

Retrieving Compatible Web Services

Vasilios Andrikopoulos
European Research Institute in Service Science (ERISS)
Tilburg University
Warandelaan 2, PO Box 90153, 5000LE, the Netherlands
V.Andrikopoulos@uvt.nl

Pierluigi Plebani
Dipartimento di Elettronica ed Informazione
Politecnico di Milano
Piazza L. da Vinci, 32 20133 Milan, Italy
plebani@elet.polimi.it

Abstract—Service retrieval holds a central role during the development of Web services and Service-Based Applications (SBAs). The higher the number of available services, the more complex it becomes to locate the service closer to the developer needs. The complexity increases further with the number of available service versions that could also be suitable for this purpose. Existing approaches on service retrieval use a similarity measure between service interfaces to identify potentially relevant services. In this work we focus on introducing information about the compatibility of services while calculating their similarity as the means for providing more suitable results. For this purpose we update and extend an existing Web services matchmaker called UDDI Registry By Example (URBE).

Keywords—Service retrieval, service compatibility, similarity.

I. INTRODUCTION

During the recent years, the number of publicly available Web services has been increasing steadily. Seekda¹, one of the largest Web services search engine, has indexed almost 30,000 Web services. An important step in enabling the Service-Oriented Architecture (SOA) paradigm is the ability of service and SBA developers (simply referred to as developers from now on) to be able to retrieve potentially relevant services. In particular, in this work we focus on the task assigned to developers to identify at design-time which activity is to be performed and to *discover* and *select* the Web services closest to their requirements [1]. Furthermore, it is also necessary to be able to identify and replace services participating in a service composition at run-time [2].

In this sense, *service retrieval*, frequently referred to also as *service discovery*, is a critical step for reusing existing services while developing other services and SBAs [3]. Several approaches have been proposed in the literature for Web services retrieval, see for example [4]. We can distinguish between two categories of solutions: registry-based and ontology-based ones. Ontology-based approaches do not usually consider the structure of the Web service interfaces and they require additional effort to produce a service description. For these reasons, and despite the effectiveness of ontology-based approaches, we focus on a registry-based solution.

In particular, we use as the starting point for our approach the URBE matchmaker [1]. URBE is an approach for service retrieval based on the evaluation of similarity between Web service interfaces. In URBE, each Web service interface is defined in Web Services Description Language (WSDL); a matchmaking algorithm combines the analysis of their structures with the similarity of the used terms in order to retrieve relevant services for purposes of replacing a service. As discussed in [1], URBE performs on average better when compared to approaches like [3] and for this reason it was selected as the baseline for our work.

Both URBE and similar solutions consider not only the *structure* but also the *semantics* of candidate services. They do not however take into account the *purpose* of each element in the service description with respect to service compatibility. *Service compatibility* refers to the property of preservation of interoperability for internalized changes to one or both interacting parties (service provider or consumer), or equivalently, of the capacity for replacing one service with another (also referred to as substitutability and replaceability) [5]. Different elements in the service description have different effects on interoperability: adding for example an operation to a WSDL document does not have any effect on existing clients of the service; removing an operation however may affect them dramatically.

For this purpose, the work in [5] uses a *subtyping* relation between the elements of service description documents in order to formally define service compatibility. Following this approach, compatibility is preserved as long as the properties of *covariance* of output and *contra-variance* of input are preserved. This is a property that is not considered in the matchmaking algorithms discussed in URBE and similar approaches.

To this effect, in this work we aim to combine service retrieval with service compatibility with the goal of improving the matchmaking of URBE. The new matchmaking algorithm takes into account not only the interface structure and term similarity, but also the importance of each element for service compatibility. As a result of this synergy, retrieved services are not only similar to the required service, but additionally, a minimum effort is demanded from the developers in order to be able to use this service on the

¹<http://webservices.seekda.com/>

composite service or SBA side. Furthermore, the updated URBE implementation is shown to perform better both in terms of precision and average response time with respect to the older version.

The rest of the paper is structured as follows. Section II provides a better background for the rest of the work by discussing compatibility between service interfaces. Section III enters into the detail on the evaluation of the matchmaking between two service interfaces, and Section IV validates the approach. Finally, Section V discusses related work and Section VI focuses on concluding remarks and possible future work.

II. SERVICE COMPATIBILITY

Service compatibility can be distinguished in two dimensions: horizontal compatibility (or service interoperability) and vertical compatibility (also known as substitutability or replaceability) [5]. *Horizontal compatibility* or *interoperability* of two services expresses the fact that the services can participate successfully in an interaction as service provider and service consumer. The underlying assumption is that there is at least one context (configuration of the environment, resource status and message exchange history) under which the two services can fulfill their roles. On the other hand, *vertical compatibility* or *substitutability* (from the provider's perspective) or *replaceability* (from the consumer's perspective) of service versions expresses the requirements that allow the replacement of one version by another in a given context.

Compatibility is traditionally further decomposed into *backward* and *forward*. A definition of forward and backward compatibility with respect to languages in general, and message exchanges between producers and consumers in particular, is given in [6]. *Forward compatibility* means that a new version of a message producer can be deployed without the need for updating the message consumer(s). *Backward compatibility* means that a new version of a message consumer can be deployed without the need for updating the message producer. *Full* compatibility is the combination of both forward and backward compatibility.

The usual approach for defining what constitutes a compatible change to a service is to enumerate all possible compatibility preserving changes to a service description, usually a WSDL document. The allowed changes essentially define the type of *delta* between two service versions for which the versions are compatible and they are usually expressed in a guideline style. These guidelines (see for example [7]) can be summarized by:

- 1) Add (optional) message data types.
- 2) Add (new) operation.
- 3) Add (new) port type.

Any other modification like the removal, or any kind of modification to an operation element is strictly prohibited, as is the modification of the message data types (with

```
<xsd:complexType name="PODocument1">
  <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="PODocument2">
  <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string"/>
    <xsd:element name="TimeStamp" type="xsd:dateTime"/>
  </xsd:sequence>
</xsd:complexType>
```

Figure 1. Example of record subtyping.

the exception of addition of optional data types). This guideline-based approach is easily applicable and requires minimum support infrastructure and for that reason it is widely accepted. On the other hand it is very restrictive and it depends on service developers for deciding what is compatible and what is not and acting accordingly. Even if these rules are codified and embedded into a service development/versioning tool as for example in the case of [8], they will always be limited by two factors: their dependency on the particular technology used (WSDL in this case) and their lack of a robust theoretical foundation. For these reasons [5] introduces the notion of *T-shaped changes* based on the formal foundation of type theory for defining what constitutes compatible service descriptions.

In order to apply type theory constructs to service interfaces, [5] assumes that each service S is comprised of *records* s that represent the conceptual dependencies inside the service interface description. A service record s is a subtype of another record s' if and only if it has at least all the typed properties of s' (and possibly more) and all the common properties are also in a subtyping relation. In this case we write $s \leq s'$.

Fig. 1 shows an example of two service records that are connected by subtyping using two versions of purchase order `PODocument` data type: $s_{PODocument2} \leq s_{PODocument1}$, that is, `PODocument2` is a subtype (a specialization) of `PODocument1` since it contains more `xsd:element` types – and the ones it contains are less general than the respective ones of `PODocument1`. The optional cardinality of the `DeliveryInfo` element in `PODocument1` is more general than the obligatory participation in `PODocument2`, since a message consumer that understands it as an optional element can also understand the message always containing it.

The records s in service S are distributed into two proper subsets S_{pro} and S_{req} , representing the set of records for which the service acts as a producer and a consumer of messages respectively. Output-type records like (WSDL) operation output or fault messages and all their affiliated data types belong to the S_{pro} set. Input-type records like

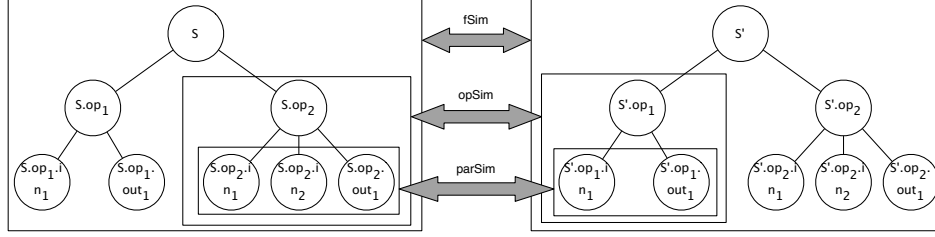


Figure 2. Service tree-based representation and nested comparison in URBE.

operation input messages and their affiliated data types are in the S_{req} set. Compatibility between service descriptions S and S' is defined based on this distribution as:

Definition 1 (Types of Service Compatibility). *We define three cases of compatibility:*

- *Forward:* $S <_f S' \Leftrightarrow \forall s \in S_{pro}, \exists s' \in S'_{pro}, s' \leq s$ (covariance of output).
- *Backward:* $S <_b S' \Leftrightarrow \forall s' \in S'_{req}, \exists s \in S_{req}, s \leq s'$ (contravariance of input).
- *Full:* $S <_c S' \Leftrightarrow S <_f S' \wedge S <_b S'$.

Definition 1 provides the general condition for the preservation of compatibility. The differences between two service descriptions S and S' is expressed by a *change set* ΔS that aggregates the set of primitive changes that occur to S and result to S' , that is, $S' = S \circ \Delta S$. We classify the change sets with respect to compatibility as follows:

Definition 2 (T-shaped changes). *A change set ΔS is called T-shaped, and we write $\Delta S \in \mathbb{T}$ where \mathbb{T} is the set of all possible T-shaped changes, if and only if, when ΔS is applied to a service S it results into a fully compatible with S service $S' = S \circ \Delta S$, that is, $S <_c S'$.*

As long as a change set ΔS results in a horizontally or vertically compatible (or both) version of a service, then it belongs to the set \mathbb{T} of all possible T-shaped changes. By reasoning on Definition 1 using the definition of service record subtyping we can conclude that the following change patterns are T-shaped:

- 1) Add (optional) message data types, add (new) operation & add (new) port type [as above].
- 2) Remove one-way (only) operations.
- 3) Add (obligatory or optional) data types to output messages.
- 4) Remove (obligatory or optional) data types from input messages.
- 5) Add output message parts to operations.
- 6) Remove input message parts from operations.

These patterns therefore constitute the T-shaped changes that can occur to a service and preserve its compatibility. As it can be seen, these patterns are a super-set of the preservation guidelines discussed in the above. Furthermore they provide a more fine-grained approach for the managing

the service compatibility while a service is evolving (see [5] for a further discussion on the subject).

For purposes of service retrieval, a T-shaped change set between service descriptions $S' = S \circ \Delta S$ expresses the fact that S' and S are *structurally equivalent*. This means that either S' can be used as-is in place of the required service description S , or that it is ensured that an adapter can be generated that will enable the use of S' , using for example an approach like [9]. In any case however, S' is a positive match for retrieving relevant to S services. This observation allows the incorporation of the theory presented above in the URBE matchmaker, as we discuss in the following.

III. SERVICE RETRIEVAL

A. The URBE Matchmaker

The similarity algorithm running in URBE implements a similarity function $fSim : (S, S') \rightarrow [0..1]$ that, given two service descriptions S – representing the requested service, and S' – representing the available service, returns the similarity degree as a value included in $[0..1]$: the higher the result of $fSim$, the higher the similarity between the two interfaces.

Fig. 2 provides a high level view of $fSim$, in which each service can be represented as a three-level tree: first, we have S representing a `portType`, then the set of operations $S.op_k$, and finally, the set of parameters $(S.op_k.in_l, S.op_k.out_m)$ representing the parameters of the supported operations. As a consequence, the functions which evaluate the similarity among the whole interfaces ($fSim$), operations ($opSim$), and parameters ($inParSim$ and $outParSim$) are nested in the same way. More specifically:

- $fSim$ returns the similarity between S and S' by computing the similarity of the names of the `portTypes` and relying on the function $opSim$;
- For each operation in S , $opSim$ identifies the operation in S' which is most similar to, computing the similarity between the names of the operations and relying on $inParSim$ and $outParSim$.
- Finally, given two sets of parameters in S , $inParSim$ and $outParSim$ find the most similar parameters in the S' considering the similarity of the names of parameters and their data types.

Table I
SIMPLE DATA TYPE SIMILARITY.

		<i>dt</i>				
		Integer	Real	String	Date	Boolean
<i>dt'</i>	<i>dataTypeSim</i>					
	Integer	1.0	0.5	0.3	0.1	0.1
	Real	1.0	1.0	0.1	0.0	0.1
	String	0.7	0.7	1.0	0.8	0.3
	Date	0.1	0.0	0.1	1.0	0.0
Boolean	0.1	0.0	0.1	0.0	1.0	

It is worth noting that we used WordNet² as a term taxonomy and the approach proposed by Pirr  and Seco [10] to compute the similarity among the terms by means of the Java WordNet Similarity Library (JWSL)³.

B. Introducing Compatibility into URBE

The principal goal of this paper is to discuss how the analysis of service compatibility as discussed in Section II influences the computation of service similarity as implemented in URBE. Generally speaking, service compatibility based on T-shaped changes relies on a binary (on/off) approach: two services are compatible if all the required changes are T-shaped, otherwise they are not. On the other hand, service similarity has a more fuzzy-oriented evaluation: two services are more or less similar, depending on the common operations and parameters. Combining the two approaches, we modified the matchmaking algorithm on which URBE is based so that it identifies and reflects the compatibility issues identified by T-shaped changes.

In particular, we focus on the *opSim* function which identifies the similarity between two operations by considering the *parSim* function, that, in turn, identifies the similarity between two parameters. In the former version of URBE, given two operations $op \in S$ and $op' \in S'$, *opSim* calculates the similarity pairwise by comparing the parameters $par \in op$ with parameters $par' \in op'$. When all similarities are computed according to the values returned by *parSim*, then *opSim* returns the global similarity by taking into account the best matches obtained by solving an *assignment in bipartite graphs* problem (as discussed in [1]). It is worth noting that the similarity returned by *parSim* depends on two factors: the similarity between the names of the parameters and the similarity between the data types of these parameters.

Focusing on the data type similarity, *parSim* differentiates between simple XSD data types (e.g., `xsd:string`, `xsd:integer`) and XSD complex data types (using `xsd:complexType` constructor)⁴. To compute the similarity between two simple data types that is implemented by the function *dtSim*, five main classes [3] of data types are identified: Integer, Real, String, Date, Boolean. In this

way, similarity between two simple data types dt and dt' is inversely proportional to the information loss that will occur if we apply a casting from dt to dt' . Table I, empirically obtained, quantifies this information loss. For instance, if dt belongs to the integer group and dt' to the real group then the similarity is 1.0 since we have no information loss. In the opposite situation instead, the similarity is 0.5 since we can convert a real into an integer but we lose the decimals. Finally, if dt or dt' is complex, the structure of the data type is not considered and the similarity only depends on the name of the data type.

Starting from this version of the algorithm, in this work we integrate the subtyping theory, discussed in the previous section, inside the *parSim* to improve the effectiveness of the discovery algorithm proposed in URBE. In particular, our work focuses on the analysis of complex data types since the simple data type compatibility, as it is driven by the casting of data types, already incorporates the notion of compatibility. More in detail, consider two complex data types dt and dt' defined as follows:

$$dt = \{dt_i\} \text{ and } dt' = \{dt'_j\}$$

where each dt_i and dt'_j could be either simple or complex data types. For the sake of simplicity, and without affecting the generality of the approach, in the following discussion we assume that the elements composing a complex data type are all simple data types. To quantify the compatibility between the two data sets, the function $parSim(dt, dt') \Rightarrow [0..1]$ reflects how many elements in dt have a correspondent in dt' , i.e.,

$$parSim(dt, dt') = \frac{|matching(\{dt_i\}, \{dt'_j\})|}{|dt'_j|}$$

The goal of the *matching* function is to run the assignment in the bipartite graphs where the elements in the two matching sets are $\{dt_i\}$ and $\{dt'_j\}$ and to return the number of elements in dt that have a correspondences in dt' , i.e., $dtSim(dt, dt') > th$, where th is a threshold. In case all the elements in dt' are covered, then *parSim* returns the maximum compatibility, i.e., 1.0. If an element in dt does not appear in dt' then a new element should be introduced and, according to the T-shaped changes definition, the data types remain compatible and the *parSim* is not affected if and only if dt and dt' are input-type elements (Definition 1). On the other hand, if not all the elements in dt' are covered by dt then some elements must be removed and the compatibility is not ensured. In that case, *parSim* will decrease proportionally to the number of these removals. This analysis is inverted if dt and dt' are output-type elements.

Fig. 3 shows an abstract example of a possible matching, where the weights labelling the edges are computed by the

²<http://wordnet.princeton.edu/>

³<http://grid.deis.unical.it/similarity/>

⁴Assuming that all the services can be described using WSDL, the data types will be expressed using XSD schemas

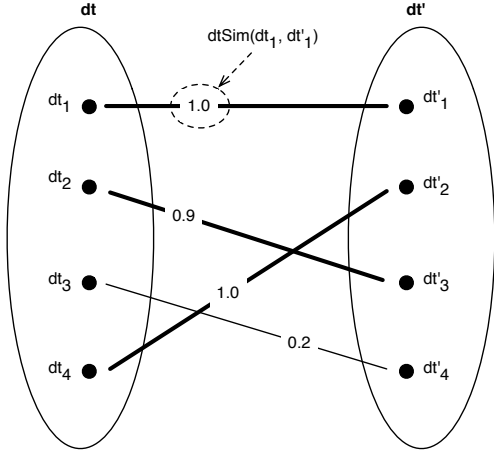


Figure 3. Example of *parSim* execution.

dtSim function which calculates the similarity of simple data types. In this case, assuming $th = 0.8$ the solution of the assignment in bipartite graphs problem recognizes a similarity between elements $\{\langle dt1, dt'_1 \rangle, \langle dt2, dt'_3 \rangle, \langle dt4, dt'_2 \rangle\}$. As a consequence $|matching(\{dt_i\}\{dt'_j\})| = 3$, $|dt'_j| = 4$ and therefore $parSim(dt, dt') = 0.75$.

According to this scenario, the value of the threshold th becomes crucial for the approach. If th was set to a too high value, two data types can be considered as matching only if they are the same; otherwise, if th was set to a too low value, it might happen that two data types are considered relevant even if they are not. By using the tuning performed for URBE this threshold is currently set to 0.3.

Furthermore, in case dt is a simple data type and dt' is a complex data type (or vice versa), the approach is similar: the algorithm looks for one of the elements composing the complex data type that is compatible to the simple data type and the similarity is calculated as discussed above.

IV. VALIDATION

A. Experimental Setting

The effectiveness of the new approach has been validated by measuring the precision and recall with respect to a public benchmark obtained from the SAWSDL [11] service retrieval test collection (SAWSDL-TC3)⁵. SAWSDL semantically enriches the WSDL-based service definition by annotations: elements of the WSDL are annotated with concepts organized in a reference ontology. The benchmark consists of 1080 Web services covering different application domains: communication, economy, education, food, medical care, travel and weaponry. The benchmark also includes 42 test queries, represented as SAWSDL documents, each of which is associated with a set of services that the proponents of the benchmark have defined as relevant.

⁵<http://projects.semwebcentral.org/projects/sawSDL-tc/>

Precision (i.e., number of relevant returned Web services w.r.t. the number of returned Web services) and *Recall* (i.e., number of relevant returned Web services w.r.t. total relevant Web services in the corpus) have been adopted as the parameters to evaluate the effectiveness of our approach [12]. More specifically, precision $P(ws_q)$ and recall $R(ws_q)$ are defined as:

$$P(ws_q) = \frac{|\{ws_i \in Ret(ws_q) | ws_i \in Rel(ws_q)\}|}{|Ret(ws_q)|}$$

$$R(ws_q) = \frac{|\{ws_i \in Ret(ws_q) | ws_i \in Rel(ws_q)\}|}{|Rel(ws_q)|}$$

where ws_q is the query, $Ret(ws_q)$ the returned services after submitting the query, and $Rel(ws_q)$ the relevant services for the given query.

An additional parameter to validate the performance of the algorithm is given by *Average Precision (AP)*. Precision and recall are measures for the entire result set without considering the ranking order, whereas *AP* depends on the precision at a given cut-off point (P^n). Thus, assuming $Ret^n(ws_q)$ as the set including the first n returned services, P^n for a given query ws_q is defined as:

$$P^n(ws_q) = \frac{|\{ws_i \in Ret^n(ws_q) | ws_i \in Rel(ws_q)\}|}{n}$$

AP is the average of precisions computed after truncating the list after each of the relevant documents, as long as all the relevant documents are retrieved:

$$AP(ws_q) = \frac{\sum_{r=1..N} P^r(ws_q)}{|\{ws_i \in Ret^N(ws_q) | ws_i \in Rel(ws_q)\}|}$$

where $N = |\{ws_i \in Ret(ws_q) | ws_i \in Rel(ws_q)\}|$ is the number of relevant documents.

Finally, the *Top-5* (obtained as $P^5(ws_q)$) and *Top-10* ($P^{10}(ws_q)$) precision metrics are also considered. These parameters give the precision considering only the first 5 or 10 entries, respectively. The higher the value, the higher the probability to have a relevant service in the first positions.

All these parameters had been calculated for each of the 42 test queries included in the benchmark. The averages of these parameters will be used as a basis for the evaluation of the algorithm in the next paragraph.

All the experiments discussed in this paper have been done on an MacOS X 10.6 installed on Intel Core 2 Due 2.33 GHz and 8 GBytes of RAM. The part of the algorithm able to solve the linear programming problem exploits the open-source application LPSolve⁶ and it has been formulated to always obtain the global optimum result.

⁶<http://sourceforge.net/projects/lpsolve/>

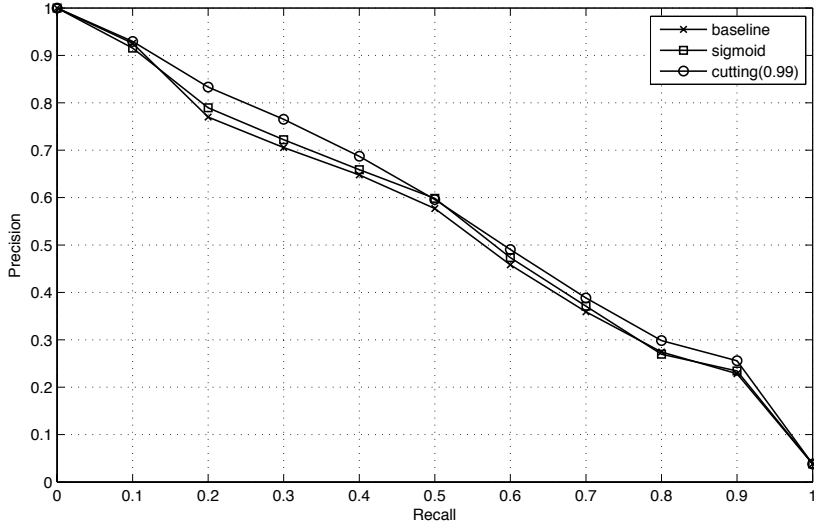


Figure 4. Precision Recall charts.

B. Results

The effectiveness of the presented approach is illustrated in Fig. 4. As a baseline for its evaluation we consider the precision/recall chart of URBE, ran without the compatibility analysis described in this paper. Such a chart is compared with the new approach implemented with two main variants. These variants, i.e., *sigmoid* and *cutting*, influence the behaviour of *dtSim*. In URBE, this function returns a value in $[0..1]$ that depends on the values in Table I as discussed in the previous section. Applying a sigmoid function to the returned value, we obtain a value that distinguishes better between similar and not similar data types. A more radical distinction is performed with the cutting variant, a binary function that returns 0 or 1 in case the value returned by *dtSim* is below or above a given threshold. The figure reports only the best curves that had been obtained running these variants with different parameters. In particular, in case of sigmoid, the center is set to 0.7, whereas about the cutting variant, the threshold is set to 0.99.

According to the results shown in Fig. 4, the mixing of compatibility and similarity improves the average precision of the system by +5.5% if running the cutting variant, while with the sigmoid variant the average precision increases by +1.6%. In addition, focusing on the Top-5 and Top-10 precision (Fig. 5), the cutting variant improves these parameters of about +3.7% and +2.4%, respectively. The sigmoid variant improvement is more limited: +0.6% +0.2%, respectively. These results suggest that overall, considering the compatibility in the matchmaking algorithm increases the precision of the service retrieval. Moreover, a binary approach when comparing two data types (as in the case of the cutting variant) is preferable to express compatibility.

Another interesting advantage of this new approach is its execution time. Indeed, one of the main limitations of URBE is its high response time. In particular, with the benchmark adopted in this validation, URBE required on average 53.26s to answer to a query, with each comparison taking about 49ms to be performed. Using the variants introduced in this paper, the response time has a significant decrease. More specifically, the average query response time with the sigmoid variant is 35.29s (-33.7%) whereas the response time for the cutting variant is 31.80 (-40.3%). Each comparison requires only 29ms. To compute the execution times, we executed the algorithm 10 times on the same machine. The reported values represent the average of the observations.

This interesting decrease of the response time is a result of the different way in which the elements composing the data types are compared. In baseline URBE, the comparison always involves a semantic similarity between the name of the complex types included in the services description. This requires to access the knowledge base collecting the terms and to calculate the similarity. With the new approach, the knowledge base is accessed only if the compatibility between the data types can not be assessed directly (as in the case of comparing simple and complex data types for example). This has a significant impact on the time required for calculating similarity, without nevertheless affecting the quality of the retrieved information.

V. RELATED WORK

Several works have been proposed in the literature for the evaluation of service compatibility. The approaches presented in [7], [8], [13], [14] for example, discuss (backward) compatibility as an enabler of controlled service evolution. As long the changes modifying a service respect a set of guidelines for backward compatibility, then the evolution of

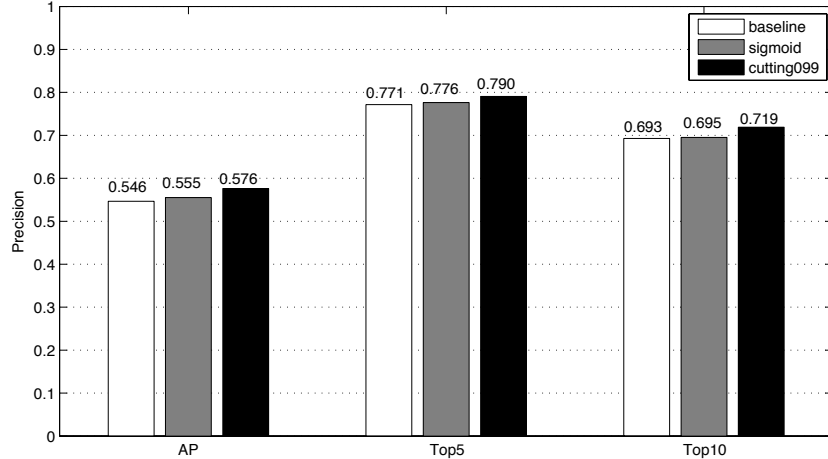


Figure 5. Average, Top-5, and Top-10 Precision.

the service leaves the service consumers unaffected. In this sense, new versions can substitute the older versions and the emphasis is on the vertical aspect of service compatibility. The approach presented here uses a theoretically-backed method based on type theory (as presented in [5]) in order to achieve the same goal.

As discussed in the previous sections, applying our approach in retrieving compatible Web services does not guarantee that the retrieved results can be used as-is and may require the use of an adapter for this purpose. Works like [15], [16], [17] and [9], among others, discuss the generation of such an adapter, ensuring the interoperability of the service with its consumers. While in these approaches it is not guaranteed that an adapter can be generated in the general case, the fact that our approach selects only compatible services can be interpreted as a guarantee that the required information for producing such an adapter is contained in the service descriptions [5].

Similarly to our approach, other works in the literature also rely on the syntax of the Web service description and compare the signature of the requested service with respect to the signatures of the existing Web services. This type of approach is closely related to the work in reusable components retrieval literature [18]. In this field, as stated by Zaremski and Wing, there are two types of methods to address this problem: signature matching [19] and specification matching [20]. In particular, signature matching considers two levels of similarity introducing the *exact* and *relaxed* signature matching. In our work, signature matching represents the core of the approach. In addition, our similarity algorithm also quantifies how similar a Web service is to another one, instead of simply dividing the retrieved Web services in exact matching and relaxed matching ones. Furthermore, as in the case of [2] and [3], our approach takes into account the structure of the service description for

the matchmaking process. In addition however, our approach considers the role of each description element with respect to the resulting compatibility between service descriptions.

VI. CONCLUSION & FUTURE WORK

In this work, we introduced an approach for service retrieval based on the analysis of service compatibility and service similarity. With respect to the former, a subtyping theory suitable for service description documents is adopted as a foundation for checking the compatibility of elements composing a service description. Concerning the latter, the URBE matchmaker is used as the baseline approach for defining how much two services are similar by reasoning on their interfaces. Combining these two ideas resulted in an updated URBE matchmaker. The validation of our approach demonstrates that the introduction of service compatibility improves the effectiveness of our service retrieval algorithm, not only in terms of precision and recall, but also in terms of response time. Both the original and the updated version of URBE are available as an open-source project at <http://sourceforge.net/projects/urbe/>.

Future work will concentrate on further improvement of the precision and recall by also considering the semantics of the data types defined by SAWSDL annotations. In the literature there exist other matchmaking algorithms, on which service retrieval is based, that perform better with respect to URBE (see for example [21]). Using the data type annotations in a semantic variant of URBE called URBE-S also results in overall good performance: the average precision with the SAWSDL benchmark is 0.749 [1]. The performance of these semantically-enabled matchmakers however heavily depends on the analysis of the annotations that characterize the SAWSDL service description. By integrating this semantic analysis to the compatibility-enabled analysis introduced in this paper, we aim at outperforming both URBE-S and the similar solutions.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community 7th Framework Programme under the Network of Excellence S-Cube Grant Agreement no. 215483.

REFERENCES

- [1] P. Plebani and B. Pernici, "URBE: Web service retrieval based on similarity evaluation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 11, pp. 1629–1642, 2009.
- [2] A. Zisman, G. Spanoudakis, and J. Dooley, "A framework for dynamic service discovery," in *23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Sep. 2008, pp. 158–167.
- [3] E. Stroulia and Y. Wang, "Structural and semantic matching for assessing Web-service similarity," *International Journal of Cooperative Information Systems*, vol. 14, no. 4, pp. 407–438, 2005.
- [4] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary Web service discovery mechanisms," *Journal of Web Engineering*, vol. 5, no. 3, pp. 265–290, 2006.
- [5] V. Andrikopoulos, *A Theory and Model for the Evolution of Software Services*. Tilburg, Netherlands: Tilburg University Press, 2010, no. 262.
- [6] D. Orchard *Ed.*, "Extending and versioning languages: XML languages [Editorial draft]," Jul. 2007. [Online]. Available: <http://www.w3.org/2001/tag/doc/versioning-xml>
- [7] K. Brown and M. Ellis, "Best practices for web services versioning," Jan. 2004. [Online]. Available: <http://www.ibm.com/developerworks/webservices/library/ws-version/>
- [8] K. Becker, A. Lopes, D. S. Milojicic, J. Pruyne, and S. Singhal, "Automatically determining compatibility of evolving services," in *ICWS 2008*, 2008, pp. 161–168.
- [9] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *Proceedings of the 16th international conference on World Wide Web*. Banff, Alberta, Canada: ACM, 2007, pp. 993–1002.
- [10] G. Pirró and N. Seco, "Design, implementation and evaluation of a new semantic similarity metric combining features and intrinsic information content," in *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1271–1288.
- [11] J. Farrel and H. Lausen, "Semantic annotations for WSDL and XML schema," <http://www.w3.org/TR/sawSDL/>, April 2007.
- [12] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
- [13] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen, and N. Du, "A version-aware approach for web service directory," in *International Conference on Web Services (ICWS) 2007*, Jul. 2007, pp. 406–413.
- [14] R. Weinreich, T. Ziebermayr, and D. Draheim, "A versioning model for enterprise services," in *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, vol. 2, 2007, pp. 570–575.
- [15] S. R. Ponnekanti and A. Fox, "Interoperability among independently evolving web services," ser. Lecture Notes in Computer Science. Toronto, Canada: Springer Berlin / Heidelberg, 2004, pp. 331–351.
- [16] P. Kaminski, M. Litoiu, and H. Müller, "A design technique for evolving web services," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, ser. CASCON '06. New York, NY, USA: ACM, 2006.
- [17] A. Brogi and R. Popescu, "Automated generation of BPEL adapters," in *ICSOC 2006*, ser. Lecture Notes in Computer Science. Springer, 2006, pp. 27–39.
- [18] E. Damiani, M. G. Fugini, and C. Bellettini, "A hierarchy-aware approach to faceted classification of objected-oriented components," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 3, pp. 215–262, 1999.
- [19] A. Zaremski and J. Wing, "Signature matching: a tool for using software libraries," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 2, pp. 146–170, 1995.
- [20] ———, "Specification matching of software components," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 4, pp. 333–369, 1997.
- [21] M. Klusch, U. Küster, B. König-Ries, A. Leger, D. Martin, M. Paolucci, and A. Bernstein, "4th international semantic service selection contest, performance evaluation of semantic service matchmakers," <http://www-ags.dfki.uni-sb.de/~klusch/s3/s3c-2010-summary-report-v2.pdf>, October 2010.