A Multi-ontology Synthetic Benchmark for the Semantic Web

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. One important use case for the Semantic Web is the integration of data across many heterogeneous ontologies. However, most Semantic Web Knowledge Bases are evaluated using the single ontology benchmark such as LUBM and UOBM. Therefore, there is a requirement to develop a benchmark system that is able to evaluate not only single but also federated ontology systems for different uses with different configurations of ontologies. To support such a need, based on our earlier work, we present a multi-ontology synthetic benchmark system that takes a two-level profile as input to generate user-customized ontologies together with related mappings and data sources. Meanwhile, a graph-based query generation algorithm and an *owl:sameAs* generation mechanism are also proposed. By using this benchmark, the Semantic Web systems can be evaluated against complex ontology configurations using the standard metrics of loading time, repository size, query response time and query completeness and soundness.

Keywords: Semantic Web, Benchmark, Web profile, Ontology profile, Ontology

1 Introduction

One of the primary goals of the Semantic Web is to be able to integrate data from diverse ontologies. To support such a need, various federated ontology systems such as KAONP2P [8], Hermes [12] and Semplore [14] have been developed to reason with Semantic Web ontologies, but the standard evaluation of such systems focuses on single ontology by using LUBM [7] - one benchmark system designed to evaluate Semantic Web systems without considering the integration of multi ontologies. Put in other words, the evaluation of multi-ontology knowledge systems by using benchmark without ontology integration considered can not truly reflect the performance of the evaluated systems. Therefore, a benchmark for evaluating multi-ontology systems is required. To our knowledge, except our previous work by Ameet Chitnis et al. [2], no similar benchmarks have been developed till now. In our previous work, we developed a benchmark to support OWLII [11] - a sublanguage of OWL, and distributed sources committing to the generated OWLII ontologies. However, this benchmark suffers from the following deficiencies:

– Because this benchmark only supports OWLII, it cannot scale to work for those users who want to customize their own ontologies to evaluate systems that are sound and complete for other sublanguages of OWL. Therefore, it is not flexible and customizable.
– This benchmark also does not consider the *owl*:*sameAs* statements in the generated data sources. In the real Semantic Web, the *owl*:*sameAs* statements are very common and play the key role in the integration of distributed ontology instances, especially in the Linked Open Data cloud.

To solve the above issues, in this paper, we improved our benchmark to implement a multi-ontology benchmark that makes the generated ontologies and data sources customizable by asking users to provide customization options in the form of a web profile and a set of ontology profiles. The former allows users to customize the distribution of different types of desired ontologies. The latter allows users to customize the expressivity of desired ontologies by making them set the relative frequency of various ontology constructors. Put in other words, the ontology profile provides users a way to define different OWL sublanguages. Unlike previous benchmarks [2], our new 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. Our new benchmark can be used to speculate about similar sized (or larger) scenarios where there are more expressive ontologies and richer mappings. In this paper, we mainly make the following four technical contributions.

– We propose a two-level customization model including ontology profile and web profile for users to describe scenarios required in their evaluations.
– We design and implement a randomized parse-tree construction algorithm to generate ontological axioms directed by the two-level customization model. Thereafter, we generate ontology mapping axioms that conform to the user customized ontologies.
– We design and implement a graph-based query generator. Based on the generated synthetic data sources, our algorithm can construct different query graph patterns. Then, we generate queries by abstracting from these patterns. In this way, we can make each query have at least one answer.
– We implemented an *owl*:*sameAs* generation mechanism that can create *owl*:*sameAs* in proportion consistent with real world data sets.

The remainder of the paper is organized as follows: Section 2 reviews the related work. In Section 3, we describe our new benchmark algorithms. Section 4 presents the methodology for carrying out an experiment and the performance metrics that can be used for evaluation. Finally, in Section 5, we conclude and discuss future work.

2 Related Work

As mentioned before, except our previous work in [2], there is seldom related work similar to our multi-ontology Semantic Web benchmark system. However, we still find some work that helps us to develop our proposed system.

The LUBM [7] 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. Li Ma et al. [10] extended the LUBM to make another benchmark - UOBM so that OWL Lite and OWL DL (except TBox with cyclic definition and Abox with inequality definition) can be supported. However, both LUBM and UOBM use a single domain/ontology namely the university domain comprising students, courses, faculty etc. They did not consider the ontology mapping requirement that are used to integrate distributed domain ontologies in the real Semantic Web. In addition, they also did not consider users' customizability requirement for their individual evaluation purposes. In the real world, different researchers could design different systems supporting different ontology languages. Therefore, for these users, they only need ontologies and data sources that meet their specific evaluation requirements.

T. Gradiner and I. Horrocks [5] developed a system for testing reasoners with available ontologies. This system mainly has two functions. The first is to process ontologies and add them to the library, and the second is to benchmark one or more reasoners using the ontologies in the library. The benefits of this approach mainly include autonomous testing and flexible analysis results. In addition, Ian Horrocks and Patel-Schneider [9] proposed a benchmark suite comprising four kinds of tests: concept satisfiability tests, artificial TBox classification tests, realistic TBox classification tests and synthetic ABox tests. The TBox refers to the intentional knowledge of the domain (similar to an ontology) and the ABox contains extensional knowledge. Meanwhile, Elhaik et al. [3] provided the foundations for generating random TBoxes and ABoxes. The satisfiability tests compute the coherence of large concept expressions without reference to a TBox. However, these approaches neither create OWL ontologies and SPARQL queries nor ontology mappings, and only focus on a single ontology at a time. Also, they did not consider users' customizability requirements.

Garcia-Castro and Gomez-Perez [4] proposed a benchmark suite for primarily evaluating the performance of the methods provided by the WebODE ontology management API. Although their work is very useful in evaluating ontology based tools, it provides less information on benchmarking knowledge base systems. J. Winick and S. Jamin [13] presented an Internet topology generator which creates topologies with more accurate degree distributions and minimum vertex covers as compared to Internet topologies. Connectivity is one of the fundamental characteristics of these topologies. On the other hand, while considering a Semantic Web of ontologies, there could be some ontologies not mapping to any other ontology thereby remaining disconnected from the graph.

3 Benchmark Algorithms

As stated in the introduction, our proposed benchmark algorithm in this paper mainly includes four parts: a two-level customization model consisting of ontology profile and web profile for users to customize ontologies, one randomized parse-tree construction algorithm used to generate ontological axioms, one graph-based query generator and one real world statistics based *owl:sameAs* generation mechanism. Thus, in this part, we first describe the two-level customization model for users to customize ontologies. Here, the sense of user customizability means that we grant the user the freedom to freely configure their individually preferred ontologies. Then, we will discuss our randomized parse-tree ontological axiom generation algorithm. In the third subsection, we will introduce our graph-based query generation algorithm. Finally, we will give our ontology mapping and *owl:sameAs* generation mechanisms.

3.1 Two-level customization model

In the Semantic Web, there are multiple languages used to describe ontologies with different features. OWL [1] is the W3C recommendation for a web ontology language and includes three sublanguages: OWL Lite, OWL DL and OWL Full. In these three sublanguages, OWL DL is the one that most closely corresponds to Description Logics (DL) and broadly used by the semantic web community. There are many Semantic Web systems and reasoners using varying subsets of OWL DL languages. For instance, KAON2 [8] is based on OWL DL without $oneof$ and $hasValue$ constructors. OBII is based on OWLII [11], which is also one subset of OWL DL. Therefore, our benchmark system chooses the set of OWL 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 DL.

To make the generated ontology properly fulfill users' personalization, we design a two-level customization model to support users to customize their own ontologies. First, we allow users to customize the relative frequency of various ontology constructors in the generated ontologies. We call this ontology profile. Basically, each ontology profile corresponds to a user-customized ontology sublanguage. Second, we allow users to customize the distribution of different types of ontologies. We call this web profile. Each entry in the web profile corresponds to one ontology profile. Put in other words, the web profile controls the selection of different ontology profiles during ontology generation. A sample input of the ontology profile and the web profile is shown in Fig.1.

In this sample input, the web profile contains four OWL DL sublanguages: RDFS, OWL Lite, OWL DL and Description Horn Logic (DHL). Their distribution probabilities are set to be 0.4, 0.2, 0.3 and 0.1 respectively. This configuration means that in our final generated ontologies, 40% ontologies use RDFS, 20% ontologies use OWL Lite, 30% ontologies use OWL DL and 10% use DHL. For each ontology profile, the distributions of different ontology constructors used in the generated ontology are displayed. According to the characteristics of every

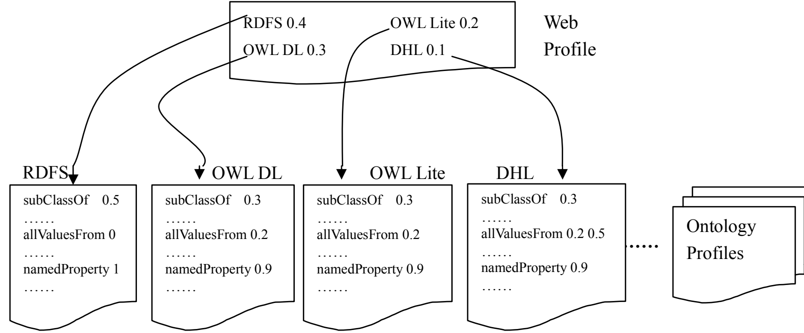

Fig. 1. Two-level customization model.

ontology constructor appearing in ontological axioms, we categorized all OWL DL constructors into three groups: top-level constructors, class constructors and property constructors. Therefore, in each ontology profile, we also let users to fill in three tables with their individual configurations. Each cell of these tables is a number between 0 and 1 inclusive, which means the percentage of this constructor appearing in a generated ontology. Another thing should be noted is that in some ontology languages such as OWLII and DHL, there are different constructor restrictions on the left hand side (LHS) and right hand side (RHS) of an axiom. To support it, users can specify two probabilities for a constructor. One is the probability for the LHS of an axiom and the other is the probability for the RHS of an axiom.

The constructors contained in each table are shown in Table 1. Since each ontological axiom can be seen as a parse tree with the elements in the axiom as nodes, the top-level constructors are those elements can be used as the root. The class and property constructors are those that can be used to create class and property expressions respectively. If we take the constructor analogously as the operator in the math formula, each node on the parse tree can have maximum two operands as children. So, in this perspective, we defined two operands for each constructor as OP1 and OP2. There are five types of operands in total: class type (CT), property type (PT), instance type (IT), named property (NPT) and an integer number (INT). The CT means the operand is either an atomic named class or a complex sub-axiom / sub-tree that has a class constructor as its root. The PT means the operand can be one of constructors listed in the table of property constructors. The NPT means named property. The IT means the operand can be a set of instances. The last type of the operand is an integer number INT which is often used to describe the cardinality restriction.

In this table, first, we should differentiate the $hasValue$ constructor and the $oneOf$ constructor. The $hasValue$ constructor uses a single individual, while the $oneOf$ constructor uses a set of individuals. Second, for cardinality constructors such as $minCardinality$, $maxCardinality$ and $Cardinality$, 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 and each generated value required to be greater than or equal to 1. Finally, our current table does not consider the OWL2 constructors such as property composition. In future work, we plan to support them.

With the inputs of these three tables, our randomized parse-tree construction algorithm will be invoked to create ontological axioms. The details are discussed in section 3.2.

Table 1. Top-level constructors, class constructors and property constructors.

Top-level constructor				Class Constructor			
Constructors	**DL Syntax**	**Op1**	**Op2**	**Constructors**	**DL Syntax**	**Op1**	**Op2**
rdfs:subClassOf	$C1 \sqsubseteq C2$	CT	CT	allValuesFrom	$\forall P.C$	PT	CT
rdfs:subPropertyOf	$P1 \sqsubseteq P2$	PT	PT	someValuesFrom	$\exists P.C$	PT	CT
equivalentClass	$C1 \equiv C2$	CT	CT	intersectionOf	$C1 \sqcap C2$	CT	CT
equivalentProperty	$P1 \equiv P2$	PT	PT	one of	$\{x1,...,x2\}$	IT	
disjointWith	$C1 \sqsubseteq \neg C2$	CT	CT	unionOf	$C1 \sqcup C2$	CT	CT
TransitiveProperty	$P^+ \sqsubseteq P$	NPT		complementOf	$\neg C$	CT	
SymmetricProperty	$P \equiv (P^-)$	NPT		minCardinality	$\geq nP$	PT	INT
FunctionalProperty	$T \sqsubseteq \leq 1P^+$	NPT		maxCardinality	$\leq nP$	PT	INT
InverseFunctionalProperty	$T \sqsubseteq \leq 1P$	NPT		Cardinality		PT	INT
rdfs:domain	$\geq 1P \sqsubseteq C$	NPT	CT	hasValue		PT	IT
rdfs:range	$T \sqsubseteq \forall U.D$	NPT	CT	namedClass			
Property constructor							
inverseOf	P^-	PT		namedProperty			

3.2 Randomized parse-tree construction algorithm

Based on the above three constructor tables in section 3.1, our algorithm randomly constructs one parse tree for each ontological axiom. In this parse tree, the root node can be only selected from the top-level constructor table. Then, each other node can be selected from either the class constructor table or the property constructor table. The tree expansion will be terminated when either all leaf nodes are named constructors including named classes (NC) or named properties (NP) or the depth of the parse tree exceeds the given depth threshold value. The detail steps of constructing this parse tree are as following:

(1) In the beginning, randomly select one constructor from the top-level constructor table as the root node.

(2) According to operand type of the selected root constructor, we will go to the class constructor table or the property constructor table. If the operand is of class type, we will randomly select one class constructor from the class constructor table. If the operand is of property type, we will randomly select one property constructor from the property constructor table.

(3) Repeat step (2) until all leaf nodes in the current parse tree are named constructors or the depth of the parse tree exceeds the given threshold value.

In order to illustrate our algorithm procedure, Fig. 2 shows us an example.

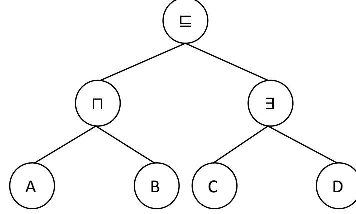

Fig. 2. One randomized parse-tree.

As shown in the this example, we want to generate one parse tree for the axiom of $A \sqcap B \sqsubseteq \exists C.D$. With our algorithm, first, suppose the $rdfs{:}subClassOf$ is selected from the top-level constructor table. Then, we check the operand type of this constructor. According to the top-level table, we know this constructor requires two class constructor operands for both LHS and RHS. Thus, we go to the class constructor table for LHS. Suppose in this step, the $intersectionOf$ is selected. Similar to the first step, we know the $intersectionOf$ needs two class constructor operands and go to the class constructor table again. Then, in this step, two named classes A and B are selected as the operands of the $intersectionOf$. Till now, we complete the construction of the LHS of the given axiom. Next step, we start to process the RHS. Because in the first step, the RHS requires a class constructor, we go to the class constructor table and the $someValuesFrom$ is selected, which needs one property constructor and one class constructor as its operands. Thus, the property constructor table will be searched and one named property C is returned. Then, we go to the class constructor table to get one named class D selected. Till this step, all leaf nodes of the parse tree are named constructors including three named classes (A, B and D) and one named property C. Therefore, our algorithm will terminate and the corresponding axiom will be generated. The pseudo code of this algorithm is displayed in Fig. 3.

As stated in the generation of the parse tree, each axiom takes one top-level constructor as the root and expands the tree by iteratively and randomly selecting constructors from the class constructor table and property constructor table. Therefore, in Algorithm 1, we take one selected top-level constructor *tc*, the class constructor table *ct* and the property constructor table *pt* as inputs. Then, for each operand variable, we judge its type (Lines 3-4 and Line 9). If the variable is of $ClassConstructor$ type, we randomly select one constructor from the class constructor table (Line 6) and then use the selected constructor to generate the corresponding operands (Line 7). At the same time, the tree depth is incremented by one (Line 5). On the other hand, if the variable is of $PropertyConstructor$ type, we randomly select one constructor from the property constructor table (Line 11) and use the selected property constructor to generate the operands

Algorithm 1 Axiom generation

```
function GenerateAxioms(TConstructor tc, CTable ct, CTable pt)
    return: an ontological axiom
    inputs: tc, the selected top-level constructor
            ct, the class constructor table
            pt, the property constructor table
1:  Let operands[] = ∅, operVars[] = set of operand variables of tc,
    axiom = ∅, depth = 0;
2:  while (depth > threshold or tc.leaves typeOf NC or NP) do
3:      for each op ∈ operVars do
4:          if op typeOf ClassConstructor then
5:              depth++
6:              ClassConstructor cc = selectConstructor(ct)
7:              operands[op] = GenerateAxioms(cc, ct, pt)
8:          else
9:              if op typeOf PropertyConstructor then
10:                 depth++
11:                 PropertyConstructor pc = selectConstructor(pt)
12:                 operands[op] =  GenerateAxioms(pc, ct, pt)
13: axiom = makeAxiom(tc, operands[])
14: return axiom
```

Fig. 3. Randomized parse-tree construction algorithm.

(Line 12). Also, the depth is incremented by one(Line 10). When all leaf nodes are type of NC or NP, or the depth exceeds the threshold, our construction will stop (Line 2). Then, we will combine tc with the obtained operands to construct one ontological axiom (Line 13). Finally, the constructed axiom is returned (Line 14).

3.3 Graph-based query generation algorithm

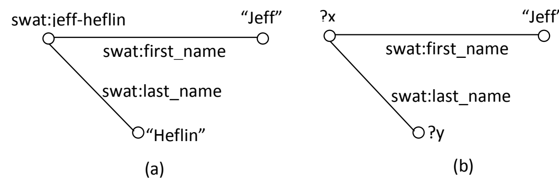

Fig. 4. Query graph.

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 basically modeled as one big graph. Then, each SPARQL query is basically one subgraph over this

big graph and different subgraphs form different query patterns that can be used to generate queries. In our algorithm, after query patterns determined, we need to replace some selected node values with query variables. In this process, if the junction node of the query pattern is replaced with one query variable, then this variable would be counted as the join variable in our final generated queries. To illustrate this process, we can give an example. Suppose we have an initial graph shown in Fig. 4(a). With this graph, we could replace *swat*:$jeff$-$heflin$ and "$Heflin$" with the variables ?x and ?y respectively shown in Fig. 4(b). Then, we could get the following $where$ clause of the generated SPARQL query:

⟨?x swat:first_name "jeff"⟩
⟨?x swat:last_name ?y⟩

Algorithm 2 Query generation

function GenerateQueries(KnowledgeBase *KB*, int *numQTP*)
  return: a SPARQL query
  inputs: *KB*, the given Semantic Web Knowledge Base
    numQTP, # of query triple patterns in the generated query
1: Let *initialGraph* = extractSubgraph(*KB*), *queryGraph* = {}
2: Let *start* = randomly select one node from *intialGraph*
3: add(*queryGraph*, *start*)
4: **while**(numEdges(*queryGraph*) < *numQTP*) **do**
5:   Randomly select one edge *Edge* starting from "*start*" within *initialGraph*
6:   add(*queryGraph*, *Edge*)
7:   add(*queryGraph*, *the ending node "end" of Edge*)
8:   Replace "*end*" with a variable in probability *P*
9:   Randomly select one node from queryGraph and assign it to "*start*"
10: **for each** edge *e* in *queryGraph* **do**
11:   **if**(hasNoVars(*e*)) **then**
12:     Replace the junction node of *e* with a variable
13: Let *sparqlquery* = formQuery(*queryGraph*)
14: **return** *sparqlquery*

Fig. 5. Graph-based query generation algorithm.

The algorithm is displayed in Fig. 5. According to this algorithm, first, we construct a large enough subgraph *intialGraph* with the number of nodes being much greater than the given number of query triple patterns in the final generated query (Line 1). The reason of constructing an *initialGraph* is that the size of the whole KB is often too big to model a graph for it. Therefore, an *initialGraph* helps us to create a set of query graph patterns that can be used to generate synthetic queries by providing a subset of knowledge in the whole KB. Then, we randomly select one node *start* from *intialGraph* as the starting node to construct a query pattern graph *queryGraph* (Lines 2 and 3). Begin with *start*, we randomly select one edge starting with *start* and add this edge with its ending node *end* into the *queryGraph* (Lines 5, 6 and 7). Then, we replace

end with a new variable in the probability P (Line 8) and update *start* (Line 9). This process is iterated until the *queryGraph* includes $numQTP$ edges (Line 4). By this step, we successfully constructed one query pattern graph. Next step, we need to check if each edge e in *queryGraph* contains at least one variable (Lines 10 and 11). If not, we need to replace the junction node of e with a new variable (Line 12). The junction node means the node shared by at least two edges in *queryGraph*. With the variable-assigned *queryGraph*, a SPARQL query can be generated and returned (Lines 13 and 14).

3.4 Mapping ontologies and *owl:sameAs* generation

The mapping generation mechanism is basically the same as that of our earlier work in [2]. We still model mappings into a directed graph of interlinked ontologies, where every edge is a map from a source ontology to a target ontology. During mapping construction, in order to guarantee the mapping connectivity and termination, before the mapping creation, we determine the number of terminal nodes and randomly mark those many domain ontologies as terminal. This way prevents a non-terminal node from attaining a zero out-degree and maintain the connectivity. More details can be found in [2].

The source generation is also basically the same as that of our work in [2] except the new function of *owl:sameAs* generation. With our current configuration, the average number of triples for each data source is around 50, which is obtained from our statistics of the real Semantic Web data by using Sindice [1] - a well-known semantic web index. The new *owl:sameAs* generation is also based on the same statistic results. In our statistics, we randomly issued one term query to Sindice and took its top 1000 returned sources as our samples. In these 1000 sources, 27.1% of them contain *owl:sameAs* statements. Furthermore, in all data sources with *owl:sameAs* statements, around 20% of them have *owl:sameAs* added, removed and updated and around 7% of them have no change of *owl:sameAs* statements. Based on these results, we have the following conclusions about *owl:sameAs*:

- In the real semantic web, *owl:sameAs* always exists. Therefore, a benchmark system for the Semantic Web should generate *owl:sameAs* statements in some proportion.
- Because the *owl:sameAs* statements could be added, removed and updated in the real semantic web, a benchmark system for the Semantic Web should also support the *owl:sameAs* change.

Currently, we only support the first point. As for the second, we plan to implement it in our future work. For the current configuration of the given two-level customization, the proportion of the data sources containing *owl:sameAs* statements is roughly at the level of 27.1%, which is implemented by applying a uniform distribution to generated RDF triples.

[1] http://sindice.com/

4 Experimental Methodology

In this part, we present our methodology of setting up an experiment for a multi-ontology Semantic Web system and also the performance metrics that could be used for the evaluation.

As mentioned in section 3.1, we ask users to provide two kinds of configuration profiles: web profile and ontology profile. In each ontology profile, users need to fill in three tables: the top-level constructor table, the class constructor table and the property constructor table. The sum of the probabilities in each table of the ontology profile should be 1. Otherwise, we will normalize them. Based on the given setting information, the user customized ontologies, ontology mappings and the related sources are generated.

With the synthetic data set, metrics such as Loading Time, Repository Size, Query Response Time, Query Completeness and Soundness could serve as good candidates for the system performance evaluation.

- Loading Time: This could be calculated as the time taken to load the Semantic Web space: domain and map ontologies and the selected data sources.
- Repository Size: This refers to the resulting size of the repository after loading the benchmark data into the system. Size is only measured for systems with persistent storage and is calculated as the total size of all files that constitute the repository. Instead of specifying the occupied disk space we could express it in terms of the configuration size.
- Query Response Time: We recommend this to be based on the process used in database benchmarks where every query is consecutively executed on the repository for 10 times and then the average response time is calculated.
- Query Completeness and Soundness: With respect to queries we say a system is complete if it generates all answers which are entailed by the knowledge base. However, on the Semantic Web partial answers will also be acceptable and hence we measure the degree of completeness of each query as a percentage of the entailed answers that are returned by the system. Similarly we measure the degree of soundness of each query as the percentage of the answers returned by the system that are actually entailed. On small data configurations, the reference set for query answers can be calculated by using state of the art DL reasoners like Racer and FaCT. For large configurations we can use partitioning techniques such as those of Guo and Heflin [6].

5 Conclusions and Future Work

In this paper, based on our earlier work [2], we proposed a multi-ontology benchmark for the Semantic Web systems. This benchmark takes a two-level customization model including the web profile and the ontology profile as its inputs and generates user customized ontologies. At the same time, it can also generate reasonable SPARQL queries for users and *owl:sameAs* statements in distributed semantic data sources. By using this system, different ontology expert users can configure their preferred ontologies to evaluate their proposed work.

However, there is still significant room for improvement. First, we need to consider the inconsistency process in our ontology generation in future work. To prevent this, we could check the ontology after generation, throw away the whole ontology if inconsistent or use a reasoner that's capable of pinpointing inconsistencies and make the minimal change to make the ontology consistent. We could also check after adding each axiom, and throw away the axiom if it makes the ontology inconsistent. Second, our benchmark system does not consider the features of the OWL 2 language, but the randomized parse-tree construction algorithm can be easily adapted to support OWL 2's new features. Finally, the *owl:sameAs* generation mechanism in our current benchmark system does not support the *owl:sameAs* update functions, which are proved to exist in real semantic web data according to our statistics. Therefore, our future work also includes improving the *owl:sameAs* generation mechanism of our system.

References

1. Web Ontology Language (OWL). Website, 2002. `http://www.w3.org/TR/2002/WD-owl-ref-20021112/`.
2. A. Chitnis, A. Qasem, and J. Heflin. Benchmarking reasoners for multi-ontology applications. In *EON*, pages 21–30, 2007.
3. Q. Elhaik, M. christine Rousset, and B. Ycart. Generating random benchmarks for description logics. In *In Proceedings of DL'98*, 1998.
4. R. Garca-castro and A. Gmez-prez. A benchmark suite for evaluating the performance of the webode ontology engineering platform. In *In Proc. of the 3 rd International Workshop on Evaluation of Ontology-based Tools*, 2004.
5. T. Gardiner, I. Horrocks, and D. Tsarkov. Automated benchmarking of description logic reasoners. In *Description Logics*, 2006.
6. Y. Guo and J. Heflin. Document-centric query answering for the semantic web. In *Web Intelligence*, pages 409–415, 2007.
7. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
8. P. Haase and Y. Wang. A decentralized infrastructure for query answering over distributed ontologies. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1351–1356, New York, NY, USA, 2007. ACM.
9. I. Horrocks and P. F. Patel-Schneider. Dl systems comparison (summary relation). In *Description Logics*, 1998.
10. 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.
11. A. Qasem, D. A. Dimitrov, and J. Heflin. Efficient selection and integration of data sources for answering semantic web queries. *International Conference on Semantic Computing*, pages 245–252, 2008.
12. T. Tran, H. Wang, and P. Haase. Hermes: Data web search on a pay-as-you-go integration infrastructure. *Web Semantics*, 7(3):189–203, 2009.
13. J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, EECS, University of Michigan, 2002.
14. L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, and Y. Yu. Semplore: An IR approach to scalable hybrid query of semantic web data. In *ISWC/ASWC*, pages 652–665, 2007.