Challenges in Audit Testing of Web Services

Cu D. Nguyen, Alessandro Marchetto, Paolo Tonella
Software Engineering Research Unit
Fondazione Bruno Kessler
Trento, Italy
cunduy,marchetto,tonella@fbk.eu

Abstract—Audit testing of Web services is a form of regression testing that applies to the integration of external services in a service composition. When such services change, the application that integrates them needs to be regression-tested, to ensure that the existing functionalities are not broken by the service changes. This means the external services must be audited for preserved compatibility with the integration.

In this paper, we analyze the main research challenges behind audit testing of services. Such challenges descend from the limited testability of external services and encompass traditional regression testing problems, such as test case selection and prioritization, for which existing solutions are nonetheless inadequate. Moreover, in the paper we discuss in detail one such challenge, i.e., the problem of defining an oracle by making the integrator’s assumptions explicit.

Keywords—Web Service Composition, Audit Testing, Regression Testing.

I. INTRODUCTION

As the SOA (Service Oriented Architecture) paradigm is becoming more and more mature, its adoption is growing fast in many areas, from business integration and social media, to end-user mobile applications. The key element of SOA is the integration of heterogeneous Web services from different business domains. These enable run-time discovery and binding, and bring enormous flexibility to service compositions such as reconfiguration and adaptation. However, Web services evolve quickly to meet technology and business changes. For instance, eBay released 8 versions1 of their finding service within each year, from 2009 to 2010. Similarly, Amazon released 12 versions2 of their Elastic Compute Cloud within the same year. As a result, service integrators face frequently a critical decision either to update their service compositions to exploit the new versions of the services they are using, 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. Testing to make sure that service compositions work properly with new services is, therefore, crucial.

Audit testing deals with such problem. It 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 is currently working properly. Audit testing is a form of regression testing since the goal is ensuring that the pre-existing functionalities are preserved when the new versions of the service are used.

In this paper, we discuss the main research challenges behind audit testing of services. Such challenges descend from the limited testability of external services and encompass traditional regression testing problems. Specifically, we analyze 4 challenges that ask for more research attention, including: (1) Online testing, (2) Service change detection, (3) Test case selection, minimization, and prioritization, and (4) Oracle. We introduce them in detail in the next section.

In addition, we present a service assumption specification language that we provide to the service integrators as a tool to specify service assumptions that they expect Web services to obey. From such assumption specifications we can derive composition-specific oracles to be used during audit testing of external services. The aim is to tackle the oracle challenge.

II. CHALLENGES

Several testing challenges descend from the intrinsic features of the services that are integrated in a SOA service composition. Services have low accessibility and observability. This means that testing them is severely limited and may involve even monetary expenses. Services evolve quickly and frequently, but their evolution is not necessarily notified or reported according to any formal protocol. Hence, even recognizing the need for regression testing of a service composition might be challenging. 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 ad-hoc, novel solutions. The problem of assessing the expected outcome of audit testing is also non-trivial. In fact, the

2http://developer.amazonWebservices.com/connect/kbcategory.jspa?categoryID=86
interface description languages for services document just the syntactical structure of the input-output relation of a service. The interaction protocol, as well as the semantics of the exchanged data, remain undocumented or expressed just informally as comments of the interface description documents (e.g., WSDL).

A. Challenge 1: Online testing

Online testing of services is a key, unavoidable phase of integration and system test [1]. It is inconceivable to deliver an application that was only run against stubs of the services to which it will be bound when executed in the field. Part of the testing effort must necessarily be devoted to online testing, i.e., testing against the actual, external services. However, services must be designed with testability as one of their key attributes in order for online testing to be possible. Otherwise online testing might be extremely troublesome or even impossible. In fact, not only service invocation may involve a monetary cost, but it may also affect the state of the world in an irreversible, possibly undesired way.

There are several models that can be adopted to add a testability interface to services [2], [3], so as to support their online testing. Here we mention some of the possibilities that have been discussed in the literature:

Test mode: Services can be invoked in test mode. Their execution will be the same as in normal mode, but in a sandbox. This means that the effects the service is supposed to have on the state of the world will remain isolated within a separate copy of the “real” world, which constitutes the test environment of the service. The business model behind the provisioning of services in test mode might include the payment of a fee for each test mode invocation, the limitation of the resources accessible (bandwidth, number of requests), etc. In any case, we can expect severe limitations on the number and duration of the test cases that can invoke a service in test mode.

Third party testing: Service integrators are not allowed to invoke a service in test mode directly, but they may delegate such a task to a third party, who offer testing as a service to multiple service integrators. Agreements between this third party and the service provider ensure the former has access to the services for testing purposes (which includes the possibility to execute the services in a sandbox, protected environment).

Test certificates: Services come with test certificates, which provide evidence of the degree of testing reached for a given adequacy criterion. Such certificates may include summary coverage information, as well as execution scenarios with the actual output. Service integrators use such certificates to decide on the compatibility of the service with the service composition under development.

While currently there is no agreement on how to support online testing of services and no testing infrastructure or standard exists yet, any of the solutions sketched above is characterized by severe limitations on the amount of testing that can be carried out by service integrators. Hence, regression testing upon service evolution become extremely constrained and demands for novel approaches.

B. Challenge 2: Change detection

Service changes may go unnoticed, since there is no standard for the notification of changes to service integrators. Developers rely on monitoring to assess whether any misbehavior occurs at run time. However, it is definitely preferable to detect misbehaviors or unexpected behaviors during regression testing, rather than the actual service invocation.

In order to trigger audit testing of an evolved service, the service change must be first detected. If not properly notified, change detection can be based on models of behavior that are automatically inferred from traced executions of the service.

In a previous work [4] we investigated a method for the automated detection of changes occurring in an application that undergoes run time adaptation and evolution. The same approach can be used to automatically identify the need for regression testing, when the monitored behavior of an integrated service deviates from the one of previously inferred models. The idea behind the approach we have proposed for automated change detection [4] is summarized in Figure 1.

A sliding window is moved over time as new execution events are collected by monitoring the service being integrated. Such a window has a size of $2w$. Its first half, of size $w$, is used to infer a model from past executions. This is indicated as model $M_1$. The second half of the window, still of size $w$, is used to infer a model $M_2$ from the most recently traced events. The idea is that when the similarity between the old model $M_1$ and the recent model $M_2$ goes below the threshold $T$, a discontinuity in the service behavior is detected, triggering regression (audit) testing of the service.

The two critical parameters involved in the automatic change detection shown in Figure 1 are the window size $w$ and the similarity threshold $T$. In fact, a too large window size $w$ introduces a big delay in the point in
time when the change is detected. However, a too small \( w \) results in models that do not account for a sufficient sample of application behaviors, such that models changes are continuously detected just because of model instability, due to the small window size, rather than actual changes of behavior. The other parameter, the similarity threshold \( T \), suffers similar problems. A too high value results in lots of false positives, i.e., similarity goes below threshold often, even when only minor changes happen. On the other hand, a too low threshold \( T \) gives raise to false negatives, since actual behavior changes may go undetected because the two inferred models \( M_1 \) and \( M_2 \) do not have enough differences. We have proposed a calibration procedure to determine the optimal tradeoff between the value to choose for \( w \) and that of \( T \) [4].

C. Challenge 3: Test case selection, minimization and prioritization

Given the extremely limited resources available during audit testing of services, the typical regression testing problems of test case selection, minimization and prioritization [5] become even more critical. We think the existing solutions developed for traditional systems are not adequate in the context of audit testing of services [6], for the following reasons:

Test case selection: Selection of 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. More powerful methods for selection would be needed.

Test case minimization: Again, a consequence of the low observability of services is that minimization based on coverage criteria (e.g., finding the smallest test suite that preserves some coverage level) is impossible to achieve during audit testing of an evolved service. Adequacy can be measured only on the side of the composition. It is unclear whether minimizing the test suite that achieves a given level of coverage on the composition is a good minimization strategy when the purpose is audit testing of an external service that undergoes evolution. We think novel methods would be needed to support minimization.

Test case prioritization: While prioritization of test cases based on the level of coverage achieved on the service composition remains a valid option, this is probably not the best option when the impact of a service evolution does not correlate with coverage or additional coverage of test cases. What might happen is that a test case with relatively low coverage and additional coverage is extremely sensitive to the changes occurring in an external service it invokes, while other test cases which have larger coverage may not have such a sensitivity. In our previous work [7], we investigated mutation analysis as a method to estimate the sensitivity of each test case to service changes and eventually to rank test cases based on such sensitivity.

D. Challenge 4: Oracle

The public interface of services is usually limited to the syntax (some proposals, such as SAWSDL\(^3\) exist to go beyond that, but none has reached consensus so far). The WSDL document which describes it does not take into account the semantics and the interaction protocol of the service. As a consequence, service integrators make implicit assumptions about the expected semantics and protocol. Such assumptions translate into the way the service composition works and violation of any such assumption may result in a failure of the service composition.

As a consequence, when conducting audit testing of a service, developers should focus on the behavior that the composition is expecting from the service, not on the behavior it is supposed to exhibit in general. Making the integrators’ assumptions explicit is thus key to the definition of accurate oracles that can be used during testing. In fact, generic oracles based just on the generic behavior of the service might miss important details related to the implicit assumptions that make the service composition work properly. The composition will fail if the service is evolved without respecting such assumptions. In the next section we present a service assumption specification language, from which we can derive composition-specific oracles to be used during audit testing of external services.

III. ASSUMPTION SPECIFICATION LANGUAGE

In our approach, service integrators specify their service assumptions explicitly, in order to simplify and automate audit testing of integrated services. The benefits of specifying explicitly the assumptions made by the service integrator about the integrated services are the following:

1) Using explicit service assumption specifications to check data received from services helps the integrator define a composition-specific oracle for the test cases used during audit testing, upon service change.
2) Putting together scattered service assumptions and the related validating code in a single place, reduces maintenance costs and makes it easier to check for completeness.
3) Specifying service assumptions in a high-level language, e.g. XML, separately from the code, allows to evolve them easily, even at runtime (with no need for code modification and recompilation).

\(^3\)http://www.w3.org
For the specification of the integrator’s assumptions, we propose an XML based assumption specification language. A service assumption consists of an XPath reference to an element in the response under consideration and it includes data restrictions regarding that element. Data restrictions have the same format as those defined in the W3C XML Schema. 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 [8] for more information regarding the structure and benefit of integrator’s service assumption specification.

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. Assumption violation indicates that the integrated service is no longer compliant with the behavior expected by the service composition. This might require further evolution of the composition so as to make it compatible with the evolved service. If not detected during audit testing, such incompatibility may cause a failure of the composition during its actual service, rather than during testing. Hence the importance of audit (regression) testing based on explicit specification of the integrator’s assumptions.

IV. CONCLUSION AND FUTURE WORK

In this paper we discussed the relevant research challenges that need to be considered when performing audit testing of external services. In particular, we identified four challenges: (1) online/offline testing; (2) detection of the changes in the service; (3) test cases selection, minimization and prioritization; and (4) definition of test oracles. For each challenge we discussed possible ways to address it during audit testing.

From this challenge-focused analysis it is quite clear that more investigation is required, since no shared and effective testing techniques and tools exist to face all these challenges and to efficiently perform audit testing of Web services.

ACKNOWLEDGMENT

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

REFERENCES


