Regression Testing Web Services-based Applications

Conference Paper · March 2006

DOI: 10.1109/AICCSA.2006.205085 · Source: DBLP

Abbas Tarhini
Lebanese American University
23 PUBLICATIONS **86** CITATIONS
SEE PROFILE

Hacène Fouchal
Université de Reims Champagne-Ardenne
113 PUBLICATIONS **403** CITATIONS
SEE PROFILE

Nashat Mansour
Lebanese American University
90 PUBLICATIONS **1,100** CITATIONS
SEE PROFILE

Regression Testing Web Services-based Applications

Abbas Tarhini (1) Hacène Fouchal (2) Nashat Mansour (3)

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

Abstract

Web applications can be composed of heterogeneous self-contained web services. Such applications are usually modified to fix errors or to enhance their functionality. After modifications, regression testing is essential to ensure that modifications do not lead to adverse effects. In this paper, we present a safe regression testing algorithm that selects an adequate number of non-redundant test sequences aiming to find modification-related errors. In our technique, a web application and the behavior of its composed components are specified by a two-level abstract model represented as a Timed Labeled Transition System. Our algorithm selects every test sequence that corresponds to a different behavior in the modified system. We discuss three situations for applying this algorithm: (1) connecting to a newly established web service that fulfills a composed web service, (2) adding or removing an operation in any of the composed web services, (3) modifying the specification of the web application. Moreover, modifications handled by the algorithm are classified into three classes: (a) adding an operation, (b) deleting an operation, (c) fixing a condition or an action.
Key-words : label transition systems, testing, verification, web service, web application.

1 Introduction

The development of web applications has 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 design applications that can span different operating systems, hardware platforms and geographical locations. Thus the design and the maintenance of reliable web applications and web services should be considered seriously.

Originally, web sites were constructed from a collection of web pages containing text documents and interconnected via hyper links. Recently, the dramatic evolution of web technology has led to web applications that can 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 challenges such as maintaining performance, reliability and availability of those systems. But the validation of such web applications remains a major 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 relevant web services from the pool of services found on the internet. With the increasing number of periodic publishing of web services, web applications need more often to be updated so that they select the most optimal and reliable service. Moreover, web systems are usually exposed to structural changes and modifications. These changes require us to retest the web application in order to provide confidence that the system functionality and unmodified parts have not been adversely affected by the modifications. Regression testing refers to selecting tests from the test suite generated during the initial development phase and to adding new tests to address enhancements and additions. One regression testing strategy is retest all which reruns every test in the initial test suite. This approach is normally very expensive and requires a lot of time. An alternative approach is to select a random subset of tests, which might be unreliable. Therefore, regression testing is a challenging task that should be both economic and reliable.

In this paper, we present a safe regression testing technique that is used to retest the web system whenever it is modified. First, we suggest a two-level abstract model to represent a web application. The generation of test sequences and test histories for the initial development phase is described in previous work [1]. Next, we present a regression testing technique that retests the modified web system with only the necessary test sequences selected from the test histories and provide confidence that modifications (including additions or deletions) are correct and have not adversely affected other parts in the system. In this work we do not deal with performance, we only target correct behavior. Moreover, the state explosion problem is handled by using the hierarchical two-level abstract model that represents the web application. With such a model, information handled by the tester at the first level of abstraction is different from that handled at the second level.

The rest of this paper is organized as follows. In Section 2 we present a brief background on web service models and our previous work done on testing web services. The modeling of web applications and needed definitions are presented in Section 3. In Section 4 we present a regression testing algorithm. We conclude the paper in Section 5.

2 Background

In this section, we present overviews of web services infrastructure and of regression testing.

2.1 Web Services Overview

Web Services were defined differently by vendors, researchers, or standards organizations. In our study, we define Web Services 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 a service broker (Figure 1). A service provider publishes services to a service broker. Service requesters find required services using a service broker and then bind to them [2]. 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).

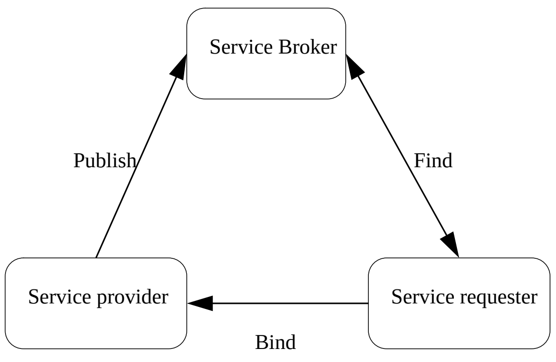

Figure 1: Web service architecture

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 transmits massages and data using XML wrapped in SOAP interfaces.

2.2 Regression Testing Overview

Regression Testing is a process which tests a system once again to ensure that it still functions as expected by specification [3]. The reason for this renewed testing activity is usually performed when changes are done to a system ω producing a modified version ω'. Regression testing is a way to test the modified version ω' using a test set T used previously to test the original system ω. The selection of suitable test cases from T can be made in different ways and a number of regression-testing methods have been proposed. These methods are based on different objectives and techniques, such as: procedure and class firewalls [4, 5]; semantic differencing [6]; textual differencing [7]; slicing-based data-flow technique [8, 9]; test case reduction [10, 11]; and safe algorithm based on program's control graph [12]. Typical regression testing procedures follow five main steps: (1) Identify the modifications made to ω; (2) test selection step: using the results in step 1, select $T' \subseteq T$, a set of tests that may reveal modification-related errors in ω'; (3) if necessary, create new tests for ω' and append to T'. These may include new functional tests required by changes in specifications, and/or new structural tests required by applicable coverage criteria; (4)run T' on ω', to provide confidence about ω''s correctness with respect to T'; and finally, (5) create T'', a new test set/history for ω'. Further, test histories should be maintained. The system's test history identifies, for each test, its input, output and execution history. An execution history consists of a list of components and their internal states exercised by the test. Moreover, [12] emphasized the creation of a test history for the test suite so that a regression test can be applied.

Rothermel and Harrold [12], presented a safe algorithm for regression test selection. It constructs control dependence graphs (CDG) for the original and modified program, then uses these graphs to generate tests.In fact, this algorithm selects all existing tests in a region when a simple change is done at the top level of the program. Moreover, this method does not identify parts of the program that should be covered by tests. These drawbacks were handled by the authors in a successive work Elaborating more on this algorithm, Rothermel and Harrold also implements and conducted empirical studies on several subject programs and modified versions [13]. The results suggests that in practice, the algorithms can significantly reduce the cost of regression testing a modified program.

Research on regression testing has focused on structured and object-oriented software and it is lacking for web services based software applications. In this work, we are concerned with the five steps of the regression testing approach mentioned before. Our regression testing algorithm selects test cases that identify modifications concerning the addition and deletions of web services to the system, and changes in the specification of the web system or any of its components. It also creates new functional tests required by changes in the specifications. Moreover, the algorithm updates the test history so that it maintains an acceptable test suite size.

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 models the internal behavior of each component in the system. This hierarchical model helps in minimizing the state explosion problem. In the following subsections we described each model and illustrate it with examples.

3.1 Web Application Representation

The functional behavior of a web system could be represented as a Task Precedence Graph (TPG). However, since we study web applications composed of components that interact by exchanging messages restricted by timing constraints, our first level of abstraction models a web applications as an input-complete Timed Labeled Transition System (TLTS), where each node in the TLTS is an abstract representation of a single component in the system that models the behavior of its software modules, and an edge joining two nodes represents the flow of actions (transitions) between components. Every edge is labeled with an action and its corresponding timing constraint. We formally define an TLTS as follows:

Definition 3.1 (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 $\sim \in \{ <, >, \leq, \geq, = \}$ or to a constant valued either true or false; C_s is a set of clocks to be reset at the execution of transition Tr_i.

Definition 3.2 (input-complete) *A* **TLTS** *M is said to be input-complete if all states accept any input $a \in A_I$. A TLTS will be input-completed by adding to each controllable state (a state whose action is only input) a loop labeled by all complementary actions in the input alphabet A_I.*

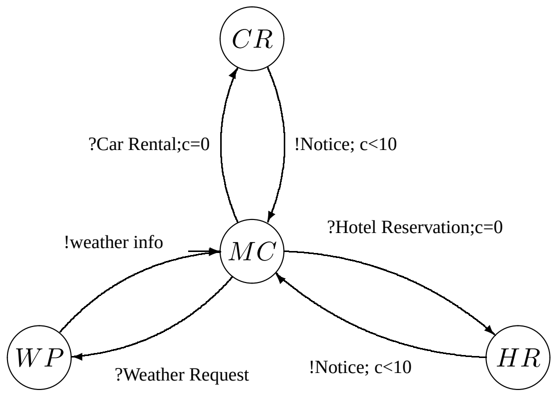

Figure 2: An example of TLTS representing a simple travel agency web application

Figure 2 illustrates a TLTS 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).

3.2 Single Component Representation

The second level of abstraction models every single component in the web application. In this level, we model each component as an input-complete 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.

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 because it accepts any input event in the input alphabet A; that is, if at state s_0 the user input an invalid date the system accepts the input but it 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

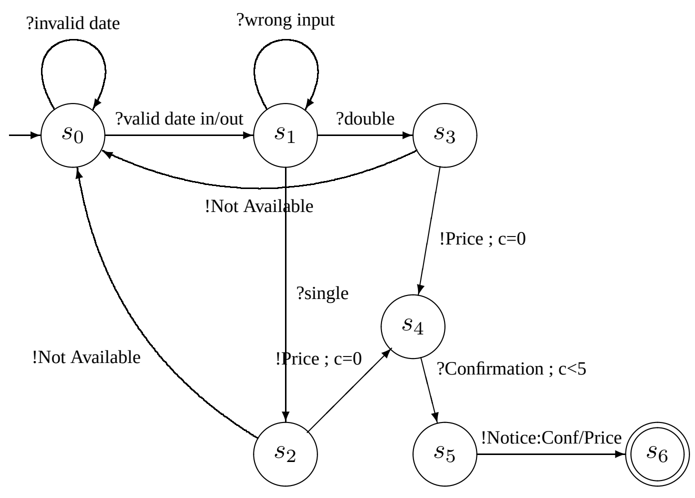

Figure 3: A TLTS representing simple hotel reservation

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 Regression Testing

Our regression testing technique for web services is based on a previous work [1] on testing web services. In the following subsections we briefly discuss our previous work on testing web services. Then, we present the regression test case selection algorithm.

4.1 Testing Web Services

In previous work [1], we presented testing technique that guarantees the availability of services in our web application. It selects and then associates all suitable web services to our web application before invocation time; moreover, it suggests testing the functionality of the web service integrated in the web application by executing three sets of test cases generated from (1) the WSDL files and (2) the specification of both the component fulfilled by a web service and (3) the specification of the whole 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 sequences used to test this component, it also contains a priority ranking number for each of these services. Using this *log file*, the web application would have a wide range of finding available and suitable web services at invocation time; further, this *log file* will be essentially needed for retesting this component after any modification. This method tests the web service individually (as a stand-alone component) and as a part of the web CBS.

The first set tests the adequacy of the web service independently. It is generated based on boundary value testing analysis [14] bounded by the limitations defined in the XML schema. The second set of test sequences is used to test the behavior of the web service individually. Thus, we generate this set by traversing all loop-free paths going from the initial state of the TLTS specification representing the component to be fulfilled. The third set of test sequences tests the interaction of the web service as a part of the web application. Thus, we generate this set by traversing all loop-free paths going from the initial state of the TLTS representing the web application including the loop-free paths of the TLTS representing the inner actions of the composed components. Next, test sequences and test histories are created. One test history is created for the web system; it consists of the third set of test sequences generated above, and an execution history for each test sequence. An execution history consists of a list of components and their internal states that experienced this test sequence. Other test histories are created for each composed component. Those test histories are attached to log files found in their corresponding component. A component's test history consists of only test sequences that experienced this component. After briefing our previous work, we will present the new work done on regression test case selection in the following sub sections.

4.2 Regression test case generation

Web application are subject to modifications. Modifications to one or more components might affect other components of an application, which might lead to errors. We classify modifications to web service based applications into the following types: (a) Type-1: integrating a newly established web service into the application, (b) Type-2: adding, removing or fixing an operation or a timing constraint in an existing component, (c) Type-3: modifying the specification (operations or timing constraints) of the web application. When modifications occur, we retest the modified system to gain confidence about its correctness. A TLTS for the modified version is constructed. It reflects all additions and/or deletions of states or edges performed by the modification. Each of these modifications together with needed examples are described in sections 4.2.2, 4.2.3, and 4.2.4.

Note that the second and third types of modifications are reflected in their corresponding TLTSs by adding and/or removing states and edges from that TLTS depending on the modification performed. Therefore, the modified TLTS should be input-completed (see Definition 3.2) so that it will not have any dangling edges in case of deletions; dangling edges does not allow generating executable test sequences.

Consider a web application ω and its modified version ω'. After any of the above modifications, we need to validate ω' by using the set of test sequences T used previously to test the web application ω. Thus, we firstly identify all modifications done to ω, then select a set of tests $T' \subseteq T$ that may reveal modification-related errors in ω'. Moreover, a new test set may be created to test required changes in the specification of the web application or any of its composed components. The algorithm for selecting the test set T' is shown in section 4.2.1.

4.2.1 Test Selection Algorithm

In this section we present a safe regression testing algorithm that selects the necessary test sequences believed to find modification-related errors. This algorithm is used for Type-2 modification, which includes (a) fixing a time condition or action, (b) removing an operation, or (c) adding an operation. It is also used for part of Type-3 modification. The algorithm is presented in Figure 4.

```
Algorithm: TestSelect
Input: TLTS for $\omega'$, Test set $T$. Test history.
Output: Test set $T'$. Updated Test history.
  • Step 1: Complete the TLTS for $\omega'$ to satisfy the Definition 3.2.
  • Step 2: Generate Test set $T'_{all}$ for $\omega'$ by traversing all acyclic
    paths of TLTS $\omega'$ from the initial state.
  • Step 3: Generate $T'$ and update the test history as follows:
      – All test sequences found in $T'_{all}$ and not found in $T$ are
        executed, that is, add to the retest set $T'$.
        Thus $T'$ = $T'_{all} \setminus T$.
      – All test sequences found in $T$ and not found in $T'_{all}$ are
        deleted from the test history.
      – All test sequences found in both $T$ and $T'_{all}$ are kept in
        the test history but not re-executed.
      – Add test sequences in $T'$ to the test history.
```

Figure 4: Algorithm to generate test set T'.

The input to the TestSelect algorithm presented in Figure 4 is the modified TLTS version for ω', the previously generated test set T for ω, and the test history. The output is a selected set of test sequences $T' \subseteq T$ that is believed to reveal modified-related errors if executed on ω', and the updated test history.

Given the input-complete TLTS for ω, we get the modified TLTS version for ω' by adding to (and/or deleting from) ω at least, (1) a new state s_i, and/or (2) an edge (s_i,s_j). The TLTS for ω' should be completed to satisfy the input-completeness definition (see Definition 3.2). In the examples below, we show why completing a TLTS is very important for $TestSelect$ algorithm to operate correctly especially if the modification is a deletion.

The set T' must include all test sequences that (1) traverse the newly added state in TLTS ω', (2) traverse all newly added edges in TLTS ω', and (3) traverse all edges with modified labels in the TLTS ω'. Thus, T' is generated by finding all acyclic paths in the input-complete TLTS of ω' and not in the input-complete TLTS of ω. The test set T' is able to experience any of the changes (fixing, deleting or adding of a timing constraint or an action) in the Type-2 and Type-3 modifications presented in Sections 4.2.3, and 4.2.4. To illustrate, we consider each of these changes separately:

(a) Fixing a condition or an action:

Consider the TLTS ω in figure 5. Assume the time restriction in the transition <!Notice;c<10;-> is changed to (c<5). thus, we get a modified version ω' with time restriction <!Notice;c<5;->. The set T generated from the original TLTS ω is:

T_1:	<?Hotel Reservation;c=0;->
T_2:	<?Car Rental;c=0;->.<!Notice;c<10;->
T_2:	<?Weather Request;c=0;->.<!Weather Info;-;->

The set T'_{all} generated from modified TLTS ω' is:

T'_{all_1}:	<?Hotel Reservation;c=0;->
T'_{all_2}:	<?Car Rental;c=0;->.<!Notice;c<5;->
T'_{all_3}:	<?Weather Request;c=0;->.<!Weather Info;-;->

Thus, the set T' = $T_{all} \setminus T$ is:

T'_1: <?Car Rental;c=0;->.<!Notice;c<5;->

(b) Removing (deleting) an operation:

Consider the original TLTS ω in figure 2. By deleting component HR we get the modified TLTS ω' in figure 5. We input-complete ω' by adding a transition loop labeled ($?HotelReservation; c = 0$).

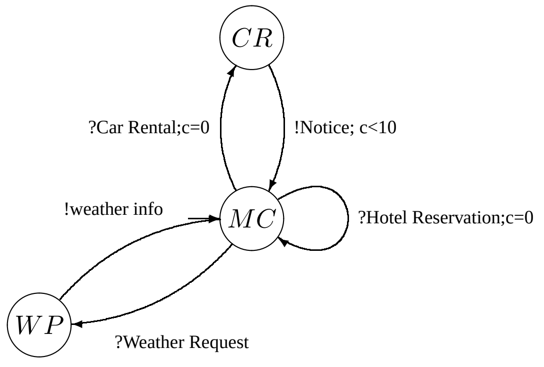

Figure 5: A modified TLTS for the original travel agency web application presented in figure 2

The set T generated from the original TLTS ω is:

T_1:	<?Hotel Reservation;c=0;->.<!Notice;c<10;->
T_2:	<?Car Rental;c=0;->.<!Notice;c<10;->
T_3:	<?Weather Request;c=0;->.<!Weather Info;-;->

The set T'_{all} generated from modified TLTS ω' is:

T'_{all_1}:	<?Hotel Reservation;c=0;->
T'_{all_2}:	<?Car Rental;c=0;->.<!Notice;c<10;->
T'_{all_3}:	<?Weather Request;c=0;->.<!Weather Info;-;->

Thus, the set T' = $T_{all} \setminus T$ is:

T'_1: <?Hotel Reservation;c=0;->

The test history will be updated as mentioned in the algorithm, a part of the updated test history is:

T_1:	<?Hotel Reservation;c=0;->
T_2:	<?Car Rental;c=0;->.<!Notice;c<10;->
T_2:	<?Weather Request;c=0;->.<!Weather Info;-;-> . . .

(c) Adding an operation:

Consider the original and modified web services shown in figure 6. The set T generated from the original TLTS ω is:

T_1: <?wrong input;-;->

T_2: <?cityName;-;->.<!humidity;-;->.<!weather Info;-;->

The set T'_{all} generated from modified TLTS ω' is:

T'_{all_1}: <?wrong input;-;->

T'_{all_2}: <?cityName;-;->.<!humidity;-;->.<!weather Info;-;->

T'_{all_3}: <?countryCode;-;->.<!humidity;-;->.<!weather Info;-;->

Thus, the set $T' = T_{all} \setminus T$ is: T'_1: <?countryCode;-;->.<!humidity;-;->.<!weather Info;-;->

Based on the illustrations presented above, note that algorithm "$TestSelect$" selects every test sequences that traverse only the modified parts in the system, and only non-redundant test sequences are re-executed. Moreover, it creates new test sequences to test added parts in the TLTS. This algorithm considers effects of additions, deletions as well as modifications in the web application. Thus, this algorithm is safe regarding the whole behavior, and it is precise since it selects only test sequences that exhibit a different behavior.

Moreover, the two-level hierarchical model helps in reducing the state explosion problem. In fact, by representing realistic web applications with such a model, the number of states to be covered in each TLTS will be very small; thus, the information handled by testers at each level of abstraction is different from that handled at the other level. Further, the generated test cases use only symbolic values for variables satisfying an adequate coverage criteria at test execution.

4.2.2 Modification type 1: connecting to new web services

In this type, the specification of the web system and composed components did not change; however, modifications occur when web services fulfilling composed components are updated; that is, a new web service is appended to other services, having the same functionality, that fulfill a component in the web system. Therefore, first we need to test this web service as stand-alone and if operates correctly when communicating with our component; further, we need to make sure it operate correctly when integrated in the web system. Thus, in this case regression testing is applied to retest the behavior of the updated component only; that is, selection of test cases should be only in the third set of test sequence. The three sets of test sequences needed to retest the modified web application ω' are created as follows:

- Generate the first set of test sequences from the corresponding WSDL of that service as described in section 4.1.
- The second set of test sequences is going to be used again as is, since we need to test the behavior of the newly added web service individually and when communicating with the updated component. This set is selected using the test history saved in the log file associated with this component.
- Using the test history of the web system, the third set of test sequences is selected from the previously generated test suite T of the web application. This set includes only test sequences that exercises the modified component. To illustrate consider figure 2, for example, assume a new web service for weather prediction is found and fulfills the component (WP). Using the web system test history, the newly generated third set consists of only one test sequence, as shown below, starting from the main component (MC) and passing through weather prediction component (WP):

  T': <?WeatherRequest;-;->.<?cityName;-;->.<!humidity;-;->.<!WeatherInfo; -;->

4.2.3 Modification type 2: modify a component's specification

In this type, the specification of a composed component is modified. A change can be either adding or removing an operation in a composed component. For example, assume a new operation that returns the weather information based on country code is needed. Thus the specification representing the weather prediction component is changed to handle the input $?countryCode$. The TLTS of the modified and original component specification is shown in figure 6.

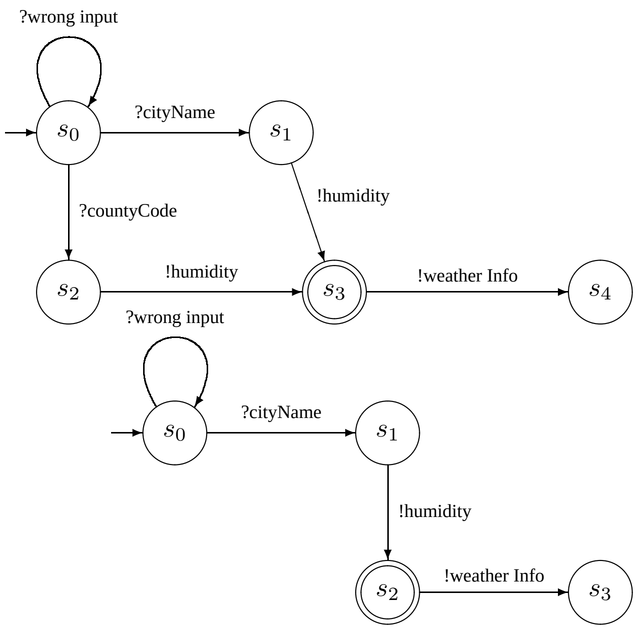

Figure 6: TLTS Original and Modified Weather Prediction web service

The three sets of test sequences needed to retest the modified web application ω' are created as follows:

- Generate first set of test sequences to test only the newly added operation (method) of that service. This is done by parsing the corresponding WSDL file of that service to get the data types and boundaries of the input parameters and output information. Thus, the above WSDL file is parsed and new test cases are generated only for operation $searchByCode$ as described in section 4.1.

- The second set of test sequences is generated using the algorithm $TestSelect$ presented in section 4.2.1. For example, the second set of test sequences for weather prediction web service shown in figure 6 will exercise only the newly added path in the TLTS as shown below:

  T': <?countryCode;-;->.<!humidity;-;->

- The third set of test sequences is also generated using the algorithm $TestSelect$ presented in section 4.2.1. It consists of all test sequences that exercise only the modified component composed in the web application. For example, consider the web application present in Figure 2 and the modified weather prediction component presented in Figure 6, the third set of test sequences, shown below, consists only of test sequences traversing the path invoked with input $?WeatherRequest$ in the web application and also traverses only the newly added path invoked with $?countryCode$ in the composed component: T': <?WeatherRequest;-;->.<?countryCode;-;->.<!humidity;-;->.<!Weather Info;-;->

4.2.4 Modification type 3: modify the web application specification

In this type, the specification of the web application is modified. A change can be either adding or removing an operation/component in the web application. To illustrate, consider the web application presented in Figure 2, assume a new component "credit card validation" (Ccv) is added to the web application so that the hotel reservation HR and car rental CR components can use it to validate credit cards. Figure 7 shows the TLTS of the modified web application and the TLTS of the component Ccv.

With this modification we need to coordinate between the newly added web service Ccv and the components HR and CR. This is done by searching the UDDI registry again to find the perfect match that coordinates between composed web services. The three sets of test sequences that make up test suite T' needed to test the modified web application are generated as follows:

- Generate first set of test sequences to test only the newly added web service. This is done by parsing the corresponding WSDL of that service to get the data types and boundaries of the input parameters and output information as described in section 4.1.

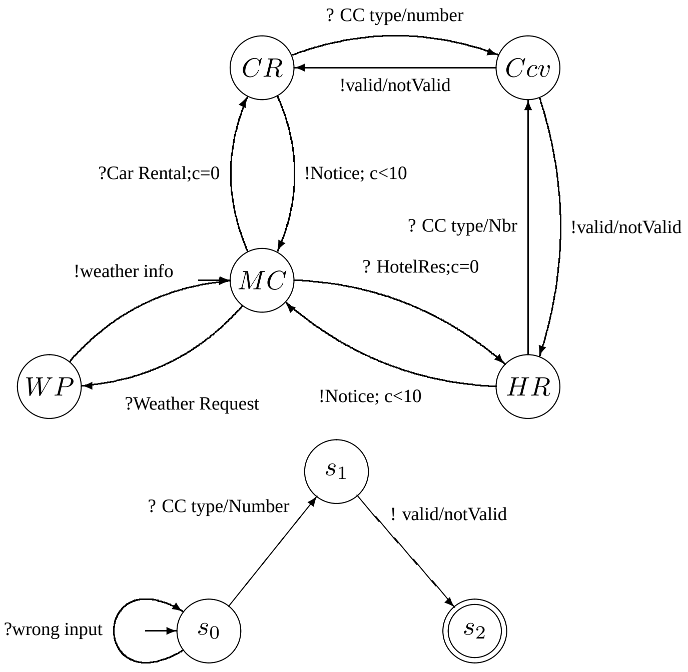

Figure 7: TLTS of the modified travel agency application and TLTS of the Ccv component

- The second set of test sequences is generated by traversing all paths starting from the initial state of the TLTS representing the newly added component. For example, the second set of test sequences for credit card validator web service generated from the TLTS shown in figure 7 looks like:

  T'_1: <?wrong input;-;-> T'_2: <?CC type/Number;-;->.<! valid/notValid;-;->

- The third set of test sequences is generated using the algorithm $TestSelect$ presented in section 4.2.1. It consists of all paths in the web application's TLTS that traverse all components affected by the new modification, including the inner paths of the TLTS representing those components. The TLTS of the composed hotel reservation HR and Credit card validation Ccv components is shown in figure 8. A sample of the third set for the modified web application shown in figure 8 would be:

T': <?HotelReservation;-; c=0><?valid date;-;->.<?double;-;->.<!Price;-;c=0>.<?Confirmation;c<5;->.
<?CC:type/Number;-;->.<! valid/notValid;-;->.<!Notice;-;c<10>

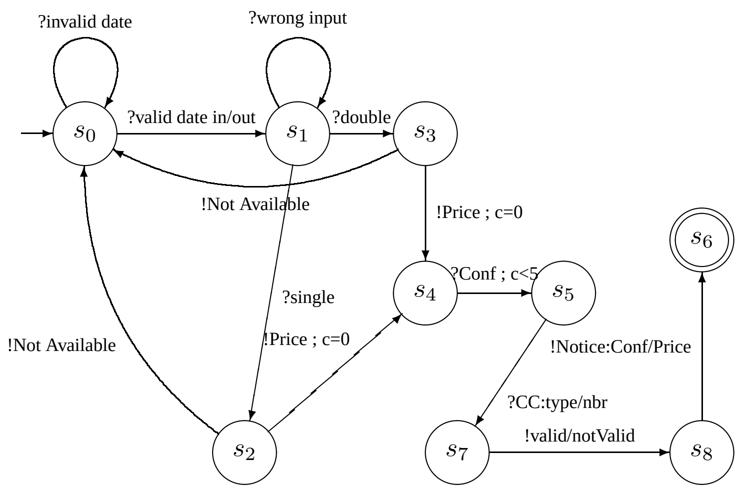

Figure 8: TLTS of the composed hotel reservation HR and Credit card validation Ccv components

5 Conclusion

We have presented a regression testing technique for retesting modified web applications. It selects only necessary test sequences needed to ensure the correctness of the modified system. In this work, A two-level abstract model is used to model the web application and the internal behavior of the composed components as a Timed Labeled Transition Systems (TLTS). Modeling with TLTS made it simple for the regression testing algorithm to select only modification-related test cases that retest the system. This algorithm is safe because it selects every test sequence that corresponds to a different behavior in the modified system. Moreover, the algorithm updates the test history by eliminating all redundant and non-used test cases; thus, the test history will remain acceptable in size. Three cases for applying this algorithm are discussed: (1) connecting to a newly established web service that fulfills a composed component, (2) adding or removing an operation in any of the composed components, (3) modifying the specification of the web application. Modifications handled by the algorithm are classified into three classes: (a) adding an operation, (b) deleting an operation, (c) fixing a condition or an action.

Further work will implement our technique on a realistic web application to prove its efficiency. We have undertaken to study some heuristics in order to select pertinent test sequences and to have a better test coverage .

References

[1] A. Tarhini, H. Fouchal, and N. Mansour. A simple approach for testing web service based applications. In *5th international conference on Innovative Internet Community Systems, I2CS 2005, Paris-France*, June 2005.

[2] IBM Web Services Architecture team. Web services overview. In *IBM*, 2004.

[3] The Information Security Glossary. Regression testing. In *www.yourwindow.to/information-security/gl_regressiontesting.htm"*, November 2005.

[4] H. Leung and L. White. A firewall concept in both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance*, pages 262–271, 1992.

[5] P. Hisa, X. Li, D.C. Kung, C.T. Hsu, Y. Toyoshima L. Li, and C. Chen. A technique for the selective revalidation of oo software. In *Journal of Software Maintenance 9*, pages 217–233, 1997.

[6] D. Binkley. Semantic guided regression cost reduction. In *IEEE Transactions on Software Engineering 23 (8)*, pages 498–516, 1997.

[7] F.I. Vokolos and P.G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proceedings of the 3^{rd} International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pages 3–21, 1997.

[8] R. Gupta, M. J. Harrold, and M.L. Sofa. Program slicing-based regression testing techniques. In *Software Testing, Verification and Reliability 6 (2)*, pages 83–111, 1996.

[9] G. Rothermel, M. J. Harrold, and J. Dehia. Regression test selection for c++. In *software. J. Softw. Testing, Verif., and Rel. 10(2)*, June 2000.

[10] N. Mansour and K. El-Fakih. Simulated annealing and genetic algorithms for optimal regression testing. In *Journal of Software Maintenance, Vol. 11, 19-34.*, June 1999.

[11] N. Mansour and R. Bahsoon. Reduction-based methods and metrics for selective regression testing. In *Information and Software Technology, Vol 44, No. 7.*, pages 431–443, 2002.

[12] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the conference on software maintenance. CA*, pages 358–367, September 1993.

[13] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. In *ACM Trans. Software Engineering and Methodology, 6(2)*, pages 173–210, 1997.

[14] B. Beizer. *Testing Techniques. Second Edition.* New York, VanNostrand Reinhold, 1990.