

Testability of Software in Service-Oriented Architecture

Wei-Tek Tsai, Jerry Gao*, Xiao Wei, Yinong Chen

Arizona State University, Tempe, AZ 85287-8809

*San Jose State University, San Jose, USA

Abstract

Service-Oriented Architecture (SOA) is a system architecture in which a collection of loosely coupled services communicate with each other using standard interfaces and message-exchanging protocols. As an emerging technology in software development, the SOA presents a new paradigm, and it affects the entire software development cycle including analysis, specification, design, implementation, verification, validation, maintenance and evolution. This paper proposes several testability evaluation criteria for SOA software, which serves as a reference for both service providers and application builders to evaluate the test support to SOA software. The proposed evaluation criteria are illustrated in a stock-trading case study.

Keywords

Testability, test criteria, service-oriented architecture (SOA).

1. Introduction

Technically, a **service** is the interface between the producer and the consumer. From the producer's point of view(s), a service is a well-defined and self-contained function module that does not depend on the context or state of other functions. In this sense a service is often referred to as a service agent. These services can be newly developed modules or just wrapped software around existing legacy software by providing them with the new interfaces. From application builders' point of views, a service is a unit of work done by a service provider to achieve the desired results to the need of a consumer. A service normally provides an application interface so that it can be called (invoked) by another service.

Service-Oriented Architecture (SOA) is a system architecture in which a collection of loosely coupled services (components) communicate with each other using standard interfaces and message-exchanging protocols [10]. These services are autonomous and platform independent. They can reside on different computers and use each other's services to achieve the desired goals and end results. A new service can be composed at runtime based on locally or remotely available services. Remote services can be searched and discovered through service brokers that publish services for public accesses. **Web Services** implement a Web-based SOA and a set of enabling standardized protocols such as XML, WSDL, UDDI, and SOAP.

Recently, ebXML (<http://www.ebxml.org/>) has introduced a set of dynamic collaboration protocols (DCP) [8] such as CPP (Collaboration Protocol Profile) and CPA (Collaboration Protocol Agreement), the key concept of DCP is that services will determine their collaboration at runtime. Previously, service collaboration is often determined by a workflow specified earlier. Thus, the DCP introduced another dimension of dynamism in SOA. This new dynamism introduces new challenges in

verification and validation (V&V) of SOA applications. One can see the evolution of challenges for verification as follows:

- Traditionally, systems and software are verified using the IV&V (Independent Verification and Validation) approach, where independent teams are used to verify and validate the system during system development.
- The SOA challenges the IV&V approach as new services will be discovered after deployment, and thus it is necessary to have CV&V (Collaborative Verification and Validation) approach, where service brokers, providers and consumers work in a collaborative manner, and some of testing need to be performed at runtime, e.g., when a new service is discovered and being incorporated into the application.
- The DCP is more challenging than the conventional SOA because even the collaboration will be determined at runtime. While it is possible to re-specify the workflow at runtime during the dynamic reconfiguration [12] in conventional SOA, it is different in DCP, where the actual collaboration is unknown until the participating services dynamically establish the protocol.

A typical DCP process can be divided into four stages:

- **Preparation/Profiling stage:** In this stage, each service that wishes to participate in DCP must prepare a CPP which contains the list of protocols that the service can understand. The CPP can be updated as the service learns from actual collaboration.
- **Establishment stage:** In this stage, participating services will exchange their CPPs, and agree on a common protocol CPA that all participating services share. .
- **Execution stage:** In this phase, participating services will collaborate based on the CPA established earlier. Data may be collected so that CPP can be updated for future collaboration.
- **Termination stage:** In this phase, participating services terminate the collaboration, and update their own CPPs based on the data collected. The collaboration session may be evaluated so that future collaboration can be improved.

Tsai [10] proposed an extended set of CPP and CPA to allow more flexible collaboration. The extension includes:

- **ECPP** (Extended Collaboration Protocol Profile)
- **ECPA** (Extended Collaboration Protocol Agreement)
- **USS** (Use Scenario Specification)
- **IPS** (Internal Process Specification)
- **CIS** (Collaboration Interface Specification)

Table 1 summarizes the comparison of traditional IV&V, SOA CV&V, and DCP Collaboration V&V.

Table 1 Comparison of IV&V, CV&V and DCP CV&V

	Traditional IV&V	Service-Oriented CV&V	DCP Collaboration V&V
Application Model	Application structure is often known at design time.	The workflow model is specified but services can be dynamically discovered.	Dynamic application composition via run-time service collaboration
Verification Approach	The verification team is independent of the development team to ensure objectivity and to avoid common-mode errors. Mostly done by software providers.	Verification by collaboration among service providers, application builders, and independent service brokers. The emphases are on runtime and just-in-time testing, and evaluation using data dynamically collected.	Before the establishment of collaboration protocol at runtime, a service has a partial view only, and thus verification will be limited. As the complete collaboration will be available at runtime, verification need to be performed at runtime.
Integration testing	Configurations and systems must be linked before integration testing	Dynamic configuration and systems are linked at runtime and verified at runtime	Dynamic configuration and collaboration are established at runtime and verified at runtime
Testing coverage	Input domain, structural (white-box), or functional (black-box) coverage.	Service providers can have traditional coverage, but service brokers and clients may have black-box (such as WSDL) coverage only.	As each service will have a partial view only, test coverage can be completed only when the actual collaboration is established.
Model Checking	Model checking on the code or state model.	Just-in-time dynamic model checking on the specification in, e.g., WSDL and OWLS.	Model checking can be performed on the partial view, and again when the complete view is available.
Simulation	Specification and models are simulated to verify the design and/or code	Just-in-time simulation for composite service	Simulation can be performed using sample applications before deployment, and just-in-time simulation can be performed when the actual collaboration protocol and parties are known at runtime.

2. Testability

Testability is an important quality indicator of services since its measurement leads to the prospect of facilitating and improving a service test process. What is **software testability**? According to IEEE standard, the term of “testability” refers to “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; and the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met.”

Services with good testability not only increase the quality of commercial composite services, but also reduce the cost of software development and testing. Thus designing highly testable services becomes an important and challenging task for service developers. Similarly, verifying and measuring the testability and quality of services is also important and challenging for application engineers and service users.

Freedman [4] defined domain testability for software components as a combination of two factors: **observability** and **controllability**. According to his definition, “observability is the ease of determining if specific inputs affect the outputs – related to the undeclared variables that must be avoided”, and “controllability is the ease of producing specific output from specific inputs – related to the effective coverage of the declared output domain from the input domain.” His approach is to treat a software component as a functional black box with a set of inputs

and outputs, and measure component testability by checking the above two factors.

Voas and Miller [13] treat software testability as one of three pieces of the software reliability puzzle --- software testability, software testing, and formal verification. Their research showed that software testability analysis is useful to examine and estimate the quality of software testing using an empirical analysis approach. In their view, software testability is “prediction of the probability of software failure occurring if the particular software were to contain a fault, given that software execution is with respect to a particular input distribution during random black box testing.” Poor component testability not only suggests the poor quality of software and components, but also indicates that the software test process is ineffective. Similar to requirements and design faults, the later the poor testability is detected during software development, the more expensive it is to be repaired.

Recently, Gao [5] presents the component testability from the perspectives of component-based software construction. Gao defined component testability based on five factors: understandability, observability, controllability, traceability, and testing support capability. For each factor, Gao also provided further refined factors. According to this work, component testability can be verified and measured based on the five factors in a quality control process. In a follow-up paper [6], Gao proposed a testability model based on the five testability factors of software components. The model further refined the five factors by providing a more practical and measurable detailed

factors, which could be used to verify and measure component testability during a component development process.

Based on the similarities and differences between software components and services, this paper proposes nine criteria on service testability evaluation, which includes the five testability factors proposed in [5] and [6]. Section 3 elaborates the newly proposed service-specific criteria and Section 4 presents a case study.

3. Understand Service Testability

The meaning of service testability is two-fold. First, it refers to the degree to which a service is constructed to facilitate the establishment of service test criteria and the performance of tests to determine whether those criteria have been met. Second, it refers to the degree to which testable and measurable service requirements are clearly given to allow the establishment of test criteria and performance of tests. Similar with component-based software, the testability of SOA software should needs to consider the five factors discussed in [6] on component-based software testability. But SOA software has its distinguished features:

1. Services in SOA may have different accessibility, i.e., each service can be published in one of the following levels:
 - Level 1: Service source code is accessible;
 - Level 2: Binary code is accessible;
 - Level 3: Model (BPEL4WS, OWL-S) is accessible
 - Level 4: Signature (WSDL) is accessible (most likely, hard for making changes)

The service publishing levels and potential testing techniques are also illustrated in Figure 1. As a result, the discussion of testability of SOA software should not only be on the code level, but also on all these four levels.

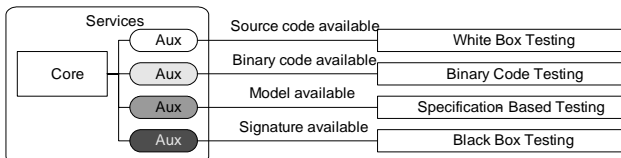


Figure 1 Different Service Publishing Levels

2. Data pedigree or provenance has received significant attention recently in SOA. In an SOA environment, the provenance of data consists of the entire processing history: the identification of the origin of the raw data set, routing of data in the SOA environment, and processing applied to the data. Note that as in an SOA system, new services can be introduced, and thus data with identical information may be routed to and processed by different services at different time.
3. Different from traditional component-based software architecture where the architecture is mainly static, the architecture of an SOA-based application can be dynamic, i.e., the application may be composed at runtime.

Due to dynamic nature of SOA, the architecture of SOA-based applications has the following characteristics:

- Dynamic architecture and dynamic re-architecture
- Lifecycle management embedded in the operation infrastructure architecture

Services in SOA are usually connected to a bus-like communication backbone such as an ESB (Enterprise Service Bus) [2]. Communications among the services are controlled by a control center, which is also attached to the communication backbone as illustrated in Figure 2. The right block in Figure 2 represents a typical control center providing fault-tolerant capability, which contains four different functional blocks to perform a variety of tasks such as analysis, modeling, management and data collection [9]. The process to change the architecture in the runtime is actually to re-integrate the services that comprised the SOA-based software. To provide enough assurance on dynamically re-architected SOA-based application, the SOA software must facilitate the establishment of corresponding test criteria.

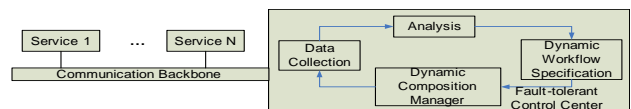


Figure 2 Factors representing the service testability.

4. V&V of DCP is currently the most challenging task in SOA V&V as it involves dynamic collaboration establishment at runtime. Testability of SOA software with DCP needs to address the testability of DCP in all the four DCP stages: preparation/profiling phase, establishment stage, execution stage and termination stage [10].

Based on these four features, this paper presents the testability of SOA software into four components:

1. **Atomic Service Testability:** focuses on the testability of atomic service with different levels.
2. **Data Provenance Testability:** focuses on the testability of the data transmitted in an SOA system.
3. **Service Integration Testability:** focuses on the testability of service integration, which involves service composition, integration, interoperability, architecting, runtime re-architecting/re-composition and the behavior of service composition control center.
4. **Service Collaboration Testability:** focuses on the testability of service collaboration.

Figure 3 illustrates the all the nine factors discussed on SOA-based application testability.

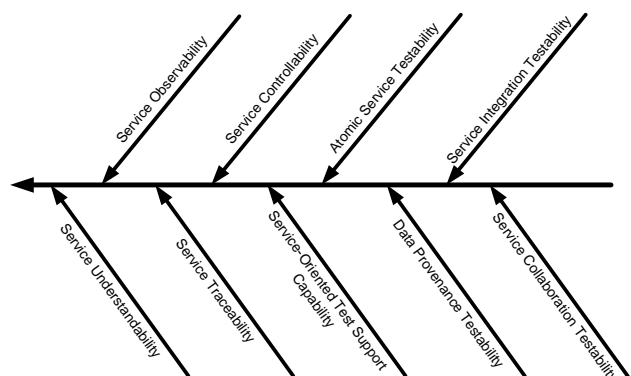


Figure 3 Factors Representing the Service Testability.

3.1 Atomic Service Testability

Test support on different service publishing levels is a new factor for SOA software testability. It refers to the indicator that represents how well atomic services are developed to facilitate the testing. As shown in Figure 4, it can be further examined based on checking the following four properties.

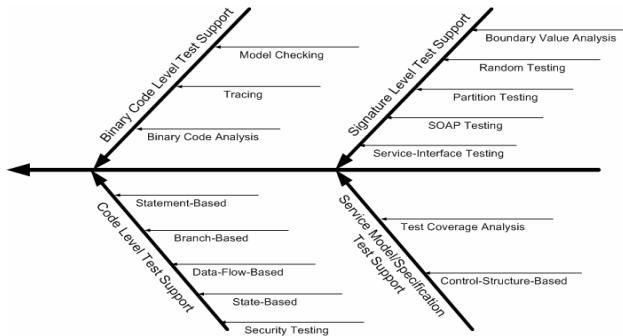


Figure 4 Atomic Service Testability

1. **Source Code Test Support:** This refers to the capabilities on the SOA service source code. As source code is available, the testability of traditional software applies here. Specifically, it includes the support on all the traditional white-box testing techniques, like statement-based, branch-based, data-flow-based, state-based and security testing.
2. **Binary Code Test Support:** This refers to the capabilities on the SOA service binary code. The major testing technique on this level is binary code analysis, tracing, and model checking. For example, Microsoft has a project Phoenix that has a tool to perform binary code analysis.
3. **Service Model/Specification Test Support:** It refers to the capabilities on the service model/specification level of the SOA software. It includes the support on all the specification-based software testing techniques, including test coverage analysis, and control-structure-based.
4. **Signature Level Test Support:** This refers to the capabilities on the signature level, including the test support on all the black-box testing techniques, like service interface testing, SOAP testing, partition testing, random testing, and boundary value analysis.

It is apparent that service with source code support testing in full strength compared to traditional software, but services that expose signatures only support the least. Services with specification/model only can have black-box testing only as no white-box testing can be used. Currently, a service often exposes at least its signature and its specification, thus it supports black-box testing. But a service provider may not expose its source code or object code, and thus making white-box testing infeasible for a service consumer.

3.2 Data Provenance Testability

Data provenance testability is another factor of SOA software testability. It refers to how data were derived, and the quality of data including reliability, dependability, trustworthiness of data as well as the processes (services) that manipulate those data during the process [3]. Groth, Luck and Moreau [7] propose seven criteria: verifiability, accountability, reproducibility, preservation,

scalability, generality and customizability. As shown in Figure 5, data provenance test support capability can be further refined to the following three factors:

1. **Data Collection Testability:** This refers to the testability of the data collection process. When monitoring and tracking the data, not all data need to be tracked. Even for those data that need to be tracked, only selected aspects need to be tracked. The decision concerning which data to track and what aspect of the data to track can be made at runtime to allow and provide dynamic data provenance for service execution. Its testability can be guaranteed by verifying the testability of the following factors:
 - a) Whether or not the selected data meet the customers' request.
 - b) Efficiency of the selection process.
 - c) Reusability of existing technology and commercial products for data collection.
 - d) Data collection strategies. Two mostly-used data collection strategies are actor-based and message-based.
2. **Data Provenance Classification Testability:** This refers to how well SOA-based software provides the testing support on the data provenance classification process. Some factors includes:
 - a) Design and selection of data classification for data provenance as not all data need to be tracked and evaluated.
 - b) The evaluation process to rank different classifications with respect to each service including atomic and composite services, and data routing in a SOA system.
 - c) Data storage, selection, mining, and tossing rules.
 - d) Data provenance attachment rules.
3. **Data Integrity Testability:** This refers to the testability of data integrity. Each service may have an integrity level, and data produced by high integrity service can go into low-integrity service, but not the other way around to protect data integrity according to the Biba Integrity Model [1] Data integrity may involve:
 - a) Evaluation of each service with respect to its integrity level;
 - b) Trace the data flow in an SOA system to determine if a high integrity data has been processed by a service of low integrity.
 - c) Evaluation of data from the original source to the current position to determine its integrity of data. .

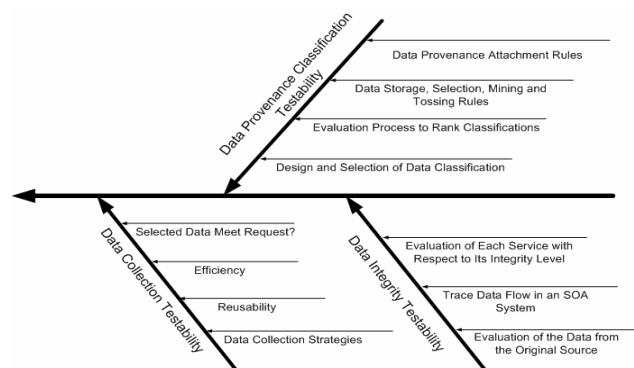


Figure 5 Different Factors of Data Provenance Testability

3.3 Service Integration Testability

Service integration is related to composite services and SOA applications with services. A composite service is formed by reusing existing atomic and composite services. Service integration testability refers to the degree to which a SOA-based application is designed to facilitate the creation of testing criteria on service integration as shown in Figure 6:

1. **Service Composition Testability:** This refers to the testability of service composition. This factor can be evaluated by checking the related workflow, service discovery and binding, service orchestration, provisioning, and whether the application has the right architecture.
2. **Runtime Service Re-composition Testability:** This refers to the testability of dynamic service re-composition. This factor can be evaluated by checking service monitoring, service configuration process including distributed service agreement and collaboration, configuration planning and decision making, as well as the configuration enforcement.
3. **Service Controller Testability:** This refers to the testability of the functions and behaviors of the service controller as shown in Figure 2. Note that a controller analyzes, designs, implements, and manages the SOA application using existing services. Thus, the testability of a controller may include the following factors:
 - a) Evaluate the overall controller process to ensure all the needed data are available for the controller to make proper decision;
 - b) Evaluate the (dynamic) modeling process to ensure that all the needed aspects are properly modeled at runtime.
 - c) Evaluate the code generation and analysis process in the controller to ensure that code produced is consistent with the model and the analysis results are correct;
 - d) Evaluate the service selection process in the controller to ensure that the services selected are proper;
 - e) Evaluate the service management process such as policy analysis, design, specification, simulation and enforcement. Evaluation of policy can be done by static analysis such as completeness and consistency checking, and/or by dynamic analysis such as simulation [11].
 - f) Evaluate the data collection and evaluation mechanisms to ensure that all the needed data are collected possibly including choreographic data, messages, and service state information.
4. **Adaptive Controller Testability:** It refers to the testability of the functions and behaviors of the adaptive controller. An adaptive controller can be implemented either by a fault-tolerant mechanism such as dynamic reconfiguration or a policy-based mechanism where the changes in the controller are governed by dynamic policies [11]. Testability of these two issues is related to the previous two items, but here they are applied to the controller in addition to the SOA application.

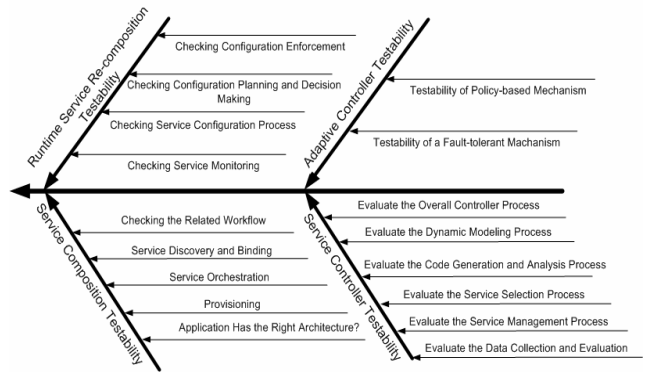


Figure 6 Factors of Service Integration Testability.

3.4 Service Collaboration Testability

This refers to how well SOA software is developed to facilitate the testing on service collaboration as shown in Figure 7.

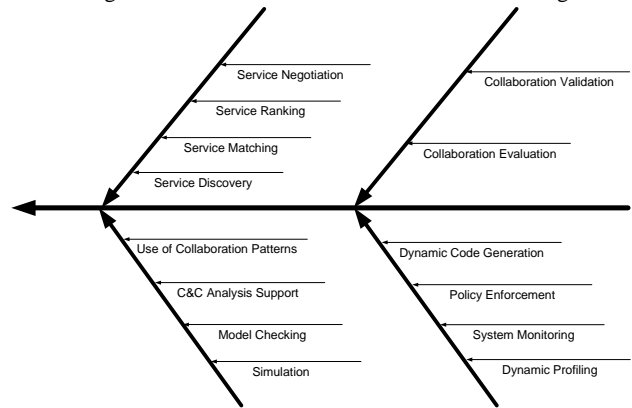


Figure 7 Factors of Service Collaboration Testability.

1. **Collaboration Preparation Testability:** This refers to how well the SOA software is developed to facilitate the testing on service collaboration preparation. In this phase, each service collaborating with other services need to be specified, generated, verified, and validated. The necessary capabilities include the support on use of collaboration patterns, completeness & consistency (C&C) analysis, model checking and simulation.
2. **Collaboration Establishment Testability:** This refers to the capabilities of the SOA software to facilitate the testing on service collaboration establishment. In this phase, the service consumer needs to submit its querying criteria to the service registry for service matching. After service discovery, service matching, and service ranking, the optimal service will be identified for further collaboration. Two services need to negotiate the collaboration agreement based on the collaboration profiles. Therefore collaboration establishment testability can be further refined to the testability support on service discovery, service matching, service ranking and service negotiation.
3. **Collaboration Execution Testability:** This refers to how well the SOA software is designed to facilitate the testing on service collaboration execution. In the collaboration execution phase, different services can be put together to

achieve a mission according to the application template. The collaboration can be reconfigured / recomposed at the system run-time on-demand. Then the new system code will be generated through dynamic code generation. Multiple dynamic analyses including model checking, simulation, testing can be performed after the system is changed. A variety of mechanisms including dynamic policy management, system monitoring, and dynamic service profiling also provide additional assurance during this phase. Therefore the collaboration execution testability can be further refined to the testability on dynamic code generation, policy enforcement, system monitoring and dynamic profiling.

4. **Collaboration Termination Testability:** This refers to the test support of the SOA software on collaboration termination. This is related to collaboration evaluation and validation.

4. Case Study

Similar with the component-based testability, the testability of SOA software can also be evaluated with the approach proposed in [6]. His approach is to score each evaluation criteria within 0 to 1, and calculates the final testability for the whole SOA software by compute the sum of all the weighted score for each evaluation criteria. The calculation formula is as following:

$$\text{Testability} = 0.5 * \sin(360/n) * \sum_{i=1}^n (\text{Score}_i * \text{Score}_{i+1})$$

A case study has been conducted using the above approach. The SOA software selected is a stock-trading software developed. The results are listed in the following table:

UMR	OMR	CMR	TMR	TSMR
0.23	0.34	0.5	0.04	0.02
SCR	DSPLR	SCMR	DPR	
0.64	0.07	0.11	0.05	

Note: UMR, OMR, CMR, TMR, and TSMR respectively represent the understandability, observability, controllability, traceability, and test support capability metric of its requirements verification[6], while SCR, DSPLR, SCMR and DPR respectively represent the service collaboration testability, test support on different service publishing levels, service control model testability and data provenance test support.

The testability is calculated as:

$$\text{Testability (Version 1)} = 0.5 * \sin(360/9) * ((0.23*0.34) + (0.34*0.5)+(0.5*0.04)+(0.04*0.02)+(0.02*0.64)+(0.64*0.07)+(0.07*0.11)+(0.11*0.05)+(0.05*0.23)) = 0.11$$

Then the software is updated to enhance the testability. The collected results of the revised software are listed in the following table:

UMR	OMR	CMR	TMR	TSMR
0.47	0.64	1.0	0.38	0.19
SCR	DSPLR	SCMR	DPR	
0.89	0.63	1.0	0.58	

The testability of the revised version is:

$$\text{Testability (Version 2)} = 0.5 * \sin(360/9) * ((0.47*0.64) + (0.64*1.0)+(1.0*0.38)+(0.38*0.19)+(0.19*0.89)+(0.89*0.63)+(0.63*1.0)+(1.0*0.58)+(0.58*0.47)) = 1.15$$

Thus, the revised version is much higher than the first version.

5. Conclusion

This paper proposes four testability evaluation criteria for SOA software. Different from the testability of traditional component-based software, the testability of SOA software needs to address the dynamism in SOA including service discovery, dynamic composition, dynamic re-composition, data provenance, and the dynamic behavior of the controller, and dynamic service collaboration, simulation and monitoring. Thus, the testability of SOA is more complicated and involved.

6. References

- [1] M. Bishop, *Computer Security: Art and Science*, 2002.
- [2] D. Chappell, *Enterprise Service Bus*, O’ Reilly Media, 2004.
- [3] DCIO, DOD OASD NII, “Net-Centric Checklist”, version 2.1.2, March 31, 2004.
- [4] R. S. Freedman, “Testability of Software Components”, *IEEE Transactions on Software Engineering*, 17(6), 553-563, June 1991.
- [5] J. Gao, J. Tsao, and Y. Wu, “Testing and Quality Assurance for Component-Based Software”, MA: Artech House, 2005.
- [6] J. Gao and M. C. Shih, “A Component Testability Model for Verification and Measurement”, *COMPSAC 2005*, July 26-28, 2005, pp. 211- 218.
- [7] P. Groth, M. Luck and L. Moreau, “A Protocol for Recording Provenance in Service-Oriented Grids”, *Proc. of 8th International Conference on Principles of Distributed Systems (OPODOS’04)*, 2004.
- [8] Tomcat: <http://www.eclipse.org/webtools/initial-contribution/IBM/evalGuides/WebServicesToolsEval.html>
- [9] W. T. Tsai, C. Fan, Y. Chen, R. Paul and J. Y. Chung, “Architecture Classification for SOA-based Applications”, *Proc. of 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC)*, pp. 295-302, April, 2006, Gyeongju, Korea.
- [10] W. T. Tsai, Q. Huang, B. Xiao, and Y. Chen, “Verification Framework for Dynamic Collaborative Services in Service-Oriented Architecture”, Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, 2006. .
- [11] W. T. Tsai, X. Liu, and Y. Chen "Distributed Policy Specification and Enforcement in Service-Oriented Business Systems," *IEEE International Conference on e-Business Engineering (ICEBE)*, Beijing, October 2005, pp. 10-17.
- [12] W. T. Tsai, W. Song, R. Paul, Z. Cao, and H. Huang, “Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing”, *COMPSAC 2004*, September 28-30, 2004, Hong Kong, pp.554-559.
- [13] J. M. Voas and K. W. Miller, “Software Testability: The New Verification”, *IEEE Software* 12(3), 17-28, May 1995.