Proceedings of the OWL Reasoner Evaluation Workshop (ORE 2012)

Collocated with IJCAR 2012 Conference
July 1st, Manchester,UK

Volume 858 of CEUR-WS.org: http://ceur-ws.org/Vol-858/

Edited by: Ian Horrocks, Mikalai Yatskevich and Ernesto Jiménez-Ruiz

Preface

OWL is a logic-based ontology language standard designed to promote interoperability, particularly in the context of the (Semantic) Web. The standard has encouraged the development of numerous OWL reasoning systems, and such systems are already key components of many applications.

The goal of this workshop is to bring together both developers and users of reasoners for (subsets of) OWL, including systems focusing on both intensional (ontology) and extensional (data) query answering.

The workshop provided several datasets to evaluate the standard tasks of the reasoning systems. Furthermore, participants also had the possibility to submit their systems using the SEALS platform[1] in order to obtain standardised evaluations.

There were 16 papers submitted each of which was reviewed by at least three members of the program committee or additional reviewers. The program committee selected 14 papers for oral and poster presentation. Four reasoning systems were also submitted to the SEALS platform for evaluation: HermiT, FaCT++, jcel and WSReasoner. The evaluation results will be discussed with the authors during the workshop and published online shortly after it.[2]

Acknowledgements

We thank all members of the program committee, additional reviewers, authors and local organizers for their efforts. We would also like to acknowledge that the work of the workshop organisers was greatly simplified by using the EasyChair conference management system (www.easychair.org) and the CEUR Workshop Proceedings publication service (http://ceur-ws.org/). The organisers were partially supported by the the EU FP7 project SEALS and by the EPSRC projects ConDOR, ExODA and LogMap.

[1] http://www.seals-project.eu/

[2] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/

Program Committee

Franz Baader	TU Dresden
Jérôme Euzenat	INRIA & LIG
Jeff Heflin	Lehigh University
Ian Horrocks	University of Oxford
Ernesto Jiménez-Ruiz	University of Oxford
Pavel Klinov	University of Arizona, Clark and Parsia, LLC
Francisco Martín-Recuerda	Universidad Politecnica of Madrid
Bijan Parsia	University of Manchester
Stefan Schlobach	Vrije Universiteit Amsterdam
Kiril Simov	Bulgarian Academy of Sciences
Mikalai Yatskevich	University of Oxford

Additional Reviewers

Ana Armas	University of Oxford
Julian Mendez	TU Dresden
Peter Patel-Schneider	
Rafael Peñaloza	TU Dresden
Giorgio Stefanoni	University of Oxford

Keyword Index

Table of Contents

Solving the Projection Problem with OWL2 Reasoners: Experimental Study

Wael Yehia[1] and Mikhail Soutchanski[2]

[1] Department of Computer Science and Engineering York University, 4700 Keele Street, Toronto, ON, M3J 1P3, Canada
w2yehia@cse.yorku.ca

[2] Department of Computer Science, Ryerson University, 245 Church Street, ENG281, Toronto, ON, M5B 2K3, Canada mes@scs.ryerson.ca

Abstract. We evaluate HERMIT, an OWL2 reasoner, on a set of test cases that emerge from an unusual but very practical way of using Description Logics (DLs). In the field of reasoning about actions, the projection problem is an elemental problem that deals with answering whether a certain formula holds after doing a sequence of actions starting from some initial states represented using an incomplete theory. We consider a fragment of the situation calculus and Reiter's basic action theories (BAT) such that the projection problem can be reduced to the satisfiability problem in an expressive description logic $\mathcal{ALCO}^U$. We adapt an approach called regression where an input query is equivalently transformed until it can be directly checked against the initial theory supplemented with the unique name axioms (UNA) without any consideration to the rest of the BAT. To study regression in practice, we implemented it in C++, defined an XML SCHEMA to describe a BAT and queries, created 7 domains some of which are inspired from well-known planning competition domains, and invented generators that can create random but meaningful instances of the projection problem. The formula resulting from regressing a projection query, together with the initial theory and UNA, is fed to an OWL2 reasoner to answer whether the regressed query holds given the initial theory and UNA. We measure the input formula using a number of metrics, such as a number of $\mathcal{U}$-role occurrences and a number of individuals, and evaluate the performance of HERMIT on formulas along each dimension.

Keywords: random generator, Description Logics, projection problem, situation calculus, Action Theory

1 Introduction

We study a new class of formulas that arise from using Description Logics to solve one of the reasoning problems known as the projection problem (PP) which occurs naturally in the area of reasoning about actions. We consider a sufficiently broad sub-case of PP that can be formulated in a fragment of the situation calculus (SC) such that PP becomes decidable. We will explain later what the PP is, but what's important to note is that it is a prerequisite to few other important reasoning problems like planning, which makes it an interesting problem to tackle.

The reasoning about actions community hasn't made solutions to the problem very practical to use so far. There are no practical implementations that can solve the PP in a logical language that is not purely propositional, and when initial knowledge is incomplete, i.e., in realistic settings without the closed world assumption (CWA) and

without the domain closure assumption (DCA). Using our approach of transforming the PP into the satisfiability problem in DL is one way we can start the ball rolling (in the DL direction). Our goal is to provide automatically generated test cases for the DL community to work on and speed up reasoning time on. These test cases can be used for stress testing the DL reasoners. Indeed, our testing clearly shows that the majority of the time spent on solving one instance of the PP is in the DL reasoner.

Our test cases usually originate from a single common setting. There are two formulas: a premise formula (translated into a DL concept representing an initial theory) and a regressed query formula (translated into another DL concept). The goal is to check if the regressed query holds given the premise. The two formulas are translated into concepts in a DL called $\mathcal{ALCO}^U$ that includes nominals ($\mathcal{O}$), the universal role ($\mathcal{U}$), and constructs from the well-known logic $\mathcal{ALC}$. In addition, each test case may contain a set of individuals that belong to certain concepts, and might be related to each other using certain roles. We also make the Unique Name Assumption (UNA) for individuals by stating they are pairwise unequal.

Finally, the test cases are based on some practical domains such as Logistics and Assembly from the International Planning Competitions [6, 2], and on other domains invented by us to illustrate the expressivity of our language $\mathcal{L}$. We provide 7 domains (Logistics, Assembly, Airport, Turing Machine Addition, Turing Machine Successor, Scheduling World, and Hemophilia from biology) in which projection queries can be formulated, and provide query and action sequence generators for the first four domains. A thorough description of the domains can be found in [5].

We have developed the C++ program that transforms a PP instance into an instance of the $\mathcal{ALCO}^U$ satisfiability problem. Using all our tools, we are able to generate random PP instances, and subsequently transform them into $\mathcal{ALCO}^U$ concepts given as input to OWL2 reasoners.

There are at least two directions that can be taken from here in terms of testing. One is to generate test cases by varying the complexity of the projection problem such as initial state complexity/size, query size, and length of the action sequence. This will lead to investigation of the projection problem in terms of reasoning time. Second is to generate arbitrary but simple PP instances, and study the final transformed formulas based on their structure. That will provide new test suites for DL reasoners, and shed light on the areas where they need improvement. We take the second approach in this paper, because the first was studied in [1] where a comparison between two approaches to solving the projection problem is carried out. In this paper, we will measure the reasoning time for HERMIT[3], an OWL2 reasoner, on a randomly generated pool of test cases. We group the test cases based on (a) a number of individuals in the domain, for similar queries, and (b) a number of $\mathcal{U}$-role occurrences, then study the behavior of HERMIT on each group.

We will now explain the nature of the problem we are solving, and how it leads us to using DL reasoners.

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

2 Background

In the field of reasoning about actions, an important problem is answering whether a formula holds after executing a sequence of actions starting from some initial state. This problem is known as the projection problem, and there are many flavors of the problem that differ by the restrictions on the query, a set of initial states, language at hand, and other properties. In description logics (DLs) and earlier terminological systems, this problem was formulated using roles to represent transitions and concept expressions to represent states. This line of research as well as earlier applications of DLs to planning and plan recognition are discussed and reviewed in [3]. Reiter showed that the projection problem in the situation calculus can be solved using a method called *regression* [7]. The problem is undecidable in the general SC, but by limiting the expressivity to a fragment of SC one can get decidability. We propose a fragment P based on a language $\mathcal{L}$ in which we solve the projection problem using regression. Regression (explained below) is a method of transforming the projection problem into a satisfiability problem in some language. This language is critical, as the fact whether SAT in the language is decidable will lead to the decidability of the projection problem. Our test case generation and testing is based on P, and the language used to solve the satisfiability problem after regression can be mapped to the DL $\mathcal{ALCO}^U$. This is how we end up using the OWL reasoners because they can easily handle $\mathcal{ALCO}^U$ concepts. The executability problem is another common problem where an action's precondition axioms are checked for satisfiability in a given state. This problem is important as mechanisms like regression depend on the assumption that the action sequence in the query is executable starting from the initial situation.

In this paper, all constants start with upper case, and all variables with lower case letters. The free variables in formulas are assumed to be $\forall$-quantified at front. We use the standard first order logic (FOL) definitions of well-formed formulas and terms.

2.1 Basic Action Theories and Language $\mathcal{L}$

This approach is based on Basic Action Theories in SC. We will be brief in our description of BATs, but the interested readers can find formal definitions in [8]. In general, a BAT $\mathcal{D}$ consists of the pre-condition axioms (PAs) $\mathcal{D}_P$, the successor state axioms (SSAs) $\mathcal{D}_{SS}$, which characterize the effects and non-effects of actions; an incomplete initial theory $\mathcal{D}_{S_0}$ about the initial situation S_0; a set of domain independent foundational axioms; and axioms for the unique-name assumption (UNA). It is also possible to augment the BAT with a TBox [9, 4]. In SC, a *situation* is a sequence of actions. The truth values of some of the predicates can vary with respect to the state pointed out by the sequence of actions. The *fluents* are the predicates with a situation term as their last argument. You can think of them as "dynamic" predicates whose truth value can change based on the situation argument. Other predicates (with no situation argument) are called "static" predicates, because their truth value does not depend on situation. We say that a SC formula $\phi(s)$ is *uniform* in s, if s is the only situation term mentioned in $\phi(s)$, and $\phi(s)$ has no quantifiers over situation variables.

Subsequently, we consider only BATs with relational fluents, and no other function symbols except $do(a, s)$ and action functions are allowed. Terms of sort object can only be constants or variables. Action functions can have any number of object arguments.

The following is a short and informal definition of the language $\mathcal{L}$ that is used to construct formulas that are allowed in our restricted BATs. The precise definition is provided in [9]. $\mathcal{L}$ is a set of FOL formulas that is divided into two symmetric subsets $\mathcal{L}_x$ and $\mathcal{L}_y$. It uses a finite set of auxiliary variables usually denoted by z_1, z_2, and so on (whose sole purpose is to be replaced by constants during regression), as well as the x and y variables. The main restriction is that z-variables cannot be quantified over, hence they are always free in $\mathcal{L}$ formulas, while the x and y variables can be quantified, but only one of them can be free in a particular $\mathcal{L}$ formula. The $\mathcal{L}_x$ set includes all formulas with free x, and $\mathcal{L}_y$ includes all formulas with free y. The $\mathcal{L}$ formulas without occurrences of x, y, and the $\mathcal{L}$ formulas where all occurrences of x and y variables are bound belong to both $\mathcal{L}_x$ and $\mathcal{L}_y$. Informally, an $\mathcal{L}_x$ formula ($\mathcal{L}_y$ formulas are symmetric) has one of these syntactic forms:

1. atomic formulas such as $true$, $false$, $x = t_1$, $A(t_1)$, and $R(t_1, t_2)$,
   where A and R are unary and binary predicate symbols respectively, t_1 and t_2 can be either constants or z-variables, and t_1 can also be x.
2. non atomic formulas such as:
   (a) $A(x) \wedge B(x)$, $A(x) \vee B(x)$, and $\neg A(x)$, where $A(x)$ and $B(x)$ are $\mathcal{L}_x$ formulas.
   (b) $\forall t_1.D(t_1)$, $\exists t_1.D(t_1)$, $\forall y.R(t_2, y) \supset C(y)$, and $\exists y.R(t_2, y) \wedge C(y)$,
   where $D(t_1)$ is a $\mathcal{L}_x$ or $\mathcal{L}_y$ formula depending on whether t_1 is x or y, $C(y)$ is $\mathcal{L}_y$ formula, R is a binary predicate symbol, and t_2 can be either x, a constant, or a z-variable.

Note any z_i variable other than x and y has to be free in a formula from $\mathcal{L}$. The intuition behind the definition of $\mathcal{L}$ is that $\mathcal{L}$ formulas with z_i variables instantiated with constants should be $\mathcal{ALCO}^U$ representable.

Lemma 1. *For any formula $\phi \in \mathcal{L}$ with all z-variables instantiated with constants, there exist a translation to an $\mathcal{ALCO}^U$ concept with no more than a linear increase in the size of ϕ. The inverse also holds, i.e. any $\mathcal{ALCO}^U$ concept can be translated into an $\mathcal{L}$ formula without z-variables.*

The variables z_i are important for our purposes because they serve as arguments of actions in axioms. Thanks to them, we can consider BATs where actions may have any number of arguments, thereby increasing expressivity of BATs that can be formulated. This becomes important when benchmark planning domains (considered as FOL specifications) have to be represented as our BATs. Notice that $\mathcal{L}$ formulas do not always have equivalent $\mathcal{ALCO}^U$ concepts; only $\mathcal{L}$ formulas **without z-variables** do have. But this is exactly what is required for our purposes of regression. The final regressed formula is guaranteed to be **z-variable free**, since in the projection formula uniform in S all arguments of actions mentioned in the situation term S must be instantiated with constants. As a consequence, all z variables become instantiated as well. The Lemma 1 is the reason why we can use the DL $\mathcal{ALCO}^U$ to solve any PP instance in our class P of BATs. Lemma 1 is proved using the standard translation between DLs and First Order Logic (FOL); the proof is similar to the proof of Lemma 1 in [4]. Notice that using the standard translation between FOL and a DL, the formula $\forall y.R(t_2, y) \supset C(y)$ translates as $\forall R.C$, the formula $\exists y.R(t_2, y) \wedge C(y)$ becomes $\exists R.C$, and the formulas $\forall t_1.D(t_1)$,

$\exists t_1.D(t_1)$ can be translated as $\forall U.D$ and $\exists U.D$, respectively. The latter formulas are handy for our purposes because in our axioms we need unguarded quantifiers. As another example, a z-free $\mathcal{L}$ formula $\exists x.(Box(x) \wedge x \neq B_1 \wedge ready(x))$, some box distinct from B_1 is ready, can be translated to $\mathcal{ALCO}^U$ as $\exists U.(Box \sqcap \neg\{B_1\} \sqcap ready)$, where B_1 is a nominal. Notice that the nominals are handy to translate exceptions.

We proceed to describing how $\mathcal{L}$ is used in the BAT, and how it expands the expressivity of the BAT to beyond DLs, while maintaining complexity of deciding the PP in the range acceptable from a DL perspective. To facilitate understanding of each part of the BAT, we will use examples from one of our domains, the Logistics domain.

The domain describes objects (boxes or luggage) that can be loaded and unloaded from vehicles (trucks or airplanes) and transported from one location to another. Trucks can be driven from one location to another in the same city, while airplanes can fly between airports (which are locations) in different cities. Let the logical language include:

1. Four **action functions**: $load(obj, vehicle, location)$, $unload(obj, vehicle, location)$, $drive(truck, locFrom, locTo, city)$, $fly(airplane, locFrom, locTo)$.
2. Three **relational fluents**: $loaded(obj, vehicle, s)$, $at(x, loc, s)$, and $ready(obj, s)$ (where x can be an object or a vehicle).
3. **Static unary predicates** (i.e. predicates with no situation term) to describe each type of entity, and one static binary predicate $in\_city(loc, city)$.

For brevity, let a vector $\boldsymbol{x}$ of object variables denote either x, y, or $\langle x, y\rangle$, and let $\boldsymbol{z}$ denote a vector of place holder variables $\langle z_1, z_2, ...\rangle$.
Action precondition axioms (PAs) $\mathcal{D}_{AP}$: the preconditions that have to hold before an action can be executed. There is one axiom per action $Act(\boldsymbol{z})$ of the following form:

$$(\forall \boldsymbol{z}, s).\ Poss(A(\boldsymbol{z}), s) \equiv \Pi_A(\boldsymbol{z}, s),$$

where $Poss$ (derived from 'possible') is a special binary predicate that occurs on the left hand side of PAs only, and Π_A is an $\mathcal{L}$ formula whose only free variables are the place holder z_i variables. The formula $\Pi_A(\boldsymbol{z}, s)$ is uniform in s. For example, the PA for action $load(Object, Vehicle, City)$ is the following:

$$Poss(load(z_1, z_2, z_1), s) =$$
$$(obj(z_1) \wedge veh(z_2) \wedge loc(z_3) \wedge ready(z_1, s) \wedge at(z_1, z_3, s) \wedge at(z_2, z_3, s))$$

Definition 1. *Let ϕ_x and ϕ_y be $\mathcal{L}_x$ and $\mathcal{L}_y$ formulas, respectively, such that they are uniform in s (ϕ_x, ϕ_y are called context conditions). Let $\boldsymbol{u}$ be a vector of variables at most containing $\boldsymbol{z}$ and an optional x variable, $\boldsymbol{v}$ be a vector of variables at most containing $\boldsymbol{z}$, and optionally x or y variables. Let $Act(\boldsymbol{u})$ and $Act(\boldsymbol{v})$ be action terms, and a be an action variable. A CC formula has one of the following two forms:*

$$\exists \boldsymbol{z}.a = Act(\boldsymbol{u}) \wedge \phi_x(x, \boldsymbol{z}, s) \qquad \textit{SSA for an unary fluent}$$
$$\exists \boldsymbol{z}.a = Act(\boldsymbol{v}) \wedge \phi_x(x, \boldsymbol{z}, s) \wedge \phi_y(y, \boldsymbol{z}, s) \qquad \textit{SSA for a binary fluent}$$

Successor state axioms (SSAs) $\mathcal{D}_{SS}$: Define the direct effects and non-effects of actions. There is one SSA for each fluent $F(\boldsymbol{x}, s)$ of the following syntactic form:

$$F(\boldsymbol{x}, do(a, s)) \equiv \gamma^+_F(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma^-_F(\boldsymbol{x}, a, s), \tag{1}$$

where each of the γ_F's are disjunctions of CC formulas. For example, the SSA for fluent *loaded* is as follows:

$$loaded(x, y, do(a, s)) =$$
$$[\exists z_1.a = load(x, y, z_1) \wedge obj(x) \wedge veh(y) \wedge loc(z_1) \wedge ready(x, S) \wedge at(x, z_1, s)]$$
$$\vee [loaded(x, y, s) \wedge \neg[\exists z_1.a = unload(x, y, z_1)]]$$

TBox axioms $\mathcal{D}_T$:
These are TBox axioms for unary predicates, where the right hand side is an $\mathcal{L}$ formula without z variables. For example:

$$veh(x) \equiv truck(x) \vee airplane(x)$$
$$loc(x) \equiv street(x) \vee airport(x)$$

Initial Theory $\mathcal{D}_{S_0}$: The $\mathcal{D}_{S_0}$ is an $\mathcal{L}$ sentence without z variables, i.e. it can be transformed into an $\mathcal{ALCO}^U$ concept. For example:

$$city(Toronto) \wedge airport(YYZ) \wedge in\_city(YYZ, Toronto) \wedge street(Yonge) \wedge$$
$$in\_city(Yonge, Toronto) \wedge box(B1) \wedge at(B1, Yonge, S0) \wedge$$
$$mail\_truck(T1) \wedge at(T1, Yonge, S0) \wedge \forall x(obj(x) \supset ready(x, S0)) \wedge$$
$$box(B2) \wedge (loaded(B2, T1, S0) \vee \neg\exists x(loaded(B2, x, S0) \wedge vehicle(x)))$$

Finally, a projection query is an $\mathcal{L}$ sentence, without z variables, and there is a ground situation term S representing the action sequence after which the formula should hold or not. For example, given the above initial theory, the action sequence represented by situation $do(drive(T1, Yonge, YYZ, Toronto), S_0)$ is executable and the following query answers true:

$$at(T1, YYZ, do(drive(T1, Yonge, YYZ, Toronto), S_0))$$

2.2 Regression for Solving the Projection Problem

In the context of BATs and SC, *regression* is a recursive transformation converting a formula uniform in situation $do(a, s)$ into a logically equivalent formula uniform in s (that is one action shorter down the situation term) by making use of the SSAs. A modified regression operator $\mathcal{R}$ is defined to guide the regression process in our class P of BATs, and it is defined recursively on formulas of the underlying language at hand, $\mathcal{L}$ in our case. We do not define the modified operator here due to space limitations, but interested readers can see [4] for more details about regression in a language that is similar to $\mathcal{L}$ (but that is a proper subset of $\mathcal{L}$). The idea is that all static predicates are not affected by regression, and hence remain the same after the regression operator is applied. Fluents ("dynamic" predicates) on the other hand are transformed by $\mathcal{R}$. On each step, the regression operator $\mathcal{R}$ replaces each fluent formula uniform in situation $do(a, s)$ by the right hand side (RHS) of the SSA for the fluent (recall that the CC formulas on the RHS are uniform in s). Subsequently, regression continues until all fluents have S_0 as the only situation term. Consider the following example query:

$$loaded(B1, T1, do(load(B1, T1, Yonge), S_0))$$

Also, let the above $\mathcal{D}_{S_0}$ be the initial theory against which this query is checked. First, replace the fluent (with its constant arguments and situation term) by the right hand side of the SSA, to get:

$$\big(\exists z_1.load(B1,T1,Yonge) = load(B1,T1,z_1) \wedge obj(B1) \wedge veh(T1)$$
$$\wedge loc(z_1) \wedge ready(B1,S_0) \wedge at(B1,z_1,S_0)\big) \vee$$
$$\big(loaded(B1,T1,S_0) \wedge \neg\exists z_1.load(B1,T1,Yonge) = unload(B1,T1,z_1)\big)$$

By applying UNA – similar action names denote the same action and similar constant names denote the same object in the world – we get a shorter FOL formula:

$$\big(\exists z_1.B1{=}B1 \wedge T1{=}T1 \wedge Yonge{=}z_1 \wedge obj(B1) \wedge veh(T1) \wedge loc(z_1) \wedge$$
$$ready(B1,S_0) \wedge at(B1,z_1,S_0)\big) \vee \big(loaded(B1,T1,S_0) \wedge \neg\exists z_1.false\big)$$

Further simplifications yield the formula, which is the result of one step of regression:

$$\big(obj(B1) \wedge veh(T1) \wedge loc(Yonge) \wedge ready(B1,S_0) \wedge at(B1,Yonge,S_0)\big) \vee$$
$$loaded(B1,T1,S_0)$$

It is clear that the first disjunct holds given the above $\mathcal{D}_{S_0}$. Hence, the answer to this projection problem is true.

Note that the resulting formula is uniform in S_0. In general, a query whose situation term mentions n ground actions, requires n consecutive regression steps to bring it down to situation S_0. The benefit of regression is that the final regressed formula is logically equivalent to the original query, but now we do not need to consider the whole BAT to answer the query, just $\mathcal{D}_{S_0}$ and UNA. Thereby, the projection problem is transformed from solving whether $BAT \models Query$ to solving whether $UNA \cup \mathcal{D}_{S_0} \models \mathcal{R}[Query]$, where $\mathcal{R}[Query]$ is the formula resulting from regression of the query. Since $\mathcal{D}_{S_0}$, UNA and $\mathcal{R}[Query]$ are z-free $\mathcal{L}$ formulas, they can be converted into $\mathcal{ALCO}^U$ concepts, and the above entailment problem can be transformed into the satisfiability problem of the $\mathcal{ALCO}^U$ concept (abusing notation) $\mathcal{D}_{S_0} \sqcap UNA \sqcap \neg\mathcal{R}[Query]$.

3 Test Case Generation

As a means of representing a PP instance, we used XML to represent each part of the BAT and designed an XML SCHEMA to characterize the representation. After performing regression on the query of the PP instance, the input to the reasoner was represented in OWL Manchester syntax. A PP test case contains a fixed part consisting of the SSAs, PAs, and TBox axioms for a particular domain, and a varying part consisting of (1) the initial theory, (2) the query and (3) the action sequence. We obviously need to generate the variable part. Due to the non-trivial expressivity of the language at hand, it is hard to generate useful test cases. We looked in the literature and found no precedence for such an attempt, i.e. generating random projection problem test cases. Planning domains [2] had some test case generation involved, but still on purely propositional level, so that inconsistencies in input data can be easily avoided. In contrast, generating random $\mathcal{ALCO}^U$ formulas usually yields meaningless queries or initial theories that are

inconsistent. Building formulas from patterns is one step forward towards generating good test cases, but it suffers from the fact that generated formulas might be similar and consequently this approach does not provide the extensive coverage that random formula generation does. We tried mixing patterns with a bit of randomness.

We have 7 domains in our disposal, and we created query generators and action sequence generators for 4 of them (lack of time is the only obstacle for the other 3). For every domain, we created a few patterns to guide in creating the query formula (6-10 per domain). Some of these patterns can take a human input, or can generate their random input if none given. A pattern for the Logistics domain might for example describe the question of whether there are any boxes on a truck X in some city Y, where X and Y is the input to the pattern. The objective is to have as versatile patterns as possible but keeping them simple because we make use of them in the next step. Next, we generate a random propositional DNF formula made up of, say n, unique literals. Then, use the patterns with random input (or guided input if used with action sequence generators) to generate n atomic queries, and replace every literal in the DNF formula with one of those queries (we map each of the n literals to a specific atomic query to avoid propositional inconsistency). The motivation for this approach is that having randomness at the propositional level is good, but not at the FOL level.

To generate random but executable action sequences, we used patterns again. But now a pattern is a generic description of a sequence of actions necessary to satisfy a goal. The pattern is represented by an algorithm that computes an executable sequence of actions based on the provided pattern. For instance, one action sequence in the Logistics domain describes the process of gathering all known boxes in a particular city and transporting them to another city. The choice of the cities is random, and picking the transportation vehicle is random as well. Basically, the generator extract all the information it can from the given initial theory, and picks its random input from the gathered information. Of course, we could have tried picking random actions with random ground arguments and check the executability of these actions, but most likely they would fail to be executable. In fact, picking an executable ground action is related to conjunctive query answering which is a totally different and nontrivial problem. Besides that, solving the executability problem will incur an expensive running time for test case generation.

It is important to note that the query and action sequence generators use the initial theory as input, so that they can generate meaningful queries and executable actions. One limitation is that only literals are used from the initial theory, assuming the initial theory is a conjunction containing some literals. This assumption simplifies significantly solving the executability problem, because the preconditions of an action can be easily verified using these literals (again, assuming the preconditions are simple enough to be verified with the information from the literals, which is true in our case).

Finally, the initial theory (IT) contains both static and dynamic incomplete knowledge. Due to lack of time and the nature of some of our metrics, we decided to manually create ITs, and did that for only one domain: Logistics. Hence, we did not make use of the generators for the other 3 domains. We created 55 unique ITs, with number of individuals per IT varying between 5 and 60 individuals. We built them starting from a small IT of size 5, and incrementally added an individual to the IT to create the next

bigger IT. This way we have a monotonically ascending order of IT size. This is important because the projection queries generated from a smaller initial theory can then be run against a bigger initial theory simply because the preconditions of an action are satisfied in the bigger IT if they are satisfied in the smaller IT. This way we can better measure the effect of varying the number of individuals on the same query.

Using the ITs, and the two generators, we were able to generate at least 10 unique combinations of query + action sequence (QAS) for each IT. And by reusing QASs from smaller initial theories, we generated a pool of around 12000 test cases.

The next step is to classify the pool of test cases based on several metrics.

3.1 Classification

We use 2 metrics to classify our generated formulas, in an effort to show how the variation of values in each metric affects the reasoning time in HERMIT. The metrics are: (a) number of constants in the initial theory, and (b) number of $\mathcal{U}$-role occurrences. Metric (b) is measured using the initial theory and the regressed query formula combined, while metric (a) can be measured using the initial theory $\mathcal{D}_{S_0}$ only because $\mathcal{D}_{S_0}$ (together with UNA) defines all the individuals allowed in a PP instance. Neither the query nor regression can add new individuals to the domain, and the query and action sequence would use individuals mentioned in the initial theory.

The number of individuals in the domain is an interesting factor because in practice it would be useful to know the effect of adding more individuals on the run time of answering a projection query. Note that the regression of a QAS is the same regardless of the IT, but the ITs are increasing in size in each test case, which enables us to see the direct affect of having more individuals in a domain. For this metric, we create groups of test cases, such that each group has a single query common to all test cases, and the number of individuals in the initial theory of each test case increases monotonically.

We picked the second metric, number of $\mathcal{U}$-role occurrences, because we noticed that even test cases that have few occurrences of $\mathcal{U}$-roles slow down solving SAT for a concept. Out of curiosity, we also tried to replace all occurrences of $\mathcal{U}$-roles with some ordinary role R (we know semantics drastically change), and the reasoning time dropped by a factor of 10 or more.

There could be other possible metrics, such as the number of disjunctions, or the depth of propositional connectives, or the depth of quantifiers in the formula, but we didn't use them due to lack of time.

One last observation we made, is that the initial theory in $\mathcal{L}$ usually contains a lot of assertive formulas, i.e. of the form $A(c)$ and $R(c, b)$ for some unary predicate A, binary predicate R, and constants/individuals c and b. For instance, in the Logistics example above, we have formulas such as $city(Toronto)$ and $at(Mt1, Main, S0)$, and they appear as conjuncts in the initial theory $\mathcal{D}_{S_0}$. Note that these assertive formulas can be fluents, not just static predicates. Instead of representing these formulas from the initial theory as an $\mathcal{ALCO}^U$ concept, we may represent them as concept and role assertions in the test case. While doing this, we may expect speed-up in reasoning time as this is the more natural way of representing this sort of information in OWL. For this reason, we created yet another set of test cases which are ABox'ed versions of the original set of test cases. We call a test case ABoxed if it represents its initial theory as

OWL assertions. Note that this representation does not leave the initial theory empty, but only shortens it by removing those assertive formulas from it, and keeping the other formulas untouched. In the next section, we deal with these two sets of test cases, where SAT test cases are the regular test cases where the initial theory is represented as a complex $\mathcal{ALCO}^U$ concept with all assertive formulas included. Finally, to get UNA in OWL2 we declare all individuals to be pairwise different (using the OWL2 construct `differentIndividuals`).

4 Results

For all testing we used a machine with the following specs: Intel® Core™ 2 Duo E8400 CPU with a clock frequency of 3.00 GHz, and 4 GB of RAM. We used JVM version 1.7.0 and HermiT 1.3.6. We used a cutoff time of 30 sec. All results and test-cases can be found at: `http://www.cse.yorku.ca/~w2yehia/ORE_results.html`

4.1 Number of Constants

We already explained how we measure this metric, and how our choice and construction of ITs is suitable for the purpose of testing this metric. In the two graphs below, we show the reasoning time taken by HERMIT as a function of the number of individuals in the initial theory.

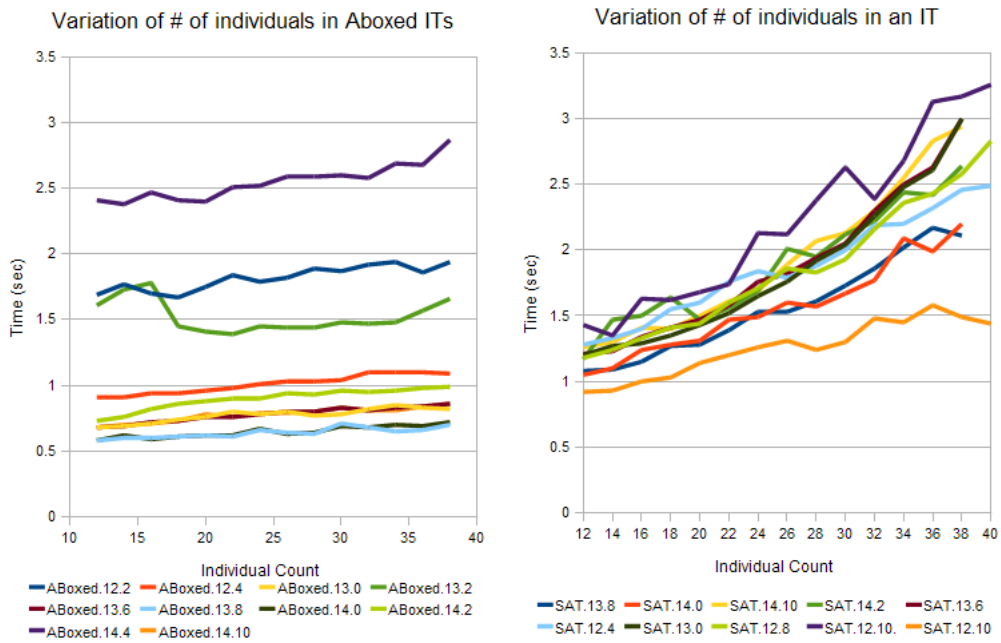


Fig. 1.

The left graph shows the running time of ABoxed test cases, and the right graph shows the regular non-ABoxed test cases. We chose to graph the part of our testing results where all test cases in a group finished in under the 30 seconds cutoff time. The groups of test cases shown below are labeled with the prefix 'ABoxed' and 'SAT', followed by the individual count in the initial theory that spawned the QAS for that particular group, and the last number is just an index of the QAS for the particular individual count.

4.2 Number of $\mathcal{U}$-role Occurrences

For this metric, we counted the number of $\mathcal{U}$-role occurrences in a test case (tc). Then, we grouped the test cases into sets, with a range of allowed number of $\mathcal{U}$-role occurrences per set. For better granularity, the ranges are narrow for small number of $\mathcal{U}$-roles, and widen as the number grows. Table 1 shows the ranges that we used in the first column, the number of test cases tested for that range (some ranges had > 1000 test cases, so we picked the first 100 for each range) in the second column, and the number of test cases that was answered in under 30 sec in the third. The fourth column simply shows the success rate for that range. The next three columns show the same as the previous three but for ABoxed test cases.

Table 1. Testing Results based on $\mathcal{U}$-role number of occurrences

# of U-role occur.	# of SAT tcs tested	# of successful SAT tcs	success rate (%)	# of ABoxed tcs tested	# of successful ABoxed tcs	success rate (%)
0-5	0	0	-	100	98	98
6-10	0	0	-	100	89	89
11-15	10	10	100	100	60	60
16-20	25	25	100	100	67	67
21-25	57	57	100	100	13	13
26-30	100	100	100	100	33	33
31-35	100	100	100	80	31	39
36-40	100	100	100	100	82	82
41-50	100	99	99	100	44	44

It is important to note that for all test cases, doing regression alone takes in most cases 1 second or less, and at most 2 seconds. Thus, most of the time spent on solving a PP is in HERMIT.

5 Discussion and Future Work

For the number of constants in the initial theory, it is clear that ABoxed test cases are less affected by the increase in individual count as compared to the non-ABoxed test cases. This shows that ABox-ing an initial theory is the more efficient way of representation when it comes to HERMIT.

For the number of $\mathcal{U}$-role occurrences, we believe that the reason why ABoxed test-cases seem to run much slower on average (lower success rate), is because of the way we translate an assertion into a concept - we use $\mathcal{U}$-roles. So, the ABoxed version will contain less $\mathcal{U}$-roles coming from the initial theory, than the regular version, simply because the assertions turned into concepts using $\mathcal{U}$-roles in the regular version, become OWL assertions in the ABoxed version. As a consequence, the remaining $\mathcal{U}$-roles come from the regressed formula so that the total number of $\mathcal{U}$-roles is same (recall the number of $\mathcal{U}$-role occurrences is counted both in the initial theory and in the regressed formula). For example, if a test-case had 10 $\mathcal{U}$-roles coming from the initial theory and 5 $\mathcal{U}$-roles from the regressed formula, then the ABoxed version will contain 5 $\mathcal{U}$-roles only, because the initial theory $\mathcal{U}$-roles disappear as a result of ABoxing, but SAT (non-ABoxed) version will contain 15 (5+10) $\mathcal{U}$-roles. This means that you cannot compare directly a SAT test-case with n occurrences of $\mathcal{U}$-roles to an ABoxed test-case with n occurrences of $\mathcal{U}$-roles because they represent different test-cases. You should expect that the ABoxed one will run slower because it has a bigger regressed formula to handle (this is where the extra $\mathcal{U}$-roles come from).

Finally, the long reasoning time in HERMIT can be attributed to either (a) inefficient representation of the regression formula when outputted by our regression program, or (b) slow performance of HERMIT when it comes to dealing with $\mathcal{U}$-roles.

Future work: We started with 7 domains, and wrote generators for 4 of them, and manually created initial theories for one domain. So there is some work left to be done, at least to make use of the generators for the 3 domains. We only managed to measure the performance of HERMIT, but the generated test cases are in Manchester syntax, and any reasoner using OWL API can run those test cases. We mentioned already that more metrics can be measured, and the effects of ABox'ing can be studied in more details.

References

1. Baader, F., Lippmann, M., Liu, H., Soutchanski, M., Yehia, W.: Experimental Results on Solving the Projection Problem in Action Formalisms Based on Description Logics. In: Description Logics W/sh DL-2012 (Accepted). (2012)
2. Bacchus, F.: The AIPS '00 planning competition. AI Magazine **22**(3) (2001) 47–56
3. Devanbu, P.T., Litman, D.J.: Taxonomic plan reasoning. Artif. Intell. **84**(1-2) (1996) 1–35
4. Gu, Y., Soutchanski, M.: A Description Logic Based Situation Calculus. Ann. Math. Artif. Intell. **58**(1-2) (2010) 3–83
5. Kudashkina, E.: An Empirical Evaluation of the Practical Logical Action Theory (Undergraduate Thesis CPS40A/B, Fall 2010 - Winter 2011). Department of Computer Science, Ryerson University, Toronto, Ontario, Canada (2011)
6. McDermott, D.V.: The 1998 AI Planning Systems Competition. AI Magazine **21**(2) (2000) 35–55
7. Reiter, R.: The projection problem in the situation calculus: A soundness and completeness result, with an application to database updates. In: In Proceedings First International Conference on AI Planning Systems, Morgan Kaufmann (1992) 198–203
8. Reiter, R.: Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. The MIT Press (2001)
9. Yehia, W., Soutchanski, M.: Towards an Expressive Logical Action Theory. In: Proc. of the 25th Intern. Workshop on Description Logics (DL-2012), Rome, Italy (2012) to appear

Chainsaw: A Metareasoner for Large Ontologies

Dmitry Tsarkov and Ignazio Palmisano

University of Manchester,
School of Computer Science,
Manchester, UK
{tsarkov, palmisai}@cs.man.ac.uk

Abstract. In this paper we present the results of running the ORE test suite and its reasoning tasks with three reasoners: Chainsaw, JFact and FaCT++. We describe the newest of these reasoners, Chainsaw, in detail, and we compare and contrast its advantages and disadvantages with the other two reasoners. We show the results obtained and how they compare to each other, and we list the cases in which the reasoners fail to complete a task in the assigned time.

1 Introduction

As it is well known to ontology engineers, ontologies become harder to manage, in terms of understanding them and reasoning with them, in a way which is, unfortunately, not proportional to the increase of their size. It is often the case that a small number of changes makes an ontology much harder for a reasoner to process. However, keeping ontologies small and simple is not always possible, because they might fail to satisfy the application requirements. On the other hand, it is possible for a small ontology to be hard to reason about.

How to best divide an ontology into smaller portions according to the needs of the task at hand is still an open problem; the rule of thumb that many approaches follow is that, when in doubt, one should try to keep the ontology as small as possible.

This is the main intuitive reason for modularisation, i.e., a set of techniques for splitting ontologies into fragments without loss of entailments for a given set of terms. To date, however, not many tools provide either user support for these techniques, nor leverage them for reasoning tasks.

Chainsaw [10], our metareasoner implementation prototype, exploits the idea in the following way. For every inference query it creates a module of the ontology that is then used to answer the query. The module is created in a way that guarantees the result of the query should be the same as for the whole ontology, i.e., there is no loss of entailments.

Chainsaw is designed as a wrapper for other reasoners. It can use reasoner factories to build reasoners on modules of its root ontology, and will integrate the ability to choose the reasoner best suited to each reasoning task on the basis of the module characteristics, such as size and/or expressivity. The most obvious characteristic is an OWL 2 profile of the module. E.g., if the profile is OWL 2

EL then some efficient EL reasoner, like ELK [4], could be used to achieve the best performance.

The advantages of using modules instead of whole ontologies inside a reasoner reside in the simplification of reasoning. The complexity of reasoning in OWL 2 DL is N2EXPTIME-hard on the size of the input, therefore being able to divide the ontology is likely to produce performance improvements orders of magnitude larger than the relative reduction in size.

Moreover, using modules inside the reasoner can help the knowledge engineer to concentrate on modelling the knowledge instead of worrying about the language pitfalls, since the extra complexity can be tackled in a transparent way. This, however, does not mean that complexity is no longer an issue. Modularisation is not a silver bullet for reasoning, as in some cases the module needed to answer a query might still include most of the ontology.

Another advantage of Chainsaw architecture is that it is able to use different OWL reasoners for each query and module; this allows for choosing the reasoner best suited for the OWL 2 profile of a specific module. In those cases where it is not clear which reasoner is best, it is possible to start more than one reasoner and simply wait for the first one to finish the task - this might require more resources, but statistical records can be kept to improve future choices. This functionality, however, is not yet implemented.

The reverse of the medal is that, for simple ontologies, Chainsaw is likely to be slower than most of the reasoners it uses; this derives from the overhead needed to manage multiple reasoners and modularisation of the ontology itself. Our objective, in this paper, is to illustrate an approach that can squeeze some answers out of ontologies that are too troublesome for a single traditional reasoner to handle.

In Section 2, we describe briefly the theory behind Atomic Decomposition and Modularisation, which are the building blocks of this approach; in Section 3 we describe the implementation details, tradeoffs and strategies adopted, and in Section 5 we present the results on the effectiveness of Chainsaw comparing to JFact and FaCT++ on the ORE test suite.

2 Atomic Decomposition and Modularisation

We assume that the reader is familiar with the notion of OWL 2 axiom, ontology and entailments. An *entity* is a named element of the signature of an ontology. For an axiom α we denote $\widetilde{\alpha}$ the signature of that axiom, i.e. the set of all entities in α. The same notation is also used for the set of axioms.

Definition 1 (Query). *Let O be an ontology. An axiom α is called an* entailment *in O if $O \models \alpha$. A check whether an axiom α is an entailment in O is an* entailment query. *A* subclass (superclass, sameclass) query *for a class C in an ontology O is to determine the set of classes $D \in \widetilde{O}$ such that $O \models C \sqsubseteq D$ (resp. $O \models D \sqsubseteq C, O \models C \equiv D$). A* hierarchical query *is either subclass, superclass, or sameclass query.*

Definition 2 (Module). *Let O be an ontology and Σ be a signature. A subset M of the ontology is called* module *of O w.r.t. Σ if for every axiom α such that $\widetilde{\alpha} \subseteq \Sigma$, $M \models \alpha \iff O \models \alpha$.*

One way to build modules is through the use of axiom *locality*. An axiom is called ($\bot$-) local w.r.t a signature Σ if replacing all entities not in Σ with $\bot$ makes the axiom a tautology. This syntactic approximation of locality was proposed in [1] and provides a basis for most of the modern modularity algorithms [3].

This modularisation algorithm is used to create an atomic decomposition of an ontology, which can then be viewed as a compact representation of all the modules in it [12].

Definition 3 (Atomic Decomposition). *A set of axioms A is an* atom *of the ontology O, if for every module M of O, either $A \subseteq M$ or $A \cap M = \emptyset$. An atom A is* dependent *on B (written $B \preccurlyeq A$) if for every module M if $A \subseteq M$ then $B \subseteq M$. An* Atomic Decomposition *of an ontology O is a graph $G = \langle S, \preccurlyeq \rangle$, where S is the set of all atoms of O.*

The dependency closure of an atom, computed by following its dependencies, constitutes a module; this module can then be used to answer queries about the terms contained in this closure.

However, for a hierarchical query the signature would contain a single entity, but the answer set would contain entities that might not be in the module built for that signature.

In order to address this problem, we use Labelled Atomic Decomposition (LAD), as described in [11].

Definition 4 (Labelled Atomic Decomposition). *A* Labelled Atomic Decomposition *is a tuple $LAD = \langle S, \preccurlyeq, L \rangle$, where $G = \langle S, \preccurlyeq \rangle$ is an atomic decomposition and L is a labelling function that maps S into a set of labels. A* top-level labelling *maps an atom A to a (possibly empty) subset of its signature $L(A) = \widetilde{A} \setminus (\bigcup_{B \preccurlyeq A} L(B))$.*

Proposition 1. *Let $LAD = \langle S, \preccurlyeq, L \rangle$ be a top-level labelled atomic decomposition of an ontology O. Then for all named classes x, y from $\widetilde{O}$ the following holds:*

1. *If $O \models x \sqsubseteq y$, then $\exists A, B \in S : x \in L(A), y \in L(B)$ and $B \preccurlyeq A$;*
2. *If $O \models x \equiv y$, then $\exists A \in S : x \in L(A)$ and $y \in L(A)$.*

Proof. 1) From [3], Proposition 11, $O \models x \sqsubseteq y$ iff for the $\bot$-locality-based module M of O w.r.t. signature $\{x\}$ holds $M \models x \sqsubseteq y$. Assume $O \models x \sqsubseteq y$. Then M is non-empty and $x \in \widetilde{M}$. Thus there is an atom $A \in S$ such that $x \in L(A)$. Due to the atomic decomposition properties, the union of an atom together with all the dependent atoms forms a module. So let $M_A = \bigcup_{B \preccurlyeq A} B$ be such a module. This module also has x in its signature, so $M \subseteq M_A$. But by the definition of the top-level labelling M_A is the smallest module that contains x in the signature;

so $M = M_A$. This also means that there is only one atom which label contains x. Now, using the results from [3], we can conclude that $y \in \widetilde{M_A}$; that means, that one of the atoms $B \in M_A$ is labelled with y. But all such atoms are dependencies of A, i.e. $B \preccurlyeq A$.

2) Assume $O \models x \equiv y$, which is equivalent to $O \models x \sqsubseteq y$ and $O \models x \sqsubseteq y$. From Case 1) this means that there are atoms A, A', B, B' such that $x \in L(A), x \in L(A'), y \in L(B), y \in L(B')$ and $B' \preccurlyeq A, A' \preccurlyeq B$. As shown in Case 1), there is only one atom that contains x (resp. y) in its label, so $A = A', B = B'$ and $B \preccurlyeq A, A \preccurlyeq B$. The latter is possible only in case $A = B$. □

This proposition provides a way to separate parts of the ontologies necessary to answer hierarchical queries about named classes. Indeed, it is enough to label the atomic decomposition with a top-level labelling and the modules for finding a subsumption relation can easily be found. This approach is orthogonal to a modularity-based one: while the latter deals easily with the entailment-like queries, the former provides a way to describe an ontology subset suitable to answer hierarchical queries.

This also explains the choice of $\bot$-locality in our approach. It is possible to define $\top$-locality in a similar manner (replace all entities not in signature with $\top$), and use it for the module extraction. Module extraction could be also done in a more complex manner, called $STAR$, interleaving $\top$ and $\bot$ extraction passes until a fixpoint is reached. However, $\top$-local modules are usually larger than $\bot$-local ones, so there is no reason to use them. The $STAR$ modularisation, although provides slightly smaller modules, does not ensure properties from Proposition 1, that are necessary for our approach.

3 Implementation of Chainsaw

The essence of Chainsaw is mirrored in the paper's title. Unlike other reasoners, which usually do the classification of the ontology before any query is asked, Chainsaw deals with requests in a lazy way, leaving classification to the delegate reasoners, which are usually at work on a small subset of the ontology.

For each query received, Chainsaw tries to keep the subset of the ontology needed to answer as small as possible without sacrificing completeness. This is achieved using different strategies according to the query; i.e., it is not possible to reduce the size of the ontology when checking for its consistency; however, other queries, as detailed in Section 2, can be answered by using modules built via LAD or locality based modules. More in detail, querying about superclasses of a term will only need the dependency closures of the top-level atoms for that term for the answers to be computed; the opposite is true for subclass requests.

During preprocessing of the ontology, a LAD of that ontology is built, using the Atomic Decomposition algorithm available in FaCT++ [5], and both dependency closure and its reverse are cached for every class name. For every query the module is constructed: via modularisation algorithm for entailment queries and via LAD for hierarchical queries. Then a suitable reasoner is created

for that module, and the query is delegated to it. The answer then is returned to a user.

A naive strategy for answering any query would consist of:

- Build a module M for the query
- Start a new reasoner R on M
- Answer the original query using R

However, it is easy to find possible optimisations to this strategy.

First, this approach creates a new reasoner for each query; if two queries with the same signature are asked, two (identical) modules would be built and two reasoners would be initialized, while just keeping the same reasoner would be enough.

Moreover, while the number of possible signatures for a query is exponential in the size of the ontology signature (not counting possible fresh entities used in the query), the number of distinct modules that can be computed against a given ontology with these signatures is much smaller [2]. This means that, given a module, there is a good chance that it can be reused for answering queries with a slightly different signature; therefore, the same reasoner can be used to answer more than one specific query.

Therefore, a tradeoff exists between reducing the size of the module to be reasoned upon, the complexity of determining such a module and the cost of starting a new reasoner for each query; to this, one must add the memory requirements of keeping a large module and reasoner cache.

Our approach in Chainsaw is to use a cache for modules and a cache for reasoners, both limited in the number of cached elements, and ordered as least recently used (LRU) caches; this has shown to perform rather well in some of our tests, where around one hundred thousand entailment checks against a large ontology have been satisfied using approximately 100 simultaneous reasoners, some of which were reused up to 20 times before being discarded. Similar results have been obtained when caching the modules to avoid rebuilding the same module for the same signature.

3.1 Future Improvements

There is one more optimisation that was not implemented: if a module is included in another module, the larger module can be used in place of the smaller one. However, this presents a slippery slope problem: at what level do we stop using the next containing module, since we do not have an easy way to predict where this series of modules will become really hard to reason with?

Determining the containment is also an expensive operation; for simple modules, this operation might cost more than the actual reasoning required. The sweet spot for this optimisation is a situation in which many fairly complex modules share a large number of axioms and are used often, and their union does not produce a module which pushes the reasoner's envelope. Using the union would provide for a good boost in performance and save memory as well, but at the time of writing we do not have an effective way of finding such spots.

It seems that atomic decomposition could provide relevant information for this task; an educated guess would be that such sweet spots reside near the parts of the dependency graph where a number of edges converge, but, to the best of our knowledge, there is no strong evidence in favor of this correlation. Future work might well explore this area.

Another improvement is to add a strategy to choose the best suited reasoner for a given module; such a strategy would have to take into account the known weak spots and strong points of each reasoner, as well as the characteristics of the module and of the query, such as size and OWL profile, or whether the query requires classification of the module or not. Where this is not sufficient, statistical records could be kept in order to create evidence based preferences and improve the strategy over time.

4 Characteristics of the Reasoners

In this paper we present the comparison between three reasoners that have something in common.

Chainsaw is an OWL 2 DL reasoner that uses modularity to improve query answering for large and complex ontologies. FaCT++ [5] is a tableaux highly optimised OWL 2 DL reasoner implemented in C++. JFact[1] is a Java implementation of FaCT++, that has extended datatypes support. The description of the reasoners' features can be found in Table 1.

Characteristic	Chainsaw	JFact	FaCT++
OWL 2 profile supported	EL, RL, QL, DL		
Interfaces	OWLAPI[2]	OWLAPI	OWLAPI, LISP
Algorithms	AD/sub-reasoner	tableaux	tableaux
Optimisations	meta-reasoning sub-reasoner	same as FaCT++	N/A
Advantages	scalability; good modularity approximation	pure Java; extended DT	good general performance
Disadvantages	AD overhead	work in progress	OWLAPI interface is complicated
Application focus	large ontologies	general purposes	

Table 1. Characteristics of compared reasoners.

Some notes about the table. The algorithm used in Chainsaw for answering hierarchical queries is based on Labelled Atomic Decomposition. For entailment queries an algorithm used in the chosen sub-reasoner is applied. The architecture

[1] http://jfact.sourceforge.net
[2] http://owlapi.sourceforge.net

provides the possibility of using different kinds of reasoners for different queries, so that more efficient reasoners can be used when possible; however, our current implementation does not provide this functionality yet.

We are not going to say much about FaCT++ and JFact here; the optimisations used in FaCT++ are described in a number of publications [6–9], and JFact is a port of FaCT++, so it uses the same techniques and contains the same optimisations. The advantage of Chainsaw is the focus: for every query it uses only the necessary part of the ontology. This allows it to work in situations where the ontology is too large and complex to be dealt with by any other reasoner. So we see the main application area of Chainsaw as large and complex ontologies. Both FaCT++ and JFact are general purpose reasoners, which can be easily integrated in any application that uses `OWLReasoner` interface. In addition to that, FaCT++ could be used directly from C and C++ programs.

Summarising the paper results, the main disadvantage of Chainsaw is the overhead that is brought in by the need to build a LAD and maintain a number of sub-reasoners that answers particular entailment queries. JFact is still a work in progress, so there is a high variance in performance depending on the task. Also, it has a small user base, by which we mean that there are probably bugs which have not been found yet. FaCT++ main problem, when used from Java applications, is its JNI interface, which makes it cumbersome to set up in some environments; for example, in web applications, native libraries are not straightforward to load and work with, and an error might cause issues to the whole application server. The datatype support is not very refined at the time of writing as well.

5 Empirical Evaluation

To check the performance of Chainsaw we ran several tests with it. For the tests we used a MacBook Pro laptop with 2.66 GHz i7 processor and 8Gb of memory.

We use Chainsaw with FaCT++ v 1.5.3 as a delegate reasoner, and compare the results with the same version of FaCT++ and JFact v 0.9.

In Table 2 we present the results coming from the ORE test suite, specifically the number of tests that our reasoners failed; these failures are either timeout failures or problems with some unsupported feature of the input ontologies.

As reported in the Total row, Chainsaw has the least amount of failures, while JFact has the highest; this can be explained in terms of time needed for the tasks: we know already that JFact performances are worse than FaCT++, and Chainsaw does not need to perform all tasks on the whole ontology. This gives it an edge over both JFact and FaCT++ in the case of large/complex ontologies; although FaCT++ is much faster on smaller ontologies, larger ontologies push it over our five minutes timeout, thus causing a failure.

We do not report detailed timings for every task in the test suite. Instead, for every task we select the minimal time for successfully completing the task

Task	CHAINSAW	JFact	FACT++
Classification	10	9	21
Class Sat	0	14	3
Ontology Sat	0	11	0
Pos. Entailment	2	3	2
Neg. Entailment	0	0	0
Retrieval	0	1	2
Total	12	38	28

Table 2. Number of failed tests in the ORE test suite.

and use it to calculate a normalised performance index for each reasoner. This provides a relative measure on how good a reasoner is compared to the others.

The index is calculated for the initialisation of a reasoner and for performing the task itself. We then average this index for every reasoner over all the ontologies in every test suite task, and the resulting values are in Table 3.

Task	Init			Main Task		
	CHAINSAW	JFact	FACT++	CHAINSAW	JFact	FACT++
Classification	1.01	3.85	1.75	46.6	43.59	1.32
Class Sat	1.04	2.86	1.74	14.15	126.9	3.9
Ontology Sat	1.05	2.77	1.85	19.17	56.6	1
Pos. Entailment	1	4.18	2.2	51.27	76.2	1
Neg. Entailment	1	4.13	2.06	58.36	119	1
Retrieval	1.04	3.35	1.88	4.39	44.37	25.57

Table 3. Average performance indexes for all reasoners over the ontologies included in each reasoning task.

As can be seen from the data in Table 3, initialisation is a simple process for CHAINSAW; in fact, it does very little work at this stage, while both JFACT and FACT++ do a substantial amount of loading and preprocessing; CHAINSAW is instead delegating this work until it becomes necessary, i.e., at query answering time.

At query time, FACT++ turns out to be the fastest reasoner by a large margin; CHAINSAW comes second and JFACT last. CHAINSAW is faster in the retrieval task, but its results were checked manually and they are incorrect; the speed is therefore probably due to a bug in the modularisation code that is used for building the module to be used, and its good performance should not be taken into account.

It is worth mentioning that CHAINSAW average includes the ontologies on which the other reasoners failed; therefore, we can conclude that, while slower than FACT++, it can provide answers in cases in which the other reasoners would not be able to answer in a timely manner at all.

6 Conclusions

In this paper, we presented the relative performances of the three reasoners, and listed their advantages and disadvantages. From these results, we deduce that CHAINSAW has the potential of providing acceptable performances where other reasoners fail; it also emerges that its current performances in other cases are not on a par with more mature reasoners such as FACT++. We consider this an encouraging result, and we presented a few improvements that are already under development.

References

1. Cuenca Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. JAIR 31, 273–318 (2008)
2. Del Vescovo, C., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: Atomic decomposition. In: Proc. of IJCAI-11. pp. 2232–2237 (2011)
3. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Extracting modules from ontologies: A logic-based approach. In: Stuckenschmidt, H., Parent, C., Spaccapietra, S. (eds.) Modular Ontologies, Lecture Notes in Computer Science, vol. 5445, pp. 159–186. Springer (2009)
4. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent classification of el ontologies. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) International Semantic Web Conference (1). Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
5. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Automated Reasoning pp. 292–297 (2006)
6. Tsarkov, D., Horrocks, I.: Efficient reasoning with range and domain constraints. In: Haarslev, V., Möller, R. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 104. CEUR-WS.org (2004)
7. Tsarkov, D., Horrocks, I.: Optimised classification for taxonomic knowledge bases. In: Horrocks, I., Sattler, U., Wolter, F. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 147. CEUR-WS.org (2005)
8. Tsarkov, D., Horrocks, I.: Ordering heuristics for description logic reasoning. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI. pp. 609–614. Professional Book Center (2005)
9. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. Autom. Reasoning 39(3), 277–316 (2007)
10. Tsarkov, D., Palmisano, I.: Divide et impera: Metareasoning for large ontologies. In: Proc. of 9th International Workshop OWL: Experiences and Directions (OWLED 2012). To Appear (2012)
11. Vescovo, C.D.: The modular structure of an ontology: Atomic decomposition towards applications. In: Rosati, R., Rudolph, S., Zakharyaschev, M. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)
12. Vescovo, C.D., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: Atomic decomposition and module count. In: Kutz, O., Schneider, T. (eds.) WoMO. Frontiers in Artificial Intelligence and Applications, Frontiers in Artificial Intelligence and Applications, vol. 230, pp. 25–39. IOS Press (2011)

Evaluating DBOWL: A Non-materializing OWL Reasoner based on Relational Database Technology

Maria del Mar Roldan-Garcia, Jose F. Aldana-Montes

University of Malaga, Departamento de Lenguajes y Ciencias de la Computacion
Malaga 29071, Spain,
(mmar,jfam)@lcc.uma.es,
WWW home page: http://khaos.uma.es

Abstract. DBOWL is a scalable reasoner for OWL ontologies with very large Aboxes (billions of instances). DBOWL supports most of the fragment of OWL covering OWL-DL. DBOWL stores ontologies and classifies instances using relational database technology and combines relational algebra expressions and fixed-point iterations for computing the closure of the ontology, called knowledge base creation. In this paper we describe and evaluate DBOWL. For the evaluation both the standard datasets provided in the context of the ORE 2012 workshop and the UOBM (University Ontology Benchmark) are used. A demo of DBOWL is available at http://khaos.uma.es/dbowl.

1 Introduction

With the explosion of Linked Data[1], some communities are making an effort to develop formal ontologies for annotating their databases and are publishing these databases as RDF triples. Examples of this are biopax[2] in the field of Life Science and LinkedGeoData[3] in the field of Geographic Information Systems. This means that formal ontologies with a large number (billions of) instances are now available. In order to manage these ontologies, current platforms need a scalable, high-performance repository offering both light and heavy-weight reasoning capabilities. The majority of current ontologies are expressed in the well-known Web Ontology Language (OWL) that is based on a family of logical formalisms called Description Logic (DL). Managing large amounts of OWL data, including query answering and reasoning, is a challenging technical prospect, but one which is increasingly needed in numerous real-world application domains from Health Care and Life Sciences to Finance and Government.

In order to solve these problems, we have developed DBOWL, a scalable reasoner for very large OWL ontologies. DBOWL supports most of the fragment of OWL covering OWL 1 DL. DBOWL stores ontologies and classifies instances

[1] http://linkeddata.org/
[2] http://www.biopax.org/
[3] http://linkedgeodata.org/About

using relational database technology. The state-of-the-art algorithm for achieving soundness and completeness in reasoning with expressive DL ontologies is the so-called Tableau procedure. Current Tableau-based implementations such as Pellet, Racer and HermiT show very good behavior in practice, but are completely memory-based and thus cannot cope with ontologies that have a large ABox. Several alternative approaches using disk-oriented implementations have been presented. These proposals can be classified into three categories. (1) Those which combine a DL main-memory based reasoner with a database, (2) Those which translate the ontology to Datalog and use a deductive database to evaluate it, and (3) Those which extend the database with reasoning capabilities. Our proposal follows a different approach: The state-of-the-art OWL reasoner Pellet[4] is currently used to classify the ontology Tbox. Information returned by Pellet is stored in a relational database. Class and property instances are also stored in the relational database as relation tuples. An algorithm which combines relational algebra expressions with fixed-point iterations is used to compute the closure of the of the ontology, called *knowledge base creation*. The use of an OWL reasoner, like Pellet, to classify the Tbox is crucial in our approach. This allows the capture of some Tbox inferences that cannot be obtained by other similar proposals such as those based on disjunctive datalog [1].

This paper presents a description and an evaluation of DBOWL. In order to evaluate DBOWL we use the standard datasets provided in the context of the ORE 2012 workshop[5] and the UOBM (University Ontology Benchmark) [2], the extremely well known benchmark for comparing ontology repositories in the Semantic Web. The rest of the paper is organized as follows. Section 2 introduces the theoretical concepts on which DBOWL is based. Section 3 describes the theoretical foundation of DBOWL, presenting the process for computing the ontology closure. Section 4 discusses the advantages and limitations of DBOWL. The evaluation of DBOWL is presented in Section 5. Finally Section 6 concludes the paper.

2 Preliminaries

2.1 The Relational Model

The relational model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper [3]. The relational model is characterized by its simplicity and mathematical foundation. The relational model represents the database as a collection of *relations*.

A **domain** D is a set of atomic values. Atomic means that each value in the domain is indivisible as far as the relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. A **relation schema** R, denoted

[4] http://clarkparsia.com/pellet
[5] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/

by $R(A_1, A_2, \ldots, A_n)$ is made up of a relation name R and a list of attributes $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role payed by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by $dom(A_i)$. The **degree** (or arity) of a relation is the number of attributes n of its relation schema. A **relation** (or **relation state**) r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by $r(R)$, is a **mathematical relation** of degree n on the domains $A_1, A_2, \ldots, A_n)$, which is a **subset** of the **cartesian product** of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n))$$

$r(R)$ is defined more informally as a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each n-tuple t is an ordered list of n values $t =< v_1, v_2, \ldots, v_n >$, where each value v_i, $1 \leq i \leq n$, is an element of $dom(A_i)$, or is a special NULL value. NULL is used to represent the values of attributes that may be unknown or may not apply to a tuple. This notation is used in the rest of the paper.

The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used. A relation is defined as a set of tuples. Mathematically elements of a set have *no order* among them.

2.2 The Relational Algebra

The basic set of operations for the relational model has an algebraic topology, and is known as the **Relational Algebra**. Operands in the Relational Algebra are Relations. Relational Algebra is closed with respect to the relational model: Each operation takes one or more relations and returns a relation. Given closure property, operations can be composed.

Relational Algebra operations enable a user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. A sequence of relational algebra operations forms a **relational algebra expression**, the result of which will also be a relation that represents the result of a database query (or retrieval request). Therefore, it is possible to assign a new relation name to a relational algebra expression, in order to simplify its use by other relational algebra expressions. Such relations are called *idb* (Intensional) relations, unlike the relations in **R**, which are called *edb* (Extensional) relations.

Operations in relational algebra can be divided into two groups: Set operations from mathematical set theory (UNION ($\cup$), INTERSECTION ($\cap$), SET DIFFERENCE ($\setminus$) and CARTESIAN PRODUCT ($\times$)), and operations developed specifically for relational databases (SELECT (σ), which selects a *subset* of the tuples from a relation that satisfied a selection condition, PROJECT (π), which selects certain attributes from the relation and discard the other attributes, JOIN ($\bowtie$), which combines related tuples from two relations into single tuples) among others.

2.3 DBOWL Ontologies

In order to simplify the implementation of the reasoner, some restrictions are imposed on the OWL ontologies supported by DBOWL. Even so, the ontologies supported by DBOWL are expressive enough for real application in the Semantic Web. DBOWL covers all of OWL 1 DL including inverse, transitive and symmetric properties, cardinality restrictions, simple XML schema defined datatypes and instance assertions. Enumerate classes (a.k.a, nominals) are only partially supported.

Let P and Q be properties, x be an individual and n be a positive number, class descriptions in DBOWL ontologies are formed according to the following syntax rule:

$$C, D \rightarrow A\ (NamedClass) \mid \neg A\ (complementOf\ NamedClass) \mid$$
$$C \sqcap D\ (intersectionOf\ ClassDescriptions) \mid$$
$$C \sqcup D\ (unionOf\ ClassDescriptions) \mid \forall P.C\ (allValuesFrom) \mid$$
$$\exists P.C\ (someValuesFrom) \mid \exists P.\{x\}\ (hasValue) \mid$$
$$\{x_1, \ldots, x_n\}\ (oneOf) \mid >= nP\ (minCardinality) \mid <= nP\ (maxCardinality)$$

Tbox Axiom	**DL syntax**
SubClassOf	$A \sqsubseteq B$
equivalentClasses	$A \equiv B$
SubPropertyOf	$P \sqsubseteq Q$
equivalentProperty	$P \equiv Q$
disjointWith	$A \sqsubseteq \neg B$
inverseOf	$P \equiv Q^-$
transitiveProperty	$P^+ \sqsubseteq P$
symmetricProperty	$P \equiv P^-$
functionalProperty	$\top \sqsubseteq \leq 1P$
inverseFunctionalProperty	$\top \sqsubseteq \leq 1P^-$
domain	$\geq 1P \sqsubseteq A$
range	$\top \sqsubseteq \forall P.A$
Abox Axiom	**DL syntax**
class instance	$A(x)$
property instance	$P(x, y)$
sameAs	$x_1 \equiv x_2$

Table 1. DBOWL ontologies axioms

Table 1 shows the Tbox and Abox axioms for DBOWL ontologies. A and B are used for specifying Named Classes and C and D for specifying Class Descriptions. DBOWL assumes that all individuals are different unless the ontology includes an *owl:sameAs* assertion or you inferred it. This is important in real applications with a large number of individuals where usually it is easier to specify if two individuals represent the same resource than which individuals are different to others. The following restrictions are imposed on the DBOWL

ontologies. These restrictions are related more to the ontology syntax than to the ontology expressivity:

1. In the Tbox, all OWL constructors are supported. However, class descriptions always appear in the ontology as an equivalence or as a superclass of a Named Class. In an RDF/XML OWL ontology, class descriptions are always involved in the definition of a Named Class.
2. In the Abox, only assertions of Named Classes and Property Names are supported.
3. Only negation of Named Classes is allowed. Nevertheless, a negation of a class description could be included in the ontology defining a Named Class as equivalent to a Class Description and negating this Named Class.
4. Properties' domain and range must be Named Classes. In the same way, for asserting a complex property's domain or range we must define a Named Class as equivalent to a Class Description and use this Named Class as Property domain or range.
5. Only disjointness of Named Classes is allowed. As in the previous cases, a disjointness of a class description could be included in the ontology defining a Named Class as equivalent to a Class Description and disjoining this Named Class.

3 DBOWL Theoretical Foundations

In this section we present the theoretical foundations of our approach to scalable OWL reasoning. Although DBOWL is basically a Description Logic reasoner, it has been designed as an OWL reasoner. This implies that not all the DL inferences are supported. The main objective of DBOWL is to classify instances in Named Classes and Properties. In order to do this, for each Named Class and Property in the ontology, a *edb* relation $R_{A_1}(id), \ldots, R_{A_n}(id)$, $R_{P_1}(subject, object), \ldots, R_{P_m}(subject, object)$ is defined, being n and m the number of Named Classes and Properties in the ontology respectively. These relations contain one tuple for each individual or pair of individual asserted as member of such Named Class or Property.

3.1 Classification Function

In order to classify instances in Names Classes and Properties, we define a *classification function* $\mathcal{F}$ (see table 2). This function takes as input a DBOWL property axiom, a DBOWL domain or range axiom (see table 1), or an axiom $(A \equiv C)$, where C is a DBOWL class description and A is a Named Class in the ontology or an auxiliary name. The function define a new *idb* relation by means of a relational algebra expression, depending on the input type, or invoke the function with a new input.

For each Named Class in the ontology, a set of *idb* relations $S_{A_{i_0}}(id), \ldots, S_{A_{i_k}}(id), i : 1 \ldots n$ are defined. In the same way, for each Property in the ontology

a set of *idb* relations $S_{P_{j_0}}(subject, object), \ldots, S_{P_{j_l}}(subject, object)$, $j : 1 \ldots m$ are defined. The values of k and l depend on the number of axioms in the ontology evolving C_i and P_j respectively.

Each $S_{A_{i_x}}$, $x : 0 \ldots k$ has the following features (similarly for each $S_{P_{j_x}}$):

- $S_{A_{i_x}} = Q_{A_{i_x}}$, where $Q_{A_{i_x}}$ is a relational algebra expression,
- $S_{A_{i_0}} = R_{A_i}$,
- $S_{A_{i_{(x-1)}}}$ always occurs in Q_{i_x}, for $x : 1 \ldots k$, and
- if $S_{A_{j_r}}$ or $S_{P_{j_s}}$ occur in Q_{i_x}, they represent the last *idb* relation defined for A_j and P_j respectively.

3.2 Knowledge base Creation

In order to create the DBOWL knowledge base, function $\mathcal{F}$ is evaluate iteratively, defining the corresponding *idb* relations, until no new tuples are generated, i.e. until a fixed-point is reached. ($S_{A_{i_x}} = S_{A_{i_{(x-1)}}}$, $i : 1, \ldots, n$, and $S_{P_{j_x}} = S_{P_{j_{(x-1)}}}$, $j : 1, \ldots, m$).

In order to improve the efficiency of the evaluation, $\mathcal{F}$ is expressed as a composition of four functions, i.e.

$\mathcal{F} = \mathcal{F}_1 \circ \mathcal{F}_2 \circ \mathcal{F}_3 \circ \mathcal{F}_4$, where,

- $\mathcal{F}_1$ takes as input only axioms such as $P \sqsubseteq Q$, $P \equiv Q$, $P \equiv Q^-$, $P^+ \sqsubseteq P$, $P \equiv P^-$
- $\mathcal{F}_2$ takes as input only axioms such as $\geq 1P \sqsubseteq A$, $\top \sqsubseteq \forall P.A$
- $\mathcal{F}_3$ takes as input only axioms such as $A \sqsubseteq B$, $A \equiv B$, $A \equiv C \sqcap D$, $A \equiv C \sqcup D$, $A \sqsubseteq \forall P.C$, $A \equiv \exists P.C$, $A \equiv \neg B$, $A \equiv \{v_1, \ldots, v_n\}$, $A \equiv \exists P.\{v\}$
- $\mathcal{F}_4$ takes as input only axioms such as $A \equiv \exists P.\{v\}$

The algorithm proceeds as follows:

1. $\mathcal{F}_1$ is evaluated iteratively, defining the corresponding *idb* relations, until no new tuples are generated, i.e. until a fixed-point is reached. ($S_{P_{j_x}} = S_{P_{j_{(x-1)}}}$, $j : 1, \ldots, m$).
2. $\mathcal{F}_2$ is evaluated defining the corresponding *idb* relations ($S_{A_{i_x}}$, $i : 1, \ldots, n$).
3. $\mathcal{F}_3$ is evaluated iteratively defining the corresponding *idb* relations, until no new tuples are generated, i.e. until a fixed-point is reached. ($S_{A_{i_{(x+1)}}} = S_{A_{i_x}}$, $i : 1, \ldots, n$).
4. $\mathcal{F}_4$ is evaluated defining the corresponding *idb* relations ($S_{P_{j_{(x+1)}}}$, $j : 1, \ldots, m$).
5. Steps from 1 to 4 are repeated until no new tuples are generated by step 4, i.e. until a fixed-point is reached ($S_{P_{j_{(x+1)}}} = S_{P_{j_x}}$, $j : 1, \ldots, m$).

$\mathcal{F}(P \sqsubseteq Q)$	A new *idb* relation S_{Q_i} is defined as $\pi_{subject,object}(S_{Q_{(i-1)}}) \cup \pi_{subject,object}(S_{P_j})$.
$\mathcal{F}(P \equiv Q)$	A new *idb* relation S_{Q_i} is defined as $\pi_{subject,object}(S_{Q_{(i-1)}}) \cup \pi_{subject,object}(S_{P_j})$.
$\mathcal{F}(P \equiv Q^-)$	A new *idb* relation S_{P_i} is defined as $\pi_{subject,object}(S_{P_{(i-1)}}) \cup \pi_{object,subject}(S_{Q_j})$.
$\mathcal{F}(P^+ \sqsubseteq P)$	If (x, y) is a tuple in $S_{P_{(i-1)}}$ and (y, z) is also a tuple in $S_{P_{(i-1)}}$, then a new *idb* relation S_{P_i} is defined as $\pi_{subject,object}(S_{P_{(i-1)}}) \cup (\pi_{subject,object}((S_{P_{(i-1)}}) \bowtie_{object=subject} (S_{P_{(i-1)}})))$.
$\mathcal{F}(P \equiv P^-)$	A new *idb* relation S_{P_i} is defined as $\pi_{subject,object}(S_{P_{(i-1)}}) \cup \pi_{object,subject}(S_{P_{(i-1)}})$.
$\mathcal{F}(\geq 1P \sqsubseteq A)$	A new *idb* relation S_{A_i} is defined as $\pi_{id}((S_{A_{(i-1)}})) \cup \pi_{subject}((S_{P_j}))$.
$\mathcal{F}(\top \sqsubseteq \forall P.A)$	A new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{object}(S_{P_j})$.
$\mathcal{F}(A \sqsubseteq B)$	A new *idb* relation S_{B_i} is defined as $\pi_{id}(S_{B_{(i-1)}}) \cup \pi_{id}(A_j)$.
$\mathcal{F}(A \equiv B)$	A new *idb* relation S_{B_i} is defined as $\pi_{id}(S_{B_{(i-1)}}) \cup \pi_{id}(A_j)$.
$\mathcal{F}(A \equiv C \sqcap D)$	A new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup (\pi_{id}(\mathcal{F}(B \equiv C)) \cap \pi_{id}(\mathcal{F}(B \equiv D)))$.
$\mathcal{F}(A \sqsubseteq C \sqcap D)$	$\mathcal{F}(A \equiv C)$, $\mathcal{F}(A \equiv D)$
$\mathcal{F}(A \equiv C \sqcup D)$	A new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{id}(\mathcal{F}(B \equiv C)) \cup \pi_{id}(\mathcal{F}(B \equiv D))$.
$\mathcal{F}(A \equiv B \sqcup C)$	If $I \equiv \neg B$, a new *idb* relation S_X is defined as $\pi_{id}(S_{A_i}) \cap \pi_{id}(S_{I_j})$, $\mathcal{F}(X \equiv C)$
$\mathcal{F}(A \sqsubseteq \forall P.C)$	A new *idb* relation S_X is defined as $\pi_{object}(S_{A_i} \bowtie_{id=subject} S_{P_j})$. $\mathcal{F}(X \equiv C)$
$\mathcal{F}(A \equiv \exists P.C)$	A new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{id}(\mathcal{F}(X \equiv C)) \bowtie_{id=object} S_{P_j})$.
$\mathcal{F}(A \sqsubseteq \exists P.C)$	If P is a functional property, a new *idb* relation S_X is defined as $\pi_{subject}(S_{A_i} \bowtie_{id=subject} S_{P_j})$. $\mathcal{F}(X \equiv C)$
$\mathcal{F}(A \equiv \neg B)$	If $B \equiv \neg I$, a new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{id}(S_{I_j})$
$\mathcal{F}(A \equiv \neg B)$	If $I \equiv B \sqcup C)$, a new *idb* relation S_X is defined as $\pi_{id}(S_{A_i}) \cap \pi_{id}(S_{I_j})$, $\mathcal{F}(X \equiv C)$
$\mathcal{F}(A \equiv \neg B)$	If $I \equiv \neg A$, a new *idb* relation S_{B_i} is defined as $\pi_{id}(S_{B_{(i-1)}}) \cup \pi_{id}(S_{I_j})$
$\mathcal{F}(A \equiv \{v_1, \ldots, v_n\})$	A new *edb* relation $T(id)$ is defined where $r(T) = \{t_1, \ldots, tn\}$ and $t_i = v_i$, $i : 1 \ldots n$. Then a new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{id}(T)$.
$\mathcal{F}(A \equiv\leq nP)$	if (x, y_i), $i : 1..n$ are instances of P, and the y_i are all different, a new *edb* relation $T(id)$ is defined where $r(T) = \{t\}$ and $t = x$. Then a new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{id}(T)$.
$\mathcal{F}(A \equiv \exists P.\{v\})$	A new *idb* relation S_{A_i} is defined as $\pi_{id}(S_{A_{(i-1)}}) \cup \pi_{subject}(\sigma_{object=v}(S_{P_j}))$.
$\mathcal{F}(A \equiv \exists P.\{v\})$	A new *idb* relation S_{P_j} is defined as $\pi_{subject,object}(S_{P_{(j-1)}}) \cup \pi_{id,v}(\pi_{id}(S_{A_j}))$.

Table 2. DBOWL Classification Function

4 DBOWL Advantages and Limitations

DBOWL is implemented using Oracle 10g as Relational Database Management Systems. *edb* relations are tables in the database while *idb* relations are SQL views. A view in SQL terminology is a single table that is derived from other tables [4]. A view does not necessarily exists in physical form; it is considered a *virtual table* (non-materialized). The query defining the view is evaluated when needed. Then, once the knowledge base is created, for each Named Class and Property in the ontology there is a SQL view which defines the set of tuples (asserted and inferred) belonging to such Named Class or Property.

In order to query the knowledge base, SPARQL queries are re-written in terms of the SQL views and evaluated on the database. The names of the Named Classes and Properties involved in the query are changed by the corresponding SQL view name. Note that the queries defining the views are evaluated when the SPARQL query is evaluated. Thus, the inferred instances are not materialized in the database. Only some intermediate results are physically stored in the database, like the results of the transitive function. This non-materialized approach allows us to deal with billions of instances without the need of very large storage repositories. However, the main advantage of this approach is regarding updates. The non-materialization of the inferred instances permits the support of low-cost updates, as well as the possibility of implementing incremental reasoning algorithms.

Another important feature of DBOWL is the management of the *owl:sameAs* statement. At the end of each loop of the algorithm for the knowledge base creation, those individuals related by the *owl:sameAs* statement are included in the SQL views. DBOWL obtains the individuals related by the *owl:sameAs* statement as: (1) Those individuals explicitly asserted as *(x sameAs y)*; (2) By means of functional and inverse functional properties; (3) By means of the maxCardinality to 1 restriction.

DBOWL is complete with respect to the DBOWL knowledge base and the implemented functions, classifying all instances in Named Classes and Properties correctly. However, it presents some limitations:

As DBOWL separates Tbox and Abox reasoning, some inferences with nominals are lost. Fortunately, this information is not relevant for DBOWL because the objective of DBOWL is to classify instances in Named Classes and these inferences do not generate additional information for classification of instances.

DBOWL presents a problem regarding the open-world semantics of a DL Abox, which implies that an Abox has several models. The problem of exploring all the possible models in DBOWL is not trivial, even so, as DBOWL supports a large number of instances, it is logical to think that it could be very inefficient. Nevertheless, we plan to study how to provide a (partial) solution to this problem in the future.

Currently updates are not efficiently supported in DBOWL.

Finally, consistency checking of the knowledge base is not completely supported. Currently only the inconsistency caused by the classification of the same instance into (or the assertion of the same instance as member of) two disjoint

		Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
20MG	DBOWL	32	2512	666	383	200	165	19	303
	UOB	32	2512	666	383	200	165	19	303
100MG	DBOWL	35	11305	666	414	200	772	145	344
	UOB	35	11305	666	414	200	772	145	344
200MG	DBOWL	26	22833	666	389	200	1668	8	340
	UOB	26	22833	666	389	200	1688	8	340

		Query 9	Query 10	Query 11	Query 12	Query 13	Query 14	Query 15
20MG	DBOWL	1057	25	1930	65	379	6893	61
	UOB	1057	25	1930	65	379	6893	61
100MG	DBOWL	1041	29	6230	37	416	6913	73
	UOB	1041	29	6225	37	10503	6913	72
200MG	DBOWL	1039	25	4353	43	408	7088	79
	UOB	1039	25	4346	43	37577	7088	79

Fig. 1. Number of instances for each UOBM query

classes, and by the classification of one instance in a unsatisfiable class are implemented.

5 DBOWL Evaluation

In order to demonstrate practically the completeness of DBOWL we use the UOBM (University Ontology Benchmark) [2], a well known benchmark to compare repositories in the Semantic Web. This benchmark is intended to evaluate the performance of OWL repositories with respect to extensional queries over a large data set that commits to a single realistic ontology. Furthermore, the benchmark evaluates the system completeness and soundness with respect to the queries defined. This benchmark provides three OWL-DL ontologies, i.e. a 20, 100 and 200 Megabytes ontologies and the query results for each one. This experiment is conducted on a VMWARE virtual machine (one for each tool) with 8192 MB memory, running on a Windows XP 64 bits professional and java runtime environment build $1.6.0\_14 - b08$.

We evaluated the UOBM-DL queries for the 20, 100 and 200 Megabytes ontologies in DBOWL and obtained the correct results for all queries. Figure 1 presents the results for each ontology and for each query. As we can see, some DBOWL results are marked in a different color. This is because DBOWL and UOBM return different results for queries 11, 13 and 15. We checked the UOBM results for these queries and we believe that they are incorrect. For query 11 DBOWL returns more results than UOBM. In the case of queries 11 and 15, it is because several owl:sameAs relationships between some UOBM individuals can be inferred. Therefore, these individuals should be in the result. In the case of query 13, it is because the UOBM result includes instances of all departments, but query 13 asks only for instances in department0. Figure 2 presents the response times for the UOBM-DL 200 Megabytes ontology.

We have also evaluated DBOWL using the standard datasets provided in the context of the ORE 2012 workshop. These datasets include a set of state

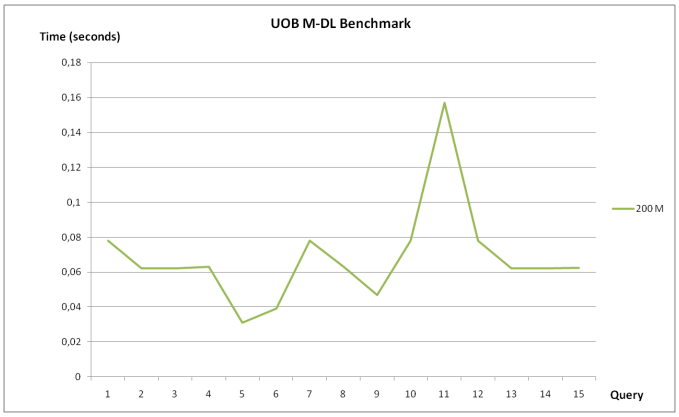


Fig. 2. Response times for UOBM-DL 200M ontology

of the art ontologies in OWL 2 language, both in RDF/XML and Functional syntax. and they are organised by reasoning services, i.e. Classification, Class satisfiability, Ontology satisfiability, Logical entailment and non entailment and Instance retrieval. DBOWL uses Pellet in order to classify the ontology Tbox and to check the class satisfiability. Therefore, datasets corresponding to these reasoning services are not included in our evaluation. As the main objective of DBOWL is to classify instances in Named Classes and Properties, we evaluate DBOWL using the Instance Retrieval test cases. Some of the ontologies in these datasets present unsatisfiable classes. We use these ontologies to test the behavior of DBOWL in such cases. However, the total time taken to load and test the satisfiability of one ontology and the satisfiability result is reported by Pellet. DBOWL only stores in the database the classes that Pellet returns as unsatisfiable. When DBOWL classifies an instance in a unsatisfiable class, it returns that the ontology is inconsistent, via a simple SQL query to the relational database. Thus, the performance of the ontology classification reasoning service falls on Pellet. Finally, as DBOWL is an OWL-DL reasoner, we use the OWL-DL Instance Retrieval test case for the evaluation.

This experiment has been carried out in two phases. In the first step, we loaded the nine ontologies in DBOWL. Most of the ontologies could not be loaded due to different problems: (1) Some ontologies are not valid DBOWL ontologies (see section 2.3). Information_397.owl, minswap.owl, and people.owl define complex property's domains or range (different from named Classes). Information_397.owl and people2.owl contain complex Abox assertions (different from Named Classes assertions). Finally, obi.owl cannot be classified by Pellet because of memory problems. (2) DBOWL presented some problems dealing with unsatisfiable classes, because the storage and management of the class *Nothing* was not completely implemented. (3) DBOWL presented some problems regarding the management of the namespaces.

Ontology Name	Class description	Load time (seconds)	Instance Retrieval time (seconds)	Number of Instances returned	Unsatisfiable classes found
artontology.owl	http://mr.teknowledge.com/DAML/artontology.daml#_anon0	6610	0.023	3	
food.owl	http://www.w3.org/TR/2003/CR-owl-guide-20030818/food#Grape	23541	0.06	17	
information_397.owl	http://www.loa-cnr.it/ontologies/ExtendedDnS#role	79876	0.1	10	
MGEDOntology.owl	http://mged.sourceforge.net/ontologies/MGEDOntology.owl#MethodologicalFactorCategory	45526	0.114	32	
mindswap.owl	http://xmlns.com/wordnet/1.6/Person	17718	0.17	38	
obi.owl	http://purl.obofoundry.org/obo/IAO_0000033	—	—	—	
people.owl	http://cohse.semanticweb.org/ontologies/people#newspaper	26702	0.037	4	&people; Mad+cod
people+pets.owl	http://cohse.semanticweb.org/ontologies/people#animal	27410	0.044	15	&people; Mad+cod
travel.owl	http://www.owl-ontologies.com/travel.owl#RuralArea	12877	0.037	4	&travel;Safari

Fig. 3. Results for OWL-DL Instance Retrieval dataset

In the second step, we solved the aforementioned problems and we loaded eight of the nine ontologies in DBOWL (obi.owl could not be loaded because it presented a problem with Pellet) and we obtained the corrected result for all of them. We followed the guidelines outlined in Section 2.3 in order to convert the ontologies into DBOWL ontologies. Figure 3 summarizes the results of the evaluation. Load time includes Tbox classification (Pellet), database creation, ontology storage and knowledge base creation (instances classification).

6 Conclusions

From the evaluation we extract some general conclusions. To the best of our knowledge, DBOWL is the only OWL reasoner able to deal with the three UOBM-DL ontologies obtaining the correct results for all queries in all cases. Furthermore, this allows us to check the UOBM results for queries 11, 13 and 15 and to conclude that they are incorrect. Finally, DBOWL response times are very good the highest one being 0.328 seconds for the UOBM 200MB ontology. The results obtained with both evaluations suggest that DBOWL is a real complement to current OWL reasoners. Currently, DBOWL supports ontologies with much bigger Aboxes than traditional systems based on description logic and satisfiability. This is especially important for some applications such as life sciences, where particularly large ontologies are used. The datasets provided in the context of the ORE 2012 workshops have allowed us to improve DBOWL in several ways. Thus, the latest version of DBOWL is able to deal with all types of namespaces, to control when a class is non-satisfiable and to check the ontology consistency in such a case. Furthermore, we empirically test that the restrictions imposed on the DBOWL ontologies are not a problem for developing real ontologies, because any ontology can be converted to a DBOWL ontology, keeping the ontology expressivity. With respect to instance retrieval, DBOWL is

able to obtain the same results as the expected result provided by the OWL-DL Instance Retrieval dataset, suggesting that the DBOWL classification functions and the algorithm for creating the knowledge base work well.

The use of a relational database to store the ontologies implies that the time for loading an ontology in DBOWL can be longer than the load time in main-memory reasoners. The advantage of our approach is that, once the knowledge base is created, the query time is really small. Furthermore, as the knowledge base is persistent, you can query it at any moment without creating it again. Although other approaches also provide solutions for instance retrieval, they present some problems regarding reasoning expressivity or response query times. SHER [6], is a platform developed by IBM which supports sound and complete reasoning for the fragment of OWL 1 DL without nominals. SHER adopts a modularisation-based approach in which the ontology breaks into small parts and is reasoned with a DL reasoner in the main memory. After the reasoning procedure is finished, the corresponding axioms are stored in the database. Reasoning with instances is performed at query time. Oracle 11g [7] is the laster version of the extremely well known RDMS Oracle. Oracle 11g includes a native inference engine able to handle a subset of OWL called OWLPrime which covers part of OWL Lite and a little part of OWL 1 DL. It also supports querying of RDF/OWL data using SPARQL-like graph patterns embedded in SQL.

As for future work, we are studying some optimisation techniques (such as database indexes, parallel computation and incremental reasoning) in order to improve the response times of the queries. We also are studying the possibility of incorporating other OWL reasoners different from Pellet, in DBOWL. The idea is to select the most convenient OWL reasoner depending on the ontology expressivity and size.

7 Acknowledgements

This work is supported by the Project Grant TIN2011-25840 (Spanish Ministry of Education and Science) and P11-TIC-7529 (Innovation, Science and Enterprise Ministry of the regional government of the Junta de Andalucía).

References

1. Ullrich Hustadt , Boris Motik , Ulrike Sattler. *Reasoning in Description Logics by a Reduction to Disjunctive Datalog*. Journal of Automated Reasoning, v.39 n.3, p.351-384, October 2007.
2. Ma, L; Yang, Y; Qiu, Z; Xie, G; Pan, Y. *Towards A Complete OWL Ontology Benchmark*. In. Proc. of the 3rd European Semantic Web Conference (ESWC 2006).
3. Codd, E. *A relational Model for Large Shared Data Banks*, CACM, 13:6, june 1970.
4. Abiteboul, S., Hull, R., Vianu, V. *Foundations of Databases*. Addison-Wesley Publishing Company. 1995.

[6] http://domino.research.ibm.com/comm/research_projects.nsf/pages/iaa.index.html
[7] http://www.oracle.com

Advancing the Enterprise-class OWL Inference Engine in Oracle Database

Zhe Wu, Karl Rieb, George Eadon
Oracle Corporation
{alan.wu, karl.rieb, george.eadon}@oracle.com

Ankesh Khandelwal, Vladimir Kolovski
Rensselaer Polytechnic Institute, Novartis Institutes for Biomedical Research
ankesh@cs.rpi.edu, vladimir.kolovski@novartis.com

Abstract.

OWL is a standard ontology language defined by W3C that is used for knowledge representation, discovery, and integration. Having a solid OWL reasoning engine inside a relational database system like Oracle is significant because 1) many relational techniques, including query optimization, compression, partitioning, and parallel execution, can be inherited and applied; and 2) relational databases are still the primary place holder for enterprise information and there is an increasing use of OWL for representing such information. Our approach is to perform data intensive reasoning as close as possible to the data. Since 2006, we have been developing an RDBMS-based large scale and efficient forward-chaining inference engine capable of handling RDF(S), OWL 2 RL/RDF, SKOS, and user defined rules. In this paper, we discuss our recent implementation and optimization techniques for query-rewrite based OWL 2 QL reasoning, named graph-based inference (local inference), and integration with external OWL reasoners.

1 Introduction

OWL [1] is an important standard ontology language defined by W3C and it has a profound use in knowledge representation, discovery, and integration. To support OWL reasoning over large datasets we have developed a forward-chaining rule-based inference engine [2] on top of the Oracle Database. By implementing the inference engine as a database application we are able to leverage the database's capabilities for handling large scale data. The most recent release of our engine, with support for RDFS, OWL 2 RL/RDF [3], SKOS and user-defined rules, is available as part of Oracle Database 11g Release 2 [4, 5, 7].

In our system semantic data is stored in a normalized representation, with one table named LexValues providing a mapping between lexical values and integer IDs and

another table named IDTRIPLES enumerating triples or quads in terms of IDs, similar to other systems [8, 9]. Inference engine rules are translated to SQL and passed to Oracle's cost-based optimizer for efficient execution. For notational convenience, SQL queries in this paper are written in terms of placeholders ID(x) and <IVIEW>. In the implementation ID(x) is replaced by the ID for the given lexical value x, which can be found by querying our LEXVALUES table, and <IVIEW> is replaced by an inline view that unions the relevant triples (or quads) in IDTRIPLES with the inferred triples (or quads) computed so far.

Additional features of our inference engine include: (1) for built-in OWL constructs we manually craft the SQL and algorithms that drive the inference, optimizing for special cases including transitive properties and equivalence relations such as owl:sameAs, (2) user-defined rules are translated to SQL automatically, (3) we leverage Oracle's parallel SQL execution capability to fully utilize multi-CPU hardware, and (4) we efficiently update the materialized inferred triples after additions to the underlying data model, using a technique based on semi-naive evaluation [7].

Our engine has been used in production systems since 2006, and has proven capable of handling many real-world applications. However, challenges remain:

- Despite support for efficient incremental inference, fully materializing inferred results via forward chaining can be a burden, especially for large frequently-updated data sets. Therefore we are introducing backward-chaining into our system with a query-rewrite-based implementation of OWL 2 QL reasoning [2].
- Some applications need to restrict inference to a single ontology represented by a named graph. For these applications inference should apply to *just* the assertions in each named graph and a common schema ontology (TBox). This kind of inference is therefore local, as opposed to the traditional global inference, and it is called Named Graph based Local Inference (NGLI) by Oracle.
- Some applications need the full expressivity of OWL 2 DL. To satisfy these applications, we have further extended our inference engine by integrating it with third-party complete OWL 2 DL reasoners like PelletDB [6].

In this paper we present our recent advances. Section 2 describes our query-rewrite implementation of OWL 2 QL reasoning. Section 3 describes our implementation of named graph local inference. Section 4 describes integration with external third-party OWL reasoners. Section 5 presents a performance evaluation using synthetic Lehigh University Benchmark (LUBM) datasets. Section 6 describes related work. Finally, Section 7 concludes this paper.

2 Support of OWL 2 QL in the Context of SPARQL

OWL 2 QL is based on the DL-Lite family of Description Logics, specifically DL-Lite*R* [10]. OWL 2 QL is an important profile because it has been designed so that data (Abox) can be queried through an ontology (Tbox) via a simple rewriting mechanism. Queries can be expanded to include the semantic information in Tbox, using query-rewrite techniques such as the PerfectRef algorithm [10], before executing them against the Abox. The query expansion could produce many complex queries, which presents a challenge for scalable QL reasoning. There have been several pro-

posals for optimizing and reducing the size of rewritten queries; see Section 6 for a brief discussion. Most of these techniques return a union of conjunctive queries (UCQ) for an input conjunctive query (CQ). Rosati et.al. [11] proposed a more sophisticated rewriting technique, the `Presto` algorithm, that produces non-recursive datalog (nr-datalog), instead of UCQ. OWL 2 QL inference is supported in the Oracle OWL inference engine based on the Presto algorithm. We elaborate on this and other optimizations for efficient executions of query rewrites in Section 2.1. To meet the requirements of enterprise data, OWL 2 QL inference engine must handle arbitrary SPARQL queries. We discuss some subtleties to handling arbitrary SPARQL queries by query expansion in Section 2.2. We will be using the following OWL 2 QL ontology to illustrate various concepts. It is described in functional syntax and has been trimmed down for brevity.

```
Ontology (SubDataPropertyOf(:nickName  :name)
    SubClassOf(:Married ObjectSomeValuesFrom(:spouseOf :Person))
    SubObjectPropertyOf(:spouseOf :friendOf)
    DataPropertyAssertion(:name :Mary "Mary")
    ClassAssertion(:Married :John)
    DataPropertyAssertion(:name :Uli "Uli") )
```

2.1 Optimizing Execution of Query Rewrites

As noted in the introduction, Oracle OWL inference engine implements an approach that is similar to that of rewriting CQs as nr-datalog. We will illustrate our approach through some examples. Consider the conjunctive query (CQ),

```
  select ?x ?y ?n where { ?x  :friendOf  ?y .   ?y  :name  ?n },
```

and its equivalent datalog query `q(?x, ?y, ?n) :- friendOf(?x, ?y), name(?y, ?n)`. Note that `friendOf(?x, ?y)` and `name(?y, ?n)` are referred to as atoms of the query. `q(?x, ?y, ?n)` can be translated into following nr-datalog using the `Presto` algorithm, and the example Tbox.

```
  q(?x, ?y, ?n) :- q1(?x, ?y), q2(?y, ?n) .
  q1(?x, ?y)    :- friendOf(?x, ?y) .
  q1(?x, ?y)    :- spouseOf(?x, ?y) .
  q2(?y, ?n)    :- name(?y, ?n) .
  q2(?y, ?n)    :- nickName( ?y, ?n) .
```

The nr-datalog can be represented by a single SPARQL query as shown below. The heads of the nr-datalog, `q1(?x, ?y)` and `q2(?y, ?n)` for example, define a view for the atoms of the query, which can be represented via UNION operation. The conjunctions in the body of nr-datalog rules can be represented via intersections of views. We refer to the resulting form of SPARQL query as the Joins of Union (JoU) form. (UCQs, in contrast, are of the form Unions of Joins (UoJ).)

```
select ?x ?y ?n where { {{?x  :friendOf ?y} UNION {?x  :spouseOf ?y}}
                        {{?y  :name  ?n}    UNION {?y  :nickName ?n}}}
```

The JoU form of SPARQL queries generated from query rewrite often contains many UNION clauses and nested graph patterns that can become difficult for the query optimizer to optimize. To improve the quality of query plans generated by the

query optimizer, our latest inference engine rewrites the UNION clauses using FILTER clauses. For example, the query above is rewritten as follows, using `sT(...)` as notational shorthand for the SPARQL operator `sameTerm(...)`. Recall that `sameTerm(A, B)` is true if `A` and `B` are the same RDF term.

```
select ?x ?y ?n where
 {{?x ?p1 ?y FILTER(sT(?p1, :friendOf) || sT(?p1, :spouseOf ))}
  {?y ?p2 ?n FILTER(sT(?p2, :name)     || sT(?p2, :nickName))}}.
```

As mentioned earlier, UNION clauses correspond to views for atoms in the query. Rules in the nr-datalog that define view for an atom of the query contain single atom in their body [11]. The former atom is entailed by the latter atoms. For example, the (SPARQL) CQ atom `{?x p ?y}` can be entailed from atoms of types `{?x q ?y}` (by sub-property relationships) and `{?y q ?x}` (by inverse relationships), and also of type `{?x rdf:type c}` (by existential class definition) if `?y` is a non-distinguished non-shared variable. There may be more than one atom for each type. In that case, the view corresponding to atom `{?x p ?y}` can be defined via filter clauses as follows. (Any of `n2`, `n3` may be zero in which case corresponding pattern is omitted; $q1_1$ equals `p`.)

```
{{?x ?q ?y FILTER( sT(?q, q1₁) || ... sT(?q, q1ₙ₁))} UNION
 {?y ?q ?x FILTER( sT(?q, q2₁) || ... sT(?q, q2ₙ₂))} UNION
 {?x rdf:type ?c FILTER( sT(?c, c1₁) || ... sT(?c, c1ₙ₃))}}
```

Note that the unions above can be further collapsed using more general filter expressions and the result of query-rewrite is an expanded query (that is no rules are generated). A key benefit of treating UCQ as JoU is that the SQL translation of a JoU query is typically more amenable to RDBMS query optimizations because it uses fewer relational operators, which reduces the optimizer's combinatorial search space; and the JoU, together with the filter clause optimization, will typically execute more efficiently in an RDBMS because the optimizer can find a better plan involving fewer operators, which reduces runtime initialization and destruction costs. Take the Lehigh Benchmark (LUBM) Query 5 and 6 for example. The JoU approach takes both less time and fewer database I/O requests, as shown in the following table, to complete the query executions against the 1.1 billion-assertion LUBM 8000 data set. The machine used was a Sun M8000 server described in Section 5.

LUBM8000 (1.1B+ asserted facts)	**JoU with FILTER Optimization**		**No Optimization**	
	Time	**# of DB I/O**	**Time**	**# of DB I/O**
Q5 (719 matches)	98.9s	73K	171.1s	271K
Q6 (63M+ matches)	25.68s	48K	28.7s	73K

Table 1 Effectiveness of JoU with FILTER optimization

2.2 SPARQLing OWL 2 QL Aboxes

The main mechanism for computing query results in the current SPARQL standard is subgraph matching, that is, simple RDF entailment [12]. Additional RDF statements

can be inferred from the explicit RDF statements of an ontology using semantic interpretations of ontology languages such as OWL 2 QL. The next version of SPARQL (SPARQL 1.1) is in preparation and various entailment regimes have been specified that define basic graph pattern matching in terms of semantic entailment relations [13]. One such entailment regime is the OWL 2 Direct Semantics Entailment Regime (ER), which is relevant for querying OWL 2 QL Aboxes. The ER specifies how the entailment is used.

The entailed graphs may contain new blank nodes (that are not used in the explicit RDF statements). ER, however, restricts semantic entailments to just those graphs which contain no new blank nodes. In other words, all the variables of the query are treated as distinguished variables irrespective of whether they are projected. This limits the range of CQs that can be expressed using SPARQL 1.1 ER. For example, consider following two queries that differ only in projected variables.

```
select ?s ?x  { ?s  :friendOf ?x .}
select ?s     { ?s  :friendOf ?x .}
```

Per ER, both queries have empty results. However, if viewed as CQs, `?x` is a non-distinguished variable in the second query and `[?s → :John]` is a valid result. Therefore, the second CQ cannot be expressed in SPARQL 1.1 ER.

For practical reasons, we would like to be able to express all types of CQs to OWL-2 QL Aboxes using SPARQL, especially when there are well-defined algorithms such as `PerfectRef` for computing sound and complete answers for CQs. We thereby adopt, in addition to ER, another entailment regime for Abox queries to OWL 2 QL ontologies, namely *OWL 2 QL Entailment Regime (QLER)*. QLER is similar to ER except that non-projected variables can be mapped to new blank nodes (not specified in the explicit triples of the Abox or Tbox). Projected variables cannot be mapped to new blank nodes under both ER and QLER. QLER, unlike ER, is defined only for Abox queries, and property and class expressions are not allowed (that is, only concept and property IRIs may be used). Note that the results obtained under ER are always a subset of the results obtained under QLER.

Now, any CQ can be expressed as BGP SPARQL query under QLER (unlike ER). The BGP query can be expanded, as discussed in Section 2.1, such that the query results can be obtained from the expanded query by standard subgraph matching. SPARQL, however, supports more complex queries than BGPs and union of BGPs such as accessing graph names, filter clauses, and optional graph patterns. Thus, a query-rewrite technique for complex SPARQL queries is also required.

Under ER, since all variables are treated as distinguished variables, individual BGPs of a complex query can be expanded separately and replaced in place. For example, SPARQL query select ?s ?n {?s :friendOf ?x} OPTIONAL {?x :name ?n} can be expanded as,

```
select ?s ?n {{ {?s :friendOf ?x} UNION {?s :spouseOf ?x} }
    OPTIONAL {{ ?x :name ?n } UNION {?x :nickname ?n}}}.
```

This expansion strategy is, however, not valid under QLER. `?x` is a non-distinguished variable under QLER, and the expanded form by that strategy will be,

```
select ?s ?n {{{?s :friendOf ?x}
   UNION {?s :spouseOf ?x} UNION {?s rdf:type  :Married} }
OPTIONAL { { ?x :name ?n } UNION {?x :nickname ?n}}.
```

The query above produces two incorrect answers, [?s → :John; ?n → "Uli"] and [?s → :John; ?n → "Mary"], and the source of the incorrect answers is that binding [?s → :John] is obtained from the pattern {?s rdf:type :Married}, and then ?x, which is implicitly bound to some new blank node, is explicitly bound to nothing (that is ?x is null). A left outer join with bindings from {?x :name ?n } produces erroneous results because null value matches any value of ?x from the optional pattern.

The way around that problem is to bind ?x to a new blank node using SPARQL 1.1 assignment expression [14] BIND(BNODE(STR(?s)) AS ?x) as shown below. The BGP { ?s :friendOf ?x } is expanded into

```
{{ ?s :friendOf ?x }  UNION { ?s  :hasSpouce ?x } UNION
  { ?s rdf:type :Married . BIND(BNODE(STR(?s)) AS ?x). }}.
```

The bindings for a non-distinguished variable are also lost when two similar atoms of a CQ are replaced by their most general unifier; cf. reduction step of the PerfectRef algorithm. Let ?x be a non-distinguished variable that is unified with term t of other atom, which may be a variable or a constant, then the binding for ?x can be retained by using SPARQL 1.1 assignment expression BIND(t as ?x), in a manner similar to that used in the above example.

So, the query-rewrite technique for complex SPARQL queries under QLER consists of the following steps: 1) identify distinguished variables for all BGPs of the query, 2) expand BGPs separately using the standard query-rewrite techniques (for CQs), including the one described in Section 2.1, 3) make the bindings for non-distinguished variables explicit whenever they are not using SPARQL 1.1 assignment expressions as discussed above, and 4) replace the expanded BGPs in place. Steps 2) and 3), even though presented sequentially, are intended to be performed concurrently. That is the bindings may be made explicit in the expansion phase for BGPs.

3 Named-Graph based Local inference

Inference is typically performed against a complete ontology together with all the ontologies imported via owl:imports. In this case inference engines consolidate all of the information and then perform tasks like classification, consistency checking, and query answering, thereby maximizing the discovery of implicit relationships.

However, some applications need to restrict inference to a single ontology represented by a named graph. For example, a health care application may create a separate named graph for each patient in its system. In this case, inference is required to apply to *just* the assertions about each patient and a common schema ontology (TBox). This kind of inference is therefore local, as opposed to the traditional global inference, and it is called Named Graph based Local Inference (NGLI) by Oracle. NGLI together with the use of named graphs for asserted facts modularizes and improves the manageability of the data. For example, one patient's asserted and inferred information can be updated or removed without affecting those of other patients. In addition, a

modeling mistake in one patient's named graph will not be propagated throughout the rest of the dataset.

One naïve implementation is to run the regular, global, inference against each and every named graph separately. Such an approach is fine when the number of named graphs is small. The challenge is to efficiently deal with thousands, or tens of thousands of named graphs. In the existing forward-chaining based implementation, Oracle database uses SQL statements to implement the rule set defined in the OWL 2 RL specification. To add the local inference feature, we have considered two approaches. The first approach re-implements each rule by manually adding SQL constructs to limit joins to triples coming from the same named graphs. Take for example a length 2 property chain rule defined as follows:

```
?u1 :p1 ?u2, ?u2 :p2 ?u3    ➔   ?u1 :p ?u3
```

This rule can be implemented using the following SQL statement. Obviously this rule applies to all assertions in the given data set <IVIEW>, irrespective of the origins of the assertions involved.

```
select distinct m1.sid, ID(p), m2.oid  from <IVIEW> m1, <IVIEW> m2
 where m1.pid=ID(p1) and m1.oid=m2.sid and m2.pid=ID(p2)
```

To extend the above SQL with local inference capability, the following additional SQL constructs (in *Italic* font) are added. The assumption here is that <IVIEW> has an additional column, `gid`, which stores the integer hash ID values of graph names. Also, as a convention, the common schema ontology is stored with a NULL gid value in the same <IVIEW>. This allows an easy separation of the common schema ontology axioms from those assertions made and stored in named graphs.

```
select distinct m1.sid, ID(p), m2.oid,  nvl(m1.gid,m2.gid) AS gid
  from <IVIEW> m1, <IVIEW> m2
 where m1.pid=ID(p1) and m1.oid=m2.sid and m2.pid=ID(p2)
   and ( m1.gid = m2.gid or m1.gid is null or m2.gid is null)
```

In the above SQL statement, a new projection of gid column is added to tag each inferred triple with its origin. This is very useful provenance information. Also, a new Boolean expression is added to the end of the SQL statement. This new expression enforces that the two participating triples must come from the same named graph or one of them must come from the common schema ontology. Note that when dealing with more complex OWL 2 RL rules, the number of joins increases and this additional Boolean expression becomes more complicated. As a consequence, it is error prone to manually modify all existing SQL implementations to support the local inference. This motivated an annotation-based approach, where each existing SQL statement is annotated using SQL comments. Using the above example, the annotation (in Italic font) together with the original SQL statement looks like:

```
select distinct m1.sid, ID(p), m2.oid    /* ANNOTATION: PROJECTION */
  from <IVIEW> m1, <IVIEW> m2  where m1.pid=ID(p1) and m1.oid=m2.sid
   and m2.pid=ID(p2)           /* ANNOTATION: ADDITIONAL_PREDICATE */
```

At runtime, the above dummy annotation texts will be replaced with proper SQL constructs, similar to those described before. Those automatically-generated SQL constructs are based on the number of joins in a rule implementation, and the set of

view aliases used in the SQL statement. Compared to the first approach, this annotation based approach is easier to implement and much more robust because all the actual SQL changes are centralized in a single function.

4 Extensible Inference

We realize in practice that, to be enterprise ready, an inference engine has to be extensible. Our engine natively supports RDFS, SKOS, OWLPrime [3], OWL 2 RL which is a rich subset of the OWL 2 semantics, and a core subset of OWL 2 EL that is sufficient to classify the well-known SNOMED ontology, in addition to user-defined positive Datalog-like forward-chaining rules to extend the semantics and reasoning capabilities beyond OWL. This is sufficient to satisfy the requirements of many real-world applications. However, some application domains need the full expressivity of OWL 2 DL. To satisfy these applications, we have further extended our inference engine by integrating it with third-party complete OWL 2 DL reasoners like PelletDB [6]. A key observation has been that even when dealing with a large-scale dataset which does not fit into main memory, the schema portion, or the TBox, tends to be small enough to fit into physical memory. So the idea is to extract the TBox from Oracle database via a set of Java APIs provided in the Jena Adapter [5], perform classification using the in memory DL reasoner and materialize the class and property hierarchies, save them back into Oracle, and finally invoke Oracle's native inference API to perform reasoning against the instance data, or the ABox.

This approach combines the full expressivity support provided by an external OWL 2 DL reasoner and the scalability of Oracle database. Such an approach is general enough and can be applied to other well-known OWL reasoners including Fact++, HermiT, and TrOWL. It is worth pointing out that such an extension to Oracle's inference capability is sound, but completeness in terms of query results cannot be guaranteed. Nonetheless, users welcome such an extension because 1) in-memory solutions simply cannot handle a very large dataset that exceeds the memory constraint, 2) more implicit relationships are made available using additional semantics provided by external reasoners.

5 Performance Evaluation

In this section, we evaluate the performance of Oracle's native inference engine. Most tests were performed on a SPARC Enterprise M8000 server with 16 SPARC 64 VII+ 3.0GHz CPUs providing a total of 64 cores and 128 parallel threads. There is 512 GB RAM and two 1-TB F5100 flash arrays incorporating 160 storage devices. Note that the performance evaluation is focused on local inference performance. A systematic evaluation of SPARQL query answering under QL semantics is ongoing.

Benchmark Data Generation. We are using the well-known, synthetic Lehigh University Benchmark (LUBM) to test the performance because 1) a LUBM dataset can be arbitrarily large; and 2) it is quite natural to extend a LUBM dataset from tri-

ples to quads. The existing LUBM data generator produces data in triple format. However, the triples are produced on a per university basis, so it is straightforward to append university information to the triples to yield quad data.

Local Inference Performance. In the following table, we compare the performance of named-graph based local inference against that of the regular, global inference. Three benchmark datasets are used and the dataset size is between 133 million and 3.45+ billion asserted facts. Such a scale is sufficient for many enterprise-class applications. The second and third columns list the number of inferred triples and elapsed time for global inference. The last two columns list the number of inferred new quads and elapsed time for local inference. Note that a parallel inference [7] with a degree of 128 was used for both the global and local inferences. In addition, at the end of both global and local inference process, a multi-column B-Tree index is built so that the inferred data is ready for query. The only factor that stopped us from testing even bigger ontologies was the 2-TB disk space constraint.

Benchmark/Inference Type	Global Inference		Local Inference	
	New triples	Elapsed time	New quads	Elapsed time
LUBM1000[1]	108M	12m 15s	111M	13m 0s
LUBM8000	869M	33m 17s	892M	40m 3s
LUBM25000	2.71B	1h 44m	2.78B	2h 1m

Table 2 Performance comparison between global and local inference

The performance of local inference is a bit slower than but still quite comparable to that of the global inference. The performance difference comes from two places: 1) local inference deals with quads instead of triples and a quad dataset is larger in size than its triple counterpart because of the additional graph names, 2) the SQL statement is more complex due to the additional expressions.

It may be counter intuitive that local inference produced more inferred relationships than global inference. An examination of the inference results suggests that there are inferred triples showing multiple times in different named graphs even though any named graph contains only a unique set of triples.

With help from a customer, we conducted a performance evaluation of local inference using OWL 2 RL profile against large-scale *real-world* data[2] from the medical/hospital domain. The machine used was a quarter-rack Exadata x2-2. It is a 2-node cluster and each node has 96 GB RAM and 24 CPU cores. Detailed hardware specifications can be found here[3]. It took around 100 minutes to complete the local inference using a parallel degree of 48. Inference generated a total of 574 million new quads.

Benchmark/Inference Type	Local Inference	
	New quads	Elapsed time
Real-world Medical/Hospital Dataset	574M	100m 28s

Table 3 Local inference performance against real-world quad dataset

[1] LUBM1000, LUBM8000, and LUBM25000 datasets have 133M+, 1.1B+, and 3.45B+ facts asserted, respectively.

[2] Private data. It has 1.163B+ quads asserted.

[3] http://www.oracle.com/technetwork/database/exadata/dbmachine-x2-2-datasheet-175280.pdf

Parallel Inference. Oracle's inference engine has benefited greatly from the parallel execution capabilities provided by the database. The same kind of parallel inference optimization, explained in [7], applies both to the regular, global inference and the local inference. Figure 1 shows the local inference performance improvement as the degree of parallelism goes higher. LUBM 25K benchmark was used for this experiment. Note that most improvement was achieved when the parallel degree went up from 24 to 64. After that, only marginal improvement was observed. This is due to the fact that the Sun M8000 has 64 cores.

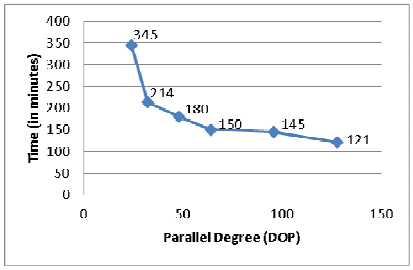


Figure 1 Local inference elapsed time versus degree of parallelism

6 Related Work

We will discuss works related to query rewriting as required for OWL 2 QL inference, implementations for OWL 2 QL reasoning. (We will be focusing on theory and techniques and not so much on relative performances).

Several techniques for query-rewriting have been developed since the `PerfectRef` algorithm was introduced in [10]; see [15] for a nice summary. The given CQ is reformulated as a UCQ by means of a backward-chaining resolution procedure in the `PerfectRef` algorithm. The size of the computed rewriting increases exponentially with respect to the number of atoms in the given query, in the worst case. But as observed by others, many of the new queries that were generated were superfluous, for example some of the CQs in a UCQ may be subsumed by others in the UCQ. An alternative resolution-based rewriting technique was proposed in [16] which avoids many useless unifications and thus UCQs are smaller even though they are still exponential in the number of atoms of the query. This alternative rewriting technique is implemented in the Requiem system[4]. Rosati et al. [11] argued that UCQs are reasons for exponential blow up, and have proposed a very sophisticated rewriting technique, the Presto algorithm, which produces a non-recursive Datalog program as a rewriting, instead of a UCQ. As noted before, we deploy the Presto algorithm for optimal performance.

The W3C's OWL implementations page[5] lists four systems that support OWL 2 QL reasoning: QuOnto[6], Owlgres[7], OWLIM[8], Quill[9].

[4] http://www.cs.ox.ac.uk/isg/tools/Requiem/

[5] http://www.w3.org/2007/OWL/wiki/Implementations

[6] http://www.dis.uniroma1.it/~quonto/

QuOnto, Quill and Owlgres implement the `PerfectRef` query-rewrite technique, but Quonto implements an optimized `PerfectRef` query-rewrite technique, `QPerfRef` [10], and Quill in addition to query-rewrite, transforms ontology into a semantically approximate ontology [17].

Owlgres is an RDBMS-based implementation [18]. It deploys PerfectRef query-rewrite technique, with some optimizations such as Tbox terms with zero occurrences in Abox are identified in a preprocessing step and CQs of a UCQ that contain such Tbox terms are discarded, in contrast to the Presto algorithm. Furthermore, the UCQs are translated into a single SQL query that is a union of SQL queries, which is reminiscent of UoJ form. In contrast, we translate UCQs into more efficient JoU form, and the unions are collapsed into compact FILTER clauses.

OWLIM supports forward-chaining style rule-based reasoning, wherein blank nodes can be inferred during rule evaluation. OWL 2 QL reasoning is supported in OWLIM by defining new ruleset that captures OWL 2 QL semantics [19], and using the same forward chaining mechanism.

7 Conclusions

This paper described the recent advances in our OWL inference engine, which is implemented on top of the Oracle Database. We described optimizations for rewrite-based backward-chaining implementation of OWL 2 QL. We showed that conjunctive queries for OWL 2 QL knowledge bases cannot be expressed in SPARQL 1.1 using its entailment regimes because the regimes are very restrictive towards bindings to new blank nodes. We introduced a new regime to overcome that and described a query-rewrite technique for general SPARQL queries (which may contain constructs such as optional graph patterns). We introduced the concept of "named-graph based local inference" and described our implementation. We described the motivation for integrating a third-party OWL reasoner in our system, and described our implementation. Finally, we evaluated the performance of named-graph based local inference as compared to traditional global inference on synthetic data sets.

Acknowledgement. We thank Jay Banerjee for his support. We thank Rick Hetherington and Brian Whitney for providing access to and guiding us on the use of the Oracle Sun M8000 server machine. We thank Christopher Hecht and Kathleen Li for their assistance in using the Exadata platform.

Reference

1. OWL 2 Web Ontology Language Direct Semantics. http://www.w3.org/TR/owl2-direct-semantics/
2. Oracle Database Semantic Technologies. http://www.oracle.com/technetwork/database/options/semantic-tech/index.html

[7] http://pellet.owldl.com/owlgres
[8] http://www.ontotext.com/owlim/
[9] http://kt.abdn.ac.uk/wiki/Projects/Quill

3. OWL 2 Web Ontology Language Profiles. http://www.w3.org/TR/owl2-profiles/
4. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: "Implementing and Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle" IEEE 24th Intl. Conf. On Data Engineering (ICDE) 2008
5. Oracle Database Semantic Technologies Developer's Guide 11g Release 2 (11.2) http://docs.oracle.com/cd/E11882_01/appdev.112/e11828/toc.htm
6. Introducing PelletDb: Expressive, Scalable Semantic Reasoning for the Enterprise http://clarkparsia.com/files/pdf/pelletdb-whitepaper.pdf
7. Kolovski, V., Wu, Z., Eadon, G.: Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System. International Semantic Web Conference (1) 2010: 436-452
8. J. Broekstra, F. van Harmelen, and A. Kampman, "Seasme: A Generic Architecture for Storing and Querying RDF and RDF Schema". International Semantic Web Conference (ISWC) 2002.
9. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, "RStar: An RDF Storage and Querying System for Enterprise Resource Management". CIKM 2004.
10. Calvanese, G. deD., Giacomo, D.G., Lembo, M.D., Lenzerini, R.M., Rosati ", R.: Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family" In J.. Journal of Automated Reasoning 39(3):) (October 2007) 385---429, 2007.
11. Rosati, R., Almatelli, A.: Improving Query Answering over DL-Lite Ontologies. In Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning. KR, AAAI Press (2010).
12. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/ Last accessed 18-April-2012.
13. SPARQL 1.1 Entailment Regimes. W3C Working Draft 05 January 2012. http://www.w3.org/TR/sparql11-entailment/ Last accessed 18-April-2012.
14. SPARQL 1.1 Query Language. W3C Working Draft 05 January 2012. http://www.w3.org/TR/sparql11-query/ Last accessed 18-April-2012.
15. Gottlob, G., Schwentick, T.: Rewriting Ontological Queries into Small Nonrecursive Datalog Programs. In Proceedings of the 24th International Workshop on Description Logics (DL 2011), Barcelona, Spain, July 13-16, 2011.
16. Pe'rez-Urbina, H., Motik, B., Horrocks, I.: Tractable Query Answering and Rewriting under Description Logic Constraints. Journal of Applied Logic 8(2) (2010) 186—209.
17. Pan, J.Z., Thomas, E.: Approximating OWL-DL Ontologies. In: Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2. AAAI'07, AAAI Press (2007) 1434—1439.
18. Stocker, M., Smith, M.: Owlgres: A Scalable OWL Reasoner. In Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, Karlsruhe, Germany, October 26-27, 2008.
19. Bishop, B., Bojanov, S.: Implementing OWL 2 RL and OWL 2 QL. In Proceedings of the 8th International Workshop on OWL: Experiences and Directions (OWLED 2011), San Francisco, California, USA, June 5-6, 2011.
20. Narayanan, S., Catalyurek, U., Kurc, T., Saltz, J.: Parallel Materialization of Large ABoxes. In: Proceedings of the 2009 ACM symposium on Applied Computing. SAC'09, New York, NY, USA, ACM (2009) 1257—1261.
21. Urbani, J., Kotoulas, S., Massen, J., van Harmelen, F., Bal, H.: Webpie: A web-scale parallel inference engine using mapreduce. Web Semantics: Science, Services and Agents on the World Wide Web 10 (2012).
22. Hogan, A., Pan, J., Polleres, A., Decker, S.: SAOR: Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples. In: 9th International Semantic Web Conference (ISWC). (November 2010).

Mini-ME: the Mini Matchmaking Engine

M. Ruta, F. Scioscia, E. Di Sciascio, F. Gramegna, and G. Loseto

Politecnico di Bari, via E. Orabona 4, I-70125 Bari, Italy
E-mail: m.ruta@poliba.it, f.scioscia@poliba.it, disciascio@poliba.it,
gramegna@deemail.poliba.it, loseto@deemail.poliba.it

Abstract. The Semantic Web of Things (SWoT) is a novel paradigm, blending the Semantic Web and the Internet of Things visions. Due to architectural and performance issues, it is currently impractical to use available reasoners for processing semantic-based information and perform resource discovery in pervasive computing scenarios. This paper presents a prototypical mobile reasoner for the SWoT, supporting Semantic Web technologies and implementing both standard (subsumption, satisfiability, classification) and non-standard (abduction, contraction) inference tasks for moderately expressive knowledge bases. Architectural and functional features are described and an experimental performance evaluation is provided both on a PC testbed (w.r.t. other popular Semantic Web reasoners) and on a smartphone.

1 Introduction

The Semantic Web of Things (SWoT) is an emerging paradigm in Information and Communication Technology, joining the Semantic Web and the Internet of Things. The Semantic Web initiative [5] envisions software agents to share, reuse and combine data available in the World Wide Web, by means of machine-understandable annotation languages such as RDF[1] and OWL[2], grounded on Description Logics (DLs) formalisms. The Internet of Things vision [11] promotes pervasive computing on a global scale, aiming to give intelligence to ordinary objects and physical locations by means of a large number of heterogeneous micro-devices, each conveying a small amount of information. Consequently, the goal of the SWoT is to embed semantically rich and easily accessible metadata into the physical world, by enabling storage and retrieval of annotations from tiny smart objects. Such a vision requires an increased autonomy and efficiency of knowledge-based systems for what concerns information memorization, management, dissemination and discovery. Particularly, reasoning and query answering aimed to resource discovery is critical in mobile computing platforms (*e.g.*, smartphones, tablets) which –albeit increasingly effective and powerful– are still affected by hardware/software limitations. They have to be taken into account

[1] Resource Description Framework, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-primer/
[2] OWL 2 Web Ontology Language, W3C Recommendation 27 October 2009, http://www.w3.org/TR/owl-overview/

when designing systems and applications: particularly, to use more expressive languages increases the computational complexity of inferences and significant architectural and performance issues affect porting current OWL-based reasoners, designed for the Semantic Web, to handheld devices. This paper presents *Mini-ME (the Mini Matchmaking Engine)*, a prototypical mobile reasoner for moderately expressive DLs, created to support *semantic-based matchmaking* [6], [16]. It complies with standard Semantic Web technologies through the OWL API [9] and implements both standard reasoning tasks for Knowledge Base (KB) management (subsumption, classification, satisfiability) and non-standard inference services for semantic-based resource discovery and ranking (abduction and contraction [6]). Mini-ME is developed in Java, adopting Android as target computing platform.

The remaining of the paper is organized as follows. Section 2 reports on related work, providing perspective and motivation for the proposal. Mini-ME is presented in Section 3, where details are given about reasoning algorithms, software architecture, data structures and supported logic languages. Section 4 relates to performance evaluation on the venue reference datasets[3] and a comparison with other popular Semantic Web reasoners is proposed. Finally conclusion and future work in Section 5 close the paper.

2 Related Work

When processing semantic-based information to infer novel and implicit knowledge, careful optimization is needed to achieve acceptable reasoning performance for adequately expressive languages [3, 10]. This is specifically true in case of logic-based matchmaking for mobile computing, which is characterized by severe resource limitations (not only affecting processing, memory and storage, but also energy consumption). Most mobile engines currently provide only rule processing for entailments materialization in a KB [14, 27, 12, 18], so basically, available features are not suitable to support applications requiring non-standard inference tasks and extensive reasoning over ontologies [18]. More expressive languages could be used by adapting tableaux algorithms –usually featuring reasoners running on PCs– to mobile computing platforms, but an efficient implementation of reasoning services is still an open problem. Several techniques [10] allow to increase expressiveness or decrease running time at the expense of main memory usage, which is the most constrained resource in mobile systems. *Pocket KRHyper* [24] was the first reasoning engine specifically designed for mobile devices. It supported the $\mathcal{ALCHIR}+$ DL and was built as a Java ME (Micro Edition) library. Pocket KRHyper was exploited in a DL-based matchmaking framework between user profiles and descriptions of mobile resources/services [13]. However, its limitation in size and complexity of managed logic expressions was very heavy due to frequent "out of memory" errors. To overcome those constraints, tableaux optimizations to reduce memory consumption were introduced in [26] and implemented in *mTableau*, a modified version of Java SE *Pellet* reasoner

[3] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/

[25]. Comparative performance tests were performed on a PC, showing faster turnaround times than both unmodified Pellet and *Racer* [8] reasoner. Nevertheless, the Java SE technology is not expressly tailored to the current generation of handheld devices. In fact, other relevant reasoners, such as *FaCT++* [28] and *HermiT* [23], cannot run on common mobile platforms. Porting would require a significant re-write or re-design effort, since they rely on Java class libraries incompatible with mosto widespread mobile OS (*e.g.*, Android). Moreover, the above systems only support standard inference services such as satisfiability and subsumption, which provide only binary "yes/no" answers. Consequently, they can only distinguish among *full* (*subsume*), *potential* (*intersection-satisfiable*) and *partial* (*disjoint*) match types (adopting the terminology in [6] and [16], respectively). Non-standard inferences, as Concept Abduction and Concept Contraction, are needed to enable a more fine-grained semantic ranking as well as explanations of outcomes [6]. In latest years, a different approach to implement reasoning tools arose. It was based on simplifying both the underlying logic languages and admitted KB axioms, so that structural algorithms could be adopted, but maintaining expressiveness enough for broad application areas. In [1], the basic $\mathcal{EL}$ DL was extended to $\mathcal{EL}^{++}$, a language deemed suitable for various applications, characterized by very large ontologies with moderate expressiveness. A structural classification algorithm was also devised, which allowed high-performance $\mathcal{EL}^{++}$ ontology classifiers such as CEL [4] and Snorocket [15]. OWL 2 profiles definition complies with this perspective, focusing on language subsets of practical interest for important application areas rather than on fragments with significant theoretical properties. In a parallel effort motivated by similar principles, in [22] an early approach was proposed to adapt non-standard logic-based inferences to pervasive computing contexts. By limiting expressiveness to $\mathcal{AL}$ language, acyclic, structural algorithms were adopted reducing standard (*e.g.*, subsumption) and non-standard (*e.g.*, abduction and contraction) inference tasks to set-based operations [7]. KB management and reasoning were then executed through a data storage layer, based on a mobile RDBMS (Relational DBMS). Such an approach was further investigated in [20] and [19], by increasing the expressiveness to $\mathcal{ALN}$ DL and allowing larger ontologies and more complex descriptions, through the adoption of both mobile OODBMS (Object-Oriented DBMS) and performance-optimized data structures. Finally, in [21] expressiveness was extended to $\mathcal{ALN}(\mathrm{D})$ DL with fuzzy operators. The above tools were designed to run on Java ME PDAs and were adopted in several case studies employing semantic matchmaking over moderately expressive KBs. The reasoning engine presented here recalls lessons learned in those previous efforts, and aims to provide a standards-compliant implementation of most common inferences (both standard and non-standard) for widespread mobile platforms.

3 System Description

The architecture of the proposed reasoning engine is sketched as UML diagram in Figure 1. Components are outlined hereafter:

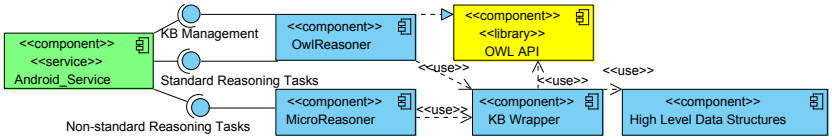


Fig. 1: Component UML diagram

- **Android Service**: implements a service (*i.e.*, a background daemon) any Android application can invoke to use the engine;
- **OwlReasoner**: OWL API [9] implementation exposing fundamental KB operations (load, parse) and standard reasoning tasks (subsumption, classification, satisfiability); it is endorsed by the OWL API open source library;
- **MicroReasoner**: interface for non-standard reasoning tasks (concept abduction, contraction);
- **KB Wrapper**: implements KB management functions (creation of internal data structures, normalization, unfolding) and basic reasoning tasks on ontologies (classification and coherence check);
- **High Level Data Structures**: in-memory data structures for concept manipulation and reasoning; they refer to reasoning tasks on concept expressions (concept satisfiability, subsumption, abduction, contraction).

Mini-ME was developed using Android SDK Tools[4], Revision 12, corresponding to Android Platform version 2.1 (API level 7), therefore it is compatible with all devices running Android 2.1 or later. Mini-ME can be used either through the *Android Service* by Android applications, or as a library by calling public methods of the *OwlReasoner* and *MicroReasoner* components directly. In the latter form, it runs unmodified on Java Standard Edition runtime environment, version 6 or later. The system supports OWL 2 ontology language, in all syntaxes accepted by the OWL API parser. Supported logic constructors are detailed in Section 3.1. Implementation details for both standard and non-standard reasoning services are given in Section 3.2. Data structures for internal representation and manipulation of concept expressions are outlined in Section 3.3.

3.1 Supported Language

In DL-based reasoners, an ontology $\mathcal{T}$ (a.k.a. Terminological Box or TBox) is composed by a set of axioms in the form: $A \sqsubseteq D$ or $A \equiv D$ where A and D are concept expressions. Particularly, a *simple-TBox* is an acyclic TBox such that: (i) A is always an atomic concept; (ii) if A appears in the left hand side (lhs) of a concept equivalence axiom, then it cannot appear also in the lhs of any concept inclusion axiom. Mini-ME supports the $\mathcal{ALN}$ (Attributive Language with unqualified Number restrictions) DL, which has polynomial computational complexity for standard and non-standard inferences in simple-TBoxes, whose depth of concept taxonomy is bounded by the logarithm of the number of axioms in it (see [7] for further explanation). Actually, such DL fragment has been

[4] http://developer.android.com/sdk/tools-notes.html

Table 1: Syntax and semantics of $\mathcal{ALN}$ constructs and simple-TBoxes

Name	Syntax	Semantics
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Atomic negation	$\neg A$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
Universal quantification	$\forall R.C$	$\{d_1 \mid \forall d_2 : (d_1, d_2) \in R^{\mathcal{I}} \rightarrow d_2 \in C^{\mathcal{I}}\}$
Number restriction	$\geq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \geq n\}$
	$\leq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \leq n\}$
Inclusion	$A \sqsubseteq D$	$A^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Equivalence	$A \equiv D$	$A^{\mathcal{I}} = D^{\mathcal{I}}$

selected for the first release of Mini-ME as it grants low complexity and memory efficiency of non-standard inference algorithms for semantic matchmaking. $\mathcal{ALN}$ DL constructs are summarized in Table 1.

3.2 Reasoning Services

Mini-ME exploits structural algorithms for standard and non-standard reasoning and then, when a knowledge base is loaded, it has to be preprocessed performing *unfolding* and *Conjunctive Normal Form* (CNF) *normalization*. Particularly, given a TBox $\mathcal{T}$ and a concept C, the **unfolding** procedure recursively expands references to axioms in $\mathcal{T}$ within the concept expression itself. In this way, $\mathcal{T}$ is not needed any more when executing subsequent inferences. **Normalization** transforms the unfolded concept expression in CNF by applying a set of predefined substitutions. Any concept expression C can be reduced in CNF as: $C \equiv C_{CN} \sqcap C_{LT} \sqcap C_{GT} \sqcap C_{\forall}$, where C_{CN} is the conjunction of (possibly negated) atomic concept names, C_{LT} (respectively C_{GT}) is the conjunction of $\leq$ (resp. $\geq$) number restrictions (no more than one per role), and $C_{\forall}$ is the conjunction of universal quantifiers (no more than one per role; fillers are recursively in CNF). Normalization preserves semantic equivalence w.r.t. models induced by the TBox; furthermore, CNF is unique (up to commutativity of conjunction operator) [7]. The normal form of an unsatisfiable concept is simply $\bot$. The following standard reasoning services on (unfolded and normalized) concept expressions are currently supported:
- **Concept Satisfiability** (a.k.a. consistency). Due to CNF properties, satisfiability check is trivially performed during normalization.
- **Subsumption test**. The classic structural subsumption algorithm is exploited, reducing the procedure to a set containment test [2].

In Mini-ME, two non-standard inference services were also implemented, allowing to (i) provide explanation of outcomes beyond the trivial "yes/no" answer of satisfiability and subsumption tests and (ii) enable a logic-based relevance ranking of a set of available resources w.r.t. a specific query [19]:
- **Concept Contraction**: given a request D and a supplied resource S, if they are not compatible with each other, Contraction determines which part of D is conflicting with S. If one retracts conflicting requirements in D, G (for *Give up*), a concept K (for *Keep*) is obtained, representing a contracted version of

the original request, such that $K \sqcap S$ is satisfiable w.r.t. $\mathcal{T}$. The solution G to Contraction represents "why" $D \sqcap S$ are not compatible.
- **Concept Abduction**: whenever D and S are compatible, but S does not imply D, Abduction allows to determine what should be hypothesized in S in order to completely satisfy D. The solution H (for *Hypothesis*) to Abduction represents "why" the subsumption relation $\mathcal{T} \models S \sqsubseteq D$ does not hold. H can be interpreted as *what is requested in D and not specified in S*.

In order to use Mini-ME in more general knowledge-based applications, the following reasoning services over ontologies were also implemented:
- **Ontology Satisfiability**: since Mini-ME does not currently process the ABox, it performs an ontology *coherence* check rather than satisfiability check (difference is discussed *e.g.*, in [17]). During ontology parsing, the *KB Wrapper* module creates a hash table to store all concepts in the TBox $\mathcal{T}$. Since CNF normalization allows to identify unsatisfiable concepts, it is sufficient to normalize every table item to locate unsatisfiability in the ontology.
- **Classification**: ontology classification computes the overall concept taxonomy induced by the subsumption relation, from $\top$ to $\bot$ concept. In order to reduce the subsumption tests, the following optimizations introduced in [3] were implemented: *enhanced traversal top search*, *enhanced traversal bottom search*, exploitation of *told subsumers*. The reader is referred to [3] for further details.

3.3 Data Structures

The UML diagram in Figure 2 depicts classes in the *High Level Data Structures* package (mentioned before) and their relationships. Standard Java Collection Framework classes are used as low-level data structures:
- **Item**: each concept in the ontology is an instance of this class. Attributes are the name and the corresponding concept expression. When parsing an ontology, the *KB Wrapper* component builds a Java *HashMap* object containing all concepts in the TBox as *String-Item* pairs. Each concept is unfolded, normalized and stored in the HashMap with its name as key and *Item* instance as value.
- **SemanticDescription**: models a concept expression in CNF as aggregation of $C_{CN}, C_{GT}, C_{LT}, C_{\forall}$ components, each one stored in a different Java *ArrayList*. Methods implement inference services: `abduce` returns the hypothesis H expression; `contract` returns a two-element array with G and K expressions; `checkCompatibility` checks consistency of the conjunction between the object *SemanticDescription* and the one acting as input parameter; similarly, `isSubsumed` performs subsumption test with the input *SemanticDescription*.
- **Concept**: models an atomic concept A_i in C_{CN}; `name` contains the concept name, while `denied`, if set to *true*, allows to express $\neg A_i$.
- **GreaterThanRole** (respectively **LessThanRole**): models number restrictions in C_{GT} and C_{LT}. Role name and cardinality are stored in the homonym variables.
- **UniversalRole**: a universal restriction $\forall R.D$ belonging to $C_{\forall}$; R is stored in `name`, while D is a *SemanticDescription* instance.

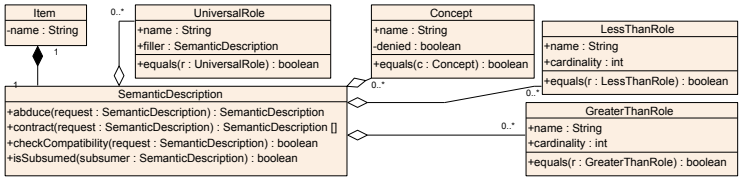


Fig. 2: Class diagram of *High Level Data Structures* package

In the last classes, the `equals` method, inherited from *java.lang.Object*, has been overridden in order to properly implement logic-based comparison.

4 Experimental Evaluation

Performance evaluation was carried out for classification, class satisfiability and ontology satisfiability, including both a comparison with other popular Semantic Web reasoners on a PC testbed[5] and results obtained on an Android smartphone[6]. The reference dataset is composed of 214 OWL ontologies with different complexity, expressiveness and syntax. Full results are reported on the project home page[7], while main highlights are summarized hereafter. Mini-ME was compared on PC with FaCT++[8], HermiT[9] and Pellet[10]. All reasoners were used via the OWL API [9]. For each reasoning task, two tests were performed: (i) correctness of results and turnaround time; (ii) memory usage peak. For turnaround time, each test was repeated four times and the average of the last three runs was taken. For memory tests, the final result was the average of three runs. Performance evaluation for non-standard inferences is not provided here.

4.1 PC Tests

Classification. The input of this task was the overall ontology dataset. For each test, one of the following possible outcomes was recorded: (i) *Correct*, the computed taxonomy corresponds with the reference classification –if it is included into the dataset– or results of all the reasoners are the same; in this case the total time taken to load and classify the ontology is also reported; (ii) *Parsing Error*, the ontology cannot be parsed by the OWL API due to syntax errors; (iii) *Failure*, the classification task fails because the ontology contains unsupported logic language constructors; (iv) *Out of Memory*, the reasoner generates an exception

[5] Intel Core i7 CPU 860 at 2.80 GHz (4 cores/8 threads), 8 GB DDR3-SDRAM (1333 MHz) memory, 1 TB SATA (7200 RPM) hard disk, 64-bit Microsoft Windows 7 Professional and 64-bit Java 7 SE Runtime Environment (build 1.7.0_03-b05).

[6] Samsung i9000 Galaxy S with ARM Cortex A8 CPU at 1 GHz, 512 MB RAM, 8 GB internal storage memory, and Android version 2.3.3.

[7] Mini-ME Home Page, http://sisinflab.poliba.it/swottools/minime/

[8] FaCT++, version 1.5.3 with OWL API 3.2, http://owl.man.ac.uk/factplusplus/

[9] HermiT OWL Reasoner, version 1.3.6, http://hermit-reasoner.com/

[10] Pellet, version 1.3, http://clarkparsia.com/pellet/

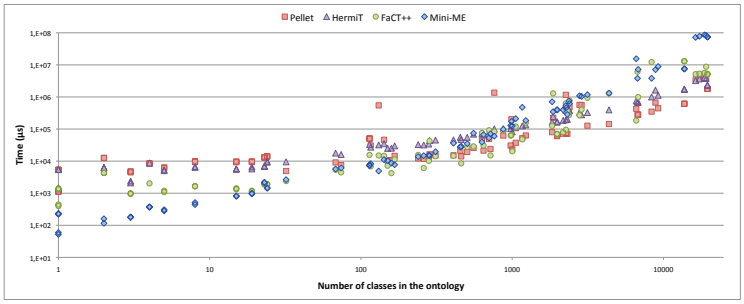


Fig. 3: Classification test on PC

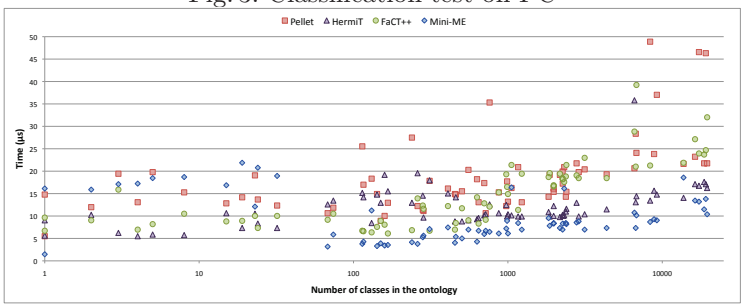


Fig. 4: Class Satisfiability on PC

due to memory constraints; (v) *Timeout*, the task did not complete within the timeout threshold (set to 60 minutes). Mini-ME correctly classified 83 of 214 ontologies; 71 were discarded due to parsing errors, 58 presented unsupported language constructors, the timeout was reached in 2 cases. Pellet classified correctly 124 ontologies, HermiT 127, FaCT++ 118. The lower "score" of Mini-ME is due to the presence of General Concept Inclusions, cyclic TBoxes or unsupported logic constructors, since parsing errors occur in the OWL API library and are therefore common to all reasoners. Figure 3 compares the classification times of each reference reasoner w.r.t. the number of classes in every ontology. Pellet, HermiT and FaCT++ present a similar trend (with FaCT++ slightly faster than the other engines), while Mini-ME is very competitive for small-medium ontologies (up to 1200 classes) but less for large ones. This can be considered as an effect of the Mini-ME design, which is optimized to manage elementary TBoxes.

Class satisfiability. The reference test dataset consists of 107 ontologies and, for each of them, one or more classes to check. However, we tested only the 69 ontologies that Mini-ME correctly classified in the previous proof. Figure 4 shows that performances are basically similar, with times differing only for few microseconds and no reasoner consistently faster or slower. Moreover, the chart suggests no correlation between the time and the number of classes in the ontology.

Ontology satisfiability. Figure 5 is similar to Figure 3, because this test implies loading, classifying and checking consistency of all concepts in the ontology; the

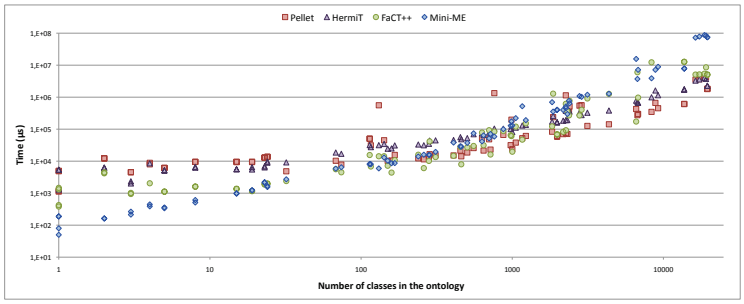
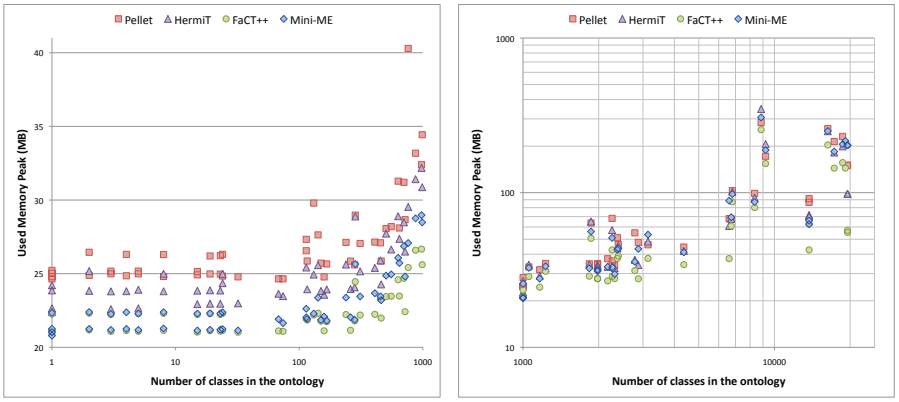


Fig. 5: Ontology Satisfiability on PC



(a) Small ontologies (b) Large ontologies

Fig. 6: Memory usage test on PC

first two steps require the larger part of the time. Results of all reasoners are the same, except for ontologies with IDs 199, 200, 202, 203. In contrast to Pellet, HermiT and FaCT++, Mini-ME checks ontology coherence regardless of the ABox. The above ontologies include an unsatisfiable class (`GO_0075043`) with no instances, therefore the ontology is reported as incoherent by Mini-ME but as satisfiable by the other reasoners.

Memory Usage. Figure 6 reports on memory usage peak during classification, which was verified as the most memory-intensive task. For small ontologies, used memory is roughly similar for all reasoners; Mini-ME provides good results, with lower memory usage than Pellet and HermiT and on par with FaCT++. Also for large ontologies, Mini-ME results are comparable with the other reasoners, although FaCT++ has slightly better overall performance.

4.2 Mobile Tests

Results for mobile tests have been referred to the above outcomes for PC tests in order to put in evidence Mini-ME exhibits similar trends (so offering predictable memory and time consumption behaviors). Anyway, figures clearly evidence the

performance gap, but they highlight the reasoner acceptably works also on mobile platforms. When out-of-memory errors did not occur, results computed by Mini-ME on the Android smartphone were in all cases the same as on the PC. 73 ontologies over 214 were correctly classified on the mobile device, 53 were discarded due to parsing errors, 56 had unsupported language constructors, 30 generated out-of-memory exceptions and 2 reached the timeout. Figure 7 shows the classification turnaround time –only for the correct outcomes– compared with the PC test results. Times are roughly an order of magnitude higher on the Android device. Absolute values for ontologies with 1000 classes or less are under 1 second, so they can be deemed as acceptable in mobile contexts. Furthermore, it can be noticed that the turnaround time increases linearly w.r.t. number of classes both on PC and on smartphone, thus confirming that Mini-ME has predictable behavior regardless of the reference platform. Similar considerations apply to class and ontology satisfiability tests (which were run for the 60 ontologies that were correctly classified): the turnaround time comparisons are reported in Figure 8 and Figure 9. Figure 10 reports on the memory allocation peak for each ontology during the classification task. Under 1000 classes, the required memory is roughly fixed in both cases. Instead, for bigger ontologies the used memory increases according to the total number of classes. Moreover, in every test memory usage on Android is significantly lower than on PC. This is due to the harder memory constraints on smartphones, imposing to have as much free memory as possible at any time. Consequently, Android Dalvik virtual machine performs more frequent and aggressive garbage collection w.r.t. Java SE virtual machine. This reduces memory usage, but on the other hand can be responsible for a significant portion of the PC-smartphone turnaround time gap that was found.

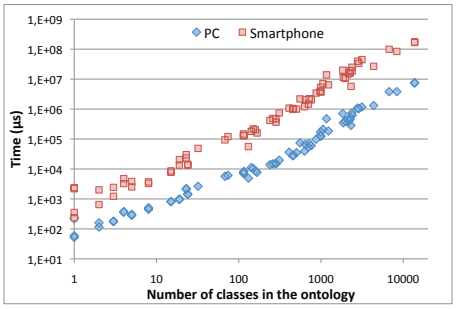


Fig. 7: Classification, PC vs mobile

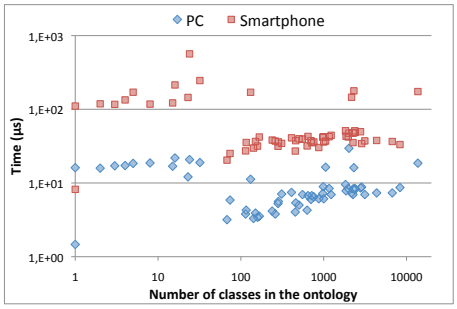


Fig. 8: Class Satisfiability, PC vs mobile

5 Conclusion and Future Work

The paper presented a prototypical reasoner devised for mobile computing. It supports Semantic Web technologies through the OWL API and implements

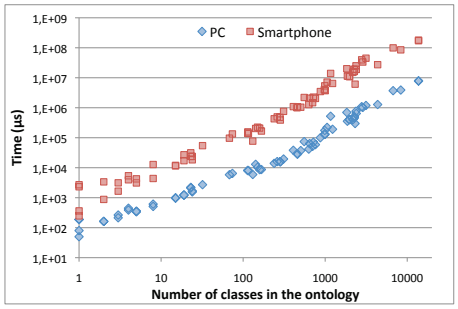


Fig. 9: Ont. Satisfiability, PC vs mobile

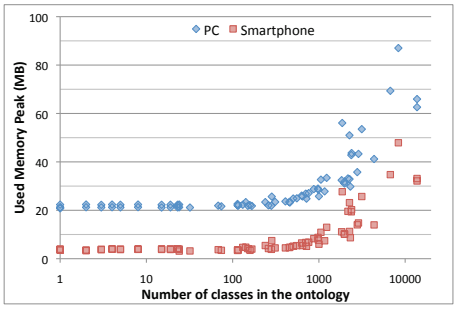


Fig. 10: Memory usage, PC vs mobile

both standard and non-standard reasoning tasks. Developed in Java, it targets the Android platform but also runs on Java SE. Early experiments were made both on PCs and smartphones and evidenced correctness of implementation and competitiveness with state-of-the-art reasoners in standard inferences, and acceptable performance on target mobile devices. Besides further performance optimization leveraging Android Dalvik peculiarities, future work includes: support for ABox management and OWLlink protocol[11]; implementation of further reasoning tasks; $\mathcal{EL}^{++}$extension of abduction and contraction algorithms.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: Int. Joint Conf. on Artificial Intelligence. vol. 19, p. 364. Lawrence Erlbaum Associates LTD (2005)
2. Baader, F., Calvanese, D., Mc Guinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press (2002)
3. Baader, F., Hollunder, B., Nebel, B., Profitlich, H., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. Applied Intelligence 4(2), 109–132 (1994)
4. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL – a polynomial-time reasoner for life science ontologies. Automated Reasoning pp. 287–291 (2006)
5. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic Web. Scientific American 284(5), 28–37 (2001)
6. Colucci, S., Di Noia, T., Pinto, A., Ragone, A., Ruta, M., Tinelli, E.: A Non-Monotonic Approach to Semantic Matchmaking and Request Refinement in E-Marketplaces. Int. Jour. of Electronic Commerce 12(2), 127–154 (2007)
7. Di Noia, T., Di Sciascio, E., Donini, F.: Semantic matchmaking as non-monotonic reasoning: A description logic approach. Jour. of Artificial Intelligence Research (JAIR) 29, 269–307 (2007)
8. Haarslev, V., Müller, R.: Racer system description. Automated Reasoning pp. 701–705 (2001)
9. Horridge, M., Bechhofer, S.: The OWL API: a Java API for working with OWL 2 ontologies. Proc. of OWL Experiences and Directions 2009 (2009)

[11] OWLlink Structural Specification, W3C Member Submission, http://www.w3.org/Submission/owllink-structural-specification/

10. Horrocks, I., Patel-Schneider, P.: Optimizing description logic subsumption. Jour. of Logic and Computation 9(3), 267–293 (1999)
11. ITU: Internet Reports 2005: The Internet of Things (November 2005)
12. Kim, T., Park, I., Hyun, S., Lee, D.: MiRE4OWL: Mobile Rule Engine for OWL. In: Computer Software and Applications Conf. Workshops (COMPSACW), 2010 IEEE 34th Annual. pp. 317–322. IEEE (2010)
13. Kleemann, T., Sinner, A.: User Profiles and Matchmaking on Mobile Phones. In: Bartenstein, O. (ed.) Proc. of 16th Int. Conf. on Applications of Declarative Programming and Knowledge Management INAP2005, Fukuoka (2005)
14. Koch, F.: 3APL-M platform for deliberative agents in mobile devices. In: Proc. of the fourth international joint conference on Autonomous agents and multiagent systems. p. 154. ACM (2005)
15. Lawley, M., Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In: Proc. 6th Australasian Ontology Workshop (IAOA10). Conf.s in Research and Practice in Information Technology. vol. 122, pp. 45–49 (2010)
16. Li, L., Horrocks, I.: A software framework for matchmaking based on semantic web technology. Int. Jour. of Electronic Commerce 8(4), 39–60 (2004)
17. Moguillansky, M., Wassermann, R., Falappa, M.: An argumentation machinery to reason over inconsistent ontologies. Advances in Artificial Intelligence–IBERAMIA 2010 pp. 100–109 (2010)
18. Motik, B., Horrocks, I., Kim, S.: Delta-Reasoner: a Semantic Web Reasoner for an Intelligent Mobile Platform. In: Twentyfirst Int. World Wide Web Conf. (WWW 2012). ACM (2012), to appear
19. Ruta, M., Di Sciascio, E., Scioscia, F.: Concept abduction and contraction in semantic-based P2P environments. Web Intelligence and Agent Systems 9(3), 179–207 (2011)
20. Ruta, M., Scioscia, F., Di Noia, T., Di Sciascio, E.: Reasoning in Pervasive Environments: an Implementation of Concept Abduction with Mobile OODBMS. In: 2009 IEEE/WIC/ACM Int. Conf. on Web Intelligence. pp. 145–148. IEEE (2009)
21. Ruta, M., Scioscia, F., Di Sciascio, E.: Mobile Semantic-based Matchmaking: a fuzzy DL approach. In: The Semantic Web: Research and Applications. Proceedings of 7th Extended Semantic Web Conference (ESWC 2010). Lecture Notes in Computer Science, vol. 6088, pp. 16–30. Springer (2010)
22. Ruta, M., Di Noia, T., Di Sciascio, E., Piscitelli, G., Scioscia, F.: A semantic-based mobile registry for dynamic RFID-based logistics support. In: ICEC '08: Proc. of the 10th Int. Conf. on Electronic commerce. pp. 1–9. ACM, New York, USA (2008)
23. Shearer, R., Motik, B., Horrocks, I.: Hermit: A highly-efficient owl reasoner. In: Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008). pp. 26–27 (2008)
24. Sinner, A., Kleemann, T.: KRHyper - In Your Pocket. In: Proc. of 20th Int. Conf. on Automated Deduction (CADE-20). pp. 452–457. Tallinn, Estonia (July 2005)
25. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: science, services and agents on the World Wide Web 5(2), 51–53 (2007)
26. Steller, L., Krishnaswamy, S.: Pervasive Service Discovery: mTableaux Mobile Reasoning. In: Int. Conf. on Semantic Systems (I-Semantics). Graz, Austria (2008)
27. Tai, W., Keeney, J., O'Sullivan, D.: COROR: a composable rule-entailment owl reasoner for resource-constrained devices. Rule-Based Reasoning, Programming, and Applications pp. 212–226 (2011)
28. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Automated Reasoning pp. 292–297 (2006)

WSReasoner: A Prototype Hybrid Reasoner for $\mathcal{ALCHOI}$ Ontology Classification using a Weakening and Strengthening Approach

Weihong Song[1], Bruce Spencer[1,2], and Weichang Du[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
{song.weihong, bspencer, wdu}@unb.ca,
[2] National Research Council, Canada

Abstract. In the ontology classification task, consequence-based reasoners are typically significantly faster while tableau-based reasoners can process more expressive DL languages. However, both of them have difficulty to classify some available large and complex $\mathcal{ALCHOI}$ ontologies with complete results in acceptable time. We present a prototype hybrid reasoning system WSReasoner, which is built upon and takes advantages of both types of reasoners to provide efficient classification service. In our proposed approach, we approximate the target ontology $\mathcal{O}$ by a weakened version $\mathcal{O}_{wk}$ and a strengthened version $\mathcal{O}_{str}$, both are in a less expressive DL $\mathcal{ALCH}$ and classified by a consequence-based main reasoner. Classification of $\mathcal{O}_{wk}$ produces a subset of subsumptions of ontology $\mathcal{O}$ and the target of the classification of $\mathcal{O}_{str}$ is to produce a superset of subsumptions of $\mathcal{O}$. Additional subsumptions derived from $\mathcal{O}_{str}$ may be unsound, so they are further verified by a tableau-based assistant reasoner. For the $\mathcal{ALCHOI}$ ontologies in our experiment, except for one for which WSReasoner has not obtained the result, (1) the number of subsumptions derived from WSReasoner is no fewer than from the reasoners that could finish the classification; (2) WSReasoner takes less time than tableau-based reasoners when the $\mathcal{ALCHOI}$ ontologies are large and complex.

1 Introduction

Ontology classification — computing the subsumption relationships between classes — is one of the foundational reasoning tasks provided by many reasoners. Tableau-based and consequence-based reasoners are two dominant types of reasoners that provide the ontology classification service. Tableau-based reasoners, such as HermiT [8], Fact++ [13] and Pellet [12], try to build counter-models $A \sqcap \neg B$ for candidate subsumption relations, based on sound and complete calculi such as [4] and [8]. These reasoners are able to classify ontologies in expressive DLs like $\mathcal{SROIQ}(\mathcal{D})$.

Consequence-based reasoners classify the ontology based on specifically designed inference rules for deriving logical consequences of the axioms in the ontology. Initially developed for the family of tractable DLs like $\mathcal{EL}^{++}$ [1], these procedures were later extended to Horn-$\mathcal{SHIQ}$ [5] and $\mathcal{ALCH}$ [11] while preserving optimal computational complexity. Reasoners belonging to this category, such as CB, ConDOR, ELK [6], CEL [1] and TrOWL [9], are usually very fast and use less memory.

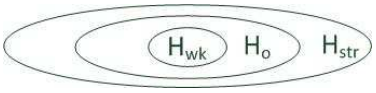


Fig. 1. Subsumption Diagram

We present a hybrid reasoning system that takes the advantages of both types of reasoners for efficient classification on large and complex ontologies in expressive DLs. Here "complex ontologies" refers to the ontologies which contains a considerable amount of cyclic definitions, which usually causes large models constructed by the tableau procedures. In our approach, for the *main* reasoner we choose one that supports a less expressive language, which we call the base language. From the original ontology $\mathcal{O}$, we first remove the axioms that are beyond the base language, and so construct a weakened ontology $\mathcal{O}_{wk}$. In the second stage, we then inject into $\mathcal{O}_{wk}$ additional axioms to simulate the effects of those removed axioms in a model expansion process, constructing the strengthened ontology $\mathcal{O}_{str}$. These injected axioms are expressed in the base language so they may not perfectly represent the original axioms. We call the stages weakening and strengthening, respectively. After applying these changes to the ontology, we still would like the subsumptions in $\mathcal{O}_{str}$ to contain all the subsumptions in $\mathcal{O}$. In Fig. 1, the results of classification, named the class hierarchy, for ontology $\mathcal{O}$, $\mathcal{O}_{wk}$, and $\mathcal{O}_{str}$ are denoted by H_O, H_{wk} and H_{str} respectively. The subsumptions pairs in H_{str} may be unsound with respect to $\mathcal{O}$. If this occurs, we will need again to verify these suspected pairs by reasoning in the language of the given ontology to remove any unsound subsumptions and other maintenance tasks. We name the reasoner that accepts the full language of $\mathcal{O}$, the *assistant* reasoner; it is potentially slower than the main reasoner.

Our main contributions are as follows:

Hybrid Reasoning using Weakening and Strengthening Approach: We propose a new hybrid reasoning approach by combining the tableau-based and consequence-based reasoning procedures for efficient classification on large and complex ontologies. Concretely, the hybrid reasoning is based on a weakening and strengthening approach and applied to classifying $\mathcal{ALCHOI}$ ontologies, for which we choose $\mathcal{ALCH}$ as the base language to take advantage of the recently developed consequence-based reasoning technique which is able to classify non-Horn ontologies [11].

Implementation and Evaluation: Our system is able to classify ontologies in DL $\mathcal{ALCHOI}$, which is not fully supported by any current consequence-based reasoner. We evaluate our procedure with nine available, practical $\mathcal{ALCHOI}$ ontologies. Except for one ontology we have not gotten the classification result, we are able to achieve soundness with no fewer subsumptions and a better performance than the tableau-based reasoners on large ontologies.

2 Preliminaries and Related Work

The syntax of $\mathcal{ALCHOI}$ uses mutually disjoint sets of *atomic concepts* N_C, *atomic roles* N_R and *individuals* N_I. The set of *roles* is $N_R \cup \{R^- \mid R \in N_R\}$. The set of *concepts* contains A, $\top$, $\bot$, $\neg C$, $C \sqcap D$, $C \sqcup D$, $\exists R.C$, $\forall R.C$, $\{a\}$, for $\top$ the top concept, $\bot$ the

bottom concept, A an atomic concept, C and D concepts, R a role, a an individual. We define $N_C^{\top} = N_C \cup \{\top\}$ and $N_C^{\top,\bot} = N_C^{\top} \cup \{\bot\}$. An ontology $\mathcal{O}$ consists of a set of *general concept inclusions* $C \sqsubseteq D$ and *role inclusions* $R \sqsubseteq S$. A concept equivalence $C = D$ is a shortcut for $C \sqsubseteq D$ and $D \sqsubseteq C$.

An interpretation $\mathcal{I}$ of $\mathcal{O}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty set, $\cdot^{\mathcal{I}}$ maps each $A \in N_C$ to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each $R \in N_R$ to a relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and each $a \in N_I$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation of concepts are defined in [2]. An interpretation $\mathcal{I}$ satisfies axioms $C \sqsubseteq D$ and $R \sqsubseteq S$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$, respectively. $\mathcal{I}$ is a model of $\mathcal{O}$ if $\mathcal{I}$ satisfies every axiom in $\mathcal{O}$. If every model of $\mathcal{O}$ satisfies an axiom α, we say $\mathcal{O}$ entails α and write $\mathcal{O} \models \alpha$.

An ontology classification task is to compute the class hierarchy $H_{\mathcal{O}}$ containing all the pairs $\langle A, B\rangle$ such that $A, B \in N_C^{\top,\bot}$ and $\mathcal{O} \models A \sqsubseteq B$. We define the role hierarchy H_{op} as the pairs $\langle R, S\rangle$ such that $R, S \in N_R \cup \{R^- \mid R \in N_R\}$ and $\mathcal{O} \models R \sqsubseteq S$.

Our work can be placed in the Theory Approximation setting [10], where a theory Σ is approximated by a lower bound Σ_{lb}, whose models are a subset of the models of Σ, and an upper bound Σ_{ub} whose models are a superset. Our weakening step creates $\mathcal{O}_{wk}$ which is an upper bound Σ_{ub}. Instead of creating a lower bound Σ_{lb}, the target of our strengthening step is to generate an $\mathcal{O}_{str}$ of which some "important" models can be transformed to models of $\mathcal{O}$ so that completeness can be achieved. The details will be explained in Section 4.2. Subsumption results from $\mathcal{O}_{wk}$ are guaranteed to be sound, exactly as queries asked of Σ_{ub} that return "yes" can be taken also as "yes" from Σ. New candidate subsumption results from $\mathcal{O}_{str}$ need to be checked, analogously as queries Σ_{lb} that return "yes" need to be checked.

TrOWL [9] is a soundness-preserving approximate reasoner offering tractable classification for $\mathcal{SROIQ}$ ontologies by an encoding into $\mathcal{EL}^{++}$ with additional data structures. Instead of merely preserving soundness, our algorithm also aims to achieve completeness, although we have not yet proven it. Another difference lies in that the classification procedure of TrOWL is an extension of [1], while our procedure treats both the main and the assistant reasoners as black boxes without changing them.

3 System Overview

The diagram of our system is shown in Fig. 2. The input is an OWL 2 ontology in any syntax supported by the OWL API.[3] The output is the class hierarchy $H_{\mathcal{O}}$ that can be accessed through the OWL API reasoning interfaces. We explain all the components in the following, among which the ones in white boxes are mainly implemented by us:

- The *preprocessor* rewrites some axioms containing constructors that are not supported by the *main reasoner*.
- The *indexer* normalizes the ontology $\mathcal{O}$ and builds an internal representation of it which is suitable for finding axioms and concept expressions. The index speeds up search for strengthening axioms.
- The *axiom injector* calculates the strengthening axioms that approximate the axioms in $\mathcal{O} \setminus \mathcal{O}_{wk}$. The algorithm will be illustrated in Section 4.

[3] Since our algorithm is designed for DL $\mathcal{ALCHOI}$, the unsupported anonymous concepts are replaced with artificial atomic concepts and the unsupported axioms are ignored.

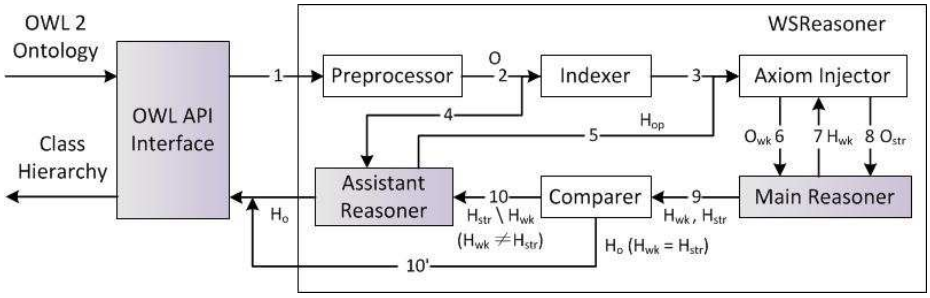


Fig. 2. Key components of WSReasoner

- The *main* and *assistant reasoners* perform the main reasoning tasks. They can be customized by the settings in the configuration instance of each *WSReasoner* object.
- The *comparer* calculates the difference between two concept hierarchies produced by the first and second round of classifications, H_{wk} and H_{str} respectively.

The arrows in the Fig. 2 represent the data flow of the overall reasoning procedure. The numbers on the arrow denote the execution order, and the symbols represent the data. The arrows between the axiom injector and the main reasoner indicates their interactions with each other.

4 The Hybrid Classification Procedure

In this section we give details of the hybrid classification procedure used in WSReasoner. The major phases include preprocessing, normalization and reasoning. Section 4.1 explains preprocessing and normalization. Section 4.2 gives a model-theoretic illustration of the weakening and strengthening approach using an example. And section 4.3 provides the details of the overall procedure and strengthening algorithms.

4.1 Preprocessing and Normalization

In the preprocessing phase, we rewrite the original ontology to make nominals and inverse roles occur only in the axioms of the forms $N_a = \{a\}$ and $R = R'^{-}$, respectively. For nominals, we first rewrite the related OWL 2 DL class expressions and axioms by their equivalent forms containing only singleton nominal concepts, according to Table 1. After that, for each $\{a\}$, we replace all its occurrences by a new concept N_a and add an axiom $N_a = \{a\}$. We call N_a a nominal placeholder for a in the following sections. For inverse roles, we replace each occurrence of R'^{-} in an axiom by a named role R and add an axiom $R = R'^{-}$.

After preprocessing, apart from the axioms of the forms $N_a = \{a\}$ and $R = R'^{-}$, the remaining axioms in the ontology are in DL $\mathcal{ALCH}$. These axioms are normalized using a procedure identical to [11]. The result ontology O contains axioms of the forms $\bigsqcap A_i \sqsubseteq \bigsqcup B_j$, $A \sqsubseteq \exists R.B$, $\exists R.A \sqsubseteq B$, $A \sqsubseteq \forall R.B$, $R \sqsubseteq S$, $N_a = \{a\}$ and $R = R'^{-}$, where A, B are atomic concepts and R, S, R' are atomic roles.

Table 1. Rewriting Nominals in OWL 2

OWL 2 Syntax	Equivalent Forms
Class Expressions	
ObjectHasValue ($R\ a$)	$\exists R.\{a\}$
ObjectOneOf ($a_1 \ldots a_n$)	$\{a_1\} \sqcup \ldots \sqcup \{a_n\}$
Axioms	
ClassAssertion ($C\ a$)	$\{a\} \sqsubseteq C$
SameIndividual ($a_1 \ldots a_n$)	$\{a_1\} = \ldots = \{a_n\}$
DifferentIndividuals ($a_1 \ldots a_n$)	$\{a_i\} \sqcap \{a_j\} = \bot$
	$1 \leq i < j \leq n$
ObjectPropertyAssertion ($R\ a\ b$)	$\{a\} \sqsubseteq \exists R.\{b\}$
NegativeObjectPropertyAssertion ($R\ a\ b$)	$\{a\} \sqcap \exists R.\{b\} \sqsubseteq \bot$

4.2 Model-Theoretic View of the Strengthening Step

Before going into the details of the reasoning procedure, we give a model-theoretic explanation of the motivation of the strengthening step. Given an ontology $\mathcal{O}$, we want to create its strengthened version $\mathcal{O}_{str}$ which satisfies $H_{\mathcal{O}} \subseteq H_{str}$. To achieve it, we try to ensure that for each $\langle A, B\rangle \notin H_{str}$, there is a certain model $\mathcal{I}$' of $\mathcal{O}_{str}$ for $A \sqcap \neg B$ which can be transformed to a model $\mathcal{I}$ of $\mathcal{O}$ and $(A \sqcap \neg B)^{\mathcal{I}} \neq \emptyset$, so that $\langle A, B\rangle \notin H_{\mathcal{O}}$. Such models $\mathcal{I}$' and $\mathcal{I}$ can be constructed using the hypertableau calculus (abbreviated as HT-calculus) [8]. In the following we first describe strengthening for nominals, followed by strengthening for inverse roles.

4.2.1 Strengthening for Nominals

Example 1. Consider a normalized ontology $\mathcal{O}$ containing the following axioms:

$$A \sqsubseteq \exists R.E \ (1) \quad A \sqsubseteq C \ (2) \quad C \sqsubseteq \forall R.D \ (3) \quad E \sqsubseteq N_a \ (4) \quad N_a = \{a\} \ (5)$$
$$A \sqsubseteq \exists S.N_a \ (6) \quad \exists S.D \sqsubseteq F \ (7) \quad F \sqsubseteq B' \sqcup B \ (8) \quad D \sqsubseteq G \ (9)$$

In Fig. 3 – 5 we give models of $\mathcal{O}$, $\mathcal{O}_{wk}$ and $\mathcal{O}_{str}$ for the concept $A \sqcap \neg B$ constructed by the HT-calculus. In the figures, each node x denotes an individual and its tags represent the concepts that it belongs to, which we call *labels* of x. Each edge $\langle x, y\rangle$ denotes a role relation between two individuals x and y, and its *labels* are the roles that it belongs to. We say that a label B of x is *added by an axiom* α in a normalized ontology if $B(x)$ is added into the model by a derivation corresponding to α in the HT-calculus, e.g. $A \sqsubseteq B$ corresponds to $A(x) \rightarrow B(x)$ and $\exists R.A \sqsubseteq B$ corresponds to $R(x, y) \wedge A(x) \rightarrow B(x)$, etc.

Fig. 3 is a model of $\mathcal{O}_{wk}$, which removes the axiom (5) from $\mathcal{O}$. In the model both of the individuals x_2, x_3 have the label N_a. To build a model of $\mathcal{O}$ based on the HT-calculus, x_2 and x_3 need to be merged into one instance to satisfy the axiom (5). After that, labels F and B' of x_1 will be added by the axioms (7) and (8), yielding the model $\mathcal{I}$ in Fig. 4.

Our strengthening step adds additional axioms $N_a \sqsubseteq E$ and $N_a \sqsubseteq D$ to $\mathcal{O}_{wk}$ to simulate the main effect of the merge operation in the HT-calculus, i.e. making all the instances of N_a have the same labels. With these axioms added, labels D, E and G are added to x_3, and labels F and B' of x_1 can be further introduced by the HT-calculus

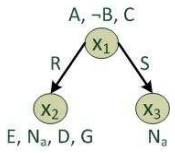


Fig. 3. Model of O_{wk}

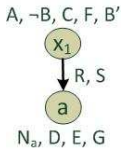


Fig. 4. Model of O

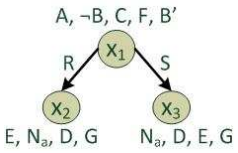


Fig. 5. Model of O_{str}

without the nominal axiom (5). The resulting model $\mathcal{I}$' in Fig. 5 can be transformed to $\mathcal{I}$ by simply merging the instances x_2 and x_3 without extra derivations.

To achieve the above-mentioned effect, we calculate the "important" labels appearing on the instances of N_a in the model of O_{wk}, which we call *major coexisting labels* of N_a. For each instance x, such an important label is the label when x is created by the HT-calculus or a label added by an axiom $A \sqsubseteq \forall R.B$. In other words, these labels are added at initialization time or through a derivation which takes a label on a predecessor of x as a premise, thus they cannot be introduced based on x's own labels. Note that the label G of x_2 in Fig. 3 is not a major coexisting label since it is added by the axiom $D \sqsubseteq G$. For each major coexisting label X of N_a, we choose either $N_a \sqsubseteq X$ or $N_a \sqcap X \sqsubseteq \bot$ as the strengthening axiom, so that X is either added to or prohibited on all the instances of N_a in the model $\mathcal{I}$' of O_{str}. With these axioms added, all the instances of N_a in $\mathcal{I}$' are likely to have identical labels so that $\mathcal{I}$' can be easily transformed to $\mathcal{I}$ to prove $O \not\models A \sqsubseteq B$.

4.2.2 Strengthening for Inverse Roles Regarding inverse roles, the corresponding derivation of an axiom $R = R'^{-}$ in HT-calculus adds $R(x, y)$ if $R'(y, x)$ exists, or vice versa. The new assertion $R(x, y)$ may lead to the following types of further derivations: (1) a label B is added to x by $\exists R.A \sqsubseteq B$; (2) a label B is added to y by axioms $A \sqsubseteq \forall R.B$; (3) labels are added to edges through axioms $R \sqsubseteq S$ and $S = S'^{-}$. To simulate these effects without deriving $R(x, y)$ using $R = R'^{-}$, the following types of axioms are added respectively: (1) $A \sqsubseteq \forall R'.B$; (2) $\exists R'.A \sqsubseteq B$; (3) all the role subsumptions based on the computed role hierarchy H_{op}. Similar axioms need to be added for the assertion $R'(y, x)$. With these axioms added, the model $\mathcal{I}$' of O_{str} can be transformed to a model $\mathcal{I}$ of O by simply satisfying $R = R'^{-}$ without extra derivations. Notice that all the strengthening axioms to handle inverse roles are implied by O, so we have $O \models O_{str}$ and $H_{str} \subseteq H_O$, thus $H_{str} = H_O$ holds and no verifications are needed.

4.3 Classification Procedure

Algorithm 1 gives the overall classification procedure for an $\mathcal{ALCHOI}$ ontology using the weakening and strengthening approach. In the procedure, MR is the main reasoner that provides efficient classification on an $\mathcal{ALCH}$ ontology, while AR is the assistant reasoner, which is slower but capable of classifying the original $\mathcal{ALCHOI}$ ontology O. Function `classify` computes the class hierarchy.

After normalization, the algorithm computes the role hierarchy H_{op}. Line 3 and 4 compute the strengthened ontology O_{istr} for inverse roles, which has the same hierarchy as O. To classify the $\mathcal{ALCHO}$ ontology O_{istr}, we get its weakened and strengthened

versions $\mathcal{O}_{wk}$ and $\mathcal{O}_{str}$ for nominals and classify them with MR, as shown in lines 5 to 8. Computations of $\mathcal{O}_{\mathcal{I}}^{+}$ and $\mathcal{O}_{N}^{+}$ will be explained in Sections 4.3.1 and 4.3.2. Subsumptions in $H_{str} \setminus H_{wk}$ are verified by AR in line 11 to 15. Note that if some $A \sqsubseteq \bot$ is disproved in line 12, $A \sqsubseteq B$ needs to be verified for almost every B in N_C. In this case the workload of verification for AR may exceed that of classifying $\mathcal{O}$, thus we choose to use AR to get $H_{\mathcal{O}}$ directly. Our approach does not add value in this case. Line 14 to 15 verifies each pair in $H_{str} \setminus H_{wk}$ one by one. The verification process can be further improved using a procedure similar to the optimized KP alorithm [3].

Algorithm 1: Classify an $\mathcal{ALCHOI}$ ontology $\mathcal{O}$ using the hybrid approach

Input: An $\mathcal{ALCHOI}$ ontology $\mathcal{O}$
Output: The classification hierarchy $H_{\mathcal{O}}$

1 preprocess and normalize $\mathcal{O}$;
2 H_{op} := AR.classifyObjectProperties($\mathcal{O}$);
3 $\mathcal{O}_{\mathcal{I}}^{+}$:= getStrAxiomsForInverseRoles($\mathcal{O}$, H_{op});
4 $\mathcal{O}_{istr}$:= $\mathcal{O} \cup \mathcal{O}_{\mathcal{I}}^{+}$ with inverse role axioms $R = R'^{-}$ removed;
5 $\mathcal{O}_{wk}$:= $\mathcal{O}_{istr}$ with nominal axioms $N_a = \{a\}$ removed;
6 $\mathcal{O}_{N}^{+}$:= getStrAxiomsForNominals($\mathcal{O}_{istr}$, H_{op});
7 H_{wk} := MR.classify($\mathcal{O}_{wk}$);
8 H_{str} := MR.classify($\mathcal{O}_{wk} \cup \mathcal{O}_{N}^{+}$);
9 remove any $\langle A, B\rangle$ from H_{wk} and H_{str} if $A \notin N_C^{\top,\bot}$ or $B \notin N_C^{\top,\bot}$;
10 $H_{\mathcal{O}}$:= H_{wk};
11 **foreach** $\langle A, \bot\rangle \in H_{str} \setminus H_{wk}$ **do**
12 | **if** AR.isSatisfiable($\mathcal{O}$, A) **then return** AR.classify($\mathcal{O}$);
13 | **else** add $\langle A, \bot\rangle$ into $H_{\mathcal{O}}$;
14 **foreach** $\langle A, B\rangle \in H_{str} \setminus H_{wk}$ **do**
15 | **if** *not* AR.isSatisfiable($\mathcal{O}$, $A \sqcap \neg B$) **then** add $\langle A, B\rangle$ into $H_{\mathcal{O}}$;
16 **return** $H_{\mathcal{O}}$

4.3.1 Strengthening for Inverse Roles Based on the discussions in Section 4.2, we calculate the strengthening axioms $\mathcal{O}_{\mathcal{I}}^{+}$ for inverse roles according to the following steps:

1. For each $\langle R', S^{-}\rangle \in H_{op}$ where S^{-} does not have an equivalent named role, introduce a new named role S' for S^{-} and update H_{op}.
2. Initialize $\mathcal{O}_{\mathcal{I}}^{+}$ with all the subsumptions between named roles in H_{op}
3. for each $\langle R, R'\rangle$ such that $R = R'^{-}$ is implied by H_{op}, if either of the following two equivalent forms is used, add the other to $\mathcal{O}_{\mathcal{I}}^{+}$:

$$A \sqsubseteq \forall R.B \Leftrightarrow \exists R'.A \sqsubseteq B$$

Here $\langle R, R'\rangle$ and $\langle R', R\rangle$ are treated as different pairs.

4.3.2 Strengthening for Nominals This section explains the calculation of strengthening axioms $\mathcal{O}_{N}^{+}$ for nominals. According to Section 4.2, for each nominal placeholder

N_a, we need to compute its major coexisting label set LS_{N_a}, and choose to add $N_a \sqsubseteq X$ or $N_a \sqcap X \sqsubseteq \bot$ into $\mathcal{O}_N^+$ for each $X \in LS_{N_a}$.

Algorithm 2: Calculate the potential major coexisting label set of N_a in $\mathcal{O}$

Input: Normalized $\mathcal{ALCHOI}$ ontology $\mathcal{O}$ and a concept $N_a \in N_C$
Output: Major coexisting label set LS_{N_a}

1 Initialize a queue Q with label N_a;
2 $Core_{N_a} := \emptyset$; visited $:= \emptyset$;
3 **repeat**
4 poll a label X from Q;
5 **if** *X is not introduced by some* $\bigsqcap A_i \sqsubseteq M \bigsqcup X$ *and* $X \notin$ visited **then**
6 add X to *proc*;
7 **foreach** $\bigsqcap A_i \sqsubseteq M \bigsqcup X \in \mathcal{O}$ **do** add each A_i into Q;
8 **foreach** $\exists S.Y \sqsubseteq X \in \mathcal{O}$ *and* $\langle R, S \rangle \in H_{op}$ *and* $B \sqsubseteq \exists R.Z \in \mathcal{O}$ **do**
9 add B into Q;
10 **foreach** $Y \sqsubseteq \forall S.X \in \mathcal{O}$ *and* $\langle R, S \rangle \in H_{op}$ *and* $B \sqsubseteq \exists R.Z \in \mathcal{O}$ **do**
11 add Z to $Core_{N_a}$;
12 **if** $X \in N_C^\top$ *or some* $B \sqsubseteq \exists R.X \in \mathcal{O}$ **then** add X to $Core_{N_a}$;
13 **until** *Q is empty*;
14 $LS_{N_a} := Core_{N_a}$;
15 **foreach** $X \in Core_{N_a}$ **do**
16 **foreach** $B \sqsubseteq \exists R.X \in \mathcal{O}$ *and* $\langle R, S \rangle \in H_{op}$ *and* $Y \sqsubseteq \forall S.Z \in \mathcal{O}$ **do**
17 add Z to LS_{N_a};
18 **return** LS_{N_a}

Algorithm 2 illustrates the computation of LS_{N_a} in Example 1. From line 3 to 13 we search for the potential core label set $Core_{N_a}$ of N_a, i.e., the concepts that may label an individual of N_a when it is created by the HT-calculus. $Core_{N_a}$ is a subset of LS_{N_a}. We search in the converse direction of the model construction process for the labels X that may cause the appearance of label N_a, which we denote by $X \mapsto N_a$. and put the potential core labels into $Core_{N_a}$. There are three cases that X is added to an individual x according to the calculus:

Case 1: (Line 7) If X is added to x by the axiom $\bigsqcap A_i \sqsubseteq M \sqcup X$, then for every conjunct condition A_i we have $A_i \mapsto X$.

Case 2: (Line 8-9) If X is added to x to by the axiom $\exists S.Y \sqsubseteq X$, then x has an S-successor, which must be introduced by some $B \sqsubseteq R.Z$ provided that $\langle R, S \rangle \in H_{op}$. For every such B there is a potential that $B \mapsto X$.

Case 3: (Line 10) X is added to x by the axiom $Y \sqsubseteq \forall S.X$, then it must have an S-predecessor in the model. Thus when x is created, the incoming role R satisfies $\langle R, S \rangle \in H_{op}$, and for all $B \sqsubseteq \exists R.Z$, Z is a potential core label of x.

Line 12 checks whether X itself can be a core label. A core label can only be introduced through: (1) the initialization step, for which X must be atomic in $\mathcal{O}$; (2) an individual-adding derivation, for which there must be some $B \sqsubseteq \exists R.X \in \mathcal{O}$.

On line 15 to 17 we follow the model construction process to find other major coexisting labels. Similarly to case 3 above, if x has a core label X added by the axiom $B \sqsubseteq \exists R.X$ and $\langle R, S \rangle \in H_{op}$, then Z may be added to X by the axiom $Y \sqsubseteq \forall S.Z$.

The test on line 5 prunes a search branch X in either of two cases: (1) X has been visited. (2) An axiom $\bigsqcap A_i \sqsubseteq M \sqcup X$ has been used on the search path from N_a to X. In case (2), when the model expands, the axiom $\bigsqcap A_i \sqsubseteq M \sqcup X$ has been satisfied and no new labels that potentially introduces N_a will be added.

We show the calculation of the strengthening axioms for N_a in Example 1. Q is initialized with N_a. E is added to Q according to Case 1 based on the axiom $E \sqsubseteq N_a$. When E is processed, it is added to $Core_{N_a}$ in line 12 and also to LS_{N_a} in line 14. D is added to LS_{N_a} in the next loop from line 15 to 17, based on the axioms $A \sqsubseteq \exists R.E$ and $C \sqsubseteq \forall R.D$. Finally we choose to add $N_a \sqsubseteq D$ and $N_a \sqsubseteq E$ based on some heuristic rules.

Termination of the outer loop from line 3 to 13 is ensured by keeping a visited set so that any label will only be processed at most once in the loop. Let n_c and n_{ax} be the number of concepts and axioms in $\mathcal{O}$. One can see that the inner loop on line 7 runs at most n_{ax} times, while the number of runs of the next two inner loops is bounded by the number n_{ax}^2 of pairs of axioms. So the worst-case complexity of the outer loop is $O(n_c \cdot n_{ax}^2)$. The case is similar for the loop from 15 to 17. This procedure needs to be invoked for each concept N_a, the number of invocations is less than n_c. Since $n_c \leq n_{ax}$ in the normalized ontology, the worst-case number of executions is polynomial in n_{ax}.

5 Evaluation

We have implemented our proposed approach in a prototype reasoner WSReasoner. We use the consequence-based reasoner ConDOR r.12 as the main reasoner and the hyper-tableau-based reasoner HermiT 1.3.6[4] as the assistant reasoner. ConDOR supports DL $\mathcal{SH}$ ($\mathcal{ALCH}$ + transitivity axioms), and HermiT supports DL $\mathcal{SROIQ}(\mathcal{D})$, which is more expressive than the $\mathcal{ALCHOI}$. We compared the performance of WSReasoner with the latest versions of mainstream reasoners, including tableau-based reasoners HermiT 1.3.6, Fact++ 1.5.3 and Pellet 2.3.0, as well as a consequence-based reasoner TrOWL 0.8.2. All the experiments were run on a laptop with an Intel Core i7-2670QM 2.20GHz quad core CPU and 16GB RAM running Java 1.6 under Windows 7. We set the Java heap space to 12GB. We did not set the time limit.

We tried all the commonly used, widely available large and complex ontologies that we have access to. Since none of these expressive ontologies are modeled in $\mathcal{ALCHOI}$, we had to make some adjustments. Galen[5] and FMA-constitutionalPartForNS (FMA-cPFNS) are currently available large and complex ontologies. We use three different version of Galen, which are Full-Galen, Galen-Heart, and Galen-EL.[6] We modify them by introducing nominals. In Galen, the concepts starting with a lower case letter and subsumed by *SymbolicValueType* could be nominals which are modeled as concepts. For Galen-Heart and Galen-EL, we used published methods [7] to produce two versions for each of them, which are Galen-Heart-YN1, Galen-Heart-YN2, Galen-EL-YN1, and

[4] HermiT 1.3.6 build 1054, 04/18/2012 release

[5] http://www.co-ode.org/galen/

[6] http://code.google.com/p/condor-reasoner/downloads/list/

Galen-EL-YN2. In addition, we produced Galen-EL-LN1 by introducing norminals only for leaf N-concepts of Galen-EL; and also add disjunction into Full-Galen and produced Galen-Full-UnionN2.

We also used two smaller complex ontologies, Wine and DOLCE. All our ontologies were reduced to $\mathcal{ALCHOI}$ and can be downloaded from our website.[7]

Table 2. Comparison of classification performance

T: Time(seconds); MS: (# of subsumption pairs missing) / (# of total subsumption pairs)

Ontology	**Criteria**	**(Hyper) tableau**			**Consequence-based**	**WSReasoner**	
		HermiT	**Pellet**	**FaCT++**	**TrOWL**	**com-role**	**fast-role**
Wine	T	29.11	377.88	7.33	0.81	7.34	1.22
	MS	0/968	0/968	0/968	1/968	0/968	0/968
DOLCE	T	5.64	8.40	0.92	0.55	8.84	2.09
	MS	0/2595	0/2595	0/2595	390/2595	0/2595	0/2595
Galen-Heart-YN1	T	115.10	-	-	-	7.59	6.94
	MS	0/45,513	-	-	-	0/45,513	0/45,513
Galen-Heart-YN2	T	63.52	-	-	-	9.50	7.19
	MS	0/45,914	-	-	-	0/45,914	0/45,914
Galen-EL-YN1	T	197,090	-	-	-	345,600+	345,600+
	MS	0/431,990	-	-	-	/	/
Galen-EL-YN2	T	289,637	-	-	-	38,350	38,272
	MS	0/457,779	-	-	-	0/457,779	0/457,779
Galen-EL-LN1	T	188,018	-	-	-	771	755
	MS	0/431,990	-	-	-	0/431,990	0/431,990
Galen-Full-UnionN2	T	604,800+	-	-	-	1625	1478
	MS	/	-	-	-	x/(431,255+x)	x/(431,255+x)
FMA-cPFNS	T	667,430	-	-	429.65	21,362	45
	MS	0/481,967	-	-	70/481,967	0/481,967	0/481,967

Note: "-" entry means that the reasoner was unable to classify the ontology due to some problems. "/" entry means the number is not available.

The results of our experiment are shown in Table 2. Since the time limit is not set, some tasks may take several days or more to finish. The '+' sign indicates the tasks are not finished within the time shown before '+'. We also report the number of missed subsumptions, since some of the reasoners are not complete, such as TrOWL and possibly ours. # represents number in Table 2 and 3. The total number includes all pairs of subsumptions between any $A, B \in N_C^{\top,\bot}$ except for $\bot \sqsubseteq A$ and $A \sqsubseteq \top$. The complete result is obtained from HermiT. For Galen-Full-UnionN2, HermiT does not get the results, while WSReasoner gets 431,255 subsumptions, but we do not know the number of missed pairs, so we denote it by x.

Since computing complete role hierarchy H_{op} takes a considerable amount of time for our assistant reasoner HermiT, we implement two versions of WSReasoner. The version 'fast-role' simply computes the reflexive-transitive relations between roles,[8] while the version 'com-role' request HermiT for a complete H_{op} on O.

[7] http://isew.cs.unb.ca/wsreasoner/resources/ontologies/

[8] to simplify implementation, we extract all object property axioms and request HermiT for H_{op}

As we can see in Table 2, WSReasoner is able to classify eight of nine ontologies, and get no fewer subsumptions than any other reasoners. On various FMA-cPFNS and versions of Galen ontologies, WSReasoner outperforms all the other reasoners considerably except for Galen-EL-YN1, on which the verification stage has more than 320,000 pairs to verify, however, it takes only several minutes to determine that WSReasoner is likely to be slower than HermiT. For the two smaller ontologies Wine and DOLCE, the fast-role approach still outperforms HermiT and Pellet and even com-role outperforms the two reasoners on Wine.

Table 3. Statistics of WSReasoner

Added-axioms: # of axioms in $\mathcal{O}_N^+$ Add-pairs: # of pairs in $H_{str} \setminus H_{wk}$
True-pairs: # of pairs verified to be correct Verify-time: verification time
fast-role-T: The role classification time using the fast-role approach (Seconds)
com-role-T: The role classification time using the comp-role approach (Seconds)

Ontology	Wine	DOLCE	Galen-Heart-YN1	Galen-Heart-YN2	Galen-EL-YN1	Galen-EL-YN2	Galen-EL-LN1	Galen-Full-UnionN2	FMA-cPFNS
Added-Axioms	36	95	2	50	79,594	981	12	1100	0
Add-Pairs	0	0	2	403	323,294	28,287	45	259	0
True-Pairs	0	0	2	403	-	25,789	0	0	0
fast-role-T	0.16	0.77	0.639	0.56	1.60	1.25	1.63	2.82	0.39
com-role-T	6.50	7.37	2.09	2.53	22.99	19.57	19.37	15.30	21322.54
Verify-Time	0	0	3.39	3.84	-	38,252	649.24	793.26	0

Table 3 shows some statistics of our reasoner on different phases. FMA-cPFNS only needs one round of classification. Wine and DOLCE need two rounds of classification but no verifications are needed, which indicates the nominals does not bring any new subsumptions. The number of strengthening axioms added for these ontologies varies a lot. The verification time becomes larger as the size of $H_{str} \setminus H_{wk}$ increases. Comparing the role classification time, fast-role is considerably faster than com-role, especially for the ontology FMA-cPFNS on which the com-role approach takes 5 hours to finish. In summary, the main factors affecting the reasoning time are: (1) The approach to choose for role hierarchy calculation. (2) the number of pairs in $H_{str} \setminus H_{wk}$.

To evaluate the performance on other existing ontologies, we also ran WSReasoner on a test suite provided by ORE 2012.[9] We used the RDF version of the ontologies in the OWL DL and OWL EL datasets, which contains 107 and 8 real-world ontologies respectively. We set the maximum Java heap space to 1GB and add an additional JVM setting `-DentityExpansionLimit=480000` to ensure successful loading of all the ontologies using OWL API. The time limit is set to 1 hour.

The results are shown in Table 4. The columns "OWL DL" and "OWL EL" refer to the two datasets, while "DL-ALCHOI" and "EL-ALCHOI" refer to the $\mathcal{ALCHOI}$ ontologies in these datasets, respectively. As seen in the table, all the $\mathcal{ALCHOI}$ ontologies have been correctly classified. Complete hierarchies of 5 ontologies in the OWL

[9] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/datasets/classification.zip

DL dataset are not obtained because their languages are beyond $\mathcal{ALCHOI}$. The largest ontology `gazetteer` in the OWL DL dataset (containing 150979 classes) causes a out-of-memory exception.

Table 4. Evaluation results on the ORE test suite

ALT: Average Loading Time ART: Average Reasoning Time
TOTAL: # of ontologies in the dataset
CORRECT: # of correctly classified ontologies
INCORRECT: # of incorrectly classified ontologies
NO-REF: # of ontologies with no reference results
EXCEPTION: # of ontologies that cause an exception
TIMEOUT: # of ontologies that cause a timeout

Outcomes	**DL-ALCHOI**	**OWL DL**	**EL-ALCHOI**	**OWL EL**
ALT (ms)	105	323	320	454
ART (ms)	632	1263	277	503
TOTAL	34	107	6	8
CORRECT	18	47	5	6
INCORRECT	0	5	0	0
NO-REF	16	54	1	2
EXCEPTION	0	1	0	0
TIMEOUT	0	0	0	0

6 Conclusions and Future Work

We present an approach combining two reasoners based on ontology weakening and strengthening to classify large and complex ontologies, for which tableau-based reasoners may take a long time or use much memory while consequence-based reasoners cannot support all the constructors in the target language. We use a consequence-based reasoner supporting a DL less expressive than the target language as the main reasoner to do the majority (sometimes all) of the work, and a more expressive but slower tableau-based reasoner to assist it in verifying the results. In the experiment dataset shown in Table 2, WSReasoner's results show better efficiency than the tableau-based reasoners in most large and complex ontologies, and no fewer subsumptions than other reasoners except for one ontology for which the result is not obtained by WSReasoner.

The weaknesses of this WSreasoner for $\mathcal{ALCHOI}$ are: (1) the complete role hierarchy classification may take a lot of time; (2) if the number of pairs verified is large, the procedure still takes a lot of time in the verification stage. The advantages of the WS approach are: (1) it may achieve better classification efficiency than tableau-based reasoners; (2) it extends the capability of consequence-based reasoners; (3) the reasoners underneath can be customized to classify different ontologies.

In the future, we will try to prove the theoretical completeness of this weakening and strengthening approach for $\mathcal{ALCHOI}$ ontology classification while optimizing the

algorithm. We will further try to apply the weakening and strengthening approach based on different reasoners and address more constructors for more expressive languages.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: IJCAI 2005, Proceedings of the 19th International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005. pp. 364–369 (2005)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge Univ Pr (2010)
3. Glimm, B., Horrocks, I., Motik, B., Stoilos, G.: Optimising ontology classification. In: Patel-Schneider, P., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J., Horrocks, I., Glimm, B. (eds.) The Semantic Web - ISWC 2010, Lecture Notes in Computer Science, vol. 6496, pp. 225–240. Springer Berlin / Heidelberg (2010)
4. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. Journal of Automated Reasoning 39, 249–276 (October 2007)
5. Kazakov, Y.: Consequence-driven reasoning for horn $\mathcal{SHIQ}$ ontologies. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 2040–2045 (2009)
6. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of $\mathcal{EL}$ ontologies. In: ISWC 2011, 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011. Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
7. Kazakov, Y., Krötzsch, M., Simančík, F.: Practical reasoning with nominals in the $\mathcal{EL}$ family of description logics. In: Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR'12) (2012)
8. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36, 165–228 (September 2009)
9. Ren, Y., Pan, J.Z., Zhao, Y.: Soundness preserving approximation for TBox reasoning. In: AAAI 2010, Proceedings of the 24th AAAI Conference on Artificial Intelligence, Atlanta, Georgia, USA, July 11-15, 2010. AAAI Press 2010 (2010)
10. Selman, B., Kautz, H.: Knowledge compilation and theory approximation. Journal of the ACM (JACM) 43(2), 193–224 (1996)
11. Simančík, F., Kazakov, Y., Horrocks, I.: Consequence-based reasoning beyond horn ontologies. In: IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, 2011. pp. 1093–1098 (July 2011)
12. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A Practical OWL-DL Reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)
13. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. Automated Reasoning pp. 292–297 (2006)

MASTRO: A Reasoner for Effective Ontology-Based Data Access

Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, Domenico Fabio Savo

Dip. di Ing. Informatica, Automatica e Sistemistica
Sapienza Università di Roma
lastname@dis.uniroma1.it

Abstract. In this paper we present MASTRO, a Java tool for ontology-based data access (OBDA) developed at Sapienza Università di Roma. MASTRO manages OBDA systems in which the ontology is specified in a logic of the *DL-Lite* family of Description Logics specifically tailored to ontology-based data access, and is connected to external data management systems through semantic mappings that associate SQL queries over the external data to the elements of the ontology. Advanced forms of integrity constraints, which turned out to be very useful in practical applications, are also enabled over the ontologies. Optimized algorithms for answering expressive queries are provided, as well as features for intensional reasoning and consistency checking. MASTRO has been successfully used in several projects carried out in collaboration with important organizations, on which we briefly comment in this paper.

1 Introduction

In this paper we present the current version of MASTRO, a system for ontology-based data access (OBDA) developed at Sapienza Università di Roma. MASTRO allows users for accessing external data sources through an ontology expressed in a fragment of the W3C Web Ontology Language (OWL).

As in data integration systems [11], mappings are used in OBDA to specify the semantic correspondence between a unified view of the domain (called global schema in data integration terminology) and the data stored at the sources. The distinguishing feature of the OBDA approach, however, is the fact that the global unified view is specified using an ontology language, which typically allows to provide a rather rich conceptualization of the domain of interest, that is independent from the representation adopted for the data stored at the sources. This choice provides several advantages: it allows for a declarative approach to data access and integration and provides a specification of the domain that is independent from the data layer; it realizes logical/physical independence of the information system, which is therefore more accessible to non-experts of the underlying databases; the conceptual approach to data access does not impose to fully integrate the data sources at once, as it often happens in data integration mediator-based system, but the design can be carried out in an incremental way;

the conceptual model available on the top of the system provides a common ground for the documentation of the data stores and can be seen as a formal specification for mediator design.

MASTRO has solid theoretical basis [3, 4]. In the current version of MASTRO, ontologies are specified in *DL-Lite*$_{A,id,den}$, a logic of the *DL-Lite family* of tractable Description Logics (DLs), which are specifically tailored to the management and querying of ontologies in which the extensional level, i.e., the data, largely dominates the intensional level. From the point of view of the expressive power, *DL-Lite*$_{A,id,den}$ captures the main modeling features of a variety of representation languages, such as basic ontology languages and conceptual data models. Furthermore, it allows for specifying advanced forms of identification constraints [5] and denials [10], that are not part of OWL 2, the current W3C standard language for specifying ontologies.

Answering unions of conjunctive queries in OBDA systems managed by MASTRO can be done through a very efficient technique that reduces this task to standard SQL query evaluation. Indeed, conjunctive query answering has been shown to be in LOGSPACE (in fact in AC^0) w.r.t. data complexity [4], i.e., the complexity measured only w.r.t. the extensional level, which is the same complexity of evaluating SQL queries over plain relational databases. One key feature of the current version of MASTRO, wrt previous ones [2], is that it adopts the *Presto* algorithm [15] for first-order query rewriting.

MASTRO is developed in Java and can be connected to any data management system allowing for a JDBC connection, e.g., a relational DBMS. In those cases in which several, possibly non-relational, sources need to be accessed, MASTRO can be coupled with a relational data federation tool[1], which wraps sources and represents them as a single (virtual) relational database.

The rest of the paper is organized as follows. In Section 2, we briefly describe the framework of ontology-based data access. In Section 3, we describe the query answering algorithm of the MASTRO system. In Section 4, we report on some real world information integration applications where MASTRO has been successfully trialed. In Section 5, we conclude the paper by discussing related work.

2 Ontology-based data access

In OBDA, the aim is to give users access to a data source or a collection thereof, by means of a high-level conceptual view specified as an ontology. The ontology is usually formalized in Description Logics (DLs) [1], which are at the basis of OWL. These logics allow one to represent the domain of interest in terms of *concepts*, denoting sets of objects (corresponding to OWL *classes*), *roles*, denoting binary relations between objects (OWL *object properties*), and *attributes*, denoting relations between objects and values from predefined domains (OWL *data properties*).

[1] E.g., IBM WebSphere Application Server (`http://www.ibm.com/software/webservers/appserv/was/`), Oracle Data Service Integrator (`http://www.oracle.com/us/products/middleware/data-integration/`).

A DL ontology is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ [1] where $\mathcal{T}$, called *TBox*, is a finite set of intensional assertions, and $\mathcal{A}$, called *ABox*, is a finite set of instance assertions, i.e, assertions on individuals. Different DLs allow for different kinds of TBox and/or ABox assertions.

The semantics of an ontology is given in terms of first-order interpretations [1]. An interpretation $\mathcal{I}$ is a *model* of an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ if it satisfies all assertions in $\mathcal{T} \cup \mathcal{A}$, where the notion of satisfaction depends on the constructs and axioms allowed by the specific DL in which $\mathcal{O}$ is expressed.

Among the extensional reasoning tasks w.r.t. a given ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, the most relevant ones are *ontology satisfiability* and *query answering*.

In particular, we are interested in the class of *conjunctive queries* (CQ). A CQ q over an ontology $\mathcal{O}$ (resp. TBox $\mathcal{T}$) is an expression of the form $q(\boldsymbol{x}) \leftarrow \exists \boldsymbol{y}.conj(\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{x}$ are the so-called *distinguished variables*, $\boldsymbol{y}$ are existentially quantified variables called the *non-distinguished* variables, and $conj(\boldsymbol{x}, \boldsymbol{y})$ is a conjunction of atoms of the form $A(z)$, $P(z, z')$, $U(z, z')$ where A is a concept name, P is a role name and U is an attribute name, and z, z' are either variables in $\boldsymbol{x}$ or in $\boldsymbol{y}$ or constants. The *arity* of q is the arity of $\boldsymbol{x}$. A CQ of arity 0 is called a *boolean conjunctive query*. A *union of conjunctive queries* (UCQ) is a query of the form $q(\boldsymbol{x}) \leftarrow \bigvee_i \exists \boldsymbol{y_i}.conj(\boldsymbol{x}, \boldsymbol{y_i})$.

Given a query $q(\boldsymbol{x})$ (either a conjunctive query or an union of conjunctive queries) and an ontology $\mathcal{O}$, the *certain answers* to $q(\boldsymbol{x})$ over $\mathcal{O}$ is the set $cert(q, \mathcal{O})$ of all tuples $\boldsymbol{t}$ of constants appearing in $\mathcal{O}$, such that, when substituted for the variables $\boldsymbol{x}$ in $q(\boldsymbol{x})$, we have that $\mathcal{O} \models q(\boldsymbol{t})$, meaning that $\boldsymbol{t}^{\mathcal{I}} \in q^{\mathcal{I}}$ for every $\mathcal{I} \in Mod(\mathcal{O})$. Notice that the answer to a boolean query is either the empty tuple, considered as $true$, or the empty set, considered as $false$.

In OBDA, the extensional level is not represented directly by an ABox, but rather by a database that is connected to the TBox by means of suitable mapping assertions[2]. Such *mapping assertions* have the form $\Phi \rightsquigarrow \Psi$, where Φ, called the *body* of the assertion, is an arbitrary SQL query over the underlying database, and Ψ, called the *head*, is a CQ over the TBox $\mathcal{T}$. Intuitively, a mapping assertion specifies that the tuples returned by the SQL query Φ are used to generate the facts that instantiate the concepts, roles, and attributes in Ψ.

All the notions given above can be easily generalized to OBDA systems, where a TBox $\mathcal{T}$ is connected to an external database $\mathcal{D}$ through mappings $\mathcal{M}$, denoted $\langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$. In particular, the *models* of $\langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$ are those interpretations of $\mathcal{T}$ that satisfy the assertions in $\mathcal{T}$ and that are consistent with the tuples retrieved by $\mathcal{M}$ from $\mathcal{D}$ (see [13] for the formal details). Satisfiability amounts to checking whether $\langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$ admits at least one model, while answering a query Q amounts to computing the tuples that are in the evaluation of Q in every model of $\langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$.

Mastro is able to deal with DL TBoxes that are expressed in *DL-Lite*$_{A,id,den}$, a member of the *DL-Lite* family of lightweight DLs [4]. In such DLs, a good tradeoff is achieved between the expressive power of the TBox lan-

[2] Note that, in the following, with some abuse of terminology, when we use the term "ontology" in the context of OBDA, we implicitly refer to the TBox only.

guage used to capture the domain semantics, and the computational complexity of inference, in particular when such a complexity is measured w.r.t. the size of the data.

Basic *DL-Lite*$_{A,id,den}$ expressions are defined as follows:

$$\begin{array}{lll} B \longrightarrow A \mid \exists Q \mid \delta(U) & Q \longrightarrow P \mid P^- & E \longrightarrow \rho(U) \\ C \longrightarrow B \mid \neg B & R \longrightarrow Q \mid \neg Q & F \longrightarrow T_1 \mid \cdots \mid T_n \\ V \longrightarrow U \mid \neg U & & \end{array}$$

where, A, P, and P^- denote an *atomic concept*, an *atomic role*, and the *inverse of an atomic role* respectively; $\delta(U)$ (resp. $\rho(U)$) denotes the *domain* (resp. the *range*) of an *attribute* U, i.e., the set of objects (resp. values) that U relates to values (resp. objects); $T_1, \ldots, T_n$ are unbounded pairwise disjoint predefined value-domains; B is called *basic concept*.

A *DL-Lite*$_{A,id,den}$ TBox is a finite set of the following assertions:

$B \sqsubseteq C$	(*concept inclusion assertion*)
$Q \sqsubseteq R$	(*role inclusion assertion*)
$U \sqsubseteq V$	(*attribute inclusion assertion*)
$E \sqsubseteq F$	(*value-domain inclusion assertion*)
(funct Q)	(*role functionality assertion*)
(funct U)	(*attribute functionality assertion*)
(id B $\pi_1, \ldots, \pi_n$)	(*identification assertion*)
$\forall \boldsymbol{y}.conj(\boldsymbol{t}) \rightarrow \bot$	(*denial assertion*)

In identification assertions [5], π_i is a *path*, i.e., an expression built according to the following syntax: $\pi \longrightarrow S \mid D? \mid \pi_1 \circ \pi_2$, where S denotes an atomic role, the inverse of an atomic role, an attribute, or the inverse of an attribute, $\pi_1 \circ \pi_2$ denotes the composition of paths π_1 and π_2, and $D?$, called *test relation*, represents the identity relation on instances of D, which can be a basic concept or a value-domain. Test relations are used to impose that a path involves instances of a certain concept or value-domain. In *DL-Lite*$_{A,id,den}$, identification assertions are *local*, i.e., at least one $\pi_i \in \{\pi_1, ..., \pi_n\}$ has length 1, i.e., it is an atomic role, the inverse of an atomic role, or an attribute. Intuitively, an identification assertion of the above form asserts that for any two different instances o, o' of B, there is at least one π_i such that o and o' differ in the set of their π_i-fillers, that is the set of objects that are reachable from o by means of π_i.

In denial assertions [10], $conj(\boldsymbol{y})$ is defined as for boolean CQs. Intuitively, a denial assertion of the above form states that there must not exist any tuple $\boldsymbol{y}$ satisfying $conj(\boldsymbol{y})$, i.e., that the answer to the boolean query $q() \leftarrow \exists \boldsymbol{y}.conj(\boldsymbol{y})$ must be empty.

Finally, in a *DL-Lite*$_{A,id,den}$ TBox $\mathcal{T}$, the following condition must hold: each role or attribute that either is functional in $\mathcal{T}$ or appears (in either direct or inverse direction) in a path of an identification assertion in $\mathcal{T}$ is not specialized, i.e., it does not appear in the right-hand side of assertions of the form $Q \sqsubseteq Q'$ or $U \sqsubseteq U'$.

Mapping assertions handled by MASTRO are assertions of the form $\Phi \rightsquigarrow \Psi$, where Φ is an arbitrary SQL query over the underlying database, and Ψ is a conjunction of atoms whose predicates are the concepts, roles, and attributes of the

TBox. Notice that, due to the fact that Ψ is a conjunction of atoms (as opposed to a query, possibly with existentially quantified variables), such mappings can be considered as a special form *global-as-view* (GAV) mappings [11].

In order to overcome the so-called *impedance mismatch* between the database, storing values, and the TBox, to be interpreted over a domain of objects, the mapping assertions are used in MASTRO to specify how to construct abstract objects from the tuples of values retrieved from the database. This is done by allowing one to use function symbols in the atoms in Ψ: together with the values retrieved by Φ, such function symbols generate so called *object terms*, which serve as object identifiers for individuals in the ontology. We notice that the semantics we adopt in MASTRO establishes that different terms denote different objects (unique name assumption), so that different terms never need to be equated during reasoning, which is coherent with the assumption of not having existentially quantified variables in the body of mappings.

For the logics of the *DL-Lite* family it has been shown that for unions of conjunctive queries (UCQs), under the unique name assumption, query answering can be carried out efficiently in the size of the data, by reducing it to SQL query evaluation over the ABox seen as a database [4]. Also satisfiability, which is easily reducible to query answering, can be solved through the same mechanism. Such techniques are implemented in MASTRO, we refer to [4, 13] for a more complete treatment.

As an example, consider the OBDA system $\langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$, where the TBox $\mathcal{T}$ is constituted by the following set of intensional assertions: $\{NationalFlight \sqsubseteq Flight, InternationalFlight \sqsubseteq Flight\}$, $\mathcal{D}$ is a database constituted by a set of relations with the following signature:

```
FL_TB[fl_num:string, departure:integer, arrival:integer],
AIRPORT_TB[airpt_code:integer, name:string, country:string],
```

and $\mathcal{M}$ contains the following mapping assertions:

```
SELECT fl_num
FROM FL_TB,AIRPORT_TB A1,AIRPORT_TB A2
WHERE departure = A1.airpt_code and          ⇝ NationalFlight(fl(fl_num))
arrival = A2.airpt_code and
A1.country = 'IT' and A2.country = 'IT'

SELECT fl_num
FROM FL_TB,AIRPORT_TB A1,AIRPORT_TB A2
WHERE departure = A1.airpt_code and          ⇝ InternationalFlight(fl(fl_num))
arrival = A2.airpt_code and
(A1.country != 'IT' or A2.country != 'IT')
```

which specify how to construct instances of the ontology concepts $NationalFlight$ and $InternationalFlight$ starting from the database relations FL_TB and AIRPORT_TB.

3 Query Answering

In this section we describe the query rewriting process of the MASTRO system. The technique is purely intensional and is performed in three steps (see Figure 1):

1. *TBox rewriting:* The first step rewrites the input UCQ according to the knowledge expressed by the TBox. The rewriting, performed using the Presto algorithm [15], produces as output a non-recursive Datalog program, which encodes the knowledge expressed by the TBox and the user query. The output Datalog program contains the definition of auxiliary predicates, not belonging to the alphabet of the ontology.
2. *Datalog Unfolding:* The output of the first step is then unfolded into a new UCQ by means of the *Datalog Unfolding* algorithm. It consists of a classic rule unfolding technique which eliminates all the auxiliary predicate symbols introduced by the Presto algorithm and produces a final UCQ expressed in terms of ontology concepts, roles, and attributes.
3. *Mapping Unfolding:* The last step takes the unfolded UCQ and the mapping assertions as input and produces an SQL query which can be directly evaluated over the data sources. In particular, the mapping assertions are first *split* into assertions of a simpler form, in which the head of every mapping assertion contains only a single ontology predicate; then, the final reformulation is produced through a mapping unfolding step, as described in [13].

More specifically, the Presto algorithm is an optimization of the well-known *PerfectRef* [4]. The latter, depending on the particular TBox being used, may lead to huge UCQs, consisting of many possibly redundant queries which can be eliminated from the final result. Presto tries to overcome such issue, rewriting the user query into a Datalog program whose rules encode only necessary expansion steps, thus preventing the generation of useless queries. It is important to note that after the Datalog unfolding program, one can have again an exponential number of queries, but MASTRO experiences on real world application showed a dramatic performance improvement w.r.t. to the performance of *PerfectRef*.

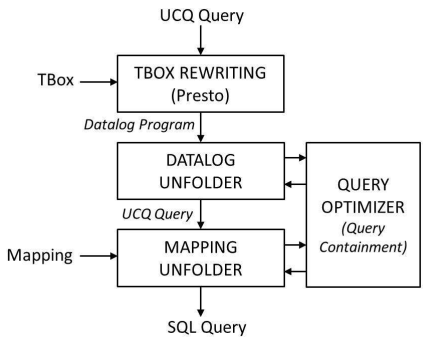


Fig. 1. The MASTRO rewriting process

4 The system at work: experiences on real cases

The usefulness of OBDA and the efficiency of the MASTRO system were proved by several real world applications in which it has been experimented. In the following, we report on the experiments carried out with Banca Monte dei Paschi di Siena (MPS), the Italian Ministry of Economy and Finance (MEF) and the Telecom Italia, the main Italian telephone company. Other experiments have been recently carried out with SELEX Sistemi Integrati (SELEX-SI), and Accenture [2].

Monte dei Paschi di Siena. Within a joint project with Banca Monte dei Paschi di Siena (MPS)[3], Free University of Bozen-Bolzano, and Sapienza Università di Roma, we used MASTRO for accessing a set of data sources from the actual MPS data repository by means of an ontology [16]. In particular, we focused on the data exploited by MPS personnel for risk estimation in the process of granting credit to bank customers. A 15 million tuple database, stored in 12 relational tables managed by the IBM DB2 RDBMS, has been used as data source collection in the experimentation. Such source data are managed by a dedicated application, which is in charge of guaranteeing data integrity (in fact, the underlying database does not force constraints on data). Not only the application performs various updates, but data is updated on a daily basis to identify connections between customers that are relevant for the credit rating estimation.

The main challenge that we tackled within the experimentation was the ontology and mapping design. This was a seven man-months process that required to both inspect the data source and interview domain experts, and was complicated by the fact that the source was managed by a specific application. The resulting OBDA system is defined in terms of approximately 600 $DL\text{-}Lite_{A,id}$ assertions over 79 concepts, 33 roles and 37 attributes, and 200 mapping assertions.

The experimentation showed that the usefulness of the MASTRO system goes beyond data integration applications and embraces data quality management. In particular, it confirmed the importance of several distinguished features of our system, namely, identification constraints and denial constraints, which have been used extensively to model important business rules. Notably, checking that such rules were satisfied by data retrieved from the sources through mappings led to highlight unexpected incompleteness and inconsistency in the data sources.

Our work has also pointed out the importance of the ontology itself, as a precious documentation tool for the organization. Indeed, the ontology developed in our project is adopted in MPS as a specification of the relevant concepts in the organization. At present we are still working with MPS in order to extend the work to cover the core domain of the MPS information system, with the idea that the ontology-based approach could result in a basic step for the future IT architecture evolution.

[3] MPS is one of the main banks, and the head company of the third banking group in Italy (see `http://english.mps.it/`).

Italian Ministry of Economy and Finance. MASTRO has been used within a joint project between Sapienza Università di Roma and the Italian Ministry of Economy and Finance (MEF). The main objectives of the project have been: the design and specification in *DL-Lite*$_A$ of an ontology for the domain of the Italian public debt; the realization of the mapping between the ontology and relational data sources that are part of the management accounting system currently in use at the ministry; the definition and execution of queries over the ontology aimed at extracting data of core interest for MEF users. In particular, the information returned by such queries relates to sales of bonds issued by the Italian government, maturities of bonds, monitoring of various financial products, etc., and are at the basis of various reports on the overall trend of the national public debt.

The Italian public dept ontology is over an alphabet containing 164 atomic concepts, 47 atomic roles, 86 attributes, and comprises around 1440 *DL-Lite*$_A$ assertions. The 300 mapping assertions involve around 60 relational tables managed by Microsoft SQLServer. We tested a very high number of queries and produced through MASTRO several reports of interest for the ministry. We point out that around 80% of the queries we tested could be executed only thanks to a series of further optimizations introduced in the system that, due to lack of space, we cannot describe here.

Telecom Italia. We finally describe a project we are carrying out in the domain of network inventory systems, together with Telecom Italia, the main Italian company for telecommunication services, which is also a world leading company in this field. The main objectives of the project are (*i*) the specification of an ontology that formalizes the entire telecommunication network owned by Telecom Italia and (*ii*) the analysis through the ontology of the information systems that are currently used for network management. The ontology we are going to develop can be partitioned into four layers: *Infrastructures and territory layer*, which represents main infrastructures used to realize the network, the way in which network elements (e.g., cables, apparatus, connection points) are localized into such infrastructures, and how both infrastructures and network elements are localized with respect to the territory; *network topology layer*, which represents how connections are realized into the network, essentially representing it as a graph in which edges represent elementary connections among apparatus, and nodes represent apparatus which realizes signal permutations between elementary connections; *service layer,* which represents all telecommunication services that are deployed on the network and offered to customer (e.g., voice communication, ADSL, voip); *data layer*, which represents the actual data exchanged on the network (e.g., data on telephone calls, internet access). In each such layer, the ontology provides the means to precisely represent the current state of the world, and, when considered of interest, also captures past situations, for example to provide tracks to all changes to which certain in the network have undergone. The use of identification assertions and epistemic constraints turned out to be crucial for faithful representation of such aspects.

5 Discussion

Accessing (possibly disperse) data through a virtual global schema has been deeply investigated in the last two decades in the field of data integration [11, 8]. From the modeling perspective, however, the main systems produced by this research suffer from some weakness, mainly due to the limited expressive power of the languages provided to model the global schema of the integration system. In this respect, Mastro aims at overcoming this limitation by providing the best expressive power allowed while preserving tractability of conjunctive query answering and of the integration tasks. As for the mappings, Mastro adopts a powerful form of the so-called Global-As-View (GAV) mappings [11], and provides optimized algorithms for rewriting global queries with respect their specification.

To the best of our knowledge, the only existing system designed for the same aims of Mastro is Quest [14], which has indeed common roots with our tool. Quest is a system for query answering over *DL-Lite*$_A$ ontologies, which can work in both "classical" (i.e., with a local ABox) and "virtual" mode (i.e., as an OBDA system). Quest implements specific optimizations for query answering, which in particular exploit completeness of the ABox with respect to the TBox. Although first experiments show effectiveness of Quest in the classical scenario [14], its usage in the virtual mode is in a still preliminary stage. In particular, we tried to compare Quest with Mastro in the OBDA scenario of the Italian Ministry of Economy and Finance described described in the previous section. Unfortunately, we have not been able to perform such experiments for two reasons: (i) the data source of this application is an SQL Server database; since Quest does not support this DBMS, we could not compare query answering in the two systems; (ii) Quest was not actually able to compute the TBox rewriting of the 23 queries used in our experiments, which are very long conjunctions of atoms, so we could not even compare the query rewriting performances of the two systems.

Nyaya [6] is a novel system which allows for query answering over ontologies specified into linear Datalog$^{\pm}$, a language that essentially corresponds to *DLR-Lite* [3] (i.e., to the extension of *DL-Lite* with n-ary predicates), and allows for FOL-rewritable query answering of UCQs. In Nyaya, Datalog$^{\pm}$ ontologies are mapped through plain Datalog rules to a specific centralized storage system which maintains both data and meta-data according to the Nyaya metamodel. As in Mastro, answering a query posed over such an ontology is done by first rewriting the query according to the ontology, using an algorithm which can be seen as a variation of the PerfectRef algorithm of [4], and then rewriting it according to the mapping, which is done in Nyaya through standard unfolding. Nyaya does not present particular optimizations for both the rewriting steps, whereas it concentrates in optimizing centralized data storage. In this respect, it is not specifically tailored to data integration and cannot be directly applied in an efficient way to this setting.

Other *DL-Lite*-based approaches and reasoners have been developed, which, however, are not able to deal with full OBDA scenarios. In [9] an alternative approach to query answering is presented. Besides a (less complex) query reformu-

lation step, such an approach requires to suitably "extend" the ABox (managed by a RDBMS) with the aim of reducing the amount of rewritten queries produced by the reformulation step. The experimental results support well this approach (notice that in MASTRO the size of the reformulation may be exponential in the size of the input query). However, the ABox manipulation that it requires makes it extremely difficult to apply this approach in an OBDA scenario.

The REQUIEM reasoner [12] implements a rewriting algorithm which reduces the number of queries in the final reformulation, still being purely intensional like MASTRO. However, it currently supports none of the MASTRO advanced features, such as identification or EQL constraint management, nor mappings to external databases.

The OWLGres prototype [19], which allows for TBox specification in *DL-Lite*, uses the PostgreSQL DBMS for the storage of the ABox, and provides conjunctive query processing. The algorithm for query answering implemented in OWLGres, however, is not complete with respect to the computation of the certain answers to user queries.

MASTRO can also be compared with ontology reasoners which support DLs different from *DL-Lite*, and in particular with their query answering capabilities. In this respect, well-known DL reasoners such as RacerPro [7], Pellet [18], Fact++ [20], and HermiT [17] provide only limited forms of query answering, i.e., instance checking/retrieval or *grounded* conjunctive query answering (c.f. [2]), since they are essentially focused on standard DL reasoning services. Although some optimizations have been implemented, such systems are not able to deal with very large ABoxes (e.g., with several millions of membership assertions) as the ones we considered in our experiments. This is mainly due to the inherent computational complexity of answering queries in the expressive DL languages supported by the above mentioned systems.

Acknowledgments. This research has been partially supported by the EU under FP7 project ACSI – Artifact-Centric Service Interoperation (grant n. FP7-257593), and by Regione Lazio under the project "Integrazione semantica di dati e servizi per le aziende in rete".

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
2. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The Mastro system for ontology-based data access. *Semantic Web J.*, 2(1):43–53, 2011.
3. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. In *Proc. of KR 2006*, pages 260–270, 2006.
4. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, 39(3):385–429, 2007.

5. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Path-based identification constraints in description logics. In *Proc. of KR 2008*, pages 231–241, 2008.
6. R. de Virgilio, G. Orsi, L. Tanca, and R. Torlone. Semantic data markets: a flexible environment for knowledge management. In *Proc. of CIKM 2011*, pages 1559–1564, 2011.
7. V. Haarslev, R. Möller, and M. Wessel. Description logic inference technology: Lessions learned in the trenches. In *Proc. of DL 2005*, volume 147 of *CEUR,* `ceur-ws.org`, 2005.
8. A. Y. Halevy, A. Rajaraman, and J. Ordille. Data integration: The teenage years. In *Proc. of VLDB 2006*, pages 9–16, 2006.
9. R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev. The combined approach to query answering in *DL-Lite*. In *Proc. of KR 2010*, pages 247–257, 2010.
10. D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo. Inconsistency-tolerant first-order rewritability of dl-lite with identification and denial assertions. In *Proc. of DL 2012*, volume 846 of *CEUR,* `ceur-ws.org`, 2012.
11. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS 2002*, pages 233–246, 2002.
12. H. Pérez-Urbina, B. Motik, and I. Horrocks. A comparison of query rewriting techniques for *DL-lite*. In *Proc. of DL 2009*, volume 477 of *CEUR,* `ceur-ws.org`, 2009.
13. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.
14. M. Rodriguez-Muro and D. Calvanese. High performance query answering over dl-lite ontologies. In *Proc. of KR 2012*, 2012. To appear.
15. R. Rosati and A. Almatelli. Improving query answering over *DL-Lite* ontologies. In *Proc. of KR 2010*, pages 290–300, 2010.
16. D. F. Savo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodríguez-Muro, V. Romagnoli, M. Ruzzi, and G. Stella. MASTRO at work: Experiences on ontology-based data access. In *Proc. of DL 2010*, volume 573 of *CEUR,* `ceur-ws.org`, pages 20–31, 2010.
17. R. Shearer, B. Motik, and I. Horrocks. HermiT: A highly-efficient OWL reasoner. In *Proc. of OWLED 2008*, volume 432 of *CEUR,* `ceur-ws.org`, 2008.
18. E. Sirin and B. Parsia. Pellet: An OWL DL reasoner. In *Proc. of DL 2004*, volume 104 of *CEUR,* `ceur-ws.org`, 2004.
19. M. Stocker and M. Smith. Owlgres: A scalable OWL reasoner. In *Proc. of OWLED 2008*, volume 432 of *CEUR,* `ceur-ws.org`, 2008.
20. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of IJCAR 2006*, pages 292–297, 2006.

A Rigorous Characterization of Classification Performance – A Tale of Four Reasoners

Yong-Bin Kang[†], Yuan-Fang Li[†], Shonali Krishnaswamy[†,§]

[†] Faculty of IT, Monash University, Australia
{yongbin.kang,yuanfang.li,shonali.krishnaswamy}@monash.edu
[§] Institute for Infocomm Research, A*STAR, Singapore

Abstract. A number of ontology reasoners have been developed for reasoning over highly expressive ontology languages such as OWL DL and OWL 2 DL. Such languages have, as a consequence of high expressivity, high worst-case complexity. Therefore, reasoning tasks such as classification sometimes take considerable time on large and complex ontologies. In this paper, we carry out a comprehensive comparative study to analyze classification performance of four widely-used reasoners, FaCT++, HermiT, Pellet and TrOWL, using a dataset of over 300 real-world ontologies. Our investigation on correlating reasoner performance with ontology metrics using machine learning techniques also provides additional insights into the hardness of individual ontologies.

1 Introduction

Ontology reasoning tasks such as classification and consistency checking are fundamental to semantics-enabled applications. Very expressive ontology languages that can model complex domain knowledge have been designed and are widely used in a number of domains. Such languages include OWL DL [14] and its successor, OWL 2 DL [11]. High expressivity, however, incurs high computational complexity. For the core reasoning tasks of classification and consistency checking, OWL DL is NExpTime-complete, while OWL 2 DL is 2NExpTime-complete. Hence, terminological reasoning over such languages is a challenging task, especially for very large ontologies.

Highly efficient TBox reasoning algorithms such as those based on tableaux [4, 15] and hypertableaux [19] have been proposed to tackle this formidable problem. Various optimization techniques such as absorption, backtracking and blocking have been developed [13] to reduce search space, therefore speeding up the processing and reducing memory footprint. Based on these algorithms, a number of efficient ontology reasoners such as FaCT++ [26], HermiT [19], Pellet [23] and TrOWL [25] have been implemented. These reasoners can handle some very large ontologies such as GALEN, Gene Ontology and NCI Thesaurus Ontology. However, it has also been pointed out that further studies are still needed for improving terminological reasoning [9].

In a lot of situations such as in the mobile context [24], it is very valuable to obtain a (rough) estimate of reasoning performance before reasoning is actually

carried out. Although theoretical worst-case complexity has been established for these languages, such complexity is not necessarily a reliable indication of real-world, typical-case performance. Part of the reason is that different reasoners implement different algorithms and optimization techniques, hence they may have widely different performance for a same ontology. In other words, the hardness of reasoning on individual ontologies is a product of the intrinsic characteristics of the ontologies (i.e., metrics [28]) and that of the reasoner employed.

Therefore, we believe it is of both theoretical and practical importance to adequately measure, benchmark and characterize performance of different reasoners. Many existing works on (TBox) reasoner benchmarking [20, 5, 7, 9, 10] have used relatively small to medium-sized datasets, which do not provide sufficient grounds for rigorously analysis of performance characteristics. These works also only focused on comparing and benchmarking performance of different reasoners – they did not provide insights into such performance.

In this paper, we attempt to conduct a rigorous and comprehensive study that characterizes performance of the above four reasoners, for the task of ontology classification, on a set of over 300 ontologies of varying sizes and hardness. We also study the relationship of the hardness of individual ontologies and their intrinsic syntactic and structural metrics [28] by applying a machine learning approach. Our preliminary results are very encouraging, showing a high accuracy of correctly predicting (discretized) performance of all the four reasoners.

2 Background and Related Work

Tremendous progress has been made in recent years in designing and implementing highly optimised inference algorithms and reasoners. Tableau- and hypertableau-based algorithms [3, 6, 19] have dominated DL inference research and many reasoners are based on these algorithms, including FaCT++ [26], Pellet [22] and HermiT [21]. With the introduction of OWL 2 and its profiles, other approaches, including completion rule-based and consequence-based algorithms have been developed to tackle inference problems on less expressive DLs such as $\mathcal{EL}^{++}$ (the OWL 2 EL profile) and DL-Lite (the OWL 2 QL profile), for which polynomial-time algorithms exist for standard DL inference tasks such as subsumption checking [1, 8]. Reasoners including `CEL` [2], CB [16] and `Snorocket` [17] are based on this approach. TrOWL [25] is an inference infrastructure that takes a hybrid approach: it applies syntactic and semantic approximation to transform OWL 2 DL ontologies to less expressive profiles (QL and EL) for different reasoning tasks, and it uses a variety of underlying reasoners for different languages.

Quite a few works have been done on benchmarking ontology reasoners. Earlier works primarily focused on OWL 1 and DAML+OIL ontologies. Bock et al. [7] benchmarked the time performance of 5 reasoners, KAON2, OWLIM, Pellet, RacerPro and Sesame, over a dataset generated from four small ontologies by varying the number of ABox assertions. Two reasoning tasks were evaluated: classification and conjunctive query answering. Because of the size of the ontologies, the majority of reasoners achieve a subsecond response time for clas-

sification on the four ontologies. On the other hand, they exhibit a more varying behavior for conjunctive query answering. Pan [20] compared three reasoners, FaCT++, Pellet and RacerPro, on a dataset of 135 (OWL 1) ontologies for the task of classification, and commented on the relative strengths and weaknesses of the reasoners. These ontologies are relatively small too: with an average of 43.7 classes and 19.3 relations per ontology. Gardiner [10] et al. also compared four reasoners, FaCT++, KAON2, Pellet and RacerPro, on 172 (OWL 1) ontologies. Their experiments showed that different reasoners have different characteristics, but did not discuss these differences in detail.

A new benchmarking framework based on justifications has recently been proposed by Bail et al. [5]. Justifications are small minimal subsets of logical axioms and assertions sufficient for an entailment to hold. The authors argued that a justification-based, but not classification-based, benchmarking approach provides better fault isolation capabilities and is useful in reasoner development.

More recently, Dentler et al. [9] conducted a comprehensive comparative study of three dimensions of eight reasoners, CB, CEL, FaCT++, HermiT, Pellet, RacerPro, Snorocket and TrOWL, that support the OWL 2 EL profile. A number of TBox reasoning tasks are performed on three large OWL 2 EL ontologies (Gene Ontology, NCI Thesaurus and SNOMED CT) and it was observed that the reasoners exhibit a significant difference in performance, and that further research is required to better understand this phenomenon.

In the SEALS project[1], the Storage and Reasoning Systems Evaluation Campaign 2010 aimed at evaluating DL-based reasoners. In the evaluation, the performance of three reasoners FaCT++, HermiT, and jcel were measured and compared in terms of a suite of standard inference services such as classification, class/ontology satisfiability, and logical entailment. This evaluation results in a framework revealing a good performance comparison of different reasoners. However, it does not seem to tackle the problem of performance prediction. Hence, our work presented here is complementary to the SEALS project.

3 Methodology

The principal aims of this paper are (1) to benchmark the performance of reasoning tasks of a number of reasoners over a large and diverse dataset, and (2) to experimentally determine whether a combination of ontology metrics can be leveraged to effectively predict the response time for specific reasoning tasks. Thus there are four dimensions which need to be considered:

Reasoning task - For our evaluation, we focus on *classification*. Classification is the process of making all class subsumption relations explicit in an ontology and it is one of fundamental TBox reasoning tasks. Another main reasoning task, consistency checking, is not chosen because of a pragmatic reason: that different reasoners perform consistency checking at different times. It is sometimes performed together with ontology loading in some reasoners, while some

[1] `http://www.seals-project.eu`

other reasoners perform consistency checking in a separate step after loading the ontology.

Ontology features - The evaluation needs to focus on a diverse set of publicly available ontologies which have different sizes (ranging from a few KB, to several MB), vocabulary sizes, structural characteristics and most importantly, different performance characteristics.

Reasoner benchmarking - The evaluation must perform classification on a number of ontologies using different publicly available reasoners. In this work, we will compare those reasoners that are actively-maintained, open-source and are able to support expressive languages such as OWL 2 DL.

Predictive models - The supervized machine learning technique, *classification*,[2] is used in the experiments to develop a predictive model to estimate inference time from metric values. Our goal is to be able to predict the membership of an ontology within a number of categories, defined over (discretized) reasoning time. A number of *classifiers* will be investigated to achieve the most effective prediction for different reasoners, since it is well-known that different classifiers will produce results of differing accuracies for different datasets.

For our specific problems of reasoner benchmarking and predictive model construction, we therefore first need to collect reasoning runtime data and metrics data. Secondly, we then need to leverage these metrics to develop a predictive model to determine the reasoning task time given the ontology metric values (for the subset of metrics that have the capacity to determine reasoning task time) and the reasoner. There are the following key steps in our approach:

1. *Data collection.* We need to collect a number of ontologies with a variety of characteristics, which may include recency, the application domain, file size, metric values, underlying ontology language, and most importantly, reasoning time. We also need to compute, for each ontology collected, its metric values, and an average time for the task of ontology classification. The *classification* reasoning task is performed on each ontology and the average reasoning time is recorded.
   Furthermore, since our goal is to learn predictive classifiers, we also need to discretize the continuous reasoning time in order to assign ontologies into separate groups based on their reasoning time.
2. *Building the predictive model.* The third stage of our approach constructs classifiers that classify ontologies into categories based on discretized reasoning time. The classifier typically builds a predictive model in the form of a Bayesian model, a decision tree, a regression model, or a set of rules. The prediction model is then evaluated for accuracy based on the widely-used 10-fold cross-validation. In this validation practice, each dataset is partitioned into k subsets. Each time, one of the k subsets is used as testing data, and the remaining (k-1) subsets form training data. The cross-validation process

[2] Note that this is an entirely different concept than ontology classification.

is then repeated k times with each of the k subsets used exactly once as the testing data. All k results from the folds can then be used as performance statistics. We use $k = 10$ as 10 is very often used in such validation practice.

4 Experiments and Analysis

Reasoners and reasoning task. We select four widely-used, actively-maintained and open-source reasoners that support OWL 2 DL, namely FaCT++ [26], HermiT [19], Pellet [23] and TrOWL [25] for our analysis of classification time. Note that TrOWL is incomplete because of the approximation it applies. In our experiment, CEL [2] is the underlying reasoner that TrOWL uses. The other three reasoners are complete OWL 2 DL reasoners. Table 1 below provides a brief summary of these reasoners.

Table 1. A brief summary of the four reasoners benchmarked.

	FaCT++	**HermiT**	**Pellet**	**TrOWL**
Version	1.5.3	1.3.5	2.3.0	0.8
Expressivity	OWL 2 DL	OWL 2 DL	OWL 2 DL	OWL 2 DL (partial)
Reasoning algorithm	Tableaux	Hypertableaux	Tableaux	Completion rules (CEL)

Consistency checking, another TBox reasoning task, is not selected. We observe that for some reasoners, consistency checking takes very short time on average (0.29s for HermiT and 0.05s for Pellet). At the same time, there is a very large discrepancy in consistency checking time between the four reasoners (mean: 4.02s for FaCT++ and 131.7s for TrOWL). Such a difference may be attributed to the different ways the reasoners report consistency checking (with or after ontology loading). Moreover, HermiT, Pellet and TrOWL all have a relatively normal distribution of consistency checking time. On the other hand, FaCT++ has quite a skewed distribution, where a single ontology takes more than 1,020 seconds while no other ontology takes more than 15 seconds. Hence, we believe it is not a fair comparison and we cannot draw useful conclusions from it.

The dataset. 358 real-world ontologies are collected, a large proportion of which are collected from the Manchester Tones Ontology Repository and NCBO BioPortal.[3] These ontologies vary significantly in file size, ranging from less than 4KB to almost 300MB. All ontologies collected from BioPortal are large ontologies with at least 10,000 terms. The expressivity of these ontologies spans simpler languages such as OWL 2 EL and QL, through OWL DL to OWL 2 DL and OWL Full, with a large number being in OWL 2 DL. At the same time, this collection also includes some well-known hard ontologies such as DOLCE, FMA, Galen, the Gene Ontology, the NCI Thesaurus and the Cell Cycle Ontology.

[3] `http://owl.cs.manchester.ac.uk/repository/` and `http://www.bioontology.org/`

Metrics As stated previously, we are interested in studying ontology metrics [28] and their capability in predicting classification time. Based on the metrics defined in [28], we propose a set of 27 metrics that we believe can characterize the structure and complexity of a given ontology. This set of metrics are derived from asserted logical axioms in an ontology are are divided into the following four categories:

- **Ontology-level (ONT)** metrics measure the size and structural characteristics of an ontology as a whole. Four ONT metrics are defined in [28]: SOV (size of vocabulary), ENR (edge node ratio), TIP (tree impurity) and EOG (entropy of graph). We define two additional metrics: CYC, that measures the Cyclomatic complexity of the ontology graph, and RCH, that measures the ratio between the number of anonymous class expressions and the total number of class expressions.
- **Class-level (CLS)** metrics measure the characteristics of OWL classes, which are first-class citizens in an ontology. Four such metrics are defined in [28], including NOC (number of children), DIT (depth of inheritance), CID (in-degree) and COD (out-degree).
- **Anonymous class expressions (ACE)** metrics count the total occurrences of each kind of anonymous class expressions that are available in OWL 2 DL. There are altogether 9 metrics: enumeration ($ENUM$), negation (NEG), conjunction ($CONJ$), disjunction ($DISJ$), universal/existential quantification (UF/EF) and min/max/exact cardinality ($MNCAR/MXCAR/CAR$).
- **Properties (PRO)** metrics measure the total occurrences of each kind of property declarations/axioms. The 8 PRO metrics records the number of occurrences of property declarations and axioms. There are 8 metrics, one each for: object/datatype property declaration (OBP/DTP), functional (FUN), symmetric (SYM), transitive (TRN), inverse functional ($IFUN$), property equivalence (EQV) and inverse (INV).

Data collection. For each ontology, values for the 27 metrics are collected. For each ontology and each reasoner, CPU time on ontology classification (but not loading) is averaged over 10 independent runs and recorded. All the experiments are performed on a high-performance server running OS Linux 2.6.18 and Java 1.6 on an Intel (R) Xeon X7560 CPU at 2.27GHz with a maximum of 40GB allocated (to accommodate potential memory leaks) to the 4 reasoner. OWL API [12] (version 3.2.4) is used to communicate with all four reasoners. Some hard ontologies take an extremely long time to classify. Hence, we apply a 50,000-second cutoff for all the reasoners.

4.1 Reasoner Performance Characteristics

The distributions of the raw reasoning time for the four reasoners can be found in Figure 1, where reasoning time is plotted in log scale due to its wide range ($0\text{s} \leq R \leq 50{,}000\text{s}$), against ontologies sorted by their reasoning time. Note that

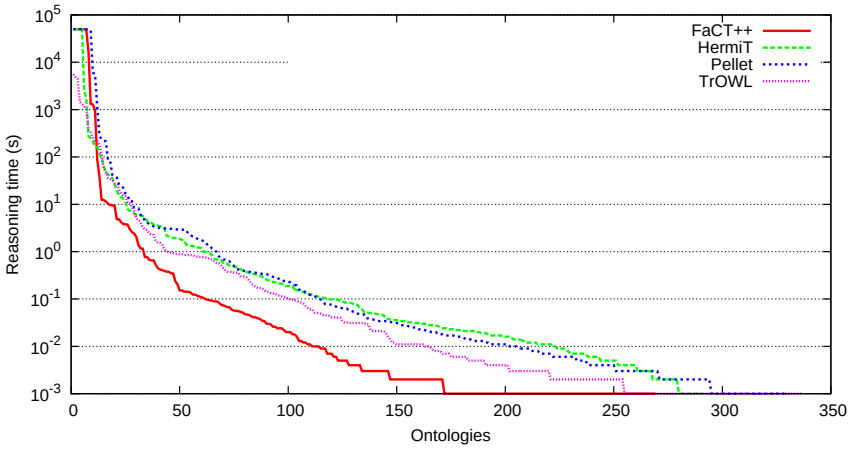


Fig. 1. Raw classification time of the four reasoners.

all reasoners except TrOWL time out (50,000 seconds) on a number of large and complex ontologies. As can be seen in the figure below, the distributions are highly skewed for all four reasoners.

Table 2 below provides some more details about the classification performance of the four reasoners, with the lowest value for each measure in **boldface** and the highest in *italic*. It can be seen in the second row that each reasoner fails to perform classification on a number of ontologies due to parsing or processing errors or the ontology being inconsistent.

It can be seen that for each reasoner, its mean is much higher than the median, indicating that the distribution is heavily skewed towards the right and that it may be the result of a small number of large values, which can be seen quite easily in Figure 1. It should also be noted that having a mean much larger than the median suggests that a distribution may be quite steep. This observation is confirmed by the high values of the skewness (Equation 1) and Kurtosis (Equation 2) measures in the table, which measure the (lack of) symmetry and the peakedness, respectively (s is the standard deviation of the sample). A skewness close to zero indicates roughly evenly distributed values. A positive skewness value indicates that the right-side tail of the distribution is longer than that of the left side, which is the case for all the four reasoners. A normal distribution has a Kurtosis measure of 0. A high Kurtosis value indicates that the data has a high peak, which is the case for all the four reasoners.

$$G_1 = \frac{n}{(n-1)(n-2)} \sum_{i=1}^{N} \left(\frac{x_i - \overline{x}}{s} \right)^2 \tag{1}$$

$$G_2 = \left(\frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{N} \left(\frac{x_i - \overline{x}}{s} \right)^4 \right) - \frac{3(n-1)^2}{(n-2)(n-3)} \tag{2}$$

It can also be seen that the reasoners exhibit quite different performance characteristics. TrOWL and Pellet successfully complete on more ontologies than FaCT++ and HermiT. FaCT++, HermiT and Pellet all time out on a number

Table 2. Summary of raw classification time of the four reasoners.

	FaCT++	**HermiT**	**Pellet**	**TrOWL**
Number of ontologies resulted in error	*89*	67	28	**21**
Number of ontologies timed out ($> 50,000$s)	6	5	*8*	**0**
Mean (s)	1,366.4	879	*1,400.3*	**65.41**
Standard deviation (s)	7,967.41	6765.28	*8,121.11*	**490.07**
Median (s)	**0.002**	*0.037*	0.02	0.007
Skewness	**3.5**	7.46	5.81	9.56
Kurtosis	**10.63**	49.68	32	95.53

of ontologies, but not TrOWL. As a result of the clipping, the true performance value distributions for the former three reasoners may be even more skewed and peaked.

The performance of the four reasoners can be further characterized below.

FaCT++ has the lowest median, its distribution is the least skewed and also the least steep (lowest skewness and Kurtosis values) among the four. From Figure 1 it can be seen that FaCT++ performs the best on a large number of ontologies. However, it also fails on the most number (89) of ontologies (not due to clipping). **HermiT** has the highest median. However, its mean is the second lowest, after TrOWL. It also fails on quite many (67) ontologies. **Pellet** times out on the most number (8) of ontologies, indicating that Pellet may have trouble handling extremely large and difficult ontologies. Moreover, it has the highest mean and standard deviation, both of which are quite close to those of FaCT++. **TrOWL** has the lowest mean and standard deviation, both of which are much lower than those of the other three reasoners. This is due in part to the fact that TrOWL does not time out on any ontology. We note that TrOWL applies syntactic and semantic approximation, and hence is incomplete. Hence, the better performance may be the result of such incompleteness and requires further analysis. It is also noteworthy to point out that TrOWL has the most skewed and the steepest distribution among the four reasoners.

4.2 Predictive Model Construction

As stated previously, being able to predict reasoning performance using ontologies metrics is highly desirable for ontology engineering and ontology-enabled applications. In this work, we use *classification* in machine learning to build predictive models that accurately estimate reasoning performance of ontology classification. This section presents a major contribution of the paper. Namely, for each reasoner, we identify an accurate predictive model for reasoning time for the classification reasoning task, and a classifier used for determining the model.

As stated in the previous section, discretization is a necessary first step before classifiers can be trained. After raw run time values are collected, trivially simple ontologies (with reasoning time ≤ 0.01s) are removed from the dataset.

Experiments on the entire dataset without removal are also performed, where trained classifiers have even higher accuracy. However, this is due to the fact that the entire dataset is much more skewed towards simple ontologies. Hence the high accuracy is not really an improvement.

Reasoning time is then discretized into 4 bins uniformly, with unit interval width. The interval width is used as exponent of the reasoning time, i.e., 10^i is the cutoff point between bin i and bin $i+1$, for $1 \leq i \leq 4$. The bins are labelled 'A', 'B', 'C' and 'D'. A summary of the discretization and the size of the dataset for each reasoner in each bin is shown in Table 3.

Table 3. Discretization of reasoning time and number of ontologies in each bin.

Discretized label	Classification time	Fact++	HermiT	Pellet	TrOWL
A	A < 1s	75	154	126	105
B	1s ≤ B <10s	16	35	38	17
C	10s ≤ C < 100s	6	12	12	13
D	100s ≤ D	11	13	16	14
Total discretized		108	214	192	149
Trivial ontologies		161	77	138	188
Ontologies in error		89	67	28	21

It is well-known that reasoning performance is affected by the intrinsic characteristics of individual ontologies and that of the reasoner applied (underlying algorithms and optimization techniques). Hence, a single classifier may not be able to accurately model classification performance for all four reasoners. Hence, we employ a number of classifiers and identify the most effective one to build a predictive model for a given reasoner.

We use classification accuracy (simply accuracy) [18] to evaluate the effective of a classifier. Classification accuracy measures the percentage of correctly classified ontologies over all ontologies, and it is often considered to be the best performance indicator for evaluating classifiers in test data. As mentioned before, 10-fold cross validation is used to evaluate the classifiers and measure their performance in accuracy.

In total, we choose ten well-known classifiers available in Weka [27], with the aim of finding the best predictive models within an extensive spectrum. These classifiers are representative of six categories of classifiers: Bayesian classifiers (BayesNet and NaïveBayes), decision tree-based classifiers (J48 and RandomForest), rule-based classifiers (DecisionTable and OneR), a Support Vector Machine algorithm (SMO), a logistic regression-based classifier (SimpleLogistic), and instance-based classifiers (kNN, $1 \leq k \leq 10$, and K*).

Fig. 2 shows the accuracy of all ten classifiers for the four reasoners, using all 27 metrics as features. A number of important observation can be made.

– Three classifiers produce the best accuracy results for the four reasoners, with RandomForest performing the best for two reasoners, HermiT and Pellet.

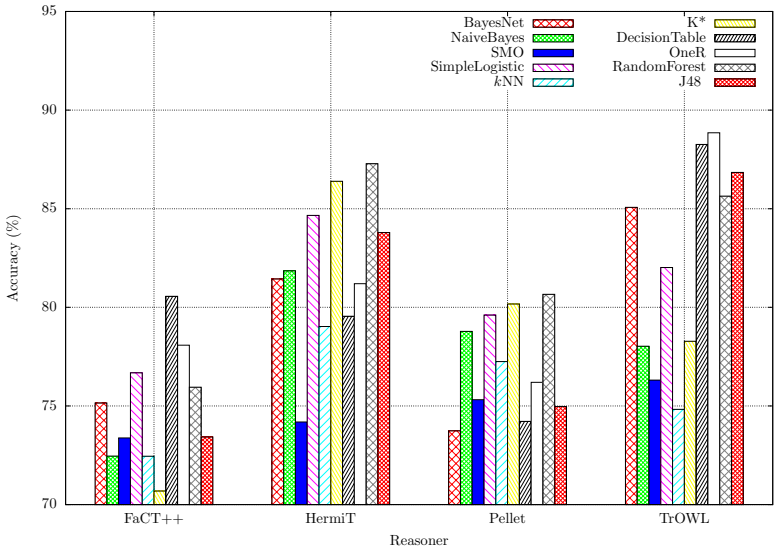


Fig. 2. Accuracy of all classifiers for the four reasoners.

- All the best classifiers for each reasoner achieve high accuracy, ranging from 80.56% for FaCT++ to 88.85% for TrOWL. Hence, it suggests that we can use such classifiers to predict classification performance with even higher accuracy.
- Overall, all classifiers produce consistently high accuracy, all higher than 70% with an average of 79.08%. This provides further evidence that (1) the predictive models found using our proposed approach can be effectively used for predicting classification performance; and (2) that ontology metrics can be used to learn predictive models for the classification task.

5 Conclusion

Our contributions in this paper are summarised are two-fold. Firstly, we study the classification performance of four widely-use, state-of-the-art OWL 2 DL reasoners, FaCT++, HermiT, Pellet and TrOWL (incomplete), comparatively. To the best of our knowledge, this is the most comprehensive of such studies in terms of the size and variability of the dataset (more than 300 ontologies with reasoning time ranging from subseconds to over 50,000 seconds). Some unique characteristics are discovered through our detailed study. Such charatceristics can be used in comparing and selecting reasoners for a given set of performance criteria.

Secondly, we further investigate the hardness of classification performance as a product of individual ontologies and reasoners. By applying machine learning techniques, we construct a model that can accurately predict performance with ontology metrics as features. Again, to the best of our knowledge, this is the

first known study to apply machine learning techniques to predicting reasoning time for inference tasks. Experimental results confirm the effectiveness of our approach as the classifiers that are learned produce high ($> 80\%$) accuracy for all the four reasoners.

Our future work will focus on further understanding the role individual metrics play in the predictive models and investigating their relative strength in predicting classification performance. We also plan to study a wider set of metrics in predicting reasoning performance. Though classification result is not the focus of this paper, we will compare that across reasoners to investigate their correctness. Moreover, we will investigate the feasibility of using metrics as a guide to generate synthetic ontologies that possess certain performance characteristics.

References

1. F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope further. In K. Clark and P. F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.
2. F. Baader, C. Lutz, and B. Suntisrivaraporn. CEL—a polynomial-time reasoner for life science ontologies. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 287–291. Springer-Verlag, 2006.
3. F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*, pages 43–95. Cambridge University Press, 2003.
4. F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69(1):5–40, 2001.
5. S. Bail, B. Parsia, and U. Sattler. JustBench: a framework for OWL benchmarking. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 32–47, Berlin, Heidelberg, 2010. Springer-Verlag.
6. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper tableaux. In J. J. Alferes, L. M. Pereira, and E. Orlowska, editors, *JELIA*, volume 1126 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.
7. J. Bock, P. Haase, Q. Ji, and R. Volz. Benchmarking OWL reasoners. In *ARea2008 - Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, June 2008.
8. D. Calvanese, G. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. Autom. Reason.*, 39:385–429, October 2007.
9. K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web Journal*, 2(2):71–87, 2011.
10. T. Gardiner, D. Tsarkov, and I. Horrocks. Framework for an automated comparison of description logic reasoners. In *Proceedings of the 5th international conference on The Semantic Web*, ISWC'06, pages 654–667, Berlin, Heidelberg, 2006. Springer-Verlag.

11. B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. OWL 2: The next step for OWL. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 6:309–322, November 2008.
12. M. Horridge and S. Bechhofer. The OWL API: A java API for working with OWL 2 ontologies. In R. Hoekstra and P. F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
13. I. Horrocks. Implementation and optimization techniques. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *Description Logic Handbook*, pages 306–346. Cambridge University Press, 2003.
14. I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.
15. I. Horrocks and U. Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 448–453, 2005.
16. Y. Kazakov. Consequence-driven reasoning for horn $\mathcal{SHIQ}$ ontologies. In C. Boutilier, editor, *IJCAI*, pages 2040–2045, 2009.
17. M. Lawley and C. Bousquet. Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In *Australasian Ontology Workshop 2010 (AOW 2010): Advances in Ontologies*, pages 45–50, Adelaide, Australia, 2010. ACS.
18. T. Mitchell. *Machine Learning*. Mcgraw-Hill International, 1997.
19. B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
20. Z. Pan. Benchmarking DL reasoners using realistic ontologies. In B. C. Grau, I. Horrocks, B. Parsia, and P. F. Patel-Schneider, editors, *OWLED*, volume 188 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
21. R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2008.
22. E. Sirin and B. Parsia. Pellet: An OWL DL Reasoner. In R. M. Volker Haaslev, editor, *Proceedings of the International Workshop on Description Logics (DL2004)*, June 2004.
23. E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.
24. L. Steller, S. Krishnaswamy, and M. M. Gaber. Enabling scalable semantic reasoning for mobile services. *Int. J. Semantic Web Inf. Syst.*, 5(2):91–116, 2009.
25. E. Thomas, J. Z. Pan, and Y. Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *ESWC (2)*, volume 6089 of *Lecture Notes in Computer Science*, pages 431–435. Springer, 2010.
26. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, pages 292–297. Springer, 2006.
27. I. H. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, San Francisco, 2000.
28. H. Zhang, Y.-F. Li, and H. B. K. Tan. Measuring Design Complexity of Semantic Web Ontologies. *Journal of Systems and Software*, 83(5):803–814, 2010.

On the Feasibility of Using OWL 2 DL Reasoners for Ontology Matching Problems

Ernesto Jiménez-Ruiz, Bernardo Cuenca Grau, and Ian Horrocks

Department of Computer Science, University of Oxford
{ernesto,berg,ian.horrocks}@cs.ox.ac.uk

Abstract. In this paper we discuss the feasibility of using OWL 2 DL reasoners to diagnose the integration of large-scale ontologies via mappings. To this end, we have extended our ontology matching system LOGMAP with complete OWL 2 DL reasoning and diagnosis capabilities. We have evaluated the new system, which we call LOGMAP-FULL, with the largest matching problems of the Ontology Alignment Evaluation Initiative, and we have compared its performance with LOGMAP, which currently relies on a highly-scalable (but incomplete) reasoner.

1 Introduction

The Ontology Alignment Evaluation Initiative(OAEI) is an international campaign for the systematic evaluation of ontology matching systems —software programs capable of finding simple correspondences (called *mappings*) between the vocabularies of a given set of input ontologies [25, 5, 6, 26]. The matching problems in OAEI are organised in several tracks, with each track involving different kinds of test ontologies [5]; for example, the ontologies in the largest tracks of OAEI 2011.5 are the Systematized Nomenclature of Medicine Clinical Terms (SNOMED CT), the Foundational Model of Anatomy (FMA), the National Cancer Institute Thesaurus (NCI), and the Adult Mouse Anatomical Dictionary (MOUSE) —all of which are semantically rich bio-medical ontologies with thousands of classes.

Ontology mappings are commonly represented as OWL subclass or equivalence axioms. Hence, the ontology $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}$ resulting from the integration of the input ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$ via the mappings $\mathcal{M}$ automatically computed by a matching system may entail axioms that do not follow from $\mathcal{O}_1$, $\mathcal{O}_2$, or $\mathcal{M}$ alone. Many such entailments correspond to logical inconsistencies caused by either erroneous mappings in $\mathcal{M}$, or by inherent disagreements between $\mathcal{O}_1$ and $\mathcal{O}_2$. Recent work has shown that even the integration of ontologies via manually curated mappings can lead to thousands of such inconsistencies [11, 12, 10].

Most matching systems do not implement any kind of reasoning technique, and hence are unable to detect and repair such inconsistencies. In recent years, however, there has been a growing interest in the development of "built-in" reasoning and diagnosis algorithms for ontology matching systems. Systems like CODI [22, 9] and LOGMAP [14, 10, 13] implement efficient techniques that substantially reduce the number of inconsistencies derived from $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}$. To achieve favourable scalability behaviour, however, the reasoning algorithms in CODI and LOGMAP are *incomplete*, and hence cannot guarantee that the output mappings will not lead to logical inconsistencies.

In this paper we evaluate the feasibility of integrating a fully-fledged OWL 2 DL reasoner in the ontology matching system LOGMAP, and hence of guaranteeing that the set of output mappings will not lead to unsatisfiable classes; we call the new system LOGMAP-FULL. Although current reasoners can easily cope with the aforementioned OAEI ontologies individually, the diagnosis of ontology mappings poses very interesting challenges for the evaluation of OWL 2 DL reasoners.

2 LOGMAP in a nutshell

LOGMAP is a highly scalable ontology matching system with "built-in" reasoning and diagnosis capabilities. The latest version of LOGMAP's algorithm [13] can be roughly divided into the stages briefly described next.

I. Computation of candidate mappings. LOGMAP efficiently computes a set of candidate mappings $\mathcal{M}$ using lexical techniques only. Those candidate mappings $\mathcal{M}_r$ involving classes that are lexically very similar are additionally identified as *reliable*.

II. Reasoning-based diagnosis of reliable mappings. The input ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$ and the reliable mappings $\mathcal{M}_r$ are encoded in Horn propositional logic. This encoding is sound (but incomplete) for checking the unsatisfiability of each class with respect to $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}_r$. LOGMAP's propositional reasoner implements the well-known Dowling-Gallier algorithm [4, 7] and extends it to record all *conflicting* mappings that may be involved in each unsatisfiability. LOGMAP then implements a greedy diagnosis algorithm that tries to delete as few such recorded mappings as possible in order to resolve the identified unsatisfiabilities.

III. Pruning non-reliable mappings. LOGMAP efficiently indexes the propositional representations of $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{M}_r$ using an interval labelling schema [1]. This type of semantic index has shown to significantly reduce the cost of computing taxonomic queries over large class hierarchies [3, 21]. LOGMAP exploits this semantic index to efficiently discard those *conflicting* candidate mappings that, if added to the reliable mappings (after diagnosis), will make some class unsatisfiable.

IV. Final diagnosis. LOGMAP uses the same technique as in Step **II** to perform diagnosis over all the remaining candidate mappings (reliable and non-reliable). The resulting set of mappings $\mathcal{M}_{out}$ is returned as output.

3 Reasoning and diagnosis in LOGMAP-FULL

As already mentioned, LOGMAP implements a sound but *incomplete* reasoning algorithm for checking class unsatisfiability. Consequently, there is no guarantee that all classes in $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}_{out}$ will be satisfiable. Furthermore, LOGMAP might fail to detect conflicting candidate mappings in Steps **II-IV**, which might lead to incorrect choices when discarding mappings.

These limitations stem from the fact that reasoning in LOGMAP is incomplete, and hence they could be addressed by integrating a complete OWL 2 DL reasoner R into LOGMAP. A straightforward possibility is to extend LOGMAP's algorithm with a final

Input: $\mathcal{O}_1$, $\mathcal{O}_2$: input ontologies. $\mathcal{M}$: set of mappings.
Output: $\mathcal{M}$: set of mappings

1: **repeat**
2: $Unsat :=$ unsatisfiable classes w.r.t. $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}$
3: $Just := \emptyset$
4: **for each** $C \in Unsat$ **do**
5: $Just := Just \cup \mathsf{SingleJustification}(C, \mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M})$
6: **end for**
7: $\mathcal{M} := \mathcal{M} \setminus \mathsf{ConflictiveMappings}(Just)$
8: **until** $Unsat \neq \emptyset$
9: **return** $\mathcal{M}$

Table 1. Diagnosis in LOGMAP-FULL.

step in which R is used to check the satisfiability of each class w.r.t. $\mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{M}_{out}$. Standard justification-based diagnosis techniques (e.g., [16, 15, 23, 8, 28]) can then exploited to fix the identified inconsistencies.[1] This approach, which guarantees a "clean" output, is adopted by systems such as CONTENTMAP [11] and ALCOMO [19]; however, the detection of conflicting mappings in Steps **II-IV** would still rely on an incomplete reasoner.

In LOGMAP-FULL we have implemented a different approach, where each call to the Dowling and Gallier algorithm in Steps **II** and **IV** and to the semantic index in Step **III** is replaced with a call to the complete reasoner R. The (obvious) technical issue with this approach is scalability, with the main scalability bottleneck being not so much in the detection of unsatisfiable classes, but rather in performing diagnosis using justification-based technologies. For example, when running LOGMAP-FULL with FMA and SNOMED as input ontologies, we obtain 3,351 unsatisfiable classes in Step **II**; computing all justifications for each unsatisfiable class required, on average, more than 9 minutes,[2] which implies that LOGMAP-FULL would need more than 3 weeks to complete Step **II** (when LOGMAP does it in under 82 seconds).

To address these scalability issues, LOGMAP-FULL implements the "greedy" diagnosis algorithm in Table 1, which avoids computing all justifications for each unsatisfiable class. The algorithm uses R to check for unsatisfiable classes (Line 2) and to compute a single justification for each unsatisfiable class (Line 5); this is feasible since computing only one justification is much easier in practice than computing all of them [16, 15]. In Line 7, the algorithm heuristically selects a set of mappings $\mathsf{ConflictiveMappings}(Just)$ containing at least one mapping per justification in $Just$. A single iteration of this process does not guarantee that all unsatisfiabilities will be resolved, so the process needs to be repeated until no more unsatisfiable classes can be found. This algorithm is quite different from the one used in LOGMAP, where the computation of justifications was not an issue (see [10] for details).

[1] Given a class A that is unsatisfiable w.r.t. an ontology $\mathcal{O}$, a justification $\mathcal{O}'_A$ is a subset of $\mathcal{O}$ such that *(i)* A is unsatisfiable w.r.t. $\mathcal{O}'_A$ and *(ii)* A is satisfiable w.r.t. each strict subset of $\mathcal{O}'_A$.
[2] Using HermiT reasoner [24, 20] and the optimisation proposed in [28].

Table 2. LOGMAP and LOGMAP-FULL diagnosis times (s)

Diagnosis of MOUSE-NCI Anatomy			
System	**Step II**	**Step III**	**Step IV**
LOGMAP	0.7	0.3	0.2
LOGMAP-FULL $_{\text{HermiT}}$	10.6	1.8	2.0
LOGMAP-FULL $_{\text{Pellet}}$	7.7	0.4	0.2
LOGMAP-FULL $_{\text{FaCT++}}$	16.4	0.6	1.9
Diagnosis of FMA-NCI			
System	**Step II**	**Step III**	**Step IV**
LOGMAP	14.6	3.4	11.6
LOGMAP-FULL $_{\text{HermiT}}$	469.7	54.3	1,550
LOGMAP-FULL $_{\text{Pellet}}$	392.5	25.8	2,787
Diagnosis of FMA-SNOMED			
System	**Step II**	**Step III**	**Step IV**
LOGMAP	81.4	21.3	87.7
LOGMAP-FULL $_{\text{HermiT}}$	2,628	479.6	11,018
LOGMAP-FULL $_{\text{Pellet}}$	21,477	1,351	$>10^5$
Diagnosis of SNOMED-NCI			
System	**Step II**	**Step III**	**Step IV**
LOGMAP	182.9	142.7	237

4 Evaluation

We have tested LOGMAP and LOGMAP-FULL with the largest ontologies of the OAEI 2011.5 campaign: SNOMED CT (306,591 classes), NCI (66,724 classes), FMA (78,989 classes), MOUSE (2,744 classes), and the anatomy fragment of NCI (3,304 classes). For the experiments we have used a high performance server with 15 Gb of RAM. Table 2 summarises the computation times for the Steps **II**-**IV** in LOGMAP and LOGMAP-FULL. LOGMAP-FULL has been tested with three well-known OWL 2 DL reasoners:[3] Pellet [27], FaCT++ [29] and HermiT [24, 20].

LOGMAP-FULL was able to efficiently handle MOUSE and NCI Anatomy for each of the three tested reasoners and reported times in line with LOGMAP. LOGMAP-FULL, however, did not terminate when given SNOMED and NCI as input for any of the evaluated reasoners. Furthermore, FaCT++ could not process any input involving FMA, and hence failed to produce an output for FMA-NCI and FMA-SNOMED.

When using HermiT and Pellet, LOGMAP-FULL did successfully compute output mappings for FMA-NCI and the computation times, although much higher than those reported by LOGMAP, were in line with many of the tools participating in the OAEI 2011.5 campaign.[4] Note that many of these matching tools do not perform any kind of reasoning, and hence LOGMAP-FULL's computation times are surprisingly good. Times for FMA-SNOMED, however, increased dramatically (especially when using Pel-

[3] LOGMAP-FULL was not tested with the OWL 2 EL reasoner ELK [17, 18] because it does not implement yet the axiom pinpointing service.

[4] `http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/index.html`

let, where Step **II** required almost 6 hours and Step **IV** did not finish after 4 days); these results are in contrast to the low computation times obtained by LOGMAP.

Finally, it is worth mentioning that LOGMAP output mappings only led to two unsatisfiable classes for the FMA-NCI case.[5] As expected, LOGMAP-FULL produced a clean output for all cases it could successfully process.

5 Conclusions

In this paper we have evaluated the feasibility of using full OWL 2 DL reasoning capabilities for "on-the-fly" mapping diagnosis. For this purpose, we have developed LOGMAP-FULL as an extension of our ontology matching systems LOGMAP.

Our empirical results suggest that the use of LOGMAP-FULL is feasible for medium-sized ontologies such as MOUSE and NCI Anatomy. For larger and semantically richer ontologies, however, computation times increase considerably; thus, LOGMAP seems to be a better choice than LOGMAP-FULL for applications with strict scalability demands (i.e., applications where user intervention is required to obtain high precision mappings); note, however, that LOGMAP-FULL's computation times are still competitive with (and in many cases faster than) most existing matching tools.

Finally, we have shown that reasoning with the integration of large-scale ontologies via mappings still poses serious problems to current OWL 2 DL reasoners. Hence, these integrated ontologies seem ideal as reasoning benchmarks.

Acknowledgements

This work was supported by the Royal Society, the EU FP7 project SEALS and by the EPSRC projects ConDOR, ExODA and LogMap. We also thank the organisers of the SEALS and OAEI evaluation campaigns for providing test data and infrastructure.

References

1. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. SIGMOD Rec. 18, 253–262 (1989)
2. Armas Romero, A., Cuenca Grau, B., Horrocks, I.: Modular combination of reasoners for ontology classification. In: Proc. of the 25th Description Logics workshop (2012)
3. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On labeling schemes for the Semantic Web. In: Proc. of WWW. pp. 544–555. ACM (2003)
4. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. pp. 267–284 (1984)
5. Euzenat, J., Meilicke, C., Stuckenschmidt, H., Shvaiko, P., Trojahn, C.: Ontology Alignment Evaluation Initiative: six years of experience. J Data Semantics (2011)

[5] To the best of our knowledge, no OWL 2 DL reasoner can cope with SNOMED-NCI (not even with the optimization proposed in [2]) and thus, we were not able to check if LOGMAP's output was 'clean" for this case. The OWL 2 EL reasoner ELK could classify SNOMED-NCI and reported no unsatisfiable classes; the version of NCI we are using, however, is not in OWL 2 EL and hence ELK might be incomplete.

6. Euzenat, J., Ferrara, A., van Hage, W.R., Hollink, L., Meilicke, C., Nikolov, A., Ritze, D., Scharffe, F., Shvaiko, P., Stuckenschmidt, H., Sváb-Zamazal, O., Trojahn dos Santos, C.: Results of the Ontology Alignment Evaluation Initiative 2011. 6th OM workshop (2011)
7. Gallo, G., Urbani, G.: Algorithms for testing the satisfiability of propositional formulae. The Journal of Logic Programming 7(1), 45 – 61 (1989)
8. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In: Proc. of International Semantic Web Conference. pp. 323–338 (2008)
9. Huber, J., Sztyler, T., Nößner, J., Meilicke, C.: CODI: Combinatorial optimization for data integration: results for OAEI 2011. In: Proc. of the 6th OM Workshop (2011)
10. Jiménez-Ruiz, E., Cuenca Grau, B.: LogMap: Logic-based and Scalable Ontology Matching. In: Proc. of the 10th International Semantic Web Conference (ISWC). pp. 273–288 (2011)
11. Jiménez-Ruiz, E., Cuenca Grau, B., Horrocks, I., Berlanga, R.: Ontology integration using mappings: Towards getting the right logical consequences. In: Proc. of ESWC (2009)
12. Jiménez-Ruiz, E., Cuenca Grau, B., Horrocks, I., Berlanga, R.: Logic-based assessment of the compatibility of UMLS ontology sources. J Biomed. Sem. 2 (2011)
13. Jiménez-Ruiz, E., Cuenca Grau, B., Zhou, Y., Horrocks, I.: Large-scale interactive ontology matching: Algorithms and implementation. In: Proc. of ECAI (2012)
14. Jiménez-Ruiz, E., Morant, A., Cuenca Grau, B.: LogMap results for OAEI 2011. In: Proc. of the 6th OM Workshop (2011)
15. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In: ISWC 2007. pp. 267–280 (2007)
16. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.A.: Debugging unsatisfiable classes in OWL ontologies. J. Web Sem. 3(4), 268–293 (2005)
17. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th International Semantic Web Conference (ISWC). pp. 305–320 (2011)
18. Kazakov, Y., Krötzsch, M., Simancik, F.: ELK reasoner: Architecture and evaluation. In: Proceedings of the 1st International OWL Reasoner Evaluation Workshop (ORE) (2012)
19. Meilicke, C.: Alignment Incoherence in Ontology Matching. Ph.D. thesis, University of Mannheim, Chair of Artificial Intelligence (2011)
20. Motik, B., Shearer, R., Horrocks, I.: Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)
21. Nebot, V., Berlanga, R.: Efficient retrieval of ontology fragments using an interval labeling scheme. Inf. Sci. 179(24), 4151–4173 (2009)
22. Niepert, M., Meilicke, C., Stuckenschmidt, H.: A probabilistic-logical framework for ontology matching. In: Proc. of AAAI (2010)
23. Schlobach, S., Huang, Z., Cornet, R., van Harmelen, F.: Debugging incoherent terminologies. J. Autom. Reasoning 39(3), 317–349 (2007)
24. Shearer, R., Motik, B., Horrocks, I.: HermiT: A highly-efficient OWL reasoner. In: Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions (2008)
25. Shvaiko, P., Euzenat, J.: Ten challenges for ontology matching. In: On the Move to Meaningful Internet Systems (OTM Conferences) (2008)
26. Shvaiko, P., Euzenat, J.: Ontology matching: State of the art and future challenges. IEEE Trans. Knowl. Data Eng. 99 (2011)
27. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Sem. 5(2), 51–53 (2007)
28. Suntisrivaraporn, B., Qi, G., Ji, Q., Haase, P.: A modularization-based approach to finding all justifications for OWL DL entailments. In: 3rd Asian Semantic Web Conference (2008)
29. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Third International Joint Conference on Automated Reasoning, IJCAR. pp. 292–297 (2006)

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/

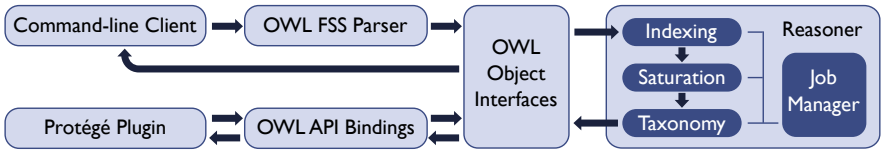


Fig. 1. Main software modules of ELK and information flow during classification

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)

$$A \sqsubseteq \exists R.(\overline{B \sqcap C}) \qquad \text{by } \mathbf{R}^-_{\exists} \text{ to (5)} \qquad (7)$$

$$\overline{B \sqcap C} \sqsubseteq B \sqcap C \qquad \text{by } \mathbf{R_0} \text{ to (6)} \qquad (8)$$

$$\overline{B \sqcap C} \sqsubseteq B \qquad \text{by } \mathbf{R}^-_{\sqcap} \text{ to (8)} \qquad (9)$$

$$\overline{B \sqcap C} \sqsubseteq C \qquad \text{by } \mathbf{R}^-_{\sqcap} \text{ to (8)} \qquad (10)$$

$$\overline{A} \sqsubseteq \exists R.B \qquad \text{by } \mathbf{R}^+_{\exists} \text{ to (7) and (9)} \qquad (11)$$

$$\overline{A} \sqsubseteq A \sqcap \exists R.B \qquad \text{by } \mathbf{R}^+_{\sqcap} \text{ to (4) and (11)} \qquad (12)$$

$$\mathsf{init}(B) \qquad \text{by } \mathbf{R}^-_{\exists} \text{ to (11)} \qquad (13)$$

$$A \sqsubseteq \exists R.\overline{B} \qquad \text{by } \mathbf{R}^-_{\exists} \text{ to (11)} \qquad (14)$$

$$\overline{A} \sqsubseteq C \qquad \text{by } \mathbf{R}_{\sqsubseteq} \text{ to (12) using (2)} \qquad (15)$$

$$\overline{B} \sqsubseteq B \qquad \text{by } \mathbf{R_0} \text{ to (13)} \qquad (16)$$

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 $\mathbf{R}^+_{\exists}$ and $\mathbf{R}^+_{\sqcap}$ 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 $\mathbf{R}_{\sqsubseteq}$ 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.

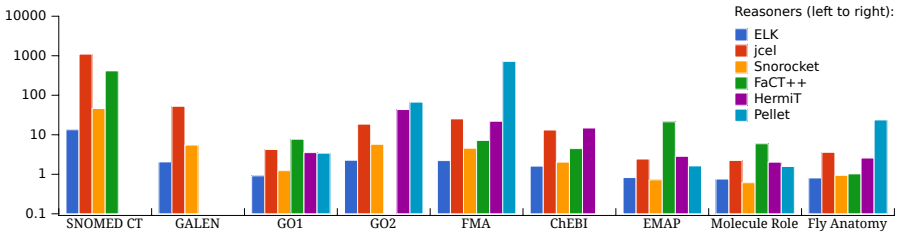
The results show that, on all tested ontologies, ELK and Snorocket far outperform all the remaining reasoners. On the smaller ontologies (GO1, ChEBI, EMAP, Molecule Role, and Fly Anatomy), ELK and Snorocket show similar performance, while on the larger ontologies (SNOMED CT, GALEN, GO2, and FMA) ELK 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 $\mathbf{R}_{\exists}^{+}$; 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 $\mathbf{R}_{\exists}^{-}$ 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., Whetzel, 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)

Evaluating Reasoners Under Realistic Semantic Web Conditions

Yingjie Li, Yang Yu and Jeff Heflin

Department of Computer Science and Engineering, Lehigh University
19 Memorial Dr. West, Bethlehem, PA 18015, U.S.A.
{yil308, yay208, heflin}@cse.lehigh.edu

Abstract. Evaluating the performance of OWL Reasoners on ontologies is an ongoing challenge. LUBM and UOBM are benchmarks to evaluate Reasoners by using a single ontology. They cannot effectively evaluate systems intended for multi-ontology applications with ontology mappings, nor can they evaluate OWL 2 applications and generate data approximating realistic Semantic Web conditions. In this paper we extend our ongoing work on creating a benchmark that can generate user-customized ontologies together with related mappings and data sources. In particular, we have collected statistics from real world ontologies and used these to parameterize the benchmark to produce more realistic synthetic ontologies under controlled conditions. The benchmark supports both OWL and OWL 2 and applies a data-driven query generation algorithm that can generate diverse queries with at least one answer. We present the results of initial experiments using Pellet, HermiT, OWLIM and DLDB3. Then, we show the approximation of our synthetic data set to real semantic data.

Keywords: Benchmark, Ontology generation, Query generation

1 Introduction

Various semantic applications based on ontologies have been developed in recent years. They differ in the ontology expressivity such as RDF, OWL, OWL 2 or some fragment of these languages or the number of ontologies such as single-ontology systems or multi-ontology federated systems. One of the major obstacles for these system developers is that they cannot easily find a real world experimental data set to evaluate their systems in terms of the ontology expressivity, the number of ontologies and data sources, the ontology mappings and so on. In order to bridge this gap, LUBM [5] and UOBM [9] were developed to evaluate Semantic Web knowledge base systems (KBSs) by using a single domain ontology. But they cannot effectively evaluate systems intended for multi-ontology applications with ontology mappings, nor can they evaluate OWL 2 applications and generate data approximating realistic Semantic Web conditions. To solve these problems, we extend our early work [8] on creating a benchmark that can generate user-customized ontologies together with related

mappings and data sources. In particular, we have collected statistics from real world ontologies and data sources and used these to parameterize the benchmark to produce realistic synthetic ontologies under controlled conditions. Unlike our previous work, this benchmark allows us to speculate about different visions of the future Semantic Web and examine how current systems will perform in these contrasting scenarios. Although Linking Open Data and Billion Triple Challenge data is frequently used to test scalable systems on real data, these sources typically have weak ontologies and little ontology integration (i.e., few OWL and OWL 2 axioms). The benchmark can be also used to speculate about similar sized (or larger) scenarios where there are more expressive ontologies and richer mappings.

Based on our previous work [8], in this paper, we first extend the two-level customization model including the web profile (to customize the distribution of different types of desired ontologies) and the ontology profile (to customize the relative frequency of various ontology constructors in each desired ontology) to support any sublanguage of both OWL and OWL 2 by taking OWL 2 constructors as our constructor seeds instead of OWL constructors. Then, we demonstrate that our benchmark can be used to evaluate OWL/OWL 2 reasoners such as Pellet [12], HermiT [10], OWLIM [7] and DLDB3 [11]. Finally, we show how well our synthetic data approximates real semantic data, specifically using the Semantic Web Dog Food corpus.

The remainder of the paper is organized as follows: Section 2 reviews the related work. Section 3 describes the benchmark algorithms. Section 4 presents the experiments. Finally, in Section 5, we conclude and discuss future work.

2 Related Work

The LUBM [5] is an example of a benchmark for Semantic Web knowledge base systems with respect to large OWL applications. It makes use of a university domain workload for evaluating systems with different reasoning capabilities and storage mechanisms. L. Ma et al. [9] extended the LUBM to make another benchmark - UOBM so that OWL Lite and OWL DL can be supported. However, both of them use a single domain/ontology and did not consider the ontology mapping requirements that are used to integrate distributed domain ontologies in the real Semantic Web. In addition, they do not allow users to customize requirements for their individual evaluation purposes. S. Bail et al. proposed a framework for OWL benchmarking called JustBench [1], which presents an approach to analyzing the behavior of reasoners by focusing on justifications of entailments through selecting minimal entailing subsets of an ontology. However, JustBench only focuses on the ontology TBoxes and does not consider the ABoxes (data sources) and the TBox mappings. C. Bizer et al. proposed a Berlin SPARQL Benchmark (BSBM) for comparing the SPARQL query performance of native RDF stores with the performance of SPARQL-to-SQL rewriters [2]. This benchmark aims to assist application developers in choosing the right architecture and the right storage system for their requirements. However, the BSBM can only

output benchmark data in an RDF representation and a purely relational representation and does not support users' customizations on OWL and OWL 2 for different applications. I. Horrocks and P. Schneider [6] proposed a benchmark suite comprising four kinds of tests: concept satisfiability tests, artificial TBox classification tests, realistic TBox classification tests and synthetic ABox tests. However, this approach neither creates OWL ontologies and SPARQL queries nor ontology mappings, and only focuses on a single ontology at a time. Also, it did not consider users' customizability requirements.

3 The Benchmark Algorithms

In this section, we first introduce our extended two-level customization model consisting of a web profile and several ontology profiles for users to customize ontologies, then briefly describe the axiom construction and data source generation, and finally give an introduction to the data-driven query generation algorithm.

3.1 The Extended Two-level Customization Model

In order to support both OWL and OWL 2, our extended two-level customization model chooses the set of OWL 2 DL constructors as our constructor seeds and is designed to be flexible in expressivity by allowing users to customize these constructors in range of OWL 2 DL. Similar to our previous work [8], we still use ontology profiles to allow users to customize the relative frequency of various ontology constructors in the generated ontologies and web profile to allow users to customize the distribution of different types of ontologies. However, in this paper, besides user customized ontology profile, our extended benchmark can also automatically collect statistics of Table 1 listed types of axioms from real world ontologies and use them to parmeterize our ontology profile in order to generate realistic synthetic ontologies.

Compared to OWL, OWL 2 offers new constructors for expressing additional restrictions on properties such as property self restriction, new characteristics of properties such as data property cardinality, property chains such as property chain inclusions and keys such as property key. OWL 2 also provides new data type capabilities such as data intersection, data union, etc. In order to support these new additions in OWL 2, we categorized all OWL 2 DL constructors into five groups: axiom types, class constructors, object property constructors, data type property constructors and data type constructors. Since the data type property constructors contain one and only one constructor ($DatatypeProperty$), in each ontology profile, we let users fill in four tables with their individual configurations: the axiom type (AT) table, the class table (CT), the object property constructor table (OPT) and the datatype constructor table (DTT). The new constructor table is shown in Table 1. Compared to the old one in [8], the new table extends AT with constructors of $disjointUnionOf$, $ReflexiveProperty$, etc. and CT with constructors of $dataAllValuesFromRestriction$, $dataSomeValues$

$FromRestriction$, etc. The new DTT contains data constructors such as $dataComplementOf$, $dataUnionOf$, etc. As a result, Table 1 contains eleven types of operands in total: class type (C), named class type (NC), object property type (OP), datatype property type (DP), instance type (I), named object property (NOP), named datatype property (NDP), which is not listed in the table because it is only for $DatatypeProperty$, facet type (F), data type (D), a literal (L) and an integer number (INT). The C means the operand is either an atomic named class or a complex sub-tree that has a class constructor as its root. The NC means the operand is a named class. The OP, DP means the operand can be one of constructors listed in the table of object property and a datatype property, respectively. The NOP, NDP means the operand is not a complex constructor but a named object property or a named datatype property respectively. The I means the operand can be a single instance. The F is the facet type borrowed from XML Schema Datatypes. The D is the data type. The L is a literal. The INT stands for an integer number for the cardinality restriction. In these types, NC, NOP, NDP, F, I, L and INT are leaf node types.

In Table 1, {x} stands for a set of instances, whose cardinality is set by a uniform distribution. For cardinality constructors such as $minCardinality$, $maxCardinality$, $Cardinality$, $minQualifiedCardinality$, $maxQualifiedCardinality$, $qualifiedCardinality$, since the involved integer value should be positive and 1 is the most common value in the real world, we apply the Gaussian distribution with mean being 1, standard deviation being 0.5 (based on our experiences) and each generated value required to be greater than or equal to 1.

Table 1. Axiom type constructors, class constructors and property constructors.

Axiom Type Constructor					Class Constructor				
Constructors	**DL Syntax**	**Op1**	**Op2**	**Op3**	**Constructors**	**DL Syntax**	**Op1**	**Op2**	**Op3**
rdfs:subClassOf	C1 $\sqsubseteq$ C2	C	C		allValuesFrom	$\forall$P.C	OP	C	
rdfs:subPropertyOf	P1 $\sqsubseteq$ P2	OP	OP		someValuesFrom	$\exists$P.C	OP	C	
equivalentClass	C1 $\equiv$ C2	C	C		intersectionOf	C1 $\sqcap$ C2	C	C	
equivalentProperty	P1 $\equiv$ P2	OP	OP		one of	{x1,...,x2}	{I}		
disjointWith	C1 $\sqsubseteq \neg$C2	C	C		unionOf	C1 $\sqcup$ C2	C	C	
TransitiveProperty	$P^+ \sqsubseteq$ P	NOP			complementOf	$\neg$C	C		
SymmetricProperty	P$\equiv(P^-)$	NOP			minCardinality	$\geq$ nP	OP	INT	
FunctionalProperty	T $\sqsubseteq \leq$1P^+	NOP			maxCardinality	$\leq$ nP	OP	INT	
InverseFunctionalP.	T $\sqsubseteq \leq$1P	NOP			Cardinality	= nP	OP	INT	
rdfs:domain	$\geq$1P $\sqsubseteq$C	NOP,DP	NC		hasValue	$\exists$ P.{x}	OP	I	
rdfs:range	T $\sqsubseteq \forall$U.D	NOP,DP	NC,D		namedClass				
disjointUnionOf		C	{C}		dataAllValuesFromR.		DP	D	
ReflexiveProperty		NOP			dataSomeValuesFromR.		DP	D	
IrreflexiveProperty		NOP			minQualifiedCardinality		OP,DP	INT	C,D
AsymmetricProperty		NOP			maxQualifiedCardinality		OP,DP	INT	C,D
propertyDisjointWith		OP,DP	OP,DP		qualifiedCardinality		OP,DP	INT	C,D
propertyChainAxiom		NOP	NOP	NOP	dataHasValue		DP	L	
hasKey		C	{C}		hasSelf		OP	TRUE	
Object Property Constructor					Datatype Constructor				
inverseOf	P^-	OP			dataComplementOf		D		
namedProperty					dataUnionOf		D	D	
					xsdDatatype				
					namedDatatype		F		
					dataIntersectionOf		D	D	
					dataOneOf		L		

A sample input of the new ontology profiles and web profile is shown in Fig.1. In this sample input, the web profile contains eight ontology profiles: RDFS, OWL Lite, OWL DL, Description Horn Logic (DHL), OWL 2 EL, OWL

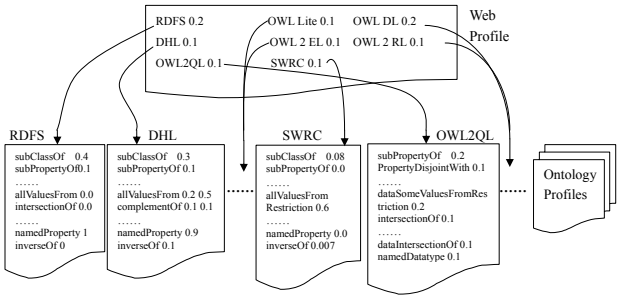


Fig. 1. Two-level customization model.

2 RL, OWL 2 QL and SWRC (Semantic Web Dog Food). Their distribution probabilities are set to be 0.2, 0.1, 0.2, 0.1, 0.1, 0.1, 0.1 and 0.1 respectively. This configuration means that in our final generated ontologies, 20% ontologies use RDFS, 10% ontologies use OWL Lite, 20% ontologies use OWL DL, 10% use DHL, 10% use each of the three OWL 2 profiles and 10% use SWRC ontology. For each ontology profile, the distributions of different ontology constructors used in the generated ontology are displayed. Each cell in the input tables is a number between 0 and 1 inclusive, which means the percentage of this constructor appearing in a generated ontology. Note, the SWRC profile is learned from the statistics of the real SWRC ontology instead of user customization. Also, in ontology languages such as DHL, an axiom has different restrictions on its left hand side (LHS) and right hand side (RHS). To support this, users can specify two probabilities for a constructor, as shown in the DHL profile of Fig. 1.

3.2 Axiom Construction and Data Source Generation

Since each ontological axiom can be seen as a parse tree with the elements in the axiom as tree nodes, the table AT actually contains those elements that can be used as the root. The tables CT, OPT and DTT provide constructors that can be used to create class, object property and datatype property expressions respectively as non-root nodes. From this perspective, the axiom construction can be seen a parse tree construction. During this process, the web profile is first used to select the configured ontology profile(s). Second, we use the distribution in the selected ontology profile to randomly select one constructor from AT table as the root node. Then, according to operand type of the selected root constructor, we use the CT, OPT or DTT tables to randomly select one class, one object property or one data type constructor to generate our ontological axioms. This process is repeated until either each branch of the tree ends with a leaf node type or the depth of the parse tree exceeds the given depth threshold. For ontology mappings, since each mapping is essentially an axiom, we apply the same procedure. Thus, the mapping ontologies have the same expressivity as the domain ontologies. Besides, we also need to consider the linking strategy of different ontologies, which is described in detail in our previous work [3].

For every domain ontology, we generate a specified number of data sources. In our configuration, this number can be set by the benchmark users according to their individual needs. For every source, a particular number of classes and properties are used for creating triples. They can be also controlled by specifying the relevant parameters in our configuration. To determine how many triples each source should have, we collected statistics from 200 randomly selected real-world Semantic Web documents. Since we found that the average number of triples in each result document is around 54.0 with a standard deviation of 163.9, we set the average number of triples in a generated source to be 50 by using a Gaussian distribution with mean 50 and standard deviation 165. In addition, based on our statistics of the ratio between the number of different URIs and the number of data sources in the Hawkeye knowledge base [4], we set the total number of different URIs in the synthetic data set to equal to the number of data sources times a factor around 2 in order to avoid the instance saturation during the source generation. In order to make the synthetic data set much closer to real world data, we ensure that each source is a connected graph, which more accurately reflects most real-world RDF files. To achieve this point, in our implementation, those instances that have already been used in the current source are chosen to generate new triples with higher priority.

In our benchmark, we also generate some *owl*:*sameAs* triples. Based on our Sindice statistics of randomly issuing one term query and took the top 1000 returned sources as samples, we found 27.1% of them contain *owl*:*sameAs* statements. Thus, our benchmark generates *owl*:*sameAs* triples in a ratio of 27.1% of the total number of triples. Furthermore, for each instance involved into the *owl*:*sameAs* triple, according to our experiences, we take a probability of 0.1 to select it from the set of all generated instances in the whole data set and a probability of 0.9 to select it from the set of all generated instances in the current source. As a result, all of *owl*:*sameAs* triples in our data set are categorized into different equivalence classes. Each equivalence class is defined to be a set of instances that are equivalent to each other (explicitly or implicitly connected by *owl*:*sameAs*). The average cardinality of the equivalence class is around 3.7.

3.3 Data-driven Query Generation Algorithm

It is well-known that the RDF data format is by its very nature a graph. Therefore, a given semantic web knowledge base (KB) can be essentially modeled as one big possibly disconnected graph. On the other hand, each SPARQL query is basically a subgraph and in order to guarantee each query has at least one answer, our SPARQL queries can be generated from the subgraphs over the big KB graph. Therefore, we proposed a data-driven query generation algorithm in our work [8]. Here, we only summarize this algorithm in order to make the paper complete.

According to the algorithm, we first identify a subgraph meeting the initial query configuration from the big KB graph. Within the identified subgraph, we randomly select one node as the starting node to construct a query pattern graph (QPG). Begin with the starting node, we randomly select one edge that

is starting with the starting node and not contained in QPG and then add this edge into QPG. If the selected edge is already in QPG, we need to select another edge that is not selected before. Then, we replace the ending node of the newly added edge with a new variable in the probability P. Currently, the default value of P is set 0.5. This process is iterated until the QPG qualifies the initial query configuration. By this step, we have successfully constructed one QPG. Then, we need to check if each edge in QPG contains at least one variable. If not, we randomly replace one node of the edge without variable nodes with a new variable. Based on the variable-assigned QPG, a SPARQL query can be generated and returned. Note, if the junction node of QPG is replaced by a query variable, this variable is counted as a join variable. For more details, please read our paper [8].

4 Evaluation

In order to demonstrate how our benchmark can be used to evaluate very different semantic reasoners, in this section, we describe two group of experiments. The first is to use our benchmark to evaluate four representative semantic reasoners: Pellet [12], HermiT [10], OWLIM [7] and DLDB3 [11]. The second is to show the approximation of our synthetic data set to real semantic data under the control of collected statistics from real world ontologies. For each experiment in each group, we issued 100 random queries, which are grouped by the number of QTPs that ranges from one to ten. Each group has ten queries. Each query has at most twenty query variables. Each QTP of each query satisfies the join condition with at least one sibling QTP. We denote an experimental configuration as follows: (nO-nD-Ont), where nO is the number of ontologies, nD is the number of data sources and Ont is the ontology profile type. In order to eliminate the outlier results, we applied the probabilistic statement of Chebyshev's inequality: $Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$, where X stands for a random variable, μ stands for the mean, σ stands for the standard deviation and $k = 3$, which counts at most 10% of each group of metric values as outliers. We applied this inequation to all metrics and for each system, any value that did not satisfy the inequation would be thrown out. The reason is that these outliers will greatly distort our experimental statistics. All our experiments are done on a workstation with a Xeon 2.93G CPU and 6G memory running UNIX.

4.1 Reasoner Evaluation

In this experiment, we want to evaluate our benchmark in various OWL and OWL 2 reasoners. In selecting the reasoners, first we decided to consider only noncommercial systems or free versions of those commercial ones. Moreover, we did not intend to carry out a comprehensive evaluation of all existing semantic reasoners and our selected ones should cover OWL and OWL 2. In addition, we also believe a practical semantic reasoner must be able to read OWL/OWL 2 files and provide programming APIs for loading data and issuing queries. As a result,

we have settled on four different reasoners including Pellet 2.2.2, HermiT 1.3.4, SwiftOWLIM and DLDB3. Except DLDB3, all candidate systems are from the W3C OWL 2 implementation system website [1]. Other candidate systems such as FaCT++, CEL, ELLY, QuOnto and Quil, we rejected due to difficulties in obtaining functions executable for the Unix platform. Our experiments are grouped by the three W3C recommended OWL 2 profiles: OWL 2 EL, RL and QL because we wanted to investigate the physical (as opposed to theoretical) consequences of these profiles. We computed the query response time, the source loading time and the query completeness respectively for each test system. Since Pellet is complete for all OWL 2 profiles and able to complete all our experiments, we chose Pellet results as our completeness ground truth. The query completeness is defined to be $\frac{\text{\# of answers returned by each test system for all tested queries}}{\text{\# of answers returned by Pellet for all tested queries}}$. All experimental results are shown in Fig.2. Note, in Fig.2 (b), (d) and (f), the HermiT curve is hiding behind the Pellet because their performances are very close.

OWL 2 EL OWL 2 EL is intended for applications that have ontologies that contain very large numbers of properties and/or classes. It captures the expressive power used by many such ontologies. Therefore, in this experiment, we evaluate the target systems by varying the number of ontologies but keeping the data sources constant at 500. Fig.2(a) and Fig.2(b) show how each system's query response time and loading time are affected by increasing the number of ontologies in OWL 2 EL. From these results, we can see that OWLIM performs best in both query response time and loading time. DLDB3 suffers from the worst loading time because it uses a persistent database backend, while the other three systems are in-memory. Pellet has better query response time than Hermit but performs very close to HermiT in loading time. Pellet and HermiT are complete, but DLDB3 and OWLIM are incomplete with 21.97% and 40.5% completeness on average respectively. The reason is that DLDB3 is only a limited OWL reasoner and does not support OWL 2, while OWLIM is only complete for OWL 2 RL and QL and incomplete for OWL 2 EL. Of the four systems, Pellet appears to be the best choice for EL. Although OWLIM is the fastest, it is significantly lacking in completeness.

OWL 2 QL OWL 2 QL focuses on applications that use very large volumes of instance data, and where query answering is the most important reasoning task. In this experiment, we evaluate the target systems by varying the number of data sources with the constant number of 5 ontologies. Fig.2(c) and Fig.2(d) show how each system's query response time and loading time are affected by increasing the number of data sources in OWL 2 QL. In both query response time and loading time, OWLIM performs best, while HermiT is worst. In particular, HermiT cannot scale to points of 5-5000-QL and 5-10000-QL because of an out of memory error but OWLIM, Pellet and DLDB3 can. Starting from point of 5-5000-QL, the loading time of Pellet performs worse than DLDB3 even though

[1] http://www.w3.org/2007/OWL/wiki/Implementations

DLDB3 uses secondary storage. We think the reason is that Pellet is in-memory and when the number of loaded data sources increases to some number (5000 in our experiment), it requires more memory to do the consistency checking during its loading period than the memory we have provided (6GB). OWLIM, Pellet and HermiT (in the first three points) are complete. DLDB3 is incomplete with 68.81% completeness on average, but it is better than it did on OWL 2 EL. In summary, of the four systems, OWLIM appears to be the best choice for OWL 2 QL. DLDB3 is an alternative for large scales where some incompleteness is tolerable.

OWL 2 RL OWL 2 RL is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate OWL 2 applications that can trade the full expressivity of the language for efficiency. Therefore, in this experiment, we evaluate the target systems by varying both the number of ontologies and the number of data sources. Fig.2(e) and Fig.2(f) show how each system's query response time and loading time are affected by increasing the number of ontologies and the number of data sources. As shown by the results, OWLIM still performs best in the query response time and loading time. HermiT suffers from the worst performance and cannot scale to points of 10-2000-*RL* and 15-3000-*RL*. As was the case to OWL 2 QL, Pellet starts to have worse loading time than DLDB3 from the point of 10-2000-*RL* because it requires more memory. OWLIM, Pellet and HermiT are complete, but DLDB3 is still incomplete with 44.96% completeness on average. In summary, OWLIM appears to be the clear winner for OWL 2 RL.

4.2 Approximation Evaluation

Constructors	**Percentage**	**Constructors**	**Percentage**
rdfs:subClassOf	8.13%	complementOf	8.98%
TransitiveProperty	0.17%	intersectionOf	8.98%
rdfs:domain	1.93%	unionOf	8.98%
rdfs:range	1.85%	namedClass	1.51%
oneOf	1.09%	allValuesFromRestriction	57.71%
inverseOf	0.67%		

Table 2. SWRC ontology constructor statistics.

In this experiment, we evaluate how approximate our synthetic data set is to real semantic data. We have chosen Semantic Web Dog Food (SWRC) corpus [2] as our real semantic data set. Since the downloaded SWRC data is in

[2] http://data.semanticweb.org/dumps/

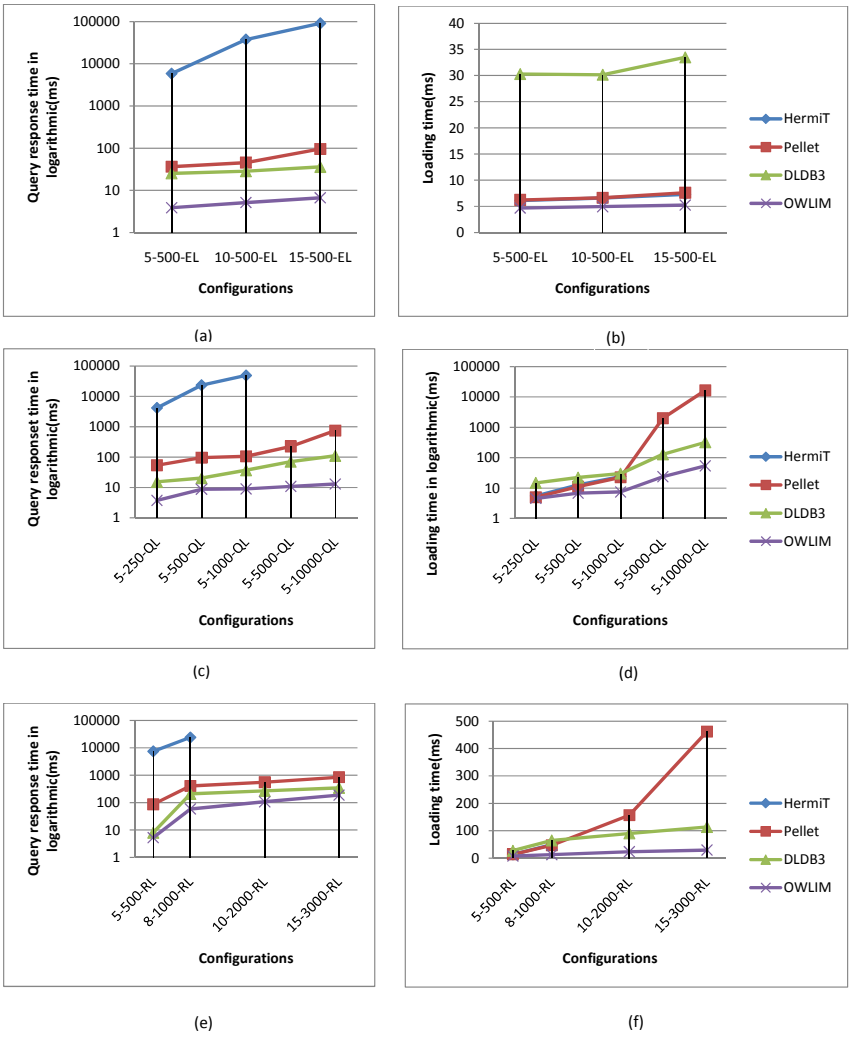


Fig. 2. Query response time and loading time of OWL 2 EL, QL and RL.

dump, in order to make it meet our experimental setup, we partitioned it into different subsets using the number of triples of each test configuration ($50 \times nD$ in each configuration). In addition, we collected the statistics of the number of each type of ontological axiom shown in Table 1 in the SWRC ontology and used them to parameterize the benchmark to produce realistic synthetic ontologies. For each test configuration, we use the learned SWRC ontology profile to generate five SWRC-like domain ontologies together with their corresponding ontology mappings. The SWRC ontology constructor statistics is shown in Table 2. We evaluate Pellet and OWLIM because both systems are memory-based and two representatives of applying two different well-known reasoning algorithms: tableau and rule-based. We compute the average query response time for each configuration. As shown in Fig. 3, although Pellet is slightly faster on benchmark data than on real data, and OWLIM is slightly slower on the benchmark data, each maintains the same trend on the benchmark that it had with the original data. Although more experiments are needed to draw significant conclusions, this suggests that our benchmark may be able to generate synthetic datasets that are representative enough for users to evaluate their systems or reasoners instead of using the real semantic data, which cannot be easily customized and has little ontology integration.

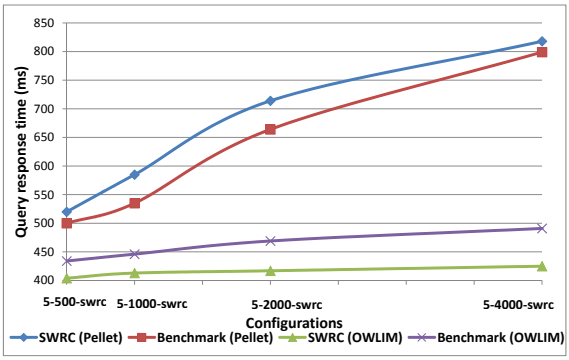


Fig. 3. Query response time of Benchmark and SWRC.

5 Conclusions and Future Work

We extend our early work [8] on creating a benchmark that can generate user-customized ontologies together with related mappings and data sources. It supports both OWL and OWL 2. It can also generate diverse conjunctive queries with at least one answer. Our experiments have demonstrated that this benchmark can effectively evaluate semantic reasoners by generating realistic synthetic semantic data. In particular, we have collected statistics from real world ontologies and used these to parameterize the benchmark to produce more realistic

synthetic ontologies under controlled conditions. According to our evaluation, OWLIM has the best performance and is complete for OWL 2 QL, RL and SWRC. Pellet has better performance than HermiT in all experimental settings. HermiT has the worst scalability in OWL 2 QL and RL because of high memory consumption. DLDB3 is incomplete in all OWL 2 profiles because it is designed for an OWL fragment reasoner, but it shows better performance in query response time than Pellet and HermiT.

However, there is still significant room for improvement. First, we need to consider to collect statistics of semantic data besides the ontology schemas. One way to do so is to find the different RDF graph patterns implied by the real semantic data and use these to guide our data generation. Second, in the approximation evaluation, we need to evaluate our benchmark using more data sets with different characteristics from SWRC. It should be also pointed out that we believe that the performance of any given system will vary depend on the structure of the ontology and data used to evaluate it. Thus our benchmark does not provide the final say on each system's characteristics. However, it allows developers to automate a series of experiments that give a good picture of how system performs under the general parameters of a given scenario.

References

1. S. Bail, B. Parsia, and U. Sattler. JustBench: A framework for owl benchmarking. In *International Semantic Web Conference (1)*, pages 32–47, 2010.
2. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
3. A. Chitnis, A. Qasem, and J. Heflin. Benchmarking reasoners for multi-ontology applications. In *EON*, pages 21–30, 2007.
4. Z. P. et al. Hawkeye: A practical large scale demonstration of semantic web integration. Technical Report LU-CSE-07-006, Lehigh University, 2007.
5. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
6. I. Horrocks and P. F. Patel-Schneider. Dl systems comparison (summary relation). In *Description Logics*, 1998.
7. A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.
8. Y. Li, Y. Yu, and J. Heflin. A multi-ontology synthetic benchmark for the semantic web. In *In Proc. of the 1st International Workshop on Evaluation of Semantic Technologies*, 2010.
9. L. Ma, Y. Yang, Z. Qiu, G. T. Xie, Y. Pan, and S. Liu. Towards a complete owl ontology benchmark. In *ESWC*, pages 125–139, 2006.
10. B. Motik, R. Shearer, and I. Horrocks. Optimized reasoning in description logics using hypertableaux. In *CADE*, pages 67–83, 2007.
11. Z. Pan, Y. Li, and J. Heflin. A semantic web knowledge base system that supports large scale data integration. In *In Proc. of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 125–140, 2010.
12. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

jcel: A Modular Rule-based Reasoner

Julian Mendez

Theoretical Computer Science, TU Dresden, Germany
mendez@tcs.inf.tu-dresden.de

Abstract. jcel is a reasoner for the description logic $\mathcal{EL}^+$ that uses a rule-based completion algorithm. These algorithms are used to get subsumptions in the lightweight description logic $\mathcal{EL}$ and its extensions. One of these extensions is $\mathcal{EL}^+$, a description logic with restricted expressivity, but used in formal representation of biomedical ontologies. These ontologies can be encoded using the Web Ontology Language (OWL), and through the OWL API, edited using the popular ontology editor Protégé. jcel implements a subset of the OWL 2 EL profile, and can be used as a Java library or as a Protégé plug-in. This system description presents the architecture and main features of jcel, and reports some of the challenges and limitations faced in its development.

1 Introduction

This system description presents jcel[1], a reasoner for the description logic $\mathcal{EL}^+$. The design and implementation details refer to jcel 0.17.1, unless other version is specified.

The lightweight description logic (DL) $\mathcal{EL}$ and its extensions have recently drawn considerable attention since important inference problems, such as the subsumption problem, are polynomial in $\mathcal{EL}$ [1,4,2]. In addition, biomedical ontologies such as the large ontology SNOMED CT[2], can be defined using this logic.

The basic entities are *concepts* (class expressions), which are built with *concept names* (classes) and *role names* (object properties).

An *ontology* is a formal vocabulary of terms which refers to a conceptual schema inside a domain. The terms are related using an *ontology language*. The main service that this reasoner provides is *classification*, a hierarchical relation of the concepts in the ontology.

2 Language Subset Supported

jcel, as an $\mathcal{EL}^+$ reasoner, includes the standard constructors of $\mathcal{EL}$: conjunction ($C \sqcap D$), existential restriction ($\exists r.C$), and the top concept ($\top$). In addition, this logic includes role composition ($r \circ s \sqsubseteq t$) and role hierarchy ($r \sqsubseteq s$).

[1] http://jcel.sourceforge.net/
[2] http://www.ihtsdo.org/snomed-ct/

Table 1. Syntax and semantics.

Name	Syntax	Semantics
concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role name	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
top	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{x \mid \exists y : (x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
role hierarchy	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
role composition	$r \circ s \sqsubseteq t$	$r^{\mathcal{I}} \circ s^{\mathcal{I}} \subseteq t^{\mathcal{I}}$

Table 1 summarizes the semantics of the mentioned constructors.

There is an experimental development of jcel [7] to support inverse and functional roles.

3 Syntaxes and Interfaces Supported

jcel is developed in Java and can be compiled in Java 1.6 or Java 1.7 indistinctly. jcel is integrated to the OWL API 3.2.4[3], which is a Java application programming interface (API) and reference implementation for creating, manipulating and serializing OWL ontologies. Using the OWL API, jcel can be used as a Protégé[4] plug-in. Protégé is a free, open source ontology editor, and also a knowledge base framework. Protégé ontologies can be exported to several formats, like RDF, OWL and XML Schema. In order to keep compatibility with Protégé 4.1, jcel is distributed as a Java binary for Java 1.6.

jcel can also be used as a library without the OWL API. It has its own factories to construct the optimized data types used in the core.

4 Reasoning Algorithm Implemented

The algorithm is rule-based, and there is a set of rules that are successively applied to saturate data structures. These rules are called *completion rules*. The process is called *classification*, and is the main part of jcel's algorithm.

An algorithm that classifies the set of axioms by applying these rules could be expensive in time if it is performed by a systematic search. The algorithm used by jcel is based on CEL's algorithm [2,3], but generalized with a *change propagation* approach [6]. This approach detects the changes in the data structure being saturated, and triggers the rules in consequence.

The input of the algorithm is a *normalized* set of axioms $\mathcal{T}$ as in the following list: $A \sqsubseteq B$, $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$, $A \sqsubseteq \exists r.B$, $\exists r.A \sqsubseteq B$, $r \sqsubseteq s$, $r \circ s \sqsubseteq t$.

[3] http://owlapi.sourceforge.net/

[4] http://protege.stanford.edu/

The invariant of the algorithm has a set S, called *set of subsumptions*, such that for each pair of concept names A, B in $\mathcal{T}$: $(A, B) \in S$ if and only if $\mathcal{T} \models A \sqsubseteq B$, and a set R such that for each triple of role r, and concept names A, B in $\mathcal{T}$: $(r, A, B) \in R$ if and only if $\mathcal{T} \models A \sqsubseteq \exists r.B$, where $\models$ has the usual meaning.

The algorithm finishes when S is saturated. The output is S itself, which tells the subsumption relation for every pair of concept names.

In Figure 1, we can observe S and R, the completion rules (CR-1, CR-2, ...), the duplicates checker, and a set Q which has a set of entries to be processed.

In each iteration, an S-entry (a pair) or an R-entry (a triple) is taken from Q and added to either S or R. This change is propagated and sent to the chain of rules sensitive to changes in S (S-chain) or in R (R-chain). Every completion rule takes the new entry as input and returns a set of S- and R-entries. The arrows indicate how these entries flow.

Every element proposed by the rules is verified by the duplicates checker before being added to Q. The dashed lines indicate that the duplicates checker uses S and R for the verification. This procedure is repeated until Q is empty.

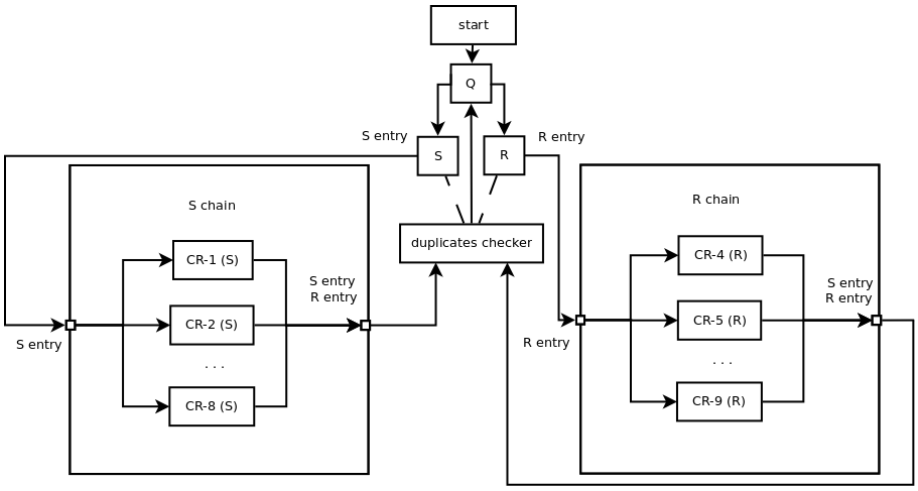


Fig. 1. Dynamic diagram.

5 Architecture and Optimization Techniques

jcel axioms are encoded using integers. This optimizes the use of memory and required time in comparisons. Any program using jcel as a library can encode its entities with integers, and take advantage of this efficient representation.

jcel is composed by several modules, as shown in Figure 2. The arrows indicate the relation "depends on".

jcel.coreontology and jcel.core are modules that use only normalized axioms. The former contains the normalized axioms themselves, the latter contains the implementation of the completion rules, together with the data structures for the subsumption graphs.

jcel.ontology is a module that contains the axioms for the ontology and the procedures to normalize the ontology. jcel.reasoner is the reasoner interface. It can classify an ontology and compute entailment.

All the modules mentioned above work with data types based on integers.

jcel.owlapi is the module that connects to the OWL API. This module performs the translation between the OWL API axioms and jcel axioms. jcel.protege is a tiny module used to connect to Protégé.

Figure 2 describes the module dependencies and shows that jcel can be used:

- extending the core (with jcel.coreontology and jcel.core)
- as a library using integers (adding jcel.ontology and jcel.reasoner)
- as a library using the OWL API (adding jcel.owlapi)
- as a Protégé plug-in (adding jcel.protege)

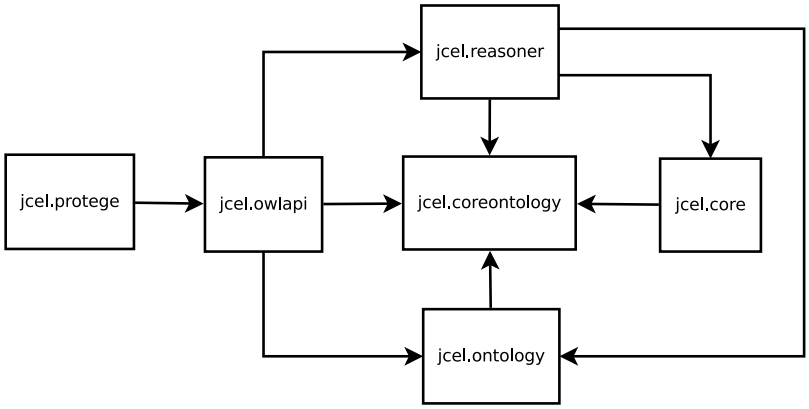


Fig. 2. Module dependencies.

6 Particular Advantages and Disadvantages

jcel is a **pure Java**, **open source** project. Its source code can be cloned with Git[5] and compiled using Apache Ant[6] or Apache Maven[7] indistinctly.

[5] http://git-scm.com/
[6] http://ant.apache.org/
[7] http://maven.apache.org/

jcel includes several advantages derived from its design and from good practices of software engineering. Regarding the design, jcel has **independent maintenance of rules**. Each rule works as an independent unit that can be implemented, improved and profiled independently.

Regarding the good practices, jcel uses **no null pointers**. Every public method is prevented of accepting null pointers (throwing a runtime exception) and no public method returns a null pointer. Referred by its inventor as "The Billion Dollar Mistake"[8], a null pointer may have different *intended meanings*. For example, $min(1, \textbf{null})$, may give the results "0" (considering **null** as 0), "1" (considering **null** as an empty argument), or "**null**" (considering **null** as an undefined value).

Another good practice is that public methods return **unmodifiable collections** when they refer to collections used in the internal representation. This prevents a defective piece of code from modifying the collection.

jcel has **no cyclic dependencies of packages** in each module. This facilitates maintenance, since modifications on one package do not alter any other package that does not depend on the former.

jcel has also some points that are not implemented yet, but can be considered as good future improvements.

One of the improvements is to apply techniques to **unchain properties** [5]. This would be useful for large ontologies.

Another improvement is **incremental classification**. This could be especially interesting for entailment, since jcel computes entailment by adding fresh auxiliary concepts and reclassifying the ontology.

Finally, the **reduction of use of memory** is an important improvement to consider. jcel 0.16.0 was the first version to include entailment, but also became significantly slower than jcel 0.15.0 when classifying large ontologies. The reason was a side effect resulting from the memory required for the entailment, producing a frequent execution of the garbage collector. This was partially solved in jcel 0.17.0.

7 Application focus

jcel can be used to classify small and medium-sized ontologies of the $\mathcal{EL}$ family.

jcel was designed to be robust, resilient, modular and extensible. Its binaries are small and can be distributed inside other tools. One tool that uses jcel is GEL[9] (Generalizations for $\mathcal{EL}$) [7], which extends the core of jcel.

Another example is OntoComP[10]. It is a Protégé plug-in for completing OWL ontologies, and for checking whether an OWL ontology contains "all relevant information" about the application domain. This tool uses the OWL API to connect to jcel as a library.

[8] http://qconlondon.com/london-2009/speaker/Tony+Hoare
[9] http://sourceforge.net/p/gen-el/
[10] http://ontocomp.googlecode.com/

jcel is also used to verify correctness in UEL[11], Unification for the description logic $\mathcal{EL}$.

8 Conclusion

jcel is a reasoner for lightweight ontologies. Its rule-based design makes it easy to be configured according to the rules. Provides a high-level interface to be used as a tool seamlessly integrated to the OWL API.

The implementation is modular, resilient and highly extensible. Implemented in a state-of-the-art technology, it has a low coupling and a high cohesion, it is portable, and has an optimal interface to connect to other technologies of the Semantic Web.

References

1. Franz Baader. Terminological cycles in a description logic with existential restrictions. In *Proc. IJCAI'03*, 2003.
2. Franz Baader and Sebastian Brandt and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. IJCAI'05*, 2005.
3. Franz Baader and Carsten Lutz and Boontawee Suntisrivaraporn. Is Tractable Reasoning in Extensions of the Description Logic $\mathcal{EL}$ Useful in Practice?. Journal of Logic, Language and Information, Special Issue on Method for Modality (M4M), 2007.
4. Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In *Proc. ECAI'04*, 2004.
5. Yevgeny Kazakov and Markus Krötzsch and František Simančík. Unchain My EL Reasoner. In Riccardo Rosati, Sebastian Rudolph, Michael Zakharyaschev, editors, *Proceedings of the 24th International Workshop on Description Logics (DL-11)*. CEUR, 2011.
6. Julian Mendez. A Classification Algorithm for $\mathcal{ELHIf}R^+$. Dresden University of Technology, 2011.
7. Julian Mendez and Andreas Ecke and Anni-Yasmin Turhan. Implementing completion-based inferences for the $\mathcal{EL}$-family. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Proceedings of the international Description Logics workshop*, volume 745. CEUR, 2011.
8. Quoc Huy Vu. Subsumption in the Description Logic $\mathcal{ELHIf}R^+$ w.r.t. General TBoxes. Dresden University of Technology, 2008.

[11] http://uel.sourceforge.net/

The HermiT OWL Reasoner

Ian Horrocks, Boris Motik, and Zhe Wang

Oxford University Department of Computer Science
Oxford, OX1 3QD, UK
{ian.horrocks,boris.motik,zhe.wang}@cs.ox.ac.uk

Abstract. HermiT is the only reasoner we know of that fully supports the OWL 2 standard, and that correctly reasons about properties as well as classes. It is based on a novel "hypertableau" calculus that addresses performance problems due to nondeterminism and model size—the primary sources of complexity in state-of-the-art OWL reasoners. HermiT also incorporates a number of novel optimizations, including an optimized ontology classification procedure. Our tests show that HermiT performs well compared to existing tableau reasoners and is often much faster when classifying complex ontologies.

1 Introduction

HermiT is an OWL reasoning system based on a novel *hypertableau* calculus [12]. Like existing tableau based systems, HermiT reduces all reasoning tasks to ontology satisfiability testing, and proves the (un-)satisfiability of an ontology by trying to construct (an abstraction of) a suitable model. When compared to tableau calculi, however, the hypertableau technique can greatly reduce both the size of constructed models and the non-deterministic guessing used to explore all possible constructions. Moreover, HermiT employs a novel classification algorithm that greatly reduces the number of subsumption (and hence satisfiability) tests needed to classify a given ontology.

Our tests show that HermiT is as fast as other OWL reasoners when classifying relatively easy-to-process ontologies, and usually much faster when classifying more difficult ontologies. Moreover, HermiT is currently the only reasoner known to us that fully supports the OWL 2 standard: it supports all of the datatypes specified in the standard, and it correctly reasons about properties as well as about classes. Most other reasoners support only a subset of the OWL 2 datatypes [11], and all other OWL reasoners known to us implement only syntax based reasoning when classifying properties, and may thus fail to detect non-trivial but semantically entailed sub-property relationships [4].

HermiT also includes some nonstandard functionality that is currently not available in any other system. In particular, HermiT supports reasoning with ontologies containing *description graphs*. As shown in [10], description graphs allow for the representation of structured objects—objects composed of many parts interconnected in arbitrary ways. These objects abound in bio-medical ontologies such as FMA and GALEN, but they cannot be faithfully represented in OWL.

HermiT is available as an open-source Java library, and includes both a Java API and a simple command-line interface. We use the OWL API [6] both as part of the public Java interface and as a parser for OWL files; HermiT can thus process ontologies in any format handled by the OWL API, including RDF/XML, OWL Functional Syntax, KRSS, and OBO.

2 Architecture and Optimizations

On OWL ontology $\mathcal{O}$ can be divided into three parts: the property axioms, the class axioms, and the facts. These correspond to the RBox $\mathcal{R}$, TBox $\mathcal{T}$, and ABox $\mathcal{A}$ of a Description Logic knowledge base $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$. All basic reasoning tasks, including subsumption checking, can be reduced to testing the satisfiability of such a knowledge base. For example, $\mathcal{K} \models A \sqsubseteq B$ iff $(\mathcal{R}, \mathcal{T}, \mathcal{A} \cup \{A \sqcap \neg B(s)\})$ is not satisfiable, where s is a "fresh" individual (i.e., one that does not occur in $\mathcal{K}$).

To show that a knowledge base $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ is satisfiable, a tableau algorithm constructs a *derivation*—a sequence of ABoxes $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$, where $\mathcal{A}_0 = \mathcal{A}$ and each $\mathcal{A}_i$ is obtained from $\mathcal{A}_{i-1}$ by an application of one *inference rule*. The inference rules make the information implicit in the axioms of $\mathcal{R}$ and $\mathcal{T}$ explicit, and thus evolve the ABox $\mathcal{A}$ towards (an abstraction of) a model of $\mathcal{K}$. The algorithm terminates either if no inference rule is applicable to some $\mathcal{A}_n$, in which case $\mathcal{A}_n$ represents a model of $\mathcal{K}$, or if $\mathcal{A}_n$ contains an obvious contradiction, in which case the model construction has failed. The following inference rules are commonly used in DL tableau calculi.

- $\sqcup$-rule: Given $(C_1 \sqcup C_2)(s)$, derive either $C_1(s)$ or $C_2(s)$.
- $\sqcap$-rule: Given $(C_1 \sqcap C_2)(s)$, derive $C_1(s)$ and $C_2(s)$.
- $\exists$-rule: Given $(\exists R.C)(s)$, derive $R(s,t)$ and $C(t)$ for t a fresh individual.
- $\forall$-rule: Given $(\forall R.C)(s)$ and $R(s,t)$, derive $C(t)$.
- $\sqsubseteq$-rule: Given an axiom $C \sqsubseteq D$ and an individual s, derive $(\neg C \sqcup D)(s)$.

The $\sqcup$-rule is nondeterministic, and the knowledge base $\mathcal{K}$ is unsatisfiable if and only if all choices lead to a contradiction.

Or-Branching This "case based" procedure for handing disjunctions is sometimes called *or-branching*. The $\sqsubseteq$-rule is the main source of or-branching, as it adds a disjunction for each TBox axiom to each individual in an ABox, even if the corresponding axiom is equivalent to a Horn clause, and so inherently deterministic. Such indiscriminate application of the $\sqsubseteq$-rule can be a major source of inefficiency, and this has been addressed by various *absorption* optimizations [2, Chapter 9], including role absorption [14] and binary absorption [8].

HermiT's hypertableau algorithm generalizes these optimizations by rewriting description logic axioms into a form which allows all such absorptions be performed simultaneously, as well as allowing additional types of absorption impossible in standard tableau calculi. Furthermore, HermiT actually rewrites DL concepts to further reduce nondeterminism, and is thus able to apply absorption-style optimizations much more pervasively.

And-Branching The introduction of new individuals in the $\exists$-rule is sometimes called *and-branching*, and it is another major source of inefficiency in tableau algorithms [2]. To ensure termination, tableau algorithms employ *blocking* to prevent infinitely repeated application of the $\exists$-rule [7]. Standard blocking is applied only along a single "branch" of fresh individuals. HermiT uses a more aggressive *anywhere blocking* strategy [12] that can reduce the size of generated models by an exponential factor, and this substantially improves real-world performance on many difficult and complex ontologies.

HermiT also tries to further reduce the size of the generated model using a technique called *individual reuse*: when expanding an existential $\exists R.C$, it first attempts to re-use some existing individual labeled with C to construct a model, and only if this model construction fails does it introduce a new individual. This approach allows HermiT to consider non-tree-shaped models, and drastically reduces the size of models produced for ontologies which describe complex structures, such as ontologies of anatomy. "Reused" individuals, however, are semantically equivalent to nominal concepts, and thus performance gains due to individual reuse are highly dependent upon the efficient handling of nominals. HermiT therefore uses an optimised *nominal introduction rule* that reduces nondeterminism, and is more conservative in its introduction of new nominals.

Caching Blocking Labels Anywhere blocking avoids repetitive model construction in the course of a single satisfiability test. HermiT further extends this approach to avoid repetitive construction across an entire set of satisfiability tests. Conceptually, instead of performing n different tests by constructing n different models, it performs a single test which constructs a single model containing n independent fragments. Although no two fragments are connected, the individuals in one fragment can block those in another, greatly reducing the size of the combined model. In practice, tests are not actually performed simultaneously. Instead, after each test a compact representation of the model generated is retained for the purpose of blocking in future tests. This naïve strategy is, however, not compatible with ontologies containing nominals, which could connect the models from independent tests.

This optimization has been key to obtaining the results that we present in Section 3. For example, on GALEN only one satisfiability test is costly because it computes a substantial part of a model of the TBox; all subsequent satisfiability tests reuse large parts of that model.

Classification Optimizations DL reasoning algorithms are often used in practice to compute a *classification* of a knowledge base $\mathcal{K}$—that is, to determine whether $\mathcal{K} \models A \sqsubseteq B$ for each pair of atomic concepts A and B occurring in $\mathcal{K}$. Clearly, a naïve classification algorithm would involve a quadratic number of subsumption tests, each of which can potentially be expensive. To obtain acceptable levels of performance, various optimizations have been developed that reduce the number of subsumption tests [3] and the time required for each test [2, Chapter 9].

HermiT employs a novel classification procedure, and exploits the unique properties of the system's new calculus to further optimise the procedure. In

Table 1: Statics of the ontologies

Ontology Name	DL Expressivity	Classes	Properties	TBox	RBox
EMap (Feb09)	$\mathcal{EL}$	13737	2	13730	0
GO Term DB (Feb06)	$\mathcal{EL}++$	20526	1	28997	1
DLP ExtDnS 397	$\mathcal{SHIN}$	96	186	232	675
LUBM (one university)	$\mathcal{ALEHI}^+(D)$	43	32	142	51
Biological Process (Feb09)	$\mathcal{EL}++$	16303	5	32286	3
MGED Ontology	$\mathcal{ALCOF}(D)$	229	104	452	21
RNA With Individual (Dec09)	$\mathcal{SRIQ}(D)$	244	93	364	310
NCI Thesaurus (Feb09)	$\mathcal{ALCH}(D)$	70576	189	100304	290
OBI (Mar10)	$\mathcal{SHOIN}(D)$	2638	83	9747	150
FMA Lite (Feb09)	$\mathcal{ALEI}^+$	75145	3	119558	3
FMA-constitutional part (Feb06)	$\mathcal{ALCOIF}(D)$	41648	168	122695	395
GALEN-doctored	$\mathcal{ALEHIF}^+$	2748	413	3937	799
GALEN-undoctored	$\mathcal{ALEHIF}^+$	2748	413	4179	800
GALEN-module1	$\mathcal{ALEHIF}^+$	6362	162	14515	219
GALEN-full	$\mathcal{ALEHIF}^+$	23136	950	35531	2165

particular, when it tries to construct a model I of $\mathcal{K} \cup \{A(a)\}$ (in order to determine the satisfiability of A), HermiT is able to exploit the information in I to derive a great deal of information about both subsumers and non-subsumers of A, information that can be efficiently exploited by the new classification procedure [4].

3 Empirical Results

To evaluate our reasoning algorithm in practice, we compared HermiT with the state-of-the-art tableau reasoners Pellet 2.3.0 [13], and FaCT++ 1.5.3 [15].

We selected a number of standard test ontologies, and measured the time needed to classify them using each of the mentioned reasoners. Unlike Pellet and FaCT++, HermiT does not include a dedicated reasoner for any tractable fragment of OWL 2. Hence, we mainly focus on ontologies that exploit most or all of the expressive power available in OWL 2. All tests were performed on a 2.7 GHz MacBook Pro with 8 GB of physical memory. A classification attempt was aborted if it exhausted all available memory (Java tools were allowed to use 2 GB of heap space), or if it exceeded a timeout of 20 minutes.

The majority of the test ontologies were classified very quickly by all three reasoners. For these "trivial" ontologies, the performance of HermiT was comparable to that of the other reasoners. Therefore, we consider here only the test results for "interesting" ontologies—that is, ontologies that are either not trivial or on which the tested reasoners exhibited a significant difference in performance (see Table 1 for details of these ontologies).

Table 2 summarizes the results of our tests on these "interesting" ontologies. Since HermiT has no special handling for tractable fragments of OWL 2, the per-

Table 2: Results of Performance Evaluation

Ontology Name	Classification Times (seconds)		
	HermiT	Pellet	FaCT++
EMap (Feb09)	1.1	0.4	34.2
GO Term DB (Feb06)	1.3	1.3	6.1
DLP ExtDnS 397	1.3	timeout	0.05
LUBM (one university)	1.7	0.7	152.7
Biological Process (Feb09)	1.8	4.0	8.0
MGED Ontology	2.1	19.6	0.04
RNA With Individual (Dec09)	2.7	0.8	102.9
NCI Thesaurus (Feb09)	58.2	12.3	4.4
OBI (Mar10)	150.0	timeout	17.2
FMA Lite (Feb09)	211.1	timeout	timeout
FMA-constitutional part (Feb06)	1638.3	timeout	396.9
GALEN-doctored	1.8	timeout	2.5
GALEN-undoctored	6.7	out of mem.	11.6
GALEN-module1	out of mem.	timeout	timeout
GALEN-full	out of mem.	timeout	timeout

formance of HermiT on such ontologies may not be competitive. For example, FaCT++ shows advantages when classifying ontologies which fall into a predefined syntactic fragment for which it uses a more efficient reasoning technique [16]. Different versions of GALEN have commonly been used for testing the performance of DL reasoners. The full version of the ontology (called GALEN-full) cannot be processed by any of the reasoners. Thus, we extracted a module (called GALEN-module1) based on a single concept from GALEN-full using the techniques from [5] in order to determine if modularisation techniques might make classification feasible. However, although the module is much smaller than the full ontology, no reasoner was able to classify it either. Our analysis has shown that, due to a large number of cyclic axioms, the reasoners construct extremely large ABoxes and eventually exhaust all available memory (or get lost in the resulting large search space). FMA-constitutional part exhibits similar features, but to a lesser extent, and both HermiT and FaCT++ were able to classify it. Because of the failure of DL reasoners to process GALEN-full, various simplified versions of GALEN have often been used in practice. As Table 2 shows, these ontologies can still be challenging for state-of-the-art reasoners. HermiT, however, can classify them quite efficiently.

4 Conclusions and Future Directions

We have described HermiT, an OWL reasoner based on novel algorithms and optimizations. HermiT fully supports the OWL 2 standard, and shows significant performance advantages over other reasoners across a wide range of expressive real-world ontologies. Although not always the fastest, HermiT exhibits relatively robust performance on our tested ontologies, and as shown in our results, it never

failed to classify an ontology in the test corpus that was successfully handled by one of the other reasoners. HermiT also includes support for some non-standard ontology features, such as description graphs.

We are continuing to develop HermiT, and to explore new and refined optimization techniques. We also continue to extend its functionality: the latest version, for example, provides support for the SPARQL 1.1 query language [1]. We are also investigating techniques for exploiting specialized reasoning techniques, such as those implemented in the ELK system [9], to speed up the classification of ontologies that are (largely) within a fragment of OWL that such techniques can handle.

Acknowledgments This work was supported by the EU FP7 project SEALS and by the EPSRC projects ConDOR, ExODA, and LogMap.

References

1. SPARQL 1.1 Query Language. W3C Working Draft, 12 May 2011.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. 2nd edition, 2007.
3. F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, and E. Franconi. Making KRIS Get a Move on. *Applied Intelligence*, 4(2):109–132, 1994.
4. B. Glimm, I. Horrocks, B. Motik, R. Shearer, and G. Stoilos. A Novel Approach to Ontology Classification. *J. of Web Semantics*, 10(1), 2011.
5. B. Cuenca Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Modular Reuse of Ontologies: Theory and Practice. *JAIR*, 31:273–318, 2008.
6. M. Horridge and S. Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In *Proc. OWLED 2009*, 2009.
7. I. Horrocks, U. Sattler, and S. Tobies. Reasoning with Individuals for the Description Logic $\mathcal{SHIQ}$. In *Proc. CADE-17*, pages 482–496, 2000.
8. A. K. Hudek and G. Weddell. Binary Absorption in Tableaux-Based Reasoning for Description Logics. In *Proc. DL 2006*, 2006.
9. Y. Kazakov, M. Krötzsch, and F. Simančík. Concurrent Classification of EL Ontologies. In *Proc. of ISWC 2011*, pages 305–320, 2011.
10. B. Motik, B. Cuenca Grau, I. Horrocks, and U. Sattler. Representing Ontologies Using Description Logics, Description Graphs, and Rules. *Artificial Intelligence*, 173(14):1275–1309, 2009.
11. B. Motik and I. Horrocks. OWL Datatypes: Design and Implementation. In *Proc. of ISWC 2008*, pages 307–322, 2008.
12. B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *JAIR*, 36:165–228, 2009.
13. E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. *J. of Web Semantics*, 5(2):51–53, 2007.
14. D. Tsarkov and I. Horrocks. Efficient Reasoning with Range and Domain Constraints. In *Proc. DL 2004*, 2004.
15. D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proc. IJCAR 2006*, pages 292–297, 2006.
16. D. Tsarkov, I. Horrocks, and P. F. Patel-Schneider. Optimizing Terminological Reasoning for Expressive Description Logics. *J. of Automated Reasoning*, 39(3):277–316, 2007.

OWLIM Reasoning over FactForge

Barry Bishop, Atanas Kiryakov, Zdravko Tashev, Mariana Damova, Kiril Simov

Ontotext AD, 135 Tsarigradsko Chaussee, Sofia 1784, Bulgaria

Abstract. In this paper we present the reasoning mechanism in the OWLIM family of semantic repositories, which is based on materialization. This mechanism is evaluated using a combination of datasets from the Linked Open Data cloud in a public service called FactForge, where the benefits of materialization are manifested in improved SPARQL query performance.

Keywords: LOD, materialization, OWLIM, RDF, semantic repository

1 Introduction

In this paper we present the reasoning mechanism employed in the OWLIM family of semantic repositories. These native RDF databases are implemented in Java and comprise storage components, inference-engine and query-answering engine. They are available in three editions: OWLIM-Lite, an in-memory and very fast RDF database that can load data at over 50,000 statements per second on a 1,000 USD machine using non-trivial inference; OWLIM-SE, that uses file-based, paged indices and data structures to be able to process tens of billions of RDF statements on standard desktop hardware; and OWLIM-Enterprise, a replication cluster based on OWLIM-SE that provides resilience and linearly scalable parallel query performance. OWLIM-Lite is free-for-use, whereas OWLIM-SE and OWLIM-Enterprise are the commercial editions licensed per CPU core. OWLIM-SE and OWLIM-Enterprise use a number of storage and query optimizations that allow it to sustain outstanding insert and delete performance even when managing tens of billions of statements of linked open data.

The experiments conducted were performed using datasets from the Linked Open Data cloud, see section 3, that constitute a reason-able view [2] named FactForge[1]. Entities described in more than one dataset are unified via `owl:SameAs` statements and a common ontology PROTON[2], called a unification ontology [1] for FactForge used for querying and data integration. PROTON is mapped to DBPedia[3], FreeBase[4] and Geonames[5]. Query performance with such dataset sizes is like-wise good, with sub-second response times for all the example queries found on the FactForge site.

[1] http://factforge.net/
[2] http://www.ontotext.com/proton-ontology
[3] http://dbpedia.org/About
[4] http://www.freebase.com/
[5] http://www.geonames.org/

2 Reasoning in OWLIM

The inferencing strategy in OWLIM is one of materialization based on R-Entailment as defined by ter Horst [3], where Datalog like rules with inequality constraints operate directly on a single ternary relation that represents all triples. In addition, free variables in rule heads are treated as blank nodes. Materialization involves computing all the entailed statements at load time. While this introduces additional reasoning cost when loading statements into a repository, the desirable consequence is that query evaluation can proceed extremely quickly. Several standard rule sets are included in all editions of OWLIM and these include:

empty – no inference;
rdfs[6] – RDFS semantics using rule entailment, but without data-type reasoning, i.e. without the literal generalization and related rules;
owl-horst – equivalent to pD*, again without data-type reasoning;
owl-max – RDFS and OWL-Lite (that can be captured in rules);
owl2-ql – a fragment of OWL2 Full based on DL-Lite$_R$, a variant of DL-Lite that does not require the unique name assumption;
owl2-rl[7] – the OWL2 RL profile, a fragment of OWL2 Full amenable to implementation on rule-engines, but without data-type reasoning.

In addition to the standard semantics, user-defined rule-sets can be used. In this case the user provides the full pathname to a custom rule file that contains definitions of axiomatic triples, rules and consistency checks. For ease of use, the rule files for the standard rule-sets are included in the distribution and users can modify or extend these for their specific purposes.

Consistency checks are used to ensure that the data model is in a consistent state and are applied whenever an update transaction is committed, for example to ensure that `owl:Nothing` has no members or that no pair of individuals have both `owl:sameAs` and `owl:differentFrom` relationships.

During loading, all inferred statements are materialized, except those generated as a result of the semantics of `owl:sameAs`. OWLIM-SE uses special data structures to maintain equivalence classes and uses the URI of the first asserted resource in each equivalence class in the statement indices. This allows for the correct expansion of results during query-answering while keeping the index sizes manageable. This technique has the further advantage that it can be switched off during query answering in order to limit the number of 'duplicate' results.

3 FactForge - a Reason-able View on LOD

FactForge is a reason-able view [] to the Web of Linked Data, made up of 11 of the central LOD datasets, which have been selected and refined in order to serve as a useful index and entry point to the LOD cloud and to present a good use-case for large-scale reasoning and data integration. The compound dataset of FactForge is the largest

[6] http://www.w3.org/TR/rdf-schema/
[7] http://www.w3.org/TR/owl2-profiles/

body of heterogeneous general knowledge on which inference has been performed. It counts 1.7 billion explicit statements; 15 billion retrievable statements available after inference and owl:sameAs expansion (cf. section 2); including 1.4 billion inferred statements. The datasets combined in FactForge are:

- DBPedia - an RDF dataset derived from Wikipedia, designed to provide as full as possible coverage of the factual knowledge that can be extracted from the InfoBoxes of Wikipedia with a high level of precision;
- Freebase - a dataset containing information about 11 million things, including movies, books, locations, companies and more, with underlying schema based on properties, and not ontologies, which exploits user generated categories;
- Geonames - a geographic database that covers 6 million of the most significant geographical features on Earth, characterised by coordinates and relations to other features (e.g. 'parent feature' in which the feature is nested);
- CIA World Factbook[8] - a collection of structured data, including statistical, geographic, political, and other information about all countries;
- Lingvoj[9] - providing descriptions of the most popular human languages; currently it contains information about more than 500 languages;
- MusicBrainz[10] (RDF from Zitgist) – a comprehensive music information suitable for browsing or useful for tagging;
- WordNet[11] - a lexical database of English. Nouns, verbs, adjectives and adverbs that are grouped into sets of cognitive synonyms (synsets).

The interlinking of datasets is facilitated by DBPedia, which provides link-sets of owl:sameAs links of DBpedia with GeoNames, Lingvoj, Freebase, MusicBrainz, UMBEL and Wordnet. These link-sets are also loaded into FactForge along with the following ontologies and schemata:

- DCMI Metadata Terms[12] (Dublin Core - DC) - a relatively small, but very popular metadata schema. It defines attributes that can be used to describe information resources;
- SKOS[13] (Simple Knowledge Organization System) - a relatively simple RDF schema for describing taxonomies of concepts linked to each other by any sort of subsumption hierarchy;
- RSS - an RDF schema designed to enable syndication of machine-readable information about updates from Web sites;
- FOAF - an ontology for defining and linking personal profiles on the Web.

FactForge provides several methods to explore the combined dataset that exploits some of the advanced features of OWLIM-SE. Firstly, 'RDF Search and Explore' allows entities to be searched by keyword with a real-time auto-suggest feature ordered by 'RDF Rank' (similar to Google's Page Rank). The results page shows all triples where the selected node appears as the subject, predicate or object, together with the

8 http://www4.wiwiss.fu-berlin.de/factbook/
9 http://lingvoj.org/
10 http://musicbrainz.org/
11 http://wordnet.princeton.edu/
12 http://dublincore.org/
13 http://www.w3.org/2004/02/skos/

preferred label, RDF Rank indicator, etc. Secondly, a SPARQL page allows users to write their own queries with clickable options to add each of the known namespaces. The results are presented in a conveniently formatted table with the option to download results in various formats (SPARQL/XML, JSON, etc). Lastly, a graphical search facility called 'RelFinder' [4] that discovers paths between selected nodes. This is a computationally intensive activity and the results are displayed and updated dynamically during each iteration. The resulting graph can be reshaped by the user with simple click and drag operations. Entities within the emerging graph can be selected and a properties box provides links to the sources of information.

4 PROTON - Unification Ontology for FactForge

In addition to the above, FactForge also uses an ontology called PROTON (developed by Ontotext) to unify concepts in the main datasets. The PROTON ontology is a lightweight, upper-level ontology serving as a modelling basis for a number of tasks in different domains. PROTON is meant to serve as a seed for ontology generation, i.e. new ontologies constructed by extending PROTON. It can also be used for automatic entity recognition and more generally Information Extraction (IE) from text for the purpose of semantic annotation (metadata generation). The PROTON ontology contains about 500 classes and 150 properties, providing coverage of the general concepts necessary for a wide range of tasks. The design principles can be summarized as follows: (1) domain-independence; (2) light-weight logical definitions; (3) alignment with popular metadata standards; (4) good coverage of named entity types and concrete domains, e.g. people, organizations, locations, numbers, etc.; and (5) good coverage of instance data in Linked Open Data Reason-able views.

The ontology is encoded in a fragment of OWL Lite and split into two modules: Top and Extent. Top module is an upper ontology covering some basic philosophical distinctions between entity types, such as: `Object` – existing entities (agents, locations, vehicles); `Happening` – events and situations; `Abstract` – abstractions that are neither objects nor happenings. The Top module also contains the main classes for each of these types. The Extent module contains more domain and application oriented classes. In FactForge PROTON is used to join the ontological classes and properties of the main datasets. The mapping between PROTON and a given dataset ontology is done in three different ways: (1) using `rdfs:subClassOf` statements between classes in both ontologies and `rdfs:subPropertyOf` for properties; (2) using OWL expressions in the mappings where there is a difference in the conceptualization in both ontologies; and (3) using inference rules in cases where additional individuals are necessary in the repository in order to support the mapping. Only the PROTON ontology is loaded in FactForge. In this way the conceptual structure implied by the particular dataset ontologies is ignored and only the PROTON definitions are presented.

5 Evaluation

Table 1 shows the loading statistics for FactForge datasets. The figures are given in thousands, note ('000) in the header of the columns. The first column lists the datasets loaded. The column "Explicit Indexed Triples" shows the number of explicit facts loaded. The column "Inferred Indexed Triples" presents the number of triples that were generated as a result of the materialization during loading. The column "Total # of Indexed Triples" gives the sum of explicit and implicit triples loaded in OWLIM. The column "Entities" outlines the number of nodes in the graph generated for each dataset, and column "Inferred closure ratio" indicates the number of inferred triples per number of explicit triples loaded.

Dataset	Dataset Statistics				
	Explicit Indexed Triples ('000)	Inferred Indexed Triples ('000)	Total # of Indexed Triples ('000)	Entities ('000 of nodes in the graph)	Inferred closure ratio
Sechmata (RSS, DC) ontologies (geonames)	5	5	10	3	0,9
DBpedia (sameAs)	15 706	0	15 706	24 778	0
NY times	346	550	896	196	1,6
MusicBrainz	198 418	103 757	302 175	54 834	0,5
Lingvoj 2012 + ontology	22	27	49	20	1,2
Lexvo	693	542	1 235	584	0,8
CIA Factbook	40	39	79	24	1
Wordnet	2 724	13 234	15 959	1 081	4,9
Geonames 2.2.1	107 832	194 040	301 872	42 758	1,8
DBpedia core 3.7	659 738	205 602	865 341	155 209	0,3
Freebase	705 161	233 026	938 187	196 947	0,3
Proton	6	637 942	637 948	4	115 297,70
Total	**1 690 691**	**1 388 764**	**3 079 456**	**476 437**	**0,8**

Table 1 Statistics over statements loaded in FactForge

The effects of the materialization and the `owl:sameAs` optimization described in section 2 above result in 79% index compression, which means that close to 12 billion triples that are not indexed are available for querying, making the total of retrievable triples of FactForge close to 15 billion. The loading speed amounts to 8 032 explicit indexed statements per second, and 14 630 indexed statements per second on a CPU - 2 x Intel Xeon X5690, 3.46GHz, 12MB cache, 6 Core, RAM - 144 GB machine.

The utility of reasoning becomes apparent during the evaluation of SPARQL queries. For example the following SPARQL query about Mass media companies in Europe which uses PROTON predicates only:

```
PREFIX ptop: <http://proton.semanticweb.org/protontop#>
PREFIX pext: <http://proton.semanticweb.org/protonext#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
```

```
SELECT * WHERE
       {
        ?Company ptop:locatedIn ?Place ;
                 pext:industryOf dbpedia:Mass_media .
        ?Place   ptop:subRegionOf dbpedia:Europe.
       }
```

returns answers indicating that "Associated Newspapers" is a media company located not only in the United Kingdom, but also in England and in London based on the materialization of the transitive relation `ptop:subRegionOf`.

Furthermore, the queries with formulated with PROTON only return results much faster than queries combining predicates and concepts from different LOD datasets in FactForge, which is due to optimization of the joins traversed.

6 Conclusion

In this paper we presented the inference mechanisms implemented in the OWLIM semantic repositories and their application to a dataset formed by several LOD datasets. The materialization of statements in the closure of the inference rules provides a sound basis for extracting inferred information at query time.

Acknowledgments

This work is partially supported by RENDER FP7-ICT-2009-5, Contract no.: 257790.

References

1. Damova, M., Kiryakov, A., Grinberg, M., Bergman, M., Giasson, F., Simov, K.. Creation and Integration of Reference Ontologies for Efficient LOD Management. In: Semi-Automatic Ontology Development: Processes and Resources, IGI Global, USA, Armando Stellato and Maria Teresa Pazienza (Eds.) 2012.
2. Kiryakov, A; Ognyanov, D; Velkov, R; Tashev, Z; Peikov, I; LDSR: a Reason-able View to the Web of Linked Data, in: SW Challenge (ISWC2009), 2009.
3. ter Horst, H. J. Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity . In Proceedings of The Semantic Web ISWC 2005, LNCS volume 3729 pp. 668–684. Springer Berlin / Heidelberg, 2005.
4. Heim, P; Hellmann, S; Lehmann, J; Lohmann, S; Stegemann, T; (2009) RelFinder: Revealing Relationships in RDF Knowledge Bases. In Semantic Multimedia, volume 5887 of LNCS, pp. 182–187. Springer Berlin/Heidelberg, 2009.