatomy	0.4	2.34	0	0.7	1.8	22.9

Table 3. Loading + classification times in seconds



	ELK	jcel	Snorocket	FaCT++	HermiT	Pellet
SNOMED CT	13.4	1100.6	46.2	414.9	time-out	mem-out
GALEN	2.1	52.6	5.5	time-out	mem-out	mem-out
GO1	0.9	4.3	1.2	7.7	3.5	3.4
GO2	2.3	18.6	5.7	time-out	43.5	67.2
FMA	2.2	25.3	4.5	7.2	22.0	720.2
ChEBI	1.6	13.2	2.0	4.5	14.9	exception
EMAP	0.8	2.4	0.7	21.3	2.9	1.6
Molecule Role	0.8	2.2	0.6	5.9	2.0	1.6
Fly Anatomy	0.8	3.6	0.9	1.0	2.6	23.8

reclassification of SNOMED CT due to small changes in the ontology is likely to take only about 6 seconds as reported in Table 2.

To judge the correctness of reasoning, we compared the taxonomies computed by different reasoners. We found that, whenever a reasoner succeeded in computing a taxonomy at all, the result agreed with the results of all other successful reasoners. Although this does not exclude the possibility that all reasoners made the same errors, such a situation seems unlikely since each ontology was successfully classified by at least three (and often more) reasoners. Therefore, we conclude that in all test cases all reasoners computed the correct taxonomy.

Finally, we wanted to find out if the test ontologies entail any subsumptions that are not already "told", i.e., which do not follow by a simple transitive closure of the subsumptions explicitly present in the ontology. For this experiment we ran ELK disabling all inference rules except the initialization rules and rule $\mathbf{R}_{\sqsubseteq}$,

and we compared the taxonomies obtained in this way to the correct taxonomies. It turned out that the two taxonomies differed only for SNOMED CT, GALEN, and GO2. The remaining ontologies entail only told subsumptions.

Note that the fact that all entailed subsumptions are told does not immediately imply that the ontologies are trivial for classification because a reasoner still needs to prove that no other subsumptions hold, which usually requires full reasoning with all axioms in the ontology. This, however, is not the case here. A closer inspection revealed that, apart from SNOMED CT, GALEN, and GO2, all our test ontologies contain only axioms of the form $A \sqsubseteq B$ and $A \sqsubseteq \exists R.B$, where A and B are atomic concepts. In such case, the axioms of the form $A \sqsubseteq \exists R.B$ cannot possibly lead to new subsumptions between atomic concepts, and therefore can be discarded for classification. This can be shown easily, for example, using our calculus in Fig. 2: a positive occurrence of an existential restriction can lead to a new subsumption only through the interaction with a negative occurrence of some other existential restriction in rule $\mathsf{R}^{+}_{\exists}$; since there are no negative occurrences of existential restriction in these ontologies, axioms of the form $A \sqsubseteq \exists R.B$ cannot lead to new subsumptions.

Since our experiments suggest that ontologies without negative existentials are relatively common, it might be worthwhile to further optimize classification by disregarding positive existentials in such cases. Looking at the classification times, we believe that no reasoner currently takes advantage of this optimization. In its current implementation, ELK will also blindly apply rule $\mathsf{R}^{-}_{\exists}$ even when there are no negative existentials in the ontology.

7 Conclusions

This paper outlines some major implementation techniques that contribute to the overall efficiency of ELK, and evaluates the classification performance of several reasoners on large OWL EL ontologies. As can be seen from the evaluation results, despite their relatively large size, most ontologies were not difficult for ELK and can be classified in less than 1 second. Furthermore, we have observed that for many ontologies the classification problem is trivial due to their very limited use of the language features. It is worth noting, however, that although some axioms do not have any impact on classification, they can be used in some other reasoning tasks, such as finding subclasses of complex class expressions.

Since we had to present evaluation results, we were not able to discuss some further interesting optimization details in ELK, including, concurrent processing, efficient implementation of set intersections, such as those in Algorithm 1, and pruning of redundant inferences. These details can be found in the extended technical report [6]. Most of these methods are not specific to OWL EL, or even to description logics, and can thus benefit other (reasoning) tools that compute a deductive closure by exhaustive application of inference rules.

Acknowledgments This work was supported by the EU FP7 project SEALS and by the EPSRC projects ConDOR, ExODA and LogMap. The first author is supported by the German Research Council (DFG).

References

1. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—a polynomial-time reasoner for life science ontologies. In: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR'06). LNCS, vol. 4130, pp. 287–291. Springer (2006)
2. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
3. Horridge, M., Bechhofer, S.: The OWL API: A Java API for working with OWL 2 ontologies. In: Proc. OWLED 2009 Workshop on OWL: Experiences and Directions. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)
4. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of $\mathcal{EL}$ ontologies. In: Proc. 10th Int. Semantic Web Conf. (ISWC'11). LNCS, vol. 7032, pp. 305–320. Springer (2011)
5. Kazakov, Y., Krötzsch, M., Simančík, F.: Unchain my $\mathcal{EL}$ reasoner. In: Proc. 24th Int. Workshop on Description Logics (DL'11). CEUR Workshop Proceedings, vol. 745, pp. 202–212. CEUR-WS.org (2011)
6. Kazakov, Y., Krötzsch, M., Simančík, F.: ELK: a reasoner for OWL EL ontologies. Tech. rep. (2012), available from http://code.google.com/p/elk-reasoner/wiki/Publications
7. Kazakov, Y., Krötzsch, M., Simančík, F.: Practical reasoning with nominals in the $\mathcal{EL}$ family of description logics. In: Proc. 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'12) (2012), to appear, available from http://code.google.com/p/elk-reasoner/wiki/Publications
8. Krötzsch, M., Simančík, F., Horrocks, I.: A description logic primer. CoRR abs/1201.4089 (2012)
9. Lawley, M.J., Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In: Proc. 6th Australasian Ontology Workshop (IAOA'10). Conferences in Research and Practice in Information Technology, vol. 122, pp. 45–49. Australian Computer Society Inc. (2010)
10. Mendez, J., Ecke, A., Turhan, A.Y.: Implementing completion-based inferences for the $\mathcal{EL}$-family. In: Proc. 24th Int. Workshop on Description Logics (DL'11). CEUR Workshop Proceedings, vol. 745, pp. 334–344. CEUR-WS.org (2011)
11. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-profiles/
12. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. of Artificial Intelligence Research 36, 165–228 (2009)
13. 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)
14. Smith, B., Ashburner, M., Rosse, C., Bard, J., Bug, W., Ceusters, W., Goldberg, L.J., Eilbeck, K., Ireland, A., Mungall, C.J., Consortium, T.O., Leontis, N., Rocca-Serra, P., Ruttenberg, A., Sansone, S.A., Scheuermann, R.H., Shah, N., Whetzeland, P.L., Lewis, S.: The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. Nature Biotechnology 25, 1251–1255 (2007)
15. 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)
16. Xiang, Z., Mungall, C., Ruttenberg, A., He, Y.: Ontobee: A linked data server and browser for ontology terms. In: International Conference on Biomedical Ontologies (ICBO). pp. 279–281 (2011)