DESIGN AND IMPLEMENTATION OF A CONTEXT-BASED SYSTEM FOR COMPOSITION OF WEB SERVICES

Wassam Zahreddine, Qusay H. Mahmoud
Department of Computing & Information Science
University of Guelph, Guelph, ON, N1G 2W1, Canada

Keywords: Context-based discovery, dynamic composition, personalization, UDDI, CC/PP

Abstract: Web Services are loosely coupled modular HTTP-based applications that represent a new approach for application integration. The reusability of web services is making them an attractive solution for businesses and consumers alike because of their simplicity and accessibility. Today's web services are designed to be modular and loosely-coupled to perform a specific set of operations. This modularity of web services, however, has left an open problem in composition – a scenario that involves an amalgamation of two or more web services to fulfil a request that no one web service is able to provide. This paper presents the design and implementation of a system that enables users of any device to dynamically discover context-based web services that will be automatically composed to satisfy a user's request. Existing web services can be easily adapted and new web services can be easily deployed. The system uses a custom UDDI-like registry that we have designed and implemented to support dynamic discovery and context-based composition.

1 INTRODUCTION

Web services are becoming an attractive solution for businesses and consumers alike because of their simplicity and reusability. Today's web services are designed to be modular and loosely coupled to perform a specific set of operations such as retrieving a stock quote. However, what if a client requires a service that no one web service can satisfy? The modularity of web services has left an open problem in composition. Web service composition involves an amalgamation of two or more web services to fulfil a request that no one web service is able to provide. A composite web service will bring forth endless possibilities and a new wave of online applications. Furthermore, the availability of web services is somewhat limited depending on the device being used. Recent efforts, such as (J2ME Web Services Specification, 2004), address the problem of accessing single web services from mobile devices but do not inherently support composition. Also there is a wide array of devices available, each with their own specific hardware and software capabilities making application deployment difficult. This paper presents the design and implementation of a novel system that will address these shortfalls in web service composition. The rest of this paper is organized as follows. Section 2 presents an overview of Web services and Web service composition. Section 3 presents the proposed system and its main concepts. The high-level architecture of the system and its components are discussed in Section 4. Section 5 presents the ServiceSearcher, which is a core component of the system that serves an alternative to using a UDDI registry. The implementation of the system is discussed in Section 6. Related work is discussed in Section 7. Finally, conclusions and future work are discussed in Section 8.

2 WEB SERVICES

The concept of accessing services over the WWW or intranet is relatively new. The earlier stages began with frameworks by CORBA (Common Object Request Broker Architecture) and Java allowing heterogeneous components to communicate and interact with each other. As businesses began to see the cost-effectiveness of reusing heterogeneous components and languages, the popularity of web services grew. Web Services are loosely coupled modular web-based applications representing new ways to share services and information between other applications (Curbera, F., et al., 2001). The reusability of web services means businesses can save money by reducing development costs. It will also allow them to make these services available for sale to create new opportunities with other businesses. Once a web service is created, it is advertised in a registry called UDDI (Universal Description, Discovery and Integration), where it can be searched upon. UDDI provides the location to the service provider's WSDL (Web Services Description Language) file, which describes the methods that can be invoked and the parameters that are required. Messages are exchanged through SOAP (Simple Object Access Protocol). SOAP works by exchanging information using GET/POST over HTTP. This allows the data to be exchanged between firewalls regardless of where the client is within a network.

Web Service composition is the combination of two or more web services to perform what no single service is able to provide. The modularity and loose coupling design of web services allows them to perform a specific task. However, this may be seen as a disadvantage if a client would hope to find a web service that could perform all its needs. Therefore, custom solutions are needed to discover and connect multiple web services into one application. Several composition techniques have been proposed, some of these are presented in Section 7 (Related Work).

3 THE PROPOSED SYSTEM

The proposed system allows end users to transparently access web services based on context information over their personal devices whether desktops or portables. Users access web services through the use of customized Java Servlets or standalone applications that hide the complexities of employing web services. The main hallmarks of this system are: (1) Static and dynamic discovery of web services; (2) Context-awareness by keeping track of user preferences and results through personalized user profiles; (3) Compatible with virtually any device that has a web browser and access to the Internet; and (4) Capable of asynchronous communication with end user's device to support the unreliability of wireless networks.

To support static and dynamic discovery the system utilizes OWL-S (OWL-based Web Service Ontology), which was originally known as DAML-S, to semantically describe web services and their methods. Our system gives greater control on how web services are dynamically discovered by allowing the application developer to specify how matches are made, which goes beyond the present techniques of semantically matching inputs and outputs along with classification taxonomies.

3.1 Deploying Context-based Services

Our system is designed with two objectives in mind: allowing users on virtually any device to discover context-based web services, and to make that experience simple and easy. Creating one service that will work on every portable device is extremely difficult because of the many different hardware and software combinations. Portable devices have all sorts of screen sizes, memory configurations, processor speeds, operating systems and creating one service that can cope with all may seem near impossible for some developers. Regardless of these new challenges, steps need to be taken to make applications adaptive to these dynamic environments. Fortunately, there have been significant advances in device compatibility techniques. The key to compatibility is finding a universal language to best describe a device and one such language is CCPP (Composite Capabilities/Preferences Profile) and UAProf (User Agent Profile). CCPP can be used to describe not only mobile phones but other devices such as PDAs or Smartphones. Industry leaders such as RIM, Nokia, Motorola and others have already begun to embrace this technology and have published profiles to support their devices. For instance, a UAProf profile can describe many attributes of a device such as: hardware and software characteristics, supported network types and browser information. CC/PP will help optimize the content for a device, reduce the testing time, and even help create future-proof applications. It is important to note that UAProf is not an alternative standard to CC/PP, instead it is a specific profile for WAP devices.

Along with the CCPP of the device, user information is also used as context for services. Data such as user's name and address is stored and used as context information, thereby making the system context-aware. Furthermore, this information can be used to help the user by having the application automatically load this data as service inputs. For instance, a field requiring a city name to be entered will automatically be filled in using the address stored for that user. Hence, context-awareness will save time for the user, especially when using devices with small keypads (e.g. cellular devices) that take longer to work with than devices with full keyboards.

The proposed system has the ability to handle context in two areas. Firstly, the system has the ability to conform the results received from web services readable on the users' device. Also it takes into consideration the capabilities of the device, such as displaying pictures and bandwidth to know whether the image is too large for the device to handle and simply not send that object. Secondly context-awareness can be handled at the web service level by giving the service the CC/PP profile of the user's device (explained in Section 4).

3.2 Discovering Services

The proposed system supports static and dynamic service discovery. A summary of the costs and benefits of each type of discovery can be found in Table 1. Knowing what service to use a priori reduces response time, since no search needs to be made. One of the drawbacks of static service discovery is whether another web service could be floating out on the Internet that could perform a service better (i.e. faster, more reliable). Or, what if that web service is no longer available and another service must be found to replace the current one.

Table 1: Pros and cons of discovery methods

Discovery Method	Pros	Cons
Static	• Quickly access the intended service thereby improving response time	• Dependent on the availability of that service • Ignores the possibility of discovering better web services
Dynamic	• New web services can be found that may offer better quality of service, such as better prices, or reliability	• Even with semantics there is no guarantee that an intended web service is found

Finding a web service normally takes a human being, a UDDI registry, and patience. Simply because a user would have to search a UDDI to find a web service that would best suit his/her needs. That involves examining "white" and "yellow pages" that represent a web service in a UDDI registry.

Discovering web services automatically has a number of issues since you are relying on a machine to find a service you need. This will involve a machine that can understand what it is looking for and what a web service will do. Hence, simply searching by keywords will not cut it. For example, searching for a service that could be described in many different ways all meaning the same thing. There are many ways a service could describe an automobile such as vehicle or car. Unless the machine knew all the different spellings for an automobile, the results could be limited to what the machine is programmed to find. More interestingly, what if the machine finds a keyword that matches but that word is a homograph? For instance, a machine looking for information on bass (freshwater bass fish) and alternatively gets information on a musical instrument. Problems such as this have fuelled research on developing a Semantic Web (Berners-Lee, T., et al., 2001); so that machines could understand what they are reading without confusion. Regardless, there is no 100% guarantee that taxonomies and semantics alone can help find an intended web service. Our system attempts to find web services that best suit the user's criteria.

4 SYSTEM ARCHITECTURE

The system is comprised of many different components to support context-based discovery, composition and delivery of services. Figure 1 depicts the architecture of this system.

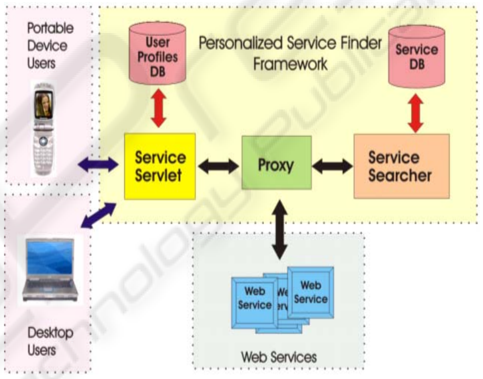


Figure 1: High-level system architecture

To begin with, users are comprised of either mobile or desktop users and will interface through a web browser, or a standalone application. This paper will focus on the web browser based interface, where a user is presented with a Java Servlet called a ServiceServlet. When services are invoked, a request is passed to the Proxy. The request is then forwarded to a ServiceSearcher to find potential web services for the Proxy to connect to. The proxy evaluates the list of potentials, executes those services and returns the results to the ServiceServlet to be displayed for the user.

An overview of the communication and interactions between components are depicted in Figure 2. To begin with, an end-user initiates a connection with the ServiceServlet and selects the services it wishes to invoke. A request is then sent to the Proxy where it is forwarded to a ServiceSearcher to find potential web services for the Proxy to invoke. Based on the given criteria of the request, a list of potential web services are returned to the Proxy. At this point the Proxy evaluates the list of potentials, executes those services and returns the results to the ServiceServlet to be displayed for the user. The following sections explain in greater detail the roles of each component in the system.

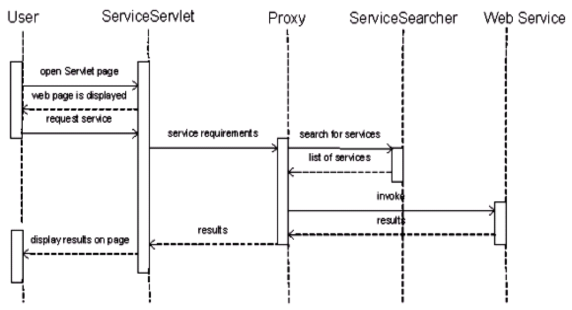


Figure 2: The process of executing a web service

4.1 The ServiceServlet

The system can be accessed by either a standalone client or a Java Servlet (ServiceServlet) through a Web browser, this paper will focus on Servlet use.

The ServiceServlet is designed so that a developer will have the option of how to set-up their services. Also, any number (or groups) of services could be supported by a ServiceServlet which implies that the organization and look of the ServiceServlet are entirely up to the designer. The only requirement is the role it plays when performing its service. It must connect to a Proxy to have services found and executed, then retrieve those results from the Proxy and display it, ideally, according to the device capabilities (by using the CC/PP profile if needed). Furthermore, unless the ServiceServlet is for anonymous public use, it will require access to a UserProfilesDB which will store data about the user such as: username and password, client name and address, client telephone and email address, CC/PP profile, and cached results from last access.

4.2 The Proxy

The proxy acts as an intermediary between the ServiceServlet and the ServiceSearcher. When the Proxy receives a request from the ServiceServlet, it contacts the ServiceSearcher to get a list of web services to execute and then returns the results back to the ServiceServlet. Even though services could potentially be executed from the ServiceServlet, we decided to separate the Proxy from the ServiceServlet. This approach is similar to the JSP Model 2 Architecture which separates the processing logic from the presentation layer. In doing so, improves the load balancing of the system by keeping the ServiceServlet and Proxy on separate servers. Not to mention the improvements in flexibility, and maintainability. Also by allowing the ServiceServlet to connect to auxiliary Proxies (if necessary) helps the system scale well (Figure 3).

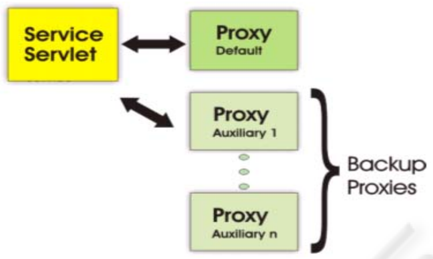


Figure 3: Backup Proxies can be accessed if needed

In addition, the Proxy supports two important features to improve the reliability and speed of its operation. Firstly, the Proxy is able to access more than one ServiceSearcher to retrieve services, hence improving the reliability of finding a matching service. In particular, if the default ServiceSearcher is unable to find a matching service or if the ServiceSearcher is offline, the Proxy has the ability to access auxiliary ServiceSearchers (Figure 4) located on other servers.

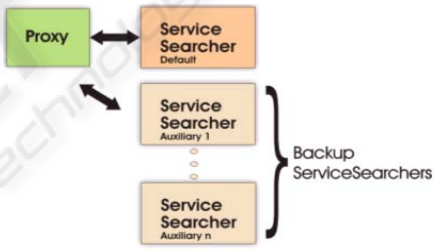


Figure 4: Using ServiceSearchers to satisfy a request

Secondly, the Proxy also supports the caching of service requests, thereby entirely bypassing the step of contacting a ServiceSearcher. Caching involves storing the request, the services used, the input values used and the results stored in memory. Therefore, if a new request matches a previous one including the input values, the results can be quickly returned to the ServiceServlet. If the new request matches a previous one but the input values are different, the service is executed again using the new input values. However, the ServiceSearcher still needs not be contacted thereby improving the speed of *re-discovery*.

As previously mentioned, the Proxy has three roles, since it communicates with the ServiceServlet, ServiceSearcher, and web services. The first role deals with accepting requests from the ServiceServlet. The second role concerns satisfying the request by performing a search on the ServiceSearcher. Lastly, once the ServiceSearcher returns a list of services, the Proxy orchestrates and executes web services.

4.3 Autonomous Execution

The Proxy in the system runs and controls web services using the OWL-S control constructs. OWL-S defines web services as either simple or complex. Simple services do not require other web services to perform their task. A Complex service is a web service that is composed of more than one simple service. This allows the Proxy to connect to individual web services as well as composite web services. When the Proxy receives a list of services to execute, it first examines the OWL-S document to map the input values with the input parameters of the services. As long as the Proxy has all the required inputs for the service, it will be executed and the output value is inserted into the list of results to be returned back to the ServiceServlet. The proxy creates an execution plan based on the inputs needed to produce a required output. This involves checking whether any output(s) from one service (let us call it Service-A) are required as inputs for another service (Service-B). Then Service-A will have to be executed before Service-B in the list. If both services require inputs and outputs from each other, then there is a deadlock and these services will be removed from the execution plan. It is at this point that the Proxy will replace Service-A and Service-B with alternate compatible services from the list of services returned from the ServiceSearcher. If no alternates are found on the current list, the Proxy can either try requesting new services from an auxiliary ServiceSearcher or report failure. Furthermore, an important benefit of OWL-S is that it also supports automatic execution monitoring to perform alternate actions; such as when a web service is offline or fails to function properly. The execution plan will have an opportunity to try an alternate service.

4.4 Incentives for Industry Support

In order to run an infrastructure such, as the one proposed by our system, on mass scale would not be cheap. Incentives are needed for Service Providers that host this system in order to cover the necessary costs brought on by it. The following are examples of possible techniques to be used:

- Advertisements displayed on the Servlet page. This will work well for desktop users because of the amount of screen space available. Although this technique would be difficult for portable users with limited screen sizes and colors.
- The Service Provider only offers web services that they personally endorse. In this scenario all profits made from web services would go solely to the Service Provider.
- End-users pay an enrolment fee and pay per service used or have monthly subscriptions for predefined usage.
- A commission based system where Third-party web services pay a fee to advertise their web service within the system's service registry.
- Third-party web services advertise for free but pay a fee (commission) every time their web service is used.
- A combination of last two techniques.

These are but a few of the potential revenue opportunities available to would-be Service Providers. Hence, in order for companies to support this system legitimate incentives are necessary. Internet and Cellular Service Providers as well as Web Service providers will be inclined to work together and their bond will come from the profits of offering services.

5 SERVICE REPOSITORY

The purpose of the ServiceSearcher is to locate potential web services that satisfy a given request from a Proxy. Our system does not take advantage of any existing UDDI repositories. Although this may seem like a disadvantage to our proposed system, after careful analysis we chose to create a new service repository. The main reason being, in order for the system to support automatic and dynamic discovery, semantic metadata must be supported by the UDDI repository. Presently UDDI has no such support. UDDI is designed primarily for humans to navigate and use. UDDI does not inherently support semantics and adding semantics will require another layer above UDDI to understand semantic requests. To support semantic information in UDDI will require an approach similar to (Paolucci, M., et al., 2002). Accessing two databases to do something that one database can is counterproductive and increases service discovery time. Moreover, it is important to note that UDDI is still far from becoming a standard and many changes will take place if and when it does become standardized by the OASIS Consortium.

The ServiceSearcher is similar to a UDDI repository, but instead should be considered a semantic UDDI registry used strictly for machine read operations. Therefore, no White or Yellow pages are implemented since they are only read by a person and not a machine. This information is still available from within the OWL-S description that is stored along with other service details (e.g. service ID, service name).

Discovery can be performed statically by searching by a unique service identifier (ServiceID). Or, the search can be performed dynamically. A dynamic search has three parts within the request. The first part is classification taxonomies such as NAICS (North American Industry Classification System), and/or UNSPSC (Universal Standard Products and Services Classification). The second and third parts include the inputs and outputs of the request. Inputs and outputs of all OWL-S services have labels (eg. car, house, film) that are part of a semantic ontology and allow semantic matching of the actual parameters used in the web service. With this information included in the request the ServiceSearcher can match classification taxonomies, inputs, and outputs to find potential matches. A list of matches are compiled and then sent back to the Proxy.

The ServiceSearcher is a server application that has access to the ServiceDB which is the database that does the actual searching. Matching involves a list of queries that are generated and then executed on the ServiceDB. This database stores services by their service ID, service name, list of inputs, list of outputs and the URI to the OWL-S document. Based on the request a ServiceSearcher receives from a Proxy, it will compile a list of potential web services represented in OWL-S documents and then send this list back to the Proxy.

The matching algorithm involves a list of queries that are generated and then executed on the ServiceDB. The ServiceDB is a database that manages services by their service ID, service name, list of inputs, list of outputs and the URI to the OWL-S document). All queries include a search based on the classification taxonomy as the first requirement to help insure that the correct service or product is located. Next, the first query generated checks to see if all inputs and outputs match a service in the ServiceDB. However, there is a chance that not all the requirements of the request can be answered by one service alone (especially if numerous inputs and outputs are in the request). Hence the search must be split. For instance two services that may be able to satisfy a request that one single service could not. Although using classification taxonomies and inputs and outputs for semantic matching have been proposed by others, what makes the ServiceSearcher unique is that it allows the ability for the searches to be tweaked in such a way to improve the accuracy of the results. The technique used involves adding dependencies to the outputs requested. Table 2 lists the potential dependencies and their corresponding syntax. In respect to the ServiceSearcher, a dependency signifies that when a service is being searched in the ServiceDB that has a particular output, that service must include all the inputs that are in the output's dependency list. For example, if the request's output variable is *review <i# movie_title>*, only services that have an output *review* and an input *movie_title* will satisfy this request.

Table 2: Output dependency codes used in a request

Dependency	Syntax	Example
An output variable that depends on one or more inputs	Output <I# Input$_1$,…, Input$_n$>	review <i#movie_title> - a review output variable that depends on a *movie_title* input variable
An output that depends on all inputs.	Output <i#ALL_INPUTS>	review <i#ALL_INPUTS> - a review output variable that depends on *all inputs* in the request
An output that does not have any dependency	Output	review - a review output variable that does not require any dependencies

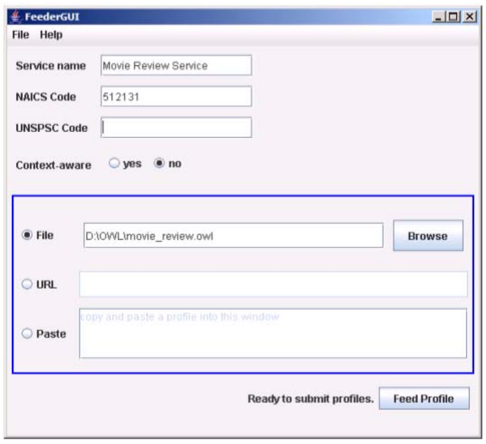
These dependencies give the ServiceServlet developer added control to what the ServiceSearcher will select as a potential match. Since most outputs require specific inputs to provide their intended results.

6 IMPLEMENTATION

Various technologies were used to implement the described system. To begin with, the system is built upon Java technology and XML. OWL-S is the technology we have used to semantically represent WSDL files, and in essence the web services. In order to speed up development time we have used the OWL-S API. This API provide the methods needed to read and parse OWL-S and WSDL files, both of which are written in XML. The OWL-S API also allowed us to execute web services.

The WSDL2OWL-S Converter was used in order to convert the WSDL file of a web services into OWL-S. Originally the authors of this tool created WSDL2DAML-S that was ported over to the OWL-S specification. WSDL2OWL-S application will accept a WSDL file and allow the user to fill in information needed for the ServiceProfile. The ServiceModel of the OWL-S file can be edited to give semantic labels to the input and output variables of the web service. Now the OWL-S file can be submitted to the ServiceDB database. To ease this process, we have created a graphical application we have named OWL Service Feeder. A screen shot of this application is shown Figure 5. To submit an OWL-S file to the database, the user first connects to the database using the File menu. Then the user enters the Service Name, NAICS, UNSPSC, Context-aware settings. And finally, the user selects the location of the OWL-S source file.

A ServiceServlet is a Java Servlet which runs on a Java application server. We chose IBM WebSphere 6 as the application server to handle the Java Servlets and the web services we created.



Service name: Movie Review Service
NAICS Code: 512131
UNSPSC Code:
Context-aware: yes [✓ no]

[✓ File] D:\OWL\movie_review.owl
URL
Paste: copy and paste a profile into this window

Ready to submit profiles.

Figure 5: The interface to the OWL Service Feeder

We chose MySQL as the database application to run the UserProfilesDB and the ServiceDB. The use of MySQL was an easy choice since it is free, easy to use, and performs well for the needs of this system.

To support the adaptive web interface generated by the ServiceServlet we employed CC/PP technology. To speed up development we used JSR 188 CC/PP Processing API. The API allowed us to parse CC/PP profiles to extract the UAProf descriptions of devices.

In addition, the system also has the functionality to locate web services that handle context information on their own. Who better to format the web service output than the web service itself? The choice is up to the ServiceServlet creator whether to allow the web service to handle contextual information or not. This is important since there may be situations where only the creators of the web service would know how to optimally fit the results on a portable device, whereas the ServiceServlet would take a best effort approach.

In order for the web service to support customized output formatting for a specific device, it must be adaptive. This requires information about the end-user's device. This is done by passing the URI of the users CC/PP profile to the web service from within SOAP header or passed in as one of the WSDL PortType parameters. The profile contains various details of what the device is capable of, such as screen, and support for a WAP browser. With this information, the web service or ServiceServlet will know what types of colours and images will work with the device's screen, and the size of pages that can be received on the device.

7 RELATED WORK

The automation of web service discovery is a relatively new idea, however, quite a few approaches have been put forth in academia to address this challenge. In particular, work done by (Sheng, Q.Z., et al., 2004) describes a framework for personalized service composition involving mobile users. They refer to their system as PCAP (Personalized Composition and Adaptive Provisioning of Web Services). A user has access to "process templates" which accomplish specific tasks. The user has the ability to modify these templates which sounds like a promising idea to allow users to adjust anything they wanted about a service. However, in order to do this, a user is expected to understand and alter state chart diagrams using the "template builder" the authors provide. Hoping a user will be able to modify their process templates seems like a tall order for a novice computer user. The templates themselves only provide for semiautomatic composition since the templates are just a compilation of predetermined services. During execution, template inputs are gathered from user input, the user profile and outputs from previous states in the template. As for service discovery, a "Template/Service Discovery Engine" is used to find advertised templates in a UDDI. However, the discovery is of a static nature since the system is incapable of dynamic discovery.

The work in (Sheshagiri, M., et al., 2004) also involves composite web services over portable devices. The framework they describe is used to support the "myCampus" context-aware environment which is designed to help students with the everyday tasks of campus life. There are a few similarities between myCampus and our system. Firstly, alike the PCAP system context, data is stored in some type of user profile. Also similar is the backward-chaining execution plan and the use of contingency web service in case of failures. Web services are unlikely to have 100% uptime. That is why our system also accounts for this by including alternate web services in the execution plan to be used as backup in the event of web service failure. Also, as with our system, OWL-S is utilized to support dynamic discovery of web services.

However, myCampus does not allow modifications to the searching algorithm. As mentioned earlier, a supported feature of our system includes using output dependencies to fine tune the searching algorithm and improve results. Furthermore, unlike our system, myCampus does not support context to sustain the hardware and software capabilities of the device.

Lastly, work done by (Paolucci, M., et al., 2002) have created a DAML-S/UDDI Matchmaker to dynamically discover web services. Web services are discovered based on the advertisement that provides a semantic description of a web service, similar to our ServiceDB. The advertisements describe the inputs and outputs of the service and a reference ID. The UDDI registry contains DAML-S data regarding the web service from within a tModel. When a request's inputs and outputs are matched with an advertisement in the Advertisement Database, the resulting reference ID is translated and point to a UDDI entry representing the service. In comparison to our ServiceDB, this approach is counterproductive since two databases must be contacted in order to retrieve the necessary data to locate a service. The ServiceDB registry provides all the necessary information about a service; our approach improves the speed and efficiency for machines to find services. Moreover, their matching engine will only declare a match if all inputs and outputs of a service match those of the request. Hence, their system is incapable of splitting the search to find more than one service that will satisfy the request, thus producing false-positives.

8 CONCLUSION AND FUTURE WORK

In this paper we have described a system that automatically and dynamically composes web services based on context information and allows them to be displayed on virtually any portable device. New techniques have been proposed to make this possible. Firstly, the use of CC/PP with ServiceServlet to aid in the compatibility of devices to view and use web services. Also CC/PP can be passed on to context-aware web services to further improve compatibility. Secondly, the use of output dependencies to customize the discovery process of services to help ensure the quality of the results. And lastly, the design of the ServiceSearcher service registry that is more efficient than current techniques that involve semantics with UDDI.

Future developments will involve security issues relating to users and the system as a whole. For instance, securing the connection between the user and the ServiceServlet, between ServiceServlets and Proxies, between Proxies and ServiceSearchers and lastly between Web Services and Proxy. Furthermore, ways to measure the quality of a web service, by keeping track how well it has performed in the past using a type of rating system. Another crucial feature would involve having some type of legal contract between the Proxy and the web service to instil the notion of liabilities and make the two parties responsible for their actions.

ACKNOWLEDGMENT

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant No. 045635.

REFERENCES

Berners-Lee T., Hendler J., Lassila, O. The Semantic Web, Scientific American, May 2001.

CC/PP (Composite Capabilities/Preferences Profile):
http://www.w3.org/Mobile/CCPP.

Curbera F., Nagy W.A., and Weerawarana S., Web Services: Why and How, OOPSLA 2001 Workshop on Object-Oriented Web Services, October 2001.

J2ME Web Services Specification (JSR 172):
http://jcp.org/en/jsr/detail?id=172.

North American Industry Classification System (NAICS) www.census.gov/epcd/www/naics.html.

Organization for the Advancement of Structured Information Standards http://www.oasis-open.org.

OWL-S: Semantic Markup for Web Services:
http://www.daml.org/services/owl-s/1.0/owl-s.html.

Paolucci M., Kawamura T., Payne T., Sycara K., Importing the Semantic Web in UDDI. E-Services and the Semantic Web Workshop. 2002.

Sheng Q.Z.,and Benatallah B., Maamar Z., Dumas M., Ngu A., Enabling Personalized Composition and Adaptive Provisioning of Web Services. 2004.

Sheshagiri M., Sadeh N., Gandon F., Proc. of MobiSys2004 Workshop on Context Awareness, Boston, June 2004.

Sycara, K.; Paolucci, M., Ankolekar, A.; Srinivasan, N., Automated Discovery, Interaction and Composition of Semantic Web Services, Journal of Web Semantics, Volume 1, Issue 1, December 2003.

Universal Standard Products and Services Classification (UNSPSC): http://www.unspsc.org.

WSDL2OWL-S:
http://www.daml.ri.cmu.edu/wsdl2owls.