2011 Tenth International Symposium on Autonomous Decentralized Systems

An Approach for Service Composition and Testing for Cloud Computing

Wei-Tek Tsai[1,2], Peide Zhong[1], Janaka Balasooriya[1], Yinong Chen[1], Xiaoying Bai[2], Jay Elston[1]
[1]School of Computing, Informatics, and Decision System Engineering
Arizona State University
Tempe AZ 85287
[2]*Department of Computer Science and Technology, Tsinghua University, Beijing, China 100084*
E-mail: Peide.Zhong@asu.edu

***Abstract*— As cloud services proliferate, it becomes difficult to facilitate service composition and testing in clouds. In traditional service-oriented computing, service composition and testing are carried out independently. This paper proposes a new approach to manage services on the cloud so that it can facilitate service composition and testing. The paper uses service implementation selection to facilitate service composition similar to Google's Guice and Spring tools, and apply the group testing technique to identify the oracle, and use the established oracle to perform continuous testing for new services or compositions. The paper extends the existing concept of template based service composition and focus on testing the same workflow of service composition. In addition, all these testing processes can be executed in parallel, and the paper illustrates how to apply service-level MapReduce [1] technique to accelerate the testing process.**

Keywords: Cloud, Software as a Service (SaaS), Service Composition, Workflow, and Testing.

I. INTRODUCTION

As cloud computing proliferates, service composition and testing for cloud applications bring new challenges. In traditional Service-Oriented Computing (SOC), service directory UDDI [2] is used to organize services information and help providers to discover service. BPEL and OWL-S are commonly used languages in service composition, However, they do not consider testing of composed services. Often, service composition and testing are distinct processes. In [3], a template-based service composition is proposed where service composition and testing are carried out in an integrated manner. Also, in cloud computing, continuous testing is proposed with testing in every stage of application development and composition [4, 5].

A service has its specification, interface (such as input and output), and implementation. Each service specification defines an interface and the functionality, as well as nonfunctional, of implementations. Note that multiple implementations may implement one service specification. A service interface describes the input/output information and thus it cannot be executed. Domain ontology can be used to organize and manage service specifications and interfaces, and maintain one-to-many relationships to service implementations. One way of service composition to design a workflow with service specifications, and let the cloud platform choose fit service implementations. Developers use service interfaces to perform service composition and let the cloud to discover the matching service implementations . However, a developer may prefer to have more control in selecting service implementations. Thus, this paper proposes service injection as a way to allow developers to designate specific service implementations. In addition, the paper elaborates the process of service testing with service composition.

As service providers publish their services in clouds, many of them will implement the same service specifications and interfaces. If users use those service interfaces to compose a workflow, there can be many possible combinations. For example, in Figure 1, if a workflow uses *n* interfaces, and each interface has *m* implementations, the number of combinations becomes m^n and this number can be huge. Therefore, testing them quickly is challenging. This paper proposes a voting mechanism based on service-level *MapReduce* [1] to expedite the testing process.

This paper addresses following issues:

- To allow the service composition mechanism in [3] to designate specific service implementations;
- To develop the testing mechanism for the workflow used in service composition; and
- To study the parallel testing technique to accelerate the testing process when the number of services combinations for the workflow is large.

This paper is organized as follows. Section II presents the related work. Section III introduces background of service management and composition on the cloud. Section IV discusses our approach in concurrent testing and service composition. Section V gives a case study on a shipping domain service composition system. Section VI presents conclusions and future directions.

II. RELATED WORK

It is important to test cloud services to provide valid and stable services for cloud users. In [6], an ontology-based approach for web service testing is proposed, where a test ontology model was used to define test concepts, relationships, and semantics. In [7], a model-based adaptive test (MAT) that can select and rank test cases to eliminate redundant testing is proposed. The technique can be used to test a group of software artifacts that implement same specification.

Ontology is often used for knowledge representation, sharing, classification, reasoning, and interoperability. Ernestas and Lina [8] proposed a method of transforming ontology representation in OWL to relational database and developed algorithms that transform domain ontology to relational database. Bianchini and Antonellis [9] presented an ontology design approach called VISPO (Virtual-district

978-0-7695-4349-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ISADS.2011.90

Internet-based Service Platform) that supported knowledge sharing and service composition in virtual districts. In [10], Woo and Radhika presented a task dependency approach for web service composition driven by business rules statically. In [3], domain ontology is used to organize service interfaces (*SInt*) and one-to-many relations to service implementations (*SImp*) and their use scenarios that is a semi-formal specification of how the system can be used by other systems [11]. Users can utilize service interfaces to compose workflow applications and use scenarios, which depends on how to map this service interface workflow to implementations or to use scenarios. If it is mapped to service implementations, it becomes a composite service. Otherwise, it becomes a use scenario to drive composite service testing with test input. By this way, service composition and testing can share the same workflow. However, this approach does not detail how to perform testing. This paper extends the techniques in [3] and uses group testing technology in [12] to do test the workflow.

Service composition is important in SOA and cloud computing. Many service composition approaches are available including model-based approaches [13-15], semantic-based approaches [16-18], template-based approaches [19, 20], and QoS-driven approaches [21-23]. Among cloud service composition approaches proposed, [24] described an ontology-based dependency-guided service composition approach for user centric SOA and applied this approach to the cloud in [3]. The SRDF framework presented in [24] proposed a domain ontology to manage service interfaces. Users can employ these service interfaces to do service composition, which it is the cloud's responsibility to map them to service implementations based on users' preference.

Guice is a dependency injection framework developed by Google for Java platform [25]. It allows an interface to be programmatically bound to implementation classes and inject constructors, methods or other fields into the classes by an @Inject annotation tool. When more than one implementation of the same interface is needed, users can create custom annotations to identify an implementation and use that annotation when injecting. Spring is another application framework for the Java platform [26], It provides more functionality than Guice does as Sring has inversion of control (IoC), injection, annotation and other functions. IoC can be called dependency injection where developers do not need to create their objects but describe how they are created and do not directly connect components but describes which components needed by objects in configuration file. IoC container is in charge of these operations. Spring-Annotation is a library that allows developers to use annotations to configure their application by spring-framework. It is possible to configure Spring's dependency injection using annotation. Both of them proposed similar mechanism like dependency injection [27], which use interfaces to inject class implementation later. Injection and annotation functions of Spring are similar to the functions that Guice provide, except Spring uses xml script instead of Java program.

III. CLOUD SERVICE MANAGEMENT AND COMPOSITION

A. Cloud service management and composition

Service Interface (SInt) describes service interface, and it may include the service's input, output, specification, test cases, and use scenarios. Interface service can be organized by a domain ontology, which maintains relationships among service interfaces and service implementations. The relationship between service interfaces and service implementations is one-to-many.

Service Implementation (Simp) implements an service specification [28].

Domain ontology: Domain ontology expresses domain information and represents entities (as nodes), relationships and constraints. It can organize and manage service interfaces. One example is illustrated as Figure 1.

- *Nodes:* A node is a unique entity that represent an service interface in domain ontology. Every node has zero or more corresponding implementations that have been verified. In addition, each node's implementation must have the same input and output so that they can be dynamically replaced by each other.
- *Relationships:* It is a connection between two nodes in the domain ontology.
- *Service Workflow or Application Template:* they are composed by service interfaces represented by a domain ontology and control structures as illustrated as Figure 3.

B. Extensions to allow users designate specific services

In [3], service implementations' selection mainly depends on dependency information among service interfaces and users service selection history. Both of them do not allow user to designate specific service implementation programmatically. This paper extends service selection mechanism using Service Injection (*SI*) that allows users to compose workflow or application templates by service interfaces and inject service implementation later. Developers have two options to specify *SI*:

1) By Configuration: Developers can fill all service selection information such as service name and method names. Inputs and other related information can be added into the the configuraion file "groundProfile.xml". This approach is based on the Spring tool. It is convenient for users who want to do configuration rather than programming.

Figure 2 gives an example of the *groundProfile.xml*. In the upper part of groundProfile, a service interface NotificationService is mapped to the method notification of "Company A Notification Service". Its input parameter is username whose value is Tommy. In the bottom part of this file, a workflow ChangeNotificationWay is defined. It uses the reference "notificationServiceByCompanyA" defined in the upper part. If developers want to use different service implementations, they can add another service interface mapping like "Company A Notification Service" and change

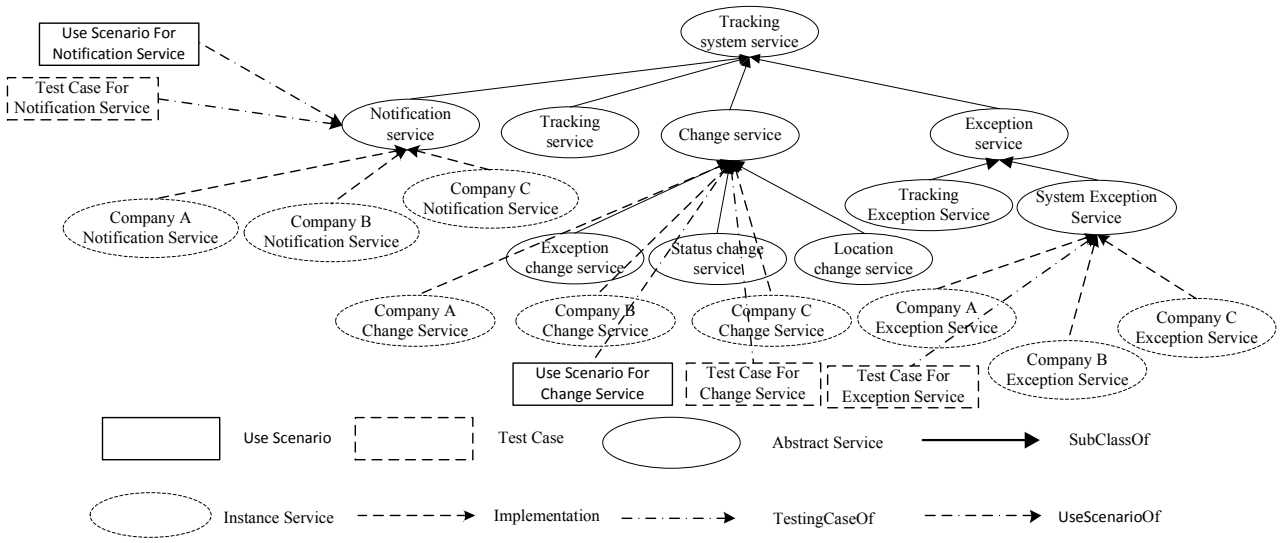

Figure 1. Domain Ontology Example

```
<?xml version="1.0" encoding="UTF-8"?>
<sevices xmlns="ASU service injection namespace"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<service id="notificationServiceByCompanyA" interface="NotificationService"
implementation="com.vaannila.SpringQuizMaster">
    <property serviceName="Company A Notification Service">
        <methodName>notification</methodName>
        <input>
            <org1 name="userName">Tommy<\org1>
        </input>
    </property>
</service>
<service id="changeNotificationWayWorkflow" interface="ChangeNotificationWay">
    <property serviceName="notificationService">
        <ref local="notificationServiceByCompanyA"/>
    </property>
</service>
</services>
```

Figure 2. A GroundProfile Example

the reference. By this way, developers can select service by injecting service into a configuration file. The composed workflows or application templates can be reused by others.

2) By Programming: Developers can fill service selection information by calling GroundService.bind() method as illustrated as following code:

```
[WebMethod]
public void bind(Object IntS, Object ImpS)
{
    Bind binder = new Bind();
    binder.bind(Ints).to(ImpS);
}
```

From the code one can see that, it mainly binds service interface to the service implementation. Developers can use GroundService like normal web service and fill in service interface and service implementation as shown below.

```
CompanyANotification ImpS = new CompanyANotification();
GroudService myBinder = new GroudService();
myBinder.bind(NotificationService,ImpS.notification("Manager"));
```

In this way, application developers can designate their service implementations. This is similar to the class injection mechanism used in Google Guice [25].

C. Code generation support

In [3], developers compose workflows by dragging and dropping services from domain ontology, which is implemented by a tool [29]. One workflow example is illustrated as shown in Figure 3 and the generated source code is presented as Figure 4. How to generate source codes of *SI* is similar with Spring [26] and Guice [25]. And it can be easily implanted into PSML-S.

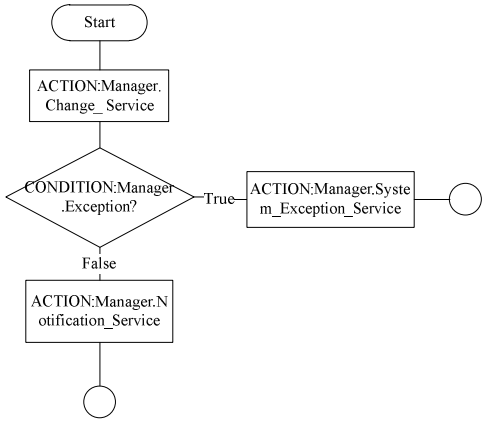

Figure 3. Workflow Example

```
do ACTION Manager.Change_Service
if CONDITION::Manager.Exception
{
   do ACTION Manager.System_exception_Service
}
else
{
   do ACTION Manager.Notification_Service
}
```

Figure 4. Generated Source Code of Workflow

D. *Service customization supports*

Service customization is important for SaaS (software as a service) [30]. One solution is to use a multi-layer customization framework OIC to support customization [31]. In this framework, templates are used to assist SaaS customization. However, services that compose the templates are not easy to change. To better reuse templates, they can be composed by service interface and do *SI* when developers customize SaaS. So, developers do not need to modify the template if they just want to change services.

IV. TESTING WORKFLOWS OF SERVICE COMPOSITIONS

Both the number of cloud services available and the size of data that cloud services need to handle are often large. Testing workflows composed of those cloud services is a challenge. This paper follows the service group testing [7, 12] to test workflows.

1) *Oracle Generation of Composite Services*

A test case is a pair (test input, expected output), but often the expected output is difficult to obtain. As users compose services by using *SInts* and each *SInts* can be implemented by different providers (*SImps*), there is a large number of combinations for the same workflow if the cloud chooses different *SImps*. Oracle generation mechanisms in [7, 12] can be used to determine the expected output. Oracle generation uses a voting mechanism to establish an oracle with a confidence level, and if the confidence level is high enough, the corresponding oracle can be used to determine the pass/fail of subsequent tests, i.e., a test case is established (input, established oracle with a confidence level).

2) *Unit Testing*

Once a test case is formed with an established oracle, it can be used to test new service implementation. Test cases can also be ranked to help the cloud and users to select the most potent test cases to run first and often.

After an oracle for a test input has been set up for a workflow, integration testing can be used to test each service by changing one service implementation at a time. If testing results are consistent, the new service is considered as correct with a confidence level with respect to the test case. When a sufficient large number of test cases pass the test, the new service implementation is considered as validated with another confidence level, otherwise the new implementation will be rejected.

3) *Integration Testing*

Once the workflow is completed, one can apply its *use scenarios* [11] as test scripts to test the workflow. A user scenario of a workflow or a service is essentially an application that uses the workflow or the service respectfully. One can say that the workflow in Figure 4 is a use scenario of all the participating services such as notification_service. Use scenarios for a service can be collected and served as a part of the service specification, and they can be used for service composition. A use scenario for a service may involve other services, and thus this use scenario provides a relationship between these two services, the relationship indicates these two services are linked to each other. During service composition, once a service is selected, the linked service becomes a candidate for composition. Furthermore, the use scenarios for a workflow can be collected and used later for service composition or as the basis for test scripts.

4) *Continuous Testing*

Continuous testing can be a part of the TDD (Test-Driven Development) process. Continuous testing is a testing process that is being applied during the development and execution stages. In traditional continuous testing, testing mostly in the form of regression testing is applied during the entire development time 24 hours a day [32]. In clouds, as new applications may be composed from existing services, continuous testing can be applied *before* and *after* application and service composition, and even during execution as a part of the service monitoring and/or policy enforcement processes.

Continuous testing can be used to test SaaS applications [4] by embedding built-in test case generation with the metadata database associated with a tenant in the SaaS. In this case, test cases can be selected to test the SaaS applications continuously. If a test script detects a failure, the ranking of the test script with its associated test case will be increased so that it will be used early and more often in continuous testing. The cloud platform can run those test scripts continuously selecting most potent test case with dynamic ranking of test cases.

5) *Metadata-driven Test Input Generation*

For inputs and outputs of service interfaces, one can use metadata to define test inputs [4]. For example, if the length of user ID for a website must be 64 bits, the simple test inputs can be generated by randomizing the 64 bits. One can generate a collection of user IDs of 64 bits, another collection with 128 bits or any other bits.

6) Execute testing processing by service-level MapReduce way

As the number of *SImp* can be large on a cloud, testing needs to be effective and efficient. So, group testing can be applied using service-level MapReduce [1] to run tests in parallel. The idea is to use different combinations of service implementations for the same workflow in the *map* step. The input of the workflow is either produced by developers or generated from the metadata. The majority of immediate results from the *map* step is reduced to generate an oracle in the *reduce* step by the voting mechanism [7]. It can be illustrated as in Figure 5:

a. Users submit the composed service to the cloud;

b. Cloud management service (*CMS*) calculates how many workers needed based on configurations;

c. *CMS* finds enough available workers to run services.

d. *CMS* discovers service combinations for the composed service and dispatches them to the workers;

e. *CMS* calls the input service or provides test input from developers to start service-level MapReduce;

f. In the *map* step, service combinations are executed; results are sent to cache services for shuffling;

e. In the *reduce* step, the voting is done to establish an oracle;

g. If an oracle is found, it is sent back to users as an validated oracle. If not, the composed service fails to establish an oracle with this test input.

From Figure 5, one can see that:

a) Each service combination for the workflow is running on different worker machine of the cloud with same inputData service.

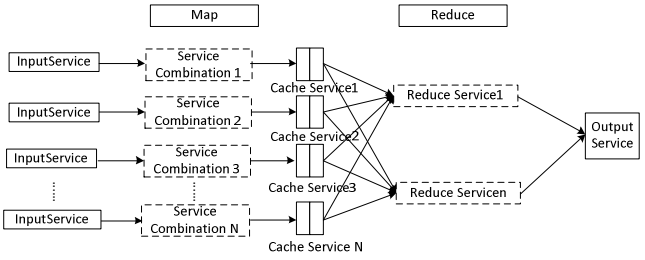

Figure 5. Oracle Generation Service Level MapReduce process

b) All service combinations can be run in parallel.

c) Cache service can be used to shuffle the immediate results of map process and dispatch them to different reduce services to get final result. This process is controlled by service-level MapReduce [1].

d) If output data produced are consistent with limited deviation [32], i.e., a majority can be established, an oracle can be established with a confidence level. Otherwise, no such oracle can be established, and another test input need to be run to establish its own oracle.

V. CASE STUDY

The following case study illustrates the techniques discussed for a cloud application. This study uses a shipping domain service composition system to illustrate the composition process. We obtained the requirements from three different companies. According to applications' requirements, three different application systems have been developed. Now, a manager wants to incorporate certain changes for the tracking system. She composes a workflow by using service interfaces illustrated in Figure 1. The manager chooses services provided by company A, one can use the following code to map the service interfaces to implementations.

```
CompanyANotification ImpS = new CompanyANotificationService();
GroudService myBinder = new GroudService();
myBinder.bind(NotificationService,ImpS.notification("Manager"));
CompanyAExceptionService ses = new CompanyAExceptionService ();
myBinder.bind(SystemExceptionService,ses);
CompanyChangeService ccs = new CompanyChangeService();
myBinder.bind(ChangeService, ccs)
```

The cloud incorporates this information for the workflow, as is illustrated as Figure 6.

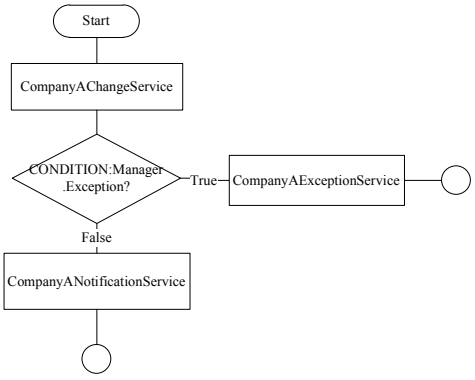

Figure 6. Workflow after Service Implementation Mapping

After applying use scenarios to generate test scripts to run testing, the oracle generation can then be used to establish oracles for test cases. After all kinds of testing have done, the workflow can be run in the cloud.

VI. CONCLUSIONS

This paper proposed a new approach to manage services on the cloud to allow users to compose services and test them. Users use an service interface to perform service composition and let the cloud environment to map the interface to a suitable service implementation. Users can designate a specific service by configuration or programming. Test scripts and cases can be developed using metadata or use scenarios. A voting mechanism is used to create the test oracles with a confidence level. As the

number of cloud services is large, a service-level MapReduce is used to execute testing.

ACKNOWLEDGMENT

This project is partly sponsored by U.S. Department of Education FIPSE project P116B060433, U.S. National Science Foundation project DUE 0942453, and Department of Defense, Joint Interoperability Test Command. The research is also partly sponsored by the European Regional Development Fund and the Government of Romania under the grant no. 181 of 18.06.2010.

REFERENCES

[1] W. T. Tsai, *et al.*, "Optimal Service Replication Strategies for MapReduce On the Cloud," *The International Symposium on Autonomous Decentralized System (ISADS), 2011*.

[2] E. Newcomer, *Understanding Web Services: XML, Wsdl, Soap, and UDDI*: Addison-Wesley Professional, 2002.

[3] W. T. Tsai, *et al.*, "Template-Based Service Composition for User-Centric SOA," *the 22nd International Conference on Software Engineering and Knowledge Engineering,* 2010.

[4] W. T. Tsai, *et al.*, "Towards a Scalable and Robust Multi-tenancy SaaS " *The Second Asia-Pacific Symposium on Internetware (Internetware'10),* Suzhou, China, Nov. 2010.

[5] D. Saff and M. D. Emst, "Continuous testing in eclipse," *In 2nd Eclipse Technology Exahcange Workshop (eTX),* Barcelona, Spain, March 2004.

[6] X. Bai, *et al.*, "Ontology-Based Test Modeling and Partition Testing of Web Services," 2008, pp. 465-472.

[7] W. Tsai, *et al.*, "A Coverage Relationship Model for Test Case Selection and Ranking for Multi-version Software," *High Assurance Services Computing,* pp. 285-311, 2009.

[8] V. Ernestas and N. Lina, "Transforming Ontology Representation From OWL To Relational Database," *Information Technology and Control, vol. 35, 2006*.

[9] D. Bianchini and V. D. Antonellis, "Ontology-based Integration for Sharing Knowledge over the Web," *DiWeb2004 - 3rd International Workshop on Data Integration over the Web, June 2004*.

[10] K. Jong Woo and J. Radhika, "Web Services Composition with Traceability Centered on Dependency," in *the 38th Hawaii International Conference on System Sciences*, 2005, pp. 89-89.

[11] W. T. Tsai, *et al.*, "Semantic Interoperability and Its Verification and Validation in C2 Systems," presented at the 10th International Command and Control Research and Technology Symposium (ICCRTS), 2005.

[12] R. Bryce, *et al.*, "Biased covering arrays for progressive ranking and composition of web services," *International Journal of Simulation and Process Modelling, 2007,* vol. 3, pp. 80-87.

[13] L. Chen, *et al.*, "ECA Rule-Based Workflow Modeling and Implementation for Service Composition," *The Institute of Electronics, Information and Communication Engineers (IEICE) Transactions,* vol. 89-D, 2006, pp. 624-630.

[14] R. Gronmo and M. C. Jaeger, "Model-Driven Semantic Web Service Composition," presented at the the 12th Asian-Pacific Software Engineering Conference (APSEC), 2005, Taipei, Taiwan.

[15] B. Orriens, *et al.*, "Model Driven Service Composition," presented at the First International Conference on Service-Oriented Computing (ICSOC), Trento, Italy, December 15-18.

[16] K. Fujii and T. Suda, "Semantics-Based Context-Aware Dynamic Service Composition," *ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2009,* vol. 4.

[17] S. Kona, *et al.*, "USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition," *Int. J. Web Service Res.,* vol. 6, 2009, pp. 20-48.

[18] A. Brogi, *et al.*, "Semantics-based composition-oriented discovery of Web services," *ACM Trans. Internet Techn.,* vol. 8, 2008.

[19] E. Sirin, *et al.*, "Template-based composition of semantic web services," 2005, pp. 85-92.

[20] K. Geebelen, *et al.*, "Dynamic reconfiguration using template based web service composition," presented at the 3rd Middleware for Service Oriented Computing (MW4SOC), 2008.

[21] Y. Dai, *et al.*, "QoS-Driven Self-Healing Web Service Composition Based on Performance Prediction," *J. Comput. Sci. Technol.,* vol. 24, 2009, pp. 250-261.

[22] L. Yang, *et al.*, "Performance Prediction Based EX-QoS Driven Approach for Adaptive Service Composition," *J. Inf. Sci. Eng.,* vol. 25, 2009, pp. 345-362.

[23] S. Meng and F. Arbab, "QoS-Driven Service Selection and Composition," presented at the 8th IEEE International Conference on Application of Concurrency to System Design (ACSD), 2008.

[24] W. T. Tsai, *et al.*, "Dependency-Guided Service Composition for User-Centric SOA," *2009 IEEE International Conference on e-Business Engineering,* vol. 0, pp. 149-156.

[25] Google. *Guice*. Available: http://code.google.com/p/google-guice/

[26] S. Source. *Spring*. Available: http://www.springsource.org/

[27] Wikipedia. *Dependency Injection*. Available: http://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=353809060

[28] W3C. *Web Service*. Available: http://www.w3.org/2002/ws/

[29] W.T.Tsai, *et al.*, "PSML-S: A Process Specification and Modeling Language for Service Oriented Computing," *The 9th IASTED International Conference on Software Engineering and Applications (SEA), Phoenix, November 2005*.

[30] Wikipedia, "Software As a Service," *http://en.wikipedia.org/w/index.php?title=Software_as_a_service&oldid=341352621,* 2010.

[31] W. T. Tsai, *et al.*, "OIC: Ontology-based intelligent customization framework for saas," presented at the Int. Conf. on Service Oriented Computing and Applications(SOCA'10), Perth, Australia, Dec, 2010.

[32] E. Smith, "Continuous testing," *Proceedings of the 17th International Conference on Testing Computer Software,* 2000.