ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/234803444

Evaluation of online testing for services

Article · January 2010

DOI: 10.1145/1808885.1808893

CITATIONS
17

READS
67

3 authors, including:

Michaela Greiler
Microsoft

18 PUBLICATIONS **283** CITATIONS

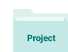
SEE PROFILE

Hans-Gerhard Gross
Esslingen University

76 PUBLICATIONS **875** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Poseidon View project

All content following this page was uploaded by Hans-Gerhard Gross on 20 May 2014.

The user has requested enhancement of the downloaded file.

Evaluation of Online Testing for Services – A Case Study

Michaela Greiler, Hans-Gerhard Gross and Arie van Deursen
Software Engineering Research Group
Delft University of Technology
Mekelweg 4, Delft, The Netherlands
{m.s.greiler|h.g.gross|arie.vanDeursen}@tudelft.nl

ABSTRACT

Service-oriented architectures (SOAs) have found their ways into industry to enable better business-to-business cooperation. With the advent of SOA, new challenges for software development and testing also appeared. In this article, we motivate the need for SOA online testing and show how it can reveal faults that slipped offline testing. The paper outlines a case study in which online testing has been implemented as proof-of-concept, but also evaluated in terms of its effectiveness to identify typical SOA runtime reconfiguration problems according to an existing fault taxonomy. The experiments of the case study demonstrate that online testing (1) can detect many typical runtime reconfiguration faults, and that (2) online testing provides additional value over offline testing.

Keywords

Online Testing; SOA Testing; Service Testing; SOA Reconfiguration; Integration Testing;

1. INTRODUCTION

Service-oriented architectures (SOAs) are increasingly emerging as backbone platforms for information and communication technology (ICT) infrastructures of enterprises and organizations, because of their dynamic nature, their loose coupling of components and sub-systems, their adaptability to changing business requirements, and their reconfigurability. Organizations migrate their systems-of-systems to SOA, in order to become more federated and service-oriented, and in order to align ICT evolution with business development. Examples of such systems-of-systems are safety and security systems, railway and aviation control systems, toll collect systems, online banking and trading systems, enterprise resource management and logistic systems, and multi-tenant systems.

Testing such systems-of-systems is aggravated by stakeholder separation and dynamic features provided by SOA infrastructures. Stakeholder separation turns third party services into black-boxes for integration testers, and limits their influence in system evolution and maintenance. Additionally, dynamic features, like ultra-late binding or dynamic composition of services, allow a flexible and dynamic way of composing a concrete system at runtime, but restrict the ability to test it a-priori. The context in which a service will be used is likely unknown at service development time, imposing problems for service testers to predict and foresee possible requirements and usage scenarios. The limited testability of service-oriented systems imposes the need of reconsidering and redesigning traditional, and inventing new testing methods and processes, moving activities from design time to runtime, as mentioned in [6, 9, 13, 17].

In this paper, we demonstrate the need for *online testing*, i.e., testing in the realm of the production system in parallel to its operational performance. Online testing is applied in addition to offline testing, in order to deal with faults that could not be addressed before. The scope of online testing excludes tests that can be done offline, like full behavioral tests of services.

The main contributions of this paper are (1) a case study demonstrating the capabilities of online testing to detect faults during system reconfiguration that could not be addressed in the test environment, and (2) an evaluation of the results including a discussion on open issues and lessons learned from the case study.

Online testing is demonstrated in a concrete dynamic SOA, namely the Open Services Gateway Initiative[1] (OSGi Alliance) framework. A small case study of a distributed sensor network which is part of a traffic control system is used as proof-of-concept. Within this case study, the ability of online testing to detect runtime reconfiguration faults is evaluated according to the SOA fault taxonomy by Bruning et al. [5]. We outline open issues and research directions of performing service integration tests during operation time in the context of SOA environments.

Section 2 gives an overview of existing research. Section 3 outlines briefly our online testing approach. The case study is presented in Section 4, followed by an evaluation in Section 5. We conclude in Section 6 and give an outlook on future work.

2. RELATED WORK AND BACKGROUND

2.1 Online Testing

Online testing is applied in order to assess a new service in the context of the actual system in which it is going to

[1] http://www.osgi.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PESOS '10, May 1-2, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-963-3/10/05 ...$10.00.

operate, according to its real usage. Similar to online testing is in vivo testing [8], also dubbed as perpetual testing[2], used to assess live applications after deployment. While in vivo testing focuses on performing unit tests, we focus on integration testing of service compositions.

The scope of the test naturally affects the test approach, the test cases required, and their number. Online testing does not replace offline integration testing, but offers an efficient approach to address reconfiguration faults that cannot be identified in the offline test environment.

Infrastructures and Testability. Some work has focused on infrastructures for testing component-based software online, such as in Vincent et al. [20], Suliman et al. [18], and Gonzalez et al. [11]. All use built-in testing, associating tests with components [14]. These tests are used by other components to assess a subject under test (SUT), or by the component itself to validate its peers.

While executing tests in the production system, undesired side effects (the problem of *test sensitivity*) have to be avoided. This requires *test isolation* which builds on *test awareness*, i.e., components must be aware that they are being tested [4, 11, 13, 14, 19].

Integration Testing of SOA. Research in service-oriented testing has gained a lot of interest [7]. Also the field of online testing has been addressed, e.g., Bertolino et al. [1, 2]. They suggest that the originally passive registry should take over an active audition role. The audition phase takes place before the service is listed in the registry, and is performed by a central instance, checking the correctness of a service against its specification. The services already active in the SOA do not take part in the testing process. In our online testing approach the service implementation is not checked against its own specification, but against the expectation (specification) of another requesting service.

Heckel et al. [15] discuss an approach, in which a central third party, called the discovery agency, checks that a service conforms with its specification, i.e., a model, before it is registered. Later on, other services can discover the registered service and can compare their requirements against the model provided by the service. This assures conformance of description and implementation of a service, by performing full functional and behavioral tests. Their assumption is that the service could be faulty, either unintentionally because of insufficient testing, or intentionally for introducing malicious services. In constrast to a full behavioral test, our online testing merely focuses on integration faults that are not easily caught using just offline testing.

2.2 Service-Oriented Faults

Fault models and fault trigger schemes are useful to highlight faults that are predestined to appear in certain types of systems [3]. For evaluating the fault finding capabilities of online testing, we focus on a fault taxonomy by Bruning et al., which is specific to service-oriented architectures [5]. As we will see in Section 5, these service-related faults are of special interest for our research, because they are likely to happen during runtime (re-)configuration. In the paper, we will refer to those faults as reconfiguration faults.

The taxonomy, summarized in Table 1, spotlights faults directly connected to the five characteristic phases in a SOA: service publication, service discovery, service binding, service composition, and service execution.

[2] http://www.ics.uci.edu/~djr/edcs/PerpTest.html

3. ONLINE TESTING FOR SERVICES

3.1 Online Testing Process

During online testing a test is triggered, whenever the system undergoes a reconfiguration.

As a first step, the new service will be deployed, in parallel to the old version, in the production system, but does not publish its service yet. Other services will not use it for nominal requests. Each service comprises a test suite to assess proper functionality of required peers.

Then the test suite of the new service will be executed. The test suite is checking proper communication and cooperation between itself and other acquired services, whenever the environment changes.

The testing activities address three main phases, each phase comprising a set of test cases:

1. *Discovery Tests* check whether all required services can be ***discovered***, and whether the services found are the services expected.

2. *Binding Tests* check whether all required services can be ***bound***, and conform to the (syntactic) expectations (according to service description).

3. *Composition Tests* **execute** required services and assess whether expectations about sequential constraints, i.e., the protocol, between message invocations are fulfilled. Simple input-output relations including return values can also be assessed, as well as non-functional requirements (e.g., timing).

If all tests have passed, the operational service is started, and can replace the previous version. If a test fails, the service will not be registered.

A test suite is detached from the service code, in order to be able to update the test cases independent of the service. This permits evolution of the testing code together with the evolution of the system. Examples of concrete test cases, in particular for OSGi, for each phase are given in Listing 1.

Listing 1: Examples for the Test Case Classes

```
/* Checks whether the service required can be found under this
 * name in the service registry */
public void testDiscover() {
    String name = IService.class.getName();
    ServiceReference ref = getServiceReference(name);
    assertNotNull("The Service Reference " + name
        + " cannot be found", ref);
    tService = (IService) getContext().getService(ref);
    assertNotNull("The Service " + name
        + " cannot be retrieved", tService);...}

/* Checks whether the service has all the expected methods */
public void testBinding() {
    try {
      tService.getInformation(null);
    } catch (AbstractMethodError e) {
      Assert.fail("Method getInformation is missing");}...}

/* Checks whether the invocation sequence expected is correct
 * and adds simple expectations for return values */
public void testComposition() {
    User u = new User();
    assertTrue(tService.register(u, "topic"));
    assertNotNull(tService.getInformation(u));
    assertTrue(tService.unRegister(u));...}
```

Category	Forms
Publishing Fault	Incorrect Service Description, or Service Deployment
Discovery Fault	Discovering no Service, or the Wrong Service
Composition Fault	Missing or Incompatible Components, or Non-fulfilled Service Contracts
Binding Fault	Binding Denied, or Bound to Wrong Service
Execution Fault	Service Crashed, or Incorrect Results

Table 1: Summary of Service-Oriented Fault Taxonomy by [5]

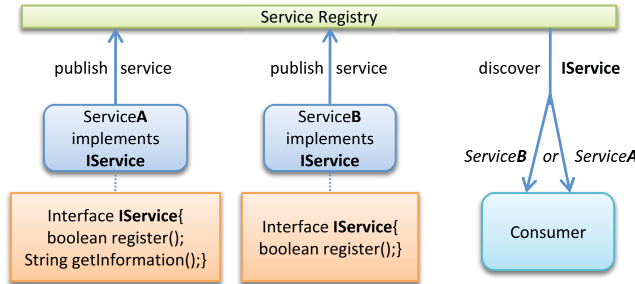


Figure 1: Inconsistent Interfaces in Registry

3.2 Testability Facilities

Controllability and observability are two important characteristics to allow efficient testing. Especially during online testing, it can be crucial that the system is aware of the test activities, and isolates test invocations accordingly. To meet these demands, a service comes in two forms: (1) the *operational service*, serving nominal requests, and (2) the *testable service*, allowing other components to test the service. The testable service extends the operational service (inheritance or delegation), following the adapter design pattern [10]. The main purpose of the testable version is to improve testability by providing test isolation, and enhancing controllability and observability.

The fact that service objects for testing and production are different helps to avoid interferences between testing a service and operating it normally. The extension mechanism permits to overwrite test sensitive parts with side-effect-free implementations.

4. CASE STUDY

We have devised an explorative case study, following Yin [21], as proof-of-concept, and in order to demonstrate, study, and evaluate the fault finding capabilities of online testing. Although, the code of the system of the case study is confidential, a small example application can be downloaded[3], which contains the whole infrastructure.

4.1 Case Study Design

The system used for this case study is based on a vessel tracking and surveillance system, named OVTS (OSGi vessel tracking system), consisting of a sensor network and several components that relay, manage, and display messages sent by vessels[4].

Goal. The goal of the case study is to assess the capabilities of online testing to find typical reconfiguration faults, following Bruning et al. [5].

Approach. First, the original system has been extended with testability infrastructure. Furthermore, as a typical reconfiguration scenario, the system's functionality has been enhanced with capabilities to monitor the Quality of Service of the sensor network. During reconfiguration, a number of faults were seeded to assess the fault detection capabilities.

Key Metrics and Execution Environment. The specific OSGi framework used was the Apache Felix 1.8 implementation. The OVTS system comprises 6 key bundles (including the enhanced visualizer), 1 utility bundle, 2176 lines of code, and 41 classes. 349 lines of code were dedicated to testing and test isolation. We implemented 39 test cases for the visualizer, consisting of 290 lines of code, based on the three test classes described in Section 3.1.

Units of Analysis. The main concern of the case study is the assessment of the detection rate for the seeded faults. For detected faults, we trace how they have been found, and for missed faults we analyze why they could not be detected. In addition, we evaluate the support of the OSGi execution environment to prevent inconsistencies in the overall system.

Research Questions Addressed. The case study was conducted to answer the following research questions:

RQ 1: "Which typical reconfiguration faults can be detected with online testing as presented?"

RQ 2: "Which faults cannot be detected?"

RQ 3: "To what extent are these faults attributed to the current state and configuration of the production system?"

4.2 The Subject System

The vessel tracking system comprises a number of *radio base stations*, a *Merger* service, a *Filter* service, and a *Visualizer* service in the control centre, as illustrated in Figure 2. The main purpose of the *Visualizer* is to display vessels and their tracks on a screen. *Base stations* receive messages from vessels, indicating properties like position, speed, time, identification, etc. The *base stations* transmit the messages to the *Filter* component in order to eliminate duplicate messages, because some *base stations* cover overlapping areas. Messages are relayed to the *Merger* service which acts as a server for sensor data. Clients, such as the *Visualizer*, can connect to it and subscribe to sensor data of a particular vessel for track information. The system is connected to a simulator providing real vessel sensor data. That way, the sensor network can be executed and tested as if it was in the real production environment.

4.3 Reconfiguration Scenario

The update requirement for the surveillance system is to add a Quality of Service (QoS) monitor and connect it to all

[3]http://swerl.tudelft.nl/bin/view/ARTOSC/Software Distribution

[4]The vessel tracking system in question has also been used as a case study in [11] and [12].

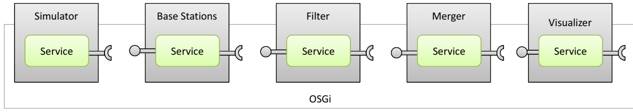


Figure 2: System Architecture

base stations in order to analyze the quality of the received messages, for provision of system health information. The existing services are reconfigured, so that the QoS service becomes part of the *Visualizer* component, leading to a test of the *Merger* by the test suite of the new *Visualizer* service.

4.4 Observations

While performing the system evolution scenario, we seeded a number of faults in the system in order to assess online testing.

Fault Finding Capabilities. We seeded faults out of every fault category defined in [5]. Table 2 summarizes the results of the experiments by stating the fault category, the specific form of fault, and the test class, as defined in Section 3, detecting this fault, or "X" if undetected.

Publishing Faults. Service description faults, either by missing features or incomplete descriptions, have been seeded in the system. For example, the new version of the service registered itself under a wrong service interface name. This fault was immediately detected by the OSGi framework when the new service was attempting to start, and the start was prohibited by the execution environment[5]. Another publishing fault based on a mismatch between service description and service was introduced when the service expected more features (e.g., methods) from a required service than were provided. This fault was detected by a *binding test*, assessing whether all expected methods exist.

Discovery Faults. We introduced faults in the discovery process, e.g., by implementing a wrong look-up query. As a result, the required service was not found. The fault was detected by a *discovery test*, checking whether required services can be found. Modifying the discovery query to look for an existing but wrong service, would lead to an exection failure. However, a *discovery test* detected that casting the service found to the expected service will raise an error.

Composition Faults. Such a fault was introduced by deviating from a contract incorporating response time, e.g., by reducing the time tolerance for a response from *Merger*. The fault was found by a *composition test* concentrating on response time for a data request. Another composition fault, introduced by changing the sequential constraints of the *Visualizer*, caused a violation of *Merger*'s protocol. The *Visualizer* expected the *Merger* to provide data, without registering for a specific vessel. This fault was found by a *composition test*, exercising a typical protocol sequence.

Binding Faults. Binding can be denied because of insufficient security, authentication, accounting or authorization problems. By changing a service into requiring authentication before requesting data, the expected protocol sequence was changed. The other service was expecting to be able to request data without authentication, which violated an assertion in a *composition test*, checking typical protocol constraints. Another binding fault was introduced by changing the implementation of a method. The service provided all methods described in the interface, but the expected behavior of the implementation did not match with the actual functionality. The results provided by the service were wrong, but this could not be detected by our tests.

[5]Some execution platforms will not detect this automatically, thus requiring an additional test.

Execution Faults. An execution fault, in form of a required service crashing before or while testing, can be found by either a *composition test*, by a *binding test* or by a *discovery test*, depending on the time the service crashes. We seeded further faults, by returning incorrect results, e.g., by changing the implementation of a method in the *Merger* to return all vessel information instead of a specific one. This fault could not be found during online testing.

5. DISCUSSION AND EVALUATION

5.1 SOA and Runtime Reconfiguration Faults

In this section, we discuss the results of the case study with respect to the relevance of online testing vs. offline testing.

Publishing Faults. Checking syntax and format of the service description can be easily done offline. Also checking consistency of service interface (description) and implementation is done offline, during compilation time. In OSGi, a service description, represented by an interface or a class, has to be syntactically correct and in the right format to allow successful compilation of the service. Also the service implementation has to match the specific interface.

On the other hand, publishing faults can occur during runtime in the production environment, because the system changes and inconsistencies between service interfaces occur. In OSGi, different services can be registered under the same interface name, as long as they are implementing this interface but it is not checked (e.g., by the registry) whether the service interfaces, registered in the environment, are consistent, as illustrated in Figure 1. Hence, it is possible to register the same interface with completely different characteristics, e.g. number of methods, parameters, return values. The proposed *binding tests* are sufficient to check whether services are consistent with the expected service description.

Discovery Faults. Correct syntax of discovery queries can be tested offline. On the other hand, a test checking that all required services are deployed and can be found, based on the query, can have different outcomes in the production and the test environment. Online testing can support finding missing services or wrong service look-ups or ambiguous queries in the production environment by execution of *discovery tests*.

Composition Faults. Composition faults are likely to occur online, because services may have been mocked in the testing environment or are missing in the production environment, and thus incompatibilities could not be foreseen. *Composition tests* cover service compositions by executing typical protocol sequences as expected by the test case. We found deviations from the protocol. It is often sufficient to perform typical scenarios to identify protocol mismatches, e.g., typical correct and incorrect sequences. Testing all possible permutations of test invocations can produce a large number of test cases.

Furthermore, a composition can be faulty, because of non-functional issues in the production environment. In our case study, we assessed service response. This can be caught with *composition tests*.

No.	Fault Class	Form of Fault	Found by
1	Publishing Fault	Service Description Fault	Binding Test
2	Publishing Fault	Service Deployment Fault	Binding Test
3	Discovery Fault	No Service Found	Discovery Test
4	Discovery Fault	Wrong Service Found	Discovery Test
5	Composition Fault	No Valid Composition	Composition Test
6	Composition Fault	Composition Faulty	Composition Test
7	Binding Fault	Binding Denied	Composition Test
8	Binding Fault	Bound to Wrong Service	X
9	Execution Fault	Service Crashed	Discovery Test
10	Execution Fault	Incorrect Result	X (Unit Tests)

Table 2: Fault Classes, Fault Forms and Test Cases

Binding Faults. Online testing adds benefits if not all authentication, authorization, and security configurations have been properly reproduced in the testing environment. Denied binding faults have been found by *composition tests*, looking either for invocation sequences or for expected output values.

Wrong service bindings can occur in the final environment, e.g., because multiple services match an ambiguous query. In some scenarios, the fault has been found based on simple assumptions about the output.

In general, these faults are hard to detect, if the syntax of service interfaces match.

Execution Faults. If a service crashes before a test is invoked, *discovery tests* will detect a missing required service. If a service crashes while testing, it will be detected by *binding* or *composition tests*. Execution faults leading to wrong results, are hard to find online. These are typical issues to be addressed in offline/unit testing.

5.2 Research Questions Revisited

RQ1: Which faults can be detected by online testing? It can indeed detect typical publishing, and discovery faults, like missing services, and inconsistencies among different service descriptions and implementations. These are typical issues appearing in dynamically evolving systems. Composition faults, violating timing constraints, are difficult to identify offline due to specific service compositions which are hard to reproduce in an offline testing environment. Online testing on the other hand is efficient to detect those faults.

RQ 2: Which faults cannot be found by online testing? The case study showed that online testing is not effective in finding all protocol deviations.

Nevertheless, by testing typical protocol sequences, obvious mismatches can be identified. Execution faults representing incorrect results provided by services could not be found; this should be done offline.

RQ 3: How are faults related with the specific state and configuration of the production system? Both fault types, publishing and discovery, are an effect of the specific state and configuration of the production system which is not reproducible in an offline testing environment assuming clean initial state and configuration. For example, OSGi keeps required packages in the environment even though they have been uninstalled. Further, these packages are hidden, meaning that they do not appear in the list of installed packages anymore. Reproduction of such an environment would require that the history of all installation activities and configurations is reproduced in an offline test setting.

5.3 Research Directions and Open Issues

Performance and Adequacy. In the case study, we did not observe any performance, or resource problems caused by online testing. In more complex systems they might well occur, leading to the question of an adequate online testing framework providing test isolation, observability and controllability, and an adequate or minimal test suite to be invoked online. Online tests are not meant to be full behavioral tests or replace offline/unit testing. They should only address issues that can go wrong in a running and evolving system. Nevertheless, protocol deviations and incorrectly bound services require more extensive test suites for their detection. Future research must study the tradeoffs.

5.4 Threats to Validity

The closed nature of the case study constitutes a threat to reliability (repeatability). To address this, we tried to be as explicit as possible in our description of the case study design and provide a downloadable example system. Another important threat to the validity of our case study concerns the external validity, i.e., the extent to which the results can be generalized [21]. This gives rise to the following considerations.

Generalization for SOA. OSGi might not be considered as a full-scale SOA. However, while looking for an experimental environment, we saw several advantages. OSGi is relatively small, which makes understanding and observing the behavior easier and more reliable, and it inherits all main characteristics of a general service-oriented architecture: application frontend, service, service repository, and service bus [16]. Furthermore, OSGi technology is used as a basis for all main SOA infrastructure products and systems, like IBM with IBM Websphere, Sun with Glassfish, IOAN with Artix and FUSE, WSO2 with Carbon, and SOPERA ASF that offer Eclipse Swordfish, a modular and distributed SOA framework[6]. In addition, OSGi bundles can directly be deployed as services in other SOA products, like in Sun's Glassfish[7].

Application Size and Architecture. The method presented was applied only in one case study, quite small in nature. Two related concerns have to be addressed: first, larger and more complex systems might be too expensive to be enhanced with online testing infrastructure. This could be addressed through careful consideration of test sensitive parts, i.e. what is an adequate/minimal online test-

[6]http://www.eclipse.org/equinox-portal/quotes.php
[7]http://blogs.sun.com/dochez/entry/glassfish _v3_extensions_part_4

ing framework?, or through code/test case generation. For discovery tests and binding tests, this can be done without additional enhancements. Generating composition tests requires a model describing input-output expectations and sequential constraints for method invocations.

Representativeness of faults. One could argue that the faults observed are specific to OSGi, and application dependent, and not representative for service-oriented architectures in general. To address this threat we based our evaluation on a service-oriented fault taxonomy. The fault taxonomy is domain independent, because it focuses explicitly on faults that appear during the five characteristic steps in SOAs. Most of the faults observed have been caused by the internal state of OSGi, that is e.g., the set of installed and active services or available packages and their wiring to components. We expect that similar faults can be observed in any other platform.

On the other hand, the results of the case study indicate some shortcomings in the fault taxonomy, that are discussed in the following section.

5.5 Lessons learned

Online versus Offline Testing. An important issue addressed in this paper is the different faults detected online or offline. In our study, all services were able to compile and passed unit as well as integration tests in the test environment, which reflected the developer's expectations about the production system.

Some seeded faults, like incorrect functionality of a service, representing execution faults, should have been caught by offline testing. Online testing does not focus on revealing this kind of faults.

An incorrect look-up query, representing a discovery fault, can be found by offline testing only if the query discovers a wrong service. For ambiguous queries, this is not the case. These queries could find one service in the test environment, but more than one service in the production environment.

Faults representing wrong expectations in provided information, like signature, are only introduced if the production environment does not impose those expectations. Theoretically, the testing environment should be equivalent to the production environment, which is not achievable in reality.

Finally, application tolerance is a problem that cannot always be addressed adequately in test environments. Application tolerance stands for the tolerance of multiple applications for each other. If applications are not isolated completely, by e.g., accessing the same parts of a file system, faults can occur. To address this problem, a test environment would have to represent all programs existing in the production environment, which is in many cases infeasible.

Shortcomings of the fault taxonomy. By analyzing the faults detected in the case study, we found out that the categorization of the fault classes does not reflect faults but concentrates on failures. Especially for faults in the later phases in a SOA, like execution and composition, the faults described are in reality effects of faults from earlier phases, propagated to later phases and then manifested as failures. Faults during update and removal of services are also not addressed.

Shortcomings in the OSGi framework. The support for consistency management in the OSGi environment is limited. The OSGi framework recognizes whenever a service is published under a wrong service interface, i.e., an interface that it does not implement. OSGi does not manage consistency between several interfaces with the same name. This is a problem because discovery queries are based on the interface name of a service. This means that every service registered under the interface name "IService", independent of the signature of the interfaces, will be found by a service querying for a service implementing "IService". Figure 1 illustrates the registration and as a consequence also the discovery of "IService"-services by a consumer. The consumer gets both services (ServiceA and ServiceB) regardless of their signature.

6. SUMMARY, CONCLUSION & FUTURE WORK

In this paper, we presented a case study demonstrating the capabilities of online testing to detect faults/failures during system reconfiguration that could not be found offline in a "traditional" test environment. We devised a proof-of-concept, seeded typical SOA faults, in order to evaluate our online testing in terms of effectiveness to identify such faults. The outcome of the case study suggests that typical reconfiguration faults can be found through online testing, and that online testing has additional value over mere offline testing. For example, inconsistencies in the execution environments can be detected, and mismatches between expectation and reality can be uncovered. Further, the case study indicated that the real operational system state and configuration is not tested adequately offline.

As a next step, we will apply online testing to other SOA infrastructures, in particular the IBM product suite, and further evaluate it with different systems. Primary activities planned for future research are (1) to expand the fault seeding experiment to other execution environments and case studies, (2) to establish a taxonomy representing faults naturally appearing in the production environment, and (3) to gain empirical evidence concerning the effectiveness of the proposed approach.

7. ACKNOWLEDGMENTS

We would like to acknowledge Alberto Gonzalez and Eric Piel for providing insights into their work on Atlas, that helped to realize the concepts and case study discussed in this paper. This work has been funded by the Dutch Government through the Jacquard program (www.jacquard.nl).

8. REFERENCES

[1] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. The PLASTIC framework and tools for testing service-oriented applications. In *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, pages 106 – 139, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In *Architecting Systems with Trustworthy Components*, pages 1–25. Springer, 2006.

[3] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, October 1999.

[4] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll. Strategies for the run-time testing of third party web services. In *SOCA '07: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*, pages 114–121, Washington, DC, USA, 2007. IEEE Computer Society.

[5] S. Bruning, S. Weissleder, and M. Malek. A fault taxonomy for service-oriented architecture. In *HASE '07: Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, pages 367–368, Washington, DC, USA, 2007. IEEE Computer Society.

[6] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[7] G. Canfora and M. Penta. Service-oriented architectures testing: A survey. In *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, pages 78–105, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] M. Chu, C. Murphy, and G. Kaiser. Distributed in vivo testing of software applications. In *ICST '08: Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 509–512, Washington, DC, USA, 2008. IEEE Computer Society.

[9] S. Dustdar and S. Haslinger. Testing of Service-Oriented Architectures – A practical approach. In *Object-Oriented and Internet-Based Technologies*, pages 97–109. Springer Berlin/ Heidelberg, 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[11] A. González, É. Piel, and H.-G. Gross. Architecture support for runtime integration and verification of component-based systems of systems. In *1st International Workshop on Automated Engineering of Autonomous and run-time evolving Systems (ARAMIS 2008)*, pages 41–48, L'Aquila, Italy, Sept. 2008. IEEE Computer Society.

[12] A. González, É. Piel, and H.-G. Gross. A model for the measurement of the runtime testability of component-based systems. In *5th Workshop on Advances in Model Based Testing (A-MOST 2009)*, pages 19–28, Denver, Colorado, Apr. 2009. IEEE Computer Society.

[13] M. Greiler, H.-G. Gross, and K. A. Nasr. Runtime integration and testing for highly dynamic service oriented ICT solutions – an industry challenges report. In *TAIC-PART '09: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 51–55, Windsor, UK, 2009. IEEE.

[14] H. Gross. *Component-based Software Testing with UML*. Springer, Heidelberg, 2005.

[15] R. Heckel and L. Mariani. Automatic conformance testing of web services. In *Proc. Fundamental Approaches to Software Engineering (FASE)*, pages 34–48. Springer, 2005.

[16] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA : Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, September 2006.

[17] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[18] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka. The MORABIT approach to runtime component testing. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 171–176, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Y. L. Traon, D. Deveaux, and J.-M. Jézéquel. Self-testable components: From pragmatic tests to design-for-testability methodology. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 96, Washington, DC, USA, 1999. IEEE Computer Society.

[20] J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component-based software systems. *Software Quality Control*, 10(2):115–133, 2002.

[21] R. K. Yin. *Case Study Research, Design and Methods*. Sage Publications, Beverly Hills, CA, second edition, 1994.

View publication stats