GENESIS - A Framework for Automatic Generation and Steering of Testbeds of Complex Web Services *

Lukasz Juszczyk, Hong-Linh Truong, Schahram Dustdar
VitaLab, Distributed Systems Group
Information Systems Institute
Vienna University of Technology, Austria
{juszczyk,truong,dustdar}@infosys.tuwien.ac.at

Abstract

Nowadays, the importance of Web services is steadily increasing in domains where interoperability is of paramount importance. This trend is especially observable in complex computer systems which consist of a large number of interacting distributed components, often implemented using Web services. Large-scale systems and high complexity usually result in higher error-proneness in the development process. This should be addressed as early as possible during the development of complex service-oriented systems, ideally before they are actually deployed on a distributed infrastructure. In this paper we present GENESIS - a software framework for solving this problem. Our framework allows automatic generation and steering of testbeds of complex Web services, thereby empowering developers to specify functional and non-functional properties of Web services, to generate and deploy Web service instances on remote hosting environments, to enhance the functionality of the framework with plug-ins, and to control the behavior of the testbed during runtime.

Keywords: Web services, Service-oriented Architecture, testbed generation.

1 Introduction

In spite of their various shapes and occurrences [31], complex computer systems have always involved a large number of (distributed) components [21] and interactions. While in the past these components were hosted in predominantly homogeneous environments, such as inside organizations or military systems, today's complex systems have evolved into being used in heterogeneous environments which incorporate, for instance, legacy systems.

*This work is partially supported by the European Union through the FP6-2005-IST-5-034749 project WORKPAD.

As a consequence, interoperability has become an important issue, which lead to the establishment of SOAP [10] as a de-facto communication standard and the encapsulation of transportation functionality in Web services technologies [12]. This trend has also influenced the design and development process of complex systems by making it attractive to engineers to follow the principles of service-oriented architecture (SOA). At first sight, SOA brings various benefits, such as flexibility, modularity, composability, and reusability, just to name the most well-known ones. However, as a side-effect, these features pose challenges to the developers due to their complexity and error-proneness. This problem needs to be addressed early in the development phase. As a consequence, much effort has been put into the development of methods and tools for automated testing and detection of error-prone components in SOAs [18, 26, 30, 34, 36, 40]. However, those solutions mainly aim at analyzing only individual Web services by performing various client-oriented checks. To our best knowledge, there is no support for the deployment of whole testbeds, consisting of real Web services, in order to test complex features of SOAs during runtime.

In this paper we present `GENESIS`, a framework for generating service-based infrastructures, which allows developers to set up SOA testbeds in a convenient manner. `GENESIS` combines an approach for automatic generation and deployment of Web services at the back-end with a programmer-friendly API at the front-end. Developers can specify functional and non-functional properties of Web services which can be deployed on-the-fly on remote hosting environments. Complex behavior of the testbed can be achieved with various plug-ins which extend the functionality of the individual Web services and can be steered remotely from the front-end. Therefore, `GENESIS` allows setting up large-scale testing infrastructures for complex service-oriented systems.

The rest of this paper is structured as follows. In

Section 2 we discuss the motivation for our contribution. Sections 3 and 4 give an insight into the architecture of GENESIS and the techniques used for creating complex services, respectively. In Section 5 we discuss how our solution can be applied in practice and present an example to illustrate the usage of our concepts and prototype implementation. Section 6 addresses some details of our implementation. In Section 7 related work is being reviewed. Finally, in Section 8 we outline our future work and conclude this paper.

2 Motivation and Requirements

Testing software for faults and failures is an essential part of the development process, which should be performed continuously during all stages of the process itself. This includes unit tests [22] for checking the quality of the individual modules, integration tests [25] when these modules are being connected, functional tests [23] verifying whether the functionality meets the specified requirements, and finally, tests of the whole system in later stages of the process. A mapping of these levels to SOA development shows that unit tests mostly involve only individual services [26, 30], while the other methods aim at testing whole service infrastructures and applications operating on them [18, 34, 36]. However, although SOA has been an important topic in research and industry during the last years, we noticed a lack of tools supporting the developers to set up such infrastructures of services for testing purposes. Such a tool should meet the following requirements:

- *Flexibility*: Specification of Web services with customizable functional and non-functional properties.
- *Extendability*: Pluggable extensions of Web service functionality.
- *Distribution*: Generation and deployment of Web services on remote hosts.
- *Complexity*: Control structures and complex interdependencies between services.
- *Integration*: Integration with existing SOA infrastructures.
- *Convenience*: Supporting the developer with a convenient API.

Hence, the tool should be *adaptable to the needs of the test cases, instead of expecting the test cases to be adapted to the limitations of the tool itself.* To our best knowledge, there is an absence of solutions which fulfill this. GENESIS was designed to fill this gap.

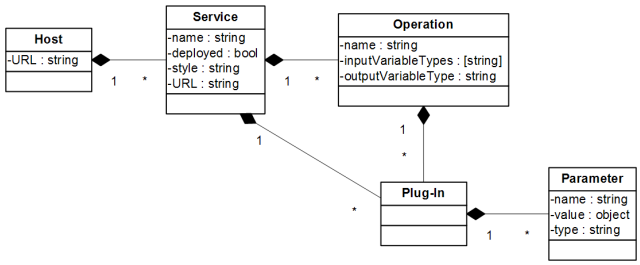

Figure 1. Model used to describe Web services in GENESIS

3 Concept and Architecture of GENESIS

The architecture of GENESIS (see Figure 2) includes a single front-end part, for centralized control, and a distributed back-end hosting the Web services. Via the front-end it is possible to specify the components and characteristics of the testbed, while the back-end's task is to generate the testbed infrastructure based on this information. For this reason, both parts share a common description model for Web services, based on which they exchange data. Figure 1 depicts a simplified class diagram of this data model, which consists of the following structures:

- *Host*: A back-end host contains a set of Web services and is identified by a unique URL pointing to the GENESIS instance running on it.
- *Service*: A Web service has a name, a unique URL, and a set of operations. It can be either deployed or undeployed and can communicate in an RPC-based or message-oriented manner. Furthermore, the service can reference plug-ins which are being invoked at deployment and undeployment.
- *Operation*: An operation has a name, a set of input types, and a single output type. A generic fault type is predefined. Operations can be extended by referencing plug-ins.
- *Plug-In*: Plug-ins extend the Web services' functionality and declare a set of parameters via which their behavior can be steered.
- *Parameter*: A parameter must be declared by a plug-in in order to be accessible. It has a name, a data type, and a value.

Based on this model, GENESIS provides a Java API for creating and manipulating Web service descriptions and for transferring them to the back-end. The developer is free to utilize the API in his/her own applications at the front-end. Alternatively, the developer can use the provided

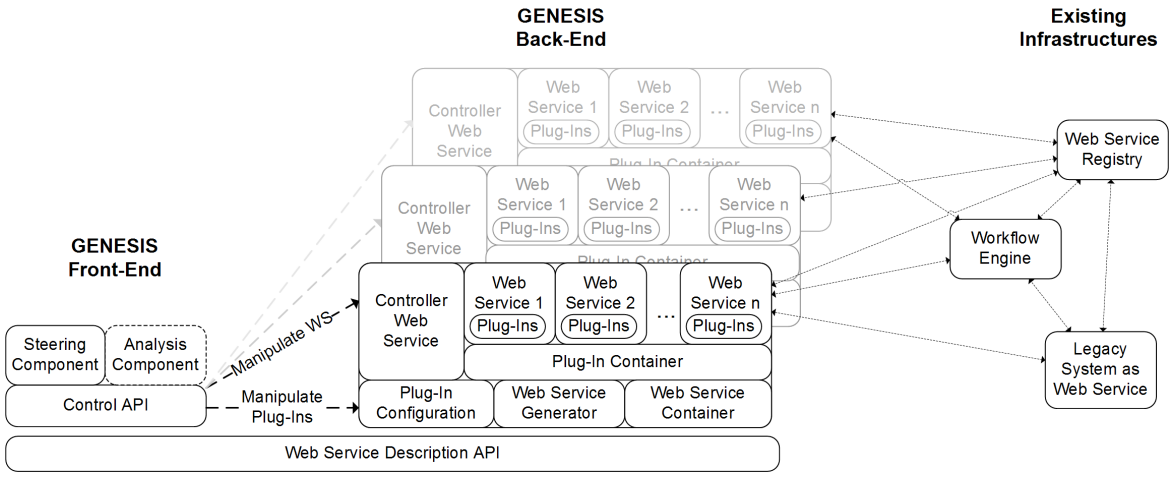

Figure 2. Architecture of GENESIS

Steering Component which is built upon the Jython script interpreter [9]. In Section 5 we show sample scripts which demonstrate how testbeds can be created and steered.

At the back-end side, the functionality is split into modules. Incoming requests, encoded as Web service descriptions, are received by the *Controller Web Service* and forwarded to the *Web Service Generator* which transforms them into real service instances. These instances are being deployed at the JAX-WS-based [8] *Web Service Container*. This step is described in more detail in Section 4.1. Plug-ins, which implement extensions to these Web services, are registered at the *Plug-in Container* and are being controlled by changing their parameters in the local *Plug-in Configuration Database*. We provide a set of basic plug-ins which implement, for example, simulation of QoS attributes and workflow functionality. In Section 4.2 we give a short overview about them and explain how custom ones can be developed.

In between the front-end and the back-end, the communication is based on SOAP as well as on a simple text- and TCP-based protocol, as illustrated in Figure 3. Although we decided to provide access to all relevant functionality of the back-end via a HTTP-based SOAP Web service, we regarded it also as necessary to establish a light-weight communication for the manipulation of remote plug-in parameters. Since these parameters may be changed frequently, it makes sense to avoid unnecessary protocol overhead, such as SOAP envelopes, but to exchange data in a fast manner.

4 Generating Complex Web Services

At the back-end, the generation of Web services is derived from techniques used in Model-driven development (MDD) [16, 17, 33, 38, 39] and the idea of extending ser-

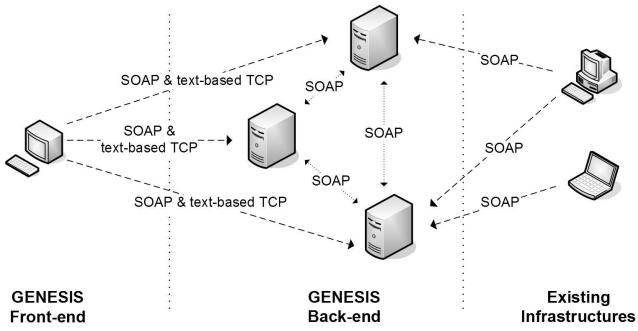

Figure 3. Communication in the testbed

vice skeletons with plug-ins. The *Web Service Generator* creates by default plain yet runnable Web services. Via plug-ins these can be extended to provide real functionality, to establish complex interdependencies inside the testbed, to simulate non-functional attributes, and to introduce any kind of customized behavior.

4.1 Generating Web Services

The *Web Service Generator* parses the service description to retrieve knowledge about the interfaces and functionality of the service, and generates a deployable Web service instance based on this knowledge. This procedure involves multiple steps which are depicted in the sequence diagram in Figure 4:

1. The Web service description is checked for referenced plug-ins. Plug-ins which are missing at the remote back-end host have to be transferred and registered.

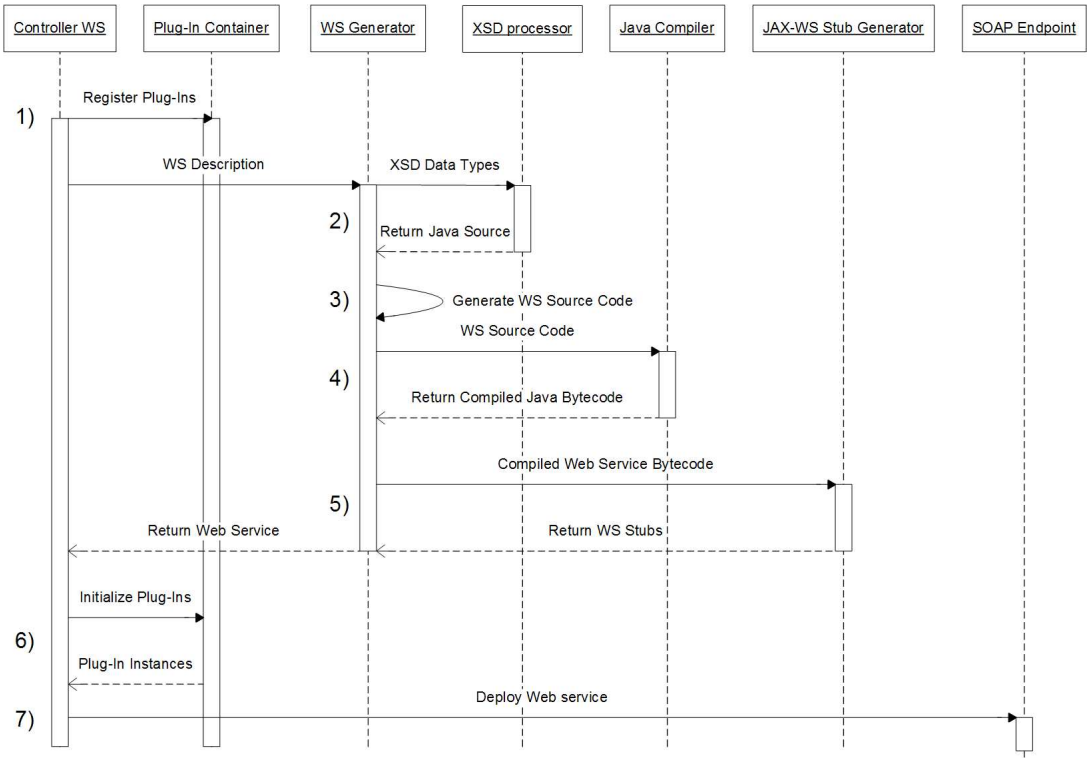

Figure 4. Web service generation

2. The description is checked whether the service uses solely primitive data types (e.g., string, integer) inside the requests and responses or whether complex data types, described in XML Schema Definitions (XSD) [14], are referenced. Complex types are passed to `xjc`, which is a XSD processor of JAX-WS, for generating corresponding Java classes.

3. The JAX-WS-compliant source code of the Web service is generated using Apache Velocity-based [1] templates.

4. The source code is passed to `javac`, the Java compiler.

5. The compiled Web service is passed to `wsgen`, which is again a part of JAX-WS, to generate the necessary stubs for deployment.

6. The class loader reads in the compiled Web service, instantiates it and initializes all plug-ins.

7. Finally, the Web service is deployed at the HTTP-SOAP endpoint.

Listing 1 shows the generated Java source code of a sample Web service named `BookService`. According to the description, the communication is set to Remote Procedure Call (Line 8) and an operation named `getISBN()` is generated (Lines 16-36). The Web service extends the abstract class `AWebService` which provides basic yet mandatory functionality. This includes the invocation of plug-ins (Lines 27-28, `callPlugin()`) as well as the generation of a context for all plug-ins (Line 21, `getContext()`). The context is used by the plug-ins to access the operation arguments, to set a return value, and also as a communication facility for passing data between individual plug-ins. Apart from being used inside the web service operation `getISBN()`, plug-ins are invoked on deployment and undeployment (Lines 39-52) to register the service at a registry.

4.2 Establishing Complex Dependencies through Plug-Ins

Service-oriented architecture (SOA) is often propagated as an all-round solution to miscellaneous software engineering issues which have existed since decades, such as integration of heterogeneous systems, component decoupling, or software reuse. The concept of SOA is based on public available services exchanging data and being coordinated and composed in a flexible and optimized manner, regard-

```
1  package repository.wsp103292;
2
3  import ...;
4  import at.ac.tuwien.vitalab.genesis.server.AWebService;
5
6  @WebService(name = "BookService",
7              targetNamespace = "http://...")
8  @SOAPBinding(style = SOAPBinding.Style.RPC)
9  public class BookService extends AWebService {
10
11   // constructor
12   public BookService() {
13     wsName="BookService";
14   }
15
16   @WebMethod
17   public String getISBN(
18     @WebParam(name="article") schemaTypes.Book article
19     ) throws Exception {
20
21     WebServiceContext context=getContext("getISBN");
22
23     // make arguments available for plug-ins
24     context.argumentValues.put("article",article);
25
26     // call the plug-in(s)
27     callPlugin("QOSPlugin.simulateDelay",context);
28     callPlugin("QOSPlugin.simulateFailure",context);
29
30     // check whether return value has been set
31     if (context.returnValue!=null) {
32       return context.returnValue;
33     }
34     // otherwise create dummy
35     return (String)createDummyObject(String.class);
36   }
37
38   // deployment hook
39   protected void onDeploy() {
40     WebServiceContext context=getContext("onDeploy");
41
42     // call the plug-in(s)
43     callPlugin("RegistryPlugin.register",context);
44   }
45
46   // undeployment hook
47   protected void onUndeploy() {
48     WebServiceContext context=getContext("onUndeploy");
49
50     // call the plug-in(s)
51     callPlugin("RegistryPlugin.deregister",context);
52   }
53 }
```

Listing 1. Source of generated Web service

ing their descriptions and properties. However, the more coordination and composition is needed, the more the system becomes complex. Taking languages for service choreography and orchestration, such as WS-CDL [13] or BPEL [3], as examples, we can identify various forms of complexity, e.g., dependencies between services, control constructs, fault handlers, optimization techniques, and service discovery. It is safe to say that the overall complexity of a SOA-based system increases with the number and intricacy of interdependencies between the services. In GENESIS, complexity inside the testbed can be realized by applying plug-ins to the individual Web services.

At the implementation level, these plug-ins must extend an abstract class called AWebServicePlugin, which defines mandatory constructor and destructor methods, provides serialization functionality for transferring plug-in code to remote hosts, and registers itself automatically at the corresponding container. Each plug-in gets access to the input and output variables of the invoked operations, to the SOAP headers, and to all other Java artifacts which are visible inside the Web service's scope. Furthermore, the plug-in is free to define a set of parameters through which it can be controlled remotely from the front-end. Taking as example a plug-in for registering the Web service at some registry, such as UDDI [11], possible parameters would specify the host name of the registry server, authentication data, and additional meta-data about the service.

With the current implementation of GENESIS, we provide four sample implementations of plug-ins:

QOSPlugin: Simulates performance- and dependability-specific QoS metrics, such as processing time, scalability, throughput, availability, and accuracy. Performance attributes are simulated by delaying responses, while dependability is simulated by throwing faults and making the service unavailable. In the current status the *QOSPlugin* works in a simple manner with predefined dependency curves between processing time, scalability, and throughput and without the possibility to simulate different QoS behavior for different input data. The plug-in can be controlled by setting the corresponding parameter for each metric, e.g., a percentage value for availability.

BPELPlugin: Integrates the bexee [2] BPEL engine into GENESIS and executes composed processes inside the Web service operations. As a parameter it accepts BPEL process definitions which can contain precise as well as abstract partnerLinks. Precise partnerLinks allow to integrate external Web services, for instance already existing SOA infrastructures. Abstract definitions just specify the portType and operation name and are being resolved during runtime to concrete services, based on the current status of the testbed. For this, the *BPELPlugin* places hooks inside GENESIS to be aware of all deployed Web services in the testbed. As a result, it is possible to express complex service interdependencies in simplified and flexible BPEL code.

LogPlugin: Logs the invocations of Web services and the interactions within them. The format and destination of the logs is specified via parameters.

RegistryPlugin: Registers and deregisters the Web service at a registry. Currently, we only support UDDI but we plan to extend it for VReSCO [27] and other standards. In contrast to the other plug-ins, the *RegistryPlugin* must be invoked at the deployment and undeployment

of the Web service, instead of being used inside the Web service operations. The host and the authentication data of the remote registry have to be specified via parameters. The meta-data about the service is mainly retrieved from the description of the Web service itself.

In the current status, plug-ins expose their functionality via public methods which can be invoked in a sequential manner. Although concurrence can be implemented by using threads and synchronization, we plan to introduce a more flexible approach in the future, where plug-ins are arranged and controlled based on event-driven programming.

5 Practical Application of GENESIS

GENESIS provides a Java API which covers all functions to specify, to generate, and to steer testbeds. The API can be embedded into any application, for instance a GUI, or into the Bean Scripting Framework [7] which seamlessly integrates scripting languages into Java. As a starting point for developers we provide a *Steering Component* based on the Jython script interpreter [9]. Jython establishes a convenient combination of the simplicity of Python scripts and the flexibility of the API and, furthermore, allows to control a simulation interactively as well as in an automated manner. In the following, we show some sample scripts which demonstrate how GENESIS can be applied in practice.

5.1 Testbed Configuration

In GENESIS, the testbed can be built from scratch by defining all properties manually or, preferably, by using the configuration facility of the API. The configuration itself contains templates and declarations which can be reused later. Listing 2 shows a sample configuration file.

First, all plug-ins are imported (Lines 2-5) and, if necessary, the default values of their parameters are overridden (Line 7). Furthermore, plug-ins can be joined to behavior groups (Lines 9-15) which can be referenced later to combine individual functionalities of plug-ins.

Second, complex data types, which are used inside request and response messages, can be defined in inline XML Schema definitions (Lines 17-21) or can be imported from external files.

Finally, Web services are specified, which are either declared as abstract templates (Lines 24-42) or as deployable instances inside host declarations (Lines 47-55). By using abstract services, the developer defines common properties which can be derived and extended for the sake of reuse. This reduces the efforts for deployment of large environments consisting of similar services. Service operations are declared (Lines 25-35, 51-54) containing a list of request data types, a single response data type, and a list of invoked

```
1  <configuration>
2    <plugins>
3      at.ac.tuwien.vitalab.qos.QOSPlugin
4      at.ac.tuwien.vitalab.qos.RegistryPlugin@/path/reg.jar
5    </plugins>
6
7    <defaultparameters qos_availability="0.95" ... />
8
9    <behavior>
10     <QOS default="true">
11       QOSPlugin.simulateDelay
12       QOSPlugin.simulateFailure
13     </QOS>
14     <EmptyBehavior/>
15   </behavior>
16
17   <schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
18     <xs:complexType name="book">
19       ...
20     </xs:complexType>
21   </schema>
22
23   <servicetemplates>
24     <service name="GenericService">
25       <operation name="echo" >
26         <!-- override default parameters -->
27         <parameters qos_processingtime="1000"/>
28         <input request="string"/>
29         <output return="string"/>
30         <!-- override default behavior -->
31         <behavior>
32           EmptyBehavior
33         </behavior>
34         </output>
35       </operation>
36       <deploy>
37         RegistryPlugin.register
38       </deploy>
39       <undeploy>
40         RegistryPlugin.deregister
41       </undeploy>
42     </service>
43   </servicetemplates>
44
45   <environment>
46     <host address="http://somehost:8080/some/path" >
47       <service name="BookService"
48                template="GenericService"
49                deploy="true">
50         <!-- extend template service -->
51         <operation name="getISBN" >
52           <input article="xs:book"/>
53           <output return="string"/>
54         </operation>
55       </service>
56     </host>
57   </environment>
58 </configuration>
```

Listing 2. Testbed configuration

plug-ins and their local parameters. If the declaration of plug-ins is left away, the default behavior is assumed, which was defined at the beginning of the file. Moreover, plug-ins can be invoked during deployment and undeployment of a service (Lines 36-41).

The sample configuration in Listing 2 defines the service BookService which derives the operation echo() from the template GenericService and extends it with getISBN(). The source code in Listing 1 was generated from this configuration.

5.2 Generation and Steering of Web Services

The following Jython code snippets demonstrate the convenience of GENESIS in deploying Web services and steering their behavior. In the first sample, the service newService is deployed on a remote host, with one plain operation named helloWorld invoking the *QOSPlugin*. A correct deployment can be verified by checking the generated WSDL description of the service at http://somehost:8080/some/path/newService?WSDL.

```
from at.ac.tuwien.vitalab.genesis.model import *

remoteHost=Host("http://somehost:8080/some/path")
newService=Service("newService")
helloOperation=Operation("helloWorld")

inputs=LinkedHashMap()          % ordered input types
inputs.put("arg","string")      % input name & type
helloOperation.setInputTypes(inputs)
helloOperation.setOutputType("string")

plugins=Vector()                % set plug-in references
plugins.add("QOSPlugin.simulateQOS")
operation.setInvocations(plugins)

newService.addOperation(helloOperation)
remoteHost.addService(newService)

newService.deploy()             % generate at back-end
```

In cases where the back-end has been populated with Web services before, the front-end application can retrieve the handles to these services from the remote hosts and start working on them.

```
remoteHost.loadServices()

remoteHost.listServiceNames()          % print names
> [someService, newService]

newService=localhost.getService("newService")
```

Control on plug-ins is achieved by modifying parameters via getter/setter methods, whereas the setter methods forward the changes to the corresponding plug-ins at the back-end. When simple setting of parameter values is not sufficient to implement a desired behavior, the developer can use API methods for sophisticated manipulation. The following code snippet shows how a plug-in parameter can be changed continuously according to a sine function.

```
param=helloOperation.getParameter( \
                                QOSPlugin.PROCESSINGTIME)

param.getValue()                    % print value
> 2000
param.setValue(2500)                % simple setter

i = 0
def sine():                         % define sine func.
... global i
... i = i +1
... return 1000+Math.round( \
                        Math.sin(Math.toRadians(i))*500)

param.setValue(sine,360,1000)       % change acc. to sine
```

The GENESIS API provides various other methods for sophisticated control of plug-ins, e.g., observe/notification mechanisms for parameters. However, for the sake of simplicity we showed rather primitive functions in the previous samples, where only the *QOSPlugin* was used inside a standalone service which was not composed of other services. The following example creates a more complex service using a template, which executes a BPEL process in the background and, in addition, simulates failures.

```
testbed=Testbed("/path/to/testbed.config")
template=testbed.getServiceTemplate("GenericService")

complexService=newService("ComplexService",template)
operation=new Operation("run")

inputs= ...                         % list of input types
operation.setInputTypes(inputs)
operation.setOutputType("xs:Statistics")    % xsd type

pluginInvocations=Vector()
pluginInvocations.add("BPELPlugin.run")
pluginInvocations.add("QOSPlugin.simulateFailure")
operation.setInvocations(pluginInvocations)

bpelParam=operation.getParameter(BPELPlugin.BPEL)
bpelParam.setValue("/path/to/some.bpel")

complexService.addOperation(run)
remoteHost.addService(complexService)

complexService.deploy()
```

A complex testbed can be set up easily by combining multiple of such services. Interdependencies between them can be handled by abstract BPEL processes which resolve abstract partnerLinks pointing to other services at runtime. Furthermore, a realistic behavior of the testbed can be simulated by alternating the QoS properties of the individual services, which in return effects the QoS of the composed ones.

5.3 Illustrating Example

In [27] Michlmayr et al. present the VReSCO project which addresses some of the current challenges and issues of service-oriented computing. In particular, they claim that the well-known *provider-broker-requester* triangle of SOA seems to be broken and today's SOA applications rely on exact endpoint addresses instead of finding services dynamically at the broker which is also referred to as the registry. According to [27], this happens mainly due to the shortcomings of the currently available Web service registries, UDDI and ebXML, which are too complicated and too heavy-weight.

The idea of VReSCO aims at solving this problem by providing a registry infrastructure which supports SOA developers with dynamic binding and invocation of services. In that approach, services are published dynamically at runtime to the other participants within the network and are described by meta-data of functional and non-functional attributes, e.g., monitored QoS attributes [30]. Based on this

meta-data, it is possible to search and query for services and to bind and invoke them dynamically. Moreover, clients can subscribe to notifications about new services appearing in the network and as well about changing attributes or interfaces of already registered services. In addition, the registry is coupled to an orchestration engine for providing semi-automatic service composition.

Since VReSCO was designed to disburden SOA developers from handling various difficulties of dynamically changing service environments, it is necessary to test the system at runtime on a dynamic testbed consisting of real services. In `GENESIS`, such a testbed can be created easily by applying:

- The *QOSPlugin* for simulating changing non-functional attributes (performance and dependability) which will be monitored periodically by VReSCO.
- The *BPELPlugin* for creating complex services whose non-functional attributes depend directly on the referenced services.
- A plug-in for registering deployed Web services automatically at the VReSCO infrastructure. This is a planned extension for the *RegistryPlugin*.
- A control mechanism at the front-end, which manipulates the QoS attributes of the services inside the testbed to simulate temporal unavailability and performance variations.

By deploying VReSCO on such a testbed, the developers could perform various checks to identify potential problems of the system at runtime and can verify whether VReSCO reacts correctly to the dynamics of the environment. In particular, various test cases can be enacted which identify performance bottlenecks of the system, determine the overall stability and scalability, and also help to point out constraints which can only be identified at runtime tests.

The VReSCO example illustrates clearly the typical area of application for `GENESIS`, where a realistic environment of Web services is needed as a testbed for runtime simulations.

6 Implementation Details

For deploying Web services the world of Java offers multiple facilities, such as application servers (e.g., IBM WebSphere, Sun GlassFish, JBoss), service engines relying on Servlet containers (e.g., Apache Axis), and standalone solutions, such as JAX-WS [8]. Since we wanted to keep `GENESIS` light-weight and avoid any unnecessary dependence on other components, we decided to use JAX-WS 2.0, especially after Sun Microsystems made it an official part of Java 6 [32]. JAX-WS 2.0 offers a convenient method to develop and deploy SOAP-based Web services. Although it is not as powerful as for instance Apache Axis, it provides all the functionality we needed for our purposes, such as support of data types described in XML Schema, automatic WSDL generation, RPC and message-oriented communication, and simple deployment on HTTP-based SOAP endpoints.

The generation of Java source code is based on the Apache Velocity [1] template engine. The source code is created by using default templates for JAX-WS service classes and replacing placeholders with concrete Java expressions derived from the Web service descriptions. This way we kept the generation as flexible as possible, making even modifications during runtime feasible with minimal effort.

Regarding the performance of `GENESIS`, the framework itself consumes only marginal amounts of CPU cycles or memory. However, we observed a bottleneck at the back-end when `xjc`, `javac`, and `wsgen` are being invoked for compiling Web services. Experiments, carried out on a Linux-based laptop with 2 GB of RAM and a 2 GHz Intel Core 2 Duo CPU, showed that the combined compilation takes at least 3 seconds for simple services. As this delay is mainly caused by the time necessary to start up the individual programs, we also observed that the overall compilation time does not increase significantly for more complex services. Furthermore, this procedure can be optimized by allowing parallel compilation if the back-end hosts are equipped with multiple CPU cores.

7 Related Work

Testing of Service Oriented Architectures requires in general support of two kinds: (a) tools for executing the test cases, including runtime-based tests as well as formal ones, and (b) tools supporting the developer in setting up these test cases. The active execution of testings and simulations has been addressed in many works of which we discuss the most relevant ones. Unfortunately, only few solutions exist for solving the second issue. Here our review of relevant work is focused more on automatic generation of Web services in general.

Formal methods, such as situation calculus [28, 29] and petri-nets [15, 37], are widely used for verification of service compositions. However, with these methods it is only possible to analyze the composition models at a high-level, neglecting their runtime behavior.

The High Level Architecture (HLA) [4, 5, 6] is an architecture standard for distributed simulations. The idea is to split a simulation into several sub-simulations which can exchange data and are being controlled by a centralized runtime infrastructure. It consists of an interface specification, an object model, and a set of rules. It is a powerful tool for

controlling simulations, which could be used in combination with GENESIS in order to have sophisticated control on a simulation operating on real Web services created on demand.

DDSOS [35] is a framework based on the principles of HLA, which provides model-and-run support for distributed simulation, dynamic model checking and verification, multi-agent simulation, etc. Furthermore it contains an automated scenario code generator, where scenarios describe the behavior of the simulations. DDSOS translates the scenarios from PSML to executable code in several phases by using the idea of code templates which are complemented according to the specified models. This happens on the levels of a service model, a system model, and an environment model, whereas the first two are platform independent. However, the environment model describes the details of the destination platform, such as middleware used, operating system, etc. By binding it to GENESIS it is possible to generate automatically real Web services environments for run-time simulation and verification of SOAs.

Puppet [19] is a tool for automatic generation of testbeds for evaluation of QoS features of Web services. In particular it aims at Web services which are under development, and therefore not deployable yet, and wraps them into generated service skeletons which simulate QoS behavior. As input it expects WSDL and WS-Agreement documents, from which it generates the interfaces and QoS simulation code. The functionality of Puppet is only limited to QoS simulation and the parameters cannot be changed during runtime. In contrast, GENESIS was designed to offer an extensible functionality to the developers, allowing to set up customized testbeds.

Much relevant work has been done in the area of Model Driven Development (MDD) of Web services. In [16] an approach for semi-automatic generation of Web service artifacts is presented. These artifacts include workflow definitions as BPEL, Web service interfaces as WSDL, and security constraints in WS-Policy. A similar approach is described in [17], where service templates and executable specifications are generated for simplifying the development of Web services. In [38] a framework is presented, which uses UML (Unified Modeling Language) specifications for Enterprise Distributed Object Computing (EDOC), which are translated into Web service skeletons and, optionally, BPEL processes. GENESIS is based on the same principles, but allows to manipulate the model on the fly and to adapt a back-end of real Web services automatically to it. However, it makes sense to combine the listed MDD approaches with GENESIS to simplify the specification of BPEL-based interdependencies inside the testbeds. Furthermore, model-driven methods are also being used to generate executable test cases from abstract and platform-independent test data models [20, 24].

8 Conclusion and Future Work

Designing and testing of distributed complex service-oriented systems is composed of a set of challenges which need to be addressed in essentially all complex systems. In this paper we presented GENESIS, a software framework for solving some crucial challenges in the development and testing of complex service-oriented systems. Our approach and implementation support automatic generation and steering of testbeds of complex Web services, by empowering developers to (a) specify functional and non-functional properties of Web services, to (b) generate the services automatically on remote hosts, to (c) enhance the functionality of the framework with plug-ins, and, to (d) control and steer the behavior of the testbed during runtime.

One powerful mechanism of GENESIS includes the plug-ins. We have implemented some to demonstrate its usefulness and plan to extend the number of them in future versions with the goal to allow a more complex arrangement of the plug-ins instead of the current linear sequence. Furthermore, we are investigating the utilization of Model-driven development (MDD) for generating input for our BPEL plug-in. We plan to release GENESIS under an open source license.

References

[1] Apache Velocity. http://velocity.apache.org.

[2] bexee - BPEL Execution Engine. http://bexee.sourceforge.net/.

[3] Business Process Execution Language for Web Services Version 1.1. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf.

[4] IEEE 1516-2000: High Level Architecture - Framework and Rules.

[5] IEEE 1516.1-2000: High Level Architecture - Federate Interface Specification.

[6] IEEE 1516.2-2000: High Level Architecture - Object Model Template (OMT) Specification.

[7] Jakarta BSF - Bean Scripting Framework. http://jakarta.apache.org/bsf/.

[8] Java API for XML Web Services (JAX-WS). https://jax-ws.dev.java.net/.

[9] Jython - Python in Java. http://www.jython.org.

[10] SOAP specifications at W3C. http://www.w3.org/TR/soap/.

[11] Universal Description Discovery and Integration. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[12] Web Services Activity at W3C. http://www.w3.org/2002/ws/.

[13] Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[14] XML Schema. http://www.w3.org/XML/Schema.

[15] N. R. Adam, V. Atluri, and W. kuang Huang. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst.*, 10(2):131–158, 1998.

[16] R. Anzböck and S. Dustdar. Semi-automatic generation of web services and bpel processes - a model-driven approach. In W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 64–79, 2005.

[17] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In A. Persson and J. Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2004.

[18] 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*, pages 612–617. IEEE Computer Society, 2007.

[19] A. Bertolino, G. D. Angelis, and A. Polini. Automatic generation of test-beds for pre-deployment qos evaluation of web services. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 137–140. ACM, 2007.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[21] H. El-Rewini and W. Halang. The engineering of complex distributed computer systems. *IEEE Concurrency*, 05(4):30–31, 1997.

[22] M. Ellims, J. Bridges, and D. C. Ince. Unit testing in practice. In *ISSRE*, pages 3–13. IEEE Computer Society, 2004.

[23] W. E. Howden. Functional program testing. *IEEE Trans. Software Eng.*, 6(2):162–169, 1980.

[24] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *AST'07*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[25] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9):30–38, 1994.

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

[27] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards recovering the broken soa triangle - a software engineering perspective. In *IW-SOSWE, Dubrovnik, Croatia*. ACM Press, 2007.

[28] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88. ACM Press, 2002.

[29] S. Narayanan and S. A. McIlraith. Analysis and simulation of web services. *Computer Networks*, 42(5):675–693, 2003.

[30] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping performance and dependability attributes of web services. In *ICWS*, pages 205–212. IEEE Computer Society, 2006.

[31] A. D. Stoyenko. Engineering complex computer systems: A challenge for computer types everywhere - part 1: Let's agree on what these systems are. *IEEE Computer*, 28(9):85–86, 1995.

[32] Sun Developer Network. Introducing JAX-WS 2.0 With the Java SE 6 Platform, September 2006. `http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/`.

[33] H. Tran, U. Zdun, and S. Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven soa. In *BPSC, Leipzig, Germany*, 2007.

[34] W.-T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, and R. A. Paul. Testing web services using progressive group testing. In C.-H. Chi and K.-Y. Lam, editors, *AWCC*, volume 3309 of *Lecture Notes in Computer Science*, pages 314–322. Springer, 2004.

[35] W.-T. Tsai, C. Fan, Y. Chen, and R. A. Paul. Ddsos: A dynamic distributed service-oriented simulation framework1. In *Annual Simulation Symposium*, pages 160–167. IEEE Computer Society, 2006.

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

[37] W. M. P. van der Aalst. Challenges in business process management: Verification of business processing using petri nets. *Bulletin of the EATCS*, 80:174–199, 2003.

[38] X. Yu, J. Hu, Y. Zhang, T. Zhang, L. Wang, J. Zhao, and X. Li. A model driven development framework for enterprise web services. In *EDOC*, pages 75–84. IEEE Computer Society, 2006.

[39] U. Zdun, C. Hentrich, and S. Dustdar. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *TWEB*, 1(3), 2007.

[40] X. Zhou, W.-T. Tsai, X. Wei, Y. Chen, and B. Xiao. Pi4soa: A policy infrastructure for verification and control of service collaboration. In *ICEBE*, pages 307–314. IEEE Computer Society, 2006.