

PAPER

# Scenario-Based Web Service Testing with Distributed Agents

Wei-Tek TSAI<sup>†</sup>, Ray PAUL<sup>††</sup>,  
Lian YU<sup>†</sup>, Akihiro SAIMI<sup>†††</sup>, and Zhibin CAO<sup>†</sup>, *Nonmembers*

**SUMMARY** Web Services (WS) have received significant attention recently. Delivering Quality of Service (QoS) on the Internet is a critical and significant challenge for WS community. This article proposes a Web Services Testing Framework (WSTF) for WS participates to perform WS testing. WSTF provides three main distributed components: test master, test agents and test monitor. Test master manages scenarios and generates test scripts. It initiates WS testing by sending test scripts to test agents. Test agents dynamically bind and invoke the WS. Test monitors capture synchronous/asynchronous messages sent and received, attach timestamp, and trace state change information. The benefit to use WSTF is that the user only needs to specify system scenarios based on the system requirements without needing to write test code. To validate the proposed approach, this paper used the framework to test a supply-chain system implemented using WS.

**key words:** *Web Services, scenarios, QoS, object-oriented test frameworks*

## 1. Introduction

Web Services (WS) is an emerging distributed computing architecture and receiving significant attention recently [3], [18], [25], [34], [47], [52], [53]. It uses standard Internet protocols such as WSDL (Web Services Description Language) [50], SOAP (Simple Object Access Protocol) [33], and UDDI (Universal, Description, Discovery and Integration) [48]. Fig.1 shows the general WS architecture:

1. The service provider registers its services with a registry maintained by a service broker. The service broker represents a set of software interfaces (a registry service) for published WS. WSDL describes the request's format - its parameters and data types.
2. The client makes a call to the broker's UDDI registry, seeking a specific service.
3. Once the right service is found, the broker returns the service's information to the client.
4. The client invokes the service by making a SOAP call to the service provider.
5. Finally, the provider delivers the application results back to the client. The transaction is now complete.

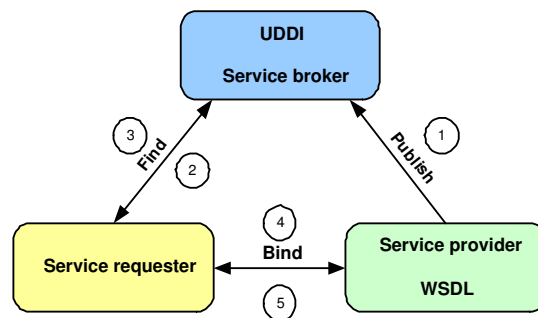


Fig. 1 General Web Services architecture.

With the wide adoption by major WS players, WS communities from the financial services, high-tech, media and entertainment are being formed. For these communities to be successful, it is important to maintain QoS for all the parties including service providers, clients and service brokers. Testing is a common QoS assurance technique; however testing WS is difficult due to the following reasons:

- WS have a loosely coupled architecture but still demand high assurance;
- WS have a dynamic behavior including discovery and binding with multi-parties with middleware and other WS at runtime;
- WS can be invoked by unknown parties with unpredictable requests by other WS or clients;
- Computing WS may involve concurrent threads and object sharing, and testing these features is often difficult; and
- Testing is difficult because it involves all three parties involved: clients, brokers, and providers.

**Clients:** A client likely has no access to the source code of the WS that will be called, and the client may suspect the service provider may provide a faulty product, and even suspect the broker involved may provide incorrect information about the WS. The client may want to use the WS supplied by reputable services provider acquired via reputable brokers only. Furthermore, the client may authenticate the brokers and service providers each time it acquires services to get the assurance that the WS called are authentic. Before the client uses the WS supplied to develop an application, the client should actually test-drive the WS first, pos-

<sup>†</sup>Department of Computer Science and Engineering, Arizona State University

<sup>††</sup>Department of Defense

<sup>†††</sup>Hitachi Software Engineering, Tokyo, Japan

sibly by using the test scripts supplied by the broker or supplier. Unlike the WS, the test scripts supplied by the service provider or broker are likely to be open so to give the client assurance the test scripts are reliable.

**Brokers:** A broker needs to provide quality WS to its clients, but this requires quality assurance mechanisms to deal with both its clients and providers. Like a client, a broker may not have access to the WS source code. In addition to provide a list of matched WS to clients and register WS from service providers, the broker may use the following mechanisms for QoS:

- It needs to ensure that it will receive WS from reputable service providers only. The broker may keep a list of service provider and the history of WS quality received from these providers. This is similar to the evaluation of sellers by buyers and vice versa at the www.Ebay.com, and the broker may even make the evaluation public so that potential buyers or sellers can see the evaluation before transaction.
- It also needs to ensure the WS received from reputable service providers are of high quality. Potentially, a reputable service provider may still deliver faulty or even malicious WS from time to time, and thus the broker must check individual WS checked in. For example, the broker may need to verify the WS to meet certain minimum standards such as consistency between the WS and the stated WSDL file. The broker may also develop its own test scripts to test the WS supplied or use the test scripts supplied by service providers. For example, the broker may publish a list of WSDL and associated test scripts for each WSDL file, and a service provider that wish to supply a WS that fits with a particular WSDL must pass the test conducted by the published test scripts [45]. In this way, the broker also provides certain quality assurance for its clients. The client may also use the published test script to test drive the WS before application.
- Furthermore, the broker may keep a history of bugs related to the WS registered. This is similar to the Consumer Report [7] where they keep a history of defects in various products to inform clients.
- The broker may keep and publish the history of faulty WS supplied so that a client may choose between different brokers for searching WS. This is one way for a reputable broker to be distinguished from its peer brokers.

**Service providers:** Unlike clients and brokers, a service provider does have access to the WS source code. Thus its primary job is to provide the evidence of quality of the WS supplied to both clients and brokers, and maintain the QoS each time a WS is called at runtime. A service provider may perform the following actions

to provide such evidence:

- Before registering a WS to a broker, the service provider checks that the WS is consistent with the WSDL.
- If the broker has a list of published test scripts available, the service provider exercises those test scripts on the WS before registering.
- The service provider may supply additional open-source test scripts to both clients and brokers so that they can have added assurance.

In summary, testing does play an important role in QoS of WS, and this paper presents a scenario-based WSTF using distributed agents to perform WS testing. This framework can be used by all the parties:

- **Clients:** The client can use the WTSF to test the application as well as those WS called at runtime. A client can use WSTF to develop code to run and control the application at runtime. The client may also use the WSTP to test the WS before integrating them into the application.
- **Brokers:** A broker may want to verify the WS when they are registered to ensure the quality of WS checked in the registry, and the WSTF can be used to perform the WS testing. The broker may actually supply the test code developed to the existing and potential clients to give them assurance that WS provided are of high quality.
- **Service providers:** Before publishing WS, a provider can use the WSTF to test and evaluate the interoperability of WS.

This WSTF has following features:

- **Enhanced WSDL:** Current WSDL standards do not contain sufficient information for an application engineer to test WS. This paper adds four kinds of extension of WSDL for testing: input-output dependency, invocation sequence, hierarchical functional description and concurrent sequence specifications. This is discussed in Section 3.
- **Scenario-based Testing:** In most cases, the source code for WS will not be available for WS clients and brokers. For them, it is possible to perform black-box testing only using the published specifications such as the WSDL files. Scenario-based testing is a specification-based testing and proved useful for integration testing and interoperability testing [38]–[40],[42]. WS testing focuses on integration and interoperability testing, Section 4 presents scenario specification techniques.
- **Automated test script generation:** WSTF provides a test master, which generates test scripts automatically based on scenario specification and by extracting information from enhanced WSDL, and stores them into database.
- **Automated distributed test execution:** Based

on the user's request, test master selects a set of test scripts from database, and initiates running test by sending test scripts to test agent(s) of WSTF, which dynamically finds and binds and invokes services at runtime according to the test scripts. In this way, a tester user does not need to write test code, which saves effort and time significantly [30]. In addition to test master and test agents, the WSTF supports a test monitor for each WS provider. Monitors capture synchronous/asynchronous messages sent and received, attach timestamp, trace state change information, and send the relevant information to test master and test agents for verification. This is discussed in Sections 5.

This paper is organized in the following way: Section 2 presents some related work. Section 3 extends WSDL to include dependency and sequence information to facilitate WS testing. Section 4 shows the scenario specification and analysis techniques. Section 5 presents the WSTF architecture. Section 6 demonstrates the techniques using a supply-chain system implemented using WS. Finally, Section 7 concludes this paper.

## 2. Related Research

Scenario is an effective technique for capturing system behavior. Since early 1990s, considerable researches have been performed on use-oriented techniques [13], [19] for software design and development both in industries and in academia. Jacobson introduces the concepts of use cases, which capture the essential functional requirements by identifying the interactions between the system and the external entities. Use cases are specified in UML using use case diagram and specification template. However, often such specification is not detailed enough to allow data and control dependency analysis, or to generate test cases. For example, Jacobson recommended that for a large system, the number of use cases should be around 70 to 80, but scenarios at such a high-level abstraction are not detailed enough for testing. Currently, other than UML, a variety of representation techniques are also available to represent scenarios, e.g., UCM (Use Case Maps) [1], and MSC (Message Sequence Charts) [15]. UCM and MSC are developed for real-time systems, specifically MSC is closely related to a state model, and UCM can be mapped to a system structure for analysis. UCM and MSC do not address dependence analysis, traceability, ripple effect analysis, or test case generation. Furthermore, both MSC and UCM are not widely used and tool support is not adequate.

Recently, Ryser and Glinz in Europe developed SCENT - a scenario-based approach for testing, and addressed dependency analysis and scenario generation

[32]. NASA Goddard Space Flight Center develops their systems using scenarios [27], and they use pre-condition, event flow, and post-condition to describe scenarios. The DOORS project [21] is also similar. They specify scenarios by specifying the information flow between actors, and specifying actions performed by actors.

From scenario specification, one can generate test scenarios/cases/scripts [38]–[40], [42], [43]. Cohen proposed an approach to testing that uses combinatorial designs to generate tests that cover the pair-wise, triple, or n-way combinations of a system's test parameters, which determine the system's test scenarios [6]. There are many systems where troublesome faults are caused by the interaction of a few test parameters. Cohen pointed that a test plan should ideally cover those interactions and the number of tests required to cover all n-way parameter combinations grows logarithmically in the number of parameters. The approach allows testers to refine a model by adding more test parameters without causing explosion of the number of tests. AETG (Automatic Efficient Test Generator) was developed using the combinatorial design theory for unit, system and interoperability testing. Cohen's research is useful for creating effective and efficient test plan qualitatively. In contrast, this paper aims at creating adaptive test infrastructure to perform WS testing by using a variety of techniques stated in the following sections.

Testing distributed systems has been studied for decades focusing on different issues. Ulrich [49] suggested two test architectures for testing distributed systems: a global tester that has total control over the distributed system, and a distributed tester comprising several concurrent tester components. Coordination between tester components is done by redundant observation of internal interactions, or by using synchronization events between tester components. Mathur [26] and Hoffmann [12] addressed the problems that arise when testing CORBA-based distributed systems. While some papers discussed the theoretical aspects of testing component-based distributed systems, most focus on unit testing of single components only [22], [55], [57].

Recently, testing WS also receives attentions [41], [44], [45]. Mani and Nagarajan raised important quality criteria for Web including security, availability and performance [20]. Bloomberg outlines three-stage for WS testing [58].

OO test frameworks have been initially proposed by Firesmith [10], McGregor [24], and Poonawala [31] independently. The frameworks proposed by Firesmith and McGregor often follow the OO structure such as inheritance graph to develop the test framework, Poonawala proposed a techniques to organize test scripts so that they can be reused to test safety-critical applications even though the System Under Test (SUT) does not have the OO structure, and this approach has been

used at an industrial site. Recently, several operational test frameworks have been developed to test OO programs in Java, e.g., JUnit [14], HttpUnit [8], Cactus [5], and Mock Objects [23], and these frameworks use OO design patterns [9], [11], [36] extensive, and is often based on specific programming languages such as Java or specific applications such as Internet publishing. Most of these test frameworks address unit testing, and are platform and protocol dependent. We have developed several OO test frameworks to test real-time embedded systems [39], [40] and they address integration testing. One common feature associated with all the existing testing frameworks is that they need the tester to understand/master the framework techniques before they can develop the test script. Unfortunately, it may take a long time for an inexperienced to master these frameworks before they can become proficient in testing applications using these frameworks. The proposed WSTF does not suffer from this problem.

### 3. Extending WSDL

WS standard protocols such as SOAP, WSDL and UDDI focus interoperability issues, but they have not addressed the QoS in detail yet. Specifically, the current WSDL standard does not contain sufficient information for an application engineer to test WS. Currently WSDL contains the following information: the number of inputs and outputs, the variable type of each input and output, the order the inputs and outputs, and how the concerned WS be invoked. These are certainly useful for interoperability, but they are not sufficient for verification. Specifically, information such as input/output dependence among WS, and the WS invocation sequence cannot be obtained from the WSDL file. These two items are needed to perform black-box testing and regression testing. To address this problem, this paper proposes to add the following items into WSDL:

- Input-output Dependency,
- Invocation Sequences,
- Hierarchical Functional Description, and
- Concurrent Sequence Specifications.

#### Input-Output Dependency

In numerous testing strategies such as regression testing [29], data flow testing and Ripple Effect Analysis (REA) [38], it is necessary to know the input and output dependency. This dependency information is not available in WSDL. Fortunately, it is possible to add input/output dependency into the WSDL schema:

- *WSInputOutputDependenceType*, which includes at least one element of type *WSIOPairType*.
- *WSIOPairType* is a complex type with two sub-elements, Input and Output, of *WSIOModeType*.

If there is a dependency between an input variable and an output variable, an engineer can use the above scheme to publish the dependency. The schema is illustrated as follows:

```
<s:complexType
  name="WSInputOutputDependenceType">
  <s:complexContent>
    <s:sequence>
      <s:element minOccurs="1"
        maxOccurs="unbounded" name="WSIOPair"
        type="WSIOPairType" />
      <s:any />
    </s:sequence>
  </s:complexContent>
</s:complexType>
<s:complexType name="WSIOPairType">
  <s:sequence>
    <s:attribute name="Mode"
      type="WSIOModeType" use="required" />
    <s:element minOccurs="1" maxOccurs="1"
      name="Input" type="s:string" />
    <s:element minOccurs="1" maxOccurs="1"
      name="Output" type="s:string" />
  </s:sequence>
</s:complexType>
<s:simpleType name="WSIOModeType">
  <s:restriction base="s:string">
    <s:enumeration value="Data" />
    <s:enumeration value="Control" />
  </s:restriction>
</s:simpleType>
```

For example, for a banking application, there is a dependency between input variable *AccountID* and output variable *Balance*, and this information can be specified as follows:

```
<WSInputOutputDependence
  xmlns="http://...org/">
  <WSIOPair Mode="Control">
    <Input>AccountID</Input>
    <Output>DepositResult</Output>
  </WSIOPair>
  <WSIOPair Mode="Control">
    <Input>AccountID</Input>
    <Output>Balance</Output>
  </WSIOPair>
  <WSIOPair Mode="Control">
    <Input>Password</Input>
    <Output>DepositResult</Output>
  </WSIOPair>
  <WSIOPair Mode="Control">
    <Input>AccountType</Input>
    <Output>DepositResult</Output>
  </WSIOPair>
  <WSIOPair Mode="Control">
    <Input>AccountType</Input>
```

```

    <Output>Balance</Output>
</WSIOPair>
<WSIOPair Mode="Data">
    <Input>DepositAmount</Input>
    <Output>Balance</Output>
</WSIOPair>
<WSIOPair Mode="Data">
    <Input>DepositAmount</Input>
    <Output>DepositAmount</Output>
</WSIOPair>
</WSInputOutputDependence>

```

The input/output dependency information can be obtained by a compiler technique such as program slicing or dependency analysis [46]. For example, one can use a parser to obtain the dependency information between the input variables and output variables within WS. Once the input/output dependency information is available, if a client of the WS changes the value of an input, it is easy to know which output variables will be or will not be affected by simply looking up the associated WSDL file. If an output variable is not affected, it is not necessary to perform the related regression testing [29] or REA, and potentially saving time and effort. This information may also eliminate unnecessary test cases.

### Invocation Sequences

A WS can call other WS to perform its tasks, and thus it is important to trace and state those calling relationship. MtSS (Method Sequence Specification) [16], [17] was developed to specify those calling relationships for OO programs [16] while IMOZU diagrams for procedural programs [28]. Similar information can be captured in WSDL. To do this, we define a WS reference type, which has WS name and the link to its WSDL document: *WSInvocationDependenceType* to represent callers and callees. There are two sub-elements in it, *WSICallers* and *WSICallees* of *WSICallersType* and *WSICalleesType* respectively. For both the *WSICallersType* and *WSICalleesType*, there is a link to their WSDL files for reference. By tracing this information among participating WS, it is possible to generate the complete calling sequence, and the calling sequence is useful in path testing and data flow testing. The following XML code illustrates the schema.

```

<s:complexType
    name="WSInvocationDependenceType">
    <s:sequence>
        <s:element minOccurs="0" maxOccurs="1"
            name="WSICallers"
            type="WSICallersType" />
        <s:element minOccurs="0" maxOccurs="1"
            name="WSICallees"
            type="WSICalleesType" />
    <s:any />

```

```

</s:sequence>
</s:complexType>
<s:complexType name="WSICallersType">
    <s:sequence>
        <s:element minOccurs="1"
            maxOccurs="unbounded"
            name="WSICallerRef"
            type="WSReferenceType" />
    </s:sequence>
</s:complexType>
<s:complexType name="WSICalleesType">
    <s:sequence>
        <s:element minOccurs="1"
            maxOccurs="unbounded"
            name="WSICalleeRef"
            type="WSReferenceType" />
    </s:sequence>
</s:complexType>
<s:complexType name="WSReferenceType">
    <s:sequence>
        <s:attribute name="Mode"
            type="WSFModeType" use="optional" />
        <s:element minOccurs="1" maxOccurs="1"
            name="Name" type="s:string" />
        <s:element minOccurs="1" maxOccurs="1"
            name="RefLink" type="s:string" />
    </s:sequence>
</s:complexType>

```

In a banking example, WS *Deposit* needs to authenticate the customer first and then get the existing balance to compute the new value. So it will call WS *AccountAuthentication* and *GetBalance*. On the other hand, *Deposit* itself will be used by WS *Transfer* to fulfill the operation. So WS *Transfer* is one of the callers here. The related WSDL for *Deposit* is illustrated below:

```

<WSInvocationDependence
    xmlns="http://...org/">
    <WSICallers>
        <WSICallerRef>
            <Name>Transfer</Name>
            <RefLink>
                http://...com/Transfer.wsdl
            </RefLink>
        </WSICallerRef>
    </WSICallers>
    <WSICallees>
        <WSICalleeRef>
            <Name>AccountAuthentication</Name>
            <RefLink>
                http://...com/AccountAuthentication.wsdl
            </RefLink>
        </WSICalleeRef>
        <WSICalleeRef>
            <Name>GetBalance</Name>
            <RefLink>

```

```

    http://...com/GetBalance.wsdl
  </RefLink>
</WSICalleeRef>
</WSICallees>
</WSInvocationDependence>

```

### Hierarchical Functional Description

In addition to providing structural information such as dependency and calling sequences, it is possible to incorporate functional descriptions into WSDL by adding a new type *WSFunctionalDescriptionType*. Furthermore functional descriptions can be organized in a hierarchical manner and can be embedded in WSDL. Once organized in a hierarchical manner, a functional description can be formed by sub-description. The WSDL can be extended to include two sub-elements, *WSFParents* and *WSFChildren*, which are of *WSFParentType* and *WSFChildrenType*. For these two types, there is a link to their corresponding WSDL files to reflect the hierarchical structure. The *WSFParents* is the main description, while the *WSFChildren* is the sub-description. By using these functional descriptions, it is possible to perform functional dependency analysis [38], and the results can be useful for functional testing and regression testing.

```

<s:complexType
  name="WSFunctionalDescriptionType">
  <s:sequence>
    <s:attribute name="Mode"
      type="WSFModeType" use="required" />
    <s:element minOccurs="1" maxOccurs="1"
      name="WSFParents"
      type="WSFParentsType" />
    <s:element minOccurs="1"
      maxOccurs="unbounded"
      name="WSFChildren"
      type="WSFChildrenType" />
  <s:any />
  </s:sequence>
</s:complexType>
<s:complexType name="WSFParentsType">
  <s:sequence>
    <s:element minOccurs="1" maxOccurs="1"
      name="WSFParentDes" type="s:string" />
    <s:element minOccurs="1" maxOccurs="1"
      name="WSFParentRef"
      type="WSReferenceType" />
  </s:sequence>
</s:complexType>
<s:complexType name="WSFChildrenType">
  <s:sequence>
    <s:element minOccurs="1" maxOccurs="1"
      name="WSFChildDes" type="s:string" />
    <s:element minOccurs="1" maxOccurs="1"
      name="WSFChildRef"

```

```

    type="WSReferenceType" />
  </s:sequence>
</s:complexType>
<s:complexType name="WSReferenceType">
  <s:sequence>
    <s:attribute name="Mode"
      type="WSFModeType" use="optional" />
    <s:element minOccurs="1" maxOccurs="1"
      name="Name" type="s:string" />
    <s:element minOccurs="1" maxOccurs="1"
      name="RefLink" type="s:string" />
  </s:sequence>
</s:complexType>
<s:simpleType name="WSFModeType">
  <s:restriction base="s:string">
    <s:enumeration value="Atomic" />
    <s:enumeration value="Complex" />
  </s:restriction>
</s:simpleType>

```

In a banking example, *WS Deposit* and *WS CheckBalance* are a part of *WS Transfer*.

```

<WSFunctionalDescription Mode="Complex"
  xmlns="http://...org/">
  <WSFParents>
    <WSFParentDes>
      This routine performs transferring from one
      account to another account.
    </WSFParentDes>
    <WSFParentRef Mode="Complex">
      <Name>Transfer</Name>
      <RefLink>
        http://...com/Transfer.wsdl
      </RefLink>
    </WSFParentRef>
  </WSFParents>
  <WSFChildren>
    <WSFChildDes>
      This routine deposits money into an account
      after obtaining the account balance.
    </WSFChildDes>
    <WSFChildRef Mode="Atomic">
      <Name>Deposit</Name>
      <RefLink>
        http://...com/Deposit.wsdl
      </RefLink>
    </WSFChildRef>
  </WSFChildren>
  <WSFChildren>
    <WSFChildDes>
      This routine checks balance.
    </WSFChildDes>
    <WSFChildRef Mode="Atomic">
      <Name>CheckBalance</Name>
      <RefLink>
        http://...com/CheckBalance.wsdl
      </RefLink>

```

```

    </WSFChildRef>
  </WSFChildren>
</WSFunctionalDescription>

```

By linking all the functional description by the related WS, we can now obtain the functional description for WS *Transfer* because it involves WS *CheckBalance* and WS *Deposit*.

- This routine performs transferring from one account to another account.
  - This routine checks account balance from the database.
  - This routine deposits into an account after getting the account balance.

By using the hierarchical functional description, a tester can perform functional testing. It is possible to further formalize the functional description to include information such as actions, conditions, actors, and events so that a more formalized analysis can be performed on these functional descriptions. If the formalized information available, it is possible to carry out more powerful analysis such as completeness checking, and data and control functional dependency analysis [38].

### Concurrent Sequence Specifications

WS may run in a multi-threaded environment with concurrent execution. Modeling synchronization among threads is important for the specification, design, and testing of concurrent WS applications. This paper applies SMtSS (Synchronized Method Sequence Specification) [50] to specify sequence constraints for concurrent WS threads. SMtSS is an extension to MtSS, and is used to specify thread cooperation such as synchronization and mutual exclusion. SMtSS can specify whether two methods within a shared object can execute concurrently or only in a mutual exclusion mode.

The following *Stock* example provides a simplified *Stock* object in a stock market application which is shared by three types of persons: manager, clerk, and customer, which are implemented as threaded objects *Manager*, *Clerk* (CK), and *Customer* (CR), respectively. Then the SMtSS for the *Stock* object is:

```

SMtSS = constructor . Entity* . destructor
Entity = BeforeOpen* .
  OpenMarket . <{M}->{CK,CR}>. AfterOpen* .
  CloseMarket . <{M}->{CK,CR}>
BeforeOpen = CheckOpen | CheckStockInfo
AfterOpen = CheckOpen | CheckStockInfo
  ((SubmitAsk | SubmitBid) .
  <{CR}->{CK}> . Transaction)

```

Where " $\langle\{S\}\rightarrow\{R\}\rangle$ " notation represents synchronization, " $S$ " represents a set of senders; " $R$ "

represents a set of receivers; and " $\rightarrow$ " represents the synchronization from senders to receivers. For example, " $((\text{SubmitAsk} | \text{SubmitBid}) . \langle\text{CR}\rightarrow\text{CK}\rangle . \text{Transaction})$ " means that after a customer submits an Ask or Bid, the customer needs to re-confirm the order before the transaction can be carried out. Once the concurrent behaviors are specified, it is possible to generate test cases to test the associated threaded applications [51]. The following WSDL specifies the above SMtSS constraints.

```

<smtss name="StockExample">
  <thread>
    <thread-name>M</thread-name>
    <thread-class>Manager</thread-class>
  </thread>
  <thread>
    <thread-name>CK</thread-name>
    <thread-class>Clerk</thread-class>
  </thread>
  <thread>
    <thread-name>CR</thread-name>
    <thread-class>Customer</thread-class>
  </thread>
  <activity name="OpenMarket" />
  <activity name="CloseMarket" />
  <activity name="CheckOpen" />
  <activity name="CheckStockInfo" />
  <activity name="SubmitAsk" />
  <activity name="SubmitBid" />
  <activity name="Transaction" />
  <activity name="BeforeOpen">
    <sequence>
      <invoke>CheckOpen</invoke>
    </sequence>
    <sequence>
      <invoke>CheckStockInfo</invoke>
    </sequence>
  </activity>
  <activity name="AfterOpen">
    <sequence>
      <invoke>CheckOpen</invoke>
    </sequence>
    <sequence>
      <invoke>CheckStockInfo</invoke>
    </sequence>
    <sequence>
      <sequence>
        <invoke>SubmitAsk</invoke>
      </sequence>
      <sequence>
        <invoke>SubmitBid</invoke>
      </sequence>
    </sequence>
    <synch>
      <sender>CR</sender>
      <receiver>CK</receiver>
    </synch>
  </activity>

```

```

    <invoke>Transaction</invoke>
  </sequence>
</activity>
<activity name="Entity">
  <sequence>
    <invoke regex="*">BeforeOpen</invoke>
    <invoke>OpenMarket</invoke>
    <synch>
      <sender>M</sender>
      <receiver>CK</receiver>
      <receiver>CR</receiver>
    </synch>
    <invoke regex="*">AfterOpen</invoke>
    <invoke>CloseMarket</invoke>
    <synch>
      <sender>M</sender>
      <receiver>CK</receiver>
      <receiver>CR</receiver>
    </synch>
  </sequence>
</activity>
<activity>
  <sequence>
    <invoke>constructor</invoke>
    <invoke regex="*">Entity</invoke>
    <invoke>destructor</invoke>
  </sequence>
</activity>
</smtss>

```

#### 4. Scenario Specification

One of key activities in testing a distributed application using WS is functional testing, which often involves specification of system behavior scenarios and development of test cases/scripts based on the specified scenarios. To derive scenarios and generate test scripts of distributed systems, a tester can use the following steps:

- Derive scenario specification for each sub-system, and formalize the scenario specification by annotating each scenario as a sequence of events, actions, and associated pre-/post-conditions;
- Specify the interaction between each pair of sub-systems;
- Derive the overall scenarios for the distributed system by combining the scenarios for individual sub-systems with the interaction; and
- Generate test scripts based the scenario specification.

##### 4.1 Derive Scenarios for Each Sub-System

This step derives scenario specification for each sub-system. The scenario specification follow the process

described in [2],[42]. Each scenario can be classified as an atomic scenario, a sub-scenario, or a complex scenario. The derived scenarios are organized into a tree structure with each sub-tree represent a group of functionally related scenarios that the tester can analyze them together in a hierarchical manner. Each scenario is annotated with Actors, Conditions, Data, Action, Timing and Events (ACDATE) by decomposition:

- Actor: the user or system component
- Condition: System decision point and states
- Data: Associated input and output
- Action: System operations and methods
- Event: External stimuli as well as action results

##### 4.2 Specify the Interaction between Systems

Most of time individual WS need to collaborate with each other to perform the mission. Fortunately, WS often provide well-defined interfaces through which they interact with each other. For example, in a Supply Chain Management (SCM) system, entities (such customers, retailers, manufacturers, and suppliers) interact with each other through API (Application Programming Interface), and many components within a SCM can be implemented as WS. The Web Services Interoperability (WS-I) defines three interaction scenarios in supply-chain WS [56]:

- One-Way: a customer sends a request message to a provider without response from the provider.
- Synchronous Request/Response: a customer sends a request message to a provider. The provider receives the message, processes it, and sends back a response.
- Basic Callback: At runtime a customer sends an initial request to the provider, which in turn sends back an acknowledgement immediately. At a later time the provider sends the final response to the customer.

In a SCM system, Retailer provides to Customer three services (or API): *login*, *getCatalog*, and *submitOrder*; to Manufacture one service: *submitShippingNotice*. On the other hand, Manufacturer provides to Retailer one service: *submitPurchaseOrder*. *submitPurchaseOrder* is one way interaction; and *login*, *getCatalog*, and *submitOrder* is synchronous interaction; while *submitShippingNotice* is callback interaction.

##### 4.3 Derive the Overall System Scenarios

Based on scenarios for each individual sub-systems and interaction among these sub-systems, a tester can now derive the overall system scenarios. An overall scenario can be constructed by combining scenarios for each sub-system with the interaction patterns. For example, scenario "Customer Successfully Purchases Goods"

consists of the following sub-scenarios "A customer accesses to the retailer system with an ID", "the client purchases a product by specifying the product ID to the retailer", "the retailer sends a request to the manufacturer", then "the manufacturer shipped the requested product to the client", and finally, "the retailer system returns a confirmation to the customer". And all three types of interactions are present in the overall scenario.

A specification language BPEL4WS (Business Process Execution Language for Web Services) [4] can also be useful to specify the overall scenarios. It provides programming constructs to specify various control sequences and interaction protocols.

#### 4.4 Generate Test Scripts from Scenarios

Scenarios describe the system behaviors responding to stimuli/request under different conditions. Test execution however needs one and only one execution path of a scenario. A thin thread describes a single execution path of the system, and thus a scenario can have at least one thin thread. Test scripts can be obtained by instantiating a thin thread by supplying the input data of thread. Once the input data are supplied, it is possible to generate test inputs based on different testing techniques [43]:

- Random testing: random data from the data set will be used as test input.
- Partition testing: selected sample data from each partition of the data in the data set will be used as test input.
- Boundary value testing: boundary values of data partitions in the data set will be used as test input.

As a WSDL file contains the signatures specification of all the WS methods, one can extract the interface and URL information from it and maps to the signatures to test scripts. For example, if a bank WS has two methods *checkBalance* and *deposit* and their signatures are as follows:

```
double checkBalance(String account);
boolean deposit(String account, double amount);
```

And the overall scenario is "a customer deposit a certain amount of money and then check the balance". The following XML code shows the generated test script.

```
<scenario name="Check Balance">
  <scenario_configuration>
    <URN>BankServices</URN>
    <SOAProuterURL>
      http://.../soap/servlet/rpcrouter
    </SOAProuterURL>
  </scenario_configuration>
  <scenario_path>
    <method sequence="1">
```

```
      <name>deposit</name>
      <data direction="input">
        <dataname>account</dataname>
        <type>string</type></data>
        ...
      <data direction="output"><dataname/>
        <type>Boolean</type></data>
    </method>
    <method sequence="2">
      <name>checkBalance</name>
      ...
    </method>
  </scenario_path>
  <scenario_testcase>
    <testcase>
      <testdata sequence="1">foo</testdata>
      <testdata sequence="1">100.00</testdata>
      <testdata sequence="1">>true</testdata>
      ...
    </testcase>
  </scenario_testcase>
</scenario>
```

## 5. Web Services Testing Framework (WSTF)

WSTF is a distributed architecture with three types of components: test masters, test agents, and test monitors. The WSTF used several design patterns such as Factory, Proxy, Composite, Command, and Mediator [11], [42] to facilitate design changes.

### 5.1 WSTF Architecture

Fig.2 shows the overall architecture with four types of components: test master, test agent, monitor, and finally the WS under test.

- **Test master:** it has GUI components for scenario specification, analyzer components for analyses such as dependency analysis, completeness and consistency check, concurrency detection, synchronization identification, and timing constraints. The enhanced WSDL dependency information (see Section 3) is useful for WS dependency analysis, which directs ripple effect analysis and regression testing. Based on the scenario specification, test scripts/cases can be generated. All the information is stored into database. Test master creates test plan according to test requirement/criteria, such as risk, usage, and coverage, and resource/time constraints. Upon the plan, test master pull out test scripts from database, and sends them to test agents via TCP/IP or SOAP. Test master acts as a test-agent coordinator or mediator so that a group of test agents cooperate with other to achieve the tasks assigned. Test master receives test results from test agents, and stores them into database

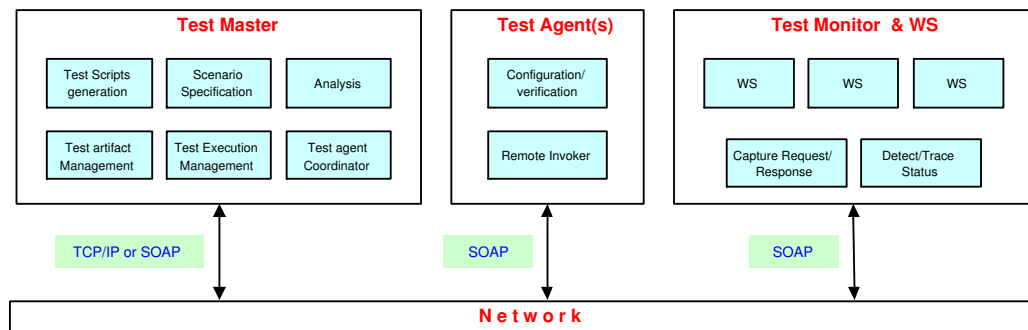


Fig. 2 Overview of WSTF architecture.

for analysis.

- **Test agent:** It has a component to perform WS testing by setting up the test environment, dynamically binding and invoking remote objects/services via SOAP. Also each test agent verifies the test results and reports the results to the coordinator component in test master.
- **Monitor:** It captures the exchange data including request and response, timestamps them and store into files/database. If the request has the specified values/data formats, the monitor can verify and validate them simultaneously. After monitor captures the exchange data, test agent coordinator collects all information for the verification. Also the monitor detects and traces state change information on WS under test, and reports to test agent coordinator.

## 5.2 WSTF Design

### Test Master

Test master provides the following functions (Fig. 3):

- Generates test scripts and test cases (*generateTestScripts()*) from scenarios by creating test scripts objects (*TestScript* class) and scenario object (*ScenarioSpecification* class);
- Initiates the testing by sending commands to test agents (*runScenario()*) through TCP/IP or SOAP depending tester's environment of configuration;
- Collects test result reports from test agent (*TestAgent* class) and messages from test monitor (*TestMonitor* class) for testing analyses by using exchange message class (*ExchangeMessage* class);
- Manages the dependency and synchronization among test agents, test monitors (*synchronize()*), that is test master acts as a test-agent coordinator; and
- Verifies interaction between systems (*verifyInteraction()*).

### Test Agent

Test agent provides the following functions to act as a proxy with respect to the WS (Fig. 3):

- Maps test script's information to methods in test agent:
  - The preconditions, events, input data to *setConfiguration()*,
  - The actions to *invokeFunction()*, and
  - The post-conditions to *verify()*;
- *setConfiguration()* sends requests to invoke setter services on service provider to setup testing via SOAP;
- Executes testing by running *invokeFunction()*, which calls *exec()* that is a Template Method [11] defining the overall calling procedure: *bind()* and *invoke()* using Command pattern, Proxy pattern and Strategy pattern (*CommandProxy* class). The Strategy pattern encapsulates communication protocols, so that the user can add new protocol to the framework without changing the structure.
- Test agent may need to invoke another services on different provider's site according to dependent information described in test script;
- Verifies the testing results against post-condition against verification patterns [59]; and
- Sends test results to the test master.

### Test Monitor

Test monitor provides the following functions (Fig. 3):

- Monitors the exchange messages with timestamps among requestors and service providers (*captureMessage()*).
- Traces state changes of the systems (*captureStateChange()*) with respect to requests. With the trace information, the systems can roll back to a previous state.
- Keeps track of asynchronous messages, notices the requestors when the asynchronous requests are served (*noticeStateChange()*).

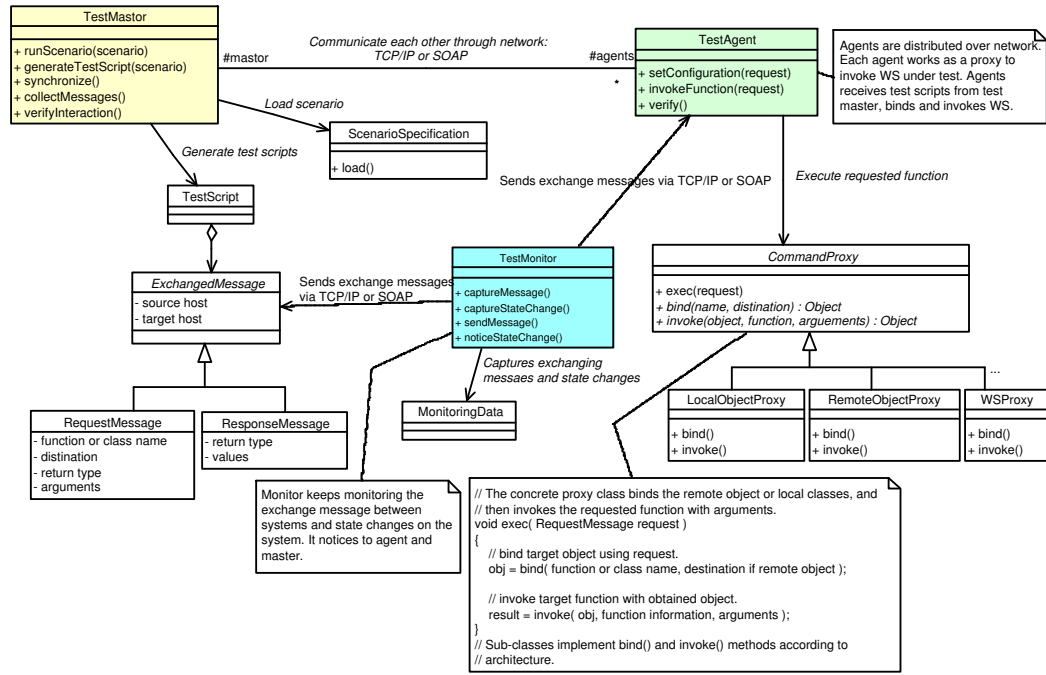


Fig. 3 Class diagram of test master, test agents and test monitor.

- Sends the exchange information to test master and test agent, which add listeners to it.

The sequence diagram in Fig. 4 shows a typical usage of WSTF for a client or service provider to test and evaluate a WS:

1. A tester specifies one or a group of particular scenarios, and starts running it.
2. Test master loads the scenario data using *Scenario* class.
3. Test master generates the Test Script with test cases from loaded scenario data.
4. Test master requests test agent to execute the system function or set configuration of system step by step of Test Script.
5. Test agent binds and invokes the system function/service.
6. Each test monitor keeps monitoring the associated system. It captures the exchange messages among systems and state changes of system.
7. Test monitor sends the captured information back to test master and test agent.
8. Test master collects the information related scenario, and verifies the test results and interaction between systems.

### 6. Experimentation

This section illustrates the proposed techniques in testing an SCM application developed using WS. The example has multiple customers, one Retailer and three Manufacturers. Table 1 lists the APIs Retailer and

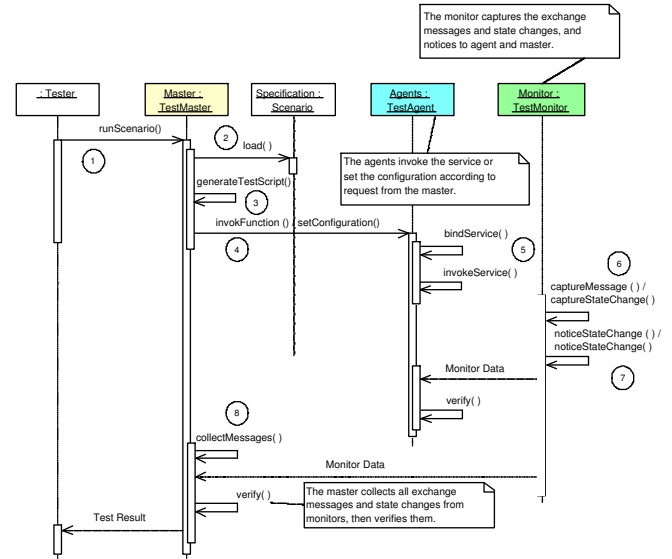


Fig. 4 Sequence diagram of test master, test agents and test monitor.

Manufacturer provide.

The example also has the following constraints:

1. The customer has only three opportunities to login into the system, i.e., the customer can fail the login in the first two times.
2. The maximum stock level is 20 items, i.e., the manufacturer cannot deliver more than 20 item at any given time.
3. The minimum stock level is 3, i.e., when the stock drops to 3 items, the retailer must place an order

**Table 1** Retailer and Manufacturer APIs.

Retailer APIs:	
login	The customers login the SCM system with their customer ID.
getCatalog	The customer requests the list of products.
submitOrder	The customer submits an order to specify the product and quantity, and then receives a response indicating the time good will be shipped.
submitShippingNotice	After manufacturer finishes processing an order, it submits the shipping notice.
logout	The customers logout the SCM system.
Manufacturer API:	
submitPurchaseOrder	The manufacturer places a purchase order when it finishes goods.

to one of manufactures.

- The limitation of number for multiple customers to access the retailer is configured to 30, i.e., at most thirty of customers can access the WS at the same time.
- The retailer will pick up one of manufactures who can deliver with a specific deadline.

Scenarios can be derived based on the above description and constraints for a single/multiple customer(s):

- Customer(s) successfully purchase all goods.
- Customer(s) successfully purchase goods, but at least one product is shipped.
- Customer(s) successfully purchase goods, then replenish due to the minimum Level.
- Retailer orders products from a manufacturer successfully.
- Customer(s) failed to purchase goods due to invalid product.
- Customer(s) failed to purchase goods due to invalid quantity.
- Customer(s) failed to purchase goods due to unavailable quantity.
- Customer(s) failed to purchase goods due to out of stock.
- Retailer failed to order products from a manufacturer.

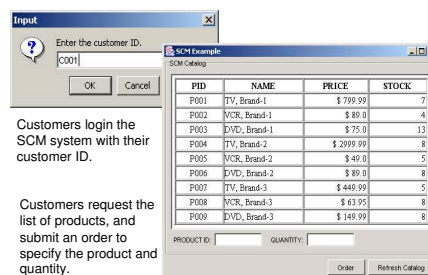
To cover all the configuration combination of interest, 10,319 number of test scripts are generated to test the SCM example. To perform testing, the experiment sets up the WS environment using the Java runtime environment (JRE) 1.4, Apache Tomcat 4.0, IBM WS toolkit (WSTK) 3.2 [54].

To evaluate WSTF, a number of faults are seeded into the example system, including bugs in APIs, internal services, and communication protocol as shown in Table 2

To test the SCM system, a tester does not need

**Table 2** Seeded faults in the example system.

Type of Seeded Bugs	Illustrations	# of Seeded Bugs
API	- Incorrect input types. - Mismatching types, e.g., incorrect number of parameters.	8
Internal services	- Malfunctions (e.g., can not update the inventory; cannot find a proper manufacturer). - Incorrect data, e.g., incorrect passwords.	7
Communication	- Incorrect protocols	3

**Fig. 5** Client GUI.**Table 3** Test results.

# of Test Scripts	# of Passed Test Script	# of Failed Test Scripts	# of Detected Bugs
10,319	6,503	3,816	18

to write any testing code; instead the tester can focus on specifying scenarios. The example uses E2E tool [30], [36], [37], [42], [43] to specify the requirement and constraints and generates all the test scripts. Fig. 5 shows the user interface for customers to login, browser product catalog, and choose products from the retailer.

Fig. 6 shows the test execution situation:

- Test master selects a set of scenarios and sends to a test agent.
- The test agent binds and invokes WS, and reports the test results to the test master. If the test case failed, it will be highlighted in red color.
- The test monitor captures all the communication messages and displays them.

With WSTF, all the seeded bugs are detected. Table 3 reports the test results.

When the SCM system is modified, it is necessary to test the modified feature as well as to perform regression testing to ensure that the modifications do not introduce new bugs. The former can be done by re-specifying scenarios that involved the modified feature, and generating test scripts based on these scenarios [42]. Regression testing can be done by identifying those scenarios that are potentially affected by the change by dependency analysis. The framework then generates or re-generates test scripts corresponding to

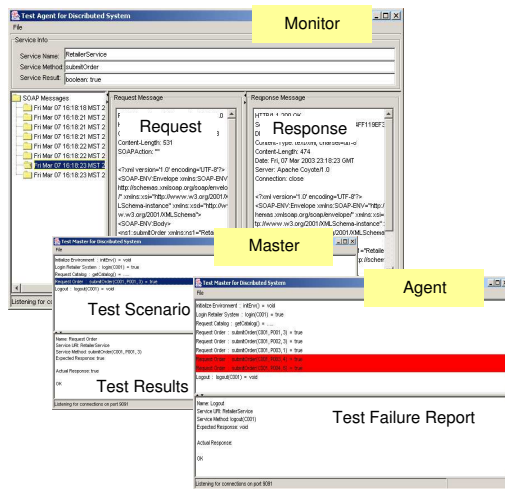


Fig. 6 Roles of Test master and Test Agent.

these scenarios for regression testing. The modified system scenarios can be further simulated to see if the runtime behavior is still valid.

7. Conclusion

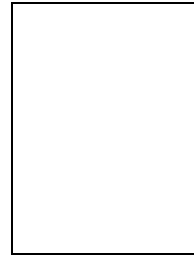
The paper presents a scenario-based testing framework for WS testing with distributed agents such as test masters, agents, and monitors. Testing WS is different testing traditional programs because WS are published, searched, bound, executed, and evaluated at runtime, and thus it is much more complicated than testing traditional applications. By using distributed agents, one can specify test scenarios, then WSTF generates test scripts, executes test, collects and monitors test results, and evaluates test results at runtime. In other words, the traditional verification mechanism now must be done by distributed agents running concurrently with the application under test. Furthermore, this paper also suggests that how some standard WS protocols can be enhanced to support WS testing. Otherwise it is difficult to carry out verification at runtime. Finally, this paper presents a working test framework with these distributed agents that are used to test a supply-chain system.

References

[1] D. Amyot, Use Case Maps as a Feature Description Notation, Fireworks Feature Constructs Workshop, UK, 2000, <http://www.usecasemaps.org/UseCaseMaps/pub/fireworks2000.pdf>.  
 [2] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, Scenario-Based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing, Proc. of IEEE WORDS 2002, pp. 140-151.  
 [3] J. Bloomberg, Testing Web Services Today and Tomorrow, The Rational Edge, October 2002.  
 [4] Business Process Execution Language for Web Services: <ftp://www6.software.ibm.com/software/developer/library/>

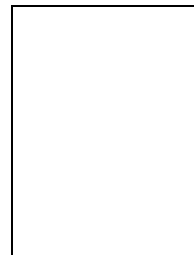
[ws-bpel11.pdf](http://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf).  
 [5] Cactus project, <http://jakarta.apache.org/cactus/index.html>.  
 [6] D. M. Cohen, The AETG System: An Approach to Testing Based on Combinatorial Design, IEEE Transaction on Software, July 1997, pp. 437-444.  
 [7] Consumer Report: <http://www.consumerreports.org/>.  
 [8] HttpUnit, <http://httpunit.sourceforge.net/>.  
 [9] M. Fayad, D. C. Schmidt, and R. E. Johnson, Building Application Frameworks, Wiley, New York, NY, 1999.  
 [10] D. G. Firesmith, Pattern Language for Testing Object-Oriented Software, Object Magazine, Vol. 5, No. 9, Jan. 1996, pp. 42-45.  
 [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, MA, 1994.  
 [12] A. Hoffmann, A. Rennoch, I. Schubert, and A. Vouffo-Feufjio, CCM Testing Environment, <http://www.fokus.fhg.de/tip>, 2001.  
 [13] I. Jacobson, M. Christerson, P. Jonson, and G. Overgarrrd, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, Reading, MA, 1992.  
 [14] JUnit.org, <http://www.junit.org/index.htm>.  
 [15] ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-Y, Geneva, 1996.  
 [16] S. H. Kirani and W. T. Tsai, Method Sequence Specification and Verification of Classes, Journal of Object-Oriented Programming, Oct. 1994, pp. 28-38.  
 [17] S. H. Kirani and W. T. Tsai, Specification and Verification of Object-Oriented Programs, Technical Report TR94-64, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 1994.  
 [18] A. Kotok, Getting Web Services Ready for Business, <http://www.webservices.org/index.php/article/articleview/1014/1/24/>.  
 [19] D. Kulak and E. Guiney, Use Cases: Requirements in Context, ACM Press, Addison Wesley, Reading, MA, 1998.  
 [20] A. Mani and A. Nagarajan, Understanding quality of service for Web services, <http://www-106.ibm.com/developerworks/webservices/>.  
 [21] Marconi Communications, <http://www.telelogic.com/download/usergroups/uk2002/Marconi.pdf>.  
 [22] A. McCarthy, Unit and Regression Testing, Dr. Dobbs Journal, Feb. 1997.  
 [23] T. Mackinnon, S. Freeman, and P. Craig, Endo-Testing: Unit Testing with Mock Objects, Proc. of Extreme Programming and Flexible Processes in Software Engineering - XP2000, 2000.  
 [24] J. D. McGregor and A. Kare, Parallel Architecture for Component Testing of Object-Oriented Software, Proc. of Annual Software Quality Week, Software Research Institute, May 1996.  
 [25] B. McKee and D. Ehnebuske, Providing a Taxonomy for Use in UDDI Version 2, <http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-taxonomy-provider-v100-20010717.pdf>.  
 [26] A. P. Mathur, Testing Distributed Systems, Status Report of NSF and SERC, 1999.  
 [27] NASA, Goddard Space Flight Center, <http://pioneer.gsfc.nasa.gov/public/sensor-web/SWAP%20Scenario.doc>.  
 [28] A. K. Onoma, W. T. Tsai, F. Tsunoda, H. Suganuma, and S. Subramanian, Software Maintenance - An Industrial Experience, Journal of Software Maintenance, Vol. 7, Dec. 1995, pp. 333-375.  
 [29] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Suganuma, Regression Testing in an Industrial Environment, Communications of the ACM, Vol. 41, No. 5, May 1998, pp. 81-86.

- [30] R. Paul, W. T. Tsai, and L. Yu, Rapid and Adaptive End-to-End Test and Evaluation of Systems of Systems, to submit to STC, 2003.
- [31] M. Poonawala, S. Subramanian, R. Vishnuvajjala, W. T. Tsai, R. Mojdehbaksh, and L. Ello, Testing Safety-Critical Systems – A Reuse-Oriented Approach, Proc. of 9th International Conference on SEKE, 1997, pp. 271-278.
- [32] SCENT: <http://www.research-projects.unizh.ch/oe/unit35100/area305/p922.htm>.
- [33] SOAP: <http://www.w3.org/TR/SOAP/>.
- [34] The Stencil Group, The Evolution of UDDI, UDDI.org White Paper, [http://www.uddi.org/pubs/the\\_evolution\\_of\\_uddi\\_20020719.pdf](http://www.uddi.org/pubs/the_evolution_of_uddi_20020719.pdf).
- [35] W. T. Tsai, R. Paul, W. Shao, S. Rayadurgam, and J. Li, Assurance-Based Y2K Testing, Proc. of IEEE HASE, 1999, pp. 27-34.
- [36] W. T. Tsai, Y. Tu, W. Shao and E. Ebner, Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates, Proc. of IEEE COMPSAC, 1999, pp. 166-171.
- [37] W. T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, End-to-End Integration Testing Design, Proc. of IEEE COMPSAC, 2001, pp. 166-171.
- [38] W.T. Tsai, X. Bai, R. Paul, and L. Yu, Scenario-Based Functional Regression Testing, Proc. of IEEE COMPSAC, 2001, pp. 496-501.
- [39] W. T. Tsai, L. Yu, R. Paul, T. Liu, and A. Saimi, Developing Adaptive Test Frameworks for Testing State-Based Embedded Systems, Proc. of IDPT, 2002.
- [40] W. T. Tsai, Y. Na, R. Paul, and F. Lu, Adaptive Scenario-Based Object-Oriented Test Frameworks for Testing Embedded Systems, Proc. of IEEE COMPSAC, 2002, pp. 321-326.
- [41] W. T. Tsai, R. Paul, W. Song, and Z. Cao, COYOTE: An XML-Based Framework to Test Web Services, Proc. of IEEE HASE, 2002, pp. 173-174.
- [42] W. T. Tsai, L. Yu, R. Paul, and A. Saimi, Scenario-based Object-Oriented Test Framework for Testing Distributed Systems, to appear in Proc. of IEEE FTDCS 2003.
- [43] W. T. Tsai, L. Yu, X. Liu, A. Saimi, and Y. Xiao, Scenario-based Test Generation Tool for Embedded Systems, in Proc. of IEEE IPCCC 2003, pp.335-342.
- [44] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, Extending WSDL to Facilitate Web Services Testing, Proc. of IEEE HASE, 2002, pp. 171-172.
- [45] W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, Verification of Web Services Using an Enhanced UDDI Server, to appear in Proc. of IEEE WORDS, 2003.
- [46] S. Uehari, W. T. Tsai, T. Sano, and I. Baba, Maintenance and Re-engineering: Architecture- and Domain-Oriented, (in Japanese) Kyouritsu, Tokyo, Japan, 2000.
- [47] UDDI.org: <http://www.uddi.org/>.
- [48] UDDI Version 2.0 Data Structure Reference, <http://www.uddi.org/pubs/DataStructure-V2.00-Open-20010608.pdf>.
- [49] A. W. Ulrich, P. Zimmerer, and G. Chrobok-Diening, Test Architectures for Testing Distributed Systems, Proc. of Quality Week 1999.
- [50] W3C, Web Services Definition Language, <http://www.w3.org/TR/wsdl>.
- [51] Y. Wang, R. V. Vishnuvajjala, and W. T. Tsai, Sequence Specification for Concurrent Object-Oriented Applications, Proc. of IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 1997, pp. 163-170.
- [52] Web Services Architect: <http://www.webservicesarchitect.com/>.
- [53] WebServices.Org: <http://www.webservices.org/>.
- [54] Web Services Toolkit, <http://www.alphaworks.ibm.com/tech/webservicestoolkit>.
- [55] E. Weyuker, Testing Component-Based Software - A Cautionary Tale, IEEE Software, Sep.-Oct. 1998, pp. 54-59.
- [56] WS-I: <http://www.ws-i.org/>, WS-I Usage Scenarios, Supply Chain Management Use case Model, Sample Application Supply Chain Management Architecture.
- [57] Y. Wu, D. Pan, and M. Chen, Testing Component-Based Software, International Conference Software & Systems Engineering and their Applications - ICSSEA '2002, December 2002.
- [58] Zaphink: <http://www.zaphink.com/>.
- [59] F. Zhu, A Requirement Verification Framework for Real-time Embedded Systems, Ph.D. dissertation, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 2002.



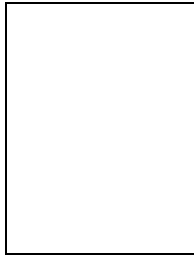
**Wei-Tek Tsai** received his Ph.D. (1986) and M.S. (1982) in Computer Science from University of California at Berkeley, CA, and SB (1979) in Computer Science and Engineering from MIT, Cambridge, MA. He is now a professor of Computer Science and Engineering at Arizona State University, Tempe, Arizona. Before coming to Arizona, he was a professor of Computer Science and Engineering at University of Minnesota, Minneapolis,

Minnesota. His main research areas are software testing, software engineering, embedded system development. His work has been sponsored by DoD, NSF, Intel, Motorola, Hitachi Software Engineering, Fujitsu, US WEST, Cray Research, and NCR/Comten.

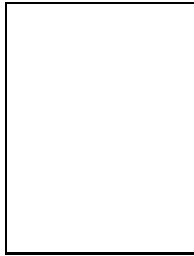


**Ray Paul** has been a professional electronics engineer, software architect, developer, tester and evaluator for the past 24 years, holding numerous positions in the field of software engineering. Currently, he serves as the deputy for C2 Metrics and Performance Measures for Software for the Department of Defense (DoD) Chief Information Officer (CIO). In this position, he supervises development of objective, quantitative data on the status of

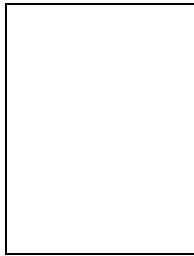
software resources in DoD information technology (IT) to support major investment decisions. These metric data are required to meet various congressional mandates, most notably the Clinger-Cohen Act. He holds a doctorate in software engineering and is an active member of the IEEE Computer Society. He has published more than 50 articles on software engineering in various technical journals and symposia proceedings, primarily under IEEE sponsorship.



**Lian Yu** received her Doctor of Engineering degree from Yokohama National University (Department of Electrical and Computer Engineering) in 1999. She joined the faculty of the Department of Information Systems and Science from 1999 to 2000. She is a faculty associate at the Department of Computer Science and Engineering of Arizona State University. Her main research areas are software engineering, validation and verification, parallel machines scheduling, fuzzy inference to production management problems, distributed computing, Web Services and supply chain management.



**Akihiro Saimi** is a software engineer at Hitachi Software Engineering, Tokyo, Japan. He currently joins the research projects of Computer Science and Engineering at Arizona State University. He works in various research projects, such as object-oriented development techniques, testing object-oriented applications, testing distributed systems, Web applications, and Web services.



**Zhibin Cao** received his M.S. (2003) in Computer Science from Arizona State University, AZ and SB (1997) in Computer Science from Beijing Information Technology Institute, Beijing, China. He pursued his Ph.D degree in Department of Computer Science of Arizona State University. He currently works in End to End testing project that focuses on scenario-based system modeling, simulation, code generation and testing. He is interested

in software engineering, software programming/testing, network and distributed operating system.