

Towards Incremental Reasoning Through Updates in OWL-DL

Bijan Parsia, Christian Halaschek-Wiener, Evren Sirin
Maryland Information and Network Dynamics Lab.
8400 Baltimore Av.
College Park, MD, 20740 USA
bparsia@isr.umd.edu, {halasche, evren}@cs.umd.edu

ABSTRACT

In this paper, we investigate incrementally updating classifications of ontologies encoded in the expressive description logic $SHOIN(\mathcal{D})$, which also corresponds to the W3C standard Web Ontology Language, OWL-DL. In particular we present various optimizations for incremental classification under both axiom addition and removal. Several of the optimizations are independent of the logic and proof theory as long as classification can be reduced to satisfiability testing. Additionally, we provide an empirical analysis of the optimizations through an experimental implementation in the Pellet OWL-DL reasoner.

Keywords: Description Logics, Information Change, Ontology, Classification

1. INTRODUCTION

Core reasoning services of description logic reasoners include consistency checking, classification and realization. Until recently, these reasoning services have only been considered for static knowledge bases. However, there exists a variety of use cases which require frequent updates at both the terminological and assertional levels. For the purpose of this work, we restrict ourselves to the task of incremental classification through update.

Classification is the determination for every two named classes in an ontology, A and B , whether $A \sqsubseteq B$ [1], that is whether A subsumes B and vice versa. Classification involves, in the worst case n^2 subsumption tests for n named classes. Since subsumption tests are typically very expensive, the key to optimizing classification is to avoid unnecessary tests and to substitute cheap (sound, but possibly incomplete) tests whenever possible [6]. Modern description logic reasoners can classify large and complex ontologies in reasonable, but noticeable, time. Today's reasoners do not however provide optimized reasoning services for updates. This can be quite problematic as there are a number of situations where the ontology itself is in flux:

- Perhaps the single most common use of description logic reasoners is in ontology editors. Most editors do not do continuous reasoning while one is editing (though see [8]), relying on an analogue of the edit-

compile-test loop of most programming environments. However, if this cycle is very long (e.g., hundreds of seconds) then users will be forced to perform larger sets of edits before testing. This discourages experimentation, particularly in debugging contexts.

- Prominent web services frameworks (e.g., OWL-S) use description logics for service discovery and matchmaking [12, 13, 10]. Services, especially device services in pervasive contexts, may register or deregister their descriptions (and supporting ontologies) quite rapidly, yet the matchmaking service must remain responsive.
- Systems which attempt to learn concepts from fluctuating data, such as news feeds processed with NLP techniques or sensor streams will constantly update their target ontologies. Not only is responsiveness an issue, but so too is the overall time the system spends in processing the ontology.
- Semantic Web portals often allow content authors to modify or extend the ontologies which organize their site structure or page content. While in some cases, a “defer update” strategy is acceptable, in general it is more gratifying to users if changes they make are reflected more immediately in the site. (This is a variant of the editor case.)

In all these cases, it is convenient to think of the classification as a *view* of the ontology, and so, the problem of maintaining the ontology through update as a *view maintenance*. Given that significant inference is performed, it is also a *truth maintenance* problem, but where the set of inferences to be maintained is a specific subset of the entailments of the ontology.

In this paper, we investigate incrementally updating classifications of ontologies encoded in the expressive description logic $SHOIN(\mathcal{D})$, which also corresponds to the W3C standard Web Ontology Language, OWL-DL. Several of the optimizations we present are independent of the logic and proof theory as long as classification can be reduced to satisfiability testing. We provide an empirical analysis of the benefit of these optimizations via a preliminary implementation in the Pellet OWL DL reasoner. Additionally, the optimizations presented are directly addressable to the task of realization. While we do not address consistency checking through update, we note that we are currently devising an approach for updating ABox completion graphs, which is showing promising results.

2. BACKGROUND

Classification is the process of determining, for any two named classes in an ontology (say, A and B), whether a *subsumption* relation holds between the classes in either direction, that is, whether $A \sqsubseteq B$ or $B \sqsubseteq A$ or both. In this paper, we will presume that the subsumption test itself is reduced to the concept satisfiability problem and performed by a tableau based reasoning system. For the reduction, we observe that $A \sqsubseteq B$ holds just in case the concept $A \sqcap \neg B$ is *unsatisfiable*. In general, optimizations for classification are independent of the way of determining concept satisfiability, but, as we discuss below, several make use of specific information derivable from the tableau. However, although we focus on the description logic $SHOIN(\mathcal{D})$ nothing depends on the specific choice of logic.

The naive classification algorithm is simple: perform the pair of subsumption tests as satisfiability tests for each pair of concepts. This results in n^2 satisfiability tests for n concepts in the ontology. Given the extremely high worst case complexity, and the in practice actual performance, of satisfiability testing, this is infeasible. Indeed, just the quadratic behavior is daunting even with small ontologies (say, of 100 classes) and at the moment there are active, developed, and used ontologies with tens of thousands of concepts.

There are two ways to speed up classification: Improve the speed of the subsumption tests or simply try to avoid those tests. For speeding up subsumption tests, one can either improve the speed of satisfiability checking, or one can substitute cheaper test that may be incomplete, only falling back on a full satisfiability test in certain circumstances. Modern DL reasoners (such as [15, 11, 14]) make use of all these approaches and achieve very good overall classification performance.

2.1 Avoiding tests

Current techniques for avoiding subsumption tests generally include using Top-Bottom search [6] and avoiding obvious subsumptions test that will not hold. In the former, classification is implemented using an optimized traversal algorithm that utilizes the concept hierarchy to reduce the number of subsumption tests. Instead of directly checking the subclass relation between *every* named class, first the concept's subsumers are computed by searching down the class hierarchy. Following this, the concepts subsumees are determined by searching up the hierarchy. The top down subsumer search exploits the transitivity of the subclass relationship by propagating failed results down the hierarchy, thus avoiding subsumptions tests between subclasses of known non-subsumers. In a similar manner, the bottom up approach for determining concept subsumees, only performs a subsumption test between two classes A and B , if it is known that A subsumes all concepts that are known to subsume B [6]. Additionally, in general, obvious non-subsumptions tests are avoided as well. For example, in the case of explicitly told disjoint classes, subsumption tests between such classes can be avoided. Critically, the classification is done incrementally, in which results from earlier tests (e.g., derived disjointness) are used to reduce the subsequent number of necessary tests. However, while the advantages of incremental classification *within* a given classification have been observed and exploited, there has been little attempt to deal with incremental classification through updates to the ontology.

2.2 Cheaper tests

In [5] the notion of *model caching* (or, as it is known in [4] “pseudo model” caching) was proposed. [4] builds upon this work to use these pseudo models to avoid subsumption tests. In the approach, pseudo models are extracted during ABox consistency test. In tableau systems, consistency checking involves building a structure, a *tableau*, from which, a model of the ABox may be extracted. The proof procedure is guaranteed to build a model if there is one, so failure to build a model corresponds to the absence of any model, and thus, inconsistency. These structures may be quite large, so extracting an entire model is infeasible (or impossible, if the model is infinite, though, of course, there is a natural finite representation, the tableau itself). Building these structures simply *is* what is expensive about the consistency test, so its natural to want to reuse that work. One way built models can be reused is for subsumption testing. If a model of A and model of $\neg B$ can be combined into a single model, then their conjunction is satisfiable. This is the basis of the pseudo model mergability test. Instead of merging (and thus having to cache) entire models, one merges just the root individuals of the model (and their type assertions). Much of the time, this simpler test is sufficient to determine non-subsumption, and when the merge fails one falls back on the normal tableau building process. While this approach is incomplete, it is sound; if the two pseudo models are mergable, then it can be assumed that there exists a model for the conjunction. Since most subsumption tests are negative, the pseudo model optimization dramatically improves classification time.

3. CLASSIFICATION MAINTENANCE

The following sections propose novel optimizations to currently developed classification techniques. The first section presents optimizations in the event of axiom additions; the second in the event of axiom deletions. We treat revisions as being reduced to a deletion plus an addition, and do not otherwise treat them specially.¹ In this work, we consider only monotonic description logic (in particular, we work with $SHOIN(\mathbf{D})$, a.k.a., OWL DL), but for many of our optimization, sheer monotonicity is key).

3.1 Incremental Axiom Additions

Given monotonicity, no addition can produce the retraction of an established conclusion. So there is never a need to recheck any previously held subsumption relationship after an addition. We exploit this basic fact by maintaining a cache of all previously known subsumptions when an axiom is added to the ontology instead of recomputing them. In ontologies in which there are large numbers of classes, such as Galen², this optimization can be quite beneficial, as the previously known subsumption tests will be avoided altogether. In Galen, for example, there are approximately 1,655 subsumptions that take .09 ms (on average) each to be tested using Pellet.

¹While correct, clearly this is not optimal. For example, by refactoring an axiom in a set of logical equivalent smaller axioms, one might confine the revision to a subpart of the axiom, which might be easier to process. Many revisions, then, might reduce to deletions of that subpart.

²Galen ontology. Available at <http://protege.stanford.edu/plugins/owl/owl-library/not-galen.owl>

As discussed in Section 2, pseudo models are built when checking the satisfiability of classes, which is done prior to classification. Later, more expensive subsumption tests can be avoided using pseudo model merging [4]. Though these tests are quite cheap, a classification cycle can involve thousands or tens of thousands of them. However, after an addition, all the tests where the pseudo models have not changed can be avoided, since, obviously, their merge will have the same result. In particular, consider two classes, A and B , in which $A \sqsubseteq B$ when the ontology is previously classified. In the event that an axiom is added, if the pseudo models of A and $\neg B$ have not changed, it can be concluded that $A \sqsubseteq B$ after the axiom is added. Since, if we know that $A \sqsubseteq B$ from the previous classification, then it is known that $A \sqcap \neg B$ is satisfiable, thus A and $\neg B$ have mergable pseudo models. Upon an axiom addition, the new pseudo models of A and $\neg B$ can be compared with the previously cached copies, and if they are the same, the merge test does not have to be redone. We have found that in large ontologies, this optimization can be quite beneficial. For example, if a random axiom is removed from the Tambis³ ontology and then it is added back to the ontology, simulating an axiom addition, there are approximately 45,061 subsumption tests which are false (taking approximately 495 ms). Using this optimization when incrementally classifying Tambis, on average 61.6% of the non-subsumption tests can be avoided (see Section 4).

3.2 Incremental Axiom Deletions

As in Section 3.1, the monotonicity of the description logic considered here can also lead to optimizations of classification under incremental removal of axioms. In the event that an axiom is removed from an ontology, by monotonicity, all previous non-subsumptions remain unchanged. Therefore in an incremental approach, all previous subsumption tests that did not hold can be avoided. While this optimization is quite straightforward, we note that it results in dramatic reduction in the number of tests, however cheap or expensive, performed. For example, our experiments show that if a random axiom is removed from SWEET-JPL⁴ and the ontology is reclassified, on average 953,397 subsumption tests are performed that are false, taking a total of 7,722 ms (on average). By caching these non-subsumptions from a previous classification, all these operations can be avoided.

The second optimization presented here relates to determining whether a removed axiom may retract some previously found subsumption between two classes. In [7], the problem of finding the *Set of Support* (SOS) for an unsatisfiable class has been tackled. By tracing the effect of axioms in producing clashes in the tableau completion graph, one can determine a set of axioms which are sufficient to entail the unsatisfiability of a class. Since a positive subsumption test corresponds to the unsatisfiability of a concept, we can use axiom tracing to find a set of support for that subsumption. Thus, if the axiom removed does not appear in the SOS of a previously known subsumption, then the test can be avoided. To clarify, given two classes A and B , where $A \sqsubseteq B$, the SOS of $A \sqcap \neg B$ can be cached (if $A \sqsubseteq B$, then

³Tambis ontology. Available at <http://protege.stanford.edu/plugins/owl/owl-library/tambis-full.owl>

⁴SWEET-JPL ontology. Available at <http://sweet.jpl.nasa.gov/ontology/earthrealm.owl>

$A \sqcap \neg B$ must be unsatisfiable, thus having some SOS). If an axiom is then removed from the KB and that axiom is not contained in the SOS for $A \sqcap \neg B$, then the subsumption test for $A \sqsubseteq B$ can be filtered. This is evident because all axioms that lead to the unsatisfiability of $A \sqcap \neg B$ will still hold in the KB.

Critical to this optimization is the fact that axiom tracing to the “last first” clash has been shown to involve only a slight bit of memory overhead to normal reasoning [7]. So, it is cheap enough to add SOS caching. Of course, since we only cache a single SOS, it’s possible that we miss opportunities to avoid recomputations. Since an SOS is only sufficient, damaging does not guarantee that the subsumption is voided. If we were to cache *all* the SOS, then we could check whether an axiom from each of them has been removed. In that case, the subsumption can be withdrawn without retesting. The single SOS caching is sufficient to get some improvement. Further experimentation awaits a reasonably efficient algorithm for finding all the SOSs.⁵ Note that even if finding all of the SOSs is quite expensive, there are certainly applications where even small speedups through update is worth a long start up cost (e.g., editing).

4. EVALUATION

In order to evaluate the classification optimizations presented in this work, we did a preliminary implementation of the techniques in the Pellet OWL DL reasoner [11]. For our experiments we used four OWL ontologies of varying complexity and expressivity, namely Koala⁶, Tambis, SWEET-JPL, and Galen.

In the first series of experiments, all four ontologies were iteratively deconstructed one axiom at a time; that is, in each iteration, a random axiom was removed and the resulting ontology was reclassified by both the optimized and regular versions of the reasoner. All results presented here have been *averaged* over all of the iterations. Table 1 presents the (average) total classification time when incrementally *removing* axioms (including the percentage difference between normal and incremental classification). As shown, the results are quite promising: in two of the cases, the total classification time under incremental removal is decreased by over 50%.

Name	C / P / I	Opt.Clas.	Reg.Clas.	% Inc.
Koala	20 / 5 / 6	0.037	0.046	20.0%
Tambis	392 / 100 / 0	0.81	2.40	66.2%
SWEET	1400 / 137 / 110	8.80	14.90	40.9%
Galen	2749 / 413 / 0	43.60	98.70	55.8%

Table 1: Optimized versus regular Pellet classification performance for iterative axiom removal from commonly used ontologies of varying complexity and expressivity ($ALCON(\mathcal{D})$, $SHIN(\mathcal{D})$, $SHOLF(\mathcal{D})$, and SHF respectively). All times are shown in seconds. Note C/P/I represents Classes, Properties, Individuals respectively.

The average number of (non) subsumption tests avoided in the optimized reasoner is shown in Table 2 (note here

⁵We expect to have one soon.

⁶Koala ontology. Available at <http://protege.stanford.edu/plugins/owl/owl-library/koala.owl>

that they are shown in comparison with the total number of (non) subsumption tests). It is interesting to note that the number of false subsumption tests (non-subsumptions) greatly exceeds those that in fact hold. This provides some insight into the overall performance improvement under axiom removal. Because the previous non-subsumption tests can entirely avoided due to monotonicity, on average 99.12% of such test are filtered, if cached from the previous classification. Thus, less expensive cache lookups can be performed (as shown in Table refigure:removal-avoidance-time) resulting in clear performance benefits.

Name	Subsumptions		Non-subsumptions	
	Average	Total	Average	Total
Koala	2.3	5.2	293	300
Tambis	26.3	101.5	91912	92024
SWEET-JPL	5	29	952466	953397
Galen	613	705	3438438	3463105

Table 2: Average subsumption and non-subsumption test avoidance under axiom removal when using optimizations.

The effectiveness of SOS caching is demonstrated here as well. In the case of Galen, on average 86% of subsumption tests which are true can be determined via a cache lookup, rather than a more expensive satisfiability check (shown in Table 3). While the performance improvement resulting from the SOS cache was greatest for the Galen ontology, on average 42.75% of subsumption test which hold can be retrieved using the more efficient cache lookup presented here.

Name	Subsumptions		Non-subsumptions	
	Opt.	Reg.	Opt.	Reg.
Koala	.20	.26	.009	.033
Tambis	.076	.19	.0018	.0071
SWEET-JPL	.36	1.03	.0045	.0081
Galen	.043	.91	.007	.02

Table 3: Average individual subsumption and non-subsumption optimization lookup time versus regular (non) subsumption test time under axiom removal. All times are shown in milliseconds.

In the second series of experiments, all four ontologies were iteratively constructed one axiom at a time; that is, each ontology started without any axioms. Then in an iterative manner, a random axiom was added and the resulting ontology was reclassified by both the optimized and regular versions of the reasoner. Again, all results presented here have been averaged over all of the iterations. Table 4 present the (average) total classification time when incrementally adding axioms. Again the results are quite promising as in two of the cases, the total classification time under incremental addition is decreased by over 20%.

The number of (non) subsumption test avoided on average in the optimized reasoner is shown in Table 5. Also, the average time for each (non) subsumption test avoided in the optimized reasoner is provided in Table 6. When comparing these results with that of incremental removal, it is apparent that the performance improvements here are not as drastic. This is because fewer non-subsumptions can be retrieved using a caching technique. While under axiom deletion monotonicity provides that all previous non-subsumptions will remain as such, under axiom addition this

Name	Opt. Clas.	Reg. Clas.	% Inc.
Koala	0.035	0.039	10.2%
Tambis	.853	.951	10.3%
SWEET-JPL	0.475	.619	23.2%
Galen	52.69	68.08	22.6%

Table 4: Optimized versus regular Pellet classification performance for iterative axiom addition from commonly used ontologies. All times are shown in seconds.

is not the case. However, we do note that a substantial portion of the non-subsumption test can be retrieved using pseudo model cache comparisons; on average 58.5% of the non-subsumptions were retrieved using the optimized technique with reasonable performance improvements (shown in Table 6).

Monotonicity does however, allow all previous known subsumptions to be assumed under axiom addition. The evaluation provided here clearly demonstrates the effectiveness of this, as the cache lookup is orders of magnitude cheaper than performing the satisfiability check in all cases.

Name	Subsumptions		Non-subsumptions	
	Avd.	Total	Sub Avd.	Total
Koala	2.55	5.24	293.3	300.26
Tambis	109	111	17272	45061
SWEET-JPL	4.98	5.00	22202	30554
Galen	1485	1655	1675287	6077843

Table 5: Average subsumption and non-subsumption test avoidance under axiom addition when using optimizations.

Name	Subsumptions		Non-subsumptions	
	Opt.	Reg.	Opt.	Reg.
Koala	.0005	.26	.0097	.035
Tambis	.0001	.117	.002	.011
SWEET-JPL	.0002	.72	.0034	.0072
Galen	.0045	.091	.002	.0063

Table 6: Average individual subsumption and non-subsumption optimization lookup time versus regular (non) subsumption test time under axiom addition. All times are shown in milliseconds.

While we see promising improvements from these optimizations, it is clear that they are unlikely to satisfy the performance requirements in our use cases. While the time to incrementally classify Tambis under deletion is lowered to under 1 second, Galen still takes approximately 48 seconds. In the case of ontology editors, for instance, a near real time reclassification is desirable. So, additional optimizations are needed. For example, in the context of ontology editors, other aspects of the current state of the editor can be exploited. In the event that only a subset of the class hierarchy is physically visible to the user (or otherwise the subject of user attention), only a small fraction of the classes need to be reclassified immediately. In Galen, which contains thousands of classes, the number of immediately reclassified classes would be reduced drastically. In cases in which this type of *lazy* classification is applicable, it is likely that the number of avoided tests using existing and the currently proposed optimizations would be sufficient.

5. PERSISTING AND SHARING DL VIEWS

Given that caching subsumptions, nonsubsumptions, sets of support, and pseudomodels is fairly effective, we have created a simple XML format to serialize these structures both for reuse between sessions of a reasoner, and for sharing between reasoners of different capabilities. For example, in pervasive environments (such as [9]) there are typically a large number of devices with weak computational abilities (e.g., cell phones, watches, or PDAs). To do sound and complete reasoning over large ontologies (such as a set of service descriptions) would be beyond their capabilities. But instead of requiring them to outsource *all* their reasoning to an available server, we can implement a smaller, weaker reasoner that exploits the classification graph, pseudo models, and other information we serialize. Then, only the tests that cannot be correctly done by these methods need involve a call to a reasoner across the networks. One question we shall investigate is whether sharing more information, e.g., all sets of support or more and more complex pseudo models, in order to further reduce the need to call a full reasoner interactive is worth the cost.

6. RELATED WORK

Recently, there has been work in optimizing various facets of description logic reasoning tasks. One recent attempt to improve pseudo model merging is [2], which proposes persisting pseudo models between runs of the reasoner in a relational database. Additionally, pseudo model merging is also provided in the database (specifically using SQL queries) in an attempt to exploit the various indexing and retrieval optimizations of modern RDBMS. Our work here differs in that we attempt to avoid the subsumption tests all together. Further the approach presented in Section 5 provides an additional mechanism to persistently cache pseudo models.

In [3], a variety of optimizations are provided for ABox inferences in DL reasoners (specifically those that are Tableau based). Among others, in this work the authors present a dependency-based instance retrieval optimization in which particular clashes in tableau branches are cached and used to filter potential individuals for future queries. Our work here differs in general in that we consider TBox optimization, specifically for concept classification.

7. CONCLUSION

Modern description logic reasoners are traditionally oriented toward classifying relatively static ontologies. In fact, however, there are numerous situations in which the ontology itself is in flux, requiring continuous or relatively frequent classification. Given this, we see a need for further optimization techniques for handling incremental axiom addition and removal, providing more responsive classification procedures for dynamic environments.

In this paper, we have proposed a variety of classification optimizations for incremental updates to ontologies encoded in the expressive description logic $\mathcal{SHOIN}(\mathcal{D})$. These approaches exploit the monotonicity of the language and make use of caching *sets of support* and *pseudo models* from prior classifications. Several of the optimizations are independent of the logic and proof theory as long as classification can be reduced to satisfiability testing.

We have provided an empirical analysis of the techniques through an experimental implementation in the Pellet rea-

soner. Our initial results are quite promising and validate the approach. These and similar optimizations can be directly applied to realizing the knowledge base (that is, finding all the most specific types of each individual). Our preliminary experiments indicate that even larger speed improvements are to be had for realization. While this work has focus on techniques for avoiding subsumption tests between classes, future work will include exploring the use of previous knowledge from classification to speedup satisfiability testing itself.

8. REFERENCES

- [1] F. Baader and W. Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.
- [2] C. Chen, V. Haarslev, and J. Wang. Las: Extending racer by a large abox store. In *Proc. of the Int. Description Logic Workshop (DL 2005)*, pages 41–50, 2004.
- [3] Volker Haarslev and Ralf Moller. Optimization techniques for retrieving resources described in owl/rdf documents: First results. In *Proc. of the Int. Description Logic Workshop (DL 2004)*, 2004.
- [4] Volker Haarslev, Ralf Moller, and Anni-Yasmin Turhan. Exploiting pseudo models for tbox and abox reasoning in expressive description logics. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 61–75, London, UK, 2001. Springer-Verlag.
- [5] I. Horrocks. *Optimising Tableau Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.
- [6] I. Horrocks. Implementation and optimisation techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 313–355. Cambridge University Press, 2003.
- [7] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. In *Journal of Web Semantics - Special Issue of the Semantic Web Track of WWW2005*, 2005. (To Appear).
- [8] T. Liebig and O. Noppens. Ontotrack: Combining browsing and editing with reasoning and explaining for owl lite ontologies. In *Proceedings of the 3rd International Semantic Web Conference (ISWC) 2004*, Japan, November 2004.
- [9] Ryusuke Masuoka, Bijan Parsia, and Yannis Labrou. Task computing - the semantic web meets pervasive computing -. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, October 2003.
- [10] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *The First International Semantic Web Conference*, 2002.
- [11] Bijan Parsia and Evren Sirin. Pellet: An owl dl reasoner. In *Proc. of the Int. Third International*

Semantic Web Conference (ISWC 2004) - Poster, 2004.

- [12] Evren Sirin, Bijan Parsia, and James Hendler. Composition-driven filtering and selection of semantic web services. *IEEE Intelligent Systems*, 19(4):42–49, 2004.
- [13] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1(1):27–46, 2003.
- [14] Dmitry Tsarkov and Ian Horrocks. Efficient reasoning with range and domain constraints. In *Proc. of the Int. Description Logic Workshop (DL 2004)*, pages 41–50, 2004.
- [15] V.Haarslev and R.Moeller. Racer system description. In *Proc. of the Joint Conf. on Automated Reasoning (IJCAR 2001). Volume 2083 of Lecture Notes in Artificial Intelligence, pages 701-705*, 2001.