A Test-Driven Approach to Behavioral Queries for Service Selection

Laura Zavala[1], Benito Mendoza[2], and Michael N. Huhns[3]
[1]*Computer Science and Electrical Engineering*
University of Maryland Baltimore County, Baltimore, MD, USA
rzavala@umbc.edu
[2]*Computer Engineering Technology*
New York City College of Technology, Brooklyn, NY, USA
bmendoza@citytech.cuny.edu
[3]*Computer Science and Engineering*
University of South Carolina, Columbia, SC, USA
huhns@sc.edu

ABSTRACT
Although the areas of Service-Oriented Computing (SOC) and Agile and Lean Software Development (LSD) have been evolving separately in the last few years, they share several commonalities. Both are intended to exploit reusability and exhibit adaptability. SOC in particular aims to facilitate the widespread and diverse use of small, loosely coupled units of functionality, called services. Such services have a decided agility advantage, because they allow for changing a service provider at runtime without affecting any of a group of diverse and possibly anonymous consumers. Moreover, they can be composed at both development-time and run-time to produce new functionalities. Automatic service discovery and selection are key aspects for composing services dynamically. Current approaches attempting to automate discovery and selection make use of only structural and functional aspects of the services and, in many situations, this does not suffice to discriminate between functionally similar but disparate services. Service behavior is difficult to specify prior to service execution and instead is better described based on experience with the execution of the service. In this chapter, we present a behavioral approach to service selection and runtime adaptation that, inspired by agile software development techniques, is based on behavioral queries specified as test cases. Behavior is evaluated through the analysis of execution values of functional and non-functional parameters. In addition to behavioral selection, our approach allows for real-time evaluation of non-functional quality-of-service parameters, such as response time, availability, and latency.

INTRODUCTION
A methodology for software development based on services as fundamental building blocks, known as service-oriented computing, has become widely used in building enterprise systems, because it greatly enhances their flexibility and adaptability. As a further incentive for this methodology, the number of publicly available services is continuing to increase and the Internet is becoming an open repository of such atomic heterogeneous software components. Multiple services can be integrated to facilitate cooperation between various business parties, achieve agility of the business integration, and even provide value-added services for service consumers. Essential to these capabilities is the detection of

entities, services, and other resources that can be used for satisfying a specification of desired functionality. The precision of the selection process improves the possibility of having services that find, connect, and communicate with one another automatically, sharing information and performing tasks without human intervention.

Web services are the current most promising instantiation of the service-oriented methodology. Web services comprise infrastructure for describing service structure, via WSDL (Christensen, Curbera, Meredith, & Weerawarana, 2001); specifying semantics and functionality via WSDL-S (Akkiraju et al., 2005), OWL-S (Martin et al., 2004), and WSMO (Roman et al., 2006)); supporting a service repository, via UDDI (Clement et al., 2004) or some less structured registry; interacting with services, via SOAP (Gudgin et al., 2007); and scheduling and orchestrating, via WSCL (Banerji et al., 2002) (Barry, 2003; Wombacher, Fankhauser, & Mahleko, 2004). The relationships among these components are depicted in Figure 1.

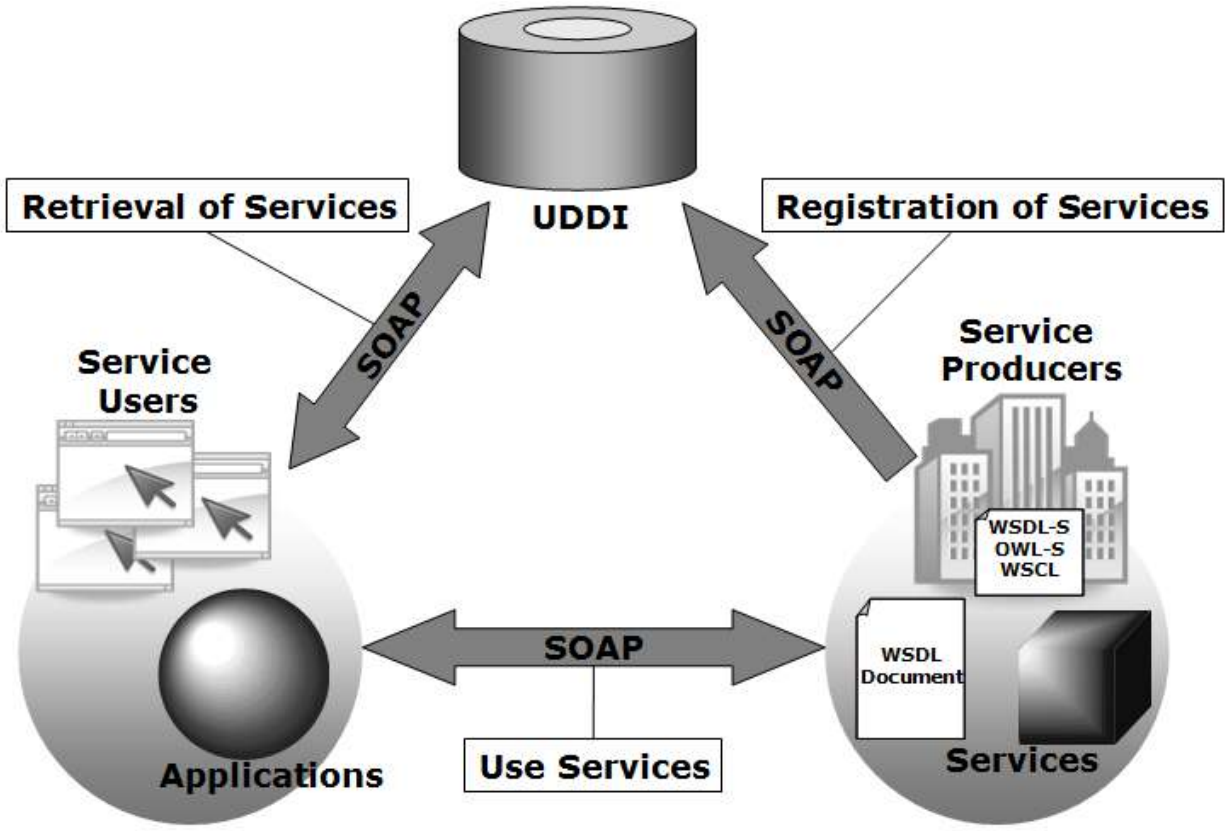

Figure 1. The general architectural model for Web services

Table 1. Different levels of description for a Web service

Description of Service	Current Representation Standard/Technique
Structure: syntactic	WSDL
Structure: semantic	WSDL and OWL, WSDL-S, OWL-S, WSMO
Function	WSDL-S, OWL-S, WSMO
Behavior	Unit Testing (our approach)

There are three orthogonal dimensions for describing a service at the knowledge level (Table 1): (1) structure, (2) function, and (3) behavior. Current approaches for automating discovery and selection of services make use of only the first two. Syntactic and semantic search based on keywords on the structural definition of the service, usually the WSDL content, are used for service discovery in repositories. Semantic descriptions of service inputs and outputs are used for the selection of services. WSDL-S, OWL-S, and WSMO are the most significant standards for such semantic descriptions.

WSDL-S, OWL-S, and WSMO are also used in attempts to describe the functionality of a service or a composition of services. They describe inputs, outputs, message flows, sequences of actions, etc., which constitute a *static* model for the services being selected. They do not have the capability to describe the dynamics of the services and how they will behave when they are invoked. Behavioral selection of services should be used to provide a more precise selection. That is, it is important to know not only *what* a service will do, but also know *when* the service will provide its functionality and *how well* it will be provided. "When" and "how well" are descriptive parts for the *behavior* of a service. Behavior is difficult to specify prior to a service's invocation and often can only be described accurately based on experience with the execution of the service. Built-in tests are one way to acquire the needed experience. The underlying idea is to furnish service clients with the ability to test the services to validate not only that they do what has been stated in their specifications, but also that what they do is indeed what the client needs them to do. Additionally, services can be tested at runtime to assure that they have not changed or, if they have, they are still appropriate. That is, behavioral tests have the added advantage of being useful for the management of services over their lifetimes.

Take for example the widely adopted similar-parameters technique for the semantic matching of services. It works by estimating the degree of similarity between the expected and the actual service's input and output parameters. However, even if these parameters are semantically similar, their execution values would still provide valuable guidance for a decision-making process determining service adequacy for the task. WSDL-S attempts to deal with this problem by providing a reference from each input and output parameter to a mutually agreed upon ontology. But this requires the existence of such an ontology or a way to reconcile the independently developed ontologies of the service requestor and the service provider, neither of which might be widely available. Moreover, ontologies rarely include constraints on the execution values of those parameters and, if such constraints were made available, the values would be context dependent and generalizable only with great difficulty.

Over time, even if a developer initially selects a correct service, the provider might update the service, such as by requiring an additional input. The application using this service would then fail, and the offending service would have to be identified and replaced. Frequent retesting of services or a more complex monitoring mechanism can help avoid such situations and retesting of previously discovered services can lessen the incurred overhead of adaptation. Further, dynamic reconfiguration facilitates the building of self-healing systems (Brazier, Kephart, Parunak, & Huhns, 2009) that adapt to changes in the environment and in requirements.

We present an approach, inspired by agile software development (Martin, 2002), to the behavioral selection of services based on the use of test cases to evaluate a service's behavior (through the analysis of execution values of functional and non-functional parameters), enabling an informed decision about which of the discovered candidate services from a repository are appropriate. The tests can also be used to assess performance and reliability. Therefore, in addition to behavioral selection, our framework allows for real-time evaluation of non-functional quality-of-service (QoS) parameters (e.g., response time, availability, and latency), scalability, and dynamism (a change in the client's requirements and/or the service's API). Finally, the approach can be applied for runtime behavior monitoring and adaptation of service oriented applications, thus providing a foundation for autonomic, self-managing, self-healing, self-optimizing, and self-adaptive applications.

BACKGROUND

Our approach builds upon the existing general architectural model for Web services and uses existing service discovery and selection mechanisms to find candidate Web services. It is inspired by agile software development and uses concepts and tools from test driven development to verify the behavior of the candidate Web services initially found. It allows for runtime adaptation given changes in the availability, interaction, and behavior patterns of the services. In this section we provide an overview of the key concepts in these areas, particularly the ones used in our work, and provide pointers to the reader to more comprehensive explanations.

Semantic Service Discovery and Selection

Service discovery is the process of finding a set of suitable services for a given task. Selection consists of choosing, from the discovered set of services, the one that best matches the requirements for the task. The search and selection can be done manually or automatically.
In order to make services discoverable to potential consumers, a provider can explicitly register a service with a registry, which is sometimes based on UDDI. In order that a consumer can use a service, providers usually augment a service endpoint with an interface description using the Web Services Description Language (WSDL).

Approaches to service discovery and selection are based on syntactic and semantic matching of a service specification with representations of the structural and/or functional description of a candidate service. Syntactic approaches match service descriptions based on keywords or interfaces. Semantic approaches extend and adapt the vocabulary used to describe services in order to give semantic meaning to the terms used in the description (e.g., input and output parameters and the names of operations). This is usually achieved through the use of semantic annotations and ontologies, as well as logical reasoning mechanisms. In general, these semantic matching solutions have provided important research directions in overcoming the limitations present in prior purely syntactic approaches for service matching. A more detailed discussion on service discovery and selection can be found in (Singh & Huhns, 2005). It covers the principles and practice of Web services and relates all concepts to practical examples and emerging standards. Barry (2003) contains a higher level view of Web Services and Service-Oriented Architectures and general discussion on implementing Web service projects. Existing works and approaches to service discovery, selection, and composition are discussed in the *Related Work* section.

Test-Driven Development

Agile software development is an iterative and incremental approach that proposes the development of software in small incremental releases or iterations. Each iteration is like a miniature software project of its own and includes all of the tasks necessary to release the mini-increment of new functionality: planning, requirements analysis, design, coding, testing, and documentation. The goal is to write code faster while increasing code quality (Martin, 2002). The best known agile method is Extreme Programming, which comprises several core practices, such as Simple Design, Continuous Integration, Collective Code Ownership, Pair Programming, Design Improvement, Small Releases, and Test-Driven Development (Ron, 2011).

Test-driven development (Beck 2002) is an evolutionary (iterative and incremental) approach to software development that relies on the repetition of a very short development cycle: before writing any piece of functional code, developers write a failing automated test case that defines a desired improvement or new function, then write just enough code to fulfill that test and finally refactor (restructure) the new code to acceptable standards.

Test-driven development breaks testing into two levels: developer tests and acceptance tests (also known as customer tests). Developer tests are unit tests written by the developers as the production code is written. The tests can be thought of as white-box testing. Inspection of the state of the code in the unit under test is important in this type of testing. Acceptance tests are more like black-box tests. They test that a feature is functioning properly. For example, a customer test for an application that has a feature to generate a graph could be: given inputs x and y, the graph should appear like z. Acceptance tests are usually specified by customers in a high level specification language, such as FitNesse[1] and FIT[2], and generally involve testing the whole system (e.g. interactions between system and user, testing multiple screens, etc.) instead of single units. If only a single entry point needs to be tested, acceptance tests can also be expressed as unit tests. In fact, some acceptance test frameworks (e.g., FIT) use JUnit[3] on the backend but allow the users to define test cases in some high level format (e.g., HTML tables), which are later converted to unit cases in JUnit.

Various xUnit frameworks (e.g., JUnit for Java, nUnit for .NET, and PyUnit for Python) have been developed to facilitate the creation of unit tests for specific programming languages. With the use of these frameworks, all a developer must do is set up a minimal program structure and call the function being tested. The program structure is very intuitive and comprises actions to be performed before and after the test, as well as preconditions and post-conditions. A unit test can be run automatically and, as it checks the results it obtains, it can provide simple and immediate feedback as to whether the tests on the unit passed or failed.

Test-driven development has the advantage that it separates the unit test from the module being tested. The unit tests can be written even before the module to be tested is coded. In this way, developers do not have to modify their program to include debugging statements within the program itself. Because the debugging statements are in a separate program, they can be changed without having to recompile the program. With acceptance tests, the test cases do not even have to be written by the developer of the program; someone other than the original developer can write the tests, execute them, and interpret the results. Indeed, it is possible to combine tests from various developers and run them together without concern about interference.

Runtime Service Behavior Monitoring and Adaptation

The management of service-oriented applications is a complex task due to the lack of central control that results from combining services from different providers. Changing the availability, interaction, and behavior patterns of services, results in undesired failures in the applications. Runtime service verification aims at detecting such problems, rather than detecting implementation errors within services, and is therefore validation-oriented rather than verification-oriented. Boehm (1984) distinguishes verification and validation as follows: “When a service user is connected to a service provider it needs to determine whether it is using the *right* service rather than whether the service it is using is *right*.”

BEHAVIORAL SERVICE SELECTION AND MAINTENANCE

Looking only into the structural and functional aspects of the services does not suffice to discriminate between functionally similar but disparate services. We suggest the use of behavioral selection of services to provide more precise results. Service behavior, that is, how the service behaves when invoked, is difficult to specify *prior* to its execution and often can only be described accurately based on experience with the execution of the service.

---

[1] www.fitnesse.org
[2] fit.c2.com
[3] www.junit.org

Since acceptance test-driven development (i.e. based on acceptance tests) allows for verification of behavior conformance at runtime, it provides a basis for behavioral selection of Web services. In (Zavala Gutierrez, Mendoza, & Huhns, 2007), we have proposed an approach to behavioral service selection and maintenance based on behavioral queries specified as test cases. The test cases are expressed in a high-level specification formalism and automatically mapped to unit tests. The unit tests are for validation —lightweight checking of behavior conformance at runtime, as opposed to verification of correct execution to detect errors in the implementation. We assume that the services have been adequately tested during development and any identified faults fixed. When services are selected at runtime, therefore, any problems that arise will more likely be due to behavior misunderstandings about the functionality of the services than implementation errors in them. Behavior is evaluated through the analysis of execution values of functional and non-functional parameters. Our approach not only improves the accuracy of selecting services, but also could work as a mechanism for runtime adaptation. In this section, we provide a discussion of our approach, a formal model for specifying expected service behavior, and two examples of the application of the model.

Test-Driven Evaluation of Service Behavior

The specific part of test-driven development that we are using is acceptance tests, since we do not want to inspect the state of the code in the service under test, which would likely not be available anyhow. We are only interested in verifying whether the service is behaving as expected. Behavior is evaluated through a defined set of behavioral constraints. The behavioral constraints verify that the execution values of functional and non-functional parameters are in the expected range for a particular chosen set of scenarios (the values of input parameters). The behavioral constraints are expressed in validation-oriented unit tests as preconditions and post-conditions. In addition to behavioral selection, our framework allows for real-time evaluation of non-functional QoS parameters, such as response time, availability, and latency.

Formal Model for Expressing Expected Behavior

Service behavior is defined as the performance of the service (outputs values, response time, etc.) under different scenarios (i.e., assignment of values to the input parameters). We have developed a formal model for expressing the expected behavior of a Web service. For this model, we define a Web service, *WS*, as a tuple <*I*, *FP*, *NFP*, *scenario*, *constraints*>, where:

- *I* is the set of input parameters
- *FP* is the set of functional parameters (the Web service outputs)
- *NFP* is the set of non-functional parameters (e.g., response time and availability)
- *scenario* is an assignment of values to the input parameters
- *constraints* is a conjunction of restrictions or behavior, $B_1 \wedge B_2 \wedge \ldots \wedge B_n$, that must hold under *scenario*.

Each constraint B_i is a tuple containing the following elements {*parameter*, *expectedRange*, *relevance*} where:

- *parameter* is an element of either *FP* or *NFP*
- *expectedRange* is a pair of values (*min*, *max*)
- *relevance* is an indicator of the importance of the constraint

I and *FP* are used for semantic matchmaking in a UDDI-based repository of available Web services. *Scenario* and *constraints* make use of *I*, *FP*, and *NFP* to specify the expected behavior of the service by indicating expected ranges of values (*expectedRange*) for functional and non-functional parameters (*FP* and *NFP*) under specific situations (*scenarios*). Figure 2 shows how the test cases and ranking table are incorporated in the application part of the general model for Web services.

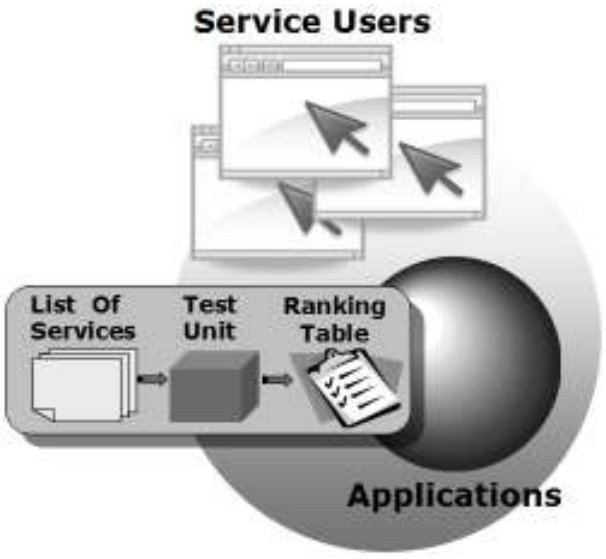

Figure 2. Extensions to the general architectural model for services to support behavioral queries

Stock Quoting and Purchasing Example
Suppose we want to find a service for purchasing financial stocks. Using current discovery techniques, we might find a list of candidate services with two inputs (a string identifying the stock and an integer specifying the number of units) and an output (a real specifying the price). There is no guarantee that all the services do in fact correspond to the financial domain. One of the services could, for example, provide a quote for purchasing livestock (the string input identifying the type of livestock). If you invoke the service for a particular stock (e.g., "IBM") for which you wish to know its price, you would expect the result to be in a certain range (e.g., between 120.0 and 180.0), while the price for livestock would be in a different range.

Using our formalism for expressing expected behavior and giving the same relevance to each of the constraints, a particular instance of WS for this example is:

I = {*stockSymbol*, *numberOfUnits*},
FP = {*price*}, NFP = {*availability*, *responseTime*},
scenario = {*stockSymbol*="IBM", *numberOfUnits*=10}, and
constraints = {B_1, B_2, B_3}, where:
B_1= {*price*, (120, 180), 1}
B_2= {*responseTime*, (0, 5000), 1}
B_2= {*availability*, (80, 100), 1}

Hotel Search Example
Suppose a developer wants to find a service that maintains a catalog of hotels along with a search component that accepts hotel criteria as inputs. Also suppose that there exists a local travel agency and a local basketball association that provides semantically equivalent services based on the hotel service. However, there is a potential difference between the two. The first one returns all the hotels available to tourists that match the search criteria, while the second one returns only those hotels that both match the search criteria and have special arrangements with the local basketball association.

Using our formalism for expressing expected behavior, a particular instance of WS for this example is:

I = {zipcode, distance},
FP = {number_of_hotels_found}
scenario = { zipcode ="21250", distance=10},
and constraints = {B1}, where:
B1= { number_of_hotels_found, (10, 15), 1}

This example sets the constraint based on the number of hotels found by the service for a particular zip code. We could also set constraints on other features, such as the price given for each hotel and the availability of other methods for reserving or getting further information. In the rest of this chapter we focus on a stock quoting and purchasing example.

Service Selection and Maintenance

Two phases are involved in the selection and maintenance of services:

- **Design-time phase**: a unit test for a WS is created from its expected behavior specification.
- **Run-time phase**: the unit test is used for either selection of Web services from a list of candidates or maintenance of the system.

Service Selection. Our approach builds upon the existing general architectural model for Web services (Figure 1). Syntactic and semantic structural searches are first performed to find candidate Web services in a repository, which is possibly UDDI-based. A syntactic search is basically a keyword search on the structural definition of the service (the WSDL content). A semantic search performs a semantic match on the operation, input, and output elements, possibly represented in OWL or RDFS. Then, the behavior of each candidate Web service is compared with the expected behavior. This is done by running a unit test for each candidate. A sound analysis of the results obtained from the test is used to determine the relevance of the candidates. Specifically, the candidate Web services are ranked according to a similarity measure that represents the similarity of the candidate to the expected service behavior specification. The similarity is a function of the number of behavioral constraints met and their relevance. Figure 3 shows a sequence diagram of this process.

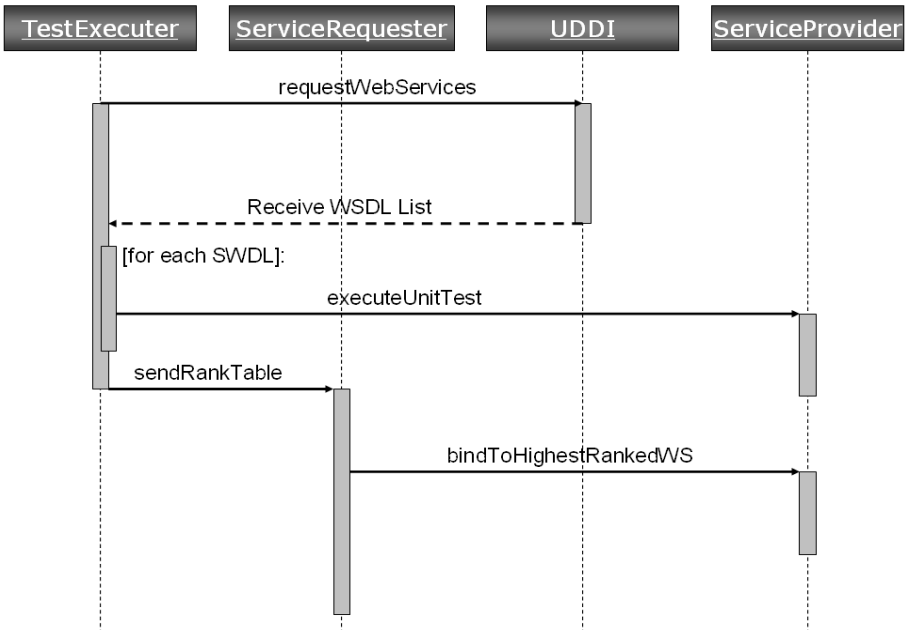

Figure 3. A sequence diagram showing the process of requesting, testing, and selecting a service

Service Maintenance. The ranked list of candidate services obtained during the selection phase can be stored and used later for maintenance purposes. Specifically, if at some point in the future a service that was selected as the most relevant one causes problems during execution (due probably to temporary or permanent removal of the service by the provider), the next candidate is selected automatically. Alternatively, the service that caused the problem could be retested with the unit test created for it during the original selection process to detect possible causes of the failure.

Issues and Applicability

An issue that needs to be considered when using the approach described herein is that some scenarios would have to be constantly reevaluated, since the expected execution values might be valid only for some time period (e.g., hotel rates and stock quotes), after which the tests would need to be revised and updated. This could be addressed by using a central authority (third party) to provide mostly cached results for test queries that clients can execute to test the service. For queries whose results frequently change (like stock quotes), the queries could be updated on a daily basis and only a small set of queries would be allowed (e.g., only queries for a small predetermined set of stocks). The advantage of a third party is that it could additionally allow the incorporation of reputation mechanisms, so that clients could rank the services based on results obtained from them. There have been some efforts addressing this idea, such as (Vu, Hauswirth, & Aberer, 2005) and (Wang & Vassileva, 2007).

Evaluating service behavior through the analysis of execution values of functional and non-functional parameters provides the basis for a new type of service selection paradigm. The approach works in scenarios where users have an idea of what the output values for specific cases should be. Additionally, it allows specifying constraints for the evaluation of non-functional QoS parameters. Its applicability for some other situations, however, would require modification of existing business models so that clients can run behavioral queries. Such is the case of commercial services that require payment for access to them. Commercial providers would need to provide a way for requesters to execute test queries without having to pay. For example, tokens can be granted for testing purposes and expire after a certain number of uses or after a specified period of time. Allowed queries could be restricted to only a few for testing purposes. Security, trust, and other issues arise, which have to be considered and investigated to assure that the mechanism is not being abused (free riders). Some commercial service providers currently offer an option for testing the service by invoking allowed method calls under testing mode. Such is the case for the stock quoting Web services found at *xmethods.net*.

Another issue regarding commercial services is the cost of the service. Even if there was a mechanism in place for allowing behavioral queries, clients would need to know the cost of using the service if it turned out to be a good match for their needs. To make use of the cost of a service would require a contracting protocol with a legal authority, which is not part of the service's behavior. Therefore, the issue is out of the realm of a behavioral service selection approach.

One case where the applicability of the behavioral queries might not be adequate is the search for Web services that perform actions and have side effects, such as printing documents or buying products. However, this is a common issue in current approaches to automated service selection and composition. In theory, the behavioral queries or any other automated selection approach could be used but certain effects might be undesired for testing purposes.

The use of validation-oriented test units to specify behavioral constraints provides a means not only to conduct selection of semantically discovered services, but also to achieve the self-healing capability envisioned in autonomic computing (Brazier, Kephart, Parunak, & Huhns, 2009). By having dynamic reconfiguration using the service ranking table, the system would adapt to changes in the environment and requirements (runtime adaptation). For example, when the highest ranked candidate service does not

behave as expected (e.g., it is temporarily unavailable, there are changes in the API, or there are login and payment requirements), the next service in the ranking could be selected. Furthermore, the first one could be retested to find the cause of the problem and a report could be sent to the administrator or stored in a log file.

AN EXAMPLE IMPLEMENTATION IN JUNIT

The fundamental aspect of our behavioral approach is the specifications of service behavior in the proposed formalism and the semantics of such specifications. The parsing of the specifications, as well as the implementation of the test cases to verify behavior compliance does not have to follow any specific methodology or use any specific language.

A search for "Stock Quoting" services with I = {*stockSymbol*}, FP = {*lastPrice*}, NFP = {*responseTime*} returned the Web services available at:

1. *http://www.webservicex.net/stockquote.asmx?WSDL*
2. *http://www.restfulwebservices.net/wcf/StockQuoteService.svc?wsdl*
3. *http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?WSDL*
4. *http://www.gama-system.com/webservices/stockquotes.asmx?WSDL*

These are the discovered services to which behavioral selection is applied to verify and select the one(s) with the expected behavior. The expected behavior and behavioral constraints expressed in our formal model are: *scenario* = {*stockSymbol*="IBM"}, and *constraints* = {B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}}, where B_1={*price,* (*120, 180), 10*}, B_2={*responseTime,* (*0, 1000*)*, 9*}, B_3={*responseTime,* (*0, 1500*)*, 8*}, B_4={*responseTime,* (*0, 2000*)*, 7*}, B_5={*responseTime,* (*0, 2500*)*, 6*}, B_6={*responseTime,* (*0, 3000*)*, 5*}, B_7={*responseTime,* (*0, 3500*)*, 4*}, B_8={*responseTime,* (*0, 4000*)*, 3*}, B_9={*responseTime,* (*0, 4500*)*, 2*}, B_{10}={*responseTime,* (*0, 5000*)*, 1*}.

This behavioral information was captured in an XML file (Figure 4), which is passed as input to the unit test generator. More visual means could be used to capture the behavioral specification, such as HTML tables as used in existing frameworks for acceptance tests (e.g., FitNesse). The XML could then be automatically generated.

We used JUnit for the implementation of the unit cases and Apache Axis WSDL2Java[4] for the automatic generation of client stubs to invoke the Web services. The behavioral constraints are expressed as assertions in the unit test. Figure 5 shows the source code of the unit test in JUnit. *Client* is an instance of the service requester class, which is the one tested by the unit test. Figure 6 shows the source code in Java for the service requester class.

After executing the unit test, we have a list of which assertions (behavioral constraints) were met by each candidate service. We calculate the similarity of the candidate to the expected behavior with a function, *rank(WS)*, of the relevance of the behavioral constraints met:

$$rank(WS) = \sum_{1 \le i \le n \mid WS(B_i)=true} relevance\big(WS(B_i)\big)$$

Among the four candidate services found for this example, the first three satisfied the first constraint, B_1. The fourth service did not satisfy B_1. This service returns -4 when getting a quote for "IBM". It turns out

[4] ws.apache.org/axis/java

that this service only provides quotes for two stock exchanges, NASDAQ and LJSE (Slovenia Stock Exchange). IBM is listed on the NYSE and thus is not found by this service, returning an error code instead. The corresponding *responseTime* for each of the tested services was 1: 922ms, 2: 1287ms, 3: 1595ms, and 4: 1254ms. Only the first candidate passed all the tests, obtaining the highest rank, 95. The other services missed one or two constraints giving them a lower ranking, as shown in Table 2. Since we would like to maintain as candidates only those services that find the testing *stockSymbol* within the corresponding price range and with a maximum *responseTime* of 5000ms, we discard all the services with a ranking equal to or less than 50. That is, a Web service could be used only if it satisfies B_1 and at least one of the other constraints.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<scenario>
   <parameter>
      <name>stockSymbol</name>
      <value>IBM</value>
   </parameter>
</scenario>

<constraints>
   <constraint>
      <id>B1</id>
      <parameter>lastPrice</parameter>
      <type>double</type>
      <range>
         <min>120</min>
         <max>180</max>
      </range>
      <relevance>50</relevance>
   </constraint>

   <constraint>
      <id>B2</id>
      <parameter>responseTime</parameter>
      <type>integer</type>
      <range>
         <min>0</min>
         <max>1000</max>
      </range>
      <relevance>9</relevance>
   </constraint>
   .
   .
   .
</constraints>
```

Figure 4. A fragment of an XML document containing the representation of the constraints for the Stock Quote example

Table 2. Ranking table for the Web services in the Stock Quote example

WS	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	rank(ws)
1	50	9	8	7	6	5	4	3	2	1	95
2	50	0	8	7	6	5	4	3	2	1	86
3	50	0	0	7	6	5	4	3	2	1	78
4	0	0	8	7	6	5	4	3	2	1	36

```
import java.util.Date;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class StockQuoteClientTest {
    static StockQuoteClient client;
    static double price;
    static long responseTime;

    public StockQuoteClientTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
        price = 0.0;
        responseTime = 0;
        client = new StockQuoteClient();
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        price = 0.0;
        responseTime = 0;
    }

    @Test
    public void testB1() {
        System.out.println("getQuote");
        assertEquals(0.0, price, 0.0);
        price = client.getQuote("IBM");
        assertTrue(price >=120.0 && price <= 180.0);
    }

    @Test
    public void testB2() {
        Date d1 = new Date();
        System.out.println("testResponseTime");
        assertEquals(0.0, responseTime, 0.0);
        price = client.getQuote("IBM");
        Date d2 = new Date();
        responseTime = d2.getTime() - d1.getTime();
        assertTrue(responseTime >= 0 && responseTime <= 1000);
    }
 .
 .
 .
    @Test
    public void testB10() {
 .
 .
 .
}
```

Figure 5. A fragment of the JUnit code of a test case for the Stock Quote example

```
import javax.xml.ws.WebServiceRef;
import net.restfulwebservices.servicecontracts._2008._01.StockQuoteService;
import net.restfulwebservices.servicecontracts._2008._01.IStockQuoteService;
import net.restfulwebservices.datacontracts._2008._01.StockQuote;

public class StockQuoteClient{
    @WebServiceRef(wsdlLocation = "WEB-
INF/wsdl/www.restfulwebservices.net/wcf/StockQuoteService.svc.wsdl")
    private StockQuoteService service;

    public double getQuote(String stockSymbol){
        double lastPrice = 0.0;
        StockQuote result;
        try {
            service = new StockQuoteService();
            IStockQuoteService port =
service.getBasicHttpBindingIStockQuoteService();
            result = port.getStockQuote(stockSymbol);
            lastPrice = Double.valueOf(result.getLast().getValue());
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
        return lastPrice;
    }
}
```

Figure 6. A client in Java for the Stock Quote example

RELATED WORK

Semantic Service Discovery and Selection

There have been a number of efforts that use ontologies, description logic (DL), and logic reasoning approaches for semantically matching services. The matchmaking framework presented in (Chakraborty, Perich, Avancha, & Joshi, 2001) uses a DAML-S based ontology for describing the services. A DL reasoner is used to classify the matches for a given request in order to get an indication of how good a match is. Matches are classified into one of its five degrees of match, which are: exact, plug-in, subsume, intersection, and disjoint. This is achieved by computing the subsumption relationship of the request description with respect to all the advertisement descriptions. The issue of semantic ambiguity between the terms used to describe services is addressed in (Akkiraju, Srivastava, Ivan, Goodwin, & Syeda-Mahmood, 2006). Using cues from domain-independent and domain-specific ontologies, they compute an overall semantic similarity score between ambiguous terms in order to find matching service descriptions. Chakraborty et. al. (2001) proposed semantic matching approaches for pervasive environments, which use ontologies to describe the services and a Prolog-based reasoning engine to facilitate the semantic matching. They define an ad hoc heuristic-based criterion for judging the closeness between the service advertisements and the request, and provide approximate matches if no exact match exists for the given request. Ontologies are used in (Wang & Stroulia, 2003) to describe and select Web services for composition by comparing the Web service output parameters with the input parameters of other available Web services. Similarly, (Paolucci, Kawamura, Payne, & Sycara, 2002) presents a service discovery approach based on DAML-S, where the semantics of input and output parameters of the provided and required services are compared in order to calculate the degree of similarity between the two services. A

markup language called USDL (Universal Service Description Language) has been proposed for describing the semantics of Web services formally (Simon, Mallya, Bansal, Gupta, & Hite, 2005). This approach uses WSDL to give a syntactic description of the name and parameters of a service, while a specialized universal OWL ontology is used to formally describe what these mean on a conceptual level. In (Ye & Zhang, 2006), semantic annotations based on domain-oriented functional ontologies are proposed to discover Web services with functional semantics. Predefined terms are used in (Sivashanmugam, Verma, Sheth, & Miller, 2003) to express pre- and post-conditions. Ontologies are used in (Xiao, Zou, Ng, & Nigul, 2010) to define the elements of contexts that are relevant to find users' needs and recommend appropriate services. Ontologies are also used in (Bandara, Payne, De Roure, Gibbins, & Lewis, 2008) to describe the requests and to facilitate the discovery of device-based services in pervasive environments. The approach includes a ranking mechanism that orders services according to their suitability and also considers priorities placed on individual requirements in a request during the matching process.

Some approaches aim at modeling the function of a service using automata or another mechanism that allows specifying the sequence of execution. In (Lei & Duan, 2005), the OWL-S process model specifying the "behavior" of a Web service is transformed into an extended deterministic finite-state automaton (EDFA). The approach in (Shen & Su, 2005) combines the use of automata for handling input and output messages with the use of OWL-S for describing semantics. A query language and query evaluation algorithms for the proposed formalism are also provided. Agarwal and Studer (2006) present a formalism based on pi-calculus to describe the functionality of Web services in annotated WSDL documents. A matchmaking algorithm that makes use of such annotations is also described. An approach for ranking semantic Web service advertisements with respect to a service request is presented in (Skoutas, Simitsis, & Sellis, 2007). Ranking is based on the use of a domain ontology to infer the semantic similarity between the parameters of the request and the advertisement. The approach is applicable to several types of ontologies, ranging from simple taxonomies to highly expressive ontologies, such as OWL ontologies.

Another commonly studied paradigm is to model the sequence of messages in the conversation protocol that the Web service follows. In (Grigori, Corrales, & Bouzeghoub, 2006), graphs representing the conversation protocol or model are used to specify interactions with a service. The problem is thus reduced to a graph matching one. An error-correcting matching algorithm allows an approximate matching. An approach in (Wang & Vassileva, 2007) makes use of message sequences to describe the interactions with services; therefore, messages can be exchanged successfully between the compatible services. However, this approach must be performed under an assumption that elements of two message sequences to be compared must come from the same WSDL document to guarantee that the same message names have identical semantics. Moreover, if the messages describe objects instead of simple data types, then consideration of how objects are serialized into the messages is crucial.

Finally, some approaches focus on QoS parameters. The work in (Srivastava & Sorenson, 2010) introduces a technique that compares functionally equivalent services on the basis of customers' perceptions of the QoS attributes rather than the actual attribute values. The goal is to assign weights that reflect not only the actual QoS attribute values, but also their importance on the basis of the customers' preferences. In (Vu, Hauswirth, & Aberer, 2005), it is assumed that providers advertise the service quality and that users provide feedback on the actual levels of the QoS delivered to them. They address the issue of detecting and dealing with false ratings by dishonest providers and users. Non-functional properties are used in (Braun, Strunk, Stoyanova, & Buder, 2008) to describe QoS as well as context of service execution. They consider two types of context information: measurable data with a certain range and a certain unit, such as the resolution in dots per inch and the queue length of a printer service; non-measurable data such as the quality (laser or inkjet), and the color depth of printed documents. A QoS

broker-based architecture for Web service selection in (Serhani, Dssouli, Hafid, & Sahraoui, 2005) defines a broker entity for the verification and certification of service qualities. Details of how selection is done are not specified, but rather left to the client that would use the information that the broker holds. The broker makes multiple concurrent calls to the Web service to check that the operations described in the service interface are available and, at the same time, calculate QoS metrics (availability, response time, and processing time). This related work is the closest to our approach. However, the work only considers non-functional parameters (QoS metrics). The values of functional parameters obtained during execution are not used and, thus, behavior is observed only from a performance point of view.

Other initiatives, such as the Business Process Execution Language for Web Service (BPEL4WS) and the OWL-S Service Model, are focused on representing service compositions (i.e., plans) where flow of a process and bindings between services are known a priori. Furthermore, approaches to automated service composition address the problem of automatically generating the plan and usually combine AI planning algorithms with semantic approaches. The focus in this paper is on service selection and runtime adaptation. Service selection is performed after discovery and prior to composition. Runtime adaptation is used when some of the selected services fail or become unavailable. Interested readers should look at (Rao & Su, 2004) for a survey of approaches to automated Web service composition.

Runtime Service Behavior Monitoring and Adaptation

Irmert, Fischer, and Meyer-Wegener (2008) present a framework, a middleware that allows replacing service implementations at runtime; that is, to add new functionality or change the functionality of a service in a Service-Oriented Architecture (SOA) environment at runtime without any side effects on the applications that are currently using it.

Atkinson, Brenner, Falcone, and Juhasz (2008) present an approach that aims to complement the semantic based service composition methods with test sheets that software engineers write and read and explicitly describe specific sequences of operation invocations representing one more usage scenarios in a visual and intuitive way. The test sheets are human readable and help application engineers, at design time, to carry out tests for quality assurance. Additionally, the authors argue that these semantically self-contained tests can be directly executed at runtime to validate a service provider's contract compliance qualitatively or to assess a service provider's reliability quantitatively.

Cardellini and Iannucci (2010) present a framework architected as a service broker that supports the QoS-driven runtime adaptation of SOA applications offered as composite services to users. It acts as an intermediary between users and concrete services, performing a role of service provider towards the users and being in turn a requestor to the concrete services used to implement the composite service. Its main task is to drive the adaptation of the composite service it manages to fulfill the Service Level Agreements (SLAs) negotiated with its users, given the SLAs it has negotiated with the concrete services.

FUTURE RESEARCH DIRECTIONS AND CONCLUSION

Semantic formalisms that capture the functional description of Web services provide a way for automating the discovery, selection, and matchmaking of services. However, the accuracy of such an automatic composition mechanism largely relies on how soundly the formal methods working on such semantic descriptions consume them.

This chapter described an agile approach to the behavioral selection of services that builds on top of existing semantic discovery approaches. Our approach works with the current characterization and technologies for a SOA; it does not suggest any modifications to any of the SOA components. Importantly, it is very easy to implement for existing services, many of which are freely available on-line.

The case for commercial Web services would require some appropriate business model, because they could not simply be tested as we have done herein without some form of compensation. We envision the appearance on the Web of third party brokers authorized to run tests on a provider's service. Similarly, if a service had side effects, such as performing transactions, e.g., buying and selling, a means for distinguishing between test executions and real executions would be needed.

Behavioral constraints are expressed in validation-oriented (as opposed to verification-oriented) unit tests. We have presented examples of the analysis that can be made with the execution values of the service parameters using range constraints. However, this analysis might have to be much more complex than just verifying conformance to a range. We plan to modify the model so that other type of constraints (e.g., exact value) can be expressed over service parameters or even properties of those parameters (e.g., size of a string parameter) Further, we envision intelligent agents reasoning about the service behavior based on the execution values of functional and non-functional parameters. This would even allow for discovery of contextual information that could later be used to provide added value services.

Service-Oriented Computing has recently been considered by the Grid community as a useful emerging paradigm to adopt. According to the Open Grid Services Architecture (OGSA) framework, the service abstraction may be used to specify access to computational resources, storage resources, and networks in a unified way. Hence, a computer service may be implemented on a single-processor or multiprocessor machine; however, these details might not be directly exposed in the service contract. The granularity of a service can vary and a service can be hosted on a single machine, or it may be distributed. Dynamic discovery and selection of services in this case will be essential for interoperability (Singh & Huhns, 2005). Our behavioral selection approach provides the means to have precise results without having access to the hidden implementation of the service.

KEY TERMS & DEFINITIONS

Services: Unassociated, loosely coupled, reusable units of functionality that have no calls to each other embedded in them, but provide descriptive metadata about their structure, functionality, and interface, so that can be used for meeting a specification of desired functionality.

Web services: Services accessible over standard Internet protocols independent of platforms and programming languages.

Service discovery: Searching over service repositories, usually by exploiting the services' metadata, to find the services satisfying a specification of desired functionality.

Service selection: The ranking mechanism used to choose a service among a list of discovered services with comparable functionalities.

Service composition: The development of customized services often by discovering, integrating, and executing existing services. Existing services are orchestrated into one or more new services that fit the desired functionality of a composite application.

Service behavior: The performance of the service (outputs values, response time, etc.) under different scenarios (i.e., assignment of values to the input parameters).

Functional parameters: The inputs required by the service as well as the service outputs.

Non-functional parameters: The properties that capture other aspects of a service aside from functionality; e.g., security, price, temporal availability, and quality.

Quality of service (QoS) parameters: Non-functional parameters related to the quality of the functionality provided by the service; e.g., performance, throughput, accuracy, reliability, availability, and trust.

Test-driven development: An approach for software development that involves repeatedly writing a unit test and implementing only the code necessary to pass the test.

Unit test: The smallest testable part of an application. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

REFERENCES

[1] Agarwal, S., & Studer, R. (2006). Automatic Matchmaking of Web Services. *Proceedings of the IEEE International Conference on Web Services* (pp. 45-54).

[2] Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M, Schmidt, M.-T., Sheth, S., & Verma K. (2005, November 7). Web Service Semantics - WSDL-S. Retrieved from http://www.w3.org/Submission/WSDL-S/

[3] Akkiraju, R., Srivastava, B., Ivan, A.-A., Goodwin, R., & Syeda-Mahmood, T. (2006). SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition. *Proceedings of the IEEE International Conference on Web Services* (pp. 37-44).

[4] Astels, D. (2003). Test Driven development: A Practical Guide. Prentice Hall Professional Technical Reference.

[5] Atkinson, c., Brenner, D., Falcone G., & Juhasz, M. (2008). Specifying High-Assurance Services. *IEEE Computer*, 41, 64-71.

[6] Bandara, A., Payne, T., De Roure, D., Gibbins, N., & Lewis, T. (2008). A Pragmatic Approach for the Semantic Description and Matching of Pervasive Resources. *Proceeding of the 3rd International Conference on Grid and Pervasive Computing* (pp. 434-446).

[7] Banerji, A., Bartolini, c., Beringer, D., Chopella. V., Govindarajan, K., Karp, A., . . . Williams, S. (2002, March 14). Web Services Conversation Language (WSCL) 1.0. Retrieved from http://www.w3.org/TR/wscl10/

[8] Barry, D. K. (2003). *Web Services and Service-Oriented Architectures (The Savvy Manager's Guides)*. San Francisco, CA: Morgan Kaufmann Publishers.

[9] Beck, K. (2002). *Test Driven Development by Example*, Addison-Wesley Professional, 1st edition, ISBN 978-0321146533.

[10]

[11] Boehm, B. (1984). Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1), 75-88.

[12] Braun, I., Strunk, A., Stoyanova, G., & Buder, B. (2008). ConQo - A Context- and QoS-Aware Service Discovery. *Proceedings of the IADIS International Conference WWW/Internet* (pp. 432-436).

[13] Brazier, F.M.T., Kephart, J.O., Parunak, H.V.D., & Huhns, M.N. (2009). Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda, *IEEE Internet Computing*, vol. 13, no. 3, (pp. 82-87).

[14] Cardellini, V., & Iannucci, S. (2010). Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications. *Proceedings of the 2010 IEEE International Conference on Web Services* (pp. 504-511). Miami, Florida, USA.

[15] Chakraborty, D., Perich, F., Avancha, S., & Joshi, A. (2001). Dreggie: Semantic service discovery for m-commerce applications. *Proceedings of the Workshop on Reliable and Secure Applications in Mobile Environment, Symposium on Reliable Distributed Systems*.

[16] Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001, March 15). Web Services Description Language (WSDL) 1.1. Retrieved from http://www.w3.org/TR/wsdl

[17] Clement, L., Hately, A., von Riegen, C., Rogers, T., Bellwood, T., Capell, S., . . . Wu, Z. (2004, October 19). UDDI Spec Technical Committee Draft. Retrieved from http://uddi.org/pubs/uddi_v3.htm

[18] Grigori, D., Corrales, J.C., & Bouzeghoub, M. (2006). Behavioral Matchmaking for Service Retrieval. *Proceedings of the IEEE International Conference on Web Services* (pp. 145-152).

[19] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., Lafon, Y. (2007, April 27). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Retrieved from http://www.w3.org/TR/soap12-part1/

[20] Irmert, F., Fischer, T, & Meyer-Wegener, K. (2008). Runtime adaptation in a service-oriented component model. *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing systems* (pp. 97-104). Leipzig, Germany.

[21] Lei, L., & Duan, Z. (2005). Transforming OWL-S Process Model into EDFA for Service Discovery. *Proceedings of the IEEE International Conference on Web Services* (pp. 137-144).

[22] Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Inc., 1st edition, ISBN 978-0135974445.

[23] Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., . . . Sycara K. (2004, November 22). OWL-S: Semantic Markup for Web Services. Retrieved from http://www.w3.org/Submission/OWL-S/

[24] Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. (2002). Semantic Matching of Web Services Capabilities. *Proceedings of the International Semantic Web Conference* (pp. 333-347).

[25] Rao, J., & Su, X. (2004). A Survey of Automated Web Service Composition Methods. *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*.

[26] Roman, D., Lausen, H., Keller, U., de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., . . . Michael Stollberg. (2006, October 21). D2v1.3. Web Service Modeling Ontology (WSMO). Retrieved from http://www.wsmo.org/TR/d2/v1.3/

[27] Ron, R. (2011). *XProgramming.com: an Agile Software Development Resource*. Retrieved March 1, 2011, from http://www.xprogramming.com

[28] Serhani, M. A., Dssouli, R., Hafid, A., & Sahraoui, H. (2005). A QoS Broker-Based Architecture for Efficient Web Services Selection. *Proceedings of the IEEE International Conference on Web Services* (pp. 113-120).

[29] Shen, Z, & Su, J. (2005). Web Service Discovery Based on Behavior Signatures. *Proceedings of the IEEE International Conference on Services Computing* (pp. 279- 286).

[30] Simon, L., Mallya, A., Bansal, A., Gupta, G., & Hite T. D. (2005). A Universal Service Description Language. *Proceedings of the IEEE International Conference on Web Services* (pp. 824-825).

[31] Singh, M. P., & Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Ltd, West Sussex, England.

[32] Sivashanmugam, K., Verma, K., Sheth, A., & Miller, J. (2003). Adding Semantics to Web Services Standards. *Proceedings of the IEEE International Conference on Web Services* (pp. 395-401).

[33] Skoutas, D., Simitsis, A., & Sellis, T. (2007). A Ranking Mechanism for Semantic Web Service Discovery. *Proceedings of the IEEE Congress on Services* (pp. 41-48).

[34] Srivastava, A., & Sorenson, P. G. (2010). Service Selection Based on Customer Rating of Quality of Service Attributes. *Proceedings of the 8th International Conferences on Web Services*.

[35] Vu, L.-H., Hauswirth, M., & Aberer, K. (2005). QoS-based Service Selection and Ranking with Trust and Reputation Management. Proceedings of the Cooperative Information System Conference.

[36] Wang, Y., & Stroulia, E. (2003). Flexible Interface Matching for Web-Service Discovery. *Proceedings of the 4th International Conference on Web Information Systems Engineering* (pp. 147- 156).

[37] Wang, Y. & Vassileva, J. (2007). Toward trust and reputation based Web service selection: A survey. *International Transactions on Systems Science and Applications*. 3, 2, (pp. 118—132).

[38] Wombacher, A., Fankhauser, P., & Mahleko, B. (2004). Matchmaking for Business Processes based on Choreographies. *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service* (pp. 359-368).

[39] Xiao, H., Zou, Y., Ng, J., & Nigul, L. (2010). An Approach for Context-aware Service Discovery and Recommendation. *Proceedings the 8th International Conference on Web Services*.

[40] Ye, L., & Zhang, B. (2006). Web Service Discovery Based on Functional Semantics. *Proceedings of the Second International Conference on Semantics, Knowledge, and Grid* (pp. 57-57).

[41] Zavala Gutierrez, R.L., Mendoza, B., & Huhns M. N. (2007). Behavioral Queries for Service Selection: An Agile Approach to SOC. *Proceedings IEEE International Conference on Web Services*, (pp. 1152-1153).