Automating the Generation of Web Service Testbeds using AOP

Lukasz Juszczyk, Schahram Dustdar
Distributed Systems Group, Institute of Information Systems
Vienna University of Technology
Argentinierstraße 8/184-1, 1040 Vienna, Austria
Email: {juszczyk,dustdar}@infosys.tuwien.ac.at

***Abstract*—One of the key concepts of service-oriented computing is dynamic binding which favors on-demand integration of services into a running system. Companies can outsource tasks to external partners by using their services and are able to switch over to alternatives at runtime. However, this flexibility comes at a high cost because it complicates the testing process significantly. The major problem with external services is their restricted usability for testing purposes, due to costs or because policies forbid trial invocations. In this paper we present our approach to solve this issue by generating automatically testbeds which emulate external services. Based on previous work on testbed generation, we have developed a technique for intercepting Web service invocations in Java-based systems, generating emulated replicas at runtime, and redirecting the invocations transparently. We describe the concepts of our approach, its applicability, and limitations.**

I. Introduction

The concept of service-oriented architecture (SOA) is grounded on modularization of functionality into reusable services and on providing these to clients for on-demand usage. Hence, software systems can be engineered faster due to software reuse and due to the possibility to outsource particular tasks to external partners/experts. Though, in spite of these benefits, engineers of outsourcing systems are facing several problems. Integration of functionality provided by external services turns an SOA vulnerable, as the services may become unavailable or may suffer from degraded quality of service. Hence, it is a potential risk for the dependability of an outsourcing SOA system. To address this issue, such systems should undergo rigorous tests, in order to make sure that they are able to handle faults properly and that there will be no bad surprises once they are deployed. Unfortunately, the whole testing procedure becomes problematic as invocations of external services often cost money or because their providers have policies which restrict trial invocations. As a solution to this issue, we have proposed in previous works the generation of testbed infrastructures. We have introduced the Genesis2 testbed generator [1] which supports SOA engineers in setting up customizable testbeds. Later we have extended our work towards fault injection [2] for emulating faults in SOA environments. In short, engineers write scripts in which they specify the composition and and functional behavior of the testbeds and Genesis2 interprets the specifications and generates running testbeds instances from these.

In the current paper, we evolve our approach towards an automated generation of testbeds and reduce the input of SOA engineers significantly. We apply AspectJ [3], an AOP extension [4] for the Java VM, for inspecting SOA systems at runtime in order to detect invocations of external Web services. On a detection, our system analyzes the remote services, generates replicas of these, and deploys them within a testbed. Eventually, the invocation is redirected to the replica transparently, leaving the original service untouched. The result of this procedure is that external SOA infrastructures can be emulated on-the-fly and their replicas act as a testbed for the engineer. Compared to our previous work, the current paper brings forward the concept of testbed generation towards more automation, requiring less specification input from the engineer. It improves the practical applicability and accelerates the testing process.

This paper has the following structure. The next section covers our vision of software testing in SOA and motivates the usage of testbed generators. Section III explains our concept for automated testbed generation. In Sections IV and V we present a basic evaluation of our approach and discuss its limitations. In Section VI we review related research and in the end we conclude the paper and summarize our results.

II. Testing in SOA Software Development

The idea of software modularization was not invented with SOA, it has been applied since decades. However, what SOA propagates is not only to compose systems out of a set of modules, referred to as services, but also to integrate remote services, provided by external partners, into a system. This is supported by the open character of Web service-based SOA, that uses open standards for communication (SOAP [5]), interface descriptions (WSDL [6]), and for the numerous WS-* extensions which are public [7]. The benefits are obvious: faster software development due to reuse and the ability to choose dynamically among available services depending on their quality, just to list to most prominent ones. External Web services can be integrated easily by analyzing their WSDL descriptions, making sure that client and service agree on communication details, such as available operations and exchanged message types. To specify the quality of their services, some providers offer service level agreements (SLA) that define minimal performance metrics and penalties, if these

are not met. SLAs do help to make a SOA system more predictable, as service providers have a strong incentive to guarantee a promised quality in order to avoid penalties. Yet in spite of this gain on predictability, the dependency on external Web services remains a risk as SLAs are sometimes missed and a failing service can have critical effects on the system that depends on it. Of course, unless it is able to handle these properly. And here appears a major problem of today's SOA software development: how can engineers develop and evaluate techniques for handling faults of remote components, if they don't have full access to these components. Naturally, external partners do not allow arbitrary invocations of their services just for the sake of testing, as this would put additional load on these and degrade their performance. Furthermore, external services do often cost money and, therefore, each test run would cost money. To summarize the dilemma of SOA engineers: on the one hand the outsourcing of tasks to external services can accelerate the development of a system but, on the other hand, dependencies on external services do imply new challenges for testing the system.

In our previous work we argued that this dilemma can be mitigated by using testbeds that emulate external SOA infrastructures. We have developed techniques for how such testbeds can be generated and we have implemented a prototype called Genesis2 [1], in short G2. By making the tested system interact with replicas of external Web services, instead of their real instances, engineers are given a new level of freedom for running their tests. Of course, the usability of such testbeds depends mainly on their realism, in terms of how precisely they replicate external Web services. To bring forward this question, we have extended G2 for multi-level fault injection to simulate various faults in Web services communication [2]. The main benefit of our approach is that SOA engineers can test their systems in a kind of separated sandbox, that fakes an existing SOA environment and in which they can perform unrestricted test runs. In our methodology, the engineers had to write specification scripts that describe the environment to be emulated, including the topology as well as functional behavior of it. In the current paper we simplify this process, by partially automating the specification of testbeds.

III. Automated Generation of SOA Sandboxes

In a nutshell, our approach is based on monitoring running Java-based SOA systems, detecting calls of external Web services, and redirecting them to generated replicas. Figure 1 depicts our approach which consists of the following steps:

1) Detection of Web service calls in the SOA system, by intercepting WSDL retrieval code in the Java runtime.
2) Analysis of the WSDL document and generation of a replica model at the front-end.
3) Rule-based customizations of the replica model.
4) Deployment of the replica instance, from the model, at the testbed infrastructure (back-end).
5) Forwarding of the replica WSDL document to the SOA system, instead of the original WSDL.
6) Actual Web service invocation, redirected to the replica.

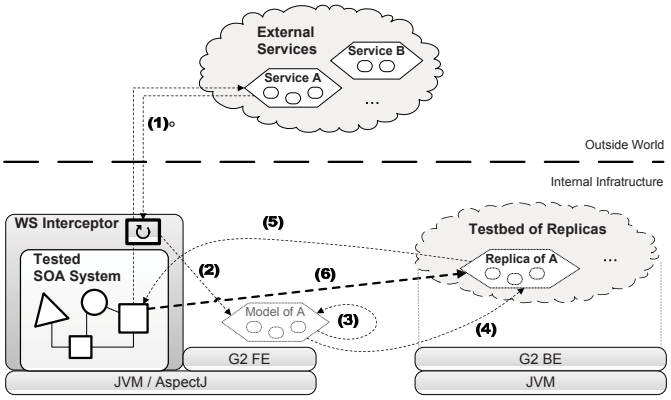


Fig. 1. Web service call interception and generation of replicas

In the next sections we explain the concepts behind each step and give an overview about the extendability and programmability of the testbeds. We start with the Genesis2 (G2) testbed generator framework, which is the base grounding of our approach, providing functionality to generate testbeds on-demand.

A. Genesis2 Testbed Generator Framework

The purpose of the G2 framework is to support the setup of testbeds for SOA. It emulates environments consisting of services, clients, registries, and other SOA components and supports programming of their behavior. G2's most distinct feature is its ability to generate running testbed instances and to integrate these into existing SOA environments which empowers testers to evaluate SOA systems at runtime.

G2 comprises a centralized front-end, from where testbeds are modeled and controlled, and a distributed back-end, consisting of hosts at which the models are transformed into real testbed instances. The front-end maintains a virtual view on the testbed, for on-the-fly manipulations via scripts, and propagates changes to the back-end in order to adapt the running testbed. For the sake of extensibility, G2 uses composable plugins which augment the testbed's functionality, making it possible to emulate diverse topologies, functional and non-functional properties, and behavior. Figure 2 depicts a simplified view on the different layers of a G2-based testbed and the interactions within them. At the two bottom layers, G2 connects the front-end to the distributed back-end and installed plugins establish their communication structures. Most important are the two top layers. Based on the provided model schema, the engineer creates models of SOA components which are then being generated and deployed at the back-end hosts. At the very top layer, the testbed instances are running and behave/interact according to the specification. The aggregation of these instances constitutes the actual testbed infrastructure on which the developed SOA can be evaluated.

In summary, at the front-end the engineer specifies via Groovy scripts [8] the details of the testbed, defining *what* shall be generated *where*, with *which customizations*, and the framework takes care of synchronizing the model with the

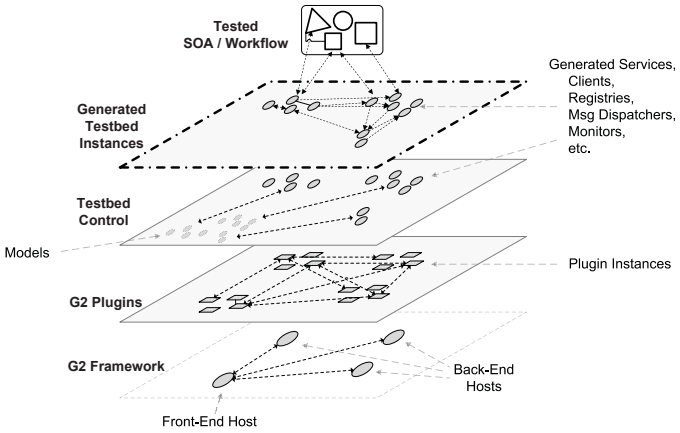


Fig. 2. Interactions within G2 layers

corresponding back-end hosts on which the testbed elements are generated and deployed. The following snippet contains a sample script for modeling a Web service, programming it's behavior (in this case just returning a String value), and deploying it at a back-end host. After deployment, it is possible to perform adaptations on-the-fly by changing the Web service's model which is immediately propagated to the generated instance at the back-end.

```
1  // import reference of cloud back-end host
2  def beHost1 = host.create("someHost.domain",8080)
3
4  // import data type from XSD file
5  def dt = datatype.create("/path/types.xsd","vCard")
6
7  // create model of TestService with one operation
8  def service = webservice.build {
9
10   TestService(binding: "doc,lit") {
11     SayHi(card: vCard, result: String) {
12       return "hi ${card.name}"
13     }
14   }
15 }[0]
16
17 service.deployAt(beHost1) // deployment
18
19 service.operations+= ... // on-the-fly adaptation
```

For detailed information about the Genesis2 framework and on our concepts for testbed generation we refer interested readers to our previous works [1], [2], [9].

B. AOP-based Interception of Web service Invocations

Aspect-oriented programming (AOP) is a paradigm which aims to increase software modularity by allowing the separation of cross-cutting concerns [4]. In particular, AOP allows developers to alter the behavior of a system (in terms of enhancing but also replacing functionality) at runtime by specifying pointcuts and join points (intercepted points/functions in a program) and executing advices (new/additional behavior) on them. For example, a developer can define aspects for enhancing a program with logging functionality. Up to date, AOP has gained high popularity among software engineers and numerous programming languages provide support for it. However in this paper we have concentrated on Java and on its AOP extension called AspectJ [3], due to Java's importance for Web (service) engineering.

In our approach we apply AOP for intercepting Web service invocations, create replica services in a testbed, and redirect the invocations to these, as depicted in Figure 1. To be precise, we intercept the retrieval of WSDL documents, as this is the first step of a typical invocation procedure [10]: WSDLs are retrieved from a registry and then passed to a client generator which creates the corresponding invocation stubs. Here we intervene in the program flow. Before the client generator processes the WSDL we analyze it in order to create a replica of the service in a dedicated testbed infrastructure (replica creation is explained in the next section). The next step is to replace transparently the original WSDL with the one of the replica, in order to make the client generator create stubs which point to the new endpoint and, eventually, to direct all invocations to it. This way, the original service is only contacted for retrieving its description but all communication is actually done with its replica in the testbed.

The interception of WSDL retrieval is realized via aspects which detect calls of the retrieval routines in the SOA system and by altering their execution. Depending on which Web service framework the system is using (e.g., Apache Axis 2 [11], Apache CXF [12], or GlassFish [13]) different aspects must be applied in order to match the corresponding API functions. The following code snippet shows a simple aspect which covers the WSDL4J [14] library/framework, used by various workflow engines. At first a pointcut is defined which catches calls of the method `WSDLReader.readWSDL(String)`. If this pointcut is matched at runtime, the following advice will be executed. It passes the URI to the `Sandbox` utility class that initiates the replication of the service in the background and returns the URI to the replica's WSDL. Finally, the intercepted method is called, however, the URI of the original WSDL is replaced with the one of the replica.

```
public aspect InterceptorAspect {

  // pointcut defining intercepted functions
  public pointcut ret(String uri, WSDLReader r) :
         call(* WSDLReader.readWSDL(String)) &&
         args(uri) &&
         target(r);

  // advice being executed when pointcut fires
  Definition around(String uri, WSDLReader r :
            ret(uri,r) {

    String repURI=Sandbox.createReplica(uri);
    return proceed(repUri,r);
  }
}
```

Having such aspects defined, a Java-bsed SOA system can be monitored at runtime by executing it on top of AspectJ which takes care of weaving the aspects into the running code (referred to as load-time weaving) and which delegates the generation of replicas to the `Sandbox` generator.

C. On-the-fly Generation of Service Replicas

In a nutshell, the process of replicating Web services comprises the following three main steps:

1) the remote Web service is analyzed and a basic model of a replica service is created,
2) the model is subject to user-defined customizations, and
3) the final model is transferred to a back-end host where a running Web service is generated out of it.

Steps 1 and 3 are fully automated and no user interactions are required in these. The only semi-automated part resides in step 2 where engineers can define own rule-based customizations for the generated services. Though, at runtime the customizations are applied automatically, which results in an automated overall execution of the replication process.

1) Creation of a Replica Model: The G2 testbed generator supports modelling of Web services via the framework's API or via the Groovy-based scripting language (like in Listing 1) which, in turn, is simply a user-friendly front-end for the API. Moreover, G2 provides a model schema (see Figure 3) which specifies a) which component model types are available, b) which customizations can be performed to these, and c) how these components can be composed into a coherent testbed infrastructure. In short, the schema represents the framework's functionality and acts as a template for modelling testbeds.

For creating the model of a replica service, our tool retrieves the original service's WSDL document in order to analyze the interface and to clone it in the model. Even though the analysis of WSDL documents could be done in a "raw" manner by processing the document directly, for instance by using the WSDL4J library [14], we apply the `wsimport` utility of JAX-WS [15] for convenience. `wsimport` takes WSDL's as input and generates corresponding Java stubs, imports referenced XML schemas automatically, checks for WS-I [16] compatibility, and performs various other necessary steps which would otherwise have to be done manually. Our tool takes the generated stubs and compiles them which provides us a binary representation of the service's interface which can be analyzed via object reflection techniques plus by checking the corresponding Java annotations. This way, we can easily determine the service's operation signatures, message types, binding information, and all the other required data for modelling a replica, and translate this data into the model representation. The next step is to perform user-defined customizations to the model.

2) Customization, Extensibility, and Programmability: The result of the first step is a model of a replica service which clones the original service's interface, however, does not provide any (proper) functionality. On an invocation it delivers response messages which are syntactically correct, according to the XML schema definition (XSD) [17] in the WSDL document, but simply fills these messages with randomized data, which makes them semantically meaningless. Therefore, at this stage the replica can be regarded as a pure dummy or mock-up service. For some test cases mock-ups are perfectly sufficient, for instance if the engineer wants to determine how many parallel service invocations his/her tested SOA system can handle, regardless of the response message content. But if the test cases become more sophisticated and message content does matter, or if the services must expose certain functional or non-functional behavior, then the replica models must be customized in order to meet these requirements and to expose the desired behavior. This is a part of the replication process that we cannot automate, as requirements vary with every tested SOA system and the purpose of the particular test case. Furthermore, it is impossible to replicate a remote service's functionality automatically and to provide 100% realism, as explained in Section V. But what we can do is to support engineers in performing the necessary customizations in order to replicate the behavior *as good as possible and/or required for the tests*! For this purpose we make use of G2's extensibility via plugins and, again, apply Groovy scripts for specifying rules for how generated replica services are customized. Plugins enhance the framework's functionality, in terms of what SOA components can be generated, plus they can augment already available component types. For example, in [2] we presented techniques for enhancing SOA testbeds with fault injection (FI) on three levels: 1) on the network level for hampering the packet flow, 2) on the service execution level for emulating degraded quality of service (QoS), and 3) on the message level for corrupting the content of SOAP messages. Each FI level was realized in a separate plugin that enhanced the basic model schema of G2 with a model type for specifying the fault behavior (depicted in Figure 3).

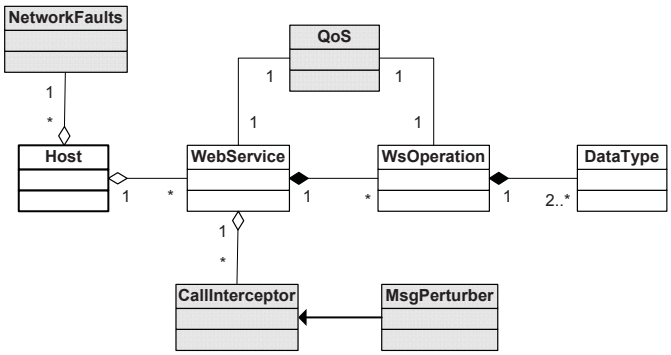


Fig. 3. Base Web service model schema with fault-injection extensions [2]

The following Listing demonstrates two sample rules for customizing replica services with FI plus for assigning functional behavior to the service's operations. The rules are wrapped in a Groovy closure [18] which takes the service model as parameter (`svc`). The first rule matches services by their namespace (Line 4) and augments them with a QoS model that simulates changing availability. In Lines 6-14 the QoS model is instantiated, bound to the service, and the availability behavior is programmed. The second rule matches services by their declared name plus by whether they contain a certain operation. In case of a match, the operation's behavior is programmed to return prepared responses from a repository containing replay data.

```
serviceCustomizer = { svc ->

  // customize service which has "infosys" in its NS
  if (svc.namespace =~ /infosys.ac.at/) {
    // instantiate QOS model and attach to service
    def svcQos = qos.create()
    svc.qos = scvQoS
    // program availability behavior model
    svcQos.availability = {
      if (new Date().getHours()<8) { // till 8 AM
        return 99/100 //set high availability of 99%
      }
      return 90/100 //otherwise, set lower rate
    }
  }

  // customize service with a particular operation
  if (svc.name == "CustomerSVC" &&
      "GetCustomerData" in svc.operations.name) {
    // program behavior of operation via closure
    svc.getOperation("GetCustomerData").behavior = {
      def response = rrRepo.get(request)
      return response
    }
  }

}
```

All in all, engineers are given a possibility to perform arbitrary customizations to the generated replica models and to use the full potential of G2, in terms of extensibility and programmability [1].

3) Generating Web service Instances in the Back-end: After the replica model has been generated in step 1 and customized in step 2, the next step is to transfer it to the back-end for generating a running Web service instance in the testbed. G2 testbeds are usually hosted on a distributed back-end infrastructure, composed of a set of hosts on which the testbed components are deployed. Therefore, a destination host must be chosen. Similar to specifying customization rules for services, engineers can define rules telling where to deploy the replica services. The following snippet shows a simple configuration which references 10 hosts in the back-end (from 192.168.1.1:8080 to 192.168.1.10:8080) and for each given replica service it choses a random one.

```
hostList = []

1.upto(10) { n-> // create list of 10 host refs
  hostList += host.create("192.168.1.$n",8080)
}

hostChoser = { svc -> // simply pick a random host
  def pos = new Random().nextInt(hostList.size())
  return hostList[pos]
}
```

The process of generating Web service instances out of the models comprises the serialization of the model, transferring it to the G2 instance at the choses back-end host, and, finally, the translation of the model into a running and deployable Web service. The first two steps are trivial but the generation of Web services from models is far more sophisticated, being based on generative programming. Once the model has been received at the back-end instance, the following steps are executed:

1) Recursive analysis of the `WebService` model to determine used customization plugins and message types.
2) Translation of message types (`DataType` models) to Java classes that represent the XSD-based data structures (using `xjc`, the Java XML Binding Compiler).
3) Automatic generation of Java/Groovy source code implementing the modeled Web service.
4) Compilation of sources using Groovy's built-in compiler.
5) Generation of customizations by corresponding plugins.
6) Deployment of completed Web service instance at local Apache CXF [12] endpoint.

For more details about the generation of Web services, we refer to [1] and [19].

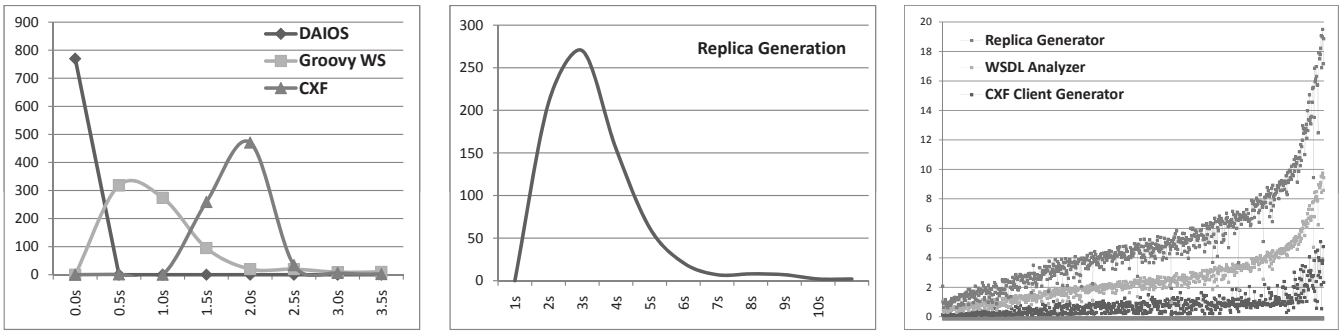
All in all, the result of the whole replication process is a running Web service that clones the original service's interface, behaves according to the engineer's customizations, and can be used for testing purposes. Yet, some readers might wonder about the applicability of this approach, the realism of such a testbed, and what kind of drawbacks exist. We are trying to answer these questions in Section V.

IV. Evaluation

To prove the quality of our approach, it would be necessary to evaluate its practical usefulness as well as its intrusiveness into the tested SOA system. However, the evaluation of usefulness would require the application of our approach in a significant number of SOA development projects in order to determine the usability, convenience, as well as how much time was saved in the development/testing process. Due to a lack of access to a sufficient number of ongoing development projects, we were not able to perform this kind of evaluation. Instead we concentrated on evaluating the level of intrusiveness. As our approach does alter the execution of the SOA system, it is important to find out how much the altered runtime differs from the original one, in terms of performance degradation. The goal is to keep the intrusiveness and the changes at runtime as small as possible.

In our evaluation we have applied our approach on intercepting dynamic binding and invocation of Web services. We agree with [20] that dynamic binding combined with message-based interactions is the proper way to implement SOA communication. SOA systems should be able to adapt to its environment and redirect invocations to "better suited" Web services at runtime, instead of being tightly bound to particular services. This requires that the invocation stubs, that handle the communication with the remote services, are not hard-coded, but are being generated dynamically out of the services' WSDL descriptions. In our evaluation we have used three different Web service frameworks/libraries which are capable of dynamic binding:

1) *Apache CXF* [21] is a rich framework, supporting SOAP [5], ReST [22], WS-* extenstions, etc. Implements the JAX-WS API [15] for Web service development.
2) The *Groovy SOAP Module* [23] provides SOAP support for the Groovy language [8]. Strongly focused on simplicity and usage convenience, less on feature support.



(a) Processing time distribution of tested client generators

(b) Processing time distribution of Web service replica generator

(c) Accumulated processing time of intercepted client generator

Fig. 4. Processing time evaluation - measuring the performance penalties of intercepting Web service invocations

3) *DAIOS* [20] aims at supporting truly dynamic interactions between clients and services. Provides composition of request messages automatically via similarity metrics between request data and WSDL contract, improving loose coupling this way.

For testing these client generators, we took the QWS dataset [24] that is basically a collection of WSDL documents of public SOAP Web services. In total it contains 2505 WSDLs, however, we were not able to use all of these for our purposes. First of all, 31% of the documents are not WS-I-compatible as they use RPC/encoded bindings or other WSDL styles which have been regarded as deprecated [16]. Furthermore, 15% of the remaining WSDLs were corrupted, for instance by referencing not existing XSD definitions or by having an invalid document structure. Moreover, we also removed 12% of valid WSDLs as they were referencing remote artifacts which had to be downloaded with sometimes significant latency which delayed and, therefore, distorted the whole processing time of the client generators. The remaining WSDLs were considered as suitable for executing our evaluation.

To determine the performance intrusiveness, we calculated how fast a client generator actually generates the corresponding stub code and how much this procedure is delayed if we apply the testbed generator aspects. We did not measure the time for eventually invoking the services on purpose, for two reasons: 1) the main load is caused during the analysis of the WSDL and the generation of the stubs, and only minimal load is caused during the actual invocation, and 2) the invocation is mainly delayed by the processing time of the remote Web service (referred to as QoS) and does not depend on the client generator at all.

Figure 4(a) visualizes the distribution of processing times for the three testes client generators, while Figure 4(b) shows the processing time of the replica generator for comparison. The performance of the client generators differs significantly, due to their level of (pre-)processing of Web service calls. For instance, DAIOS does not generate and compile code into Java classes but simply analyzes the content of the WSDL document. This makes DAIOS, by far, the fastest client generator. Groovy SOAP and Apache CXF, however, take the WSDL document as input for generating stub classes and for translating message types into Java classes. Even though this provides several benefits, it comes at the cost of performance. For the set of WSDL document of the QWS dataset we measured an average processing time of 12 msec for DAIOS, 871 msec for Groovy SOAP, and 1622 msec for CXF[1]. For the replica generator we measured an average processing time of 4617 msec which is significantly slower than the client generators. As a consequence, if our testbed generator approach is applied on a SOA system that uses dynamic binding, the following delays will occur for the generation of clients: DAIOS-based calls would be slowed down by a factor of 384 - which is, however, exceptional as DAIOS performs only restricted processing of the WSDL -, for Groovy SOAP the slow down factor would by 6.3, and for CXF it would be 3.8. Of course, this delay happens only *once*, namely when the client stub gets generated during the first invocation of the service. Each subsequent call will not be delayed, as the replica needs to be created only once.

For a better visualization of the delay caused by replication, we measured the performance of the individual steps of a Web service call interception: a) of the client generator, b) of the WSDL analyzer, and c) of the actual replica generator. Figure 4(c) displays the results. Again we performed the tests on the QWS dataset, sorted the WSDLs according to their size/complexity, and this time we used only Apache CXF as the base client generator. The bottom graph displays the performance of CXF, which is quite constant except for the very complex WSDL's. The middle graph shows the accumulated performance of the WSDL analyzer, in addition to CXF' processing time. On the top, the graph displays the sum of all three modules, which constitutes the actual processing time for intercepting a Web service call plus for generating the corresponding replica service. The results show that approximately 30% of the time is consumed for analyzing the remote WSDL and creating the model, 44% is spent on

[1]The measurements were performed on an Intel i5 M520 CPU with 2.4GHz. However, we mainly compare the relative performance difference between the WS frameworks and the replica generator, which renders the system's hardware secondary.

generating a running replica Web service out of it, while the generation of the local client stubs takes only 26% of the time.

These tests demonstrate that intercepting Web service calls for the purpose of generating replicas and redirecting the calls to them does significantly slow down the client generation process. What does that mean for the level of performance intrusiveness and how does that affect the whole system's runtime and, consequently, the testing process? First of all, this evaluation represents the current implementation status and without doubt further optimizations would be possible. But still, the performance degradation would be noticeable, even if only at the client generation and not at the invocations of a Web service. Though, how much this slowdown really affects the testing of an SOA system depends on many factors: on the applied Web service framework, on the number of Web services being invoked and on their complexity, on whether the invocations are blocking or asynchronous, etc. In a nutshell, many factors play a role when determining how much this all affects the systems performance and whether this matters for the tests at all. There are many scenarios where it makes sense to replicate external Web services on-the-fly, while for some it is not reasonable. We try to draw up the main concerns and pro/contra arguments for/against our approach in the following section.

V. Discussion

The usage of testbeds and, especially, the replication of existing Web services raises questions about the applicability of the concept, whether it is possible to replicate SOA environments in a realistic way, and various other concerns which deserve to be discussed in more depth. In the following we give our opinion on the questions we regard as most relevant.

How realistic are such testbeds? What about replication of functional behavior?
To make it short: replication of functional behavior, without having access to the remote Web service's code, is not possible. Let's consider the example of a complex and stateful remote service which performs sophisticated calculations, uses a data base system as data source, interacts with some legacy components, etc. There is no way of replicating its functional behavior in 100% if one has only access to the Web service's interface description. There are means to record and playback Web service interactions, e.g., [25], [26], but still the replication of functionality is only limited to the recorded data. That is something we have to live with. But what is a SOA software engineer supposed to do, if he/she is developing a system that must interact with such a complex service, but he/she is not allowed to use this service for testing purposes? Basically, his/her hands are bound and the best solution is to use a replica of the service in a testbed. And for this replica, it is up to the developer to implement a "realistic" behavior according to the requirements of the test run. Often it is sufficient to emulate just QoS properties (e.g., by taking over the collected QoS data from the QWS dataset), or to implement a rudimentary clone of the services functionality, etc. It all depends on how much is known about the original service and how much of it must be replicated. We cannot solve this problem in a *fully automated* manner, but we can support developers by replicating the service's interface and by providing means for assigning functional behavior to them, as shown in [1]. So the answer is that it is the task of the testing engineer to assign *sufficiently realistic* behavior to the replicas. We have no possibility to unburden him/her from that.

Does it really make sense to generate replicas on-the-fly at runtime, instead of having a pre-generated testbed?
In our previous work we argued that testbeds need to be generated *before* the test runs, mostly by specifying all details via G2's scripting language and deploying the testbed on a back-end. In the current paper we try to convince readers about generating testbeds *on-thy-fly*, which is in general the exact opposite. Why is that? First of all, in the previous papers we described the current state and on-the-fly generation had not been developed back then. But now, as it is possible, is it always preferable? In our opinion it makes sense if the testbed composition is not known a-priori or a generation of the complete testbed infrastructure is not reasonable, e.g., because it could get too large-scale even though only a few services of it would be actually invoked. For such scenarios on-the-fly generation is more efficient and, therefore, preferable.

How does the presented approach support engineers at running the tests or at evaluating test results?
In the current state, there is almost no support at all as we regard it out of scope. The current focus is is only on testbed generation, which is still work in progress as many challenges have not been solved yet. Regarding the execution of the test runs and the evaluation of test results (e.g., logs) we do not provide any support so far, however, we will tackle these challenges in our future research projects.

VI. Related Research

Several research groups have investigated the generation of testbed infrastructures for SOA, yet we have not discovered any works which would be similar to our approach of creating testbeds by intercepting Web service invocations. In spite of that, some works have brought forward research on automated generation of testbeds, they have developed concepts related to ours, and are, therefore, relevant for comparison.

For instance, SOABench [27] provides sophisticated support for benchmarking of BPEL engines [28] via modeling experiments and generating service-based testbeds. It provides runtime control on test executions as well as mechanisms for test result evaluation. Regarding its features, SOABench is focused on performance evaluation and generates Web service stubs that emulate QoS properties, such as response time and throughput. The testbeds are generated from BPEL models and from the WSDL documents of the referenced Web services.

Like in SOABench, the authors of PUPPET [29] examine the generation of QoS-enriched testbeds for service compositions. PUPPET does not investigate the performance but verifies the fulfillment of Service Level Agreements (SLA) of composite services. This is done by analyzing WSDL and WS-Agreement documents [30], generating testbeds out of it,

and emulating the QoS of the generated Web services in order to check the SLAs.

Both tools, SOABench and Puppet, have in common with our approach that they support the generation of Web service-based testbeds by reading in WSDL documents and generating stubs from these, plus extend these in order to emulate QoS properties, e.g., by delaying the execution. The main difference is, however, that they rely on an a-priori knowledge about the testbed environment, e.g., the set of required Web services. Our approach is more focused on dynamic SOA systems that determine at runtime which services they invoke and, therefore, testbeds for them cannot be built in a static manner.

An other solution is provided with soapUI [26], a software for functional testing of Web services and clients. It supports the generation of mock-up services and the customization of these with functional behavior. However, soapUI does also rely on a-priori specifications and is focused on testing single services or clients, not supporting a convenient generation of large-scale testbeds.

Apart from these related works, no more research has been published on testbed generation for SOA. As shown in recent detailed surveys on Web service's testing, such as [31], the research community has rather concentrated on testing single services, e.g., [32], [33], or composite ones, e.g., [34], [35], but has neglected too much the necessity of testbed infrastructures.

VII. Conclusion

In this paper we have presented a technique for generating testbeds for Web service-based systems automatically by applying aspect-oriented programming. We define aspects via which we intercept the invocation of Web services in a running SOA system, generate replicas of the services on-the-fly, and redirect the initial invocations to the replicas in a transparent manner. With this work we have extended our research on SOA testbed generation towards more automation, as testers are not required anymore to specify all Web services of a testbed manually but this task is performed by our tool.

Of course, our approach is not able to generate fully functional replicas in a perfectly automated manner. Still testers need to specify the functional behavior in our scripting language. However, we can reduce the amount of required specification for setting up testbeds significantly and, therefore, can speed up the whole process of testing SOA systems.

Acknowledgment

The research leading to these results has received funding from the European Community 7th Programme FP7/2007-2013 under grant agreement 215483 (project *S-Cube*) and from the Austrian Science Fund (FWF) via the project *Audit4SOAs*.

The authors would also like to thank Matej Pavlovic for implementing parts of the WSDL analyzer code.

References

[1] L. Juszczyk and S. Dustdar, "Script-based generation of dynamic testbeds for soa," in *ICWS*. IEEE Computer Society, 2010, pp. 195–202.

[2] ——, "Programmable fault injection testbeds for complex soa," in *ICSOC*, ser. Lecture Notes in Computer Science, vol. 6470, 2010, pp. 411–425.

[3] "AspectJ - Project Web site," http://www.eclipse.org/aspectj/.

[4] G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 4es, p. 154, 1996.

[5] "SOAP - W3C Specification," http://www.w3.org/TR/soap/.

[6] "Web Services Description Language (WSDL) - W3C Specification," http://www.w3.org/TR/wsdl.

[7] "List of Web Service Specifications - Wikipedia article (accessed on Mar 23, 2010)," http://en.wikipedia.org/wiki/List_of_web_service_specifications.

[8] "Groovy - Project Web site," http://groovy.codehaus.org/.

[9] "Genesis Testbed Generator - Prototype Web site," http://www.infosys.tuwien.ac.at/prototype/Genesis/.

[10] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards recovering the broken soa triangle: a software engineering perspective," in *IW-SOSWE*. ACM, 2007, pp. 22–28.

[11] "Apache Axis 2 - Project Web site," http://axis.apache.org/axis2/.

[12] "Apache CXF - Project Web site," http://cxf.apache.org.

[13] "GlassFish - Web site," http://http://glassfish.java.net/.

[14] "WSDL4J - Project Web site," http://sourceforge.net/projects/wsdl4j/.

[15] "JAX-WS - Project Web site," http://jax-ws.java.net/.

[16] "Web Services Interoperability Orgnization (WS-I) - Web site," http://www.ws-i.org/.

[17] "XML Schema Definition (XSD) - Primer / Specification," http://www.w3.org/TR/xmlschema-0/.

[18] "Groovy Closures - User Guide Web site," http://groovy.codehaus.org/Closures.

[19] L. Juszczyk, H. L. Truong, and S. Dustdar, "Genesis - a framework for automatic generation and steering of testbeds of complex web services," in *ICECCS*. IEEE Computer Society, 2008, pp. 131–140.

[20] P. Leitner, F. Rosenberg, and S. Dustdar, "Daios: Efficient dynamic web service invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 72–80, 2009.

[21] "Apache CXF Dynamic Clients - Documentation Web site," http://cxf.apache.org/docs/dynamic-clients.html.

[22] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, Irvine, California, 2000.

[23] "Groovy SOAP Web Services - User Guide Web site," http://groovy.codehaus.org/Groovy\%2BSOAP.

[24] E. Al-Masri and Q. H. Mahmoud, "Qos-based discovery and ranking of web services," in *ICCCN*. IEEE, 2007, pp. 529–534.

[25] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella, "Tulafale: A security tool for web services," in *FMCO*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 3188. Springer, 2003, pp. 197–222.

[26] "soapUI - Product Web site," http://www.soapui.org/.

[27] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware: a case study on bpel engines," in *WWW*. ACM, 2010, pp. 141–150.

[28] "Business Process Execution Language for Web Services (WS-BPEL) - OASIS Standard Web site," http://www.oasis-open.org/committees/wsbpel/.

[29] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in *TestCom/FATES*, ser. Lecture Notes in Computer Science, vol. 5047. Springer, 2008, pp. 266–282.

[30] "Web Services Agreement - Specification," http://www.ogf.org/documents/GFD.107.pdf.

[31] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing web services: A survey," Department of Computer Science, King's College London, Tech. Rep. TR-10-01, January 2010.

[32] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping performance and dependability attributes of web services," in *ICWS*. IEEE Computer Society, 2006, pp. 205–212.

[33] W. Xu, J. Offutt, and J. Luo, "Testing web services by xml perturbation," in *ISSRE*. IEEE Computer Society, 2005, pp. 257–266.

[34] H. Huang, W.-T. Tsai, R. A. Paul, and Y. Chen, "Automated model checking and testing for composite web services," in *ISORC*. IEEE Computer Society, 2005, pp. 300–307.

[35] H. J. A. Holanda, G. C. Barroso, and A. de Barros Serra, "Spews: A framework for the performance analysis of web services orchestrated with bpel4ws," in *ICIW*. IEEE Computer Society, 2009, pp. 363–369.