Abstract—Web services evolve frequently to meet new business demands and opportunities. However, service changes may affect service compositions that are currently consuming the services. Hence, audit testing (a form of regression testing in charge of checking for compatibility issues) is needed. As service compositions are often in continuous operation and the external services have limited (expensive) access when invoked for testing, audit testing has severe time and resources constraints, which make test prioritization a crucial technique (only the highest priority test cases will be executed).

This paper presents a novel approach to the prioritization of audit test cases using information retrieval. This approach matches a service change description with the code portions exercised by the relevant test cases. So, test cases are prioritized based on their relevance to the service change. We evaluate the proposed approach on a system that composes services from eBay and Google.

Keywords—Webservice composition, audit testing, test case prioritization.

I. INTRODUCTION

Services evolve quickly to meet technology and business changes. Service providers may optimize the performance (e.g., changing data format from XML to JSON), or they may develop new features to meet new requirements. For instance, eBay released 8 versions\(^1\) of their eBay Find service within each year, from 2009 to 2010. Similarly, Amazon released 12 versions\(^2\) of their Elastic Compute Cloud service per year. As a result, service integrators face frequently a critical decision either to update their service compositions 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 services are dismissed; hence migration to the new services is mandatory.

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 [1], [2], 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 test cases 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, and when the time budget for testing is limited, only less important test cases are skipped.

In this paper, we proposed a novel approach to the prioritization of test cases for audit testing of service compositions. Our approach takes advantage of (1) the availability of service change descriptions, often in release note documents of the services that are changed; (2) the connection by means of binding code from service compositions to services; (3) recent advances of information retrieval (IR) techniques. In our approach, traces collected during past test case executions are analyzed to extract an identifier document, containing the terms that appear in the identifiers of the executed methods and functions. Then, queries are defined based on service change descriptions. Information retrieval tools perform the matching between queries and identifier documents to search for the most relevant test cases for each change. We evaluated our approach with the help of a case study. Results show that IR-based prioritization techniques perform better than coverage-based prioritization.

The remainder of this paper is organized as follows. After the introduction, Section II describes some characteristics of service compositions relevant for test prioritization and motivates our new technique. Section III presents our approach. Section IV describes our evaluation of the proposed approach on a case study. Finally, the last two sections are about related works and conclusion.

\(^{2}\)http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=86
II. Motivation

A. Prioritization challenges in service testing

Several testing challenges descend from the intrinsic features of the services that are integrated in a SOA (Service Oriented Architecture) service composition. Services have low accessibility and observability. The reduced testability of external services, together with the costs associated with invoking them for testing purposes, make the constraints on the number of test cases that can be reasonably executed much stricter than with traditional software. Hence, the typical problems of regression testing (namely, test case selection, minimization and prioritization) are exacerbated in the context of audit testing of services and demand for adhoc, novel solutions.

Test case selection with respect to the modification traversing test cases is not possible in audit testing of external services, because of the lack of observability of the service execution. Coverage of the service achieved by each test case is usually unknown, hence so is also coverage of any changed code portion of the service. Test case selection is hence limited to just disregarding the test cases that do not invoke the changed service. However, such a selection criterion is extremely weak. Test case prioritization also suffers the problem of the lack of coverage information of the service execution, techniques that are based on coverage have only coverage information of the service composition, not the external services. They might have issues with specific service changes. Furthermore, prioritization techniques have problem with the peculiar properties of service compositions, which are discussed in what follows.

B. Service composition

Two important characteristics of service compositions are globalization and personalization. Service compositions are globalized in that they aggregate data and functionality provided by different services, from different domains and locations. They are personalized in that they serve different types of users, taking into account their specific profiles: the “when” and “where” of their requests. A popular example of service compositions is travel service portals, eDreams.com for instance, that access many other service providers, e.g. flights, hotels, cars, trains, cruise ships, and then provide a service to the users, customizing it by location, language and preferences. For example, as clients of eDreams in Italy, we can access the Italian train ticket system, while when we are in UK, we do not have such a possibility.

With regards to input and output, we can characterize a service composition in terms of multiple input and output channels. Input channels of a service composition are connections to resources and data of other service providers, e.g. car rental services, hotel services. Output channels are connections by which the service composition serves different users, satisfying their demands. Since the input channels connect to different providers from different business domains, and the output channels serve different users with their own profiles, such channels are often independent and their aggregation is transparent to the specific instance. Hence, problems with one channel tend to have no or little effect on some of the others. For example, a service composition may have problems with a channel to a particular car rental service, while the other channels to other services are still good, so that only specific output channels are affected: there may be problems only with the class of users who share certain information, e.g. renting cars from the same location; the other users are not affected.

The following is a specific example that illustrates this characteristic. A travel service composition aggregates information from car, hotel, flight services and provides a search interface to the end-users. Its pseudo-code is shown in Listing 1.

Listing 1. Excerpt of service composition pseudo-code

```java
BackEndComponent{
1. synCarData(UK, ITALY, ...)
2. synHotelData(UK, ITALY, ...)
3. synFlightData(UK, ITALY, ...)
}

FrontEndCarInterface{
4. results = queryCar(country, keywords)
5. handler = getCarHandler(country)
6. handler.handle(results)
}

FrontEndHotelInterface{
7. results = queryHotel(country, keywords)
8. handler = getHotelHandler(country)
9. handler.handle(results)
}

FrontEndFlightInterface{
10. results = queryFlight(country, keywords)
11. handler = getFlightHandler(country)
12. handler.handle(results)
}
```

According to the channel view, this composition has three input channels that connect to a car, a hotel, and a flight service to get business information about cars, hotels, flights in two countries, UK and ITALY. It provides also three output channels for users to query information. Now, if the car service changes, most of the other services work fine. This change might affect the users who query for cars. Moreover, since the car results are handled differently with respect to different countries (lines 4–6), it is likely that only users of a specific country might be affected, or users from different countries are affected differently.

When a service changes, audit testing is performed. Usually, not all existing test cases can be run during every audit testing session, because of the involved costs (both monetary and computational). Test prioritization aims at
INFORMATION RETRIEVAL FOR TEST PRIORITIZATION

Information retrieval (IR) techniques have been used in many areas of software engineering, including program analysis and comprehension. One of the main uses of IR is to rank a collection of text documents with respect to a given query. The most relevant (i.e. similar) documents that match the query are ranked first. Given a service change description and a set of available test cases, we prioritize them using IR according to the following steps:

- **Step 1: Trace Extraction**, traces of previous test executions (e.g. collected during past integration or regression testing), are used to extract identifier documents. Identifier documents include terms extracted from the identifiers appearing in the executed code. Specifically, terms are extracted from identifiers appearing in method signatures, variable names, class names. Moreover, during execution, the composition under test might send requests to other services. These requests also contain identifiers that are extracted and appended to the identifier documents. After this step, we obtain a set of identifier documents, each associated with a test case.

- **Step 2: Querying**, once the set of extracted identifier documents representing the test cases is available, we apply IR to search for relevant test cases with respect to the change description. A change description or important keywords extracted from it play the role of the input query for IR. IR tools will return a ranked list of identifier documents from which we can identify the corresponding test cases.

Identifier documents are extracted from traces automatically, using program instrumentation. Logging code is injected into the system source code or byte code automatically using available tools (e.g. ASM\(^3\)), or Java 1.5+.

During test execution the injected logging code traces the

---

\(^3\)http://asm.ow2.org
classes/methods that are instantiated/invoked. As another option, we can extract identifier documents from the coverage reports of test cases. Coverage reports contain the lines of code covered by each test case. We can then extract identifier documents from the text source of those lines. Terms included in identifier documents are obtained by splitting the identifiers into composing terms (e.g., resorting to naming conventions, such as camel-casing and use of underscore), filtering the terms (so as to remove stop-words, such as articles and prepositions) and stemming (i.e., reducing each word to its root; e.g., from plural to singular).

Table I shows a real-world change description in version 1.7.0 of the eBay Finding Service\(^4\). The description is about the condition specification of items listed on eBay, and condition ID value 3000 is going to be replaced by value 2750. From the description, we can define a query like Item Condition ID 3000 to search for test cases that are the most relevant to test a system with respect to this service change.

<table>
<thead>
<tr>
<th>Change description: Item condition ID value 3000, Like New, will be deprecated around the end of September 2010 for media categories, such as Books or DVDs &amp; Movies. It will be replaced by item condition ID value 2750. eBay will update existing items listed with condition ID value 3000 to use condition ID value 2750, but there may be a transition period of up to a month.</th>
<th>Query: Item Condition ID 3000</th>
</tr>
</thead>
</table>

Table I

AN EXAMPLE OF QUERY

A. Query definition guidelines

The query of a search is a key factor that drives its results. Good queries combining with good information retrieval tools result in good results, if there are; while bad queries might obtain irrelevant results. Query definition involves human factor and the quality of change description, this could we a weak point of our approach. Regarding the task of defining queries to apply our technique, we propose the following guidelines that work in general cases:

- GL1 Consider mainly technical keywords related to the changes from the change descriptions.
- GL2 Give priority to the technical keywords that we use in the code, if we have such knowledge.
- GL3 Gradually adjust the query, focusing on important keywords first and then adjusting it by considering more or less keywords to obtain relevant results.

B. Information retrieval tools

The Vector Space Model (VSM) is one of the most widely used models in IR [4]. A text document is represented as a vector of weights, each weight representing the occurrence of a term in the document. Weights can indicate just the frequency of occurrence (TF: Term Frequency), or may involve more complex calculations, such as TF-IDF (IDF: Inverse Document Frequency). Both the documents being ranked and the query are presented as vectors of weights:

\[
d_j = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})
\]

\[
q = (w_{1,q}, w_{2,q}, \ldots, w_{t,q})
\]

The ranking score of document \( j \) is then computed based on the angle between its vector \( d_j \) and the vector of the query \( q \), and cosine of the angle is used as the numeric score assigned to document \( j \). The documents are sorted depending on their scores. LSI (Latent Semantic Indexing) builds on top of VSM, by mapping each vector from the original term-document space to a space with reduced dimensionality, which (automatically) gets rid of noisy information and keeps only the relevant dimensions. Singular value decomposition is used for this purpose.

There exist many IR tools that improve VSM and LSI. Among them, we selected Apache Lucene\(^5\) as the first tool to prioritize test cases. It is a high-performance, full-featured text search engine library and it is suitable for nearly any application that requires full-text search. We selected Google Desktop\(^6\) as the second tool. It is developed by Google and provides searching functionality for files and documents on local computers.

IV. CASE STUDY

This section discusses the results obtained when prioritizing the test cases for a service composition case study, in the presence of changes of the external services.

A. Evaluation Metrics

There might be different prioritization goals, including maximizing the rate of fault detection or the rate of code coverage. We focus on the rate of fault detection, a key performance indicator for audit testing of service compositions. Different techniques sort test cases differently. According to our goal, those that reveal faults earlier are regarded as superior ones.

We use the fault matrix to measure the rate of fault detection. A fault matrix shows the relationships between test cases and faults, i.e., each test case’s capability of revealing each fault. Given the test case prioritization produced by a specific technique (e.g., additional-coverage), from the fault matrix we can compute the detection index of each fault. The detection index of a fault is the rank index of the first test case that reveals the fault (smaller means the fault is detected earlier, hence is better).

It should be noticed that the detection index metrics can be computed only once full information about the


\(^5\)http://lucene.apache.org

\(^6\)http://www.google.com
fault detection capability of each test case (i.e., the full fault matrix) is available. When test case prioritization is conducted in the field, the fault matrix is of course unknown, hence the detection index cannot be computed. However, in our experimental setting we executed all test cases in advance, so as to obtain the full fault matrix, in order to have the information necessary for a comparative evaluation of the alternative prioritization techniques considered in this study.

B. Service Composition – Application

We use a service-based application, called eBayfinder, in order to evaluate the effectiveness of different prioritization techniques. eBayfinder was developed by our group internally for research purposes, in part to address the lack of publicly available service compositions (including source code and test cases).

The application uses XML-based Webservice standards to access to eBay Finding APIs\textsuperscript{7}, eBay Shopping APIs\textsuperscript{8}, and Google Map APIs\textsuperscript{9}. It allows users to search for items listed on eBay stores\textsuperscript{10}, to filter and compare search results, e.g. by condition: Used, New, and to visualize the locations of the items found on a Google Map layer.

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)\textsuperscript{11}, 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 160 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. It is worth to mention that this application has been used in \cite{5}, however the test suite used here has been improved in terms of coverage and with more test cases.

We focus on the service eBay Finding APIs used by eBayfinder. We have analyzed the release notes\textsuperscript{12} of this eBay service and identified some real changes that could potentially bring the service composition eBayfinder to failure. Then, we manually injected artificial faults into the code of eBayfinder, so as to simulate a context in which the service change actually results in a failure of the composition.

As discussed in Section II, faults may be generic (i.e., affecting all input-output channels) or profile-specific (i.e., affecting only specific combinations of input and output channels). We conducted two separate studies to evaluate the effectiveness of different test case prioritization techniques in these two contexts. Specifically, in the generic faults study, we manually injected faults whose locations are executed independently of the user profile settings. In the profile-specific study, faults are injected only in code portions devoted to the management of users that have a specific configuration of their profile (e.g., their location being a specific country). Hence, only one particular input-output channel is affected by the fault, in the latter case.

C. Study 1: Generic Faults

We used 5 changes from the release notes of eBay Finding service to inject 5 possible faulty lines of code into eBayfinder’s code. We executed all 160 test cases against eBayfinder with the current version of the service, i.e. without changes, and then performed coverage-based (COV), additional-coverage-based (A-COV) and IR-based (LUCENE, GOOGLE) test prioritization. COV and A-COV use statement coverage to rank test cases. COV sorts test cases based on how many lines of code are executed (covered), while A-COV sorts them based on how many additional lines of code are covered by test cases, i.e., for a test case we consider only newly covered code, not already covered before. We used and extended EMMA\textsuperscript{13}, a free Java code coverage tool, to do coverage analysis.

Since COV and A-COV use only coverage information regardless of the changes, both techniques produce only one ranking. On the contrary, since IR-based techniques take into account the change description, each change gives rise to a corresponding ranking.

For the IR-based technique, we extracted identifier documents for each test case. In this study, we distinguish two types of identifier documents. The first type, called CM-I, contains Class and Method signatures covered by a test case, and terms extracted from those signatures. For example, if a test case executes a method called sendRequest() belonging to a class named Handler, the CM-I identifier document of the test case will contain the following content: \{\texttt{Handler, sendRequest, send, request}\}. The second type, called CMR-I, contains not only covered classes and methods, it contains also the Request contents sent by the service composition during the execution of the test case. The following is an example of CMR-I identifier document: \{\texttt{Handler, sendRequest, send, request, FindingRequest, Finding, Request, Keyword, Canon, SortBy, Sort, By, BidCount, Bid, Count}\}. We investigated also two types of query: long queries (the exact change description), and short queries

(manually shortened version of the description, containing only the main keywords).

Table II gives the details of the 5 change descriptions and of the short queries defined according to these descriptions. The first change, **F1**, is about a new filter used to define more precise eBay queries. The second change, **F2**, specifies the only eBay stores where sorting search result by bid count is supported. The third change, **F3**, mentions a new requirement for the request messages sent to eBay. The forth change, **F4**, specifies a new list of eBay stores. Finally, the last change, **F5**, is about the condition name of items listed on eBay. Based on these changes, we injected 5 generic faults into the code of eBayfinder.

Table III shows the result of this study. There is one test case (with the highest coverage score) that can detect three faults (F3, F4, F5). The detection indexes of these faults are, thus, 1 for COV and A-COV. IR-based techniques (LUCENE/CM-I, GOOGLE/CM-I, LUCENE/CMR-I, GOOGLE/CMR-I) perform well on these faults as well, they all found the same test case, even when queries are defined differently for each fault. COV performs badly at F1 and F2, because the test cases that detect these faults have low coverage. A-COV, which takes into consideration only additional code coverage, outperforms COV with reasonably good detection indexes. LUCENE/CM-I and GOOGLE/CM-I cannot find any test cases, because F2 is at a location for which all CM-I identifier documents have no keyword or term related to the query. However, if the terms appearing in the requests sent from eBayfinder during test execution are included in the identifier documents (i.e. IR-based techniques on CMR-I identifier documents), LUCENE and GOOGLE perform very well also on F2, resulting in the best detection index.

In summary (see Table III), when applied to generic faults, IR-based prioritization using LUCENE and CM-I (with short queries) perform equally well as A-COV and much better than COV. We have also evidence of one instance where IR-based prioritization outperforms A-COV (F2, LUCENE, CMR-I, short queries). In general, these results are quite encouraging, because the faults considered in this study are generic, i.e., they are those for which existing, state-of-the-art techniques are expected to perform well, since they affect all combinations of input-output channels (hence coverage is a good indicator of priority for them). We conclude that these faults can be revealed equally well by state of the art, coverage-based (A-COV) and by IR-based (LUCENE/CMR-I) techniques.

In addition, we also performed tests with IR-based techniques using long queries (the exact change descriptions shown in Table II). The results, shown in Table IV, indicate that the performance is worse than IR-based techniques with short queries. Therefore, we recommend the use of IR-based test prioritization with short queries extracted from change descriptions.

### Table II

<table>
<thead>
<tr>
<th>Changes and Queries</th>
<th>COV</th>
<th>A-COV</th>
<th>CM-I</th>
<th>CMR-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>eBay filter values</td>
<td>62</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>eBay store ids</td>
<td>116</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>eBay global ids</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eBay country ids</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eBay market ids</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table III

<table>
<thead>
<tr>
<th>Detection Index with Generic Faults: Short Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>F1</td>
</tr>
<tr>
<td>F2</td>
</tr>
<tr>
<td>F3</td>
</tr>
<tr>
<td>F5</td>
</tr>
</tbody>
</table>

### Table IV

<table>
<thead>
<tr>
<th>Test Prioritization</th>
<th>COV</th>
<th>A-COV</th>
</tr>
</thead>
<tbody>
<tr>
<td>eBay filter values</td>
<td>62</td>
<td>2</td>
</tr>
<tr>
<td>eBay store ids</td>
<td>116</td>
<td>9</td>
</tr>
<tr>
<td>eBay global ids</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eBay country ids</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eBay market ids</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The results, shown in Table IV, indicate that the performance is worse than IR-based techniques with short queries. Therefore, we recommend the use of IR-based test prioritization with short queries extracted from change descriptions.
D. Study 2: Profile-Specific Faults

In this study, we focus specifically on the separation characteristic of service compositions described in Section II. From some real changes of the eBay Finding service, we derived 8 artificial faults that impact the way search results are handled by 8 specific eBay stores (EBAY-IT, EBAY-US, EBAY-DE, etc.). 42 test cases that target specific eBay stores have been also selected from the initial set of 160 test cases.

<table>
<thead>
<tr>
<th>Fault</th>
<th>COV</th>
<th>A-COV</th>
<th>LUCENE &amp; GOOGLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6</td>
<td>34</td>
<td>6</td>
<td>I</td>
</tr>
<tr>
<td>F7</td>
<td>33</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>F8</td>
<td>17</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>F9</td>
<td>9</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>F10</td>
<td>16</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>F11</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>F12</td>
<td>26</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>F13</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table V Detection index with profile-specific faults; short queries

Table V shows the detection index results of COV, A-COV, and IR-based techniques (using CMR-I and short queries). The results of LUCENE and GOOGLE are the same and, therefore, are put in one column. With the set of faults, F6÷F13, located in specific code portions dedicated to specific eBay stores, COV has quite poor performance, while A-COV is a bit better, but both are substantially worse than IR-based prioritization. If we have a budget of 5 test cases (out of 42) to be executed, COV and A-COV might have a chance to detect 1 (F13) and 2 (F13, F8) faults respectively. IR-based techniques, using a specific query for each specific change (e.g., global id EBAY-US), return the best detection index, 1, in all cases. If we consider the budget of 5 test cases again, all 5 faults are detected by IR-based techniques.

This study has shown that, with regard to the separation of channels characteristic of service compositions, IR-based techniques substantially outperform both COV and A-COV, i.e., the state-of-the-art, coverage-based test case prioritization techniques.

V. RELATED WORKS

In this section we present the state-of-the-art works on test selection and test prioritization in the context of Web services. Several works propose techniques to test service compositions, however, our approach based on IR techniques is completely novel. In fact, no work exists, at least to our knowledge, that considers the content of the change descriptions to prioritize/select test cases for Web services.

Ruth et al. [6], [7] defined a safe regression test selection technique for service-based composition. A three-step approach is followed: (i) a simulation tool is used to convert the (non-local) services used in a composition, in local Java code [6]; (ii) dynamic analysis is then performed using the composition to derive its Java Interclass Graph (JIG), a dynamic control-flow graph; and (iii) the JIG is used to determine the test suite [7], 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 to detect changes. These changes are “dangerous points” to be tested, thus they are considered to perform the test case selection on the old test suite and suggest the need for new test cases.

In Tarhini et al. [8], 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 interactions among the used services; and (ii) the second layer represents the behavior of each service, in isolation. When a composition changes, a TLTS-based model of the new composition is constructed. By traversing the model and evaluating its changes, a set of test cases focused on the changed parts is derived.

In Wang et al. [9], given a service composition, expressed by means of BPEL, an eXtensible BPEL[10] 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. However, when a composition changes, its XBFG needs to be changed. The XBFG graph is traversed to extract paths that can be converted into test cases for the composition. In this last step, in 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 case of dynamic binding, a table with the possible service bindings is additionally considered in the test selection.

Hou et al. [11] 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 provider impact the testing process. 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); (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 to better schedule test case execution in the next time slot).

Mei et al. [12], [13] 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 that prioritizing 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 prioritizing test cases from the process coverage. When test cases have similar process coverage, the coverage of the XPath expressions are considered (usually, XPath expressions are used to manage the XML-based messages sending/coming to/from services). Again, when test cases have similar XPath coverage, they propose to distinguish them by considering their coverage of the WSDL schema elements.

VI. CONCLUSION AND FUTURE WORK

Web services evolve quickly to meet technology and business changes, hence their audit testing requires that test cases are prioritized, to ensure important test cases are ranked high and have thus more chances of being selected for execution. Existing techniques for test prioritization are coverage-based [1], [2]. While they are expected to perform well on generic faults, these techniques are inadequate when a service composition exhibits the separation of channels characteristic, i.e., only specific combinations of input-output channels are affected by a specific service change.

We proposed information retrieval to search for test cases relevant for a given service change. Search results form a ranked list, used to prioritize the test cases. We conducted a case study to test the effectiveness of IR-based prioritization in comparison with coverage-based prioritization. Results indicate that while on generic faults the advantages of IR-based techniques are minor and depend on the specific features of the considered faults, for profile-specific faults (e.g., faults affecting only users that have a specific location), coverage information is inadequate to rank test cases, while IR-based ranking is definitely more appropriate (we obtained always the best possible ranking with these type of faults).

As always happens with case studies, the possibility to generalize our findings to other systems is quite limited. We tried to use a service composition having all main features commonly exhibited by existing compositions and we took care of choosing one with realistic size, complexity and functionalities offered to the final users. However, in order to corroborate our findings, further studies on additional systems should be conducted.

ACKNOWLEDGMENT

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

REFERENCES


