Scenario-Based Service Specification and Testing

Jing-Ying Chen Chun-Han Lin
Department of Computer Science
National Chiao Tung University, Hsinchu, Taiwan
jyc@cs.nctu.edu.tw

ABSTRACT

Web Services is emerging as an Internet-based interoperability platform and is attracting enormous research and development efforts currently. Because one major objective of Web Services is to allow building distributed systems using Web services developed by different parties, service composition and coordination become an important part of the Web service architecture. One challenge for Web services-based system development is to ensure the correctness and quality of the services developed by different parties. Currently, Web Services only provides basic interface description and registry standards but leaves behavioral specification open. In this paper we propose to supplement Web service description with scenario-based specification so that not only the semantics and intentions of individual services, but also the interrelations between services can be codified and become easier for developers to understand. In addition, we develop a testing framework which is capable of generating testing stubs and drivers automatically based on provided scenarios, and performing test cases involving multiple, distributed Web services in a coordinated manner. In the long run, we believe our approach not only can improve the consistency and quality of Web services developed in a decentralized manner, but also can speed up the overall development process due to its support for enhanced requirements elicitation and rapid prototyping activities.

1: INTRODUCTION

Web Services [1] is emerging as a new interoperability platform on top of the Internet, and is attracting intensive research and development efforts from industry and academics. Web services (or just *services* from now on) are autonomous software systems exchanging XML-based messages among each other using standard the SOAP protocol [2], where different services can be implemented using different development technologies and by different, independent organizations. In order to achieve interoperability in such an Internet scale, services need to expose their interfaces through standard interface specification languages such as WSDL [3]. When services are registered and advertised in public directories, people can look for services they need and combine them to form custom distributed software systems.

We are concerned with challenges pertaining to service-oriented software development (SOD) in the emerging global service market entailed by the Web Services. In an SOD project, the target system consists of a number of services and other software entities such as databases, client applications, legacy systems, and so on. These entities may be completed already or under development, and they may be developed internally or purchased from the market.

In this sense services are a special but interesting class of software components, thus SOD is closely related to component-based development (CBD [4, 5]). Accordingly, many issues and challenges related to CBD, in particular COTS-based software development, also occur for SOD. Still, there are some differences between components and services. Most importantly, components are often static, reusable units that can be purchased and become the buyer's assets. Services, on the other hand, are dynamic, autonomous, and maintained by their owner, respectively. "Singleton" services are also common, such as geography map services offered by Internet companies such as Google and Yahoo!.

Unlike traditional software projects that are initiated and managed within an enterprise, in SOD projects services are independently developed and maintained, each designed with its own problem domain and considerations. In such a decentralized environment, to harness the heterogeneity exhibited by these services and tailor them for the project at hand, developers need to carefully study and validate the interfaces and related documents. Although the interfaces can be defined formally using WSDL, the behavioral aspects described in the associated documents may not be as rigorous [6]. Those documents may not contain sufficient information describing the services; or worse, they may contain misleading or out-dated information, either by mistake or due to the mismatch between documents and service implementation. In addition, during runtime, whether a service behaves as it claims to may not be easy to determine. Such a "trust problem" as presented in [7] should be resolved in order for SOD to be viable in the long run, but currently there is no suitable answer yet that is widely acceptable.

As mere interface specification is insufficient for SOD, recently many organizations and institutes put substantial efforts into creating standards for specifying desirable collaboration patterns among services. Notable examples include OASIS WS-BPEL [8] and W3C CDL [9]. These service orchestration and choreography

standards are especially important in business context, where it is common to involve multiple services and applications to carry out a business transaction in a predefined manner. Although these standards can help clarifying the behavioral implications of the participating services, they are essentially separated from the specification of the services themselves. Furthermore, it is not always straightforward, nor complete, to understand and verify services based on interface and choreography specifications.

Another equally important issue about SOD is the management of SOD processes. Due to the decentralized nature of SOD, the project manager may not have sufficient control over the whole development process of a project. The situation may become worse when considering that different services – either developed in-house or externally – may be in different phases of construction with different plans or rate of progress. Generally speaking, careful configuration management for SOD projects is necessary in this case. However, techniques and mechanisms to ensure the consistence of services being composed for a specific project, and at the same time not to compromise the evolution and improvement of individual services, remain a challenge to be tackled.

Although we do not attempt to answer all of the SOD challenges discussed thus far, we do believe that by providing suitable mechanisms and tools, many of the problems can be relaxed. In this paper, we propose to use *scenarios* as supplement information to describe the behavioral aspect of services. Briefly speaking, a scenario describes a particular sequence of interactions between services and applications using example messages. Unlike more comprehensive model-based specification techniques which attempt to describe software systems completely and rigorously, scenario-based specifications do not intend to cover all possible interaction sequences the system may exhibit. However, scenario-based approaches are more cost effective and easier to understand and implement when compared to model-based specification techniques.

Scenario-based specification technologies have another important benefit for SOD, because they fit nicely with software testing. In fact, scripts for unit testing can be regarded as simple scenarios. Similarly, larger scenarios involving multiple users and software entities also provide useful information for integration testing. As test-driven development has become one of the widely adopted development approaches today, we believe infrastructure and tool support for service testing will prove to be valuable for SOD.

As a proof of concept, we design and implement a scenario-based framework for service specification and testing. By specifying suitable scenarios and service interfaces, the framework can generate appropriate test drivers and stubs to facilitate unit testing and integration testing, and can execute and monitor test cases in a distributed manner. Because developers can implement services incrementally, perform testing, and obtain timely feedbacks, we believe our framework can be integrated in an overall SOD process nicely and contributes to the overall productivity.

The rest of the paper is organized as below. In section 2 we outline the architecture of our scenario-based specification and testing framework. In section 3 we describe further details about the scenario language. In section 4, we describe the use of scenarios for concurrent service testing. After some discussions about our approach in section 5, we conclude the paper and discuss some future work in section 6.

2: A SERVICE SPECIFICATION AND TESTING FRAMEWORK

The architecture of our service specification and testing framework, called WST, is depicted in Figure 1, which consists of three basic subsystems:

Service Container hosts and manages services. Service containers themselves are also services so that clients can query and manipulate services through the containers that host them. When executing tests in WST, the testing manager (below) can create required services (test drivers and stubs) automatically based on scenarios, or use hand-crafted services implemented by developers.

Scenario Manager is responsible of storing and organizing scenarios. A scenario is usually referred to as a sequence of interactions between the user and the system, and is usually more relevant in requirement engineering and analysis phases. In this paper, however, we consider a more general case in which a scenario is an example sequence of interactions between the user and possibly more than one internal entities of the system. For proper separation of concerns, scenarios are organized hierarchically (e.g. [10, 11]). A scenario can include other scenarios in its specification, hence increases reusability and simplifies scenario representation.

Testing Manager governs all testing-related activities, such as creating test cases (handled by the test case manager), and generating testing stubs and drivers (handled by the emulator manager).

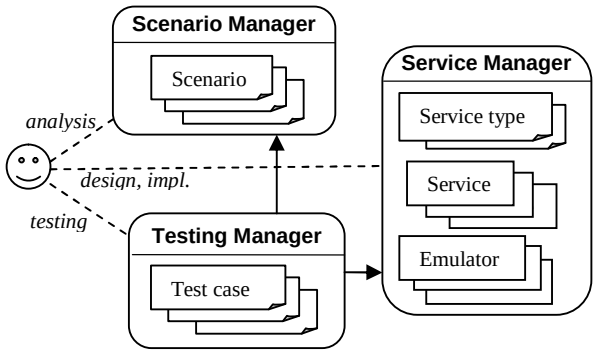

Figure 1. The WST architecture

In an SOD project, developers develop and maintain scenarios through scenario manager during the requirements elicitation phase, during which related requirements and/or UML diagrams may also be created.

On the other hand, the design and creation of service interfaces can be done before, after, or in parallel with scenario development. In particular, a service interface can be associated with a set of scenarios so that implementing services need to conform to the semantic constraints imposed by the scenarios. In addition, scenarios involving multiple services or applications can also be created and stored. These scenarios specify possible collaboration patterns and can help developers understand the design ideas and usage information of related services.

When the set of scenarios become sufficient, the developers then request the testing manager to generate appropriate test drivers and stubs, followed by actual unit testing and integration testing. Under the development plan, the developers may choose to implement some of the services with higher priorities and perform timely testing in an incremental manner, by replacing stubs with implemented services gradually. Such process is repeated until all system is developed.

3: SCENARIO-BASED SPECIFICATION

A service-oriented system contains multiple, mutually interacting services and other software entities, which we refer to as *components* for brevity. User interacts with the system by issuing requests to and receiving replies from the system in an order prescribed by some business rules. Upon receiving a request, the system may initiate a particular message exchange pattern among the constituent components. The message exchange pattern depends on the nature and content of the request as well as the current system state, and is not necessarily in a simple sequential order. Some exchanges may occur concurrently when the participating components have their own control flow.

Based on the system model above, we describe the behavior of the system using a collection of scenarios, which should be properly organized, classified, and managed. In order to achieve such divide-and-conquer goal, we define a supporting XML-based scenario language. Each scenario represents one particular message exchange pattern using example (but meaningful) messages. The abstract syntax for scenarios is shown below:

```
<scenario name="ncname">
   <role name="qname" type="qname"?/>+
   <seq>
      EVENT+
   </seq>
</scenario>

EVENT:
<call from="qname" to="qname">
   MESSAGE
</call> |
<endcall from="qname" to="qname">
   MESSAGE
</endcall> |
<notify from="qname" to="qname">
   MESSAGE
</notify> |
<state name="ncname" role="qname" ref="ncname"?
   state="qname"?/> |
<endstate name="ncname" role="qname" ref="ncname"?
   state="qname"?/> |
<sync from="qname" to="qname"/> |
<thread role="qname"/> |
<endthread role="qname"/>
```

As shown above, the definition of a scenario is divided into two parts. First the roles of participants in the scenario is declared, followed by a sequence of events each may indicate state transition or an interaction between two components.

Some basic types of events are described as follows. Firstly, <call> and <endcall> are events representing service invocation and completion, and should occur in pairs. <state> declares the state of the component and itself does not involve message exchanges. The name of the state is important since it is used as the identifier when combining multiple scenarios. <notify> represents asynchronous message notification and the caller do not wait for message receiver's response. In contrast, <sync> represents a synchronization step and its objective is to align the execution progress of two components. Normally, a component may initiate a sequence of actions after receiving a message. All events following the message reception (<call> or <notify> events from another component) and originated from the component are considered the actions to be taken by that component in the same order, until the event indicating the end of the sequence (e.g. by <endcall> or other incoming events). In case concurrent groups of actions need to be initiated, <thread> and <endthread> suit the purpose.

Note that although synchronization mechanisms is common among various concurrent programming models and languages (e.g. join operations or rendezvous mechanisms), their use is not common for ordinary Web services except for services that orchestrate other services (such as within a WS-BPEL process). In our language <sync> is primarily for testing purpose, which is necessary to coordinate concurrent services correctly; it is considered outside the responsibility of ordinary Web services.

The syntax of the scenario language is designed to be as simple as possible, so that the choreography among participants is just a sequence of events. However, the sequence is interpreted differently from the perspectives of individual components. In particular, a component only concerns the events that directly involve itself and discard the rest, which is what the semantics of our scenario language is based on.

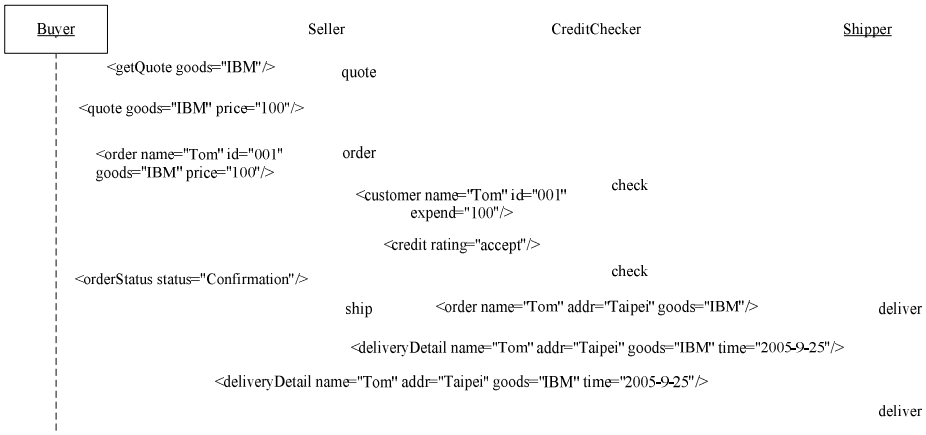

Figure 2. An ordering goods business process

Consider a goods-purchasing business process illustrated in Figure 2. In the beginning, Buyer interacts with Seller to determine the price. When Buyer decides to order the goods when the quote is acceptable, Seller interacts with CreditChecker to check for Buyer's credit. When the credit check is passed, Seller notifies Buyer with order confirmation and then sends a delivery request to Shipper. Finally, Shipper informs both Seller and Buyer about the delivery details.

To describe the process using our language, first the roles should be declared each representing a potential component of the system, as illustrated below.

```
<scenario name="orderGoods/orderGoodsSuccess">
   <role name="Buyer" type="BuyerType"/>
   <role name="Seller" type="SellerType"/>
   <role name="CreditChecker" type="CreditCheckerType"/>
   <role name="Shipper" type="ShipperType"/>
```

Note that the optional type attribute indicates the (service) interface the role should support, and is not required for "drivers" such as client applications.

A scenario combines all interactions among the roles. An interaction indicates an information exchange between two roles. The fragment below shows the first few interactions: when Seller is in "quote" state, Buyer sends requests to Seller for quotes, and then Seller responds with actual quotes.

```
<seq>
   <state name="quote" role="Seller"/>
   <call from="Buyer" to="Seller">
      <getQuote goods="IBM"/>
   </call>
   <endcall from="Buyer" to="Seller">
      <quote goods="IBM" price="100"/>
   </endcall>
   <endstate name="quote" role="Seller"/>
   . . .
```

Note that the scenario fragment above means differently for Buyer and Seller. From Buyer's perspective, during testing it will generate requests in the specified order. From Seller's perspective, it needs to remember the "right" answer for each request based on its current state. Note that although not shown here, additional scenarios but with different details, such as the goods ordered or whether credit checking is passed or not, can be created similarly.

As another example, assume that it takes time for Seller to send the delivery request to Shipper and wait for reply. In case the reply is irrelevant to the actions that follow, Seller may move such request-reply pair into a thread and proceed immediately, as shown below:

```
. . .
<thread role="Seller"/>
<call from="Seller" to="Shipper">
   <order name="Tom" addr="Taipei" goods="IBM"/>
</call>
<endcall from="Seller" to="Shipper">
   <deliveryDetail name="Tom" addr="Taipei" goods="IBM"
      time="2005-9-25"/>
</endcall>
<endthread role="Seller"/>
. . .
```

4: SERVICE TESTING

In WST, a test case contains multiple tests. An example test case is shown below, which is derived from more than one scenarios. In addition to the scenarios to be tested, the test case also binds the roles with corresponding components. As suggested in this example, different scenarios may exercise different "collaboration paths" of the same business process.

```
<testcase name="TestCaseExample">
   <service name="BuyerService" instance="buyerService"/>
   <service name="SellerService" instance="sellerService"/>
   <service name="CreditCheckerService" instance="creditChecker"/>
   <service name="ShipperService" instance="shipperService"/>
   <scenario name="creditCheck/creditCheckSuccess">
      <role name="Seller" is="SellerService"/>
      <role name="CreditChecker" is="CreditCheckerService"/>
   </scenario>
   <scenario name="orderGoods/orderGoodsSuccess">
      <role name="Buyer" is="BuyerService"/>
      <role name="Seller" is="SellerService"/>
      <role name="CreditChecker" is="CreditCheckerService"/>
```

```
    <role name="Shipper" is="ShipperService"/>
  </scenario>
</testcase>
```

Given a test case, test drivers and stubs can be generated. A driver simulates a client that calls the whole system under test or its components. A stub, on the other hand, simulates a component that has not been developed yet. When testing a large system that consists of multiple subsystems, instead of developing all subsystems completely and then performing a "big-bang" integration testing, one may adopt more incremental integration testing strategies, so that each subsystem can be developed concurrently, possibly with different priorities and timelines. During integration testing, to test a given subsystem, a test driver is needed to drive the interaction with the system under test. If the subsystem is yet to be developed, a test stub is created and used instead to simulate its behavior. Ideally, the stub should appear indistinguishable from the actual subsystem.

In our framework, both drivers and stubs are supported uniformly by *emulators*, namely, an emulator can act as a test driver and/or a test stub. Specifically, for a given test case the testing manager can generate a set of emulators corresponding to the roles in the test case, respectively.

An emulator consists of multiple states, and each state is further divided into multiple sections each corresponding to one distinct input message. Each such section, which we refer to as an *action block*, in turn contains a sequence of actions. The abstract syntax for emulators is shown below:

```
<emulator>
  <state name="qname">
    <accept>
      MESSAGE
      <actions> ACTION* </actions>
    </accept>+
  </state>+
</emulator>

ACTION:
  <return> MESSAGE </return> |
  <call to="qname"> IN_MESSAGE OUT_MESSAGE </call> |
  <notify to="qname"> MESSAGE </notify> |
  <wakeup to="qname"/> |
  <wait/> |
  <changeState name="qname"/> |
  <thread> ACTION+ </thread>
```

`<call>` represents a normal procedural invocation: when performing a call action the emulator will "invoke" the target service with the specified input message, and wait for response message which in turn is matched against the expected output message. `<return>` represents the end of the invocation originated from the input message that started the action block. `<changeState>` changes the state of the emulator, while `<notify>` sends a message to the target component without waiting for reply. `<thread>` is a composite action that encloses a sequence of *non-thread* actions; when a thread is executed the enclosed actions are executed in order, but different threads can execute concurrently. `<wait>` lets the emulator wait until a `<wakeup>` message from the expected participant arrives.

Emulators have straightforward operational semantics. An emulator can be in one of the designated states. Upon receiving an input message, the emulator looks up the corresponding action block based on the input message, performs the actions one by one, and finally changes to a new state if instructed to.

To simulate test drivers, we distinguish a special state, i.e. "driver" state, from other states in an emulator. The action block for the "driver" state is consider the driver part of the emulator, so that when the emulator starts execution, this driver part is performed spontaneously without waiting for an input message. Below is an emulator example that contains a stub and a driver.

```
<emulator role="Seller">
  <state name="driver">
    <accept>
      <run scenario="creditCheck/creditCheckSuccess"/>
      <actions>
        <call to="CreditChecker">
          <customer name="Tom" id="001" expend="100"/>
          <credit rating="accept"/>
        </call>
      </actions>
    </accept>
    . . .
  </state>
  <state name="orderGoods/orderGoodsSuccess/quote">
    <accept>
      <getQuote goods="IBM"/>
      <actions>
        <return>
          <quote goods="IBM" price="100"/>
        </return>
        <changeState
          name="orderGoods/orderGoodsSuccess/order"/>
      </actions>
    </accept>
  </state>
    . . .
</emulator>
```

With the operational semantics of the emulator language outlined above, the semantics of our scenario language is defined via an unambiguous mapping from a set of scenarios to a set of emulators. Due to space limit we do not describe further details about the mapping here; additional information can be found in [12].

5: DISCUSSION AND RELATED WORK

The development of WST is an ongoing process. Currently we focus primary on the core scenario management and testing facilities. Our long-term goal is to integrate the framework in an SOD process. There are many benefits when using WST. First, scenarios are straightforward to understand; they can be used in conjunction with UML use cases and sequence diagrams to help requirements elicitation.

More importantly, since scenarios can be executed immediately, requirement analysts can gain more timely and interactive feedbacks. With further tool support, a simulating "mockup" can be constructed the minute

when scenarios are created. Hence it is possible to use WST for rapid prototyping. From this perspective, future development in near term is not only to develop convenient scenario editors similar to UML sequence diagram editors, but also to develop interpreters that can interpret scenarios and present users with graphical user interfaces simulating the look and feel of the application.

The capability of WST to generate test drivers and stubs as emulators also facilitate rapid and incremental development cycles. The developers can choose to implement a (small) subset of services of the overall system and can test them immediately. For example, if the developers want to verify or test choreography which has been implemented by WS-BPEL, scenarios can be created in parallel to describe interaction behaviors among multiple Web services, and emulators generated automatically from multiple scenarios to substitute incomplete or unfinished Web services. Although it is necessary to transform the interface of emulators into standard Web service interface, the implementation is not too involving.

Our objective has been to exploit the use of scenarios for service specification and testing. To achieve the goal we have focused equally on both theoretical and practical aspects. From a practical perspective, it is desirable that our approach can be applied to actual development process for real-world Web services. One major issue is that there are currently multiple standards for Web services choreography under construction. Since there is no common agreement about how services are composed dynamically, let alone how they can be synchronize to enable correct concurrent testing, creating a public Web service testing framework that is universally applicable seems impossible. The issue is apparent when we consider complicated choreography scenarios involving multiple services that are not necessarily "memoryless. One possibility is to employ coordination mechanisms such as WS-Coordination [13] to enable consistent and automatic coordination [14]. Currently, our framework is limited in scope to the SOD projects where member service providers agree on technical issues such as how services are instantiated, managed, and composed.

From a theoretical perspective, developing a complete analytical framework covering all language constructs of WS-BPEL or WS-CDL is challenging, because both standards are quite complex. In contrast, our proposal permits straightforward emulator generation and simulation. Furthermore, the complexity of verification and validation will also be lower. However, our bias towards ease of implementation and analysis also poses a barrier when applying our work to practical situations, because there are still Web services that can not be easily modeled using our scenario-based language. This issue is resolved, similar to above, by limiting our scope to those Web services whose behavior can be approximated nicely by emulators.

As SOD grows important, many testing tools and techniques are being developed continuously. The Web service testing framework and approaches proposed in [10, 11] and [15] are some such examples. In particular, [10] and [11] propose to extend WSDL with information to facilitate testing, and to place supplement information such as test scripts for Web services inside UDDI so that verification can be performed when a Web service is checked in and out. Not surprisingly, the researchers who developed the systems above also work on scenario-based modeling and testing framework for distributed (OO) systems in a more general context [16, 17].

The central idea of their work is similar to ours, that is, with additional, testing-based or scenario-based information associated with Web services, users gain more insights into the behavior of the services. In addition, automated verification to some extent becomes possible. However, their work essentially corresponds to unit testing in that scenarios and testing scripts are associated with individual services, and we are more interested in service choreography, especially when multiple services are involved in potentially complex business processes. In fact, scenarios can coexist with and complement choreographies (e.g. WS-CDL documents) and enhance their clarity yet permit easier test script generation. Furthermore, we also attempt to integrate scenario-based specification techniques with the overall SOD process, in particular unit testing and integration testing, and more importantly in requirements elicitation. To achieve this goal, our specification language is simplified to facilitate automated test driver and stub generation.

6: CONCLUSION

We have proposed a scenario-based service specification language and a corresponding testing framework. We showed that scenarios can be used as supplement information to the syntax-only interface specification language, i.e., WSDL, thus providing a cost-effective approach to behavioral specification compared to either natural language or formal model-based approaches. More importantly, the design of the syntax and semantics of our scenario-based language permit automated test generation, including test drivers and stubs, in a straightforward way that not only increases the understandability of the language but also makes implementation less burdensome. Similar to integration testing that is commonly used in software development process, our scenario-based specification and testing approach can be an important contribution to Web service-based development where decentralization is the norm. As modern software development methodologies pay more attention to practices such as use case driven, test first development, rapid prototyping, incremental and iterative planning (agile methods), we believe WST will become relevant for the emerging SOD wave.

REFERENCES

[1] Web Services Activity, W3C. http://www.w3.org/2002/ws/

[2] W3C, “Simple Object Access Protocol (SOAP)”. http:// www.w3.org/TR/soap/
[3] W3C, “Web Services Description Language (WSDL)”. http://www.w3.org/TR/WSDL
[4] C. Szyperski, *Component Software: beyond Object-Oriented Programming*, Addison-Wesley, 1998.
[5] D.F. D'Souza and A.C. Wills, *Objects, components, and frameworks with UML: the Catalysis Approach*, Addison-Wesley, 1998.
[6] Sebastian Uchitel, Jeff Kramer, and Jeff Magee, “Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios“, ACM Transactions on Software Engineering and Methodology (TOSEM), 2004, pp. 37-85.
[7] A. Bertolino and A. Polini, “A Framework for Component Deployment Testing”, Proceedings of IEEE Software Engineering, 2003, pp. 221-231.
[8] OASIS WS-BPEL, “Web Services Business Process Execution Language”. See http://www.oasis-open.org/
[9] W3C, “Web Services Choreography Description Language Version 1.0”, http://www.w3.org/TR/ws-cdl-10/
[10] 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.
[11] W. T. Tsai, R. Paul, W. Song, and Z. Cao, “Coyote: An XML-Based Framework for Web Services Testing”, Proc. of IEEE HASE, 2002, pp. 173-174.
[12] Chun-Han Lin, "A Scenario-based Framework for Web Service Specification and Testing", MS Thesis, Computer Science Department, NCTU, January, 2006.
[13] IBM WS-Coordination, “Web Services Transactions specifications”,http://www-128.ibm.com/developerworks/library/specification/ws-tx/
[14] G. Alonso, *et al*, “Web Services Concepts, Architectures and Applications”, Springer-Verlag, 2004.
[15] W. T. Tsai, R. Paul, Z. Cao, L.Yu, A.Saimi, and B. Xiao, “Verification of Web Services Using an Enhanced UDDI Server”, Proc. of IEEE WORDS, 2003, pp. 131-138.
[16] W. T. Tsai, L. Yu, A. Saimi, and R. Paul, “Scenario-Based Object-Oriented Test Frameworks for Testing Distributed Systems”, Proc. of IEEE Future Trend of Distributed Computing Systems, 2003, pp. 288- 294.
[17] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, “Scenario-Based Modeling And Its Applications”, Proc. of IEEE WORDS 2002, pp. 140-151.