On Testing and Evaluating Service-Oriented Software

W.T. Tsai, Xinyu Zhou, and Yinong Chen, Arizona State University
Xiaoying Bai, Tsinghua University, China

As service-oriented architecture matures and more Web services become available, developers must test an ever-increasing volume of services. A framework that defines and evaluates test-case potency based on coverage relationships can reduce testing effort while maintaining testing’s effectiveness.

Service-oriented architectures (SOAs) support Web services by facilitating sharing and communication among collaborators. The many standards and related techniques that address Web service security issues—WS-Security, WS-Trust, Extensible Access Control Markup Language (XACML), Security Assertion Markup Language (SAML), and so on—have produced a level of security that customers can trust. For example, people can bank, pay bills, and shop through the Internet with confidence.

Despite such progress in SOAs, service reliability, testing, and verification techniques aren’t mature enough to support dependable and trustworthy computing. As the “Related Work in Testing SOA Applications” sidebar describes, current Web service and SOA research focuses largely on protocols, functionality, transactions, ontology, composition, the semantic Web, and interoperability, while little research has explored service dependability and trustworthiness.

SOAs let application builders search and discover services from service brokers and use services from different service providers. Because service brokers aren’t responsible for the quality of the services they refer to or host, service trustworthiness isn’t guaranteed. Traditional dependability techniques—correctness proof, fault-tolerant computing, model checking, testing, evaluation, and so on—can improve individual services’ trustworthiness. However, implementers must redesign these techniques to handle the dynamic applications composed of services at runtime.

SOA applications can use traditional independent verification and validation approaches to enforce verification in each phase of the SOA development life cycle (modeling, assembly, deployment, management, and so on). In this case, however, all the code must be available and an independent team must test each service and all possible service combinations. This approach is expensive to implement because the number of services available and their combinations can be huge. In addition, service providers might be unwilling to share the code, which often runs on their own servers.

To address SOA dependability and trustworthiness issues, we propose an open framework that implements group testing to enhance test efficiency. The framework identifies and eliminates test cases with overlapping coverage. It also ranks newly added test cases and reranks existing test cases using updated coverage relationships and recent test results. A case study demonstrates the framework’s effectiveness in reducing testing effort while maintaining effectiveness.

SERVICE VERIFICATION

In Web 2.0, users can be active contributors. Therefore, an open verification framework for testing SOA software would benefit all parties. Such a framework...
Related Work in Testing SOA Applications

Several studies have addressed the problems of testing service-oriented architecture applications. Gerardo Canfora and Massimiliano Di Penta performed integration testing, functional testing, non-functional testing, and regression testing of SOAs. They also discussed the responsibilities and needs of SOA stakeholders, such as service providers and service consumers. Luciano Baresi and Elisabetta Di Nitto presented various SOA verification and testing techniques, including monitoring, modeling, and reliability analysis. Yinong Chen and Wei-Tek Tsai systematically studied SOA software development, including the life cycle and techniques applied in each phase.

Table A lists the types of testing SOA software must address.

Commercial tools for service testing are available. IBM’s Web Services Navigator, a testing and debugging tool, traces and visualizes the execution of services and thus helps programmers find bugs. The Web Services Interoperability Organization, an industry organization chartered to promote WS interoperability across platforms, released a service testing tool in March 2004. The tool has two components: WS-I monitor and WS-I analyzer. WS-I monitor is placed between the client and the Web service to log all messages, requests, and responses as they travel back and forth. The WS-I analyzer then analyzes these logs against the interoperability requirements.

References

<table>
<thead>
<tr>
<th>Type of testing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA protocol testing</td>
<td>Test functionality and performance of SOA protocols—for example, test SOAP, service publishing, and discovery; and evaluate protocol performance and the asynchronous nature of SOA operations</td>
</tr>
<tr>
<td>Unit testing and individual service evaluations</td>
<td>Test, evaluate, and rank services with source code (both white-box and black-box testing) and those without source code (blackbox testing only)</td>
</tr>
<tr>
<td>Integration testing and evaluation of application or composite services</td>
<td>Multilevel integration testing and evaluation with or without all the source code</td>
</tr>
<tr>
<td>Interoperability testing</td>
<td>Conformance testing to ensure compliance with SOA protocols and standards</td>
</tr>
<tr>
<td>Regression testing</td>
<td>Modification compliance, including unit testing and integration testing</td>
</tr>
<tr>
<td>Load/stress testing</td>
<td>Examine system behavior at different loads</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Track and evaluate runtime behavior</td>
</tr>
<tr>
<td>Security testing</td>
<td>Ensure and enforce security properties</td>
</tr>
<tr>
<td>Service broker testing</td>
<td>Ensure that service brokers, such as a UDDI or ebXML broker, perform registration and discovery properly, and enhance a service broker to act as an independent verification service to verify and rank services submitted for publication</td>
</tr>
<tr>
<td>SOA testing and evaluation framework</td>
<td>Develop integrated frameworks with processes and tools to test and evaluate services and applications</td>
</tr>
</tbody>
</table>

could provide an experimental testbed for regulators, service providers, service brokers, service consumers, and researchers. Using the framework,

- regulators could decide the metrics to establish service-quality criteria;
- service providers could evaluate their services before and after publishing them, and publish their evaluation data and mechanisms to increase consumer confidence;
- service brokers could provide value-added services by adding service verification as part of service discovery;
- service consumers could select services objectively, using quantitative data from the framework and could submit their test cases to the test-case repository; and
- researchers could develop and contribute techniques such as service performance and reliability assessments.

Thus, in this open framework, the services, test cases, test-case generation mechanisms, reliability models, and ranking algorithms are open to all parties, and can therefore be ranked as well.

In Web service testing scenarios, it’s necessary to rapidly test multiple versions of services with the same
specifications, as Figure 1 shows. Developers can specify atomic and composite services in the Business Process Execution Language for Web Services (BPEL4WS), the Semantic Web Ontology Language (OWL-S), or the Process Specification and Modeling Language for Service-Oriented Computing (PSML-S) with a Web Services Description Language (WSDL) interface. Based on the specifications, testers can use test-case generation techniques to generate test cases. The SOA verification techniques include testing, model checking, simulation, policy enforcement, and other mechanisms, such as completeness and consistency. We focus on testing—in particular, on how to test services efficiently.

GROUP TESTING SERVICES

The service-oriented design lets service developers and providers, based on published service specifications, register services and compose complex (composite) services from other services. As a result, for each service specification, alternative implementations might be available. How can we test a large number of services efficiently? The group-testing technique, which was originally developed for testing blood samples and later for software regression testing, is a promising approach.\textsuperscript{2,3}

We can apply group testing to both atomic and composite services. Figure 2 outlines the testing mechanism for a group of atomic services implementing the same specification. The group testing mechanism broadcasts the test cases $T_1, T_2, \ldots, T_m$ to all atomic services $S_1, S_2, \ldots, S_n$ under test. A voting service, which can automatically generate an oracle for each test case based on the majority principle, collects the outputs. It compares each service’s output with the oracle. It then dynamically logs the number of disagreements into each service profile and uses this number to evaluate and rank the service’s reliability and the test cases’ effectiveness and to establish each test case’s oracle.

Furthermore, we can calculate the oracle’s reliability. In the next testing phase, we apply the most effective test cases first, to reduce the testing time as a service that fails can be eliminated from further consideration. We use these results to automatically update the service and test case rankings and the oracles’ reliability.

Assume a composite service consisting of $n$ component services, with each component service having several equivalent services. Thus, there are $n$ sets of component services. Assume $Set_i$, has $m_i$ equivalent component services. If the composite service consists of one component service out of each set, $m_1 \cdot m_2 \cdot \ldots \cdot m_n$ possible composite services exist.

![Figure 1. Service verification scenarios. In the figure, four services follow the same specification but with different implementations. Test cases can be generated from the specification.](image)

![Figure 2. Group testing for atomic services. The total number of tests run is $m \cdot n$. However, the services can execute each test case in parallel to save test time.](image)
For example, if a composite consists of six atomic services, and each atomic service has 10 candidates, the total number of possible composite services is $10^6$. Using a group-testing mechanism can save test effort for composite services as well.

Figure 3 shows the group-testing process using a supply-chain example. The supply-chain service consists of six atomic services: retail, lodging, transportation, payment, ordering, and discovering. Group testing collects the outputs from all the retail services and establishes the oracle using the majority voting mechanism. Incorrect retail services can be identified and eliminated using the established oracle. Repeating this process for all atomic services can eliminate candidates from consideration. Integration testing can be conducted using the same approach but with the best candidates from each atomic service only. This process greatly reduces the number of combinations. If the remaining combination is large, group testing at this level can eliminate candidate combinations.

**TEST-CASE SELECTION AND RANKING**

Statistical analysis is often used in software testing. James Whittaker and Michael Thomason proposed a statistical software testing model using a Markov chain. They represent the statistical testing model as a usage Markov chain and a testing Markov chain. The usage Markov chain models the software's state diagram, and the testing Markov chain collects the testing profiles. In group testing, test-case selection and ranking are the key factors in enhancing testing efficiency. We use statistical data to rank test cases using two criteria:

- **potency**—the probability that a test case can detect a fault (for example, if a test case fails 30 services out of 100, its potency is 0.3); and
- coverage relationship among test cases—the amount of additional coverage one test case can bring given that we’ve applied other test cases.

**Motivation**

Figure 4 shows an example consisting of three test cases ($T_a$, $T_b$, $T_c$) and five services ($S_1$~$S_5$). In Figure 4, testing and service selection: (a) services are grouped by their output values and (b) services are grouped into two sets: correct services set, and incorrect services set. For some type of services (for example, a forecasting service), the correct output might be unknown. In this case, the majority voting mechanism can establish the oracle, evaluate the services, and then group the services according to their output values.
4a, for each test case, the gray circles denote a correct output and the white circles denote various incorrect outputs. After applying $T_a^1$ to the five services, $S_1$, $S_2$, and $S_3$ generate the same output, while $S_4$ and $S_5$ generate different outputs. If the correct output is already given or is identifiable through the established oracle, we can group the services into two sets by their outputs—correct services and incorrect services—as Figure 4b shows.

If all other criteria are the same, group testing always applies the test case with the highest potency first, because statistically, the more potent the test case is, the more likely it will detect faults in the next services. Applying potent cases first eliminates incorrect services early and thus reduces test efforts.

**Coverage relationship**

If we consider only potency, the rank of the test cases in Figure 4 is $T_a^1 T_b^1 T_c^1$. However, this ranking doesn’t consider the coverage relationship among the tests. If $T_a^1$ and $T_b^1$ cover the same aspects (for example, test the same path), running one of them might be sufficient. Thus, the correct ranking should be $T_a^1 T_b^1 T_c^1$, or $T_a^1 T_b^1 T_c^1$. We use a coverage relationship model (CRM) as an additional test-case selection criterion. Because we’re concerned with two output sets, we propose the following simplified CRM (S-CRM) to reduce the computational complexity of constructing the full model:

- In S-CRM, each test case has two output sets: the correct output set, containing the services that generate the correct output; and the incorrect set, containing the services that generate incorrect output.
- We use $\Delta(T_1, T_2)$ to denote the relation that test case $T_1$ can cover what test case $T_2$ can cover, or simply say $T_1$ covers $T_2$. We use $\Delta(t_1, t_2)$ to denote that $t_1$ doesn’t cover $t_2$.
- The coverage probability ($P$) parameter—that is, the probability that one service exists in the correct sets of both $T_1$ and $T_2$—quantifies the coverage relationship between any two test cases $T_1$ and $T_2$.

Figure 4b shows the S-CRM for the example in Figure 4a. The intersection of the correct sets of $T_a^1$ and $T_b^1$ is $S_1$, and the size of the correct set of $T_a^1$ is 3. Therefore, the coverage probability from $T_a^1$ to $T_b^1$ is 0.33. Similarly, the coverage probability from $T_b^1$ to $T_a^1$ is 1 because all services in the correct set of $T_b^1$ are also in $T_a^1$, so $\Delta(T_b^1, T_a^1)$. Therefore, the final rank is $T_a^1 T_b^1$ after we eliminate $T_c^1$.

Using the $P$ from $T_a^1$ to $T_b^1$, we perform the following actions:

- If $P$ is a small value (for example, $P < 0.05$), $T_a^1$ doesn’t cover $T_b^1$, so we calculate $P$ from $T_b^1$ to $T_a^1$ to see if $T_b^1$ covers $T_a^1$.
- If $P$ is a medium value, $T_b^1$ partially covers $T_a^1$, so we calculate $P$ from $T_a^1$ to $T_b^1$ to see if $T_a^1$ covers $T_b^1$.
- If $P$ is a large value (for example, $P > 0.95$), $T_a^1$ covers $T_b^1$, so we eliminate $T_b^1$ and use $T_a^1$ only.

Thus, ranking by potency only subjects the software to being penalized by the same faults multiple times. We must therefore rank test cases by both potency and test case coverage relationships.

One way to identify test case coverage relationships is to analyze how test cases are derived. Specifically, if two test cases aim to evaluate the same software aspect (for example, control flow) on the same software segment, they have similar coverage.

Instead of evaluating how test cases are derived, CRM evaluates the test case coverage by learning the previous results. If $T_a^1$ and $T_b^1$ fail the same set of services, their coverage is highly correlated. If they fail a completely different set of services, they might have little coverage overlap.

Because CRM is based totally on test results, two test cases derived from two different testing techniques and addressing two different code segments might have identical coverage in the CRM. This doesn’t imply that the two test cases address the same code segment or aspects. Instead, it means that they occurred together accidentally. As the CRM collects more data, test cases developed using different techniques will eventually detect different faults. Statistically, test cases derived from different testing techniques should have different test coverage.

Although analyzing how test cases are derived might yield accurate results, assigning coverage relationships by examining test results has distinct advantages. It allows automating the entire process to eliminate human errors, and it isn’t necessary to track and record the derivation of test cases. An identical test-derivation process applied to the same code segments might still produce different test cases with different results. For example, one test case might detect an incorrect action within a path, whereas another might detect a different fault in a decision in the same path. Thus, test derivation might still provide incorrect guidance with respect to the coverage relationship. However, the CRM approach is based completely on testing results collected.

**Adaptive test-case ranking**

Group testing can implement the test-case selection and ranking mechanism using CRM. We can use CRM to
select and rank test cases and identify and remove test cases with similar coverage. CRM can also rank test cases adaptively because it can rerank the test case whenever new test cases are added. In this way, CRM can continuously rank test cases while we perform the test. It selects only the most potent test cases with the least coverage overlap with previously applied test cases for execution. In this manner, the test process becomes adaptive as it learns from the previous test results. For a new test case \( t \), CRM calculates \( t \)'s potency and establishes its result sets by testing the services. For all test cases that are more potent than \( t \), CRM calculates the coverage relationship to check if these test cases cover \( t \). For all test cases that are less potent than \( t \), CRM calculates the coverage relationship to check if \( t \) covers these test cases.

**CASE STUDY**

Although CRM saves testing efforts by eliminating test cases, it retains its effectiveness. To demonstrate this effectiveness, we developed 60 Web services independently and developed 32 test cases based on the specification for experimentation.

Figure 5 shows the coverage for all 32 test cases. Rows represent services; columns represent test cases, listed according to their potency; and cells represent the test results when we apply the cases to the services. White cells denote correct outputs; black cells denote incorrect outputs.

CRM retains eight test cases (\( T_{1-4}, T_7, T_{10}, T_{16}, \) and \( T_{18} \), denoted in bold). It identifies the other 24 test cases (\( T_{5, 6, 8, 9, T_{11-15}, T_{17}, T_{19-32} } \)) as covered. As Figure 5 shows, \( T_{18} \) can detect all the faults that \( T_{17} \) and \( T_{19-32} \) detected. Similarly, \( T_1 \) covers \( T_5 \). Therefore, CRM eliminates these 24 test cases.

For each Web service, the “CRM” column shows the number of faults detected by all 32 test cases, whereas the “Total” column shows the number of faults detected by the eight test cases retained by the CRM. For example, 16 of the 32 test cases failed Web service 13, while seven of the eight test cases failed the same service. A comparison of the two columns shows that if the 32-test-case set can detect certain faults in a given service, the reduced test-case set can detect the same faults. Therefore, in this experiment, the CRM doesn’t lose any effectiveness, despite the smaller number of test cases.

Table 1 summarizes the test results. For the 32-test-case set, CRM detects 262 failures in all 60 services. The size of the test-case set the CRM selected is 8, reducing the test effort to 8/32 = 25 percent.

Table 1. Effectiveness and efforts analysis.

<table>
<thead>
<tr>
<th>Test cases</th>
<th>Effectiveness (%)</th>
<th>Test effort (%)</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total: 32</td>
<td>100</td>
<td>100</td>
<td>262</td>
</tr>
<tr>
<td>S-CRM: 8</td>
<td>100</td>
<td>25</td>
<td>123</td>
</tr>
</tbody>
</table>

In addition to testing service-oriented software, our CRM technique can be applied to other domains where multiple versions of applications are available. In N-version programming, for example, multiple versions of a system implement the same function. Regression testing, in which multiple versions of software are developed, is another potential application area. A third area is standards-based testing, which involves systems that implement the functionality and interfaces specified in a published standard. For example, vendors can develop their own software to implement OASIS and W3C standards. Standards-making organizations, such as OASIS and the W3C, often need to publish test cases to ensure that the software produced by vendors meets the standards requirements.

Traditional software testing often focuses on code or design coverage. Our approach uses previous testing results and an adaptive testing process. Because service-oriented computing focuses on composition, possibly even dynamic composition at runtime, the need for adaptive testing will become important.

**Acknowledgment**

The authors thank Raymond A. Paul for his contributions to this article.
References


Wei-Tek Tsai is a professor in the Computer Science and Engineering Department and director of the Software Research Laboratory at Arizona State University. His research interests include software engineering and Web services testing and verification. Tsai received a PhD in computer science from the University of California, Berkeley. Contact him at wtsai@asu.edu.

Xinyu Zhou is a software engineer at VMware. His research interests include Web services design, testing, and implementation. Zhou received a PhD in computer science from Arizona State University. Contact him at xinyu.zhou@asu.edu.

Yinong Chen is a lecturer in the Computer Science and Engineering Department at Arizona State University. His research interests include service-oriented software development, fault-tolerant computing, reliability modeling, and embedded systems. Chen received a PhD in computer science from the University of Karlsruhe, Germany. Contact him at yinong@asu.edu.

Xiaoying Bai is an associate professor in the Department of Computer Science and Technology at Tsinghua University, China. Her research interests include software engineering, software testing, and service-oriented architectures. She received a PhD in computer science from Arizona State University. Contact her at baixy@tsinghua.edu.cn.