CAGE: Customizable Large-Scale SOA Testbeds in the Cloud*

Lukasz Juszczyk[1], Daniel Schall[1], Ralph Mietzner[2], Schahram Dustdar[1], and Frank Leymann[2]

[1] Distributed Systems Group, Vienna University of Technology
lastname@infosys.tuwien.ac.at
[2] Institute of Architecture of Application Systems, University of Stuttgart
lastname@iaas.uni-stuttgart.de

Abstract. Large-scale and complex distributed systems are increasingly implemented as SOAs. These comprise diverse types of components, e.g., Web services, registries, workflow engines, and services buses, that interact with each others to establish composite functionality. The drawback of this trend is that testing of complex SOAs becomes a challenging task. During the development phase, testers must verify the system's correct functionality, but often do not have access to adequate testbeds. In this paper, we present an approach for solving this issue. We combine the *Genesis2* testbed generator, that emulates SOA environments, with *Cafe*, a framework for provisioning of component-based applications in the cloud. Our approach allows to model large-scale service-based testbed infrastructures, to specify their behavior, and to deploy these automatically in the cloud. As a result, testers can emulate required environments on-demand for evaluating SOAs at runtime.

1 Introduction

Service-oriented computing (SOC), based on messages being exchanged among loosely-coupled components, provides a high level of flexibility and scalability which benefits the realization of large-scale and complex distributed systems [1], called service-oriented architectures (SOAs). SOAs do not only comprise Web services, clients, and registries/brokers, as depicted in the Web service triangle [2], but integrate diverse components, such as service buses, message mediators, monitors, governance systems, etc. By applying asynchronous communication via exchanging SOAP messages, and avoiding blocking RPC-like invocations, these systems can potentially scale to large dimensions. Moreover, due to the ability of dynamic binding, SOA systems can integrate new components/services and grow (and shrink) dynamically at runtime. Taking a look at the composition of typical SOAs, two categories of components emerge: (a) stand-alone components, such as single Web services which do not depend on others, and (b) complex components,

* This work is supported by the EU through the projects COMPAS (No. ICT-2008-215175) and S-Cube (No. ICT-2007-215483).

E.M. Maximilien et al. (Eds): ICSOC 2010 Workshops, LNCS 6568, pp. 76–87, 2011.

which interact with others and, this way, manage the SOA and/or establish composite functionality. Obviously, also SOA systems must be tested intensively before final deployment in order to verify their correct execution. For stand-alone components, the testing procedure is well-supported (as outlined in Section 5) or, at least, does not pose new challenges compared to traditional software testing. However, engineers that develop complex components which operate in dynamic SOA environments are facing the problem of how to test their software. They must ensure that their components are able to scale with a growing number of participating services, that quality of service requirements are met, that the software is stable and dependable, etc. Characteristics like these can only be verified by testing the developed component at runtime and in a multitude of real(istic) scenarios. This, however, implies that the component must be deployed in these different designated environments for getting meaningful test results. Unfortunately, testers often do not have access to the designated environments during the development phase, e.g., either because some parts are not available yet or because it integrates commercial external services, which would make testing costly. Furthermore it is often impossible to perform multiple tests (for example, regression tests and load tests) at the same time because the test infrastructure does not offer enough of resources.

In this paper we are trying to solve these issues. We present *Cage*, a framework and methodology for emulating SOA environments (for utilization as testbeds) and for deploying these automatically in the cloud. Our approach combines two complementary frameworks: *Genesis2* [3], for generating SOA testbeds, and *Cafe* [4], for provisioning these across a cloud platform. Cage allows testers to specify *testbed families* consisting of diverse SOA components and variability, to customize their behavior and non-functional properties, and to automatically generate running testbed instances based on the customization. Furthermore, by using the cloud as a platform, Cage provides a convenient and cost-efficient way to set up multiple arbitrarily large testbed instances simultaneously on-demand. In a nutshell, the most distinct contributions of our approach are:

1. Separation of testbed development and testing via *testbed families*
2. A self-service *testbed portal* for testbed customization
3. An infrastructure to provision and run complex, flexible testbeds on-demand

Our approach is presented as follows. Next, we present the motivation for our research. In Section 3 we introduce the two base frameworks Cage consists of. Section 4 describes the combined Cage framework and its functionality. Finally, in Sections 5 and 6 we review related work and conclude our paper.

2 Motivation: Large-Scale SOA Testbed Infrastructures

Loose coupling and dynamic binding have always been listed among the most important features of Web service-based SOA. By avoiding static interactions but being able to choose services dynamically, SOAs are capable of incorporating new participating services at run-time and to cope with unavailable ones

by rebinding to alternatives. This flexibility makes it possible to build systems which are not restricted to an environment of fixed size and topology, but are able to deal with dynamic and large-scale ones. Let us take Amazon Mechanical Turk[1] (MTurk), a crowdsourcing platform, for example. MTurk registers Web services of human workers and provides these to clients which incorporate the offered functionality into their applications/workflows. MTurk must be able to handle load peaks, scale with the number of registered services and consuming clients, and, despite all possible difficulties, provide stable and dependable services. However, loose coupling, dynamic binding, or other typical SOA-features do not solve the scalability issue per se. The system's internal mechanisms must be still able to cope with a high number of partner components (e.g., services, clients, workflow engines) and incoming requests. During the development of such systems the question appears of how to test these mechanisms and how to verify their correct execution in critical scenarios. These scenarios can comprise thousands of components which have diverse functional as well as non-functional properties, and, therefore, put high load on the system. Setting up such scenarios for testing purposes is an intricate task. In a nutshell, testers are confronted with the problem of a) how to create testbeds which emulate realistically the final deployment environment and b) where to host large-scale testbeds in a cost-efficient manner.

We have elaborated on the first issue in [3] by showing how our Genesis2 framework supports the generation of customizable SOA testbed infrastructures. In our approach testers model SOA environments and Genesis2 takes care of generating and deploying running testbed instances implementing the modeled behavior. Of course, an adequate back-end hardware infrastructure is required in order to be able to host all generated SOA components, which can get very costly for large-scale testbeds. In the last years cloud computing emerged as an interesting solution to this problem, as it enables users to rent hardware on-demand, also referred to as Infrastructure as a Service (IaaS). Instead of buying expensive hardware for hosting testbeds, which is most likely running idle in periods when no test runs are performed, engineers can rent remote hardware for an arbitrary time and quit the service when it is no longer required. In our Cage approach we make intense use of IaaS. We apply the Cafe framework which supports automatic allocation hosts/servers in the cloud and, this way, provides a flexible hosting infrastructure for SOA testbeds. The most significant contribution of our approach is that we enable testers to generate functional SOA testbeds on-demand on a dynamically allocated back-end infrastructure.

3 Base Frameworks for CAGE

Cage derives it functionality (and name) from combining two base frameworks: *Cafe* and *Genesis2*. Cafe provides support for an automated provision of component-based applications into the cloud, while Genesis2 makes use of the infrastructure provided by Cafe and generates customizable and dynamic SOA

[1] `http://www.mturk.com/`

testbeds. In the following these frameworks are shortly introduced to outline their concepts.

3.1 Cafe

Cafe (composite application framework) [4] is a framework for definition, customization and automatic provisioning of complex composite applications in the cloud. Cafe is centered around the basic concept of an *application template* which consists of an *application model* and a *variability model*. The application model describes the basic components of the application, whereas the variability model describes variability of the application. Application templates are offered to customers by a provider in an *application portal*. The customers are guided through the customization of the template by a *customization flow* that is generated from the variability model of the application. Once a the template is transformed into an customer-specific *application solution*, this solution is then automatically provisioned by the *provisioning infrastructure*. Cafe makes use of the interfaces of different cloud providers, such as Amazon EC2[2], to setup components.

A Cafe application model contains a set of components that realize the functionality of the application. Components can be arbitrary elements of an application such as middleware components that are supplied by a provider, or components where the code is shipped with the application (*internal components*), such as services, Web applications, or business processes. Provider-supplied components must have a special *component type* that indicates a class of components such as JBOSS application server components. Internal components are of a certain *implementation type*, e.g., JEE application or BPEL process. Component types define if components of a certain implementation type can be deployed on them. Components can have deployment relations among them, indicating that one component must be deployed on another component. Application template developers use internal and provider supplied components and their deployment relations to describe their application.

3.2 Genesis2

The purpose of the Genesis2 framework [3] (in short, G2) is to support the setup of testbeds for SOA. It allows to emulate environments consisting of services, clients, registries, and other SOA components and to program their behavior. G2's most distinct feature is its ability to generate running testbed instances (in contrast to performing pure simulations) and to integrate these into existing SOA environments, which empowers testers to evaluate SOA systems at runtime.

G2 comprises a centralized front-end, from where testbeds are modeled and controlled, and a distributed back-end, consisting of hosts (in Cage these are located in the cloud) at which the models are transformed into real testbed instances. The front-end maintains a virtual view on the testbed, allows to manipulate it on-the-fly via scripts, and propagates changes to the back-end in order

[2] `http://aws.amazon.com/ec2/`

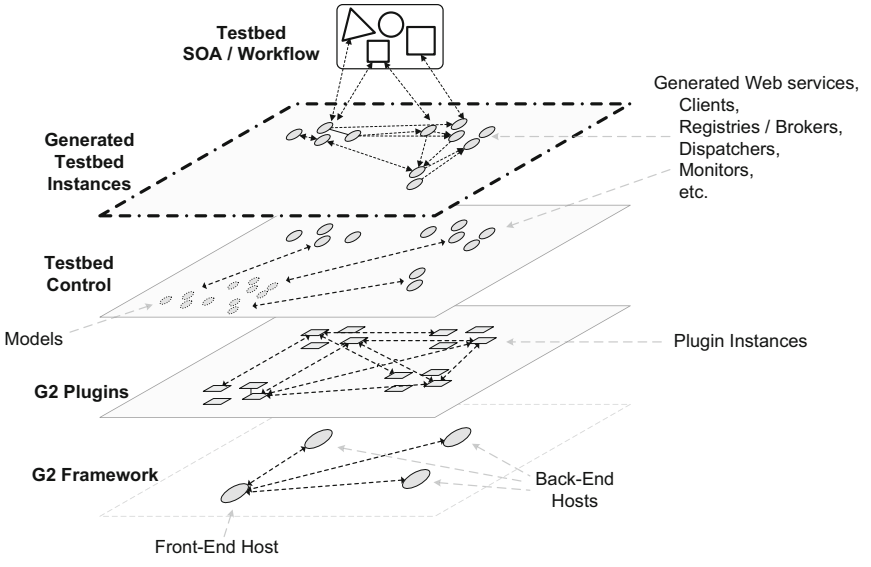

Fig. 1. Layered topology of a G2 testbed

to adapt the running testbed. For the sake of extensibility, G2 uses composable plugins which augment the testbed's functionality, making it possible to emulate diverse topologies, functional and non-functional properties, and behavior. Fig. 1 depicts a simplified view on the different layers of a G2-based testbed and the interactions within them. At the two bottom layers, G2 connects the front-end to the distributed back-end and installed plugins establish their communication structures. Most important are the two top layers. Based on the provided model schema, the tester creates models of SOA components which are then being generated and deployed at the back-end hosts. At the very top layer, the testbed instances are running and behave/interact according to the specification. The aggregation of these instances constitutes the actual testbed infrastructure on which the developed SOA can be evaluated.

In summary, at the front-end the tester specifies via Groovy scripts the testbed details, defining *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 following snippet contains a sample script for modeling a Web service, programming it's behavior (in this case just returning a String value), and deploying it at a back-end host. After deployment, it is possible to perform adaptations on-the-fly by changing the Web service's model which is immediately propagated to the generated instance at the back-end.

```
// import reference of cloud back-end host
def beHost1 = host.create("someHost:8080")

// import data type from XSD file
def dt = datatype.create("/path/to/types.xsd","vCard")
```

```
// create model of TestService with one operation
def service = webservice.build {

    TestService(binding: "doc,lit") {
        SayHi(card: vCard, result: String) {
            return "hi ${card.name}"
        }
    }
}[0]

service.deployAt(beHost1) // deployment

service.operations+= ... // on-the-fly adaptation of deployed service
```

4 The Cage Framework

4.1 Cage Methodology and Roles

The Cage methodology specifies how the framework can be used to generate large-scale customizable SOA testbeds and to deploy them on-demand in cloud-based infrastructures. The methodology comprises three steps (*modeling, setup, & execution*) that are performed by two different roles (*testbed engineer & tester*) during the establishment of Cage testbeds. Fig. 2 depicts a high-level overview of the three steps and the Cage tools supporting them.

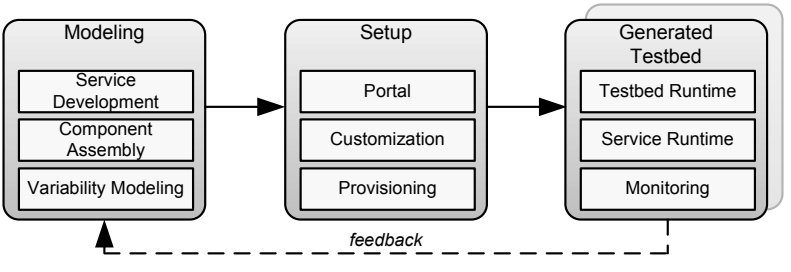

Fig. 2. Overall Cage approach and architecture

Modeling. At first, in the modeling step, the testbed engineer creates a specification/model of the testbed. The model defines the composition and topology of the testbed's components and, in addition, allows the testbed engineer to program the functional behavior. Due to the extensible nature of Genesis2, it is possible to model diverse types of testbed components, such as Web services, clients, registries, and message dispatchers, and to compose these into an emulated SOA consisting of mock-up components as well as optional own and third-party services integrated into the testbed. To be able to automatically deploy the whole testbed infrastructure, these components and their deployment relations are modeled in the component assembly modeling tool in which the testbed engineer defines the variability for the testbed, for example, different qualities of services and/or functional aspects.

Setup. Based on the created model, the testbed engineer uploads the corresponding artifacts to a portal. Testers have then the ability to request instances of the uploaded model via a control interface (part of the portal). A testbed model can be customized according to different aspects and requirements. For example, customization could be based on different perturbation and fault handling strategies, as show in [5]. Customization is accomplished by binding points of variability, e.g., by selecting one of multiple alternatives or by entering values for a point of variability.

Provisioning. The final step is to provision the testbed including the test services (emulated behavior), real services and middleware components. This is done automatically by the Cage framework once all variability has been bound by the tester.

The monitoring layer of the generated testbed captures various activities within the testbed environment such as service invocations and lookup requests (registry access). Low level logs are aggregated into metrics to analyze statistical variation of service behavior, as explained in [6]. The tester has the ability to analyze these metrics (using visualization tools) and to adjust models of variability accordingly. This cycle is depicted through the feedback arrow in Fig. 2.

4.2 Modeling Testbeds

Fig. 3 shows a component assembly example to illustrate various Cage concepts. Component assembly in Cage closely follows the Cafe approach [4] where modeling of composite application templates in Cafe is centered around components. Cage introduces a new component type to Cafe, namely the *Genesis Framework* component type. It represents a middleware on which certain other components can be deployed, namely those that have an implementation type of *Genesis Test Service*. This notion is similar to other component types in Cafe, such as the component types JBoss and Apache ODE shown in Fig. 3 which allow to deploy components of implementation type JEE Application and BPEL Process on them, respectively.

As a result, the testbed engineer can compose a testbed by combining middleware components including the Genesis Framework, application servers, Web servers and process engines. Also, components can be composed such as test services, real services, Web applications or business processes that run on the aforementioned middleware components. This enables engineers to systematically define complex testbed scenarios with the support of the Cage approach.

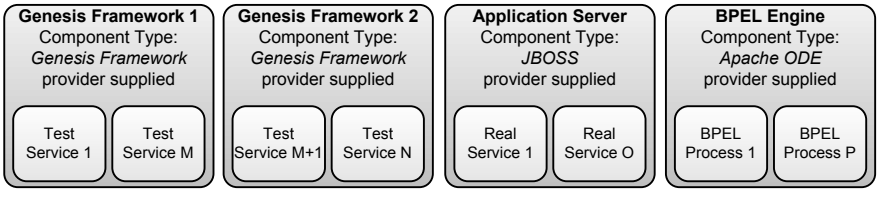

Fig. 3. Component assembly example

In Cafe, application templates are annotated with a *variability model* that is specified using the Cafe *variability meta-model* [4]. Such a variability model contains a set of *variability points* that specify possible configuration *alternatives* (see Fig. 4). Different types of alternatives exist that can be arbitrarily combined in one variability point:

- *Explicit alternatives* allow to specify a concrete value that can be selected, i.e. a fixed value for a delay.
- *Expression alternatives* allow to specify an expression (in XPath) that is evaluated and calculates the value that is entered at the variability point, for example, based on the values entered at another variability point.
- *Free alternatives* allow to prompt a user for an input, for example, to specify a specific failure rate.

Each variability point contains one or more *locators* that point into documents of the template that the variability point affects. Variability points can have complex *dependencies* that indicate that one or more variability points depend on one or more other variability points. These dependencies allow to specify temporal dependencies, i.e., a variability point can only be bound after all variability points that it depends on are bound. *Enabling conditions* can be defined for each variability point that specify under which condition which alternatives of this variability point can be chosen. This allows to specify complex dependencies such as "if the response-time of component A is greater than 2s the response-time of component B must be greater than 3s". In a Cage testbed a variability point can, for example point, into the WSDL document of a BPEL process to customize the endpoint of a test-service that should be invoked by that process. Thus, a variability point can indicate provisioning time that must be filled by the provisioning environment. Variability points can also express functional and

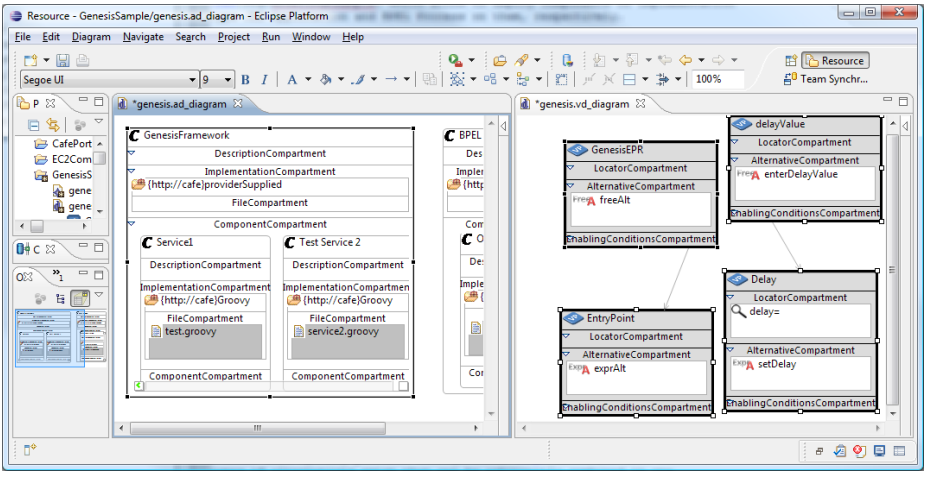

Fig. 4. Cage modeler: testbed components on the left, variability model on the right

non-functional variability. For example, a locator of a variability point can point into a Groovy script implementing a test service to configure its behavior or the average response-time this service should simulate.

4.3 Testbed Setup

Once a testbed, including its variability, has been modeled by the testbed engineer, it can be offered to testers for retrieval via the *test portal*. The tester is guided through the setup process via a *customization flow* which is a workflow generated from the variability model of a testbed, as introduced in [7]. The customization flow prompts the tester for all necessary decisions. In addition to the configuration of the testbed the tester specifies for how long the testbed is to be used. The duration is an important property as it allows to free the used resources for testing after a pre-defined amount of time. Once a tester has customized the testbed for the particular test to be performed (for example, the testbed is configured for load-testing instead of regression testing), the provisioning infrastructure sets up the necessary components and configures them as described in the next section.

4.4 Testbed Provisioning

After a testbed has been customized, it is being generated, deployed, and provided to the tester. This procedure comprises these steps:

- *Component provision & deployment*: Components available in the infrastructure are bootstrapped via their corresponding *provisioning services* [8]. For example Genesis2 back-end instances, workflow engines, and other base components are started, in order to deploy test services, real services and test clients on top.
- *Component configuration*: Components are configured by the provisioning infrastructure in order to establish links among then and to create a composite testbed. For example, a BPEL process that orchestrates a set of test-services must be configured with the endpoints of these test services as specified in the variability model for the respective testbed.

Testbed provisioning is done via a *provisioning flow* that is generated by the Cage framework from the model of the testbed. The provisioning flow respects the component dependencies introduced by the deployment relations (i.e. which component must be deployed on which other component) and the variability dependencies (i.e. which component must be configured with properties of which other component).

The following code snippet shows parts of a simplified G2 testbed template script which contains placeholders (uppercase strings) to be customized by Cafe at deployment time. Moreover, it returns a list of URLs of the deployed service endpoints, to be passed to other components. This provides a convenient method for the parameterized deployment of cloud-based testbeds.

```
// placeholder for references to cloud back-end host
def hostList = {{BACKENDHOSTS}}

def rand=new java.util.Random()
def randomHost = { -> hostList[rand.nextInt(hostList.size())] }
urlList=[]

serviceList.each { s->
    s.qos.responsetime={{QOS_DEF_RESPONSETIME}}
    s.qos.availability={{QOS_DEF_AVAILABILITY}}

    h=randomHost()
    urlList+=s.deployAt(h) // deployment at random host, collect URLs
}

return urlList
```

5 Related Work

Today, testing of complex service-based applications is a tedious task. Setting up of complex testbeds requires the acquisition and setup of computing, middleware and software resources which in most enterprises takes a considerable amount of time and effort to do.

To overcome this burden, several authors propose to use cloud resources that can be acquired on-demand, to deploy complex testbeds [9,10]. However, deploying the components to be tested on these cloud resources is still manual labor in these approaches and thus is time-consuming and error-prone. To automate the setup of complex component based applications in a reproducible way, automated provisioning environments [4,8,11] have been proposed. These environments first require the modeling of *component topologies* including the required components and their relations among each other. In this paper we investigate how this modeling affects the testing of complex service-based applications. In particular the modeling of the testbed components and their variability is similar to the *domain engineering phase* in software product line engineering in which a platform is developed that can then be customized into runnable applications in the *application engineering phase* [12,13]. Thus we adapt the notion of *testbed platform* as the basic testbed artifacts that can be customized into a concrete testbed. The application engineering phase in software product line engineering then corresponds to the customization and deployment of a testbed into a running testbed instance.

Several approaches have been developed which could be applied for testing adaptation mechanisms. SOABench [14] and PUPPET [15], for instance, support the creation of mock-up services in order to test workflows. However, these prototypes are restricted to emulating non-functional properties (QoS) and cannot be enhanced with programmable behavior. By using Genesis2 [3] which allows to extend testbeds with plugins we were able to implement a testbed which was flexible enough to test diverse adaptation mechanisms.

6 Conclusion and Future Work

In this paper we introduced the Cage framework to model, setup and automatically provision large-scale SOA testbeds in the cloud. We introduced the role of testbed engineers that model testbed families including their variability. Such a testbed family is then made available in a testbed portal for testers. Instead of manually configuring and deploying a large-scale SOA testbed, a test engineer can select the required testbed family from the portal, customize it using a generated customization wizard, and have it automatically deployed by the Cage infrastructure. Compared to existing approaches, the Cage approach facilitates testing, saves setup and configuration time, and enables testing in the cloud taking full advantage of pay-per-use models of, for example, IaaS providers.

In future work we will investigate how the results of individual test-runs can be used to automatically derive certain parameters of a testbed in order to iteratively optimize a given system to avoid cumbersome (low-level) configuration of the corresponding production system. Also, we plan to further implement metrics for testbed analytics and tools for visualizing complex behavior in cloud-based testbeds.

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperative Inf. Syst. 17(2), 223–255 (2008)
2. Huhns, M.N., Singh, M.P.: Service-oriented computing: Key concepts and principles. IEEE Internet Computing 9(1), 75–81 (2005)
3. Juszczyk, L., Dustdar, S.: Script-based generation of dynamic testbeds for soa. In: ICWS 2010, pp. 195–202. IEEE Computer Society, Los Alamitos (2010)
4. Mietzner, R., Unger, T., Leymann, F.: Cafe: A generic configurable customizable composite cloud application framework. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 357–364. Springer, Heidelberg (2009)
5. Juszczyk, L., Dustdar, S.: Programmable Fault Injection Testbeds for Complex SOA. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 411–425. Springer, Heidelberg (2010)
6. Psaier, H., Juszczyk, L., Skopik, F., Schall, D., Dustdar, S.: Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In: SASO, pp. 164–174. IEEE, Los Alamitos (2010)
7. Mietzner, R., Leymann, F.: Generation of bpel customization processes for saas applications from variability descriptors. In: IEEE SCC (2), pp. 359–366. IEEE Computer Society, Los Alamitos (2008)
8. Mietzner, R., Leymann, F.: Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. In: SERVICES I, pp. 3–10. IEEE Computer Society, Los Alamitos (2008)
9. Varia, J.: Cloud architectures. Amazon white paper (2008)
10. Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., Candea, G.: Cloud9: a software testing service. SIGOPS Oper. Syst. Rev. 43(4), 5–10 (2010)
11. Arnold, W., Eilam, T., Kalantar, M.H., Konstantinou, A.V., Totok, A.: Pattern based soa deployment. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 1–12. Springer, Heidelberg (2007)

12. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. Addison-Wesley Professional, Reading (2000)
13. Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer, Heidelberg (2005)
14. Bianculli, D., Binder, W., Drago, M.L.: Automated performance assessment for service-oriented middleware. Technical Report 2009/07, Faculty of Informatics - University of Lugano (November 2009)
15. Bertolino, A., De Angelis, G., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 266–282. Springer, Heidelberg (2008)