Abstract—Modern software systems have often the form of Webservice compositions. They take advantage of the availability of a variety of external Webservices to provide rich and complex functionalities, obtained as the integration of external services. However, Webservices change at a fast pace and while syntactic changes are easily detected as interface incompatibilities, other more subtle changes are harder to detect and may give raise to faults. They occur when the interface is compatible with the composition, but the semantics of the service response has changed. This typically involves undocumented or implicit aspects of the service interface.

Audit testing of services is the process by which the service integrator makes sure that the service composition continues to work properly with the new versions of the integrated services. Audit testing of services is conducted under strict (sometimes extreme) time and budget constraints. Hence, prioritizing the audit test cases so as to execute the most important ones first becomes of fundamental importance. We propose a test case prioritization method specifically tailored for audit testing of services. Our method is based on the idea that the most important test cases are those that have the highest sensitivity to changes injected into the service responses (mutations). In particular, we consider only changes that do not violate the explicit contract with the service (i.e., the WSDL), but may violate the implicit assumptions made by the service integrator.

Keywords—Webservice Composition, Audit Testing, Test Prioritization.

I. INTRODUCTION

As Webservices standards are becoming more and more mature, the adoption of Webservices is growing fast in many areas, from business integration and social media, to end-user applications. There are more and more service providers who offer Webservices facilities, e.g. Web API, REST[1] or SOAP[2] Webservices, allowing service consumers to easily access, integrate, and share their business data. At the time of writing this paper, according to the site Programmable Web¹, there are about 2,500 APIs available and about 5,500 Mashup systems that exploit those APIs. These numbers are increasing constantly.

Webservices, however, evolve quickly to meet technology evolution and business changes. For instance, eBay released 8 versions² of their finding service within each year, from 2009 to 2010. Similarly, Amazon released 12 versions³ of their Elastic Compute Cloud service per year. Changes are new business features, bug fixes, and performance improvements. As a result, service integrators face frequently a critical decision either to update the service composition to the new versions of the services they are exploiting, or staying with the old ones knowing that the old services might have issues, risks, and limited support. More often than not, the former is the preferred choice. Sometimes, the old service is dismissed, hence migration to the new service is mandatory.

Audit testing aims at assuring that the new versions of the services work properly with the current service composition.

In the web services context, audit testing aims at checking the compliance of a new service, including a new version of an existing service or a newly-discovered service from a new provider, with a system under test that integrates the service and currently works properly. Audit testing is a form of regression testing [3], [4], since the goal is ensuring that the pre-existing functionalities are preserved when the new versions of the service are used. As such, it shares with regression testing the problems of: (1) test case selection; (2) test case minimization; and, (3) test case prioritization. In the context of service compositions, test case selection accounts for determining which test cases are actually invoking the changed service. Test minimization aims at obtaining a minimal subset of the selected and minimized test suite that preserves some adequacy criterion (e.g., coverage). Test prioritization aims at finding an ordering of the test cases so that the most important (e.g., likely fault revealing) test cases are executed first.

Audit testing of services differs from traditional regression testing because the testing budget is typically much more limited (only a small subset of the selected and minimized test suite can be executed) and because the kind of changes that are expected in the service composition is known and

¹See http://www.programmableweb.com

³http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryIDs=86
quite specific. In fact, audit testing is used to check the proper functioning of a service composition when some external services change. Some adaptations of the service composition that are required to make the composition work are trivially detected at the syntactical level, by the compiler which will immediately report any interface change that needs adaptation on the composition side. It is only those semantic changes that do not affect the service API description (for example WSDL[5]) that may go undetected and require audit testing. We take advantage of this narrow model of the changes to be regression tested in order to define a novel test case prioritization method, specifically tailored to the Web service compositions. It is based on the idea that the most important test cases are those that can detect mutations of the service responses, when such mutations affect (possibly) the semantics, while leaving the WSDL syntax unchanged.

Let $s$ be the service under consideration, $s_{\text{new}}$ be a candidate that can substitute $s$. $s_{\text{new}}$ is the subject of audit testing. Let $TS$ be the set of available test cases that are selected as candidate for audit testing. These are the test cases whose executions result in the invocation of $s$. $TS$ is used in audit testing of $s_{\text{new}}$ with respect to the composition under consideration. In practice, execution of the full suite $TS$ might involve too much time or might require a big monetary budget, if only a limited number of invocations of $s_{\text{new}}$ is available for testing purposes (i.e., invocations in test mode), and/or if the invocation of $s_{\text{new}}$ requires payment. Hence, the service integrator has to minimize and prioritize the test cases from $TS$ that are actually run against $s_{\text{new}}$ during audit testing. In fact, often only a small subset of $TS$ will be executed, due to time as well as monetary constraints. It is thus important that the most critical test cases are executed first. The goal is then to prioritize the selected test cases in such a way that issues, if any, are detected as early as possible, in order to allow the system integrator to start debugging and fixing them soon.

In this paper we address the problem of test case prioritization during audit testing of services, by introducing a novel prioritization approach that determines the change sensitivity of the test cases and uses this metrics to rank the test cases, from the most sensitive to service changes to the least sensitive one. Change sensitivity measures how sensitive a test case is to changes occurring to the service under consideration. The rationale underlying this approach is that new versions of services may produce service responses that are still compliant with the service interface (WSDL), but violate some assumptions made (often implicitly) by the service integrator when building the service composition. Thus, test cases that are more sensitive to these changes are executed first.

Specifically, we have defined a set of new mutation operators and we apply them to the service responses to inject artificial changes. Our mutation operators are based on a survey of implicit assumptions commonly made on Web service responses and on manual inspection of those parts of the service API which is described only informally (usually by comments inside the WSDL document). After applying our mutations, we measure change sensitivity for each test case by comparing the outputs of each test case in two situations: with the original response, and with the mutated responses. A change is detected if the behavior of the service composition differs when the original service is used as compared to the mutated response. Similarly to the mutation score, the change sensitivity score is the number of changes detected (mutants killed) over all changes.

The remainder of the paper is organized as follows. Section II discusses our approach. In Section III we introduce a preliminary evaluation of our approach applied on a Webservice case study. Section IV concludes this work and discusses our on going research on the same problem. Finally, Section V surveys related work on regression testing and test prioritization of Webservice compositions.

II. Change Sensitivity Measure

In the context of Webservices, service clients communicate with services via message passing: clients send requests to services and receive responses. During an audit testing session at the client side (see Figure 1) test inputs are provided to the client system, which, then, sends requests to services, and receives and processes the responses. Finally, the outputs from the system are evaluated (the test oracle takes usually the form of assertions, the default assertions being that the application should not crash or raise exceptions). Since the client lacks controllability and observability over the service, and SLA (Service Level Agreement) [6] concerns only high-level quality contracts (e.g. performance, response time), the developer of the client (service integrator) has to make assumptions about technical details regarding the format of the responses. We call these assumptions as service integrator’s assumptions. For example, the integrator might expect a list of phone numbers, separated by commas, from the service, when searching for a phone number. Changes in the data format of the response (e.g., using colon instead of comma as phone number separator) may break the assumptions of the service integrator, which may lead the client to misbehave, thus making the test cases not to pass.

![Figure 1. A test session at a service client](image-url)
It is worth noticing that we focus on data changes (e.g., format, range, etc.). Changes regarding the structure of the service responses can be easily detected, because they require the interface definition of the service, written for instance in the Web Service Definition Language (WSDL) [5], to be changed. Adopting a new service interface involves rebinding and recompiling, and the compiler is able to detect syntactic incompatibilities. What the compiler cannot detect is (subtle) semantic incompatibilities (such as the change of a list item separator). This requires regression (audit) testing and test prioritization.

In our approach, service integrators specify their service assumptions explicitly, in order to simplify and automate audit testing of integrated services. For the specification of the integrator’s assumptions, we propose an XML based assumption specification language. A service assumption consists of an XPath reference [7] to an element in the response under consideration and it can include data restrictions regarding that element. Data restrictions have the same format as those defined in the W3C XML Schema [8]. Service assumptions specify what a client expects from a service for its own purposes. Therefore, the data restrictions specified by one service integrator may differ from those in the service definition (e.g. in the WSDL interface of the service) or from those specified by another integrator. We refer the reader to our technical report [9] for more information regarding the structure and benefit of integrator’s service assumption definition.

The listing in Figure 2 shows three examples of service assumptions. The first one, lines 1÷6, says that the client expects the length of the title to be between 5 and 256. The second one, lines 7÷13, specifies the possible values of conditionId. The last one, from line 14, constrains the timeLeft element to a regular expression. The restrictions defined in these examples can be used to validate the values of the elements of any XML response (e.g. the SOAP message) that matches the corresponding XPath expressions.

The mutation operators that we propose use service assumptions to inject artificial changes into service responses. The changed responses (also called mutated responses) and the original ones are, then, used to measure change sensitivity of test cases. In the following sections we discuss mutation operators and how change sensitivity is measured.

### A. Mutation Operators

Based on a survey of service assumptions commonly made on Web service responses and on manual inspection of those parts of the service interface which is described only informally, usually by annotations or comments inside the WSDL document, we identified 9 main data restriction types and their corresponding mutation operators, showed in Table I.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>Enumeration restriction limits the content of an XML element to a set of acceptable values. Using this restriction <em>Enumeration</em> operator generates a new value which is not accepted by the restriction to replace the original one.</td>
</tr>
<tr>
<td>Length</td>
<td>Length restriction limits the precise length of the content of an XML element. Using this restriction <em>Length</em> operator generates a new content having its length differs from the required one.</td>
</tr>
<tr>
<td>MaxLength</td>
<td>Length restriction limits the length of the content of an XML element to be inferior than a specific value. Using this restriction <em>MaxLength</em> operator generates a new content having its length greater than the allowed one.</td>
</tr>
<tr>
<td>MinLength</td>
<td>Length restriction requires the length of the content of an XML element to be greater than a specific value. Using this restriction <em>MinLength</em> operator generates a new content having its length smaller than the allowed one.</td>
</tr>
<tr>
<td>MinInclusive, MaxInclusive, MinExclusive, MaxExclusive</td>
<td>These restrictions are specified on numeric types, e.g. double, integer. The corresponding mutation operators generate new numeric numbers that are smaller or greater than the acceptable minimum or maximum values.</td>
</tr>
<tr>
<td>RegEx</td>
<td>Regular expression restriction requires the content of an XML element to follow a specific pattern. <em>RegEx</em> based operators change slightly the original regular expression and generates new values based on the mutated expression. We propose 6 <em>RegEx</em> based operators, their detailed descriptions are in Table II.</td>
</tr>
</tbody>
</table>

Table I

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
</table>

The *Enumeration* mutation operator randomly generates a new item, added to the finite set of items admitted in the response according to the service assumption. The *Length* mutation operator changes the size of a response element, so as to make it differ from the integrator’s assumptions. Similarly, the *MaxLength* and *MinLength* mutation operators make the size of a response element respectively greater than or lower than the limit admitted in the integrator’s assumptions. When the numeric value of a response element is constrained to be within boundaries, the *MinInclusive, MaxInclusive, MinExclusive, MaxExclusive* mutation operators produce values that lie beyond the integrator’s assumed boundaries. The *RegEx* mutation operators can be used when the content of a response element is supposed to follow a pattern specified by means of a regular expression. Such regular expression is mutated (e.g., by replacing a constant character with a different one; by making a mandatory part of the expression optional; by concatenating an additional subexpression) and mutated responses are generated by means of the mutated regular expression, see Table II. For example, the regular expression specifying a list of phone numbers as a list of comma separated digit sequences can
be mutated by replacing the constant character ',', with ':'.

<table>
<thead>
<tr>
<th>RegEx Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegEx-CO</td>
<td>Concatenate the specified regular expression with a simple expression from a library of common regular expressions, e.g. [0-9], [a-zA-Z].</td>
</tr>
<tr>
<td>RegEx-RA</td>
<td>Change a range specification in the specified expression, e.g. [0-9] changed to [1-9].</td>
</tr>
<tr>
<td>RegEx-MO</td>
<td>Change the specified expression, substituting mandatory specification with optional specification.</td>
</tr>
<tr>
<td>RegEx-OM</td>
<td>Change the specified expression, substituting optional specification with mandatory specification.</td>
</tr>
<tr>
<td>RegEx-RE</td>
<td>Mutate the specified expression by removing one part of it, e.g. remove the head or tail specification ∧, $.</td>
</tr>
<tr>
<td>RegEx-SW</td>
<td>Mutate the specified expression by swapping the position of its parts.</td>
</tr>
</tbody>
</table>

Table II

**REGEX BASED MUTATION OPERATORS**

As examples of new content generated by the proposed mutation operators, the second example in listing 2, lines 7:13, shows an enumeration restriction. The value of conditionId must be Used, New, or Unspecified. Applying Enumeration operator we can get a new value, e.g. a random string Uesn, for conditionId in the mutated response. The last example in listing 2 specifies that the element timeLeft has to obey this expression: P[0-9]{2}DT[1-2]D[0-9]H[1-5]?[0-9]M[1-5]?[0-9]S, example of the content of timeLeft: P0DT0H0M13S, By applying RegEx-CO mutation operator, the expression can be changed by concatenation with a simple expression, e.g. [0-9], the new content can be: P0DT0H0M13S0.

Taking a data restriction specified in a service assumption as an input, the corresponding mutation operator can generate new data that are not acceptable according to the service assumption. This is to simulate the fact that when we plan to substitute s with s_{new}, s_{new} may have data that violate the current service integrator’s assumptions.

### B. Measuring Change Sensitivity

The measuring of change sensitivity is divided into three steps: (1) executing the test set TS and monitoring service request and response messages; (2) generating mutated responses based on the monitored responses and service assumptions; (3) for each test case, comparing the behavior of the composition when receiving the original response and when the mutated ones are received. Then, the sensitivity (mutation) score is calculated.

Since we are dealing with audit testing, when the system is deployed and running, many testing activities have been done already, before deployment, including some online testing of the composition against the actual, external services integrated in the composition. As a result, the first step of measuring change sensitivity can be skipped and we can reuse the previously monitored request and response messages.

Since the original responses (each is one XML document) are monitored and stored locally, at step 2, we apply the mutation operators discussed in Table I (based on the service integrator’s assumption specifications) to generate mutated responses. For each response and for each assumption, we select a mutation operator based on the type of restriction specified. Then, the XPath is used to query the elements in the response to be injected with new contents generated by the mutation operator. Eventually, we obtain a set of N mutated responses, each of them containing one change.

At step 3, each test case in TS is executed offline against
the $N$ mutated responses. Instead of querying $s$ to get the response, the system receives a mutated response and processes it. In this way we can conduct test prioritization without any extra cost for accessing $s$, if applicable, and without any time delay, due to the network connection with external services. The behavior of the system reacting to the mutated responses is compared to its behavior with the original response. Any deviation observed (i.e., different outputs, assertions violated, crash or runtime exceptions reported) implies the change is detected (or the mutant is killed). Change sensitivity of test case $tc_i$ is measured as the proportion of mutants that are killed by each test case:

$$CS(tc_i) = \frac{\#\text{response changes detected}}{N}$$

The change sensitivity metrics is then used to prioritize the test cases. Only those at higher priority will be run against the real, external services, during online audit testing. The monetary and execution costs associated with such kind of testing make it extremely important to prioritize the test cases, such that the top ranked ones are also the most important, in terms of capability of revealing actual problems due to service changes that are not managed at the purely syntactic level. We conducted a case study to assess the effectiveness of our prioritization method in terms of fault revealing capability, under the assumption that the resources for online audit testing are extremely scarce, hence the fault revealing test cases should be ranked very high in the prioritized list produced by our method, in order for them to have a chance of being actually executed during real audit testing.

### III. Preliminary Results

We apply the proposed approach on a service-based application, namely eBayfinder, to evaluate its effectiveness. (eBayfinder was developed by our group internally for research purposes.) The application uses XML-based Webservice standards to access to eBay Finding APIs, eBay Shopping APIs, and Google Map APIs. It allows users to search for items listed on eBay, to filter search results, e.g. by condition: Used, New, and to visualize the location of the items found on a Google Map layer. Figure 3 shows the interface of the application.

The application is Web-based, in that the Web browser is used to send queries and to visualize the results in a Web page. However, thanks to Google Web Toolkit (GWT), most of the client code is written in Java (hence it can be analyzed like any other Java program). The size of the application is about 20K lines of code and its test suite includes 159 test cases. The test suite for eBayfinder has been defined based on the functional adequacy criterion (black box testing). Every functionality offered to the end users (according to the application’s requirements) has been exercised by at least one test case. When a functionality can be executed under different configurations or with different parameters, we defined a test case for each variant that provides a distinct usage mode to the end users.

We focused on the service eBay Finding APIs used by eBayfinder. By analyzing its release notes we identified some changes of the new service releases that could potentially impact our service composition. We injected 5 faults, which relate to the changes, into the application. The details of the faults are described in Table III. These faults can be detected by test cases that challenge the features of the application that are impacted by the changes.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>eBay changed condition specification from “Used, New, Unspecified” to a new system: “1000, 2000, 3000, etc.”. In the transition period both systems are supported. However, from April 2011, if the request uses the old condition filter, the result will be null, which may give raise to a fault in the composition. eBay Doc v1.5 states: “To sort by bid count, you must specify a listing type filter to limit results to auction listings only (e.g., itemFilter.name=ListingType &amp; itemFilter.value=Auction)”. In previous versions, there is no such a requirement. The result will be null if this listing type is not specified and the user selects to sort by bid count. Such a null value may give raise to a fault in the composition.</td>
</tr>
<tr>
<td>F2</td>
<td>As of January 15, 2010, eBay evaluates the CONTENT-TYPE header against the request payload format. If the CONTENT-TYPE value is incorrect for the payload, the request will fail. Previously, eBay did not validate the CONTENT-TYPE header in requests, hence the composition did not need to perform strict checks on it. This may give raise to faults in the composition.</td>
</tr>
<tr>
<td>F3</td>
<td>eBay introduced new Store with new Global-ID, which is not recognized by the application. This leads to a problem in processing store information and search results.</td>
</tr>
<tr>
<td>F4</td>
<td>eBay introduced new Store with new Global-ID, which is not recognized by the application. This leads to a problem in processing store information and search results.</td>
</tr>
</tbody>
</table>

### Table III

| Faults injected into eBayFinder |

Regarding service assumption specification with respect to the eBay Finding APIs, we analyzed the WSDL document of the service and the actual responses received from eBay to define a specification of service assumptions. These assumptions and the monitored responses from the service are used to generate mutated responses to measure the change sensitivity score, $CS$, of each test case. Finally, we obtained a sorted list of test cases based on $CS$. Using the top 15 test cases from this list, out of 159, we detected 4 faults F1, F2, F3, and F4.
F3, F4, F5; only fault F2 was missed, the reason being that there is no test case that specifies the required inputs.

We further evaluated our approach by comparing it with coverage-based prioritization, which favors test cases that cover more code. We analyzed statement coverage of each test case in order to rank them. Then, we took the 15 top test cases from the coverage-based sorted test case list. As a result, 3 faults F3, F4, F5 were detected; coverage-based prioritization cannot detect faults F1 and F2.

By analyzing the detected faults and the test cases selected by both techniques (sensitivity-based and coverage-based prioritization) we found that F3 can be detected by any test case, so prioritization is unnecessary for it. F4 and F5 are located in a code portion that requires a specific sequence of execution steps. The test cases that detect these two faults must exercise all such steps in sequence and thus will have high coverage scores. For this reason, coverage-based prioritization performs well in these cases. Our sensitivity-based technique can detect F4 and F5 as well. Moreover, it can detect F1 with a test case that has high sensitivity score but low coverage, in this case we see that our technique offers an advantage over coverage-based prioritization. In fact, a low coverage test case that is highly sensitive to the eBay response changes has more chances to reveal F1 than a high coverage test case that exhibits little dependence on the specific response values and format.

We developed a framework for monitoring Webservices responses and measuring the change sensitivity score of test cases. We will use this framework for conduct further research on this topic and we plan to make it an open source tool for the community.

IV. CONCLUSION AND FUTURE WORK

Webservices represent an enabling technology for many business integrations, social media, and end-user applications. However, as Webservices evolve quickly and their technology allows for runtime discovery and binding, they demand much effort from the service integrators to assure the compliance of a new service, including a new version of an existing service or a newly-discovered service from a new provider, with the system under test, which integrates the service and is currently working properly.

We proposed a novel approach to the prioritization of test cases for audit testing of Webservices. Our approach embodies a new set of mutation operators that use service assumptions to produce changes in service responses. Similarly to mutation score, we introduced the change sensitivity score, a measure that shows the sensitivity of a test case to changes in service responses. Test cases that have high score, i.e., that are more sensitive to the service response changes, are executed first.

We conducted a preliminary evaluation of the approach on a service composition that integrates eBay and Google
maps services. This application consists of roughly 20K lines of code. The results show that using only a subset of around 10% of the available test cases (those top-ranked by our approach), 4 over 5 injected faults can be detected. Our technique discovers one more fault than coverage-based prioritization technique does with the same number of test cases. This is a quite encouraging result.

In the future we will further evaluate the approach on bigger size and industrial Webservice compositions. We will compare it with other approaches, including coverage-based prioritization [4].

V. RELATED WORK

In this section we survey state-of-the art work on regression testing, test selection and prioritization in the context of Webservices and Webservice composition. Some work on the prioritization of test cases for service composition shares the same research objective like ours in this paper. However, our approach based on change sensitivity of test cases is completely novel. We have done a preliminary evaluation of the proposed approach and compared it with code-coverage-based prioritization. A thorough comparison with existing approaches in the same Webservice composition/integration context will be conducted in our future work.

A. Webservice Regression Testing

Ashok et al. [10] provide a four-step approach to test WSDL changes: (i) a WSDL is analyzed to detect methods and parameters; (ii) input values for the identified parameters are extracted from a database; (iii) based on those inputs, soap messages are created and sent to the server (in which the service runs); and (iv) the server responses are analyzed, according to a Golden Response, to detect changes.

In Khan et al. [11], a model is used to describe the interfaces of services, in particular, the model describes: components/operations and data model, the usage protocol (by using a state machine for describing service interactions and message in input/output), specifications of operations (by using OCL pre/post conditions [12]), and specifications of the process (by activity diagrams enriched with object flow information). From such a model, different types of test cases can be derived, e.g., considering the data or considering the protocol. When a service (or a composition of services) is changed, its model needs to be changed. Models describing interfaces of services before and after the change are compared in order to analyze the evolution of a system, identifying which tests need to be rerun and where new ones are required.

Bruno et al. [13] and Di Penta et al. [14] propose the use of test cases as a contract between a service provider and a service user. In their proposal, the testing suite needs to be published from the provider (in terms of: XML-encoded test cases and QoS assertions) as a part of the service specification. The user can run periodically the suite to discover changes in the service. In [14], they propose also an idea (i.e., the use of monitoring data to mimic service behavior during testing) to reduce the overall amount of service requests during a testing phase.

B. Webservice Test Selection and Prioritization

Ruth et al. [15] and [16] defined a safe Regression Test Selection (RTS) technique for service-based composition. A three-step approach is followed: (i) a simulation tool is used to convert the (non-local) services, involved in a composition, in local Java code; (ii) dynamic analysis is then performed using the composition (with “mock” objects) to derive its Java Interclass Graph (JIG), a sort of dynamic control-flow graph; and (iii) the graph is analyzed to determine the minimum set of test cases that need to be rerun, according to a coverage criterion. A JIG is constructed for each service and an overall JIG is constructed for the composition. When the composition changes, its JIG is derived and the old JIG is compared with the new one by performing a dual-traversal of the two graphs. Changes detected in the graph are considered “dangerous points”. Given a test suite for the composition, the test case selection is performed by considering the discovered dangerous points and coverage information regarding the test cases on the graph. Note that [15] focuses on the simulation part of such a testing technique, while [16] focuses on the test cases derivation from the JIG.

In Tarhini et al. [17], the composition is modeled by means of a Timed Labeled Transaction System (TLTS). The TLTS-based model for a composition is a two-layer model in which: (i) the first layer represents the behavior of the whole composition, in terms of interactions between services; (ii) the second layer represents the behavior of each service, in isolation, with respect to the composition. When a composition changes, a TLTS-based model of the new composition is constructed. According to the type of the composition changes (three possible changes are considered: connection to a new service, modification of the service specification, modification of the composition), some nodes, edges and conditions of the initial TLTS model can be changed/removed/added. By traversing the model and evaluating its changes, a set of test cases focused on the changed parts are derived. The focus is to derive non-redundant test cases focused on detecting modification-related faults.

In Wang et al. [18], given a service composition expressed by means of BPEL: (i) an eXtensible BPEL [19] Flow Graph (XBFG) is extracted. The XBFG is a control flow graph that mainly describes the control-flow of the composition process. It contains calls to the external services used in the composition. (ii) the graph is traversed to extract paths that can be converted into test cases for the composition; and, (iii) when a composition changes, its XBFG changes. In this last step, in the case of static binding of services, by detecting differences in the graphs (before and after the
changes), a regression test suite is derived focusing on those paths that exercise the changes in the initial graph. In the case of dynamic binding, a table with the possible service bindings is additionally considered in the test selection.

Hou et al. [20] distinguish off-line testing (in which actual services are simulated) from on-line testing (in which the actual services are used), in this second type of testing all the limitations (e.g., request quota equals to the upper limit of the number of requests) imposed by the service provided impact the testing process. Examples of limitations are: 60k requests per user per month, 60k requests per IP, 60k queries per day per API, 60k queries per license key per day. They propose to consider request quotas in test case prioritization. The proposed testing technique is based on three steps: (i) time slot partitioning (the time is subdivided in subsequent slots according to the time slot of the services under testing); (ii) test case selection and prioritization (for each time slot test cases are selected and then prioritized with the aim of maximizing the testing coverage per slot); (iii) information refreshing (after each test selection and prioritization, the information about the executed test cases, the achieved test coverage, and the remaining quota are recomputed for better setting the testing with respect to the next time slot).

Mei et al. [21] and [22] propose a multi-level coverage model for testing. They use coverage information coming from three sources: (1) business process; (2) XPath; and, (3) WSDL, for prioritizing test cases in regression testing of service-based compositions. They have shown how to prioritize test cases according to the process workflow is not always sufficient since it does not take into account information related to services integrated by means of messages. Thus, they propose to start in prioritizing test cases from the process coverage. When test cases reveal similar process coverage capability, the coverage of the XPath expressions are considered (since, usually, XPath expressions are used to manage the XML-based messages sending/coming to/from services, i.e., the message based interactions). Again, when test cases reveal similar XPath coverage, they propose to distinguish them by considering their coverage of the WSDL schema elements (since the exchanged messages need to be conform to WSDL specification). According to this overall model, they define: (i) several coverage criteria (e.g., number of covered process branches, number of covered XPath branches) subdivided for each considered level; and (ii) two main approaches, summation and refinement, for composing the coverage information coming from the three different levels and defining a unique and overall coverage index for a given test cases (rather than having three different indexes, each one for each level). Note that [22] focuses on the overall approach while [21] focuses only on the use of tags embedded in XML-based messages, exchanged when integrating services, for doing test prioritization. Basically, in [21] they study the use of WSDL tag coverage information computed according to input and output messages associated with regression test cases for prioritizing them.

C. Mutation Testing for Webservice Composition

E. Botaro et al. [23] discuss and evaluate a set of mutation operators used for improving test cases for Webservice composition written in WS-BPEL [19]. A series of experiments was conducted, aiming to determine how selective mutation operators are for the qualification of test cases and, more generally, to assess the quality of the operators themselves. This work shows a novel application of mutation testing to Webservices and discusses the potentiality of mutation testing in the area of service engineering.

ACKNOWLEDGMENT

This work has been partially funded by the European Union FP7 project FITTEST (grant agreement n. 257574).

REFERENCES


