Web Service Search: Who, When, What, and How

Jianguo Lu[1], Yijun Yu[2]

[1]School of Computer Science, University of Windsor
jlu@cs.uwindsor.ca

[2] Computing Department, The Open University
y.yu@open.ac.uk

Abstract: Web service search is an important problem in service oriented architecture that has attracted widespread attention from academia as well as industry. Web service searching can be performed by various stakeholders, in different situations, using different forms of queries. All those combinations result in radically different ways of implementation. Using a real world web service composition example, this paper describes when, what, and how to search web services from service assemblers' point of view, where the semantics of web services are not explicitly described. This example outlines the approach to implement a web service broker that can recommend useful services to service assemblers.

Keywords: Web service searching, web service composition, signature matching, XML Schema matching

1. Introduction

Web service reuse is the number one drive for service oriented architecture. To reuse web services, it is paramount to develop web service repository architectures and searching methods. There have been tremendous researches on web service searching [2, 4, 7, 16]. However, in many cases, web service searching means different things for different people. Before implementing web service searching platforms and methods, we need to discuss who needs to search web services, when searches happen exactly, what are the queries to be sent out, and, once the queries are formulated, how to execute the queries.

This paper delineates various stakeholders in web service searching, and tries to give answers to the above questions using a real world web service composition example, where the semantics of web services are not explicitly described. In this example, we constructed a real web service from five atomic services. During the integration process, various searches are carried out in order to find relevant and reusable services in this scenario.

This paper details web service searching from a web service assembler's perspective. An assembler starts from an abstract description for the composite web service. From the description, an initial query is constructed in the form of service signature, i.e., the name of the operation and its input and output types. Based on the

search results and the current service signature, subsequent queries are derived. As service signatures are well-structured in XML, such queries can be found using approximate XML Schema matching [8].

2. Cube of web service searching

Unlike web pages that are presented for humans to read, web services are meant to be invoked by programs. Hence web services are usually searched by programmers, or sometimes by software agents that can automatically adapt their behavior by using new services. Either way, web services are consumed by programs.

Different stakeholders search for web services for different purposes, using different resources, and in different ways. Main stakeholders in web service searching can be categorized as follows:

1) *Web service end users:* End users are programmers who search for web services in order write a program to invoke them directly as is.
2) *Web service assemblers*: web service assemblers search for web services in order to compose them to perform some tasks that cannot be fulfilled by a single service. Once reusable atomic services are found, assemblers can use conventional programming languages to compose the services, either manually or supported by service composition tools.
3) *Web service brokers*: web service brokers are programs that assist web service assemblers by recommending relevant web services during the assembly process. Just the same as various code recommendation systems for conventional programming languages [24], web service brokers can watch over the shoulders of assemblers and are able to recommend services proactively according to existing code that has been written by service assembler.
4) *Web service agents:* They are intelligent programs that are able to automatically find relevant web services to use at system run time, when a new task occurs or when existing web services is not functioning properly and a replacement is called for.

The classification of various web service searches can be depicted in Figure 1. In addition to the main stakeholders in web service searching, there are a variety of forms of queries to search web services, including:

1) A set of keywords [5, 17];
2) Signature or part of the signature of the service [25, 17];
3) Context of the service to be used [24, 22, 23];
4) Semantic description of the service [16, 17].

These different kinds of queries form the Y axis in Figure 1.

Another dimension is *when* the searches are carried out. Roughly speaking, web services searching can happen at development time or run time. For web service end users and assemblers, web service searching happens at development time and is initiated by humans. Service brokers can recommend the services proactively while a

web service is being developed. Service agents will search and consume the services dynamically at run time. In this case the service agent needs to have the complete semantic description of the service in order to conduct the correct search without human intervention.

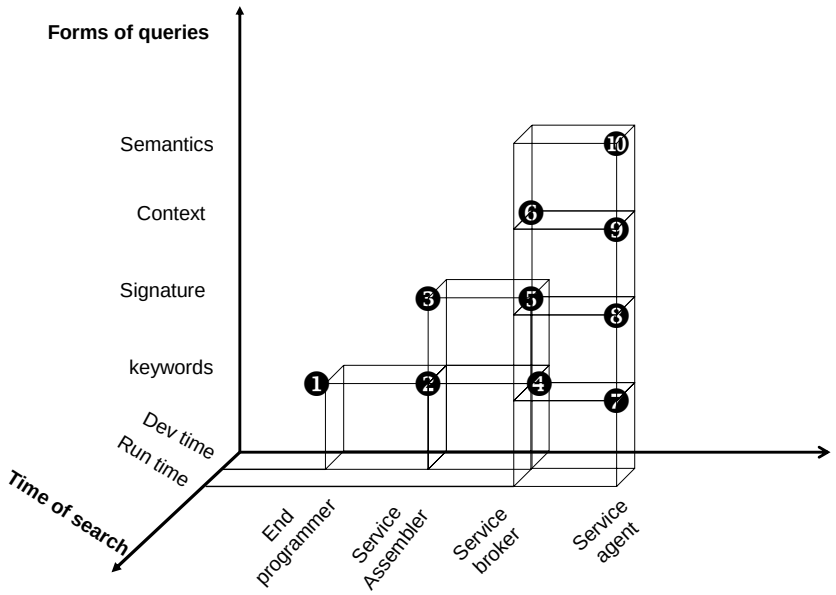

Fig. 1. Varieties of web service searches

Combinations of the three parameters (i.e., who, what, and when) constitute the variety of the searches. However, not every combination makes sense. For example, for end-user programmers or service assemblers, searches should happen in development time instead of run time, and usually keyword based search is more convenient (search type ❶). If the number of returns is large, maybe signature search can be performed to narrow down the results. Although semantic description is essential in determining equivalent web services automatically, for end user programmers it is neither necessary nor practical to write semantics such as ontology or functional specification to search for services.

On the other hand, if an intelligent agent wants to replace an existing service, semantics for the services must to provided in order to decide whether they are performing the same task (search type ❿ in Figure 1). In this case, search happens at run time instead of development time.

For service assemblers, keywords or signatures may be enough since programmers can judge whether the search results are good. Besides, it would be too cumbersome for programmers to spell out the semantics or the context of the web services in details. Hence the search type could be ❶ and ❸.

For a service broker, the program has the knowledge of the current code that service assembler has written, hence it has the context the service will be used (type ❻ search). This context, including other services already used and even the documentation, can be utilized to recommend the next service to be used [24, 1, 22].

Each combination determines how the search should be implemented. For example, search type ❶ is usually implemented by information retrieval methods such as vector space model. Type ❿ includes inferences on ontology and functional specifications [17, 16].

3. When do we need to search for web service

Since web service searching can happen in many different situations, it is not possible in this paper to discuss all of them in details. In the following anatomy of web service searching, we will focus on search type ❸ in Figure 1, i.e., we suppose that web service assemblers will search for the services. We will discuss exactly when we need to search the services, and what the queries will be.

As a running example, let us start with the following task for service assemblers:

Given a zipcode, find its closest airport name. *(1)*

Given the large number of web services that are available on the web, it is reasonable to assume that there should be a solution for such a problem. But how to solve the problem is by no means obvious. Before starting to write the program, service assemblers should first formalize the problem. Following the conventional program specification methods, the task could be formalized as follows by defining the concepts *zipcode*, *airport* and "*closest*":

Description 1 for all *Zipcode*, find *Airport*, such that

1) $isAirport(Airport) \wedge distance(Zipcode?, Airport ?, Distance)$,

and for any other airport *Airport'*,

2) $isAirport(Airport') \wedge distance(Zipcode?, Airport'?, Distance') \rightarrow Distance' \geq Distance.$ (2)

Here *distance* and *isAirport* are two predicates that need to be refined so that they can correspond to some web services. There are arguments in the predicates. For example, in *distance(X?, Y?, Z)*, *X*, *Y*, and *Z* are arguments in the predicate. An argument with a question mark adornment such as *X?* denote that the value of argument *X* needs to be provided. Arguments without a question mark such as *Z* denote the returned value after the service is executed.

If there is an existing service that implements (2), then the task has been fulfilled. Otherwise, which is true in this case, we need to refine (2) into subtasks (2.1) and (2.2) that may be implemented by existing Web services.

isAirport(Airport) *(2.1)*

distance(Zipcode?, Airport?, Distance) *(2.2)*

The task *isAirport(Airport)* returns a set of airports without any input. The other task *distance(Zipcode?, Airport?, Distance)* accepts a zipcode and an airport code, and returns the distance between them.

At this stage, web service assembler needs to search for those two services. Searching for a service for the *isAirport(Airport)* specification using signature

isAirport: → Airports

doesn't return an exact match. However, a similar service (http://www.farequest.com/ FASTwebservice.asmx?WSDL [1]) can be found, whose signature (i.e., the name of the operation and its input and output types) is

stateAirport: stateAbbr→Airports,

where *Airports* is the Schema for *airport(code, city, state, country, name)**. The predicate representation of the service is

stateAirport(StateAbbr?, Airports) *(2.1')*

As a service assembler, what is the next service to search for? Next section will give more cases as for when searches are carried out as development of the composite web service unfolds. Searching for this kind of similar services is also not a trivial task. Section 5 will discuss in more details regarding how to find this kind of related services.

4. What are the queries to search for the relevant web services

Even though now we are assuming using type 3 search and the queries are in the form of signatures, it is not always clear what the queries are exactly. In the running example, once we have the problem description, service assemblers know that we need to search for the predicates referred in the specification, such as *isAirport*. The query in the form of signature is *isAirport: → Airports.* In other cases queries to be issued may not be straightforward, as we will see in the next section.

4.1 Query formulation

By issuing the query *isAirport: → Airports*, we can find the service *stateAirpor(StateAbbr?, Airports).* At this stage we cannot invoke the *stateAirport* service yet in our composite service since it needs to use a *StateAbbr* as input in order to return the airport data. In order to obtain the state name, we need a service in the following signature:

→ <stateAbbr/> *(S1)*

Or, we can utilize some known values from our existing input list. Currently, the only

[1] All the web services listed in this paper are active during the month of April 2007. As web services are volatile, some of them may not be functioning now.

input is *ZipCode*. Hence we can search for a service of the following signature as an alternative:

$$<ZipCode/> \rightarrow <stateAbbr/>* \qquad \text{(S2)}$$

From here and hereafter, we omit the service name in signature when it is not important.

Now using these two signatures S1 and S2 to search for web services, we found the following web service(http://www.farequest.com/FASTwebservice.asmx?WSDL):

zipState(ZipCode?, State),

Whose signature is <ZipCode/> → <State/>*

Up to this stage, is*Airport(AirportCode)* is refined into

zipcodeState(ZipCode?, StateAbbr)
∧ stateAirport(State?, AirportCode)

Generalizing from this example, the service assembler can use the following rule to form the query:

$$\frac{A \rightarrow B}{\rightarrow C} \qquad B \rightarrow C \qquad A \in \Sigma \qquad \text{(Rule 1)}$$

The meaning of the rule is that to derive a service of signature →C, suppose that we already have a service of signature B→C, and suppose we have A in the known list, we need to find a service of signature A→B.

Now Description 1 is refined as the following:

Description 2 for all *Zipcode*, find *Airport*, such that

2) *zipcodeState(Zipcode?, State) ∧ stateAirport(State?, AirportCode) ∧ distance(Zipcode?, AirportCode?, D),* and
3) for any other AirportCode',
   zipcodeState(Zipcode?, State)
   ∧ stateAirport(State?, AirportCode')
   ∧ distance(Zipcode?, AirportCode'?, D')
   → D' ≥ D. *(3)*

In Description 2, predicates *zipcodeSate* and *stateAirport* correspond to two real web services. Before Description 2 can be implemented, the predicate *distance* needs to be refined further into a real service. Similar to the previous steps, first we search for *distance(Zipcode?, Airport?, Distance)*. There is no exact match again. The closest match is the following service (http://ws.cdyne.com/psaddress/addresslookup.asmx?wsdl):

calculateDistanceMiles(latitute1?,longitude1?, latitude2?, longitude2?, DistanceInMiles)

Whose signature is

calculateDistanceMiles: (latitute1,longitude1, latitude2, longitude2)
→ DistanceInMiles

Since the inputs *latitude* and *longitude* in this service are not in the known list, we need to find a service that provides those parameters, i.e., we need to find a service that is compatible to the following signature:

<Zipcode | State/> → <latitude/><longitude/>

And

<AirportCode | State/> → <latitude/><longitude/>

We add <AirportCode>, <Zipcode> and <State> in the input type because those values are already available at this stage.

To formalize the process of generating the above query, we need Rule 2 as below:

$$\frac{A \to B}{A \to C} \qquad B \to C \qquad \textit{(Rule 2)}$$

The meaning of the rule is that to derive a service of type A→C, suppose that we already have a service of type B→C, then we need to find a service of type A→B so that A→B and B→C can be composed into a service which is of type A→C.

In general, Rule 2 can seldom be applied directly. A general form would be the following:

$$\frac{A_1 |D_1| ... |D_n \to B_1 \quad A_2 |D_1| ... |D_n \to B_2}{(A_1, A_2) \to C} \qquad (B_1, B_2) \to C \quad D_i \in \Sigma \qquad \textit{(Rule 3)}$$

Rule 3 means that to find a service of type (A1, A2)→C, and if we already have found a service of type (B1, B2)→C, what we need is to find a service of type A1|D1|…|Dn→B1, and A2|D1|…|Dn→B2, where Di is the type of available values.

Corresponding to Rule 3, we need to find a service of type

(*Zipcode, Airport)* → *Distance*.

And suppose that we have already found a service of type

(latitute1, longitude1, latitude2, longitude2) → *Distance*

Hence the services we need to search for should be compatible to the following types:

<Zipcode | State/> → <latitude/><longitude/>

And
<AirportCode | State/> → <latitude/><longitude/>

Using those two queries, the following two web services are found:

airportCoordinate(AirportCode?,
LatitudeDegree,LatidudeMinute, LongitudeDegree, LatitudeMinute)

zipCodeCoordinate(ZipCode?, LatDegrees, LonDegrees).

Using those two web services, the *distance* predicate is refined into the following three services (predicates):

airportCoordinate(AirportCodeCode? LatitudeDegree, LatitudeMinute,
LongitudeDegree, LongitudeMinute)
∧ zipCodeCoordinate(ZipCode?, LatDegrees, LonDegrees).
∧ calculateDistanceMiles(latitute1?,longitude1?, latitude2?,
longitude2?, Distance)

Using the above three web services found, Description 2 is derived into the following:

Description 3 for all *ZipCode*, find *Airport*, such that
1) zipcodeState(Zipcode?, State) ∧ stateAirport(State?, AirportCode)
∧ airportCoordinate(AirportCode?LatitudeDegree, LatitudeMinute,
LongitudeDegree, LongitudeMinute)
∧ zipCodeCoordinate(ZipCode?, LatDegrees, LonDegrees)
∧ calculateDistanceMiles(latitute1?, longitude1?, latitude2?,
longitude2?, Distance)
and for any other *AirportCode*',
2) zipcodeState(Zipcode?, State) ∧ stateAirport(State?, AirportCode')
∧ airportCoordinate(AirportCode'?LatitudeDegree, LatitudeMinute,
LongitudeDegree, LongitudeMinute)
∧ zipCodeCoordinate(ZipCode?, LatDegrees, LonDegrees)
∧ calculateDistanceMiles(latitute1?,longitude1?, latitude2?,
longitude2?, Distance')
→ Distance' ≥ Distance (4)

Now that all the predicates in the composite web service definition refer to existing web services, we can generate the program to glue those web services. The integration program can be written in existing general purpose programming languages such as Java, or in languages for web service composition such as BPEL. The overall picture of the composition is depicted in Figure 2.

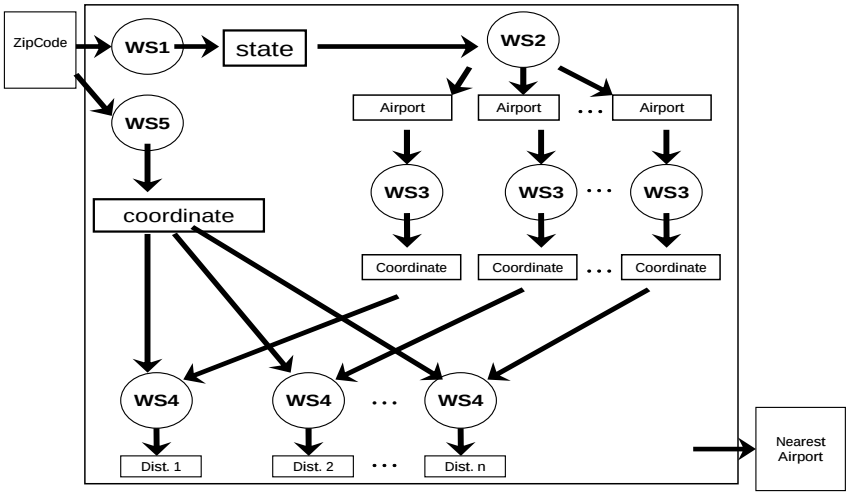

Fig. 2. The composite web service

5. How to search for relevant services

Now that we have described the ways to formulate the queries, the next task is to construct a query to locate relevant services. Given our first query for example, it is not a trivial task to run the query "*isAirport:* → *Airport"* in order to find the approximate matching

stateAirport: stateAbbr → *airport(code, city, state, country, name)**

Note that there are at least two issues need to be tackled. One is the matching between tag names in XML Schema. For example, <state/> should be matched with <stateAbbr/>. The other is the matching between the structures of the schemas. For example, *Airport* in Description 1 needs to be matched with the structure *(airport(code, city, state, country, name))*.*

In order to find approximate signature matchings, we need to construct a matching algorithm between XML Schemas, since input/output types in web services are described in XML Schema. Because XML Schemas are trees, we reduced schema matching problem to the classic tree matching problem [8], and developed Common Substructure algorithm to find the matching effectively. Instead of giving rigorous definitions for such matching problem, we use the following example to illustrate the problem and the solution.

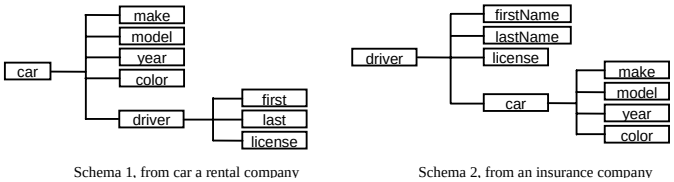

Figure 2. Car-Driver Schemas

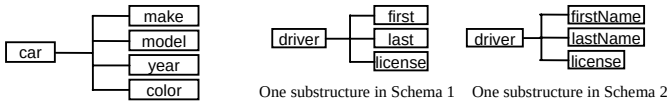

Figure 3. A common substructure (left) and two similar substructures (middle and right figures)

Figure 2 shows two similar schemas. In order to find the matches, we locate the largest common substructures as described in Figure 3. In the system, we use Wordnet to capture the synonyms. In addition, composite tag names such as *StateAbbr* is broken into a word list (*State, Abbr*).

6. Conclusions

There have been tremendous researches on web service searching. The notion of web service searching varies greatly. We classify various searches in terms of the stakeholder who will initiate the search. In the case of web service assembler, we described in detail as for when the searches are needed, what are the queries should be issued, and how the queries should be executed, by locating five real world web services that are needed in creating a new web service. In particular, we give the formal rules to derive the service queries in the process of service composition. This formalism can be used to implement a service broker that can recommend the services to programmers, i.e., the search type ❺ in Figure 1. If context information is included, the service broker can be expanded to search type ❻.

This paper outlines the plan for implementing a web service broker, illustrated using a concrete real word web service composition example. While service assembler refines the definition of a composite web service, service broker recommend the relevant atomic services that could be used, mainly rely on approximate signature matching between atomic web services and the tasks at hand. We have already implemented the XML Schema matching system [8] as our first step in implementing such a system. The query will be automatically generated using the rules outlined in this paper. In addition, context information will be used to increase the precision of recommendation.

Our early work on a generic matching system encompasses all kinds of queries ranging from keywords to signatures and ontology in the form of description logic [17]. While it is a comprehensive matching system involving various matching

algorithms, yet we need to answer the questions such as who will use the system, and how the queries are formed. This paper is a step towards answering those questions in one particular scenario.

Acknowledgements We would like to thank Debashis Roy and Deepa Saha for implementing the composite web service, and the anonymous reviewers for their helpful comments.

References

1. Agarwal, V., Dasgupta, K., Karnik, N., Kumar, A., Kundu, A., Mittal, S., and Srivastava, B. 2005. A service creation environment based on end to end composition of Web services. In *Proceedings of the 14th international Conference on World Wide Web* (Chiba, Japan, May 10 - 14, 2005). WWW '05. ACM Press, New York, NY, 128-137.
2. Benatallah, B., Hacid, M., Leger, A., Rey, C., and Toumani, F. 2005. On automating Web services discovery. *The VLDB Journal* 14, 1 (Mar. 2005), 84-96.
3. Tevfik Bultan, Jianwen Su, Xiang Fu: Analyzing Conversations of Web Services. IEEE Internet Computing 10(1): 18-25 (2006)
4. Caverlee, J., Liu, L., and Rocco, D. 2004. Discovering and ranking web services with BASIL: a personalized approach with biased focus. In Proceedings of the 2nd international Conference on Service Oriented Computing (New York, NY, USA, November 15 - 19, 2004). ICSOC '04. ACM Press, New York, NY, 153-162.
5. X. Dong, A. Halevy, J. Madhavan, E. Nemes, J. Zhang. Similarity Search for Web Services. Proc. of VLDB, 2004.
6. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web and Grid Services 1 (2005) 1-30
7. Elgedawy, I., Tari, Z., and Winikoff, M. 2004. Exact functional context matching for web services. In *Proceedings of the 2nd international Conference on Service Oriented Computing* (New York, NY, USA, November 15 - 19, 2004). ICSOC '04. ACM Press, New York, NY, 143-152.
8. Jianguo Lu, Ju Wang, Shengrui Wang, XML Schema Matching, IJSEKE, International Journal of Software Engineering and Knowledge Engineering, in Press.
9. Jianguo Lu, Yijun Yu, John Mylopoulos, A Lightweight Approach to Semantic Web Service Synthesis, ICDE Workshop, International Workshop on Challenges in Web Information Retrieval and Integration, Tokyo, 2005.
10. Matskin, M. and Rao, J. 2002. Value-Added Web Services Composition Using Automatic Program Synthesis. In *Revised Papers From the international Workshop on Web Services, E-Business, and the Semantic Web* (May 27 - 28, 2002). C. Bussler, R. Hull, S. A. McIlraith, M. E. Orlowska, B. Pernici, and J. Yang, Eds. Lecture Notes In Computer Science, vol. 2512. Springer-Verlag, London, 213-224.
11. McIlraith, S.A., Son, T.C.: Adapting golog for composition of semantic web services. In: Proc. of the 8th Int. Conf. on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France. (2002)
12. Medjahed, B., Bouguettaya, A., Elmagarmid, A.K.: Composing web services on the semantic web. The VLDB Journal 12 (2003) 333-351.
13. ProgrammableWeb, http://www.programmableweb.com.
14. Ponnekanti, S.R., Fox, A.: SWORD: A developer toolkit for web service composition. In: Proc. of the 11th Int. WWW Conf. (WWW2002), Honolulu, HI, USA. (2002)

15. Rao, J., Su, X.: A survey of automated web service composition methods. In: Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition, SWSWPC2004, LNCS, San Diego, USA. (2004)
16. E. Sirin, B. Parsia, and J. Hendler, Composition-driven filtering and selection of semantic web services, AAAI Spring Symposium on Semantic Web Services, 2004.
17. K. Sycara, J. Lu, M. Klusch, Interoperability among Heterogeneous Software Agents on the Internet, Technical Report CMU-RI-TR-98-22, CMU, Pittsburgh, USA.
18. Wong, J. and Hong, J. I. 2007. Making mashups with marmite: towards end-user programming for the web. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (San Jose, California, USA, April 28 - May 03, 2007). CHI '07. ACM Press, New York, NY, 1435-1444.
19. Wu, D., Parsia, B., Sirin, E., Hendler, J.A., Nau, D.S.: Automating DAML-S web services composition using SHOP2. In: Proc.of the 2nd Int. Semantic Web Conf.(ISWC2003), Sanibel Island, FL, USA. (2003)
20. Yijun Yu, Jianguo Lu, Juan Fernandez-Ramil, Phil Yuan, Comparing Web Services with Other Software Components, International Conference on Web Services, ICWS 2007.
21. Zhang, L., Chao, T., Chang, H., and Chung, J. 2003. XML-Based Advanced UDDI Search Mechanism for B2B Integration. *Electronic Commerce Research* 3, 1-2 (Jan. 2003), 25-42.
22. Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. 2005. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 12 - 15, 2005). PLDI '05. ACM Press, New York, NY, 48-61.
23. Wong, J. and Hong, J. I. 2007. Making mashups with marmite: towards end-user programming for the web. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (San Jose, California, USA, April 28 - May 03, 2007). CHI '07. ACM Press, New York, NY, 1435-1444.
24. Yunwen Ye and Gerhard Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," Proceedings of 2002 International Conference on Software Engineering (ICSE'02), Buenos Aires, Argentina, (to appear), May 19-25, 2002
25. Amy Moormann Zaremski , Jeannette M. Wing, Specification matching of software components, ACM Transactions on Software Engineering and Methodology (TOSEM), v.6 n.4, p.333-369, Oct. 1997.