A Simple Approach for Testing Web Service Based Applications

Abbas Tarhini[1,3], Hacène Fouchal[2], and Nashat Mansour[3]

[1] LICA/CReSTIC, Université de Reims Champagne-Ardenne Moulin de la Housse, BP 1039, 51687 Reims Cedex 2, France
Abbas.Tarhini@univ-reims.fr
[2] GRIMAAG, Université des Antilles et de Guyane, F-97157 Pointe-à-Pitre, Guadeloupe, France
Hacene.Fouchal@univ-ag.fr
[3] Computer Science Division, Lebanese American University, PO Box 13-5053, Beirut, Lebanon
nmansour@lau.edu.lb

Abstract. The cost of developing and deploying web applications is reduced by dynamically integrating other heterogeneous self-contained web services. However, the malfunctioning of such systems would cause severe losses. This paper presents a technique for building reliable web applications composed of web services. All relevant web services are linked to the component under test at the testing time; thus, the availability of suitable web services is guaranteed at invocation time. In our technique, a web application and its composed components are specified by a two-level abstract model. The web application is represented as *Task Precedence Graph (TPG)* and the behavior of the composed components is represented as a *Timed Labeled Transition System (TLTS)*. Three sets of test sequences are generated from the WSDL files, the TLTS and the TPG representing the integrated components and the whole web application. Test cases are executed automatically using a test execution algorithm and a test framework is also presented. This framework wraps the test cases with SOAP interfaces and validates the testing results obtained from the web services.

Keywords: label transition systems, testing, verification, web service, web application.

1 Introduction

The development of web applications received significant attention in the past few years. They have been remarkably introduced into all areas of communication, information distribution, e-commerce and many other fields. The use of web services also provided a common communication infrastructure to communicate through the internet, and enabled developers to create applications that can span different operating systems, hardware platforms and geographical locations. Thus building reliable web applications and web services should be considered seriously.

Originally, web sites were constructed form a collection of web pages containing text documents and interconnected via hyper links. Only recently, the dramatic changes

A. Bui et al. (Eds.): IICS 2005, LNCS 3908, pp. 134–146, 2006.

of web technology lead web applications to be built by integrating different components from variety of sources, residing on distributed hardware platforms, and running concurrently on heterogeneous networks. The construction of systems from different types of software components faces various complexities and challenges such as maintaining performance, reliability and availability of those systems. Thus the validation of such web applications remains the main challenge. A Web application might invoke multiple web services located on different servers with no design, source code or interface available. This forces designers to use black-box notions to select the relevant web services from the pool of services found on the internet.

The technique presented in this paper is testing on the fly during the building of web systems. That is, during the development process, we test the web application and its interaction with remote services to select only relevant web services that build a correct web system. First, we suggest a two-level abstract model to represent a web application. Then, we generate three sets of test sequences that are used to automatically test the web system. Next, an automated testing technique is presented: it selects all relevant web services that interact with our web application and integrates them into the system before invocation time. This will guarantee the reliability and availability of services in the web application. Moreover, we present a test framework adapted from [13] for supporting both test execution and test scenario management. In this work we do not deal with performance, we only need a correct behavior. Moreover, the generated test cases use only symbolic values for variables satisfying an adequate coverage criteria. In order solve the state explosion problem, for test execution we may use either, boundary values for variables, or, we use a heuristic to choose values. In both cases, the test coverage will not be complete.

The rest of this paper is organized as follows. In Section 2 we present a brief background on web service models and previous work done on testing web services. The modeling of web applications and needed definitions are presented in Section 3. Section 4 presents our technique for testing web applications. In this section we show how to generate and execute test cases. In Section 5 we present the testing framework that is used to configure, generate and execute test cases. We conclude the paper in Section 6.

2 Background

In this section, we present an overview of web service infrastructure and definitions, and a brief survey of previous work done on testing web services.

2.1 Web Services Overview

Web Services were defined differently by vendors, researchers, or standards organizations. IBM defined web services as self-describing, modular applications that can be published, located and invoked across the web [12]. They are Internet-based, modular applications that uses the Simple Object Access Protocol (SOAP) for communication and transfer data in XML through the internet [10]. In our study, we define Web Services

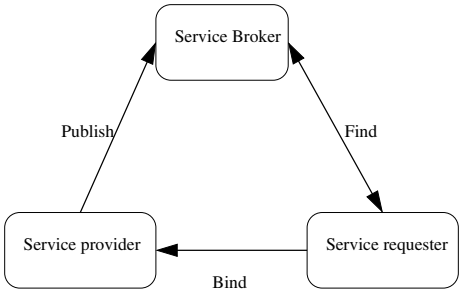

Fig. 1. Web service architecture

as self-contained component-based applications, residing on different servers, and communicating with other applications by exchanging XML messages wrapped in SOAP interfaces.

Web services infrastructure is based on service-oriented architecture(SOA) that involves three kinds of participants: service providers, service requesters and service broker (Figure 1). The provider publishes services to service broker. Service requesters find required services using the broker and bind to them [12].

This infrastructure uses the following standards to make web services function together: Web Services Description Language (WSDL), Universal Description, Discovery and Integration (UDDI), The Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP).

After creating a web service, the service provider generates the corresponding WSDL file and publishes it on the internet. The WSDL file is a description of how to access the web service and what operations this service can perform. On the service broker, the UDDI registry holds the specification of services and the URL that points to the WSDL file of services. The service requester searches for a web service in the UDDI registry, then binds to it, and transmit massages and data using XML wrapped in SOAP interfaces.

2.2 Testing Web Services

Several aspects make testing web services a challenging task. The heterogeneity of web services that uses different operating systems and different server containers makes the dynamic integration of these services a non-easy task. Moreover, web services do not have user interfaces to be tested [11] and therefore they are hard to test manually. Consequently, test frameworks, techniques, and tools have been studied by several researchers.

Song et al. [13] proposed an XML-based framework to test web services. The framework consists of two parts: the test master and test engine. The test master allows testers to specify test scenarios and cases as well as performing various analysis and converts WSDL files into test scenarios. The test engine interacts with the web service under test and provides tracing information. [6] proposed ideas towards enabling testing web services by using Design by Contract to add behavioral information to the specifica-

tion of a web service. The behavioral information includes contracts that describe behavior offered and behavior needed (pre and post conditions) by a web service. Then graph transformation rules are used to describe contracts at the level of models. Such contracts would help much during the execution of test cases; however, a list of issues are left open like creating XML-based language and UML-based notations for contracts.

[15] proposed a mobile agent-based technique to test web services. This approach needs the authentication of servers to allow mobile agents execute external code on them. It dynamically selects reliable web services at run time where no backup plan presented in case of unavailability of services satisfying the test criteria. [4] presented a white-box coverage testing of error recovery code in Java web services by provoking exceptions and evaluating how the web service handles them. This method covers many testing aspects, however, testing scenarios apply only to Java web services at a time web services are platform independent. Moreover, this technique requires the knowledge of web service code, at a time most web services are published as executable files and should be treated as black boxes, thus such techniques could be considered for in-house testing. [10] highlighted the difference between traditional applications and web services; web services use a common infrastructure, XML and SOAP, to communicate through the internet. The author presented a new peer-to-peer approach to testing web services by using test cases generated from the modification of existing XML messages based on rules defined on the message grammar. This approach is based on data perturbation. It uses data value perturbation (based on data type) and the interaction perturbation that tests communication messages (RPC communication and data communication). This approach relies on syntactic information about the XML messages; thus lacks behavioral information that are supported more in specification-based testing approaches which allow more detailed kinds of analysis.

Other testing tools and techniques focused on testing WSDL files and SOAP messages [8], and some recommended general best-practices that developers of web services can apply such as functional, regression and load testing [5]. [2] highlighted on what web services are and how to put to use. The author presented how to test web services manually through a web page and automatically through a programming language. Both approaches recommends to focus on what the web service expects as inputs and what it defines as its outputs. [14] proposed to extend WSDL files to support information useful for testing such as dependency information. By using these extensions, we can easily retrieve the necessary useful information for web service testing. This can greatly reduce the effort and cost to do these tasks and make the automation of these tasks possible. [7] surveyed testing techniques that can be applied to web services and detailed the advantages and drawbacks of some methods and tools. Then, the authors suggests fault injection technique as a promising line of research that can be applied to this problem.

Many papers have suggested testing Web services at invocation time, but performing a full-scale test of Web services integrated in Web applications before launch remains a considerable issue. Our testing technique selects and then associates all suitable web services to our web application before launch time; moreover, it suggests testing the functionality of the web service integrated in the web application by executing test cases generated from (1) the WSDL files and (2) the specification of both the

component fulfilled by a web service and the specification of the whole web application. This method guarantees the reliability and availability of services in our web application by ignoring both all services that act errantly in a composed environment and hosts of web services that act maliciously at invocation times.

In the following section we introduce some basic notations, the modeling of a web application and the modeling of a single component in a web application which are used in subsequent discussions.

3 Modeling Web Applications

Web-based software systems are constructed by integrating different interacting-components from a variety of sources. The schedule of invoking the interacting-components is restricted by the requirements specification of the web application and by time constraints. These components interact with the main application as well with other components by exchanging messages (actions) that might also involve timing constraints. To model such systems, we suggest a two-level abstract model. The first level models the interaction of components with the main application. The second level of abstraction models the internal behavior of each component in the system. In the following subsections we describe each model and illustrate it with examples.

3.1 Web Application Representation

Since Web applications are composed of components that interact by exchanging messages restricted by timing constraints, our first level of abstraction models a web applications as a *Task Precedence Graph (TPG)*, where each node in the TPG is an abstract representation of a single component in the system and an edge joining two nodes represents the flow of actions (transitions) between components. Every edge is labeled with an action and its timing constraint.

Figure 2 illustrates a *TPG* representing a simple travel agency web application that is composed of four components: Main Component (MC), Hotel Reservation (HR), Car Rental (CR), and Weather Prediction (WP). The Main component (MC) is assumed to

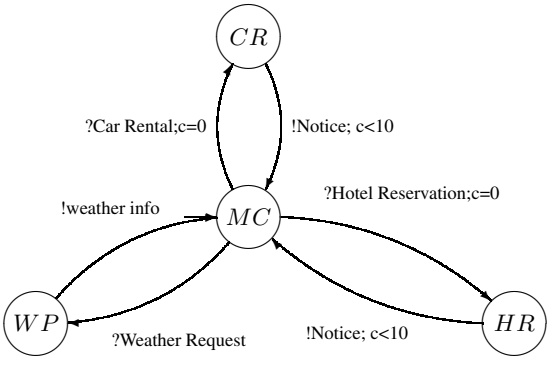

Fig. 2. An example of TPG representing a travel agency web application

be the background component that handles requests from the main web page in the web system. Each of the attached components is invoked whenever its corresponding input is selected and it returns the output back to the Main Component (MC). Thus, if the user wants to reserve a hotel, the transition labeled (?Hotel Reservation; c =0; -) is executed as soon as the input $?HotelReservation$ is invoked, and the clock c is set to zero, so that it counts the time taken by the web service to fulfill the user request. The output from the component HR should be sent back with in the time limit (c<10); if not, that means the invoked web service might be not available, thus, the MC component will request HR to contact another web service.

3.2 Single Component Representation

The second level of abstraction models every single component in the web application. In this level, we suggest to model each component as a *Timed Labeled Transition System (TLTS)*. Each state in the TLTS represents a state of the modeled component. An edge joining two states is labeled with an action and its corresponding timing constraint. It represents a transition from one state to another. We formally define an TLTS as follows:

Definition 1 (Timed Labeled Transition System (TLTS)). *An* **TLTS** *is defined by* $M = (S, A, C, T, s_0)$ *where* S *is a finite set of states,* s_0 *is the initial state, and* A *is a set of actions.* A *is partitioned into 2 sets:* A_I *is the set of input actions (written ?i),* A_O *is the set of output actions (written !o).* C *is a set of clocks.*

T is a transition set having the form $\{Tr_1.Tr_2...Tr_n\}$; Tr_i **= <s; a; d; EC;** C_s**>**, where: **s** $\in S$ and **d**$\in S$ are starting and destination states; **a** $\in A$ is the action of the transition; **EC** is an enabling condition evaluated to the result of the formula $a \sim b$ where $\sim \in$ { <, >, $\leq$, $\geq$, = }; C_s is a set of clocks to be reset at the execution of transition Tr_i.

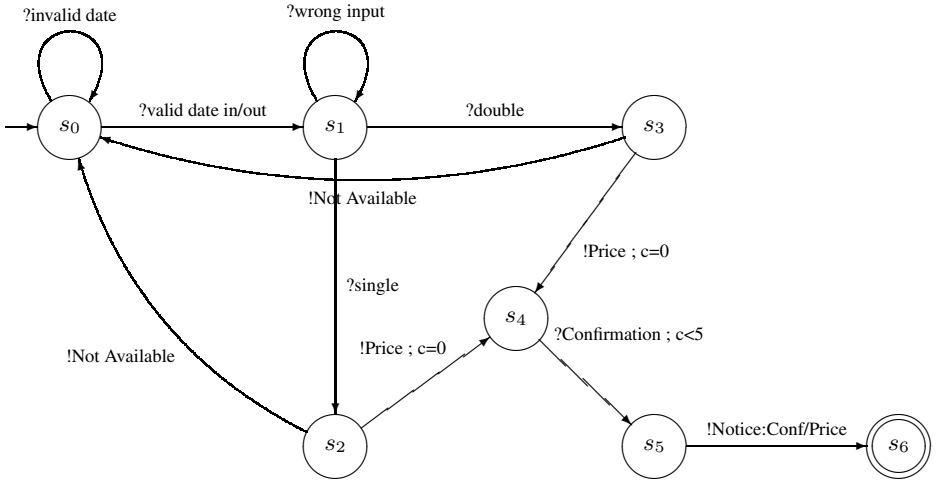

Fig. 3. An Example of TLTS representing simple hotel reservation

Figure 3 shows an example of TLTS representing a simple hotel reservation component (HR) with initial state s_0. A transition is represented by an arrow between two states and labeled by the action, the timing constraint and clocks to reset (action; EC; Cs). The TLTS in figure 3 is input-complete, if at state s_0 the user input an invalid date the system stays in s_0; otherwise, it moves to state s_1 where the users may choose either a *single* or a *double* room, thus, the system may move to either state s_2 or s_3. As soon as the appropriate input is selected the corresponding price is given, and clock c is set to zero in order to count the time for the conformation back from the user, then, the system moves to state s_4. If the conformation is not sent with in the time (c<5) the session will be timed-out.

4 Testing Methodology

Consider the web application illustrated in figure 2. In this work, this application is thought to be a Component Based system (CBS) that contains a set of interacting components (MC, HR, CR, WP), where the requirements of each component is already defined and represented as a TLTS. Assume that component HR will be fulfilled by a web service; therefore, we have to find all suitable web services, having similar functionality, that satisfy the requirements of HR and do not act errantly in our composed system (Figure 4), and then, link the selected services to our web application so that we can use any of them at invocation time.

This is usually done by searching the UDDI registry each time our system requires a web service. The UDDI registry holds the URL's and the corresponding WSDL specification of services that are published by the service providers. After selecting the "optimal" web service, our system binds to the service's web site and invokes the Web service. In this dynamic invocation model it may not be possible to know which web service will be used until run time [9]. Moreover, searching and testing web services

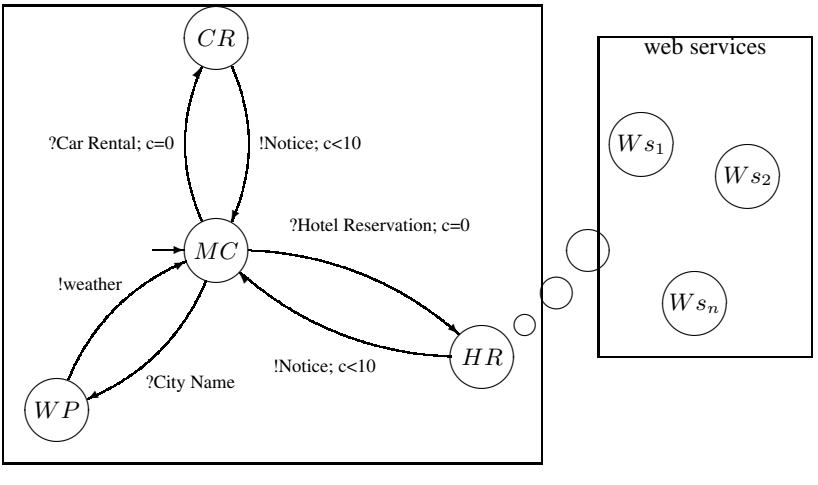

Fig. 4. An Example Web service oriented web CBS

whenever the system requires them would generate enormous network traffic and, still, may not find a suitable web service.

In our method, we suggest to select the web services during the development of our CBS. The method proceeds as follows: while building the CBS, if a component,HR, is implemented by a web service, the UDDI registry is searched and a set of WSDL files describing the candidate web services is found. Next, our task is to find all suitable web services and to eliminate all web services that does not satisfy the requirements of HR. Then, we test the selected web services to ignore all services that act errantly when integrated in our system. The new set of selected web services is saved into a log file linked to the component HR so that any of these suitable web services could be used later depending on its availability, without having to search the UDDI registry every time the service is needed. To reach our aim, we have to generate three sets of test sequencess. The first is used to select all adequate candidate web services that satisfies the requirements of our component. The second set is used to test the selected web services individually, and the third set is used to test the interaction of the suitable web services as a composed component in our web application. The generation of test sequences is detailed in section 4.2.

4.1 Testing Web Applications

Contrary to other testing techniques, our proposed testing method selects all suitable web services only once, during the testing of the web application, rather than selecting them each time the web application is invoked. This will help the developer to build a reliable and available web application. The links to all selected suitable web services are saved into a *log file* associated with the component to be fulfilled by the web service. The *log file* contains the *urls* of all suitable web services and the set of test cases used to test this component. Using this *log file*, the web application would have a wide range of finding available and suitable web services at invocation time. This method tests the web service individually (as a stand-alone component) and as a part of the web CBS. The method consists of four main steps described in the following algorithm:

Step 1: Search the UDDI registry for candidate web services. For each candidate web service found in the UDDI registry, we parse the WSDL file of web service under test (WSUT) to check whether the interface of this web service matches with the specification of our component. If the interface does not match, this process is stopped and we check another candidate web service in the UDDI registry; otherwise, we move into the second step.

Step 2: We connect to the web service's site and start testing the actual web service as a stand-alone component by sending SOAP messages generated from the **first set** of test cases, then we check the correctness of the information received as SOAP responses from the web service by matching them with the corresponding outputs in the test cases. If the web service does not pass this test, it is ignored and we start checking another web service; otherwise, we move to the third step.

Step 3: We continue testing the actual web service as a stand-alone component by sending SOAP messages generated from the **second set** of test cases, then we

check the correctness of the information received as SOAP responses from the web service by matching them with the corresponding outputs in the test cases. If the web service does not pass this test, it is ignored and we start checking another web service; otherwise, we move to the fourth step.

Step 4: We test the interaction of this web service as a component in our system by sending SOAP messages generated from the **third set** of test cases, then we check the correctness of the information received as SOAP responses from the web service by passing those outputs to the respective components in the web CBS and monitor the behavior of the whole system, taking into consideration the time restriction on responses. If the web service does not pass this test, it is ignored.

If the web service under test (WSUT) passes the four steps of the above algorithm, then the information -*(including the url, the first and second sets of test cases)*- about WSUT is saved into a *log file* associated with the component to be fulfilled by a web service. Next, this process is repeated until all candidate web services are tested. In any of the above steps, information about errors occurred during testing is saved in an *error log file* associated with the component under test. If non of the web services matches our component under test, then the error log file should be considered to modify the requirements of that component.

4.2 Test Case Generation

To make a decision about selecting a web service that fulfills a component, we have to (1) find all adequate candidate web services, (2) test those web services independently (as a stand-alone components) and select the ones that fulfill the functional requirements of our component, and (3) test the reliability of the selected web services' interaction as a part of our web component based system. Thus, test sequences are divided into three sets. In this work, the generated test cases use only symbolic values satisfying an adequate coverage criteria.

The first set, which tests the adequacy of the web service independently, is generated from information found in the WSDL file of the web service. In this set, test cases are generated based on boundary value testing analysis [3]. Traditional boundary value testing typically involved either boundaries in numerical data types such as integers, floating point numbers, or real numbers or else the end points of enumeration types. In this work, for numerical data types, the negative and positive numbers would be bounded by the limitations defined in the XML schemas: the most possible negative number, zero, and the most possible positive number. With string data types, the boundary values are maximum length and minimum length as defined in the XML schemas, and for boolean it is true and false. The generated test cases contains information about (a) the input boundary values to be sent to the web service and (b) the output boundary values to be used for validating the output received from the web service. To illustrate, we consider the following WSDL file that describes a web service for weather forecasting taken from [1]. It takes as an input the $CityName$ and returns the corresponding $Humidity$.

```
<types>
<xsd:schema targetNamespace="http://www.capeclear.com/AirportWeather.xsd" xmlns:SOAP-ENC=
"http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
</types>
<message name="cityName">
  <part name="arg0" type="xsd:string" />
</message>
<message name="getHumidityResponse">
  <part name="return" type="xsd:double" />
</message> ....
<portType name="Station">
<operation name="getHumidity">
  <input message="tns:cityName" />
  <output message="tns:getHumidityResponse" />
</operation> ...
```

Based on the boundary value analysis method, the **first set** of test cases for method *"getHumidity"* generated form the above WSDL file could rely on the following:

The Input argument is of type string:
maximum value: "00000000000000000000000000". minimum value: "null".
The Output is of type double:
maximum value: $2^{63} - 1$. minimum value: -2^{63}. zero: 0.

The second set of test sequences is used to test the behavior of the web service individually. Therefore, this set should be able to test the functionality of all possible actions in the service. Thus, we generate this set by traversing *all paths going from the initial state* of the TLTS representing the component to be fulfilled. To illustrate, consider Figure 3 that shows the TLTS for the HR component. Due to space limitation we only list three test sequences of the **second set** generated from the TLTS paths:

T1: <?invalid date;-;->
T2: <?valid date;-;->.<?single;-;->.<!Price;-;c=0>.<?Confirmation;c<5;-><!Notice;-;->
T3: <?valid date;-;->.<?double;-;->.<!Price;-;c=0>.<?Confirmation;c<5;-><!Notice;-;->

The third set of test cases tests the interaction of the web service as a part of our web CBS. Therefore, this set should perform a full-test coverage of the whole system. The whole system is covered by invoking all possible actions in the main web application as well as invoking the internal actions of the composed components. Thus, we generate this set by traversing *all paths going form the initial state* of the *TPG* representing the web CBS including the paths of the *TLTS* representing the inner actions of the composed components. To illustrate, consider figure 4 that shows the TLTS for the web CBS. A sample test sequences of the **third set** generated from the TLTS paths would be:

T4: <?HotelReservation;-; c=0>.<?invalid date;-;->
T5: <?HotelReservation;-; c=0>.<?valid date;-;->.<?double;-;->.<!Price;-;c=0>.
<?Confirmation;c<5;-><!Notice;-; c<10>
T6: <?HotelReservation;-; c=0>.<?valid date;-;->.<?single;-;->.<!Price;-;c=0>.
<?Confirmation;c<5;-><!Notice;-; c<10>
T7: <Weather Request;-c=0>.<?cityName;-;c=0>.<!WeatherInfo; c<10;->

A sample SOAP request and response message that would wrap the above test case (T_7) would look like:

```
    ...
SOAPAction: "http://www.myasptools.com/GetWeather"
    <?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetWeather xmlns="http://www.myasptools.com/">
      <cityName>Paris</cityName>
    </GetWeather>
  </soap:Body>
</soap:Envelope>
...

    <?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetWeatherResponse xmlns="http://www.myasptools.com/">
      <GetWeatherResult>
        <Humidity>70</Humidity>
      </GetWeatherResult>
    </GetWeatherResponse>
  </soap:Body>
</soap:Envelope>
```

In order to have an adequate state coverage, test sequences are generated by traversing, from the initial state, all paths of components' *TLTSs* (for the second set) and all paths of the *TPG* concatenated with the paths of the components' *TLTSs* (for the third set), still, we do not fear path explosion since both the web application and web service have a finite number of states to be covered by the test sequences and the test cases are assumed to be generated using only symbolic values for variables satisfying an adequate coverage criteria.

To assure a full-test coverage, test cases may use all possible values; however, this process is too expensive. In order solve the state explosion problem, for test execution we may use either, boundary values for variables, or we use a heuristic to choose values. However, in both cases, the test coverage will not be complete.

In the next section, we present the test framework that is used to implement our test method. It wraps the test cases with SOAP interfaces and validates the testing results back from the web services. It supports both execution and test scenario management.

5 Web Service Testing Framework

The framework that we use to test our web services is adapted from [13] with some modifications. It consists of two parts: test master and test engine (Figure 5).

The test master (1) extracts the interface information from the WSDL file and maps the signatures of the service into test scenarios, (2) extracts paths from the TLTS and maps them into test scenarios. The test cases are generated from the test scenarios in the XML format which is interpreted by test engine in the second stage. Test engine reads the test scripts produced by the test master and executes the test at the target web services, it also saves the execution trace into *log files* and sends the test results back to the test master. The actual test execution involves three phases:

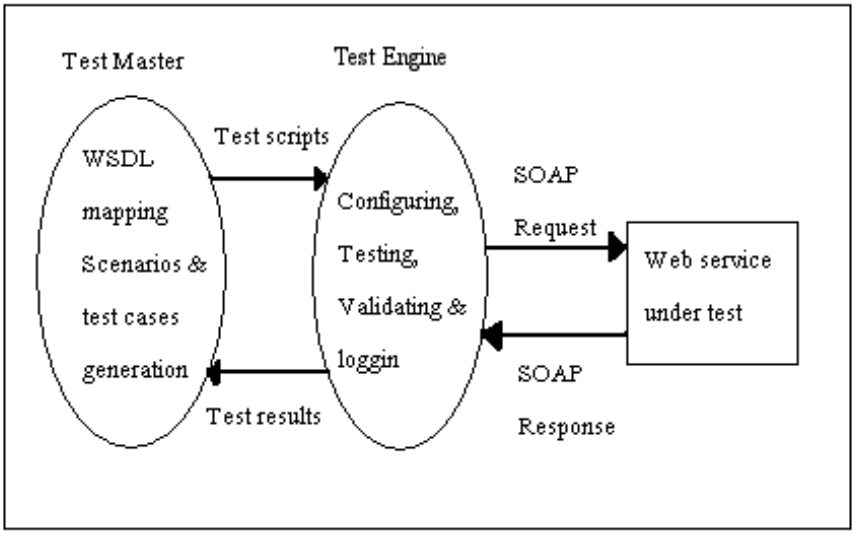

Fig. 5. Web Service Testing Framework

- Configuration: Configure the test scenarios from the WSDL and TLTS.
- Test: Generate the SOAP request messages, and invoke the particular service method with the respective input parameters.
- Validating: Check and assess the testing results in the SOAP response messages against the expected output specified in the test scripts, and save the suitable services in the log file.

6 Conclusion

This paper has presented a simple approach for building reliable web applications. A two-level abstract model is introduced to model the web application as a Task precedence graph (TPG) and the internal behavior of components as Timed Labeled Transition Systems (TLTS). One contribution of this paper is that it allocates all suitable web services that fulfill a component during the testing of the web application rather than during invocation time. This will give a wide range for rapid selection of available web services at invocation time. Another contribution is the full-coverage test cases that are generated form the WSDL files and the TLTS of the integrated components and the TPG of the whole web application. A third contribution is the Test-execution algorithm that generates two log files. The first log file contains all suitable web services that fulfill a component. The second is the error log files that contains information about non-suitable service that could be re-considered by the test architect. Finally, a testing framework is presented, it supports both execution and test scenario management.

Further research will focus on regression retesting method that may reveal any modification-related errors in the web application. We intend to implement our technique on a real industrial web application to prove its applicability. Moreover, a heuristic for test case selection will be studied.

References

1. ALTOVA. Web service description language for weather forecasting. In *www.altova.com*, November 2005.
2. T. Arnold. Testing web services (.net and otherwise). In *Software Test Automation Conference*, March 2003.
3. Boris Beizer. *Testing Techniques. Second Edition.* New York, VanNostrand Reinhold, 1990.
4. A. Milanova C. Fu, G. Ryder and D. Wonnacott. Testing of java web services for robustness. In *Proceedings of the International symposium on Software Testing and Analysis (ISSTA' 04) , July 11-14, 2004, Boston, Massachusetts, USA*, pages 23–33, July 2004.
5. J. Clune and L. Chen. Testing web services (methods for ensuring server and client reliability). In *Web Sphere Journal*, February 2005.
6. R. Heckel and M. Lohmann. Towards contract-based testing of web services. In *International Workshop on Test and Analysis of Component Based Systems, Bercelona*, March 2004.
7. N. Looker, M. Munro, and J. Xu. Testing web services. In *The 16th IFIP International Conference on Testing of Communicating Systems, Oxford*, 2004.
8. Vance McCarthy. A roadmap for web services management. In *www.oetrends.com*, November 2002.
9. N. Gold, C.Knight, A.Mohan, and M.Munro. Understanding service-oriented software. In *IEEE Software*, March 2004.
10. J. Offutt and W. Xu. Generating test cases for web services using data perturbation. In *Workshop on Testing, Analysis and Verification of Web Services. July 2004, Boston Mass.*, September 2004.
11. N. Davidson.The Red-Gate software technical papers. Web services testing. In *www.red-gate.com*, 2002.
12. IBM Web Services Architecture team. Web services overview. In *IBM*, 2004.
13. W. Tsai, R. Paul, W. Song, and Z. Cao. Coyote:an xml-based framework for web service testing. In *Proceedings of the 7th IEEE International Symposuim on High Assurance System Engineering*, October 2002.
14. W. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending wsdl to facilitate web service testing. In *Proceedings of the 7th International Symposium On High Assurance Systems Engineering*, 2002.
15. J. Zhang. An approach to facilitate reliability testing of web services components. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE' 04)*, November 2004.