Automatic Web Services Generation

Ernest Cho
Computing & Software Systems
Institute of Technology
Univ. of Washington, Tacoma
xxx@u.washington.edu

Sam Chung
Computing & Software Systems
Institute of Technology
Univ. of Washington, Tacoma
chungsa@u.washington.edu

Daniel Zimmerman
Computing & Software Systems
Institute of Technology
Univ. of Washington, Tacoma
dmzimm@u.washington.edu

Abstract

This paper discusses the design and implementation of a Service Generator Toolkit (SGT) that allows web services researchers to easily create large numbers of web services. When developing a web services tool, such as a service broker, it is necessary to obtain a large collection of web services for testing and benchmarking purposes. Since it is difficult to manually create or collect a large number of web services, we chose to implement a system that can automatically generate web services from a service graph model, the SGT. The SGT automatically generates web services by creating an abstract graph model of the services and then converting that model into implementation files. By converting the services graph model into deployable and executable web services, the SGT provides support for a greater range of tests than previous efforts that use service models to facilitate testing.

1. Introduction

This paper discusses the design and implementation of a Service Generator Toolkit (SGT). This toolkit automatically generates a large collection of web services for use in the testing and development of web services tools, in particular service brokers. A service broker is a system that allows service consumers to discover web services published by service producers.

With traditional web services, a service broker is little more than a basic directory service and thus requires significant human interaction to work. With semantic web services, the service broker takes on more responsibility and automates service discovery and service composition [2]. This is accomplished by using ontologies, sets of concepts within a domain and the relationships between them, to provide a basis for common understanding between different web services. These ontologies allow service brokers to automatically perform tasks that would otherwise require extensive human interaction [4, 2, 6].

In order to experimentally explore and validate the many designs and algorithms used to implement service brokers, it is necessary to have an extensive collection of web services. There are a few ways for researchers to obtain large number of services either by creating services using existing tools or generating services from a services model.

The first option is to collect existing web services from real world service repositories. Unfortunately this approach is plagued by availability and compatibility problems. Fan et al found that half of all entries in public service repositories contained significant syntactical errors and many were simply offline [9]. Also, there is the issue of finding relationships between services that use different terminology. Since semantic web services and supporting ontologies are not widely used, the web services are manually connected.

Another option is to use existing tools, such as Apache Tomcat/Axis or Microsoft .NET, to create the necessary test services. This approach is often used to create proof of concept demonstrations. Unfortunately, for more than a handful of services it becomes an impractical and laborious process.

The final option is to automatically generating the necessary web services based upon an abstract model of the services. This approach expands upon the ideas presented by Constantinescu et al and Oh et al. Constantinescu et al discusses service modeling and model generation [3]. Oh and et al. goes a bit further and uses service modeling to automatically produce service descriptions [12]. The SGT goes a step further and exports web services implementation files. This allows the generated web services to be deployed onto

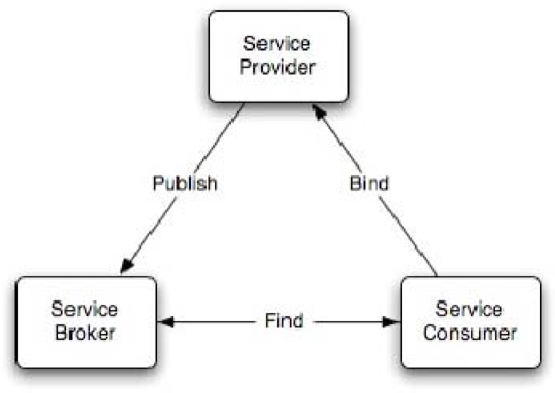

Figure 1. Service Oriented Architecture (SOA) diagram.

servers and executed, thus allowing a greater range of potential tests and experimentation.

2. Background

2.1. Software as a Service (SAAS)

A Service is an implementation of some high-level application functionality that has a well-defined, abstract interface. Most often these are Web-based software applications intended for use by customers over the Internet. This often refers to either Representational State Transfer (REST) based services or Web Services. Generally, REST systems are targeted towards end users, while Web Services are more business-oriented. This project focuses on Web Services [14].

2.2. Service-Oriented Architecture (SOA)

SOA is a design strategy where applications are dynamically built from loosely coupled Services using a Publish-Find-Bind model, which is shown in Figure 1. First, service providers publish descriptions of their service to a service broker. Service consumers then find a suitable service by querying the service broker. Once a service is found and selected, the consumer binds to that service for use. Services can be composed of other services. Note that the find step can include any matchmaking and composition necessary to meet the Service requester's needs [10, 14].

2.3. Web Services

Web Services are a middleware technology used to allow machine-machine interaction over a network. It is one technology used to implement SOA. The basic technologies include WSDL, SOAP, UDDI and BPEL. WSDL is a XML format used to describe the input and output interface of a service. SOAP is a XML format used to encode the messages between services. UDDI defines a metadata aggregation service and is used to build web service registries. BPEL is a composition language used to describe business processes and the use of services to implement that process. As a relatively new technology, Web Services are still in the process of being refined and standardized [13].

2.4. Semantic Web

The World Wide Web (WWW) was originally intended to facilitate the interchange of web documents between humans. Consequently, computers have a particularly difficult time extracting and making use of the data within web documents. The Semantic Web uses ontologies and markup languages to attach meaning to the data within web documents. This makes it easier for computers to work with web documents without human assistance. Currently, Resource Description Framework (RDF), Web Ontology Language (OWL), and Web Service Modeling Language (WSML) are the markup languages most often used in Semantic Web projects [6, 15].

2.5. Semantic Web Services (SWS)

Semantic Web Services combines the ideas behind the Semantic Web with Web Services in order to deliver greater automation of Web Services related tasks, such as service discovery and service composition. This allows the development of Service Brokers that can perform tasks with significantly less human interaction. Several frameworks have been proposed to facilitate the use of Semantic Web technology with Web Services including OWL-S, METEOR-S and WSDL-S [4].

2.6. Service Composition

Any Service can be used to build more complex Services through composition. For example, given simple Services that provide a square function, an addition function and a square root function it is possible to create a Composite Service that solves for the hypotenuse of a triangle when given the two shorter sides of the triangle. This is illustrated in Figure 2.

For traditional Web Services, building a composition is a manual process most often done using the Business Process Execution Language (BPEL).

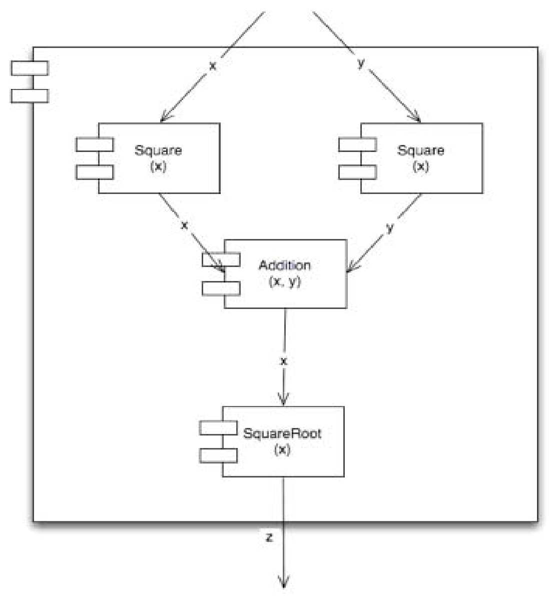

Figure 2. Composite service which solves for the hypotenuse of a triangle, z, given the two shorter sides of the triangle, x and y

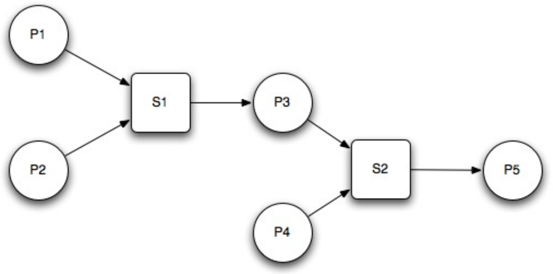

Figure 3. An example illustrating that reachability is not equivalent with composability. Service S2 is reachable from Service S1, however without Parameter P4, S1 cannot be composed with S2. In this case, S1 is a partial match for S2.

With Semantic Web Services the composition process can be largely automated. It should be noted that automatic composition has been shown to be NP-complete if partial matches of input and output parameters are allowed [3].

When looking at graph representations of Services it is easy to forget that reachability does not guarantee composibility. This is illustrated in figure 3. Service S2 is reachable from Service S1, however without Parameter P4, S1 cannot be composed with S2. In this case, S1 is a partial match for S2.

3. Related Works

The SGT allows researchers to create large numbers of web services. It does so by building an abstract model of the web services, then converting that model into implementation files. This approach is similar to a couple previous research efforts that use a service model to facilitate testing and experimentation with services: Constantinescu et al.'s large-scale test bed of services and Oh et al.'s tool for benchmarking planning composition algorithms.

Constantinescu and et al developed a large-scale test bed of services for the purposes of testing type compatible service composition [3]. They model a service as transformer that transforms a set of input parameters from one domain into a set of output parameters from another domain. The input set and output set of any service should come from different domains. Using this model they randomly generate a large number of services. This collection of test services is entirely abstract and is not suitable for any tests that require either a concrete interface description or an implementation.

Oh and et al developed a tool for benchmarking planning composition algorithms called WSBen [12]. They model services as a directed, bipartite graph consisting of service nodes, operation nodes and parameter nodes. A service node consists of one or more operation nodes. The operation nodes form a bipartite graph with the parameter nodes. Like Constantinescu et al., they also separate the parameters into domains and require that the input parameters and output parameters come from different domains. Unlike Constantinescu et al., they use their generated model to create web service interface descriptions in the WSDL format. They also provide means to visualize the graph model. Since WSBen provides interface information without an implementation, it is unsuitable for testing scenarios that involve service execution.

4. Services Generator Toolkit

The overall goal was to create a Service Generator Toolkit capable of automatically generating a large number of web services suitable for testing web services tools, such as Service Brokers. The generated web services should be syntactically correct, deployable and executable.

To accomplish this goal, the Service Generator Toolkit, which is shown in Figure 4, provides main components: a graph model generator called Random Graph generator, a representation of the graph model

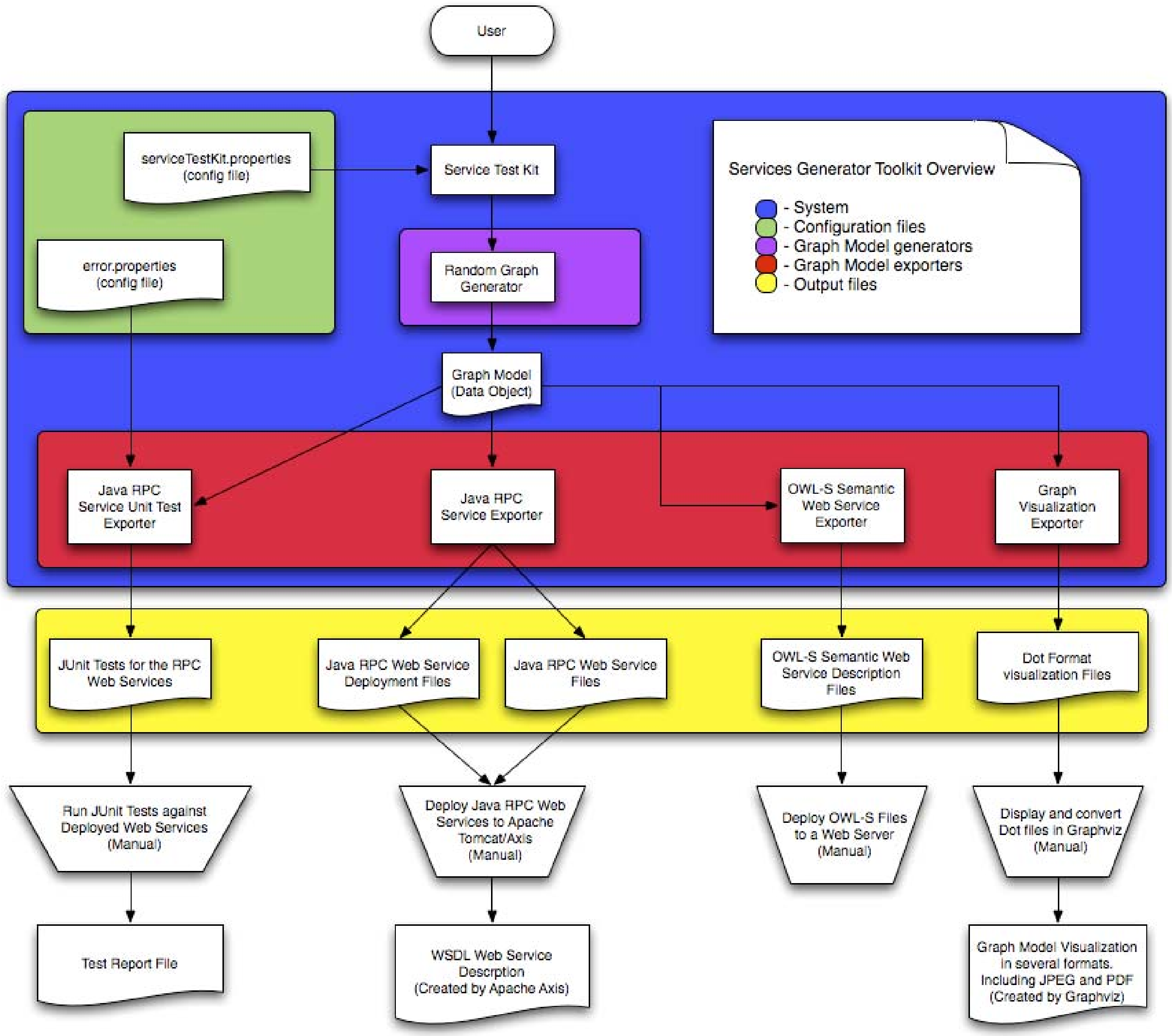
User
serviceTestKit.properties (config file)
Service Test Kit
error.properties (config file)
Random Graph Generator
Services Generator Toolkit Overview
🔵 - System
🟢 - Configuration files
🟣 - Graph Model generators
🔴 - Graph Model exporters
🟡 - Output files
Graph Model (Data Object)
Java RPC Service Unit Test Exporter
Java RPC Service Exporter
OWL-S Semantic Web Service Exporter
Graph Visualization Exporter
JUnit Tests for the RPC Web Services
Java RPC Web Service Deployment Files
Java RPC Web Service Files
OWL-S Semantic Web Service Description Files
Dot Format visualization Files
Run JUnit Tests against Deployed Web Services (Manual)
Deploy Java RPC Web Services to Apache Tomcat/Axis (Manual)
Deploy OWL-S Files to a Web Server (Manual)
Display and convert Dot files in Graphviz (Manual)
Test Report File
WSDL Web Service Descrption (Created by Apache Axis)
Graph Model Visualization in several formats. Including JPEG and PDF (Created by Graphviz)

Figure 4. Service Generator Toolkit (SGT) Overview. This diagram shows the data flow between components of the SGT and a few 3rd party tools. Items within the blue box represent the SGT. Items within the yellow box represent the output files of the SGT. Items at the bottom of the diagram represent tasks the user can do with the output of the SGT. Detailed information about each manual step and the 3rd party tools can be found in the SGT tutorial.

called Graph Model, and several graph model exporters such as Java Remote Procedure Call (RPC) Service Unit Test Exporter, Java RPC Service Exporter, Ontology Web language – Services (OWL-S) Semantic Web Service Exporter, and Graph Visualization Exporter.

The Graph Model is a specialized graph implementation used to represent web services and their relationships. It is essentially a data container that provides little additional functionality. The graph model generator called Random Graph Generator generates an instance of the graph model. This graph model instance is then passed to several graph model exporters. Each graph model exporter exports the graph model in a specific file format. For instance, the Graphviz Visualization Exporter will export the graph model as DOT formatted text files for use in the Graphviz visualization tool [7].

5. Graph Model

A service graph is a collection of services along with the relationships between the services. Often service graphs are depicted as a simple directed graph, as shown in Figure 5. Each vertex represents a service.

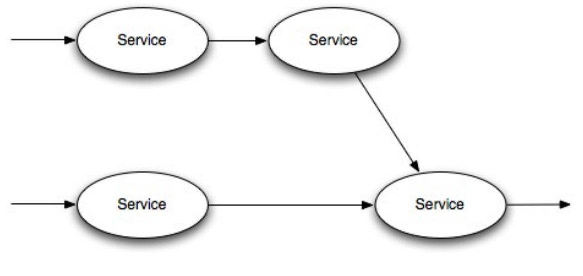

Figure 5. A simple service graph model. Services are vertices that are connected by directed edges if the output of the source Service can serve as input to the destination Service.

Two services are connected by a directed edge if the output of one service can be used as the input of the other service. This simple model is insufficient to model real world services.

In order to capture the necessary detail we use an alternative graph representation similar to that used by Oh et al., shown in Figure 6. Operations and parameters are now vertices. A directed edge exists from an operation to a parameter if the operation returns that parameter. Similarly, a directed edge exists from a parameter to an operation if the parameter can be used as input for that operation. Note that this is a directed, bipartite graph. Unlike Oh et al., we do not consider Services to be vertices but rather containers for operations. Also parameters can be tagged as optional, allowing them to represent a greater range of input and output conditions such as quality of service conditions. [3, 12]

Our implementation of this graph model enforces very few restrictions on the structure of the graph. This design choice allows graph generators a great deal of freedom and control over graph creation. It also allows graph generators to creation contradictory and invalid graph models.

6. Graph Generation

Ideally a graph generation algorithm will produce a randomized instance of a service graph with characteristics similar to that of real world services. To this end, we collected information about the characteristics of web services and service graphs from the existing literature.

In general, the number of operations per service is rather low, with the majority having less than five operations. Also, the number of input or output parameters per operation is also expected to be rather

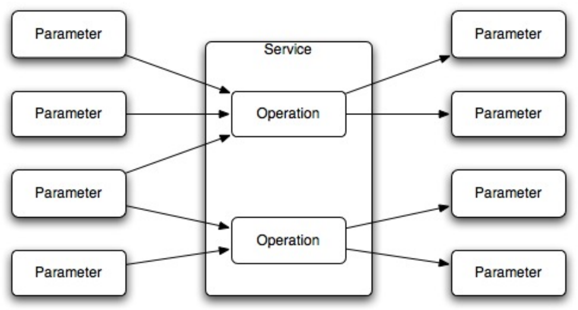

Figure 6. Detailed service graph model. In this model, Parameters and Operations are vertices that form a directed, bipartite graph. Services are containers for Operations.

low. Services are unlikely to self-loop. In addition, graph density is very sparse and connectivity is very disjoint for real world services. However this last consideration may change as more services are deployed [9].

Based on this information, we devised a graph generation algorithm with some user modifiable constraints. The user modifiable constraints are as follows:

a. The number of services created.
b. The maximum number of operations per service.
c. The maximum number of input and output parameters per operation.
d. The number of domains.

In this paper, domain means simply a cluster of related parameters. If there is more than a single domain, the creation of some graph structures such as self-loops is not possible. Since some researchers feel domains provide a more accurate representation of web services, we support them at the graph generation level. This is an implementation choice as it could be done at the graph model level [3, 12].

The random graph is generated by following the random graph generation algorithm:

1. Create a pool of parameters and assign the parameters to domains.
2. Create a pool of operations and randomly assign them input and output parameters. If possible, the input and output parameters should come from different domains.
3. Create services and randomly assign operations to the services.

As more information about service graphs becomes available, new graph generators can be implemented and added to the Service Generator Toolkit without modification to the rest of the SGT.

7. Graph Exporters

The Graph Exporters are a collection of classes responsible for building output files from a Service Graph. All of these exporters work by filling out a template with the information from the Service Graph. The current implementation has four exporters: Java RPC JUnit Test Exporter, Java RPC Service Exporter, OWL-S Semantic Web Service Exporter and Graph Visualization Exporter.

7.1. Graph Visualization Exporter

Often understanding is enhanced with the use of a visual aid. Therefore the Graph Visualization Exporter exports the generated service graph in dot format. Dot is a plain text format for describing small to medium sized graphs. This file format is used by the open source graph visualization application Graphviz [7]. Figure 7 shows a small Service Graph in Graphviz.

7.2. Java RPC Services Exporter

The Java RPC Services Exporter provides the means to convert a service graph into executable web services by exporting Java implementation source files. These source files can be deployed onto an Apache Axis/Tomcat server for invocation. The Apache Axis/Tomcat server will also generate WSDL web services description files.

In our graph model, each service takes list of inputs and returns a list of outputs. For the purposes of our concrete implementation we add the additional constraint that all inputs and outputs are lists of Strings. When a service operation is invoked, it checks whether the input String array matches the expected String array. If matched, it will return the output String array. The current implementation only supports RPC style encoding.

7.3. Java RPC JUnit Test Exporter

The Java RPC JUnit Test Exporter generates

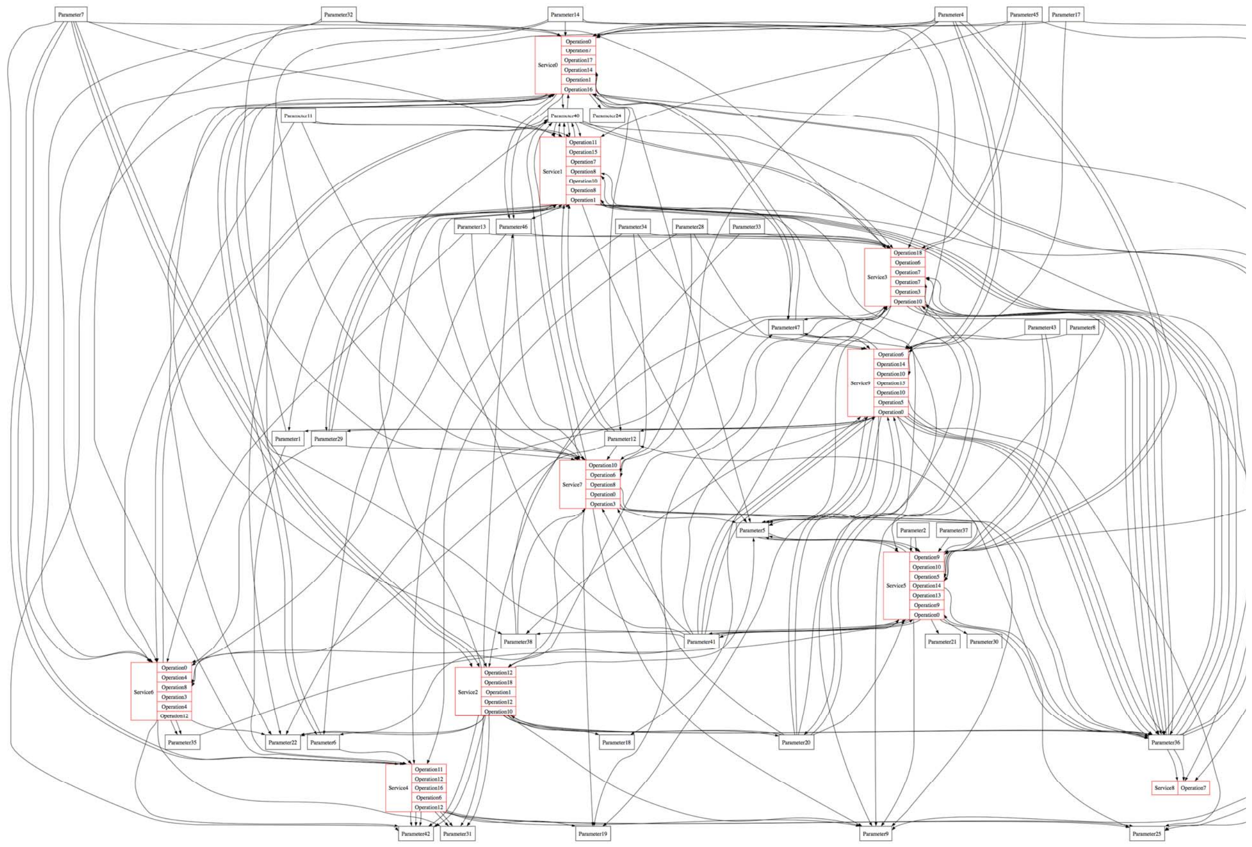

Figure 7. Visual representation of a generated Service Graph Model. This particular example shows the connections between services and parameters.

Table I.
Comparison of Service Generator Toolkit to Other Model-Based Service Testing Tools

Approach	Service Graph Model	WSDL Web Services Description	Semantic Web Services Description	Java RPC Implementation	Unit Tests	Visualization Support
Services Generator Toolkit (SGT)	Yes	Yes Generated when Java implementation is deployed.	Yes	Yes	Yes	Yes
WSBen	Yes	Yes	No	No	No	Yes
Large-Scale Test Bed	Yes	No	No	No	No	No

functional unit tests to help verify that the generated services are deployed and working correctly. These tests can be run using JUnit.

7.4. OWL-S Semantic Web Exporter

OWL-S, WSDL for Semantics (WSDL-S) and Web Service Modeling Ontology (WSMO) are a few of the competing formats for Semantic Web Services. We chose to work with OWL-S mainly due to available tools and API support.

An OWL-S Semantic Web Service interface description consists of a Service owl, Grounding owl, Profile owl and Process owl. The Service owl is the top level Service definition. The Grounding owl describes how to access the Service. The Profile owl describes what the Service does. The Process owl describes the Service process model. The Java web service must also be deployed as OWL-S only describes the interface [4].

The OWL-S template used in the SGT is derived from the template in the WSDL2OWLS tool that is included with the OWLS API [8].

9. Demonstration of the Toolkit

We demonstrate how a web services researcher can automatically generate web services using the SGT: A web services researcher wishes to create ten Java Web Services with OWL-S descriptions for use in testing a service composition engine.

Prior to using the SGT, the following tools must be installed: 1) Java 5, 2) Apache Tomcat 5.5, 3) Apache Axis 1.4, 4) Apache Ant 1.6.5, 6) JUnit 4, 7) Graphviz, and 8) any Web Server.

Once the tools are installed, verify that everything is in a clean starting state. The output folders for unit tests, web services and owls should be empty. It is also a good idea to reset Axis using the 'ant remove' and 'ant clean' commands from the root Axis directory.

For most test scenarios, including the case study, the default graph generation settings are sufficient. The default settings will create a dense, randomized graph model. It is possible to change these settings in the configuration file, 'serviceTestKit.properties.' This configuration file is a standard Java properties file, for more information on Java properties files please see the Java documentation.

The SGT can be run from the command line or from within the Eclipse IDE. In both cases run the class 'ServiceTestKit' to generate the graph and output files. The SGT will output progress messages while it creates a graph model and converts it to the output files. The 'ServiceTestKit' does not take any command line arguments.

To visualize the Service Graph, open the exported DOT file in Graphviz. It is also possible to export the visualization file into a more common file format such as JPEG or PDF from Graphviz. Figure 7 shows an example of the graph visualization.

To deploy the Java RPC Web Services, copy the generated Web Service class files to the appropriate Axis directory. On Unix-like systems a 'symlink' is recommended. Next, build and install the Web Services onto Axis by using the 'ant all' and 'ant install' commands. Deploy the web services using the Deploy tool. WSDL files are automatically generated by Axis. Use the JUnit unit tests to verify that the Web Services work properly. A screenshot of the deployed the Web Services is in Figure 8.

Deploy the OWL-S Semantic Web Services description files to the Web Server.

11. Discussion & Future Research

In comparison to collecting or manually building web services, the Services Generator Toolkit provides researchers with a simple, easy and reliable way to generate web services for use in testing. Also it takes the idea of using service models to generate test services further than previous efforts by providing the means to invoke the created services. Table 1 summarizes our comparisons of the SGT to other approaches. Among the three approaches, the SGT provides more features for web service researchers.

Future research could add support for Java Doc-Style Web Services and WSDL-S Semantic Web Services. Doc-Style Web Services have different performance characteristics than RPC-Style Web Services therefore also providing Doc-Style would be helpful when testing Service Broker designs [11]. Major industry players, such as IBM, are moving towards WSDL-S based Semantic Web Services over OWL-S based Semantic Web Services [1]. Given this industry trend, it would be helpful to provide support for WSDL-S.

12. References

[1] Akkiraju et al. “Web Service Semantics – WSDL-S.” www.w3.org. November 7, 2005. Access September 15, 2008. http://www.w3.org/Submission/WSDL-S/

[2] Burstein, M. Bussler, C. Finin, T. Huhns, M.N. Paolucci, M. Sheth, A.P. Williams, S. and Zaremba, M. A Semantic Web Services Architecture. IEEE Internet Computing, Vol. 9, No. 5. pp. 72- 8. (10 pages)

[3] Constantinescu, B. Faltings, and W. Binder. Large Scale Testbed for Type Compatible Service Composition. ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services, Whistler, Canada, June 2004.

[4] D. Martin et al. OWL-S: Semantic Markup for Web Services. Access September 15. http://www.daml.org/services/owl-s/1.1/overview/

[5] D. Martin et al. “OWL-S: Semantic Markup for Web Services.” www.w3.org. November 22, 2004. Access September 15. 2008. http://www.w3.org/Submission/OWL-S/

[6] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. R. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara (SRI, CMU, Univ. Toronto) "Bringing Semantics to Web Services: The OWL-S Approach." First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) 6-9, 2004, San Diego, California, USA.

[7] Ellson, J. and Ganser, E. Graphviz – Graph Visualization Software. Access September 15, 2008. http://www.graphviz.org/

[8] Mindswap. “OWL-S API.” www.mindswap.org. Access September 15, 2008. http://www.mindswap.org/2004/owl-s/api/index.shtml

[9] Fan, J. and Kambhampati, S. A Snapshot of Public Web Services. ASU CSE TR 04-004. August 2004.

[10] IBM Services Architecture Team. “Web Services architecture overview.” www.ibm.com. September 6, 2000. Access September 15, 2008. http://www.ibm.com/developerworks/web/library/w-ovr/

[11] IBM Services Architecture Team. “Which style of WSDL should I use?” www.ibm.com. October 31, 2003. Access September 15, 2008. http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[12] Oh, S.C., Kil, H.Y., Lee, D.W., and Kumara, S. WSBen: A Web Services Discovery and Composition Benchmark. In IEEE Int'l Conf. on Web Services (ICWS), page 239-246, Chicago, IL, USA, September 2006

[13] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. F. 2 Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Upper Saddle River, NJ: Prentice Hall PTR. 2005. ISBN 0131488740. pp 3-59.

[14] “W3C Semantic Web Activity.” 2004. World Wide Web Consortium. June 1, 2006. Access September 15, 2008. http://www.w3.org/2001/sw/

[15] Web Service Modeling Language (WSML). Access September 15, 2008. http://www.wsmo.org/wsml/