A purely logic-based approach to approximate matching of Semantic Web Services

Jörg Schönfisch[12], Willy Chen[12], and Heiner Stuckenschmidt[2]

[1] SysTec-CAx GmbH, München, Germany
{joerg.schoenfisch,willy.chen}@systec-cax.de
http://www.systec-cax.de

[2] KR & KM Research Group, University of Mannheim, Germany
heiner@informatik.uni-mannheim.de
http://ki.informatik.uni-mannheim.de

Abstract. Most current approaches to matchmaking of semantic Web services utilize hybrid strategies consisting of logic- and non-logic-based similarity measures (or even no logic-based similarity at all). This is mainly due to pure logic-based matchers achieving a good precision, but very low recall values. We present a purely logic-based matcher implementation based on approximate subsumption and extend this approach to take additional information about the taxonomy of the background ontology into account. Our aim is to provide a purely logic-based matchmaker implementation, which also achieves reasonable recall levels without large impact on precision.

Keywords: semantic web services, approximate matching, approximate subsumption, logic-based

1 Motivation

Web service discovery and matchmaking is a field of ongoing research. For semantic web services, logic-based reasoning offers the possibility of high precision. However, in general it achieves only poor recall levels. Due to this, most current matcher implementations utilize a combination of logic- and non-logic-based, or even only non-logic-based similarity measures.

The best performing matcher during 2009's Semantic Service Selection contest[3] S3, URBE [15], uses only non-logic-based matching. Most of the other matchers we found so far (SAWSDL-MX2 [8], DAML-S Matcher [14], COCOON Glue [1], SAWSDL-iMatcher3/1[3]) utilize a hybrid strategy, merging the results of a logic and non-logic matching step. However, they only support coarse degrees of a logic match, i.e., *exact*, *plug in*, *subsumes*, *subsumed-by*, *intersection* and *fail* [22].

LOG4SWS.KOM [18] improves this approach by defining adaptive degrees of match and assigning numerical values to them, which can easily be combined

[3] http://www-ags.dfki.uni-sb.de/~klusch/s3/s3-2009-summary.pdf

with other numerical similarity values. These numerical values are determined through the taxonomic distance of the concepts used to annotate the service offer and request. As a fallback strategy, LOG4SWS uses WordNet[4] to determine the similarity of interface, operation or parameter names, if no concepts are available or an error occurs during processing. It toop part non-competitively in 2009's S3 as a beta version and outperformed URBE and SAWSDL-MX2 on average precision [18].

iSeM [7] is based on concept abduction [2], which is a similar notion of approximate subsumption to the one we use in our implementation. Consequently, it also supports approximate and more fine-grained degrees of match. Through concept abduction iSeM determines which properties of a request or an offer prevent the subsumption from succeeding and ranks request-offer pairs according to the loss of information by omitting those properties. Additionally it implements non-logic-based similarity measures to further improve the ranking.

Another service matcher which directly takes the structure of the taxonomy into account when computing matches is F-Match [21]. Yau et al. compute semantic similarity of concepts by their distance and relationship in the taxonomy as proposed in a graph matching approach by Zhong et al. [23] and use this value, together with some others, for ranking the services. Their evaluation based on randomly generated service offers and requests shows a surprisingly high precision and recall of 100%.

Our matcher is based on the notion of approximate subsumption as proposed in [19]. We extend this approach in three ways: our first addition utilizes information about the taxonomy of the background ontologies to achieve a more fine grained ranking. The other two extensions are aimed at lowering query response times through a slight change in the definition of approximate subsumption and optimized query execution order. Our goal is to improve recall through approximation, but keeping high precision and a fast query response time. For our matcher we assume web services to be annotated with semantic information according to the SAWSDL standard [9].

This paper is structured as follows: In section 2 we define approximate subsumption and give an example of its usage. The concept of approximate subsumption as we applied it to semantic web services and our implementation is described in section 3. We evaluate our implementation in section 4 and conclude in section 5.

2 Approximate Subsumption

In this section we briefly describe approximate subsumption. In [19] S-Interpretations [16] are applied to description logic, assigning each concept not in the set S the top $\top$ or bottom $\bot$ concept, depending on which interpretation is used. Consequentially, concepts not in S will be ignored by a subsumption test or cause it to fail. The interpretation which maps concepts to the top concept is called

[4] `http://wordnet.princeton.edu/`

upper approximation $\mathcal{I}_S^+$ and the interpretation mapping concepts to the bottom concept is called lower approximation $\mathcal{I}_S^-$. By applying these approximations to the definition of standard subsumption $\forall \mathcal{I} : \mathcal{I} \models C \sqsubseteq D \Leftrightarrow (C \sqcap \neg D)^{\mathcal{I}} = \emptyset$ (with C and D being concept expressions), an approximate subsumption operator is defined: $\underset{S}{\sqsubseteq}$:

$$\forall \mathcal{I} : \mathcal{I} \models C \underset{S}{\sqsubseteq} D \Leftrightarrow (C \sqcap \neg D)^{\mathcal{I}_S^-} = \emptyset$$

It is shown that this approach has the property of generalized monotonicity, making it possible to generate weaker version of the subsumption operator with every approximation step by decreasing the size of S. This allows to rank a result list based on the degree the operator had to be weakened to receive a specific result. Possible strategies for altering S include contraction by removing concepts sequentially or permutation by creating every possible combination of concepts in S.

A great advantage of this approach is, that it can be implemented by syntactic modifications of concept expressions and then be evaluated by standard description logic reasoners [20]. The definition of the rewriting rules for theses modifications for the lower approximation $(.)^-$ are as follows (with A being an atomic concept):

$$(A)^- \rightarrow \bot \text{ if } A \in S \tag{1}$$
$$(\neg A)^- \rightarrow \bot \text{ if } A \in S \tag{2}$$
$$(\neg C)^- \rightarrow \neg (C)^+ \tag{3}$$
$$(C \sqcap D)^- \rightarrow (C)^- \sqcap (D)^- \tag{4}$$
$$(C \sqcup D)^- \rightarrow (C)^- \sqcup (D)^- \tag{5}$$

The definition of the upper approximation $(.)^+$ is analogous, only equations 1 and 2 are adapted:

$$(A)^+ \rightarrow \top \text{ if } A \in S \tag{6}$$
$$(\neg A)^+ \rightarrow \top \text{ if } A \in S \tag{7}$$

Applying these rewriting rules to the approximate subsumption operator we get the following alternative definition:

$$\forall \mathcal{I} : \mathcal{I} \models C \underset{S}{\sqsubseteq} D \Leftrightarrow (C)^- \cap \neg (D)^+ = \emptyset$$

So to check for approximate subsumption we have to create the lower approximation of a service offer and the upper approximation of the service request and test whether the intersection is equal to the empty set.

In our implementation we call this original definition of approximate subsumption Simple strategy. The following gives an example of how this strategy is carried out[5]: Lets consider a search application for pizza delivery services. The user may specify different toppings he likes to have on his pizza, and the search then returns a number of delivery services offering pizzas with these ingredients. For example a user wants a vegetarian pizza with *Broccoli* and *Artichoke.* Unfortunately no delivery can satisfy this wish. Subsequently, the system approximates the request by creating two new request with *Broccoli* and *Artichoke* replaced with $\top$ respectively. Executing these two new requests, the system discovers pizza delivery services which offer *Broccoli* and *Mandarine* pizzas or *Artichoke* and *Mushroom* pizzas, which both might be relevant to the user. The next approximation step would approximate both *Broccoli* and *Artichoke* to $\top$ and subsequently return every pizza with two vegetarian toppings.

3 Applying Approximate Matching to Semantic Web Services

This chapter explains in more detail how we applied approximate subsumption to semantic web services and how we implemented partial matchmaking for service discovery and which extensions we made.

The implementation consists of two parts: The first part defines the web service ontology, processes the service offers and creates requests represented as concepts of the ontology. The second part, a generalized matchmaker, reasons on this ontology and computes matches to the request based on its subsumption relation to service offers.

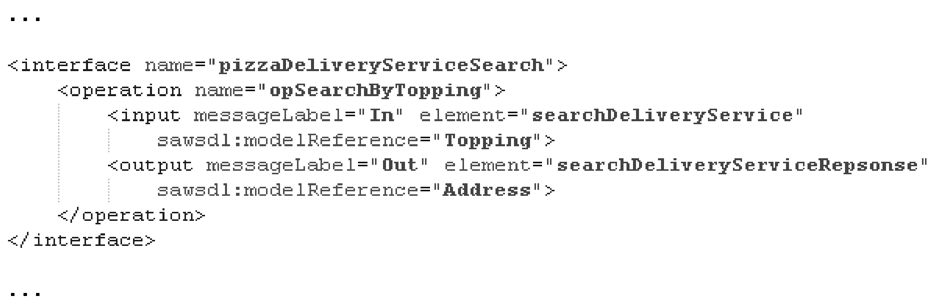
The typical usage of our matchmaker is divided into an offline initialization and classification of service offers, and the online processing of requests. During the initialization semantic annotations are extracted from SAWSDL services and added to a service ontology. This ontology is then classified with the help of a description logic reasoner. After the initialization any number of request can be issued to the matchmaker without further reclassification.

3.1 Extracting Semantic Annotations

In order to build an ontology of all known web services the semantic annotations have to be extracted from the web service descriptions. A simplified version of a web service with SAWSDL annotation is shown in Fig. 1.

The concepts, which are extracted from the web service annotations, are added to a service ontology. This is a minimal ontology for describing a service, modelled in OWL [5]. It is quite similar to other upper ontologies for Web services, like OWL-S [12] or WSMO [10], but its only classes are *Service*, *Operation*, *Input* and *Output* and the object properties *hasOperation*, *hasInput* and *hasOutput* connecting them. Other aspects of a service, like how to connect to it, or

[5] To simplify this example we do not create the lower approximation of all offered pizzas, but only the upper approximation of the request. Nonetheless, the general procedure stays the same.



```
...

<interface name="pizzaDeliveryServiceSearch">
    <operation name="opSearchByTopping">
        <input messageLabel="In" element="searchDeliveryService"
            sawsdl:modelReference="Topping">
        <output messageLabel="Out" element="searchDeliveryServiceRepsonse"
            sawsdl:modelReference="Address">
    </operation>
</interface>

...
```

Fig. 1. Simplified excerpt from a SAWSDL-annotated Web service

information about availability or reliability, which are present in more complex ontologies, e.g. OWL-S or WSMO, are not modelled in our prototype. Figure 2 shows how this looks like for the Web service example from above.

```
pizzaDeliveryServiceSearch SubclassOf
   (hasOperation some opSearchByTopping)

opSearchByTopping SubclassOf
   (hasInput only Topping) and
   (hasOutput only Address) and
   (hasInput some Topping or hasOutput some
   Address)
```

Fig. 2. Excerpt from the service ontology showing a web service instance

While loading a service, the approximator also counts the occurrences of each concept used to annotate its parameters, which is important when determining the order in which they should be approximated (cf. approximation strategies). Furthermore, it maintains a list of cumulated occurrences, which are the sum of a concept's and all of its sub concepts' occurrences throughout the whole service ontology. This number is a better indicator for how restricting a specific constraint of the request is than the number of occurrences of a concept alone. For example, *OWLThing* (the super concept of all concepts) and *OWLNothing* (the sub concept of all concepts) will most likely never be used to annotate a Web Service, so they both have a number of occurrence of 0. However, the cumulated occurrence of *OWLThing* is the sum of all other concepts' occurrences, whereas the cumulated occurrence of *OWLNothing* is still 0. This means if a service parameter is enforced to be of type *OWLNothing*, this is much more restricting, than enforcing it to be of type *OWLThing*[6].

[6] Actually, a constraint for a parameter to be of type OWLNothing is unsatisfiable and a constraint for a parameter to be of type OWLThing is always fulfilled

3.2 Creating and Approximating Queries

Query Creation A request is represented in the same way as a normal service offer (cf. section 3.1). There are two possibilities of creating a new request:

– A *Query-by-Example* approach, by which an existing annotated service is used to create a request from it. This can be useful, if someone wants to find a replacement for an existing service, or services with duplicate abilities should be found.
– Input and Output parameters are specified directly and a request is built using these, for example when a user searches a service to fulfill a specific task.

Query Approximation The Web Service Approximator creates queries using permutations to change the S set. Two strategies define the approximation of concept expressions: The first strategy determines the order in which concepts are approximated, depending on their cumulated number of occurrences. This strategy supports three different variants, influencing the ranking of the results (cf. [17]):

– LESS: concepts are approximated in ascending order of their cumulated number of occurrences
– MORE: concepts are approximated in descending order of their cumulated number of occurrences
– RANDOM: concepts are approximated in a random order

The second strategy defines how fine grained the approximation should be. The Simple variant approximates each concept of the request by replacing it with *OWLThing*, which is the approach in [19] described earlier. We extended this strategy with a variant that takes the taxonomy of the domain ontology into account:

Taxonomy Approach Instead of replacing a concept to be approximated with *OWLThing*, we use its direct super concept according to the taxonomy. We call this strategy Taxonomy. Our definitions of lower and upper approximation for the Taxonomy approach are the following:

$$(A)^- \rightarrow directsub(A) \text{ if } A \in S \tag{8}$$
$$(\neg A)^- \rightarrow directsub(A) \text{ if } A \in S \tag{9}$$

$$(A)^+ \rightarrow directsuper(A) \text{ if } A \in S \tag{10}$$
$$(\neg A)^+ \rightarrow directsuper(A) \text{ if } A \in S \tag{11}$$

The intention behind the TAXONOMY strategy is to create a much more detailed ranking of results and therefore achieve a higher precision than with the SIMPLE variant. This approach benefits especially if ontologies with a deep taxonomy are used for annotations and if concepts from every level of the taxonomy are used.

Revisiting the example of the pizza delivery service search, the first approximations according to the TAXONOMY strategy are the following: instead of replacing *Broccoli* and *Artichoke* with $\top$, we use their direct superconcept, which in this case is *Vegetable* for both. The search for pizzas with *Broccoli* and any other *Vegetable* or pizzas with *Artichoke* and any other *Vegetable* now only finds the delivery service offering *Artichoke* and *Mushroom* pizzas. The service delivering *Broccoli* and *Mandarine* pizzas is not found during the first approximation step, as *Mandarines* are a *Fruit* and no *Vegetable*. This pizza is found during the second step, when *Vegetable* is further approximated to *VegetarianFood*, of which *Fruits* are naturally are subconcept. Thus, the *Broccoli* and *Mandarine* pizza would be considered less relevant, as the request had to be further weakened to retrieve it.

Note that the approximator always generates all possible approximations of a request at once, avoiding duplicates by checking the parameters of each request. So for each super- or subconcept of a concept, respectively, a new request is generated. If the concept does not have a direct super- or subconcept, respectively, then no approximation is created. This is only the case for *OWLThing* and *OWLNothing*, as every other concept has *OWLThing* as superconcept and *OWLNothing* as subconcept.

Additionally, the approximator tries to avoid term collapsing [4] by prohibiting queries whose terms are all approximated to OWLThing. Term collapsing is a negative effect of approximate subsumption, which occurs if the approximated concept consists of conjunctions and one term is changed to *OWLThing*, or if it consists of disjunctions and one concept is approximated to *OWLNothing*, effectively turning the whole concept into *OWLThing* or *OWLNothing*, respectively.

During the creation of approximations, we generate a graph which captures the relations between the original request and its approximations. The root node of the graph is the original request, its direct children are the approximations created during the first step, their children are the approximations generate from them during the second step, and so on. This graph is later used to optimize the execution time of the matchmaker, especially when approximating along the taxonomy, which produces a large amount of queries: $Q \in \mathcal{O}(N^P)$, with Q being the number of created queries, P the number of parameters the request has and N the maximum number of superclasses a concept has. So the number of queries, and consequentially the execution time of the matcher, grows exponentially with the number of parameters. However, web service have mostly only a couple of parameters, so we hope to still calculate results in a reasonable amount of time.

The object representing an approximated query also stores the step in which it was approximated and the sum of the cardinalities of all concepts which were approximated to create this query. These two numbers are important to determine the position of each query's results in the final aggregated result list, as the partial matchmaker may process queries out of approximation order, and thus their results can not simply be appended to the list.

Query-only Approximation A drawback of the definition of approximate subsumption in [19] is the fact, that the concepts on both sides of the operator, in our case service requests and service offers, have to be rewritten. This means the whole ontology containing the service offers has to be approximated and reclassified by the reasoner for each request. Obviously, given an arbitrarily large service ontology, this is not feasible in a reasonable amount of time. We therefore changed the definition of the approximate subsumption operator to only approximate the request, leaving the concepts of the service ontology untouched.

$$\forall \mathcal{I} : \mathcal{I} \models C \underset{S}{\sqsubseteq} D \Leftrightarrow C^{\mathcal{I}} \cap (D)^{\mathcal{I}^+_S} = \emptyset$$

According to this definition we only apply the upper approximation to the request and test for subsumption with the service offers.

Optimizations During Query Execution As mentioned before, the approximation along the taxonomy produces a large amount of queries, leading to long delays until the matchmaker returns the results. To reduce the number of queries which have to be executed, we take advantage of the monotonicity property of the approximate subsumption and the graph structure created when approximating a request. The approach is based on the observation, that if two different queries, which were created during different steps and are related as ancestor / descendant, produce the same result lists, then, due to the monotonicity, every query between these two queries must produce the exact same result list and subsequently need not be executed. Our matchmaker utilizes this property by first executing the root and leave queries of the graph, and compares their results pairwise. If the results of a pair are equal, every query between this pair are omitted, otherwise the graph is split in two parts, and again the top and the bottom queries of this subgraphs are executed and compared.

Intersection Query After having executed the original query and all of its approximations, we also add all service offers to the result list, which have at least one concept in common with the request, no matter if it is used as input or output parameter. Surprisingly, this has a quite large impact on the precision of the SIMPLE strategy, as we will show in the evaluation.

4 Evaluation

In this chapter we evaluate the performance of the implemented matcher. The following points are considered:

– Retrieval performance of the implemented matcher compared to others
– Response time to user requests

We conducted the performance tests on a Mac Pro 3,1[7] running Windows XP SP3 and Java 1.6.20.

4.1 SME^2

The Semantic Web Service Matchmaker Evaluation Environment[8] (SME^2) is a Java-based tool developed at the German Research Center for Artificial Intelligence (DFKI) and is used during the Semantic Service Selection (S3) contest to evaluate and compare the performance of matchmakers. It supports the addition of matchmakers and test collections as a plug-in; currently collections for OWL-S and SAWSDL and the matchmakers written by the DFKI are publicly available.

SME^2 evaluates the matchers' performance with standard performance measures of information retrieval [11]:

– Precision: Fraction of the retrieved results, which are relevant
– Recall: Fraction of all relevant documents, which are retrieved
– Fallout: Fraction of retrieved irrelevant documents.
– F-Measure: Weighted harmonic mean of Precision and Recall.

Furthermore, the average precision for each query is computed and other statistics like execution time, query response time and memory consumption are recorded.

We use SAWSDL-TC[9], which is supplied with SME^2, as test collection for our matchmaker. It consists of 894 service offers and 26 service requests. For each request the relevant service offers are specified, allowing the calculation of the service measures mentioned above. The services are annotated with concepts from several different ontologies from different domains, ranging from military over education to health care and food.

The ontology with the extracted annotations of 894 services from SAWSDL-TC contains 2149 named and 3261 anonymous classes (SubclassOf axioms). This ontology is then classified by the matcher using a description logic reasoner suitable for OWL.

4.2 Comparison of Retrieval Performance

This section compares the implemented matcher's retrieval performance to that of SAWSDL-MX2, which is, like SME^2, developed at the DFKI. It implements several possibilities for calculating matches [6]:

[7] Two 2.8 GHz Intel Xeon Quad-Core processors, 2GB RAM
[8] `http://www.semwebcentral.org/projects/sme2/`
[9] `http://projects.semwebcentral.org/projects/sawsdl-tc/`

– Logic-based: computes matches based on the semantic annotations of input and output parameters
– Text similarity: uses standard information retrieval methods for finding similar descriptions
– Structural similarity: compares the structure of services: interfaces, bindings, number of parameter, etc.

Additionally, SAWSDL-MX2 implements a machine learning feature to support the weighting of each of this properties, when it ranks the result list.

Table 1 gives an overview of the matchers' overall performance. The average precision is the average of the average precision for every query. The query response time is the time needed on average to execute a single query. The number of executed queries is the number of requests issued to the reasoner. This number is only available for our implementation.

	SIMPLE	TAXONOMY	SAWSDL-MX2
∅ precision	65.52%	73.97%	70.50%
∅ query response time	1.06sec	15.15sec	3.18sec
# executed queries	153	2487	n/a

Table 1. Overall performance of SIMPLE, TAXONOMY and SAWSDL-MX2

Overall Precision and Recall First, we will compare the overall precision and recall of our matcher utilizing the SIMPLE and TAXONOMY strategies and SAWSDL-MX2 (Fig. 3).

For every level of recall, TAXONOMY achieves a higher precision than SIMPLE. Because TAXONOMY returns the same services as results as SIMPLE does, and only refines the ranking produced by it, both strategies deliver an almost identical performance at lowest and highest recall levels. Between those, TAXONOMY shows the advantage of the fine grained approximation of concepts, achieving a precision which is up to 30% higher than SIMPLE's. The TAXONOMY approach also has the highest average precision of all 3 matchers. However, its precision is worse than that of SAWSDL-MX2 at the first quarter of the result list, where SAWSDL-MX2 is able to reach a precision of almost 100%. Overall, the precision of both our implementations decreases slower than SAWSDL-MX2's, which drops below SIMPLE's precision towards the end of the result list.

Overall Recall and Fallout The recall/fallout diagram (Fig. 4) shows the same tendencies as the overall recall and precision, and again emphasizes the good proportion between precision and recall of the TAXONOMY strategy. It returns less false positives than SAWSDL-MX2, which produces almost twice as much at full recall.

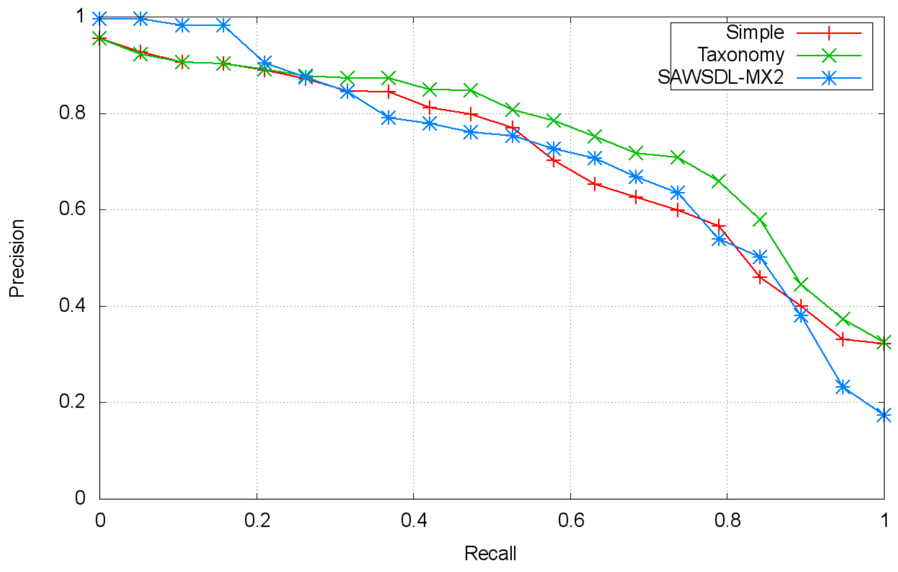


Fig. 3. Recall/Precision of Simple, Taxonomy and SAWSDL-MX2

Influence of the Intersection Query Figure 5 shows the influence of the intersection query on the performance of the Simple and Taxonomy approach. The recall/precision curves for Taxonomy with and without the intersection query show only a slight difference. For Simple, however, the intersection query increases the precision significantly; by up to 10% at full recall. The beginning of the curves is identical between the variants with and without intersection query, because it only adds offers to the bottom of the result list, as mentioned before, and does not change the ranking of previous results.

4.3 Response Time to User Requests

Besides the retrieval performance, for a real world usage of the matcher, it is also important to deliver results in a reasonable amount of time [13].

query response time	∅	median	std dev	cv
Simple	1054ms	892ms	438ms	0.42
Taxonomy	15204ms	7292ms	25863ms	1.70
SAWSDL-MX2	3178ms	2313ms	1541ms	0.48

Table 2. Average and median value, standard deviation (std dev) and coefficient of variation (cv) for the query response time of some variants.

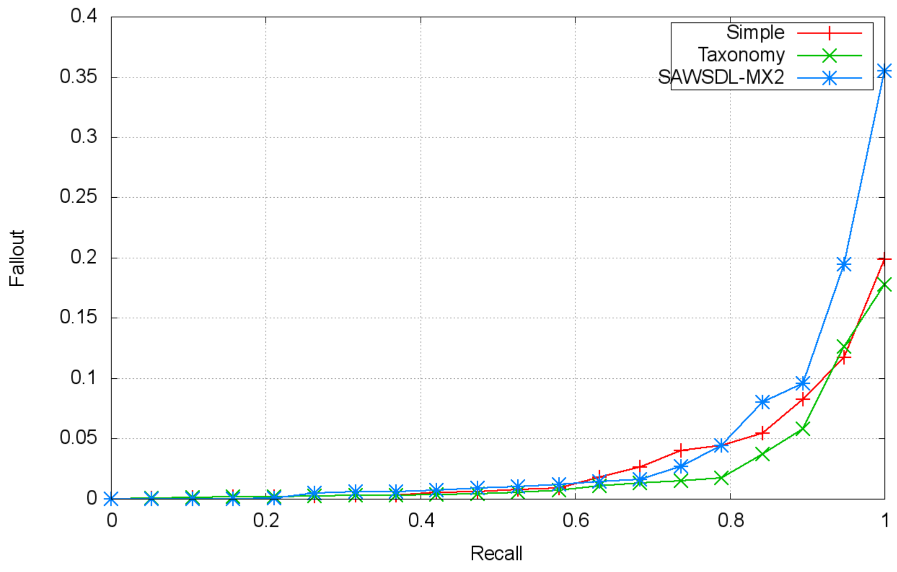


Fig. 4. Recall/Fallout of Simple, Taxonomy and SAWSDL-MX2

Table 2 shows the average value, median value, standard deviation and coefficient of variation of the response times the matchers need for evaluating a single request. These numbers show that a higher precision comes at the cost of longer wait time for the users. Simple is quite fast with about one second on average and SAWSDL-MX2 is not much slower, taking around three seconds to deliver results; Taxonomy uses a lot more time: 15 seconds on average.

Figure 6 and the median value, standard deviation and coefficient of variation show, that Simple and SAWSDL-MX have quite steady query response times, which do not vary wildly. For Taxonomy, however, the response times show huge differences. Fortunately, most times are lower than the average, as indicated by the median value only being half of the average precision. For some queries, Taxonomy is faster than SAWSDL-MX, for others, there are some outliers for which execution took around two minutes.

In general, Simple is faster than SAWSDL-MX, which is in turn faster than Taxonomy, what corresponds directly with their average precision.

Optimization of Query Response Time The response times of Simple and Taxonomy are already improved through the optimization described in Section 3.2. Without it, Taxonomy would need 20 seconds on average to answer a request (Table 3). The benefits for Simple are minimal, as it only produces a small query graph. However, Taxonomy issues almost one third less queries to the reasoner, saving as much time for a request.

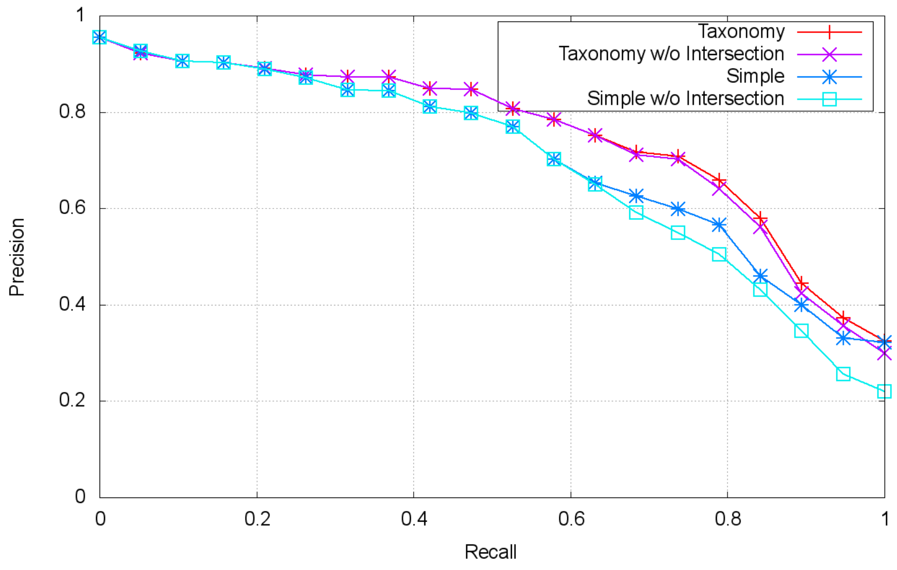


Fig. 5. Recall/Fallout of Simple, Taxonomy and SAWSDL-MX2

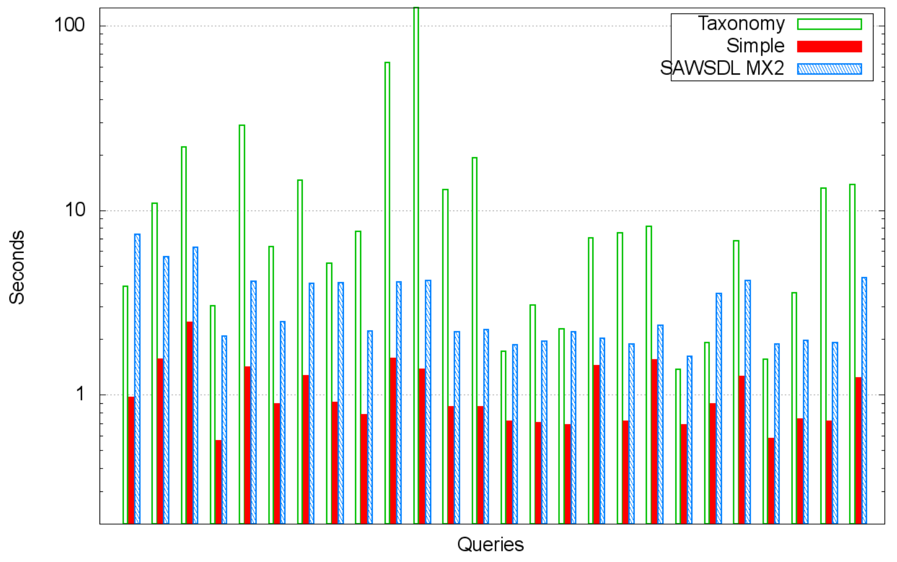


Fig. 6. Query response time of WSA Simple, WSA Taxonomy and SAWSDL-MX2

	Simple	Taxonomy
# created queries	153	3680
# executed queries	130	2487
∅ savings/request	15%	32%
∅ response time savings	3%	33%

Table 3. Number of created and executed queries, and savings through optimizations

5 Conclusion

We presented our implementation of a purely logic-based approximate web service matcher, which is based on approximate subsumption. The results of our evaluation are promising, especially for our goal to increase recall and still maintain a high precision. Considering the recall, our approach even performs better then all other matchers contending in the S3, and still achieves competitive precision. Our extensions aimed at decreasing query response time have generally helped to achieve a reasonable speed for our matcher, despite the additional executed queries. However, in some special cases our implementation still needs too much time for executing all approximations.

Future goals are to improve query performance, where we will consider several possibilities: As the approximated queries are independent from each other, they could be executed concurrently, but unfortunately parallelization is supported poorly by current reasoners. Another possibility to improve the perceived performance of the matcher is the implementation of an anytime behaviour, to show the user some first results and then refine or extend the list in the background.

Long term goals are the integration of user preferences in the approximation and ranking process, e.g., black or white lists defining which concepts should or should not be approximated. Furthermore, the matcher could create proposals for combining several services, which together can fulfill the users request. To improve the matchers performance in a heterogeneous environment, where service are annotated with concepts from different, not formally related ontologies, an ontology alignment [3] process could improve retrieval performance.

References

1. E. Della Valle and D. Cerizza. Cocoon glue: a prototype of "wsmo" discovery engine for the healthcare field. In *Proceedings of the WIW 2005 Workshop on WSMO Implementations, Innsbruck, Austria, June 6-7*, volume 134 of *CEUR-WS*, pages 1–12, 2005.
2. T. Di Noia, E. Di Sciascio, F.M. Donini, and M. Mongiello. Abductive matchmaking using description logics. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 18, pages 337–342. Citeseer, 2003.
3. J. Euzenat and P. Shvaiko. *Ontology matching.* Springer-Verlag New York Inc, 2007.

4. P. Groot, H. Stuckenschmidt, and H. Wache. Approximating description logic classification for semantic web reasoning. In *European Semantic Web Conference (ESWC), Heraklion, Greece, May 29 - June 1*, volume Volume 3532 of *Lecture Notes in Computer Science (LNCS)*, pages 318–332. Springer, 2005.
5. W3C OWL Working Group. OWL 2 web ontology language document overview. Technical report, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.
6. M. Klusch and P. Kapahnke. Semantic web service selection with "sawsdl-mx". In Rubén Lara Hernandez, Tommaso Di Noia, and Ioan Toma, editors, *Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMRR), Karlsruhe, Germany, October 27*, volume 416 of *CEUR Workshop Proceedings*, 2008.
7. M. Klusch and P. Kapahnke. isem: Approximated reasoning for adaptive hybrid selection of semantic services. In *Proceedings of 4th IEEE International Conference on Semantic Computing (ICSC)*, 2010.
8. M. Klusch, P. Kapahnke, and I. Zinnikus. "sawsdl-mx2": A machine-learning approach for integrating semantic web service matchmaking variants. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 335–342, Washington, DC, USA, 2009. IEEE Computer Society.
9. J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell. "sawsdl: Semantic annotations for wsdl and xml schema". *IEEE Internet Computing*, November / December:60 – 67, 2007.
10. H. Lausen, A. Polleres, and D. Roman. Web service modeling ontology (wsmo). W3C member submission, W3C, June 2005. http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/.
11. C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval.* Cambridge Univ Pr, 2008.
12. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. W3C member submission, W3C, November 2004. http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.
13. R.B. Miller. Response time in man-computer conversational transactions. In *AFIPS Joint Computer Conferences 1968, San Francisco, CA, USA, December 9-11*, pages 267–277. ACM, 1968.
14. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Importing the semantic web in "uddi". In *Web services, E-Business, and the Semantic Web, (WES) CAiSE-Workshop, Toronto, Canada, May 27-28*, volume 2512 of *LNCS*, pages 815–821. Springer, 2002.
15. P. Plebani and B. Pernici. Urbe: Web service retrieval based on similarity evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 21:1629 – 1642, 2009.
16. M. Schaerf and M. Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74(2):249–310, 1995.
17. S. Schlobach, E. Blaauw, M. El Kebir, A. Ten Teije, F. Van Harmelen, S. Bortoli, et al. Anytime classification by ontology approximation. In Ruzica Piskac, Frank van Harmelen, and Ning Zhong, editors, *New forms of reasoning for the Semantic Web*, volume 291 of *CEUR-WS*, pages 60–74, 2007.
18. S. Schulte, U. Lampe, J. Eckert, and R. Steinmetz. "log4sws.kom": Self-adapting semantic web service discovery for "sawsdl". In *IEEE Congress on Services, Miami, FL, USA, July 5-10*, pages 511–518. IEEE Computer Society, 2010.

19. H. Stuckenschmidt. Partial matchmaking using approximate subsumption. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI-07)*, pages 1459–1464, Vancouver, British Columbia, Canada, July 2007. AAAI Press.
20. H. Stuckenschmidt and M. Kolb. Partial matchmaking for complex product- and service descriptions. In *Proceedings of Multikonferenz Wirtschaftsinformatik (MKWI 2008), Special Track on Semantic Web Technology in Business Information Systems, Munich, Germany, February 26-28*, 2008.
21. S.S. Yau and J. Liu. Functionality-based service matchmaking for service-oriented architecture. In *International Symposium on Autonomous Decentralized Systems (ISADS'07), Sedona, USA, March 21-23*. IEEE Computer society, 2007.
22. A.M. Zaremski and J.M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):170, 1995.
23. J. Zhong, H. Zhu, J. Li, and Y. Yu. Conceptual graph matching for semantic search. In *International Conference on Conceptual Structures: Integration and Interfaces (ICCS), Borovets, Bulgaria, July 15-19*, volume 2393 of *LNCS*, pages 92–106, 2002.