Structural and Semantic Similarity Metrics for Web Service Matchmaking*

Akın Günay and Pınar Yolum

Department of Computer Engineering, Boğaziçi University, Bebek 34342, Istanbul, Turkey
{akin.gunay,pinar.yolum}@boun.edu.tr

Abstract. Service matchmaking is the process of finding appropriate services for a given set of requirements. We present a novel service matchmaking approach based on the internal process of services. We model service internal processes using finite state machines and use various heuristics to find structural similarities between services. Further, we use a process ontology that captures the semantic relations between processes. This semantic information is then used to determine semantic similarities between processes and to compute match rates of services. We develop a case study to illustrate the benefits of using process-based matchmaking of services and to evaluate strengths of the different heuristics we propose.

1 Introduction

Web services are pieces of software that provide a functionality and can be invoked over the Web in a machine independent manner [1]. An important challenge in the usage of Web services is finding appropriate Web services for different service needs. Current standards, such as UDDI, only provide limited keyword search capabilities, which are insufficient to handle the requirements of the current users. With such protocols, users try to guess keywords that are relevant to their requests and Web services that advertise themselves with the same keywords are assumed to be good matches for each other.

Another influential trend in Web service matchmaking is that of input-output matching, where a service request is considered to match a Web service if the inputs and outputs are identical [2]. An enhancement to this approach is the addition of semantic information, where instead of identical matching partial matches are computed using the underlying semantic knowledge in the service descriptions [3][4]. In these semantic approaches, input-output fields are associated with semantic concepts represented in ontologies. The result of the matchmaking operation is a degree of semantic similarity, such as exact, plug-in and

* This research has been partially supported by Boğaziçi University Research Fund under grant BAP07A102 and the Scientific and Technological Research Council of Turkey by a CAREER Award under grant 105E073. The first author is supported by a Graduate Scholarship Program from the Scientific and Technological Research Council of Turkey.

subsumes match. Exact match shows that the request and service interfaces match exactly to each other. Plug-in match shows that the service returns a more general output concept than the requested. Subsumes match shows that the service returns a more specific output concept than the requested.

Although input-output matching is easy to implement, it has two important drawbacks.

- *Granularity*: The results of the matchmaking is coarse-grained. That is, the matching services are associated only with some general similarity degrees (exact, plug-in, and so on) and we cannot further discriminate between services that have the same similarity degree. This level of granularity is unacceptable, especially when the number of matching services is large. A better matching approach should provide more precise matching information and should be able to rank the services based on this rating.
- *Precision*: The matchmaking algorithm should provide high precision; meaning that those Web services that are actually labeled as matching should be compatible with the requests. Most input-output matching techniques suffer from low precision, since they do not consider the internal processes of services while performing the matchmaking operation [5]. As a result, different services with identical interfaces are counted as good matches although they perform completely different tasks.

Accordingly, in this paper we propose a novel service matchmaking approach that uses internal process models of the services to achieve good precision and recall performances, while providing fine-grained similarity degrees. To achieve this, we propose to represent the underlying processes of Web services and the requests for Web services as finite state machines (FSMs). We perform matchmaking of requests to Web services by comparing the two FSMs and measuring their similarity using different metrics. Further, we associate each atomic process with a semantic concept represented in a process ontology and compare the semantic similarity of atomic processes during matchmaking. Our matching results combine the structural similarity with semantic similarity and provide fine-grained scores.

The rest of this paper is organized as follows. Section 2 shows our modeling of services as FSMs. Section 3 explains our matching approach in detail. Section 4 explains our case study and elaborates on our evaluations. Finally, Section 5 reviews some relevant literature and provides directions for future work.

2 FSM for Service Modeling

An FSM [6] is a formalism to capture the flow of processes. Using FSMs we can model the fundamental control structures (sequences, choices and loops) of process flows. A finite state machine is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set called *states*, Σ is a finite set called *alphabet*, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final states.

A is the language (the set of all strings) that machine M recognizes and we show this by A $L(M)$.

In our approach, each element of the alphabet Σ represents an atomic process (or simply a process) of the modeled service. We use the states in Q to investigate the order of flow. Each transition function captures a process can be performed in the internal flow of the service. The start state is the entry point of the service and final states are the termination points. Each string in the language A is a sequence of consecutive processes that corresponds to a service flow.

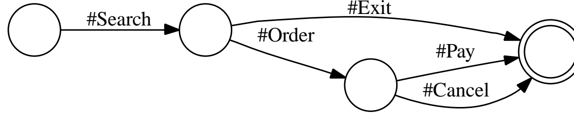

Fig. 1. A sample e-commerce service flow

Figure 1 shows an example of an e-commerce service's internal process flow modeled as an FSM. From the entry point (the start state) of the e-commerce service, the first process that we can perform is the `#Search` process, which returns some item(s) according to the requesters search criteria. According to the result of the `#Search` process, the requester may choose to buy the resulting item by performing `#Order` process or may end the interaction with the service by `#Exit` process. If the requester selects `#Order` process in the previous step, she may continue with the payment by performing the `#Pay` process or quit the service by canceling the order via `#Cancel` process. Figure 2 shows all possible flow sequences for the FSM presented in Figure 1.

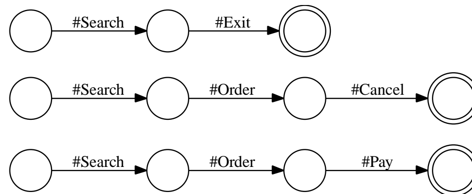

Fig. 2. All process flow sequences (language) generated by the FSM in Figure 1.

3 Matchmaking Approach

Given a service request, it is necessary to match it to a set of services from a pool of available services. To determine the similarity of a request to a particular Web service, we compute its *structural* and *semantic* similarity.

3.1 Structural Similarity

Since we use FSMs to model services, our first option to perform matchmaking is the use of formal definition of FSM equality. According to this definition, two FSMs are equivalent to each other if they recognize the same language. In our context this definition is too restrictive because the equality definition supports only services that exactly match each other. However, we are also interested in finding partially matching services as well as determining their match rates.

Because of these reasons we need a more flexible mechanism that can find partially matching services in addition to exact matches. Further, the matchmaking mechanism should assign a numeric similarity value to each service in order to differentiate between services. To achieve these goals, we use the following approach instead of the equality definition of the FSM.

Our approach is made up of the following steps:

1. Generate all possible flow sequences (all strings in the language) for the service and request using the associated FSMs.
2. Compare each sequence of the request against all the sequences of the service.
3. For each sequence comparison, compute a similarity value between the compared request and service sequence using a heuristic function (explained below).
4. Select the sequence with highest similarity value from service sequences.

Optionally, we can compute the scores using different heuristics and combine the results as the normalized sum of the selected pairs. We perform the above procedure for all the available services and sort the services in decreasing order according to their overall similarity values and return the top n percent as result.

To generate all flow sequences of the FSM, we expand the FSM from the start state up to the final states. This procedure is simple if the FSM structure is an acyclic directed graph. But if we introduce loops to the FSM, the graph turns to a cyclic directed graph where the number of possible sequences is infinite. To handle this case we modify our expansion algorithm so that it can detect loops and stop expanding a sequence when the same loop occurs more than once.

To compute the structural similarities we use four heuristics. These heuristics are *common process count* (CPC), *longest common substring* (LCStr), *longest common subsequence* (LCSeq) and *edit distance* (ED). Below, s_1 is a service request and s_2 is a Web service.

Common Process Count Heuristic The common process count (CPC) heuristic calculates the number of processes that appear in the service request and in given Web service sequences, without regard to the order of processes and normalizes the count with the total number of processes in the sequences. The underlying intuition is that when the number of common processes for the services increase, the two sequences are more similar to each other. The similarity between s_1 and s_2 is computed as follows where N_p is the number of common processes and N_{s_i} is the number of processes in s_i.

$$sim(s1, s2, CPC) = \frac{2N_p}{N_{s_1} + N_{s_2}} \quad (1)$$

Longest Common Substring Heuristic The LCStr heuristic [7] finds the longest common contiguous substring of two strings. In our context, this corresponds to the number of contiguous processes between the request and a Web service. Formally, using LCStr heuristic, similarity between s_1 and s_2 is computed and normalized as follows:

$$sim(s1, s2, LCStr) = \frac{length(LCStr(s_1, s_2))}{length(s_1)} \quad (2)$$

where $length(s)$ is a function that returns the length of the string s. $LCStr(s_i, s_j)$ is a function that returns the common longest substring of s_i and s_j.

Longest Common Subsequence Heuristic The LCSeq heuristic [7] finds the longest common subsequence (may not be contiguous) of two strings. In our context, this corresponds to finding the number of common processes between a request and a Web service without considering the contigiousness. Formally, using LCSeq heuristic, similarity of s_1 and s_2 is computed and normalized as follows:

$$sim(s1, s2, LCSeq) = \frac{length(LCSeq(s_1, s_2))}{length(s_1)} \quad (3)$$

where $length(s)$ is a function that returns the length of the string s. $LCSeq(s_i, s_j)$ is a function that returns the common longest subsequence of s_i and s_j.

Edit Distance (Levenshtein) Heuristic Edit distance [7] is the minimum number of operations needed to transform one string into another, where an operation is an insertion, deletion, or substitution of a single character. In our context the strings are the flow sequences and the characters are the processes. We formally calculate the similarity of s_1 and s_2 as follows:

$$sim(s1, s2, ED) = 1 - \frac{ED(s_1, s_2)}{Max(length(s1), length(s2))} \quad (4)$$

where $ED(s_1, s_2)$ is the function that computes the edit distance of the sequences and $Max(l_1, l_2)$ is the function that returns the maximum of two integers.

3.2 Semantic Similarity

Our structural similarity method works only on the syntactic level and do not take the underlying semantics into account. That is, if two Web services have entirely different atomic processes, the structural similarity of these services will be zero. However, if the atomic processes are related to each other, then the

two Web services can still be considered similar. For example, one Web service may use allow *payByCreditCard* process whereas a second Web service may use *payByCash* process to handle payment. For a structural similarity metric, these are two totally different processes. However, one can easily see that they are two variations of the same process and hence may substitute each other in many settings. To consider such subtleties, the relations between the processes should be captured. We do this using an ontology of processes that represents the meanings and relations of the processes. Each concept in this ontology corresponds to an atomic process. The children of a concept are the specializations of the process. In this ontology we assume that a more general process can perform all the tasks performed by its more specialized processes (sub-concepts). Figure 3 shows a part of our process ontology for the e-commerce domain.

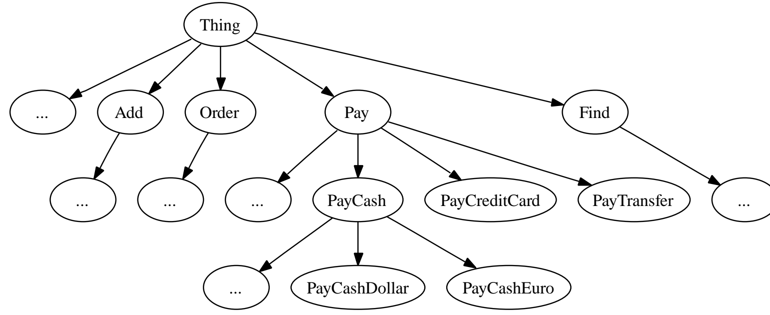

Fig. 3. A part of the process ontology for e-commerce domain.

To compute the semantic similarity of two concepts we develop a new semantic similarity metric called *semantic cover rate* (SCR). Using the SCR semantic similarity of two concept c_1 and c_2 is calculated as follows:

$$SCR(c_1, c_2) = \begin{cases} 1, & \text{if } c_1 \supseteq c_2 \\ \theta^{\|c_1, c_2\|}, & \text{if } c_1 \subset c_2 \\ \gamma^{\|R, c_1\|}, & \text{if } c_1 \not\supseteq c_2 \text{ and } c_1 \not\subseteq c_2 \text{ and } R \supset c_1 \text{ and } R \supset c_2 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where θ and γ are two control parameters in the range [0,1] and $\|c_i, c_j\|$ is the arc distance in the ontology between the concepts c_i and c_j. It is important to note that $SCR(c_i, c_j) \neq SCR(c_j, c_i)$.

In the first case, the request is a subset of a given Web service. Since we assume that a general process can perform all the tasks performed by its more specialized processes, we assign a SCR value of 1 for this case. However, if the service provides a more specific process than the requested process, the service process can handle only some of the needs of the requested process and therefore we assign a SCR value smaller than 1 by using θ parameter. In the third case, the

considered concepts are siblings. Although in this case there is no super or sub-concept relation between the two concepts, since these concepts have a common root they are still related to each other. However, in this case the resulting SCR value must be smaller than the previous case, since the relation between the concepts are weaker. We achieve this effect by assigning γ a smaller value than θ. In the last case, since there is no relation between the concepts, we assign zero as SCR value.

4 Evaluation

The following case study presents a service request and six services, which match to the request by different degrees.

4.1 Case Study Setup

Both the request and the services consist of only one flow sequence. We ignore the case of multiple choices and loops in our case study, since these control structures are converted into single sequences during the matchmaking process and do not have any effect on the computation of the similarity values.

Request: `#ConnectNonSecure` → `#SearchBook` → `#AddCartBook` → `#OrderCartBook` → `#Authenticate` → `#PayCreditCard`. This request is for a book shopping service. It requires a nonsecure connection and basic shopping functionalities like search and order. For payment it requires a credit card payment capability using any kind of authentication mechanism.

Service-1: `#ConnectNonSecure` → `#SearchBook` → `#Authenticate` → `#AddCartBook`. This service provides search and cart functionalities but it is not possible to order the card or make any payments. This service is a bad match for the request, since it does not provide most of the fundamental functionalities of the request like ordering the cart and payment.

Service-2: `#ConnectSecure` → `#SearchBook` → `#AddCartBook` → `#OrderCartBook` → `#AuthenticateHTTPS` → `#PayMoneyTransfer`. This service provides all the fundamental functionalities required by the request. However some processes that provide these functionalities are different than the requested service. For example the service provides the payment functionality by money transfer, which is different than the requested credit card payment. This service is an average match for the request, since it provides all the functionality to buy a book, but with some different processes.

Service-3: `#ConnectNonSecure` → `#SearchDVDBasicTitle` → `#AddCartDVD` → `#OrderCartDVD` → `#Authenticate` → `#PayCreditCard`. This service is to buy a DVD but not a book. Therefore this is not a good match for the request.

Service-4: `#Connect` → `#Search` → `#AddCart` → `#OrderCart` → `#CancelOrder` → `#Authenticate` → `#PayCreditCard`. This service is a general service where it is possible to search, order and pay for any consumable item. It is also possible

to cancel a given order, which is not required by the request. Since this service covers also purchase of a book, it is a good match for the request.

Service-5: `#ConnectNonSecure` → `#SearchBookBasicTitle` → `#AddCartBook` → `#OrderCartBook` → `#AuthenticateSSH` → `#PayCreditCardVisa`. This service provides more specific functionalities than the requested service. For example it accepts search only by book title, where the request looks for a service that can provide any type of search capabilities. This service can be accepted as an average match, since it mostly provides the requested functionalities.

Service-6: `#ConnectNonSecure` → `#Authenticate` → `#SearchBook` → `#AddCartBook` → `#OrderCartBook` → `#PayCreditCard`. This service provides exactly the requested functionality but in a different order. The authentication is performed after the connection instead of before payment, which causes shift of all processes in the flow order.

Overall, one would expect services 4 and 6 to be the better matches for the request. Next, we study the performance of our approach.

4.2 Results

To measure the individual performance of each structural heuristic first we compute the similarity between the request and all services for each heuristic separately. Than we integrate the SCR heuristic to each structural heuristic and perform the same procedure to observe the effect of the semantic knowledge. In addition to the individual heuristics, to observe the effect of the combination of different heuristics, we also compute the weighted linear combination of all the heuristics.

Table 1 presents our results. Each column shows the computed similarities between the request and each service using the associated heuristic. We add the letter S to the beginning of the heuristic names to indicate that they use SCR metric in addition to the structural similarity. The column named as Comb shows the results that we obtain by the linear combination of all heuristics (with equal weights). In the SCR computations we take θ as 0.75 and γ as 0.5. Considering

Table 1. Matchmaking results of each heuristic

	CPC	SCPC	LCStr	SLCStr	LCSeq	SLCSeq	ED	SED	Comb	SComb
Serv-1	0.60	0.94	0.50	0.33	0.75	0.50	0.33	0.46	0.55	0.56
Serv-2	0.50	0.79	0.50	0.79	0.50	0.79	0.50	0.79	0.50	0.79
Serv-3	0.50	0.69	0.33	0.69	0.50	0.69	0.50	0.69	0.46	0.69
Serv-4	0.15	0.86	0.14	0.67	0.14	1.00	0.14	0.86	0.15	0.85
Serv-5	0.50	0.84	0.33	0.84	0.50	0.84	0.50	0.84	0.46	0.84
Serv-6	1.00	1.00	0.50	0.50	0.83	0.83	0.67	0.67	0.75	0.75

these results, we observe the following:

- CPC is particularly useful when the request and the service have the same functionality but different process flows like in the case of Service-6. CPC can also successfully differentiate the unrelated services such as Service-3. However, it cannot detect that Service-1 is not a good match since CPC does not consider the difference between the number of processes in the request and the number of processes in the service.
- In general LCStr shows the worst performance compared to the other heuristics, since it is strictly dependent on the order of the process flow. It is only successful in the case of Service-1, where the service provides less functionality and therefore the longest common substring between the service and request is short.
- LCSeq is especially successful if the service covers the request and provides some additional functionality like in the case of Service-4. It can also successfully detect the two poor matches Service-1 and Service-3.
- ED can successfully differentiate between good and poor matches. The only exception occurs when flow orders are different like in the case of Service-6.
- Considering both the structural and semantic similarity gives more accurate results compared to the use of structural similarity alone.
- An intuitive approach to improve the quality of results is to use a combination of the heuristics instead of using them individually. The results obtained by the Comb prove this idea.

5 Discussion

In this study we propose a new semantic matchmaking approach for Web services that is based on the internal process flows of the services. Our approach combines structural similarity heuristics with a semantic similarity metric based on ontologies. To determine the similarity of two services, structural heuristics compare the individual atomic processes that are involved by the service flows. In this comparison if two atomic processes are identical, structural heuristics assign 1 as the similarity value and 0 otherwise. This approach restricts us to exactly matching processes. To be able to discover partially matching processes, we relax the similarity of non-identical processes into the range [0,1]. Our approach requires that both services and requests are modeled as FSMs by developers and users. Especially for the users developing a FSM of a service might be complex. However, in real life applications of this approach , we can count on the existence of support tools that can help users define their requests as FSMs.

Klusch *et al.* [5] extend the signature matching approach by using syntactic matching techniques from information retrieval on service descriptions in order to improve granularity and precision performance. Although this approach provides a mechanism to associate each service with a numeric similarity value, it still suffers from low precision. Additionally, Dong *et al.* [8] state that syntax based information retrieval techniques are not efficient for Web service matching, since in most of the real world situations service descriptions do not contain enough textual data that is required by syntactic methods to work properly.

There are other approaches for matchmaking that use process modeling techniques. Klein and Bernstein [9] propose an indexing mechanism to create a hierarchical ontology of process models and develop a query language to perform matching on the created ontology. Wombacher *et al.* [10] propose another approach which also use FSA to model Web services. Different than our approach they concentrate on the syntactic level matching of FSA. They do not consider any ontologies for semantics or do not work on a rank mechanism or partial matching. Addition to discovery processes modeling, FSA is also studied for other Web service issues like composition. Berardi *et al.* [11] propose a service composition approach that can work with time constraints.

In our future work, we plan to optimize SCR performance by deciding on the δ and γ values at run time. This will enable us to achieve more accurate matches. Another interesting direction is the investigation of fuzzy is-a relations to represent the domain. The weights again may be learned at runtime to improve personalized match performance.

References

1. Singh, M.P., Huhns, M.N.: Service-Oriented Computing: Semantics, Processes, Agents. John Wiley & Sons, Chichester, UK (2005)
2. Zaremski, A.M., Wing, J.M.: Specification matching of software components. In: SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, New York, NY, USA, ACM Press (1995) 6–17
3. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of Web services capabilities. In: Proceedings of the First International Semantic Web Conference, Springer-Verlag (2002) 333–347
4. Sycara, K., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. Autonomous Agents and Multi-Agent Systems **5**(2) (2002) 173–203
5. Klusch, M., Fries, B., Sycara, K.: Automated semantic web service discovery with owls-mx. In: AAMAS '06: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, New York, NY, USA, ACM Press (2006) 915–922
6. Sipser, M.: Introduction to the Theory of Computation. 2 edn. Course Technology (2005)
7. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
8. Dong, X., Halevy, A.Y., Madhavan, J., Nemes, E., Zhang, J.: Simlarity search for web services. In: VLDB. (2004) 372–383
9. Klein, M., Bernstein, A.: Toward high-precision service retrieval. IEEE Internet Computing **8**(1) (2004) 30–36
10. Wombacher, A., Fankhauser, P., Mahleko, B., Neuhold, E.: Matchmaking for business processes based on conjunctive finite state automata. International Journal of Business Process Integration and Management **1**(1) (2005) 3–11
11. Berardi, D., Giacomo, G.D., Lenzerini, M., Mecella, M., Calvanese, D.: Synthesis of underspecified composite e-services based on automated reasoning. In: Proceedings of the 2nd International Conference on Service Oriented Computing, ACM Press (2004) 105–114