# Programmable Fault Injection Testbeds for Complex SOA

Lukasz Juszczyk and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria
{juszczyk,dustdar}@infosys.tuwien.ac.at

**Abstract.** The modularity of Service-oriented Architectures (SOA) allows to establish complex distributed systems comprising e.g., services, clients, brokers, and workflow engines. A growing complexity, however, automatically increases the number of potential fault sources which have effects on the whole SOA. Fault handling mechanisms must be applied in order to achieve a certain level of robustness. In this paper we do not deal with fault-tolerance itself but regard the problem from a different perspective: how can fault-tolerance be evaluated? We argue that this can be best done by testing the system at runtime and observing its reaction on occuring faults. Though, engineers are facing the problem of how to perform such tests in a realistic manner in order to get meaningful results. As our contribution to this issue we present an approach for generating fault injection testbeds for SOA. Our framework allows to model testbeds and program their behavior, to generate running instances out of it, and to inject diverse types of faults. The strength of our approach lies in the customizability of the testbeds and the ability to program the fault-injecting mechanisms in a convenient manner.

## 1 Introduction

The principles of SOA propagate building distributed systems based on modular and loosely-coupled components. The spectrum of these components ranges from stand-alone and composite Web services, clients, brokers and registries, workflow engines, monitors, governance systems, message dispatchers, service buses, etc. Considering the dependencies within a complex SOA, it becomes evident that each component is a potential fault source and has an impact on the whole system. Moreover, faults can happen at different levels, e.g., at the network layer, at the interaction level, or as errors in the exchanged messages. As a consequence, sophisticated fault handling mechanisms are required in order to mitigate the effects of faults, to prevent failures, and to guarantee a certain level of robustness. This problem has already been addressed in several works [1–4] and is out of the scope of this paper. Instead, we are facing it from a different perspective: how can engineers evaluate fault handling mechanisms of a SOA? How can they verify that their systems will behave as expected once deployed in their destination environment? These issues cannot be solved by simply performing simulations but require thorough tests at runtime. But how

can engineers perform such tests *prior* to final deployment, without having access to a real(istic) environment which would serve as a testbed? In this paper we address this question and present our contribution.

In our previous work [5] we have introduced the *Genesis2* framework which supports engineers in setting up testbeds for SOA. *Genesis2* allows to model testbeds consisting of various types of components, to program their behavior, and to generate real instances of these on a distributed back-end. In the current paper we extend this approach in order to generate multi-level fault injection testbeds. We empower engineers to generate emulated SOA environments and to program fault injection behavior on diverse levels: *at the network layer, at the service execution level, and at the message layer.*

Our paper is structured as follows: in the next Section we present the motivation for our research. In Section 2 we present *Genesis2* and explain in Section 3 how we generate fault-injection testbeds. Section 4 covers the implementation the practical application of our approach. In Section 5 we review related work and outline our contribution. Finally, Section 6 concludes this paper.

### 1.1 Motivation

Today's SOAs comprise large numbers and varieties of components. This is not only limited to services, clients, and brokers (as conveyed in the famous Web service triangle [6]), but includes also more sophisticated components, such as governance systems and monitoring registries [7], which are performing complex tasks on the service-based environment. In general, we can divide SOA components into three groups: a) stand-alone components which are independent, b) complex services/components which have dependencies and, therefore, are affected by others, and c) clients which are simply consuming the offered services. Each of the components is prone to errors, but the complex ones are affected in a twofold manner as they have also to deal with remote faults of the components they depend on. As outlined correctly in [8] and [9], faults do happen on multiple levels, to be precise, on each layer of the communication stack. This includes low-level faults on the network layer (e.g., packet loss/delay), faults on the transport layer (e.g., middleware failures), on the interaction layer (quality of service), as well as directly at the exchanged messages which can get corrupted. Depending on the structure and configuration of the SOA, each of these faults can cause a chain of effects (also referred to as error propagation), ranging from simple execution delays to total denial of service. These challenges can only be met if engineers perform intense tests during the development phase, execute scenarios in erroneous SOA environments, and check their system's behavior on faults. However, the main problem remains how to set up such scenarios, in particular, the question how engineers can be provided with proper testbeds which emulate SOA infrastructures in a realistic way. We argue that engineers must be given a possibility to configure testbeds according to their requirements. Depending on the developed system, this includes the ability to customize the topology and composition of the testbed, to specify the behavior of all involved components, and to program individual fault injection models for each of these. In Section 5

we will show that research on fault injection for SOA has been already done by several groups, yet that these works mostly aim testing only individual Web services, for instance, by perturbing their communication channels. The problem of testing complex components which are operating on a whole SOA environment still remained unsolved. This has been our motivation for doing research on a solution which allows to generate large-scale fault-injection testbeds, provides high customizability, and offers an intuitive usage for engineers.

## 2 Genesis2 Testbed Generator Framework

The *Genesis2* framework (*Generating Service-oriented testbed Infrastructures*, in short G2) [5] assists engineers in creating testbed infrastructures for SOA. It 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 provides a virtual view on the testbed, allowing engineers to manipulate it via scripts, and propagates changes to the back-end in order to adapt the running testbed. To ensure extensibility, the G2 framework follows a modular approach and provides the functional grounding for composable plugins that implement testbed generator features. 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 instances at the back-end.

For a better understanding of the internal procedures inside G2, let us take a closer look at its architecture. Figure 1 depicts the layered components, comprising the framework, installed plugins, and, on top of it, the generated testbed:

- At the very bottom, the basic runtime consists of Java, Groovy, and 3$^{\text{rd}}$-party libraries, such as Apache CXF [10].
- 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.

In G2, the engineer creates models according to the provided schema at the front-end, specifying *what* shall be generated *where*, with *which customizations*,

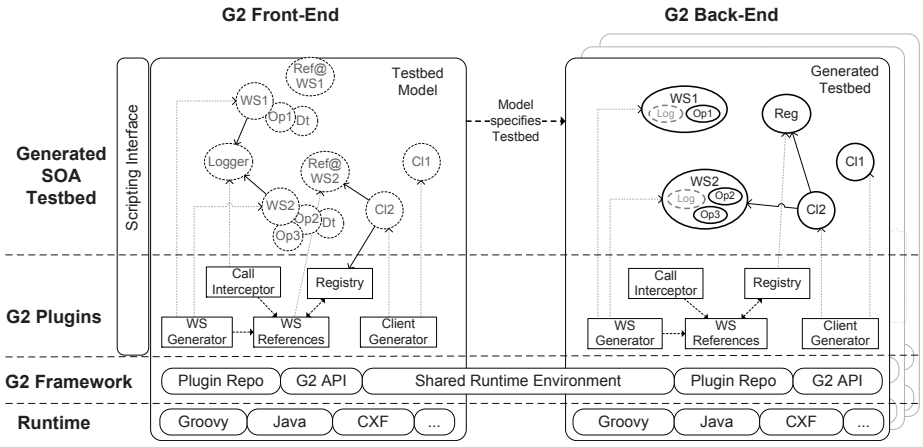


**Fig. 1.** Genesis2 architecture: infrastructure, plugins, and generated elements

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. Figure 2 illustrates the model schema used in this paper. By default, G2 provides model types for specifying Web services (which includes also Web service operations and used data types), clients, registries, and other basic SOA components. In this paper, we are extending this schema with models for specifying faulty behavior (marked gray), which are explained in more detail in the next section.

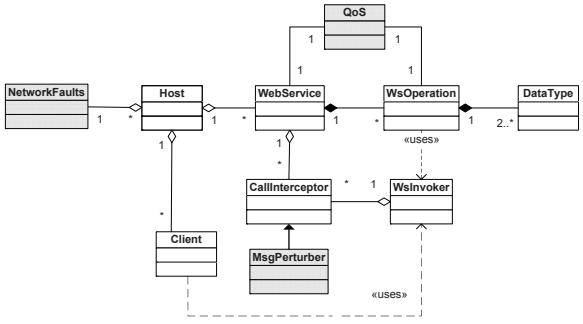


**Fig. 2.** Testbed model schema for fault injection

Listing 1.1 contains a sample specification for demonstrating how testbeds are modeled based on the applied model schema. Basically, G2 is controlled via

Groovy scripts [11] that are executed on top of the *shared runtime environment* (SRE, see Figure 1). Each applied plugin extends the SRE by registering itself, its provided model types, additional macros and other artifacts via aliases. The engineer references these in his/her scripts in order to make use of the provided features and to integrate them into his/her testbeds. Moreover, he/she is free to customize the testbed's behavior via Groovy code blocks.

The sample script starts with referencing remote back-end hosts and importing a data type definition from an XSD file. This is done via instantiating the corresponding types from the model schema via their aliases (`host` and `datatype`). In Lines 5 to 10 a simple Web service is created, which comprises only a single operation (named `SayHi`) and which uses the imported data type in its request message. By using the Builder feature [12] of Groovy, which simplifies the creation of nested datatypes, we are automatically binding the `webservice` to its `wsperation` and the used `datatype`'s. In Line 12 the service is being deployed at two back-end hosts. In this step G2 serializes the service's model and propagates it to the remote back-ends where real Web service instances, which implement the modeled behavior, are being generated and deployed. Next, a simple client is started (Lines 14 to 21), which uses the registry plugin to discover a desired Web service and, eventually, invokes it. The main purpose of clients is to bootstrap activity into the testbed which would otherwise be purely passive and wait for invocations. Finally, in Lines 24 to 27 a `callinterceptor` is created and attached on-the-fly to the running Web service instances. Call interceptors step in to Web service invocations and allow, for instance, to extract the SOAP message (as done in the sample for logging) and to manipulate it. We have used this feature for implementing message perturbation in our fault injection testbeds.

```
1  def beHost1 = host.create("192.168.1.11:8080") //import BE host refs
2  def beHost2 = host.create("192.168.1.12:8080")
3  def vCard = datatype.create("/path/types.xsd","vCard") //import XSD type

5  def service = webservice.build { //model Web service via Groovy builder
6    //create model of TestService with one operation
7     TestService(binding: "doc,lit", tags: ["test"]) {
8        SayHi(card: vCard, result: String) { return "Hi ${card.name}" }
9     }
10 }[0]

12 service.deployAt(beHost1,beHost2) //deployment at two back-end hosts

14 def cl = client.create() //model simple client which calls TestService
15 cl.code = {
16     def refList=registry.get{ s-> s.name=="TestService"} //query
17     def ref=refList[0] //take first from result list
18     println ref.SayHi(vCard.newInstance()) //WS invocation
19 }

21 cl.deployAt(beHost2) //deployment at back-end

23 // attach call interceptor
24 def pi=callinterceptor.create()
25 pi.code={ ctx-> logger.logToDB(ctx.soapMsg) } //calling logger plugin

27 service.interceptors+=pi // on-the-fly attachement of interceptor
```

**Listing 1.1.** 'Sample script specifying a service, a client, and a call interceptor'

All in all, G2 supports engineers in customizing testbeds, programming their behavior, and implementing extensions via plugins. Though, due to space constraints it is impossible to provide a closer introduction to G2 and, therefore, we direct interested readers to [5] which explains more details of our base framework.

## 3 Programmable Multi-level Fault Injection Testbeds

Taking into consideration the complexity of a typical SOA, which comprises diverse components being deployed on heterogeneous platforms and interacting with each others, it becomes evident that each host, each component, each communication channel, and each exchanged message is a potential source of faults, erroneous behavior, and service failures [13]. Basically, faults can occur at every level/layer of the communication stack and, therefore, if testbeds are supposed to emulate realistic scenarios they must be also able to emulate a wide range of fault types. Based on the G2 framework we have developed an approach for generating SOA testbeds and injecting programmable faults. Due to G2's generic nature and its extensibility it is possible to emulate a wide variety of faults by writing plugins which augment the testbed's components and impair their execution. However, in the current state of our work we have concentrated on the following:

1. **Faults at the message layer**, in terms of message data corruption.
2. **Faults at the service execution**, affecting Quality of Service (QoS).
3. **Faults at the network layer**, hampering the packet flow between hosts.

Each type of fault is affecting a different part of the overall SOA and, therefore, we have split their emulation into three independent plugins. Each plugin extends the model schema and offers possibilities to customize and program the fault injection behavior. Figure 2 depicts the provided model types and their position within the schema. Since network faults affect the whole communication between hosts, their model does directly extend the `Host` type. Service execution faults can be caused by the whole service (e.g., low availability) or only by individual operations (e.g., erroneous implementation), therefore their model is bound to both. Finally, for message faults we have extended the `CallInterceptor` which provides access to the request and response messages for perturbation purposes. In the following sections we are explaining the individual fault injection mechanisms in more detail.

### 3.1 Message Faults

SOAP Web services are using WSDL documents [14] to describe their interfaces. Consequently, the service can define the expected syntax of the request messages and the client is aware of the response message's syntax. However, malicious components can produce corrupted messages which either contain meaningless content (message errors on a semantical level), which violate the message's XML

schema definition [15] (high-level syntax errors), or which even do not represent a correct XML document at all (low-level syntax errors). Depending on the degree of corruption, fault handling mechanisms can be applied to allow the integration of faulty components into a SOA. To test such mechanisms we have developed a plugin which allows to intercept exchanged SOAP messages and to perturb them on each of the mentioned levels. Engineers can program the perturbation via the `MsgPerturber` model and the plugin attaches the faulty behavior to Web services and clients, by using Apache CXF's interceptors [16]. We have built the perturbation mechanism upon the visitor pattern [17]. Perturbation code, wrapped in visitor objects, is propagated recursively along the XML tree and/or the unmarshalled objects and has full read/write access for performing manipulations.

For pure semantic perturbation the engineer can overwrite the message's values, but cannot violate the XML structure. The plugin unmarshalls the SOAP body arguments, as well as the headers, into Java objects and applies the visitor code on them. The first sample block in Listing 1.2 shows an interceptor that is programmed to assign random values to all integer fields named `sum`. Moreover, it deletes all postcodes for matching addresses.

For high-level syntax manipulation, the engineer can alternate both, the content and the structure of the XML document. In this case the visitor is applied on the DOM tree of the message. In the second sample block, the visitor is looking for nodes which have children named `country` and appends a new child which violates the message's XSD definition. However, the result is still a well-formated XML document. For low-level corruption, the message must be altered directly at the byte level, as demonstrated in the last snippet which corrupts XML closing tags. Finally, in Line 23, the interceptors get deployed at a Web service and start injecting faults into its request and response messages.

```
1  def valuePert = msgperturber.create("args") //pert. data values
2  valuePert.code = { it ->
3      if (it.name=="sum" && it.type==int) { //get by name and type
4          it.value*=new Random().nextInt()
5      } else if (it.name=="Address" && it.value.country=="AT") { //by value
6          it.value.postcode=null
7      }
8  }

10 def xmlPert = msgperturber.create("dom") //pert. XML structure
11 xmlPert.code = { node ->
12     if (node.children.any { c-> c.name=="country" }) {
13         Node newChild = node.appendNode("NotInXSD")
14         newChild.attributes.someAtt="123"
15     }
16 }

18 def bytePert = msgperturber.create("bytes") //pert. msg bytes
19 bytePert.code = { str ->
20     str.replaceFirst("</","<") //remove closing tag from XML doc
21 }

23 service.interceptors+=[bytePert, xmlPert, valuePert] //attach to service
```

**Listing 1.2.** 'Programming message perturbation'

### 3.2 Service Execution Faults

Service execution faults usually result in degraded Quality of Service (QoS) [18]. Examples are slower processing times which delay the SOA's execution, scalability problems regarding the number of incoming requests, availability failures which render parts of the SOA inaccessible, etc. Especially in the context of Web services, QoS covers a wide spectrum of properties, including also security, discoverability, and also costs. However, in our work we only deal with those concerning service execution, as defined in [19], comprising response time, scalability, throughput, and accuracy of Web service operations and the availability of the whole service. For emulating these, we developed the `QoSEmulator` plugin, which has access to the generated Web service instances in on the back-end and intercepts their invocations in order to simulate QoS. To model a service's QoS, engineers can either assign fixed values to the individual properties (e.g., processing time = 10 seconds) or define more sophisticated fault models via Groovy code closures [20], resulting in programmable QoS. The main advantage of closures consists in the ability to incorporate diverse factors into the fault models. For example, engineers can set the availability rate depending on the number of incoming requests or to define the processing time according to a statistical distribution function, supported via the *Java Distribution Functions* library [21].

Listing 1.3 contains a sample specification of two QoS models, one for defining the availability of a Web service and one for controlling the execution of its operations. The availability is defined according to the daytime in order to simulate a less overloaded service during the night (Lines 1 to 7). For the service operation, the response time is derived from a beta distribution (alias `dist`) while throughput and error rate (accuracy) are assigned with constant values. At the end, the models are bound to the service and its operations.

```
1  def svcQos = qos.create()
2  svcQos.availability = {
3      if (new Date().getHours()<8) { //from 0 to 7 AM
4          return 99/100 //set high availability of 99%
5      }
6      return 90/100 //otherwise, set lower availability rate
7  }

9  def opQos = qos.create()
10 opQos.responseTime = { dist.beta.random(5000,1,null) } //beta distrib.
11 opQos.throughput = 10/60 //restrict to 10 invocations per minute
12 opQos.errorRate = 15/100 //15% of invocations will fail with exceptions

14 service.qos=svcQos //attach QoS model to service definition

16 service.operations.grep { o-> o.returnType!=null }.each {
17     o.qos=opQoS //and to all 2-way operations
18 }
```

**Listing 1.3.** 'Programming QoS emulation'

### 3.3 Low-level Network Faults

Network faults, such as loss and corruption of IP packets, play a minor role in SOA fault handling, mainly because they are already handled well by the

TCP/IP protocol which underlays most of the service-oriented communication. But they can cause delays and timeouts, and this way slow down the whole data flow. Apart from that, there exist Web service protocols which are built upon UDP, such as *SOAP over UDP* [22] and *Web Service Dynamic Discovery* [23], which are, therefore, more vulnerable to network faults. Creating testbeds which emulate low-level faults requires a much deeper intrusion into the operating system, compared to the other plugins. It is necessary to intercept the packet flow, to perform dropping, duplication, reordering, slowing down, etc. This can hardly be done on top of the *Java Virtual Machine* which hosts the G2 framework. To by-pass this issue, we have developed our `NetworkFaultEmulator` plugin based on the Linux tool *Traffic Control* (tc) [24] (with *netem* module [25]) which allows to steer packet manipulation at the kernel level. Unfortunately, this deprives G2 of its platform independence but, on the other hand, allows to reuse tc's rich set of features. We have presented a first version of this approach in [26]. Similar to the previously presented plugins, engineers create fault models but, in this case, attach them directly to the back-end hosts. There the fault models are locally translated into tc commands for manipulating the host's packet flow.

Listings 1.4 and 1.5 comprise a sample for illustrating the mapping from the model to the resulting tc commands. The model is created by assigning self-explanatory parameters and is finally being attached to the hosts. At the back-end, the plugin first sets up a virtual network interface which hosts all generated instances, such as Web services, registries, etc. This step is necessary for limiting the effect of the fault emulation only on the testbed instances, instead of slowing down the whole physical system. Eventually, the modelled faults are translated into tc commands applied on the virtual IP.

```
1 def nf = networkfaults.create()
2 nf.loss = 2/100 //2% packet loss
3 nf.duplicate = 1/100 //1% packet duplication
4 nf.delay.value = 100 //100ms
5 nf.delay.variation = 20 //20ms of variation
6 nf.delay.distribution = "normal" //normal distribution

8 nf.deployAt(beHost1,beHost2) //attach to BE hosts
```

**Listing 1.4.** 'Programming network faults'

```
1 ifconfig lo:0 add 192.168.100.1 #set up virtual IP addr. for BE instances

3 tc qdisc change dev lo:0 root netem loss 2.0%
4 tc qdisc change dev lo:0 root netem duplicate 1.0%
5 tc qdisc change dev lo:0 root netem delay 100ms 20ms distribution normal
```

**Listing 1.5.** 'Network fault model translated to Traffic Control commands'

## 4 Implementation and Practical Application

### 4.1 Implementation and Extensibility of Genesis2 Prototype

The G2 framework has been developed in Java SE 6 [27] and Groovy [11]. The critical parts, which handle the framework's internal logic and the communication between front-end and back-end, are written in Java. Groovy, however, has

been exploited for having a flexible scripting language for modelling the testbeds and for programming customizations via Groovy closures [20].

G2 provides a generic framework which outsources the generation of testbed instances to the corresponding plugins, supporting the procedure via its API. Based on that grounding, plugins define their extensions to the model schema and interpret their provided extensions for generating deployable testbed components. For instance, the `WebServiceGenerator` plugin analyzes `webservice` models and translates them into deployable Apache CXF-based [10] Web service instances. All plugins provide access to their generated components but offer also interfaces for customization, so that other plugins can intervene and/or extend their functionality. The fault injection plugins presented in this paper make intense use of this feature. Though, of course they do not cover all possible error sources in a SOA but only represent the current state of our work. For testing complex SOAs it would be also necessary to emulate, for example, middleware faults, fault in the execution of WS-* protocols, or misbehavior of human provided services [28] integrated into workflows. Depending on the next steps in our research, and which fault injection mechanisms will be required for evaluating our prototypes, we will develop further plugins in the future.

### 4.2 Practical Application in Testing of SOA Prototypes

The question how G2 should be used to generate fault injection testbeds depends strongly on the type of the tested SOA, its composition, purpose, as well as its internal fault handling mechanisms. In the end, engineers have to generate testbeds which emulate the SOA's final deployment environment as realistically as possible. While Groovy scripts are G2's primary interface for modeling testbeds, we are currently investigating techniques for importing external specifications of SOAs and their components into G2 models (e.g., from BPEL process definitions [29] and WSDL documents [14]). Independent on how the testbed got specified, whether from scratch or via imports, the engineer is always operating on a set of models describing SOA components which have their pendant generated instances located in the back-end. For providing a better conception of how a testbed is actually structured Figure 3 illustrates its layered topology. At the two bottom layers G2 connects the front-end to the distributed back-end and the plugins establish their own communication structures. Most important are the two top layers which comprise the results of the engineer's activities. Based on the provided model schema, he/she 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 engineer's specification, which includes also fault injection behavior. The aggregation of these instances constitutes the actual testbed infrastructure on which the developed SOA can be evaluated.

Of course, the evaluation of a software system also comprises the monitoring of the test cases as well as the analysis of collected data. These data are, for instance, log files, performance statistics, captured messages, and other resources, depending on the tested SOA and the fault-handling mechanisms to be verified.

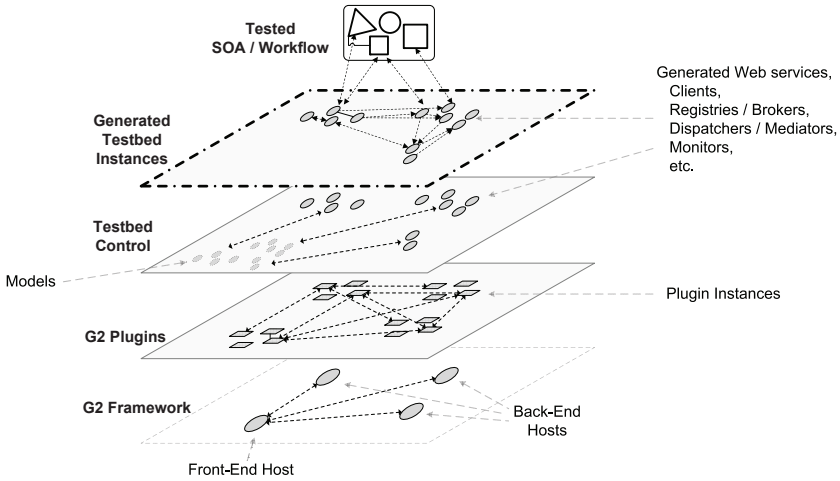


**Fig. 3.** Interactions within layers of a G2 testbed

These data must be also gathered from both, the tested SOA, to analyze internal procedures and reactions on faults, as well as from the testbed itself, to know which faults have been injected at which time. By correlating both, it is possible to narrow down errors in the SOA and to detect causes of misbehavior. G2 provides means for gathering relevant information about the execution inside the testbed, such as an eventing mechanism that allows to track and log all changes within the testbed configuration or call interceptors for logging of interactions. However, regarding the gathering of log data from the tested SOA system, we do not provide any tool support yet. Also, for the analysis of test results and the narrowing down of errors/bugs inside the SOA we have not come up yet with any novel contribution but regarded this problem as out of scope of the current paper which just describes how we generate the testbeds. But we intend to address these problems in future research.

### 4.3 Managing Large-scale Testbeds

In [5] we have shown how G2 facilitates convenient generation of large-scale testbeds as well as manipulation of these in an efficient multicast-like manner. We are exploiting the multicast feature for adapting larger testbed on-the-fly, e.g., for injecting faults. Listing 1.6 demonstrates its usage for updating hosts and Web services. The command expects the type of the instances which shall be altered (in the sample: `webservice` and `host`) and two closure code blocks. The first closure specifies the filter which determines the designated instances, while the second one contains the manipulation commands. In the presented sample, fault models are attached to all Web services matching their namespace and annotation tags. Moreover, all hosts within a defined subnet are being enhanced

with network fault emulation. As a result, multicast updates help to manage large-scale testbeds in a clear and compact manner.

```
1  webservice { ws-> //filter
2      "faulty" in ws.tags && ws.namespace =~ /www.infosys.tuwien.ac.at/
3  } { ws-> //command
4      ws.qos = qosModel
5      ws.interceptors += [xmlPertModel]
6  }

8  host { h-> //filter
9      h.location =~ /192.168.1./
10 } { h-> //command
11     netFaultModel.attachTo(h)
12 }
```

**Listing 1.6.** 'Injecting faults to hosts and Web services'

Regarding the performance of the framework and our fault injection mechanisms, we have omitted putting a detailed study into this paper, mainly due to space constraints and because we believe that it would not emphasize the message of our paper, which is the presentation of the concepts. Due to the fact that G2 is a programmable framework, the actual performance and system load also depend heavily on how it is applied and what kind of a testbed is being generated, with which components and functionality. Therefore, we believe that presenting a performance evaluation would be only of limited use for the readers.

### 4.4 Open-source Prototype

As we did before with G2's predecessor, the first Genesis framework [30], we will also publish the current prototype as open-source via its homepage [31].

## 5 Related Research

Fault injection has been a well-established testing technique since decades. Numerous tools have been developed and applied for evaluating quality of software/systems. Due to the vast number of available tools, as well as due to their diversity, we do not present a survey on them but refer readers to [32] and [33], which provide a good introduction and overview of available solutions. In general, research on fault injection has produced a lot of results, covering sophisticated techniques for generating meaningful faults as well as for the analysis of their effects on the tested software.

In the domain of Web services and SOA, several works deal with fault injection. Xu et al. have presented an approach for perturbing the XML/SOAP-based messages exchanged by Web services in order to evaluate how well the tested systems can handle message corruption [34]. Moreover, Nik Looker has investigated fault injection techniques in his dissertation [35] and has developed the WS-FIT framework [9, 36]. WS-FIT intercepts SOAP messages at the middleware level and supports a rich set of features for manipulating these. This includes discarding of messages, reordering of them in the interaction flows, perturbing the

XML content, and other features. In general, works like those of Xu and Looker assume the presence of already existing Web services and inject faults for testing their runtime behavior and/or fault-handling mechanisms. However, we do not regard these works as direct competitors, since we have not developed any novel techniques for fault injection in the strict sense. Instead we empower engineers to generate SOA testbeds from scratch and to extend these with programmable faulty behavior. And this we regard as our most distinct contribution. Due to the novelty of our work, we have not identified many related works, but have only found SOABench, PUPPET, and ML-FIT to be relevant for testbed generation.

SOABench [37] provides sophisticated support for benchmarking of BPEL engines [29] 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 [38, 39] 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 [14] and WS-Agreement documents [40] 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 focused on evaluating workflows/compositions and do not support fault injection beyond emulating QoS.

Only the Multi-Level Fault-Injection Testbed (ML-FIT) [8] has a similar focus to our work. It also aims at emulating SOA faults at different levels, builds upon existing fault injection mechanisms, and the authors intend to use collected field data for creating realistic fault models. However, ML-FIT is still under development and, therefore, not much has been published about it yet. It is unclear how testbeds will be generated, which types of SOA components will be supported, and how faults will be modelled and injected. Without knowing these details it is difficult for us to compare our approach to ML-FIT.

## 6 Conclusion

In this paper we have presented our approach for generating programmable fault injection testbeds for SOA. Based on the Genesis2 framework, which allows engineers to model testbeds and to generate real instances of these, we have developed techniques for specifying faults and injecting them into running testbeds. Due to the extensibility of Genesis2, our approach supports the emulation of diverse types of faults. In the current state of our work, we have developed mechanisms for emulating network faults, service execution faults, and for corrupting exchanged messages. In a nutshell, our main contribution consists of the ability to generate SOA testbeds via scripts and the programmability of the injected faults. As a result, engineers can customize the fault behavior to their requirements, in order to have realistic testbeds for evaluating SOA systems.

For future plans, we will be working on further fault injection mechanisms to extend the spectrum of supported fault types and we will publish the prototype implementation as open-source.

## Acknowledgments

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

The authors would also like to thank Philipp Leitner for contributing parts of the `QoSEmulator` plugin.

## References

1. Goeschka, K.M., Froihofer, L., Dustdar, S.: What SOA can do for software dependability. In: DSN 2008: Supplementary Volume of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE Computer Society (2008) D4–D9
2. Psaier, H., Skopik, F., Schall, D., Dustdar, S.: Behavior Monitoring in Self-healing Service-oriented Systems. In: COMPSAC, IEEE Computer Society (2010) (forthcoming)
3. Dialani, V., Miles, S., Moreau, L., Roure, D.D., Luck, M.: Transparent fault tolerance for web services based architectures. In: Euro-Par. Volume 2400 of LNCS., Springer (2002) 889–898
4. Modafferi, S., Mussi, E., Pernici, B.: Sh-bpel: a self-healing plug-in for ws-bpel engines. In: MW4SOC. Volume 184 of ACM International Conference Proceeding Series., ACM (2006) 48–53
5. Juszczyk, L., Dustdar, S.: Script-based generation of dynamic testbeds for soa. In: ICWS, IEEE Computer Society (2010) (forthcoming)
6. Gottschalk, K.D., Graham, S., Kreger, H., Snell, J.: Introduction to web services architecture. IBM Systems Journal **41**(2) (2002) 170–177
7. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-end support for qos-aware service selection, binding and mediation in vresco. IEEE T. Services Computing (2010) (forthcoming)
8. Reinecke, P., Wolter, K.: Towards a multi-level fault-injection test-bed for service-oriented architectures: Requirements for parameterisation. In: SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems, Naples, Italy, AMBER (2008)
9. Looker, N., Munro, M., Xu, J.: Simulating errors in web services. International Journal of Simulation Systems **5**(5) (2004) 29–37
10. Apache CXF: `http://cxf.apache.org/`.
11. Groovy Programming Language: `http://groovy.codehaus.org/`.
12. Groovy Builders Guide: `http://groovy.codehaus.org/Builders`.
13. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. **1**(1) (2004) 11–33
14. Web Services Description Language: `http://www.w3.org/TR/wsdl`.

15. XML Schema Definition: http://www.w3.org/TR/xmlschema-0/.
16. Apache CXF Interceptors: http://cxf.apache.org/docs/interceptors.html.
17. Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: COMPSAC, IEEE Computer Society (1998) 9–15
18. Menascé, D.A.: Qos issues in web services. IEEE Internet Computing **6**(6) (2002) 72–75
19. Rosenberg, F., Platzer, C., Dustdar, S.: Bootstrapping performance and dependability attributes of web services. In: ICWS, IEEE Computer Society (2006) 205–212
20. Groovy Closure Guide: http://groovy.codehaus.org/Closures.
21. Java Distribution Functions library: http://statdistlib.sourceforge.net.
22. SOAP Over UDP: http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.html.
23. Web Services Dynamic Discovery: http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf.
24. Linux Advanced Routing & Traffic Control: http://lartc.org.
25. netem - Network Emulator: http://www.linuxfoundation.org/en/Net:Netem.
26. Juszczyk, L., Dustdar, S.: Testbeds for emulating dependability issues of mobile web services. In: EMSOS, IEEE Computer Society (2010) (forthcoming)
27. Java 6 Standard Edition: http://java.sun.com/javase/6/.
28. Schall, D., Truong, H.L., Dustdar, S.: The human-provided services framework. In: CEC/EEE, IEEE Computer Society (2008) 149–156
29. Business Process Execution Language for Web Services: http://www.oasis-open.org/committees/wsbpel/.
30. Juszczyk, L., Truong, H.L., Dustdar, S.: Genesis - a framework for automatic generation and steering of testbeds of complex web services. In: ICECCS, IEEE Computer Society (2008) 131–140
31. Genesis Web site: http://www.infosys.tuwien.ac.at/prototype/Genesis/.
32. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. IEEE Computer **30**(4) (1997) 75–82
33. Wikipedia on Fault Injection (accessed on Jun 13, 2010): http://en.wikipedia.org/wiki/Fault_injection.
34. Xu, W., Offutt, J., Luo, J.: Testing web services by xml perturbation. In: ISSRE, IEEE Computer Society (2005) 257–266
35. Looker, N.: Dependability Assessment of Web Services. PhD dissertation, Durham University (2006)
36. Looker, N., Munro, M., Xu, J.: Ws-fit: A tool for dependability analysis of web services. In: COMPSAC Workshops, IEEE Computer Society (2004) 120–123
37. Bianculli, D., Binder, W., Drago, M.L.: Automated performance assessment for service-oriented middleware: a case study on bpel engines. In: WWW, ACM (2010) 141–150
38. Bertolino, A., Angelis, G.D., Polini, A.: A qos test-bed generator for web services. In: ICWE. Volume 4607 of LNCS., Springer (2007) 17–31
39. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: TestCom/FATES. Volume 5047 of LNCS., Springer (2008) 266–282
40. WS-Agreement: http://www.ogf.org/documents/GFD.107.pdf.