Script-based Generation of Dynamic Testbeds for SOA

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

***Abstract*—This paper addresses one of the major problems of SOA software development: the lack of support for testing complex service-oriented systems. The research community has developed various means for checking individual Web services but has not come up with satisfactory solutions for testing systems that operate in service-based environments and, therefore, need realistic testbeds for evaluating their quality. We regard this as an unnecessary burden for SOA engineers. As a proposed solution for this issue, we present the Genesis2 testbed generator framework. Genesis2 supports engineers in modeling testbeds and programming their behavior. Out of these models it generates running instances of Web services, clients, registries, and other entities in order to emulate realistic SOA environments. By generating real testbeds, our approach assists engineers in performing runtime tests of their systems and particular focus has been put on the framework's extensibility to allow the emulation of arbitrarily complex environments. Furthermore, by exploiting the advantages of the Groovy language, Genesis2 provides an intuitive yet powerful scripting interface for testbed control.**

I. Introduction

In the last years, the principles of Service-oriented Architecture (SOA) have gained high momentum in distributed systems research and wide acceptance in software industry. The reasons for this trend are SOA's advantages in terms of communication interoperability, loose coupling between clients and services, reusability and composability of services, and many more. Moreover, novel features which are associated with SOA [1] are adaptivity [2], self-optimization and self-healing (self-* in general) [3], and autonomic behavior [4]. The result of this evolution is that, on the one hand, SOA is being increasingly used for building distributed systems, but, on the other hand, is becoming more and more complex itself. As complexity implies error-proneness as well as the need to understand how and where such complexity emerges, SOA-based systems must be tested intensively during the whole development process and, therefore, require realistic testbeds. These testbeds must comprise emulated Web services, clients, registries, bus systems, mediators, and other SOA components, to simulate real world scenarios. However, due to missing tool support, the set up of such testbeds has been a major burden for SOA engineers. In general, the lack of proper testing support has been regarded as one of the main problems of SOA [5]. Looking at currently available solutions, it becomes evident that the majority aims only at testing of single Web services [6], [7], [8] and composite ones [9], [10] which, however, only covers the service provider part of SOA. For testing systems which operate in service-based environments themselves, the engineer is facing the problem of setting up realistic test scenarios which cover the system's whole functionality. There do exist solutions for testbed generation but these are restricted to specific domains, e.g., for checking Service Level Agreements by emulating Quality of Service [11]. However, if engineers need generic support for creating customized testbeds covering various aspects of SOA, no solutions exist to our knowledge which would relieve them from this time-consuming task. We believe, this issue is a severe drawback for the development of complex SOAs.

In this paper we present the current state of our work on a solution for this issue. We introduce the Genesis2 framework (*Gene*rating *S*OA Testbed *I*nfrastructure*s*, in short, G2) which allows to set up SOA testbeds and to manipulate their structure and behavior on-the-fly. It comprises a front-end from where testbeds are specified and a distributed back-end on which the generated testbed is hosted. At the front-end, engineers write Groovy scripts to model the entities of the testbed and to program their behavior, while the back-end interprets the model and generates real instances out of it. To ensure 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.

The rest of the paper presents our work as follows. In Section II we give an overview of related research. Section III is the main part of the paper and describes the concepts of the G2 framework. Section IV demonstrates the application of G2 via a sample scenario. Finally, sections V and VI discuss open issues, present our plans for future work, and conclude.

II. SOA Testbeds

Comparing the state of the art of research on SOA in general and the research on testing in/for SOA, an interesting divergence becomes evident. SOA itself has had an impressive evolution in the last years. At its beginning, Web service-based SOA had been mistaken as yet another implementation for distributed objects and RPC and, therefore, had been abused for direct and tightly-coupled communication [12]. After clearing up these misconceptions and pointing out its benefits derived from decoupling, SOA has been accepted as an architectural style for realizing flexible document-oriented distributed computing. Today's SOAs comprise much more than just services, clients and brokers (as depicted in the

outdated Web service triangle [13]) but also message mediators, service buses, monitors, management and governance systems, workflow engines, and many more [1]. As a consequence, SOA is becoming increasingly powerful but also increasingly complex, which implies higher error-proneness [14] and, logically, requires thorough testing. But looking at available solutions for SOA testing (research prototypes as well as commercial products), one might get the feeling that SOA is still reduced to its find-bind-invoke interactions because most approaches deal only with testing of individual Web services, and only few solutions deal to some extent with complex SOAs. All in all, it is possible to test whether a single Web service behaves correctly regarding its functional and non-functional properties, but testing systems operating on a whole environment of services is currently not supported. Let us take the case of an autonomic workflow engine [15] for example. The engine must monitor available services, analyze their status, and decide whether to adapt running workflows. To verify the engines' correct execution it is necessary to perform runtime tests in a real(-istic) service environment, in short, a service testbed. The testbed must comprise running instances of all participants (in this simple case only Web services), emulate realistic behavior (e.g., Quality of service, dependencies among services), and serve as an infrastructure on which the developed system can be tested. Of course, for more complex systems, more complex testbeds are required to emulate all characteristics of the destination environment. But how do engineers create such testbeds? Unfortunately, up to now, they had to create them manually, as no proper support had been available. To be precise, some solutions do exist but are too restricted in their functionality and cannot create testbeds of arbitrarily complex structure and behavior. This has been our motivation for doing research on supporting the generation of customizable testbeds for SOA. In the following, we give an overview on the current state of the art of research and discuss the evolution of Genesis since its first version.

A. Related Research on SOA Testing

Available solutions have been mostly limited to testing Web service implementations regarding their functional and non-functional properties. This includes, for instance, tests for performance and Quality of Service (QoS) [16], [7], robustness [17], reliability [8], [18], message schema conformance [19], but also techniques for testing composed services [9], [10] as well as generic and customizable testing tools [6]. In spite of their importance, these solutions only support engineers in checking the service providers of a SOA. Which means that they can be only used for testing the very basic building blocks but not the whole integrated system. This makes these works out of scope of our current research and, therefore, we do not review them in detail. Unfortunately, the challenging task of testing complex SOAs and their components, such as governance systems which operate and also depend on other services, has not gained enough attention in the research community. Some groups have done research on testbed generation but their investigations have been focused only on specific domains such as QoS or workflows.

For instance, SOABench [20] provides sophisticated support for benchmarking of BPEL engines [21] 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. Similar to SOABench, the authors of PUPPET [11] 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 [22] and WS-Agreement (WSA) documents [23] and emulating the QoS of generated Web services in order to check the SLAs. Both tools, SOABench and Puppet, support the generation of Web service-based testbeds, but both are restricted to a specific problem domain (workflows/compositions & QoS/SLA). In contrast, G2 provides generic support for generating and controlling customized testbeds. Though, if desired, G2 can be also used for emulating QoS.

Further related work has been done on tools for controlling tests of distributed systems. Weevil [24], for example, supports experiments of "distributed systems on distributed testbeds" by generating workload. It automates deployment and execution of experiments and allows to model the experiment's behavior via programs written in common programming languages linked to its workload generation library. We do not see Weevil as a direct competitor to G2, but rather as a complementary tool. While Weevil covers client-side tests of systems, G2 aims at generating testbeds. We believe that a combination of both systems would empower engineers in setting up and running sophisticated tests of complex SOAs and we will investigate this in future work.

Another possible synergy we see in combining G2 with DDSOS [25]. This framework deals with testing SOAs and provides model-and-run support for distributed simulation, multi-agent simulation, and an automated scenario code generator creating executable tests. Again, this framework could be used to control tests on G2-based testbeds.

B. Evolution of Genesis

Our work on SOA testbeds had first led to the development of Genesis [26] (in short, G1), the predecessor of G2. To our knowledge, G1 was the first available "multi purpose" testbed generator for SOA and we have published the prototype as open-source [27]. Similar to G2, it is a Java-based framework for specifying properties of SOAP-based Web services [28] and for generating real instances of these on a distributed backend. Via a plug-in facility the service testbed can be enhanced with complex behavior (e.g., QoS, topology changes) and, furthermore, can be controlled remotely by changing plugin parameters. At the front-end, the framework offers an API which can be integrated, for instance, into the Bean Scripting Framework (BSF) [29] for a convenient usage. However, G1 suffers from various restrictions which limit the framework's

functionality and usability. First of all, the behavior of Web services is specified by aligning plugin invocations in simple structures (sequential, parallel, try/catch) without having fine-grained control. This makes it hard to implement, for instance, fault injection on a message level [19]. Also, deployed testbeds can only be updated by altering one Web service at a time, which hampers the control of large-scale testbeds. Moreover, G1 is focused on Web services and does not offer the generation of other SOA components, such as clients or registries.

In spite of G1's novel features, we regarded the listed shortcomings as an obstacle for further research and preferred to work on a new prototype. By learning from our experiences, we determined new requirements for SOA testbed generators:

- customizable control on structure, composition, and behavior of testbeds,
- ability to generate not only Web services, but also other SOA components,
- ability to create and control large-scale testbeds in an efficient manner, supporting multicast-like updates,
- and, furthermore, a more convenient and intuitive way for modeling and programming the testbed.

The appearance of the listed requirements made it necessary to redesign Genesis and to rethink its concepts. These efforts resulted in our new framework, Genesis2.

III. The Genesis2 Testbed Generator

Due to the breadth of G2, it is not feasible to introduce the whole spectrum of concepts and features in a single paper. Hence, we concentrate on the most relevant novelties and present an overall picture of our framework and its application. We give an overview on G2's capabilities, explain shortly how testbeds are generated and how G2 benefits from the Groovy language, and introduce the feature of multicast-based updates for managing large-scale testbeds.

To avoid ambiguities, we are using the following terminology: *model schema* for the syntax and semantics of a testbed specification, *model types* for the single elements of a schema, *model* for the actual testbed specification, *testbed (instance)* for the whole generated testbed environment consisting of individual *testbed elements*, such as services, registries, etc.

A. Basic Concepts and Architecture

G2 comprises a centralized front-end, from where testbeds are modeled and controlled, and a distributed back-end at which the models are transformed into real testbed instances. In a nutshell, the front-end maintains a virtual view on the testbed, allows engineers to manipulate it via scripts, and propagates changes to the back-end in order to adapt the running testbed.

The G2 framework follows a modular approach and provides the functional grounding for composable plugins that implement generator functionality. The framework itself offers a) generic features for modeling and manipulating testbeds, b) extension points for plugins, c) inter-plugin communication among remote instances, and d) a runtime environment shared across the testbed. All in all, it provides the basic management and communication infrastructure which abstracts over the distributed nature of a testbed. The plugins, however, enhance the model schema by integrating custom model types and interpret these to generate deployable testbed elements at the back-end. Taking the provided `WebServiceGenerator` plugin for example, it enhances the model schema with the types `WebService`, `WsOperation`, and `DataType`, integrates them into the model structure on top of the default root element `Host` (see Figure 1), and, eventually, supports the generation of Web services at the back-end. Furthermore, the provided model types define customization points (e.g., for service binding and operation behavior) which provide the grounding for plugin composition. For instance, the `CallInterceptor` plugin attaches itself to the `WebService` type and allows to program the intercepting behavior, which will be then automatically deployed with the services.

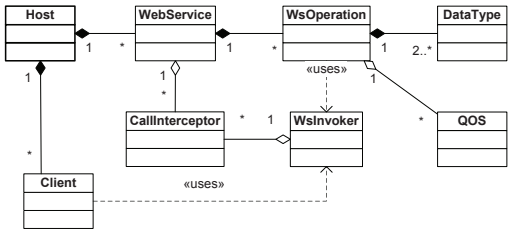

Fig. 1. Sample model schema

In G2's usage methodology, the engineer creates models according to the provided schema at the front-end, specifying *what* shall be generated *where*, with *which customizations*, and the framework takes care of synchronizing the model with the corresponding back-end hosts on which the testbed elements are generated and deployed. The front-end, moreover, maintains a permanent view on the testbed, allowing to manipulate it on-the-fly by updating its model.

For a better understanding of the internal procedures inside G2, we take a closer look at its architecture. Figure 2 depicts the layered components, comprising the base framework, installed plugins, and, on top of it, the generated testbed:

- At the very bottom, the basic runtime consists of Java, Groovy, and 3rd-party libraries.
- At the framework layer, G2 provides itself via an API and a shared runtime environment is established at which plugins and generated testbed elements can discover each other and interact. Moreover, an active repository distributes detected plugins among all hosts.
- Based on that grounding, installed plugins register themselves at the shared runtime and integrate their functionality into the framework.
- The top layer depicts the results of the engineer's activities. At the front-end he/she is operating the created testbed model. The model comprises virtual objects which act as a view on the real testbed and as proxies for manipulation commands. While, at the back-end the actual testbed is generated according to the specified model.

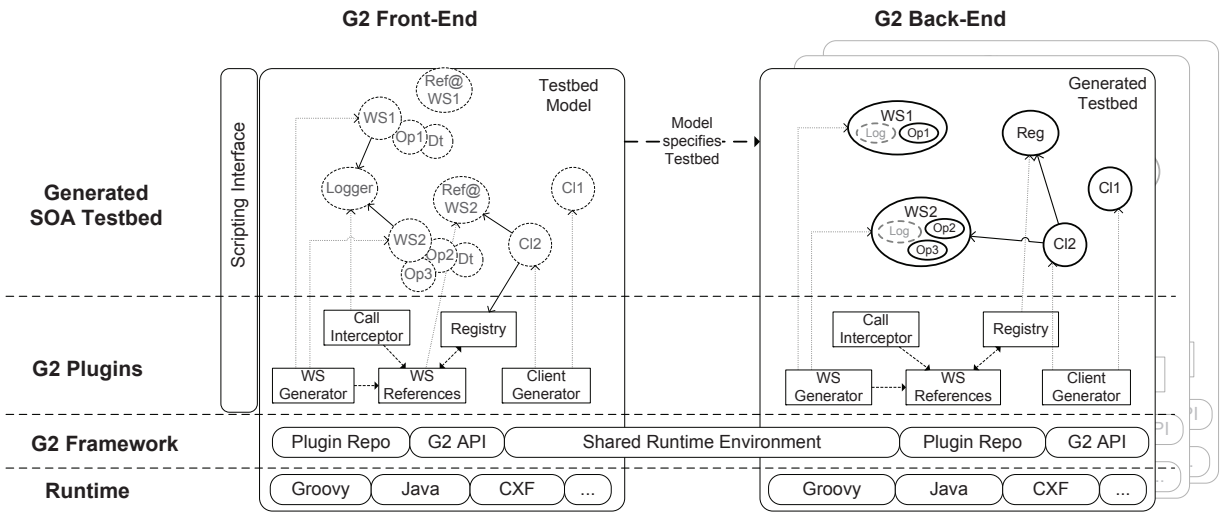

Fig. 2. Genesis2 architecture: infrastructure, plugins, and generated elements

However, Figure 2 provides a rather static image of G2, which does not represent the system's inherent dynamics. Each layer establishes its own communication structures (see Figure 3) which serve different purposes:

- On the bottom layer, the G2 framework connects the front-end to the back-end hosts and automatically distributes plugins for having a homogeneous infrastructure.
- For the plugins, G2 allows to implement custom communication behavior. For example, plugins can exchange data via undirected gossiping or, as done in the `SimpleRegistry` plugin, by directing requests (e.g., service lookups) to a dedicated instance.
- The testbed control is strictly centralized around the front-end. Each model object has its pendants in the back-end and acts as a proxy for accessing them.
- Finally, in the running testbed, G2 does not restrict the type and topology of interactions but outsources this to the plugins and their application. For instance, Web services can interact via nested invocations and, in addition, can integrate registries, workflow engines, or even already existing legacy systems into the testbed.

The framework's shared runtime environment deserves further explanation due to its importance. In G2, the SOA engineer writes Groovy scripts for modeling and programming of testbeds. The capabilities of the system, however, are defined by the applied plugins which provide custom extensions. The runtime environment constitutes a binding between these by acting as a distributed registry. Every object inside the testbed (e.g., plugin, model type, generated testbed instance, function/-macro, class, variable) is registered at the environment via aliases, in order to make it discoverable and G2 provides a homogeneous runtime infrastructure on each host. This offers high flexibility, as it ensures that locally declared scripts, which reference aliases, are also executable on remote hosts.

In the following sections we give a more detailed insight into selected features of G2 in order to convey its potential.

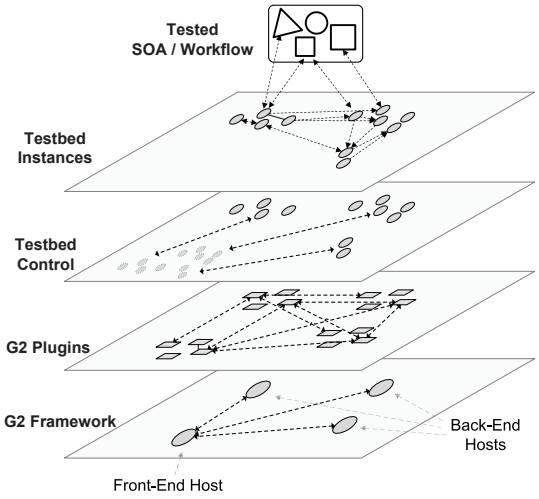

Fig. 3. Interactions within G2 layers

B. Extensible Generation of Testbed Instances

Because of its generic nature, which provides a high level of extensibility, the G2 framework outsources the generation of testbed elements to the plugins. It does also not predefine a strict methodology for how they must be generated, but rather provides supporting features. This might raise the false impression that we are just providing the base framework and leave the tricky part to the plugin developers. The truth is that we kept the framework generic on purpose, in order to have a basic grounding for future research on testbed generation, which might also include non-SOA domains. For our current needs, we have developed some plugins covering basic SOA:

- `WebServiceGenerator` creating SOAP Web services
- `WebServiceInvoker` calling remote SOAP services, both generated and preexisting ones (e.g., 3rd-party .NET-based)

- `CallInterceptor` processing SOAP calls on a message level (e.g., for fault injection [19])
- `DataPropagator` providing automated replication of data/functions among back-end hosts
- `QOSEmulator` emulating Quality of Service properties
- `SimpleRegistry` for global service lookups
- `ClientGenerator` seeding testbeds with standalone clients (e.g., for bootstrapping testbed activities)

Of these, the `WebServiceGenerator` plays a major role and, therefore, serves as a good example for demonstrating the testbed generation process. We have reused selected parts of the generation code from G1 [26], however, we were able to simplify it significantly by using Groovy features. Basically, the process comprises the following steps:

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 [30] endpoint.
7) Subscription to model changes for automatic adaptation of deployed Web service instance.

The whole generation procedure depends completely on the plugins functional purpose. For instance, the `CallInterceptor` translates intercepting code into Apache CXF Features and binds them to service and client instances, the `ClientGenerator` simply implements a programmable thread, and the `QOSEmulator` does not generate any deployable elements but works in the background.

Evidently, in G2, plugins are more than just simple extensions but provide essential features for testbed generation. They define the model schema, implement testbed capabilities, and handle the actual generation of testbed instances. Consequently, they can become quite complex. To support the implementation of new plugins, G2 provides a base class that carries out fundamental tasks for installation, deployment, and communication among remote instances, so that developers can focus on the plugin's primary features.

C. Exploitation of Groovy Features

G2 derives a great deal of its flexibility and extensibility from Groovy [31]. In short, Groovy is a dynamic programming language for the Java Virtual Machine, providing modern features such as dynamic typing, closures, and support for meta programming. Also, it has a compact syntax and can be used as an easy-to-read scripting language.

G2 uses Groovy's dynamic *Expando* type as a base class for model types. This allows to expand the model (ergo the generated testbed) on-the-fly and to observe changes, which facilitates automatic front-end/back-end synchronization. Moreover, by intercepting model manipulation requests, plugin developers can customize the handling of these (e.g., to log everything) and can restrict the model's expandability.

Internally, model objects are realized as flexible hash maps and entire testbed models are constructed by aggregating these, e.g., by attaching a `WsOperation` instance to the corresponding list inside a `WebService`'s map. However, aggregating model objects by hand is rather cumbersome and inefficient, especially for complex testbeds. As a solution, we use Groovy's *Builder* support which helps to create nested data structures in an intuitive manner. The following sample demonstrates the convenience of builders:

```
// hash map–based creation of web service model
def s1 = webservice.create("TestService")
s1.binding = "doc,lit"
s1.tags += "test"
def op = wsoperation.create("SayHello")
op.paramTypes += [name: String]
op.resultType = String
op.behavior = { return "hello $name" } // <– closure
s1.operations += op

// usage of model builder
def s2 = webservice.build {
  TestService(binding: "doc,lit", tags: ["test"]) {
    SayHello(name: String, result: String) {
      return "hello $name"
    }
  }
}[0]
```

By default, model types allow to encapsulate executable code (e.g., operation behavior in previous sample) in order to program the testbed and to implement fine-grained customizations. For this purpose, G2 is using Groovy *closures* which are executed on top of the shared runtime environment (introduced in previous section). The environment provides access to all registered entities inside the testbed (plugins, deployed instances, shared data, etc.) and offers introspection via model reflection. For example, the next snippet defines a simple operation behavior that determines the number of available Web service models and invokes the `Logger` plugin. During execution, Groovy would resolve the aliases `webservice` and `log`, and provide access to the referenced instances.

```
op.behavior = {
  def num = webservice.getAll().size()
  log.write("Currently $num service models exist.")
}
```

As the structures of testbed models can get quite complex, we use *GPath* expressions to offer simplified access via compact queries. In the following example, the query checks whether there exist local services which are tagged with `''test''` and which contain an operation named `''SayHello''`. Such queries are applied to select and manipulate specific parts of the model with one single command.

```
localhost.webservice.grep{s–> "test" in s.tags}.
    operation.any{o–> o.name == "SayHello"}
```

In all, G2 benefits from its Groovy binding in a twofold manner. The dynamic features provide the functional grounding for generating extensible testbeds, while the language's brevity helps to model them by using a clear and compact syntax.

D. Multicast Testbed Control

A drawback of G1 was that testbed manipulations had to be done in a point-to-point manner, updating one Web service at a time. This was an issue for controlling large-scale testbeds, such as the one used in the VReSCo project [32] consisting of up to 10000 services. To overcome this issue, G2 supports multicast-based manipulations. This feature is inspired by multicast network communication, where a single transmitted packet can reach an arbitrary large number of destination hosts with the help of replicating routers. To provide similar efficiency, G2 uses filter closures which specify the destination of a change request and reduces the number of request messages. In detail, G2 applies the filter at the local testbed model to get the resulting set of designated elements and checks at which back-end hosts these are deployed. Then it wraps the change request, including the filter, and sends it to the involved hosts. Eventually, the hosts unwrap it, run the filter locally, and perform the changes on each matched testbed element. This way, G2 reduces the number of request messages to the number of involved back-end hosts, which significantly improves efficiency. The following snippet shows a sample multicast manipulation. It addresses Web services matching a namespace and performs a set of modifications on them, e.g., appending a new operation and setting model properties.

```
def newOp=operation.create("newOperation")

webservice (op:newOp) { s->   // filter closure
  s.namespace =~ /infosys.tuwien.ac.at/
} { s->   // command closure
  s.operations+=op
  s.someProperty = "someValue"
}
```

IV. QoS Testbed Scenario

In this paper we do not evaluate the performance of G2. Instead, we chose to demonstrate G2 in practice in order to give a better understanding of the previously presented concepts and also to give an impression about the intuitiveness of G2's script-based control.

Our scenario covers the creation of a rather simple testbed for testing the QoS monitor [16] used in the VReSCo project [32]. The monitor performs periodical checks for determining a Web service's execution time, latency, throughput, availabiltiy, robustness, and other QoS properties. Most of the monitoring is done in a non-intrusive manner, while for some checks local sensors need to be deployed at the service. For verifying the monitor's correct functionality, runtime tests must be performed on a testbed of generated Web services simulating QoS properties. Furthermore, the QoS properties must be controllable during test execution and the Web services must support the application of local sensors. Even though, the creation of such a testbed is perfectly feasible with G2, we had to restrict its functionality due to space constraints. We omitted testbed features, such as registration of generated services at a broker, and replaced the usage of the `QoSEmulator`. Instead, we just simulate processing time and failure rate via simple delaying and throwing exceptions at the Web operations. However, for demonstration purposes, we have included some additional features, such as nested invocations, dynamic replacement of functionality, and generation of active clients. For setting up the testbed, we are using the plugins `WebServiceGenerator`, `WebServiceInvoker`, `CallInterceptor`, `ClientGenerator`, `SimpleRegistry`, and `DataPropagator`, which establish the model schema depicted in Figure 1. We divided the scenario into three parts: in the first step we generate the service-based testbed, then we generate clients invoking the testbed's services, and, finally, show how the running testbed can be altered at runtime.

```
// reference 10 back-end hosts                                    1
1.upto(10) { n-> host.create("192.168.1.$n",8080) }               2

// load message type definitions from XSD file                    4
def inType=datatype.create("types.xsd","inputType")               5
def outType=datatype.create("types.xsd","outputType")             6

prop.randomListItem={ list-> //get random item                    8
  list[new Random().nextInt(list.size())]                         9
}                                                                 10

def serviceList=webservice.build {                                12
  1.upto(100) { i-> //create Service1 .. Service100               13
  "Service$i"(delay: 0, failureRate: 0.0) {                       14
    tags=["worker"]                                               15
    //Web service operation "Process"                             16
    Process(input: inType, response: outType) {                   17
      Thread.sleep(delay)                                         18
      if (new Random().nextDouble()<failureRate) {                19
        throw new Exception("sorry!")                             20
      }                                                           21
      return outType.createDummy()                                22
    }                                                             23
  }}                                                              24

  1.upto(20) { i-> //create 20 delegator services                 26
  "CompositeService$i"() {                                        27
    tags=["delegator","composite"]                                28
    processError={} //initially empty function                    29
    //Web service operation "Delegate"                            30
    Delegate(input: inType, neededResults: hdr(int),              31
             response: arrayOf(outType)) {                        32
      def gotResults=0                                            33
      def result=[]                                               34
      while (gotResults<neededResults) {                          35
        def refs=registry.get{"worker" in it.tags}                36
        def ref=randomListItem(refs)                              37
        try {                                                     38
          result+=ref.Process(input).response                     39
          gotResults++                                            40
        } catch (e) { processError(e) }                           41
      }                                                           42
      return result                                               43
    }                                                             44
  }}                                                              45
}                                                                 46

serviceList.each { s-> //deploy at random hosts                   48
  s.deployAt(randomListItem(host.getAll()))                       49
}                                                                 50
```

Listing 1. 'Generation of Web services for task delegation example'

Listing 1 covers the specification of the services. First, a set of back-end hosts is referenced and the service's mes-

```
def initClient=client.create()
initClient.run=true //boolean flag 'run'
initClient.code={ //client code as closure
  while (run) {
    Thread.sleep(5000) //every 5 seconds
    def refs=registry.get{"delegator" in it.tags}
    def r=randomListItem(refs) //pick random
    def arg=inType.newInstance()
    r.Delegate(arg, 3) //initiate delegation
  }
}

initClient.deployAt(host.getAll()) //run clients
```

Listing 2. 'Generation of clients invoking delegator Web services'

```
def pi=callinterceptor.create()
pi.hooks=[in:"RECEIVE", out:"SEND"] //where to bind
pi.code={ msg-> qosmon.analyze(msg) } //sensor plugin

webservice(i:pi) { s-> "delegator" in s.tags} { s->
  s.interceptors+=i //attach to author services
  s.processError= { e->
    def url="http://somehost.com/reportError?WSDL"
    def reportWs=wsreference.create(url)
    reportWs.Report(my.webservice.name, e.message)
  }
}

int cycles=1000
while (--cycles>0) {
  Thread.sleep(2000) //every 2 seconds
  def workers=webservice.get{"worker" in it.tags}
  def w=randomListItem(workers)
  w.delay=new Random().nextInt(20*1000) //0 - 20sec
  w.failureRate=new Random().nextFloat() //0.0 - 1.0
}

initClient.run=false //shut down all clients
```

Listing 3. 'On-the-fly manipulation/extension of running testbed'

sage types are imported from an XSD file. In Line 8, the `DataPropagator` plugin is invoked, via its alias `prop`, to bind a global function/closure to the shared runtime environment. The testbed itself comprises 100 simple worker services and, in addition, 20 delegators that dispatch invocations to the workers. In Lines 13 to 24, the worker services are built, for each we declare variables for controlling the simulation of QoS, and add a tag for distinction. For the worker's Web service operation `Process` we specify its I/O message types and customize its behavior with simple code for simulating delay and failure rate, controlled via the service's variables. The composite delegator services are created in a similar manner, but contain nested service invocations and a user-defined customization (`processError()`). Furthermore, a header argument is specified (`neededResults`), which means that it is declared as part of the SOAP header instead of the body. In Line 36 the `SimpleRegistry` is queried to get a list of references to worker services. Of these random ones are picked and invoked (Line 39) in sequence, until the required number of correct responses has been reached. On faults, the customizable error handling routine named `processError()` is called. Eventually, the delegator service returns a list of responses. At the end of the script, the testbed is generated by deploying the modeled Web services on random hosts.

Though, in this state the testbed contains only passive services awaiting invocations. In order to make it "alive", by generating activity, Listing 2 specifies and deploys clients which invoke random delegator services in 5 second intervals.

Finally, Listing 3 demonstrates how running testbeds can be altered at runtime. At first, a call interceptor is created, which can be, for instance, used to place the QoS sensors. We make use of G2's multicast updates and enhance all delegator services by appending the interceptor to the service model. In the same request we replace the (formerly empty) `processError()` routine and instruct the services to report errors to a 3rd-party Web service. At the back-end, the `WebServiceGenerator` plugins will detect the change request and automatically adapt the addressed services. Furthermore, by making use of G2's immediate synchronization of models with running testbed instances, the simulation of QoS is altered on the fly by changing the corresponding parameter variables of worker services in a random manner. In the end, the clients are shut down by changing their `run` flag.

In this scenario we have tried to cover as many key features of G2 as possible, to demonstrate the simplicity of our scripting interface. We have used builders to create nested model structures (service→operation→datatype), designed Web services and clients with parameterizable behavior, customized behavior with closures, applied plugins (e.g., call interceptors and service invokers), performed a multicast manipulation request, and steered the running testbed via parameters. The generated testbed consists of interconnected Web services and active clients calling them. To facilitate proper testing of the QoS monitor [16], it would require to simulate not only processing time and fault rate, but also scalability, throughput, and other properties which we have skipped for the sake of brevity. In any case, we believe that the presented scenario helps to understand how G2 is used and gives a good impression about its capabilities.

V. Discussion and Future Work

Certain concepts of G2 might be considered with skepticism by readers and, therefore, require to be discussed in this paper. First of all, the usage of closures, which encapsulate user-defined code, for customizations of behavior is definitely risky. As we do not check the closures for malicious code, it is, for instance, possible to assign {`System.exit(0)`} to some testbed instance at the back-end, to invoke it, and hereby to shut down the remote G2 instance. This security hole restricts G2 to be used only by trusted engineers. For the current prototype we accepted this restriction on purpose and kept closure-based customizations for the vast flexibility their offer.

Some readers may also consider the G2 framework as too generic, since it does not generate the testbed instances but delegates this to the plugins, and may wonder whether it deserves to be called a "testbed generator framework" at all. In our opinion this is mainly a question of where to define

the boundary between a framework and its extensions. We implemented a number of plugins which generate basic SOA artifacts, such as services, clients, and registries. If we decide to direct our future research towards non-SOA testbeds, we will be able to base this work on the G2 framework.

Moreover, in the introduction of this paper we said that SOA comprises more than just Web services, but also clients, service buses, mediators, workflow engines, etc. But looking at the list of plugins which we developed (see Section III-B), it becomes evident that we do not cover all these components. This is partially true, as this paper presents the current state of our work in progress. However, we are continuously extending our plugin repertoire and will make up for the missing ones soon, e.g., by porting G1's BPEL workflow plugin to G2.

Also, G2 is currently missing sophisticated support for WS-* standards which are an essential asset for SOAP-based communication. In the strict sence, it is possible to use call interceptors for WS-* processing but the engineer must handle the complex processing. We regard it as necessary, to unburden him/her by providing plugins for the common standards (e.g., WS-Addressing for asynchronous communication, WS-Policy, WS-Security) and to support the creation of additional ones.

Last but not least, the question might be raised why we prefer a script-based approach. The reason is that we derive a lot of flexibility from the Groovy language and see high potential in the ability to program the testbed's behavior compared to, for instance, composing everything in GUIs, which provides user convenience at the cost of flexibility.

VI. Conclusion

In this paper we have introduced Genesis2, a framework supporting engineers in generating testbed infrastructures for SOA. We have given an overview of the framework's concepts and outlined its novel features which offer a high level of extensibility and customizability. Furthermore, we have used a scenario example to demonstrate how engineers can specify and program testbeds via an intuitive scripting language. We regard Genesis2 as an important contribution for the SOA testing community, as it is the first generic testbed generator that is not restricted to a specific domain but can be customized to set up testbeds of diverse components, structure, and behavior. We plan to release the software via our Web site [27] and expect that it will have significant impact on future research on automated testbed generation.

Acknowledgment

The research leading to these results has received funding from the European Community Seventh 7th Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

The authors would also like to thank their collegues Harald Psaier, Daniel Schall, Florian Skopik, and Martin Treiber for their valuable feedback and discussions.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *Int. J. Cooperative Inf. Syst.*, vol. 17, no. 2, pp. 223–255, 2008.

[2] G. Denaro, M. Pezzè, D. Tosi, and D. Schilling, "Towards self-adaptive service-oriented architectures," in *TAV-WEB*. ACM, 2006, pp. 10–16.

[3] R. B. Halima, K. Drira, and M. Jmaiel, "A qos-oriented reconfigurable middleware for self-healing web services," in *ICWS*. IEEE Computer Society, 2008, pp. 104–111.

[4] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *ICAC*. IEEE Computer Society, 2004, pp. 2–9.

[5] G. Canfora and M. D. Penta, "Testing services and service-centric systems: challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10–17, 2006.

[6] W.-T. Tsai, R. A. Paul, W. Song, and Z. Cao, "Coyote: An xml-based framework for web services testing," in *HASE*. IEEE Computer Society, 2002, pp. 173–176.

[7] M. D. Barros, J. Shiau, C. Shang, K. Gidewall, H. Shi, and J. Forsmann, "Web services wind tunnel: On performance testing large-scale stateful web services," in *DSN*. IEEE Computer Society, 2007, pp. 612–617.

[8] J. Zhang, "A mobile agents-based approach to test the reliability of web services," *IJWGS*, vol. 2, no. 1, pp. 92–117, 2006.

[9] 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.

[10] 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.

[11] 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.

[12] W. Vogels, "Web services are not distributed objects," *IEEE Internet Computing*, vol. 7, no. 6, pp. 59–66, 2003.

[13] 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.

[14] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[15] K. Verma and A. P. Sheth, "Autonomic web processes," in *ICSOC*, ser. Lecture Notes in Computer Science, vol. 3826. Springer, 2005, pp. 1–11.

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

[17] E. Martin, S. Basu, and T. Xie, "Websob: A tool for robustness testing of web services," in *ICSE Companion*. IEEE Computer Society, 2007, pp. 65–66.

[18] J. Zhang and L.-J. Zhang, "Criteria analysis and validation of the reliability of web services-oriented systems," in *ICWS*. IEEE Computer Society, 2005, pp. 621–628.

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

[20] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware," Faculty of Informatics - University of Lugano, Tech. Rep. 2009/07, November 2009. [Online]. Available: http://www.inf.usi.ch/research_publication.htm?id=55

[21] "OASIS - Business Process Execution Language for Web Services," http://www.oasis-open.org/committees/wsbpel/.

[22] "Web Services Description Language," http://www.w3.org/TR/wsdl.

[23] "WS-Agreement," http://www.ogf.org/documents/GFD.107.pdf.

[24] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Automating experimentation on distributed testbeds," in *ASE*. ACM, 2005, pp. 164–173.

[25] W.-T. Tsai, Z. Cao, X. Wei, R. A. Paul, Q. Huang, and X. Sun, "Modeling and simulation in service-oriented software development," *Simulation*, vol. 83, no. 1, pp. 7–32, 2007.

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

[27] "Genesis Web site," http://www.infosys.tuwien.ac.at/prototype/Genesis/.

[28] "SOAP," http://www.w3.org/TR/soap/.

[29] "Jakarta Bean Scripting Framework," http://jakarta.apache.org/bsf/.

[30] "Apache CXF," http://cxf.apache.org/.

[31] "Groovy Programming Language," http://groovy.codehaus.org/.

[32] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-end support for qos-aware service selection, binding and mediation in vresco," *IEEE T. Services Computing*, 2010 (forthcoming).