FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Techniques for Automated Generation of Testbed Infrastructures for SOA

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Łukasz Juszczyk
Registration Number 9925140

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. Dr. Schahram Dustdar

The dissertation has been reviewed by:

(Prof. Dr. Schahram Dustdar)

(Prof. Dr. Frank Leymann,
University of Stuttgart)

Wien, 27.09.2011

(Dipl.-Ing. Łukasz Juszczyk)

Zusammenfassung

Service-orientierte Architektur (SOA) ist ein Paradigma der Softwarearchitekturen, welches die Realisierung von verteilten Systemen aus modularen Diensten (Services) beschreibt. Dieses Paradigma hat in den letzten Jahren sowohl in der akademischen Forschung als auch in der Softwareentwichlungsindustrie stark an Bedeutung gewonnen, was man an der hohen Anzahl an wissenschaftlichen Arbeiten zu SOA und an SOA-basierten Softwareprodukten beobachten kann. Jedoch wurde sowohl in der Forschung als auch in der Entwicklung ein Thema vernachlässigt: das Testen von SOAs. Obwohl durchaus zahlreiche Konzepte und Lösungen zum Testen einzelner Services entwickelt wurden, gibt es kaum Arbeiten die das Testen von komplexen SOA-Systemen behandeln. In Besonderen gibt es einen Mangel an Konzepten zum Testen von Systemen, welche später in einem Umfeld von externen Services betrieben werden, dieses Umfeld aber zur Entwicklungszeit nicht verfügbar ist. Um solche Systeme trotzdem testen zu können bedarf es eines "Testbeds", also eines zu Testzwecken generierten jedoch realistischen Abbildes vom Umfeld.

In dieser Dissertation behandeln wir das *Automatische Generieren von Testbed-Infrastrukturen für SOA*. Das Ziel unserer Arbeit war es, Lösungen zu erforschen die SOA-Entwickler beim Generieren von Testbeds unterstützen. Die präsentierten Resultate beinhalten: erweiterbares Modellieren von SOA-Umgebungen, eine Skriptsprache zum Spezifizieren der Modelle, Mechanismen zum Generieren von Testbed-Instanzen aus den Modellen sowie Ansätze zum Automatisieren des gesamten Prozesses. Wir präsentieren die entwickelten Konzepte und Techniken im Detail, und erklären wie diese eingesetzt und erweitert werden können um SOA-Testbeds zu generieren. Wir haben die Konzepte in einem Prototypen namens GENESIS implementiert und diesen für das Testen und Evaluieren diverser SOA-basierten Systeme eingesetzt. Darüber hinaus haben wir GENESIS als Open-Source-Software veröffentlicht, als Beweis für die Korrektheit und Sinnhaftigkeit unserer Lösungen sowie als Beitrag an die SOA-Forschungsgemeinschaft.

Abstract

Service-oriented architecture (SOA), as a paradigm for flexible distributed computing based on modular services, has become a major topic in computer science and in software industry. As a result, numerous SOA-based concepts have been published by the research community and many software solutions are nowadays provided by the industry. However, both, the research community and the industry, have neglected the need for sophisticated support for testing of SOA's. While most of the effort has been put into supporting the testing of single Web services, only very few works actually aim at testing of complex SOA-based systems. In particular, there is a lack of solutions for testing of systems that will be deployed in service-based environments and that need proper testbeds at development time.

In this thesis we focus on solving this issue and present our results on *automated generation of testbed infrastructures for SOA*. The purpose of our work is to provide means for developers to generate customized testbeds that emulate missing SOA environments in order to have an infrastructure for testing the developed system at runtime. We have developed techniques for modeling SOA environments in an extensible manner, a scripting language for specifying the models, mechanisms for generating running testbeds from these models, and techniques for automating the whole process to a certain degree. We have implemented a prototype framework, called GENESIS, and applied it for testing and evaluation of SOA-based systems. We present the developed concepts and techniques in detail, explain how they have evolved, how they can be extended for implementing custom testbed features, and, eventually, how the framework is applied in practice. In addition, we provide all developed software prototypes as open-source for proving the correctness of the developed techniques, plus, as a contribution to the research community.

Acknowledgements

First of all, I would like to thank my advisor Prof. Schahram Dustdar for the opportunity to carry out my PhD studies at the Distributed Systems Group (DSG). I am very grateful for the liberty to choose a research field and to develop my own ideas within such a great environment as the DSG. Especially, I am much obliged for the invaluable support Prof. Dustdar gave me during the hard days of my studies and for the mentoring during all phases of my research.

Additionally, I would also like to thank Prof. Frank Leymann for being my second advisor and examiner, and for his valuable comments and suggestions that helped me to improve this thesis.

I would also like to thank my colleagues at the DSG, with whom I had a lot of fruitful discussions and who gave feedback on my ideas and this way contributed to this thesis. In particular I would like to acknowledge Christoph Dorn, Karl M. Göschka, Roman Khazankin, Philipp Leitner, Atif Manzoor, Anton Michlmayr, Harald Psaier, Daniel Schall, Florian Skopik, and Martin Treiber for their input. It was a great pleasure to work with you!

Moreover, I want to thank our secretaries Christine Kamper, Margret Steinbuch, and Renate Weiss for unburdening us of all the management tasks and for keeping the DSG running smoothly.

Most importantly, my sincere thanks are given to my parents Urszula and Krzysztof who have always supported me, to my beloved partner Veronika for her love and understanding, and to my daughter Magdalena for making me the proudest father in the world. This thesis is dedicated to you!

This work was supported by the European Union projects WORKPAD (grant no. 034749) and S-CUBE (grant no. 215483).

Łukasz Juszczyk
September, 2011
Vienna, Austria

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The paradigm of service-oriented computing (SOC) and service-oriented architecture (SOA) has appeared for the first time in the year 1996 [144, 145] and has gained high momentum since then. Being based on established principles of software engineering, such as modularization and distributed computing, SOA has evolved into a major topic in computer science [135] and has been increasingly applied for realizing flexible distributed systems. In general, the impact of SOA has advanced into a multitude of research domains, such as autonomic computing, business process management, and cloud computing, which provides evidence about its acceptable in both, research and industry. However, typical for a novel technology, SOA is still suffering from problems which have not been solved yet to a satisfactory level and which do hamper its wider acceptance. As outlined in literature [87–89], one of the major problem is testing, in particular the lack of support for testing of complex SOA systems. The research work in the scope of this dissertation has been focused on investigating this problem and bringing forward a solution for it.

1.1 Motivation

Comparing the state of the art of research on SOA in general and the research on testing in/for SOA, an interesting divergence becomes evident. SOA itself has had an impressive evolution in the last years. At its beginning, Web service-based SOA had been mistaken as yet another implementation for distributed objects and remote procedure call (RPC) and, therefore, had been abused for direct and tightly-coupled communication [162]. After clearing up these misconceptions and pointing out its benefits derived from decoupling, SOA has been accepted as an architectural style for realizing flexible document-oriented distributed computing. Today's SOAs comprise much more than just services, clients and brokers (as depicted in the outdated Web service triangle [104, 124]) but comprise also message mediators, service buses, monitors, management and governance systems, workflow engines, and many other component types [135].

As a consequence, SOA is becoming increasingly powerful but also increasingly complex. This implies higher error-proneness [75] and, logically, requires thorough testing. But looking at available solutions for SOA testing (research prototypes as well as commercial products), one might get the feeling that SOA is still reduced to its find-bind-invoke interactions because most approaches deal only with testing of individual Web services, and only few solutions deal to some extent with complex SOAs [79, 80, 84, 85, 137]. All in all, it is possible to test whether a single Web service behaves correctly regarding its functional and non-functional properties, but testing systems operating on a whole environment of services is currently not supported at a satisfactory level.

Let us take the case of an autonomic workflow engine, such as the one presented in [161], for example. The engine must monitor available services, analyze their status, and decide whether to adapt running workflow instances. To verify the engine's correct execution it is necessary to perform runtime tests in a real(-istic) service environment, in short, a service testbed. The testbed must comprise running instances of all participants (in this simple case only Web services), emulate realistic behavior (e.g., quality of service, dependencies among services, faults), and serve as an infrastructure on which the developed workflow engine can be tested. But how can engineers be provided with such testbeds? Unfortunately, so far they had to create them manually, as no proper support had been available.

The second motivating scenario is tightly bound to one of SOA's most prominent advantages: faster software development due to reuse of services. Building systems as compositions of modular services does not only accelerate the development process, but the ability to exchange services with equivalent ones at runtime offers a new degree of adaptivity. As a consequence, it is possible to outsource tasks to external partners/companies, by using their services and incorporating their functionality into a system or workflow. However, in spite of the gained flexibility, outsourcing bears a significant risk as it implies dependencies on remote services which can become unavailable at any time. Some providers offer service level agreements (SLA) that specify minimal performance metrics and, penalties, if these are not provided. SLAs do help to make a SOA system more predictable, as service providers have a strong incentive to guarantee a promised quality in order to avoid penalties. But still the dependency on external Web services remains a risk as SLAs are sometimes missed and an failing service can have critical effects on the system that depends on it, unless it is able to handle these properly. And here anew appears the problem of testing: how can engineers develop and evaluate techniques for handling faults of remote components, if they don't have full access to these. Obviously, external partners do not allow volatile invocations of their services, as this would put additional load on these and degrade their performance. Furthermore, external services do often cost money and, therefore, each test run would cost money. Again this problem could be solved by applying testbeds which emulate the external partner services, simulate faulty behavior, and facilitate a thorough evaluation of the system's fault handling routines. In a nutshell, SOA's ability to integrate easily external services comes, unfortunately, at the cost of complicating the testing process.

Scenarios like these point up the need for testbed infrastructures. Despite that, the SOA research community has not put sufficient effort into solving this problem. Even though some groups have investigated the generation of testbeds for SOA, their work was usually focused on a particular problem and did not provide generic support for creating testbeds of customizable

structure and behavior. We regard this as a serious drawback which hampers the testing process and, consequently, slows down the whole software development process. This issue has been the main motivation for doing research on supporting the generation of testbeds in order to accelerate and improve the process of software engineering for SOA.

1.2 Research Challenges & Achievements

The research questions and challenges handled in the scope of this dissertation are twofold. They comprise *research challenges*, in terms of developing concepts and methodologies, as well as *engineering challenges* for developing a prototype implementation as a proof of concept.

The research challenges are as follows:

- **To analyze the state of the art of testing solutions for SOA systems, in order to discover shortcomings and the main burdens of software engineers.**
  - What support do available open-source and industrial/commercial solutions provide?
  - How far is academic research advanced and which concepts and prototype implementations have been developed? What is still missing?
- **To investigate the progress of testbed generation for SOA.**
  - Which solutions have been developed by industry and academia?
  - How advanced are these and what are the limitations?
  - How well do these solutions cover the actual problems of SOA engineers?
  - Which problems have not been addressed yet?
- **To come up with techniques and methodologies for modelling testbeds.**
  - How can engineers be supported in specifying structure and behavior of testbeds?
  - What is the right compromise between simple usage and rich functionality?
  - How to achieve good extensibility and customizability?
- **To develop techniques for generating running testbed instances.**
  - How can testbed models be tranformed into real testbed infrastructures?
  - How can testbeds expose customizable functional- and non-functional properties?
- **To automate the establishment of testbed infrastructures.**
  - How much of the testbed specification process can be automated?
  - How can we reduce necessary input from engineers and accelerate the specification?

The engineering challenges comprise:

- **Implement the developed concepts in a prototype framework.**
- **Apply the framework in various testing scenarios, in order to assess its benefits and discover limitations.**

1.2.1 Key Contributions

In this dissertation we have made progress in the field of testbed generation for SOA. We have developed novel techniques for solving the previously listed challenges and implemented a software prototype to prove the applicability of our concepts.

The most significant contributions and achievements are:

- **A simple yet powerful technique for specifying testbeds**, that is based on a scripting language and seems to be intuitive for engineers.
- **An open model for SOA testbed**s, providing extendability.
- **Techniques for generating running testbeds instances from the models** and for controlling them remotely at runtime.
- **A new level of customizability and programmability of the testbeds**, derived from the scripting approach.
- **Techniques for performing on-the-fly adaptations to the testbed**, in order to adjust test runs at runtime.

Moreover, we have published the software prototype as open-source, as a contribution to the research community.

1.2.2 Research out of this Thesis' Scope

As outlined in the previous section, this thesis covers only research on generation of testbed infrastructures for SOA. It does clearly not cover any research on supporting the execution of test runs nor does it deal with the evaluation of test results. Due to the intricacy and complexity of testbed generation we decided to focus on this topic in order to push the research on it. This means that instead of covering the whole breadth of testing in SOA software development, we preferred to advance as deep as possible on testbed generation with the aim of being able to present convincing solutions to the inherent challenges. Without doubt, test execution and test result evaluation are complex problems as well and deserve to be investigated intensively, however, this has not been done in the scope of this thesis.

1.2.3 Evaluation of our Contribution

The presentation of novel concepts always requires an evaluation in order to prove their applicability, usefulness, and correctness. Depending on the type of concepts, different types of evaluation make sense to be applied. For this thesis, however, the evaluation was not trivial.

We have not performed a *comparative evaluation*, by matching our approach to other available ones, in order to prove our improvements and the superiority of our concepts. This is mainly due to the novelty of our work and the lack of direct competitors.

Also we have not done a precise *performance evaluation*, as this thesis' contribution is not about performance issues nor does it prove the quality of our approach in any case. It would merely assess the quality of our prototype implementation, which is, however, not of primary importance being a proof of concept.

Without doubt, a *real-world evaluation*, where our concepts and prototypes are applied in real SOA development projects would make most sense and give valuable insights into how much the development process got improved by our contribution. Unfortunately, this was not possible as a) we did not have access to a significant number of SOA projects and b) it would have been not easy to convince the developers to apply our prototype implementation.

Instead, we evaluated our concepts in a selective manner, choosing what we regarded as reasonable and realizable. For instance, we included a performance evaluation for a technique which deals with generating testbeds *at runtime* and where the delays do have an effect on the execution of the tested systems. In contrast to that, we avoided to evaluate the performance of testbed generation which happens *before* the test runs and, therefore, does only have an effect on the patience of the testers but not on the test results.

Moreover, we applied our approach at our research group in several projects for an internal assessment and as a proof of usability of the prototype implementation. The projects published in [105,112,123,138,139,143,146–148] all used G2-based testbeds, which we regard as a strong indicator for the applicability of our results.

1.3 Structure of the Thesis

This structure of this dissertation is strongly oriented on the papers published during this thesis' research work (listed in next section). Basically, the structure is split into four main parts:

1. An *introductory part* providing a review of the state of the art on testing of SOAs. In Chapter 2 we explain the problem that has been solved in this thesis. We analyze the *state of the art* of SOA software development, the progress on testing solutions for SOA, outline unsolved problems, and specify the *problem statement* for this thesis. Furthermore, we review *related work* and compare it our research, in order to outline the contribution done in the scope of this thesis.

2. The main part presenting this *thesis' major contribution*, the GENESIS testbed generator, is structured into four Chapters (3 - 6) which relate to the four essential publications about GENESIS and which represent the evolution of our concepts.

- In Chapter 3 we present our first approach on testbed generation, the *GENESIS* framework.
- Chapter 4 comprises the next step in the *evolution* of the initial concepts, resulting in the successor framework *GENESIS 2*.
- In Chapter 5 we demonstrate the extendability of GENESIS 2 and present our techniques for generating *fault injection testbeds* for SOA.
- In Chapter 6 we evolve testbed generation towards more *automation* and describe an approach for intercepting Web service invocations in a running SOA system and for *generating testbeds on-the-fly* for these.

3. In addition to the chapters explaining the concepts of GENESIS, we do also present two works in which GENESIS was applied as a testbed, plus, as a runtime environment for adaptive Web services. These comprise the following parts.
   - In Chapter 7 we describe an approach for generating *large-scale SOA testbeds* in a *cloud* infrastructure.
   - In Chapter 8 we explain how the *programming model* of GENESIS can be used for *engineering evolvable Web services*.
4. Finally, in Chapter 9 we close this thesis with an outlook and an overview of possible *future research* and conclude with a *summary of the contributions* of our research work.

1.4 Publications

The results of our research work have been published in *five conference papers* and *four workshop papers* that provide the content for this thesis.

1. **Juszczyk L.**, Truong H.-L., Dustdar S.

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

   13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08), 31. March - 4. April 2008, Belfast, Northern Ireland.

2. Treiber M., **Juszczyk L.**, Schall D., Dustdar S.

   Programming Evolveable Web Services.

   2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS). 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), 2. - 8. May 2010, Cape Town, South Africa.

3. **Juszczyk L.**, Dustdar S.

   Script-based Generation of Dynamic Testbeds for SOA.

   8th IEEE International Conference on Web Services (ICWS'10), 5. - 10. July 2010, Miami, USA.

4. **Juszczyk L.**, Dustdar S.

   Testbeds for Emulating Dependability Issues of Mobile Web Services.

   1st International Workshop on Engineering Mobile Service Oriented Systems (EMSOS). 6th IEEE World Congress on Services (SERVICES'10), 5. - 10. July 2010, Miami, USA.

5. Psaier H., **Juszczyk L.**, Skopik F., Schall D., Dustdar S.

   Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems.

   4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10), 27. September - 01. October 2010, Budapest, Hungary.

6. Psaier H., Skopik F., Schall D., **Juszczyk L.**, Treiber M., Dustdar S.

   A Programming Model for Self-Adaptive Open Enterprise Systems.

   5th Workshop on Middleware for Service Oriented Computing (MW4SOC). 11th ACM/IFIP/USENIX Middleware Conference, 29. November - 3. December 2010, Bangalore, India.

7. **Juszczyk L.**, Dustdar S.

   Programmable Fault Injection Testbeds for Complex SOA.

   8th International Conference on Service-Oriented Computing (ICSOC'10), 07. - 10. December 2010, San Francisco, USA.

8. **Juszczyk L.**, Schall D., Mietzner R., Dustdar S., Leymann F.

   CAGE: Customizable Large-scale SOA Testbeds in the Cloud.

   6th International Workshop on Engineering Service-Oriented Applications (WESOA). 8th International Conference on Service-Oriented Computing (ICSOC'10), 07. - 10. December 2010, San Francisco, USA.

9. **Juszczyk L.**, Dustdar S.

   Automating the Generation of Web Service Testbeds using AOP

   9th European Conference on Web Services (ECOWS'11), 12. - 16. September 2011, Lugano, Switzerland.

1.5 Published Prototypes

In addition to our scientific publications, we contributed to the research community by publishing our software prototypes as open-source, available at:

`http://www.infosys.tuwien.ac.at/prototype/Genesis/`

CHAPTER 2

Testing of Service-oriented Architectures - State of the Art

2.1 Principles of Service-oriented Archtitecture

The principles of service-oriented computing (SOC), and the related architectural style called service-oriented architecture (SOA), as derived from several formerly existing concepts, such as component-based systems, distributed computing, and modularization. Even though, some researchers see SOA as a revolution [93], basically due to its impact, we rather refer to it as an evolution of established concepts, technologies, and methodologies. In a simplified manner, SOA propagates the usage of fine grained services that expose specific functionality and the composition of these into a service-oriented system. Compared to monolithic approaches, SOA systems are more flexible as services can be replaced or new ones can be added into a running system and, this way, provide a good grounding for adaptive systems which change their behavior, configuration, composition at runtime.

Thomas Erl defines in his book "Service-oriented Architecture: Concepts, Technology, and Design" [96], as well as on his Web sites about SOA engineering [45], the following key principles of service-orientation:

- *Standardized Service Contract*: The contract must specify the service's functionality and the interface, via which it can be accessed, in an expressive manner. It must define exchanged data types, usage policies, and any other information that is important for understanding the service's purpose and for invoking it.

- *Service Loose coupling*: Reduces the level of dependency between a service consumer, the service's implementation, and its contract.

- *Service Abstraction*: Related to loose coupling, service abstraction propagates to hide as many details of the service as possible and to publish (within the contract) only what is essential for understanding its functionality.
- *Service Reusability*: This is one of the key principles and one of the most prominent features of service-orientation. Services encapsulate logic that can be easily reused by other components in order to establish (more) complex functionality.
- *Service Autonomy*: Services must have maximum control over the environment and resources they depend on. By reducing dependencies, services can maintain a higher reliability and predictability.
- *Service Statelessness*: In order to increase a service's scalability and reliability it is recommended to keep it stateless, as the management of a state implies additional complexity. Stateless services can be replicated easily, for instance for the sake of load balancing, which is an asset for scalability.
- *Service Discoverability*: To guarantee reusability it is essential that a service is discoverable. This requires to publish the service, e.g., at a well-known registry, with sufficient metadata so it can be easily identified and its purpose can be understood by potential consumers.
- *Service Composability*: The composition of complex functionality out of a set of is a major feature of SOA. Services are expected to participate in such compositions, which is a feature that is closely related to the previously presented principles.

These are the commonly accepted major principles of SOA, even though some authors/groups enhanced this set with additional ones [165], such as *Service Optimization*, *Service Relevance*, *Service Encapsulation*, or *Service Location Transparency*. All in all, SOA adapts a wide number of well-known and accepted principles of distributed computing and the principles can be safely summarized into "building flexible systems composed of modular components referred to as services". In the next section we take a closer look at how these systems are being built and engineered, and which shortcomings pose burdens to SOA software engineers.

2.2 Engineering SOA Systems - Gains and Problems

2.2.1 Choosing a Communication Standard

As outlined, SOA is rather a set of principles and concepts than a concrete technology. It is not even a single standard. Depending on the requirements, a SOA system can be implemented using diverse technologies or communication standards as long as the essential concepts are followed. However, not all technologies and standards are equally suited to this purpose and, therefore, some have gained particular importance and popularity among the SOA community, while others have not got that much of acceptance. SOA systems can be built, for instance,

by using the Common Object Request Broker Architecture (CORBA) (e.g., [12]), Java Remote Method Invocation (RMI) (e.g., [13]), and various other standards, even proprietary ones.

However, two communication standards have gained paramount acceptance for realizing SOA system: SOAP and Representational State Transfer (REST). Though as explained in [136], SOAP and REST differ significantly and both have their strengths and weaknesses.

- *SOAP* (formerly an acronym for Simple Object Access Protocol) [47] is basically a standard that specifies the structure of XML-based [19] messages that are exchanged among distributed components. SOAP messages contain an envelope that comprises an arbitrary number of header elements and a single body element (see Listing 2.1). The header is a major advantage of SOAP-based communication, as it allows to implement arbitrary extensions (e.g., for signing message content, routing/addressing of messages, additional meta data, etc.), as it is done in the numerous WS-* specifications [166].

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">

<soap:Header>
  <ns1:header1 xmlns:ns1="http://sampleheaderNS">
    <!-- ... -->
  </ns1:header1>

  <ns2:header2 xmlns:ns1="http://secondheaderNS">
    <!-- ... -->
  </ns2:header2>

  <!-- ... -->
</soap:Header>

<soap:Body>
  <msg:Response xmlns:msg="http://sampleServiceNS">
    <!-- ... -->
  </msg:Response>
</soap:Body>

</soap:Envelope>
```

Listing 2.1: SOAP Message Structure

  SOAP Web services are described using the Web Service Description Language (WSDL) [56] which defines the service's interface, exchanges message types specified in XML Schema Definition (XSD) [64], meta-data for more detailed specification of the service's functionality, its endpoint address, as well as binding information which defines the used transport method.

  SOAP is transport agnostic, which means that it does not put any restrictions on how the messages are being transferred among SOA components. Possible transport mechanisms are HTTP(S), SMTP, message queuing protocols (e.g., Java Message Service [35]), or other middleware implementations which support message exchange.

- *REST-ful services* follow a different approach than SOAP-based ones. While SOAP is centered around the exchange of messaged/documents, REST-ful services mainly represent

resources which are accessible via the the HTTP protocol [28]. These resources can be static ones, e.g., simple files, as well as dynamically computed ones, similar to SOAP-like responses. Being tightly bound to HTTP, resources can be manipulated via the operations `PUT`, `GET`, `POST`, and `DELETE` and can be represented in a variety of formats, e.g., XML, PDF, or even graphics.

Compared to SOAP, REST seems more restricted as it does not support any transport mechanism apart from HTTP and does not provide anything comparable to WS-*. However, this makes REST simpler and results in a higher acceptance in the Web community, where simplicity is of paramount importance. This trend becomes especially evident as today most of the Web API's are based on REST.

Unfortunately, REST-ful services do not have any description language comparable to WSDL, which hampers an automated discovery and invocation of the services. There exists the Web Application Description Language (WADL) [52] that is supposed to fulfill this purpose but its usability and sense is highly controversial in the community [107].

Both service standards, SOAP and REST, have their advantages and disadvantages and, there is no simple answer for which one should be preferred. Depending on the requirements of the developed system, several design decisions must be made, which will have to be taken into account when a service standard is chosen. Pautasso et al. present in [136] a detailed comparison of both standards, guide through their concepts, pros & cons, and help to make the right decision when engineering a SOA system. Though, one can observe that SOAP services have gained higher importance for enterprise computing, as they are extensible and provide better support for building complex systems, while REST-based convinced the Web community with their simplicity.

Nevertheless, in this thesis we have not covered both standards, but have concentrated our research only on SOAP-based SOA. We regard it as a more interesting research topic, as engineering SOAP-based systems is on the one hand supported by a multitude of frameworks, tools, and methodologies, but on the other hand several issues, that burden software engineers, have still not been solved.

2.2.2 Engineering SOAP-based Systems

Following the SOA principles, which demand to design systems as a flexible collection of modular and exchangeable services, almost no restrictions are posed on the possible topology and complexity of an SOA system. In fact, SOA provides does benefit engineering of complex and large-scale distributed systems [104]. In its most simple and primitive form, an SOA comprises a set of services, a set of clients, and at least one broker. This is well known as the *publish-find-bind* SOA triangle (depicted in Figure 2.1). Available Web services are *published* at the broker(s), clients use the broker(s) to *find* services, and, eventually *bind* to these in order to invoke them.

Though, this scenario is not up to date any more as SOA comprises today a wide variety of components which provide sophisticated functionality and allow to establish complex systems.

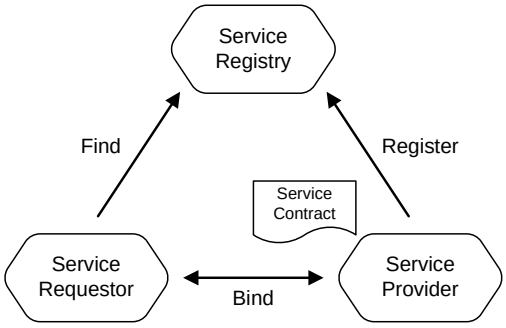


Figure 2.1: Publish, Find, & Bind - also known as the SOA triangle (from Michlmayr et al. [124])

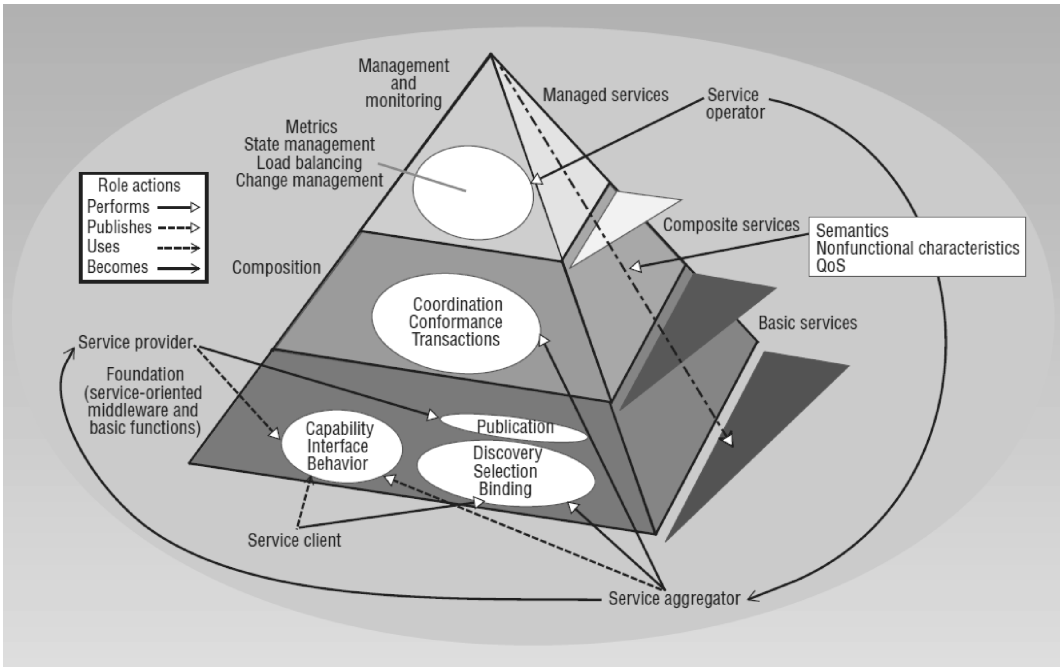


Figure 2.2: Research roadmap showing diverse SOA components (from Papazoglou et al. [134])

Figure 2.2 shows how Papazoglou et al. observe the current progress of SOA and where they expect future work to be focused at. At the bottom of the pyramid they place basic services which constitute the building blocks of an SOA system. Basically, this covers the *publish-find-bind* triangle. On top of the basis, composite services are put together out of the basic services, enhanced with techniques for coordination, transaction processing, etc. On the very top of the pyramid, management services are governing the SOA, performing load balancing, state management, change management, monitoring, etc. This all demonstrates that the primitive triangle,

consisting of services, clients and brokers, has been enhanced with higher-level components such as:

- *Workflow engines* which establish composite services by executing business processes in the background and, in turn, reuse other services. In this domain the Business Process Execution Language (BPEL) [17] is dominating due to its focus on Web services and has been established as a de-facto standard in industry. There exist many BPEL engines, open-source ones such as Apache ODE [9] as well as commercial ones like IBM Websphere Process Service [30] and Oracle BPEL Process Manager [43]. Apart from BPEL, there are other competing standards such as YAWL [160] or very light-weight approaches like BeanFlow [11].

  In addition to the common workflow engines, which execute preconfigured business processes precisely according to their specification, there exist engines, e.g., [161], which monitor their environment and optimize the execution of the workflow instances at runtime.

- *Choreography engines* are similar to workflow engines, but describe message flows not from a centralized perspective, but rather define valid flow patterns which have to be followed by the participating components. Up to date, the WS-Choreography Description Language (WS-CDL) [54] has received the status of a W3C recommendation but, unfortunately, no implementations are available so far.

- *Enterprise Service Buses* (ESB) provide a transport mechanism for passing messages (e.g., SOAP) among the distributed components of an SOA. Compared to direct invocations, e.g., via HTTP, where clients discover Web services and send their request messages directly, ESB's provide a bus infrastructure at which Web services are registered and client messages are dispatched/routed automatically among these. Furthermore, ESB's can improve the dependability of an SOA by providing mechanisms for reliable message transport. Engineers can choose among a large variety of ESB implementations, e.g., Apache Service Mix [10] or the JBoss ESB [37].

- *Monitoring and Governance Systems* keep track of the situation inside an SOA and the status of its components, and execute corresponding actions on detected misbehaviors. The Web Services Distributed Management standard (WSDM) [57] specifies the corresponding protocols and implementations, such as Apache Muse [8], exist as well. Moreover, runtime environments, such as VRESCO [51], support engineers in using monitoring results for implementing adaptive service-based systems.

- *Transaction Frameworks* bring transaction management, as known from database systems, to Web services. Supported by the WS-Transaction [62] standard, that consists of WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity, however, implementations exist only in large commercial solutions like IBM Websphere [29].

- The *Web Services Resource Framework* (WSRF) provides a clean way to introduce statefulness to Web services which are supposed to be stateless by default. Web services can

access a state from, so called, resource services and client pass resource identifiers with each invocation. It is implemented, for instance, in Apache Muse, IBM Websphere, or the Globus Toolkit for grid computing [22].

- *Publish/Subscribe Frameworks* for event-driven programming allow components to register to event notifications by specifying topics of interest. Notifications are sent out via SOAP messages to the requesters callback service interface. As it is tightly coupled with the WSRF, the same frameworks provide an implementation for WS-Notification [61] standard.
- Moreover, an SOA can incorporate legacy systems, which are accessible via a Web services interface that wraps the original software, and even human workers who expose their expertise via Web services by following standards such as BPEL4People [63] or WS-HumanTask [59].

To sum up the state of the art, SOA systems are not limited to services, clients, and registries only but can comprise a variety of components, can grow to large-scale, and can implement arbitrary complexity. Engineers can use diverse tools and software solutions, commercial and open-source ones, in order to accelerate the development of SOA systems. However, there are several issues which (unnecessarily) complicate the development process and which slow down the acceptance of SOA.

One major issue is the promised interoperability of SOAP-based Web services, which is, unfortunately, not fulfilled. SOAP is an open standard, is based on other open standards (XML, XSD, etc.), and most often is realized by using an open transport mechanism as well (JMS, HTTP, etc.). Consequently, SOAP is propagated to be the interoperable glue between components in heterogeneous environments. The problem is, however, that on top of open standards one can still implement non-interoperable behavior and this happened with SOAP: SOAP messages can be formatted in different binding styles and uses. Possible styles are *Remote Procedure Call (RPC)* and *Document (DOC)*, and these can be combined with a *literal* oder *encoded* use. Which means that four possible combinations are possible: RPC/enc, RPC/lit, DOC/enc, and DOC/lit. Though, while theoretically all four are usable, only DOC/lit is considered as recommendable [142] and the rest has been declared as deprecated as it is not compatible with the Web Services Interoperabilty (WS-I) [60] initiative. In order to push this initiative and not encourage SOA developers to use the deprecated styles, many modern Web service frameworks, such as Apache CXF [4] dropped the support for non-WS-I-compatible styles and these services cannot be accessed with these frameworks any more. The problem is that still many public available Web services use the old deprecated styles and, therefore, have become incompatible! For example, in the QWS dataset [68, 69], that is a collection of WSDL documents of public Web services (collected in 2007), we noticed that 31% of the WSDL's were not WS-I compatible. The conclusion is that SOAP is not as interoperable as promised and developers must be aware of this trap.

The second well-known issue is the interaction style. Web services can be invoked via RPC, which makes them yet another distributed object technology [86, 162], or via exchanging messages/documents. Again, both styles are possible, though that does not mean that both

are equally recommendable and suited for implementing distributed communication. RPC is simply an extension of traditional proceduce/method calls in order to access remote objects. Requests, which means information about which procedure is called on which object with which arguments, as well as responses, are wrapped into SOAP messages and are transferred between client and Web service. The main problem is that this communication style is synchronous, where clients wait in a blocking manner until the service has responded and keep the connection alive while the call is being processed. This does not only make the call vulnerable to communication faults, as a broken connection causes the call to be aborted, but it also reduces the whole system's scalability due to the delays caused. For this reason, we agree with [124] that RPC-based interactions should be avoided. In contrast to this, message-oriented SOC relies on asynchronously exchanged documents which are sent and routed among the components of an SOA. Decoupling of components by keeping the communication asynchronous does benefit the whole systems agility and scalability as it allows flexible dispatching of requests (keeping messages in queues, performing load balancing, etc.) and does not require the client to block its execution until the response arrives but to be notified via a callback once it is available. Consequently, RPC-based SOA is more an abuse of the principles of service-orientation than a proper interaction style. SOA can only provide its promised features of agility if the coupling among components is a low as possible, and this is only possible with message-oriented communication.

Another important, yet sometimes problematic, feature of SOA is late binding, also referred to as dynamic binding. Instead of having Web service endpoints hard-coded into the client's logic, these are being chosen dynamically at the time of the invocation. The effect is that systems can be much more adaptive, in terms of accepting and incorporating newly available services at runtime and switching over transparently to more capable services, e.g., if the old ones perform worse of became unavailable. Frameworks like VRESCO [51] aim at optimizing this feature, monitor available services, and recommend pro-actively to clients to switch over to alternatives. While late binding does in general decrease the level of coupling among components, it introduces new challenges at runtime. Dynamic links complicate the whole system's execution, making it more less predictable and error-prone. If service links are established at runtime and the set of available services is not constant (new ones appear, old ones become unavailable or are subject to failures) this poses a risk, especially if the services are essential for the systems functionality. Furthermore, clients or any other components, such as ESB's, which must dispatch a request must provide some logic which decides where to dispatch and how to handle (unexpected) failures. This additional complexity is the drawback of late/dynamic binding.

In general, complexity is one of the most criticized issues for SOAP-based Web services. SOAP's ability to be extensible caused the development of numerous standards, specifications, and implementations that enhance SOC with various aspects, e.g., message encryption or reliable message transfer. These are usually named "Web Services {SpecName}" or "WS-{SpecName}" and are, therefore, referred to as *WS-**. A recent listing of those standards can be found at Wikipedia [166] and other Web sites about SOA [106, 150]. However, WS-* is not regarded by everyone as an asset of SOAP, but some see it rather as an disincentive as it introduces a high level of complexity, both for the engineering process as well as for the runtime of an SOA system. WS-* is essential for enterprise computing, where security and reliability are of paramount importance and the additional complexity is acceptable. Here, one does not have an

alternative option to using SOAP. Though, in the web community SOAP has been more and more replaced by REST-ful services, in order to keep the service interactions simple. For instance, Google has shut down the SOAP version of their API and since then only supports REST in order to promote its application [23]. Also the Ruby of Rails framework has dropped the official SOAP support on purpose for the same reason [110] (unofficial libraries are still available). This emphasizes the trend to apply REST on the Web, while SOAP remains the de-facto standard in enterprise computing.

To sum up the state-of-the art of engineering SOAP-based systems, it is safe to say that, in spite of its drawbacks related to complexity, SOAP has gained high popularity in industry and has made impressive progress in academic research. SOAP support is provided for almost all programming languages and platforms, and engineers can benefit from a large number of available Web service frameworks and software solutions. Numerous open-source projects deliver essential SOA components, such as workflow engines and service buses, and large companies, e.g., IBM and Oracle, offer sophisticated software suites for developing SOA systems. All in all, SOA engineers can benefit from a large variety of software solutions in order to accelerate the development process. However, industry and academia have failed to support SOA engineers during one essential step: the testing process!

2.3 Testing SOA Systems - Current Status and Research Progress

As outlined in the last section, SOAP-based SOA introduces new gains and benefits for engineers of distributed systems, however, poses new challenges too. The dynamic nature of SOA systems is definitely a valuable gain, given that the system works as expected and is actually also able to handle the dynamics. And this requires thorough testing of the whole system in real, or at least realistic, scenarios. The more complex a system gets, the more it becomes error-prone and, consequently, the more important becomes the testing in order to evaluate its correct functionality. Unfortunately, not enough attention has been given to supporting the testing process of SOA systems. Engineers are facing the problem that setting up proper tests is time-consuming and slows down the whole development process. In the worst case the effect is that testing is neglected, bugs and malfunctions are less likely to be discovered, and the whole system becomes unstable.

Up to date, testing solutions for SOA do exist but they do not cover all variations of necessary testing. The research community has done a lot of work on testing Web services and produced useful solutions. But Web services are only the basic building blocks of SOA systems and, for sure, not the only fault sources. In complex SOA systems, comprising large-scale environments of diverse interacting components, faults can happen basically everywhere: at the Web services, in the client's or dispatcher's logic, workflow engines, in the registries, the monitors, the middleware, at application servers, and any other SOA component. Furthermore, faults can happen at different levels of the network stack: at the low-level packet flow, at the service communication (e.g., message corruption), at the interaction models, etc. Recent surveys on testing for SOA [87] demonstrate that strong effort has been made in investigating testing solutions for single Web services and in investigating the testing of service compositions, e.g., BPEL processes. But

they also make evident the lack of further testing techniques aiming at complex systems which comprise large-scale environments of diverse SOA components and/or must interact with external partner services. Let us consider again the two motivating scenarios from Section 1.1: a) a self-optimizing workflow engine which monitors its environment and adapts workflow instances according to the observed situation and b) a system outsourcing tasks to external partners and, therefore, being dependent on their availability and quality of service. Both systems operate in an environment of services which is out of their control, though, essential for their functionality. In order to handle the risk of being dependent on external components, these systems must provide mechanisms to handle sudden unavailabilities, various faults, and any other kind of unexpected situations. And, obviously, these mechanisms must be well tested before deployment in order to guarantee their correctness. However, during the development process the engineer does usually not have such environments of external services which he/she could use as testbeds for evaluating his/her software. Either because he/she has no access to them or is not allowed to use them, as the provider of the external services forbids trial invocations in his policies. Nevertheless, the developed system must be tested, in spite of all the involved difficulties. *But how?* In our opinion, the only reasonable solution is to have testbed infrastructures which represent (or emulate) the final deployment environment and serve for testing purposes. But up to date, this topic has been neglected too much by the research community.

In the next sections, we give an overview of selected testing solutions, outline unsolved issues and define the problem statement which has been handled within this thesis.

2.3.1 Testing of Web Services and Web Service Compositions

The nature of Web services can be summarized as follows. They are 1) *encapsulated units of functionality accessible via their interfaces*, 2) *invoked from remote* 3), *described in WSDL documents*, 4) *discovered at registries*, and, eventually, 5) *reused by other components within an SOA to establish composite functionality*. Regarding Web services according to these characteristics, one can categorize the different aspects under which they can be tested. The *1st characteristic* allows to consider them as normal software modules (or as classes or libraries) and, therefore, testing techniques can be inherited from traditional software testing research, e.g., regression testing [111], performance and robustness testing [74, 120, 141], and white/grey/black box testing [71, 152]. The *2nd characteristic* is related to communication issues, whether messages are transported reliably, in terms of delivery assurances, content authenticity, etc. Here, testing solutions can be adopted from distributed systems research, e.g., by injecting faults [140] and corrupting messages [167]. The *3rd* and the *4th characteristics* can be combined and relate to how correct and precise Web services are described and how they can be discovered dynamically at a registry, based on their descriptions. We believe that this is a rather novel issue and there is not much of available research to be reused from other disciplines. Instead new research directions have evolved, such as semantic Web services [76, 121] which promise automated discovery. Finally, the *5th characteristic* relates to the consumption of Web services and how they can be combined into composite systems. In the SOA research community this point is mostly dominated by investigations of workflow models and executions, as done in [73, 101, 103, 118].

Bozkurt et al. follow in their detailed survey on Web service testing [87] a different and more fine-grained categorization of testing techniques. These include *test case generation* [115, 119], *partition testing* [81], *contract-based testing* [90, 100], *unit testing* [149], *model-based testing and formal verification* [169], *fault-based testing* [116, 159, 167], *interoperability testing* [82, 168], *integration testing* [137], *collaborative testing* [157], *testing quality of service* [141], and, *regression testing of Web services* [111]. They have found in sum 102 publications on Web service testing and identified a strongly growing number of publications each year, which illustrates the growing importance of research on testing solution in the SOA community.

Similar to Bozkurt's work, Canfora and Di Penda divide in their survey [89] testing techniques into different perspectives and and testing levels. The perspectives comprise the *service developer*, the *provider*, the *integrator*, a *3rd-party certifier*, and the *end-user*. As levels they count *unit testing, integration testing, regression testing, dependability testing*, and *non-functional testing*. They outline that many of the benefits of SOA pose particular challenges to testing services and service-centric systems, and that the research community should focus more on these challenges.

In this thesis, however, we do not review particular testing techniques for Web services. This is because, as stated in the beginning of Section 2.3, these aim at testing only the basic building blocks of SOA systems and this is out of scope of our research. Our work is concentrated on testing complex SOA systems which interact with other services and operate in (large-scale) SOA environments.

2.3.2 Related Research: Testing of Complex SOA Systems

Testing of complex SOA systems (CSS) is significantly more intricate and challenging than the testing of single or composite Web services. In our definition a CSS fulfills connective and controlling functionality within an SOA environment. It offers its services to other components, consumes other components' functionality (services), controls components and/or the interactions between them, or performs any other kind of complex participation and governing inside an SOA. Especially, CSS' that must interact with external components in an open manner, e.g., by incorporating dynamically discovered Web services, are difficult to test, as the external components are usually not available during the development process. But, nevertheless, the CSS' must be tested somehow. This problem becomes even more severe if the CSS' are supposed to operate in large-scale environments. Taking Amazon Mechanical Turk (mturk) [2] as an example, which is a broker for human-provided services and must handle tens of thousands of concurrent requests and forward them to registered participating services, makes the problem of testing CSS' evident. The developers of mturk, or similar systems, cannot test the its runtime qualities at the development time, as the participating services and clients are not available yet but will appear once the system is up and running and open to public usage. As argued before, such problems can be solved by applying testbeds which emulate the deployment environment and, this way, facilitate testing. Some groups have recognized the need for testbeds but, unfortunately, there has not been as much of invested effort into research on testbeds as into testing of single and composite Web services. We believe this is mostly caused by the intricacy of the problem. In the strict sense, only two groups have investigated SOA testbeds intensively and

have published their results. These are the groups of *Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"* in Pisa, Italy, who developed PUPPET [44] and and the *Faculty of Informatics* of the *Universita della Svizzera Italiana* in Lugano, Switzerland, who developed SOABench [46].

PUPPET[1] is a framework for generating Web service-based testbeds. It is mainly focused on performance issues and emulates quality of service (QoS) properties of Web services, in order to evaluate the fulfillment of service level agreements (SLAs) of composite services. PUPPET takes WSDL descriptions and WS-Agreement documents [53] as input in order to replicate the described services (by analyzing the WSDL) and to emulate QoS of these in order to verify the SLAs (taken from WS-Agreement). The applied techniques are presented in [79, 80] and the emulation is done by generating Java stubs of the services and extending the operation routines with code for hampering the execution (e.g., by injecting delays or throwing faults). In [77] they extend their approach towards emulating functional behavior using automata models, referred to as functional specification. This specification basically defines a state machine, where state transitions are triggered by service invocations and, depending on which state the service is in, can cause different effects, e.g., to fail if a particular operation has not been called before. In [78] they combine their appraoch with the ns-2 network simulator [42] in order to simulate mobility of nodes, which also has an effect on the QoS in terms of availability, etc.

SOABench [84, 85] provides sophisticated support for benchmarking of Web service middlewares (in particular BPEL engines [17]) in order to determine scalability, throughput, and other performance properties. It supports modeling of experiments, generation of service-based testbeds, runtime control on test executions, plus mechanisms for test result evaluation. Regarding its features, SOABench is strictly focused on performance evaluation and generates testbeds of services that emulate QoS properties. Furthermore, it creates BPEL process models for the test runs, plus it generates test clients which put workload on the BPEL engines by triggering the execution of the processes. With its focus on BPEL, it does not support generic customization of the testbed in the sense of opening it to non-BPEL systems or to augmenting the testbed's services with dependency structures or complex functional behavior.

These two works, PUPPET and SOABench, provide semi-automatic generation of testbeds for SOA and are, to our knowledge, the only published works on that topic, that are based on implemented prototypes. Each of these has its specific purpose (verification of SLAs in service compositions and performance evaluation of BPEL engines) and is strictly focused on solving that problem. However, this also means that they cannot be regarded as "generic purpose testbed generators" that can be applied to a multitude of testing purposes.

soapUI [49] is a commercial testing solution for Web services, offering creation and execution of automated functional, regression, compliance, and load tests. It supports the generation of mock-up Web services which can be extended with functional and non-functional behavior, e.g., by returning custom responses, in order to test client functionality. soapUI's main restriction is its strict focus on point-to-point communication. Ergo, it is useful for testing a client's or service's quality, but it is not applicable for CSS' which operate in large-scale environments and participate in complex interactions.

[1] **P**ick **up** **P**erformance **E**valuation **T**estbed

Further related work has been done on techniques for controlling tests of distributed systems. Weevil [163], for example, supports experiments of "distributed systems on distributed testbeds" by generating workload. It automates the deployment and execution of experiments and allows to model the experiment's behavior via programs written in common programming languages linked to its workload generation library. Similar to Weevil, the DDSOS framework [155] deals with testing SOAs and provides model-and-run support for distributed simulation, multi-agent simulation, and an automated scenario code generator creating executable tests. However, solutions like these aim at supporting tests (of SOAs) executed *on existing testbeds* but do not generate these testbeds themselves. Therefore, we see them rather as complementary tools to testbed generators.

2.4 Problem Statement

Looking at the published efforts on testing of SOAs and on the survey papers covering this topic, it becomes evident that certain kinds of testing have been addressed with great effort, but there still exists a lack of solutions which would support engineers in setting up tests for complex SOA systems. Consequently, the problem can be summarized into the following two sentences:

> *How can we facilitate the generation of customizable testbed infrastructures that emulate SOA's in a realistic manner?*
>
> *How can we make this process as simple and intuitive as possible, in order to maximize the usability for engineers?*

The next sections comprise our answers to these questions, present the outcomes of our research work, and the give an overview of our proof of concept implementation.

CHAPTER 3

GENESIS - Specification, Generation, and Steering of Web Service Testbeds

Published in:

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

Juszczyk L., Truong H.-L., Dustdar S. (2008).

13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08), 31. March - 4. April 2008, Belfast, Northern Ireland.

Outline. In this chapter we present the very first iteration of GENESIS. After we had realized the urgent need for an extensible testbed generator for SOA, we developed the concepts of GENESIS and implemented a first prototype. Compared to available works on testbed generation, GENESIS contributed by offering more flexibility for implementing functional behavior as well as non-functional properties of the testbed's Web services. It deploy the testbed automatically at a distributed back-end infrastructure and makes it possible to control the testbed's Web services at runtime from a centralized front-end. A major feature of the GENESIS framework was the ability to enhance the generated Web services with plugins, providing a basis for extendability and, hence, for maximizing its applicability for testing of complex SOA-based systems.

3.1 Motivation

Complex computer systems have always involved large numbers of (distributed) components that interact with each others [94, 151]. While in the past these were hosted in predominantly

homogeneous environments, such as inside closed organizations or military systems, today's complex systems have evolved into being used in heterogeneous environments which incorporate, for instance, legacy systems or components based on different platforms and/or runtime environments. As a consequence, interoperability has become an important issue, which favored the acceptance of SOAP-based Web services as a de-jure communication standard. 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). As outlined in Chapter 2, SOA brings various benefits, such as flexibility, modularity, composability, and reusability, just to name the most well-known ones. Though, as a side-effect, these features pose challenges to the engineers as they cause additional complexity and error-proneness. This problem must 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, e.g. in [74, 120, 141, 156, 158, 170]. However, those solutions mainly aimed at analyzing only individual Web services by performing various client-oriented checks and did not cover the testing of systems which are consuming other Web services themselves and, therefore, require testbed infrastructures in order to be evaluated at runtime. There has been only limited support for the deployment of whole testbeds, consisting of real Web services, which we regarded as a major problem.

This chapter explains the concepts of the first GENESIS[1] testbed generator framework. GENESIS combines an approach for automatic generation and deployment of Web services at the back-end with a API at the front-end. Engineers can specify functional and non-functional properties of Web services which are deployed on-the-fly on remote hosting environments. Complex behavior of the testbed can be achieved with various plugins which extend the functionality of the individual Web services and can be steered remotely from the front-end. Therefore, GENESIS allows engineers to set up testbed infrastructures for complex service-oriented systems.

3.1.1 Requirements

Testing software for faults and failures is an essential part of the software development process, which should be performed continuously during all stages of the entire process. This includes unit tests [95] for checking the quality of the individual modules, integration tests [108] when these modules are being connected, functional tests [102] 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 [120, 141], while the other methods aim at testing whole service infrastructures and applications operating on them [74, 156, 158]. However, although SOA has been an important topic in research and industry during the last years, we noticed a lack of tools supporting engineers to set up such infrastructures of services for testing purposes. We argue that such a tool should meet the following requirements:

- *Customizability*: Specification of Web services with customizable functional and non-functional properties.

[1] **Gene**rating **S**ervice-based Testbed **I**nfrastructure**s**

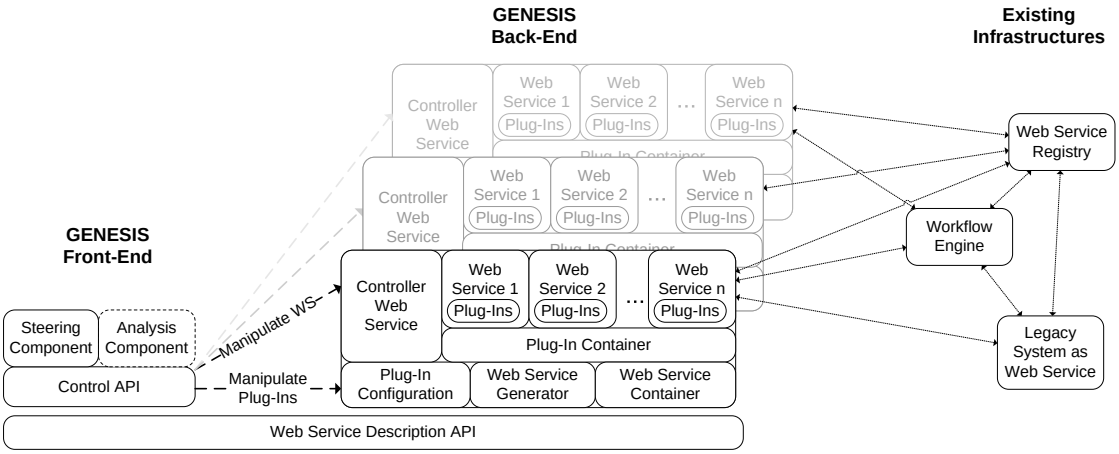


Figure 3.1: Architecture of GENESIS

- *Extendability*: Pluggable extensions of Web service functionality.
- *Distributed hosting*: Generation and deployment of Web services on a distributed back-end.
- *Support for functional complexity*: Control structures and complex interdependencies between services.
- *Integration*: Integration with existing SOA infrastructures.
- *Convenience*: Provision of the generator's functionality via 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 was an absence of solutions which fulfill this. GENESIS was designed to fill this gap.

3.2 Concept and Architecture of GENESIS

The architecture of GENESIS (see Figure 3.1) comprises a centralized front-end, for controlling the testbed, and a distributed back-end hosting the generated 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 3.2 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.

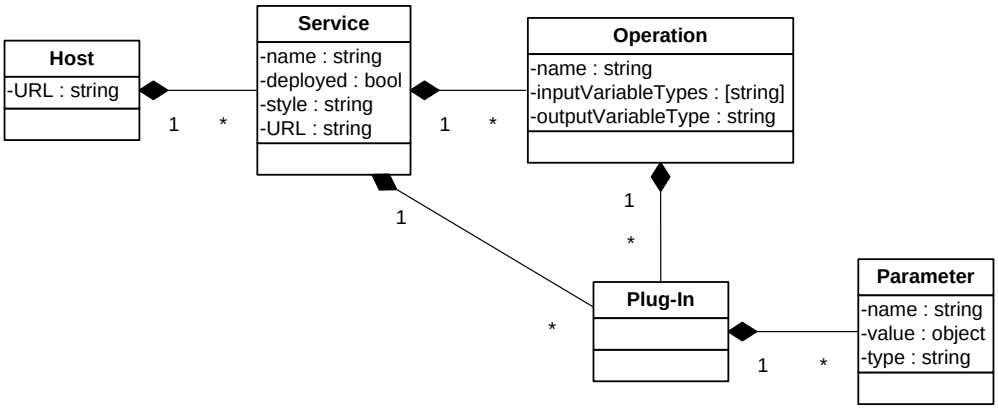


Figure 3.2: Model used to describe Web services in GENESIS

- `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 plugins which are being invoked at deployment and undeployment.
- `Operation`: An operation has a name, a set of input types, a single output type, and a fault type. Operations can be extended by plugins too.
- `Plug-In`: Plugins 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 plugin in order to be accessible. It has a name, a data type, and a value.

Based upon this model, GENESIS provides a Java API for instantiating and manipulating Web service descriptions and for transferring them to the back-end for deployment.

At the *front-end* the API can be either directly accessed inside Java-based applications which control the testbed or, alternatively, can be accessed via our `Steering Component` which is built upon the Jython script interpreter [38]. In Section 3.4 we show sample scripts which demonstrate how testbeds can be created and steered.

At the *back-end* side, the functionality is split into several modules. Incoming requests, comprising Web service descriptions, are received by the `Controller Web Service` and forwarded to the `Web Service Generator` which initiates the transformation procedure in order to generate deployable service instances. These instances are being deployed at the JAX-WS-based [32] `Web Service Container`. This step is described in more detail in Section 3.3.1. Plugins, which augment the generated Web services with functionality, are registered at the `Plugin Container` and are being controlled via parameters that are stored in the local `Plugin Configuration Database`.

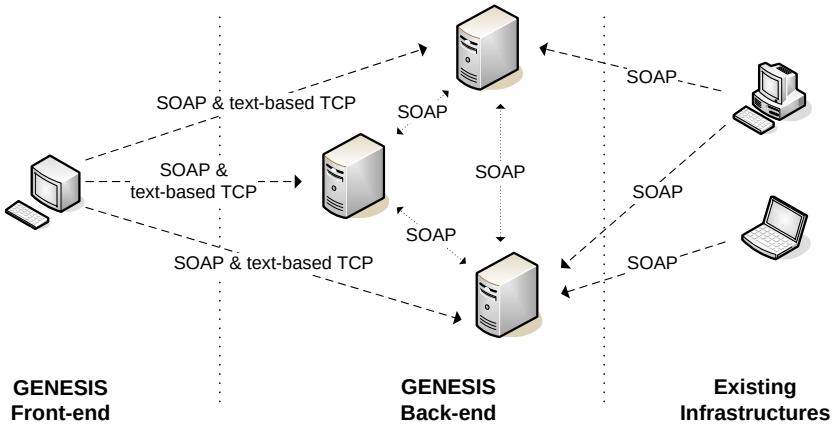


Figure 3.3: Communication between GENESIS front-end, back-end, and existing SOA infrastructures

In *between the front-end and the back-end* (see Figure 3.3), the communication is based on SOAP plus on a simple text-based protocol. The back-end's SOAP service provides access to all functionality, however, SOAP implies significant communication overhead (in terms of HTTP headers, SOAP envelope, and XML encoding). This may become a problem if the testbed becomes large-scale and change request are executed frequently. In order to provide better throughput, we provide the additional text-based protocol for manipulating plugin parameters in light-weight manner without unnecessary overhead.

3.3 Generating Complex Web Services

The techniques for generation of deployable Web services out of their descriptions is derived from model-driven development (MDD) and generative programming. The `Web Service Generator` creates by default empty skeleton code which implements the described Web service, yet, does not provide any functionality. This dummy service can be extended with plugins, for instance, to provide proper computational functionality, to establish interdependencies inside the testbed (e.g., nested invocations among services), to simulate quality of service (QoS), or to introduce any kind of required behavior.

3.3.1 Generating Web Services

The `Web Service Generator` parses the service description to analyze the service's interface, operation signatures, binding style, and referenced plugins. Out of this information it generates a runnable Web service instance and deploys it. This procedure involves multiple steps handled by the back-end's module, which are depicted in the sequence diagram in Figure 3.4:

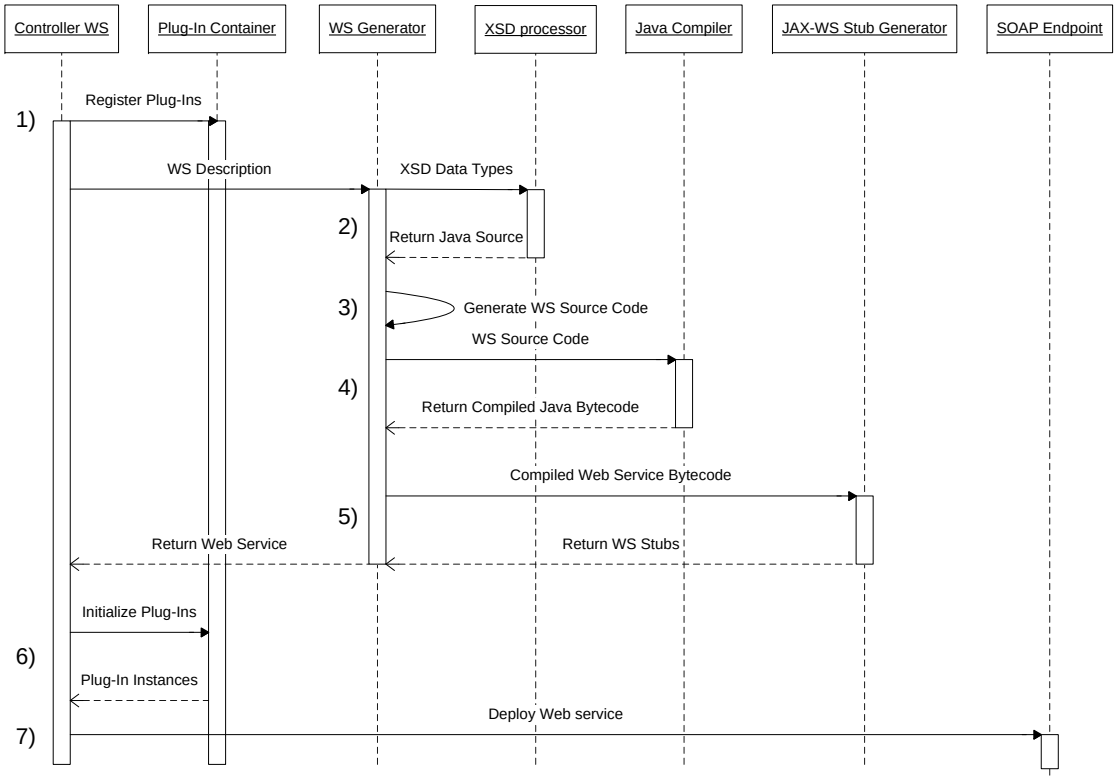


Figure 3.4: Web service generation process

1. The Web service description is received by the `Web Service Generator` module and is checked for referenced plugins (explained in more detail in Section 3.3.2). Referenced yet missing plugins are transferred from the front-end and registered locally in order to establish the required runtime environment.

2. The description is checked whether the service uses solely primitive data types (e.g., string, integer) for the request- and response-values or whether complex data types, described in XML Schema Definitions (XSD) [64], are referenced. Complex types are passed to `xjc`, which is the *XML to Java compiler* provided by JAX-WS, for generating corresponding Java classes out of the XSD definitions.

3. Using Apache Velocity [14], the `Web Service Generator` module generates a JAX-WS-compliant source code of the Web service. The Velocity template files are listed in Appendix A.2.

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

5. The compiled Web service is passed to `wsgen`, which is provided by JAX-WS too, 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-based SOAP endpoint.

The following listings show the model of a sample Web service, the source code of the generated Web service, and the WSDL document of the deployed instance. In Listing 3.1 a simple service is modeled in Jython. In Lines 5-8 two plugins are instantiated and registered for usage, and any empty model of the Web service is created. It contains only a single service operation, named `getISBN`, that is created in Lines 10-15. For simplicity we have only used simple data types (strings). In Line 15 the operation's functionality (referred to as behavior in the script) is set to use two methods of the `QOSPlugin` and invoke them in sequence. The alignment of plugins is specified in a simple language that support sequential and parallel calls, and try-catch structures for error handling. In Lines 17-21 two special operations are defined: hooks, which are not exposed as service operations but are executed automatically with the service deployment and undeployment. These can be extended with plugins as well, e.g., in order to register the service at some broker, as done in this sample. Finally, the operations are attached to the service, a back-end host is referenced (Line 27), and the service is sent to this host for deployment.

```
1  from at.ac.tuwien.vitalab.genesis.model import *
2  from at.ac.tuwien.vitalab.genesis.server.plugin import *
3  from java.util import *
4
5  QOSPlugin()
6  RegistryPlugin()
7
8  newService=Service("BookService")
9
10 operation=Operation("getISBN")
11 inputs=LinkedHashMap()
12 inputs.put("bookName","string")
13 operation.setInputTypes(inputs)
14 operation.setOutputType("string")
15 operation.setBehavior(" ( QOSPlugin.simulateDelay -> QOSPlugin.simulateFailure ) ")
16
17 dep=Operation("deploy")
18 dep.setBehavior("RegistryPlugin.register")
19
20 undep=Operation("undeploy")
21 undep.setBehavior("RegistryPlugin.deregister")
22
23 newService.addOperation(operation)
24 newService.addOperation(dep)
25 newService.addOperation(undep)
26
27 remoteHost=Host("http://localhost:7000/WebServices/GeneratorService")
28 remoteHost.addService(newService)
29 newService.deploy()
```

Listing 3.1: Jython Model of sample Web service

In Steps 2 and 3 of the generation process, the `Web Service Generator` translates the modeled Web service into JAX-WS-compliant Java code. The next listing demonstrates the

outcome of translating the BookService. Basically, it generates the class `BookService`, attaches annotations for JAX-WS (e.g., which binding style is used (Lines 11-15) and which operations/methods are meant to be exposed with the service (@WebMethod for `getISBN()`)), and implements the service's operations. Plugins, that are referenced within the operations, are called via `AWebServicePlugin.callPlugin("{Pluginname}.{Functionname}", context)` and can exchange data via the `context` variable.

```
1  package repository.wsp1947109707;
2
3  import java.lang.*;
4  import javax.jws.*;
5  import javax.jws.soap.*;
6  import java.util.*;
7  import at.ac.tuwien.vitalab.genesis.server.*;
8  import at.ac.tuwien.vitalab.genesis.server.plugin.*;
9  import at.ac.tuwien.vitalab.genesis.model.*;
10
11 @WebService(name = "BookService",
12             targetNamespace = "http://vitalab.tuwien.ac.at/generatedService/BookService"
                   )
13 @SOAPBinding(style = SOAPBinding.Style.DOCUMENT,
14               use = SOAPBinding.Use.LITERAL,
15               parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
16 public class BookService extends AWebService {
17
18   public BookService() {
19
20   }
21
22
23   @WebMethod
24   public java.lang.String getISBN(@WebParam(name="bookName") String bookName) throws
         Exception
25   {
26     LinkedHashMap<String,Object> arguments=new LinkedHashMap<String,Object>();
27     arguments.put("bookName",bookName);
28
29     final PluginContext context=new PluginContext(this,"getISBN",arguments);
30
31     try {
32       callPlugins_getISBN(context);
33     } catch (Exception e) {
34       throw new Exception("Operation 'getISBN' of service 'BookService' threw exception
             : "+e.getMessage());
35     }
36
37     if (context.getReturnValue()!=null) {
38       return (java.lang.String)MessageType.deserialize(context.getReturnValue(),java.
             lang.String.class);
39     }
40     return new String("s");
41   }
42
43   private void callPlugins_getISBN(final PluginContext context) throws Exception
44   {
45     AWebServicePlugin.callPlugin("QOSPlugin.simulateDelay",context);
46     AWebServicePlugin.callPlugin("QOSPlugin.simulateFailure",context);
47   }
48
49
```

```
50    protected void onDeploy() throws Exception
51    {
52      LinkedHashMap<String,Object> arguments=new LinkedHashMap<String,Object>();
53
54      final PluginContext context=new PluginContext(this,"deploy",arguments);
55
56      try {
57        callPlugins_onDeploy(context);
58      } catch (Exception e) {
59        throw new Exception("Operation 'onDeploy' of service 'BookService' threw
              exception: "+e.getMessage());
60      }
61    }
62
63    private void callPlugins_onDeploy(final PluginContext context) throws Exception
64    {
65      AWebServicePlugin.callPlugin("RegistryPlugin.unregister",context);
66    }
67
68
69    protected void onUndeploy() throws Exception
70    {
71      LinkedHashMap<String,Object> arguments=new LinkedHashMap<String,Object>();
72
73      final PluginContext context=new PluginContext(this,"undeploy",arguments);
74
75      try {
76        callPlugins_onUndeploy(context);
77      } catch (Exception e) {
78        throw new Exception("Operation 'onUndeploy' of service 'BookService' threw
              exception: "+e.getMessage());
79      }
80
81    }
82
83    private void callPlugins_onUndeploy(final PluginContext context) throws Exception
84    {
85      AWebServicePlugin.callPlugin("RegistryPlugin.deregister",context);
86    }
87
88  }
```

Listing 3.2: Source of generated Web service

As final steps of the generation process, the Java code is compiled, service stubs are generated, and it is being deployed at the JAX-WS runtime, resulting in a Web service with the following WSDL description.

```
<definitions targetNamespace="http://vitalab.tuwien.ac.at/generatedService/BookService"
     name="BookServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/BookService"
          schemaLocation="http://localhost:7000/WebServices/BookService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="getISBN">
    <part name="parameters" element="tns:getISBN"/>
  </message>
  <message name="getISBNResponse">
```

```
    <part name="parameters" element="tns:getISBNResponse"/>
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception"/>
  </message>
  <portType name="BookService">
    <operation name="getISBN">
      <input message="tns:getISBN"/>
      <output message="tns:getISBNResponse"/>
      <fault message="tns:Exception" name="Exception"/>
    </operation>
  </portType>
  <binding name="BookServicePortBinding" type="tns:BookService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getISBN">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal"/>
      </fault>
    </operation>
  </binding>
  <service name="BookServiceService">
    <port name="BookServicePort" binding="tns:BookServicePortBinding">
      <soap:address location="http://localhost:7000/WebServices/BookService"/>
    </port>
  </service>
</definitions>
```

Listing 3.3: WSDL document of generated Web service

Basically, the generation process transforms the service's model into a running instance by implementing the interface details (e.g., operation signatures) but it does not program the service's functional properties. These are outsourced to the plugin mechanism of GENESIS.

3.3.2 Establishing Complex Dependencies through Plugins

SOA's building blocks are services which are supposed to be maximally autonomous, have as little dependencies as possible, and simple in usage. However, taking a look at today's state of the art of SOA, it becomes clear that SOA can be arbitrarily complex, especially when service composition comes into play. Taking languages for service choreography and orchestration, such as WS-CDL [54] or BPEL [17], as examples, we can identify various sources of complexity, e.g., dependencies between services, control constructs, and fault handlers. 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 order to emulate this in GENESIS, complexity inside the testbed can be realized by applying plugins to the individual Web services.

At the implementation level, these plugins must extend an abstract class which defines mandatory constructor and destructor methods, provides serialization functionality for transferring plugin code to remote hosts, and registers itself automatically at the corresponding `Plugin`

`Container`. Each plugin 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 plugin is free to define a set of parameters through which it can be controlled remotely from the front-end. Taking as example a plugin for registering the Web service at some registry, such as UDDI [50], possible parameters would specify the host name of the registry server, authentication data, and additional meta-data about the service.

If one wants to apply a plugin for extending a service's operation, he/she must reference it via its name and the called method. For instance, by specifying `"QOSPlugin.simulateDelay"` as a behavior, the called operation will execute the corresponding plugin at invocation time. If multiple plugins must be combined, we support sequential alignment (`"(Plugin1.method -> Plugin2.method)"`), in parallel (`"[Plugin1.method || Plugin2.method]"`), and try/catch blocks for error handling (`"{Plugin1.method # Plugin2.method}"`) where Plugin2 is only called on faults thrown by Plugin1. Furthermore, plugins can accept arguments (e.g., from the service request), return values, and, consequently, exchange data among each others. The following listing shows a more sophisticated plugin alignment using the `InvocationPlugin` that calls remote services. At first, two remote services (identified by their name) are called in parallel (Lines 3-7), followed sequentially by an further call which takes their responses as input (Line 9). If, and only if, any of these calls throw an exception, e.g., due to I/O faults, the fault is forwarded to the reporting service.

```
1  {
2    (
3      [
4        InvocationPlugin."d1=getAndCheckService1.getAndCheck(arg.name)"
5      ||
6        InvocationPlugin."d2=getAndCheckService2.getAndCheck(arg.name)"
7      ]
8    ->
9      InvocationPlugin."return=retrievalService.processData(d1,d2)"
10   )
11 #
12   InvocationPlugin."reportService.reportError()"
13 }
```

Listing 3.4: Sample plugin alignment statement

The alignment language's JavaCC-based [33] grammar definition is attached in Appendix A.1.

With the first prototype of GENESIS, which got published as open-source [20], we provide five sample implementations of plugins:

- `QOSPlugin`: Simulates performance- and dependability-specific QoS properties, such as processing time, scalability, throughput, availability, and error rate. Performance attributes are simulated by delaying responses, while dependability is simulated by throwing faults and altering the service's availability (deploying and undeploying). The plugin can be controlled by setting the corresponding parameter for each QoS property, e.g., a percentage value for availability.
- `InvocationPlugin`: Performs nested invocations of other Web services in the testbed.

- `BPELPlugin`: Integrates the bexee [16] BPEL engine into GENESIS and executes composed processes inside Web service operations. As a parameter it accepts BPEL process definitions which can contain precise as well as abstract partner links. Precise partner links allow to integrate external Web services, for instance from 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 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 [123] and other standards. In contrast to the other plugins, 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 must be specified via parameters. The meta-data about the service is mainly retrieved from the description of the Web service itself.

3.4 Practical Application

GENESIS provides a Java API which covers all functionality for specifying, generating, and steering of testbeds. The API can be embedded into any application, for instance a GUI, or into the Bean Scripting Framework [31] which seamlessly integrates scripting languages into Java. As a starting point for engineers we provide a `Steering Component` based on the Jython script interpreter [38]. 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.

3.4.1 Testbed Configuration

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 3.5 shows a sample configuration file.

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

Second, complex data types, 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 plugins and their local parameters. If the plugin declaration is omitted, the default behavior is assumed, which was defined in Line 10. Moreover, plugins can be invoked during deployment and undeployment of a service (Lines 36-41).

```
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"> <!-- Simulate selected QoS -->
11       ( QOSPlugin.simulateDelay -> QOSPlugin.simulateFailure )
12     </QOS>
13     <EmptyBehavior> <!-- Do nothing at all -->
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"
```

```
                deploy="true">
        <!-- extend template service -->
        <operation name="getISBN" >
          <input article="xs:book"/>
          <output return="string"/>
        </operation>
      </service>
    </host>
  </environment>
</configuration>
```

Listing 3.5: Testbed configuration

A more complex configuration example can be found in Appendix A.3.

3.4.2 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 the URL `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")

operation.setBehavior("QOSPlugin.simulateQOS") # set plugin references

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 plugins is achieved by modifying parameters via getter/setter methods, whereas the setter methods forward the changes to the corresponding plugins 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 plugin 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 plugins, e.g., observe/notification mechanisms for parameters in order to trigger chains of updates. The following listing simply listens to parameter changes of an operation and forwards the new value to another operation.

```
param1=op1.getParameter(QOSPlugin.PROCESSINGTIME)
param2=op2.getParameter(QOSPlugin.PROCESSINGTIME)

def fwd():
... param2.setValue(param1.getValue())

param1.observeParam(0,fwd)

param1.setValue(3000)
param2.getValue()
> 3000
```

For the sake of simplicity, we have shown rather primitive applications 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 delays.

```
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

beh="[ BPELPlugin.run || QOSPlugin.simulateDelay ]"
operation.setBehavior(beh)

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.

3.4.3 Illustrating Scenario

In [123] 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 [124] 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 them, this happens mainly due to the shortcomings of the currently available Web service registry standards, UDDI [50] and ebXML [18], 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 engineers with dynamic binding and invocation of services. In that approach, services are published dynamically at runtime to other participants within the network and are described by meta-data of functional and non-functional attributes, e.g., monitored QoS attributes [141]. 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 has been designed to disburden SOA engineers 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 plugin for registering deployed Web services automatically at the VReSCO infrastructure. This is a potential 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, engineers 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. A report on testing VReSCO on a GENESIS testbed is presented in [123].

CHAPTER 4

GENESIS2 - Dynamic Testbeds for SOA

Published in:

Script-based Generation of Dynamic Testbeds for SOA.

Juszczyk L., Dustdar S. (2010).

8th IEEE International Conference on Web Services (ICWS'10), 5. - 10. July 2010, Miami, USA.

Outline. In this chapter we present the evolution of GENESIS, now referred to as GENESIS 2 (or in short, G2). Although we have made significant progress in testbed generation with the first framework, we noticed later on several restrictions which hampered a convenient application of it. For instance, it was limited to generating testbeds consisting only of Web services, plus it did not offer fine grained control on the functional properties of these services. As some of these restrictions had their roots deeply in the concepts of GENESIS, we decided to redesign it with the lessons learned and to work on a new testbed generator from scratch. The outcome, G2, differs significantly from its predecessor. It has been designed for being generic, supports the generation of various SOA components, has a more flexible plugin system, and provides many novel features for generating customizable testbeds for SOA.

4.1 Motivation

As outlined in the state of the art analysis in Chapter 2, today's SOA does not only comprise services, clients, and brokers which interact according to the publish-find-bind paradigm but has been extended with a multitude of components which communicate via exchanging SOAP messages and benefit from SOA's flexibility. Moreover, novel features which are today associated

with SOA [135] are adaptivity [91], self-optimization and self-healing (self-* in general) [99], and autonomic behavior [164]. The result of this evolution is that, on the one hand, SOA is being increasingly used for building distributed systems, but, on the other hand, is becoming more and more complex itself. As complexity implies error-proneness as well as the need to understand how and where such complexity emerges, SOA-based systems must be tested intensively during the whole development process and, therefore, require realistic testbeds. These testbeds must comprise emulated Web services, clients, registries, bus systems, mediators, and other SOA components, to simulate real world scenarios. However, due to missing tool support, the set up of such testbeds has been a major burden for SOA engineers. Even the first version of GENESIS solved only a subset of the problems. For testing complex systems which operate in service-based environments, the engineer is still facing the problem of setting up realistic test scenarios which cover the system's whole functionality. There do exist solutions for testbed generation but these are restricted to specific domains, e.g., for checking Service Level Agreements by emulating Quality of Service [77]. However, if engineers need generic support for creating customized testbeds covering various aspects of SOA, no solutions exist to our knowledge which would relieve them from this time-consuming task.

In this chapter we present the next iteration of our work on a solution for this issue. We explain the GENESIS2 framework (in short, G2) which allows to set up SOA testbeds and to manipulate their structure and behavior on-the-fly. It comprises a front-end from where testbeds are specified and a distributed back-end on which the generated testbed is hosted. At the front-end, engineers write Groovy scripts to model the entities of the testbed and to program their behavior, while the back-end interprets the model and generates real instances out of it. To ensure extensibility, G2 uses composable plugins which augment the testbed's functionality, making it possible to emulate diverse topologies, functional and non-functional properties, and behavior.

4.1.1 Evolution of GENESIS

To our knowledge, GENESIS (will be now called G1) was the first available "multi purpose" testbed generator for SOA. However, G1 suffers from various restrictions which limit the framework's functionality and usability. First of all, the behavior of Web services is specified by aligning plugin invocations in simple structures (sequential, parallel, try/catch) without having fine-grained control. This makes it hard to implement, for instance, fault injection on a message level [167]. Also, deployed testbeds can only be updated by altering one single Web service at a time, which makes the management of large-scale testbeds not efficient. Moreover, G1 is focused on Web services and does not offer the generation of other SOA components, such as clients or registries.

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

- customizable control on structure, composition, and behavior of testbeds,

- ability to generate not only Web services, but also other SOA components,
- ability to create and control large-scale testbeds in an efficient manner,
- and, furthermore, a more convenient and intuitive way for modeling and programming the testbed.

The appearance of the listed requirements made it necessary to redesign GENESIS and to rethink its concepts. These efforts resulted in our new framework, GENESIS2.

4.2 The GENESIS2 Testbed Generator

To avoid ambiguities, we will be using the following terminology: *model schema* for the syntax and semantics of a testbed specification, *model types* for the single elements of a schema, *model* for the actual testbed specification, and *testbed (instance)* for the whole generated testbed environment consisting of individual *testbed elements*, e.g., services, registries, or message dispatchers.

4.2.1 Basic Concepts and Architecture

As done in the first GENESIS, G2 too comprises a centralized front-end and a distributed back-end. Engineers write scripts in which they model and program their testbeds at the front-end, and at the back-end G2 transforms the models into real testbed instances. The front-end provides a virtual view on the testbed, allows engineers to manipulate it at runtime, and propagates changes to the back-end in order to adapt the running testbed. The framework itself offers

- generic features for modeling and manipulating testbeds,
- extension points for plugins,
- inter-plugin communication among distributed instances, and
- 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 G2 framework follows a modular approach and provides the functional grounding for composable plugins that implement generator functionality. This feature is based on having an extensible model schema (e.g., as in Figure 4.1), that specifies *what* can be generated in the testbed and *which* customizations are possible. The schema can be regarded as a common denominator which defines the framework's functionality and is shared among the front-end and all back-end hosts. To put it in other words, the engineer knows that if the schema at the front-end allows to model certain components, then the back-end instances will be able to generate the corresponding instances. Basically, plugins enhance the model schema by integrating custom

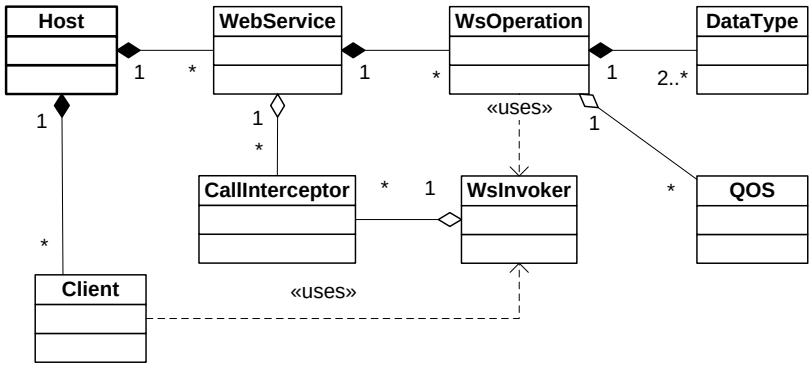


Figure 4.1: Sample G2 model schema

model types and interpret these to generate deployable testbed elements at the back-end. Taking the provided `WebServiceGenerator` plugin for example, it enhances the model schema with the types `WebService`, `WsOperation`, and `DataType`, integrates them into the model structure on top of the default root element `Host`, and, eventually, supports the generation of Web services at the back-end. Furthermore, the provided model types define customization points (e.g., for service binding and operation behavior) which provide the grounding for plugin composition. For instance, the `CallInterceptor` plugin, which intercepts SOAP messages, attaches itself to the `WebService` type and allows to program the intercepting behavior which will be then automatically deployed with the services.

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

For a better understanding of the internal procedures inside G2, we take a closer look at its architecture. Figure 4.2 depicts the layered components, comprising the base framework, installed plugins, and, on top of it, the generated testbed:

- At the very bottom, the basic runtime consists of Java 6, Groovy, and some 3rd-party libraries.
- At the framework layer, G2 provides itself via an API and a shared runtime environment is established at which plugins and generated testbed elements can discover each other and interact. Moreover, an active repository distributes detected plugins among all hosts.
- Based on that grounding, installed plugins register themselves at the shared runtime and integrate their functionality into the framework.
- The top layer depicts the results of the engineer's activities. At the front-end he/she is operating the created testbed model. The model comprises virtual objects which act as a

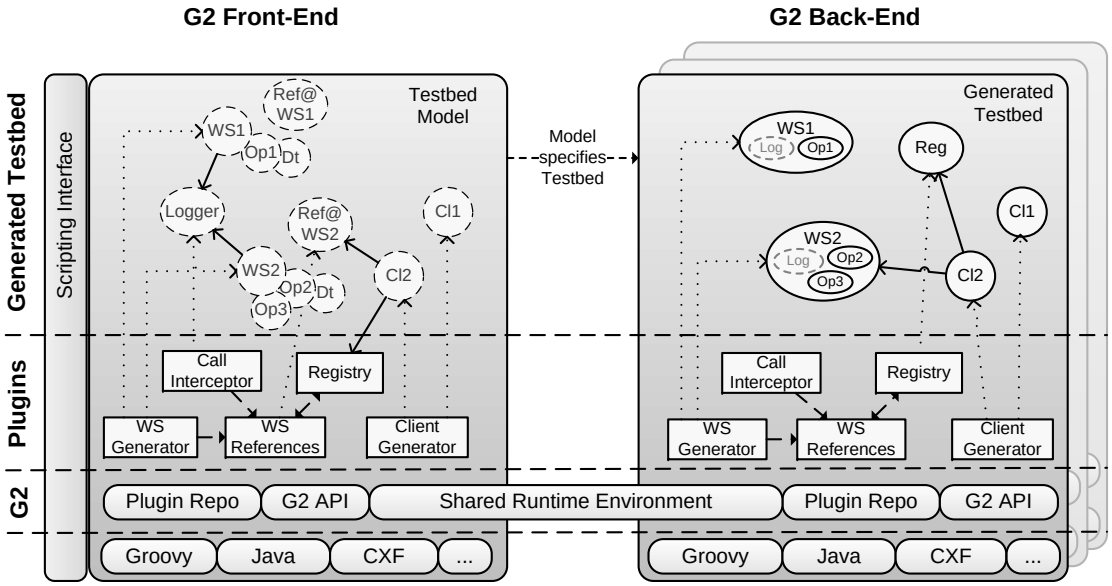


Figure 4.2: G2 architecture: infrastructure, plugins, and generated elements

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.

However, Figure 4.2 provides a rather static image of G2, which does not represent the system's inherent dynamics. Each layer establishes its own communication structures (see Figure 4.3) which serve different purposes:

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

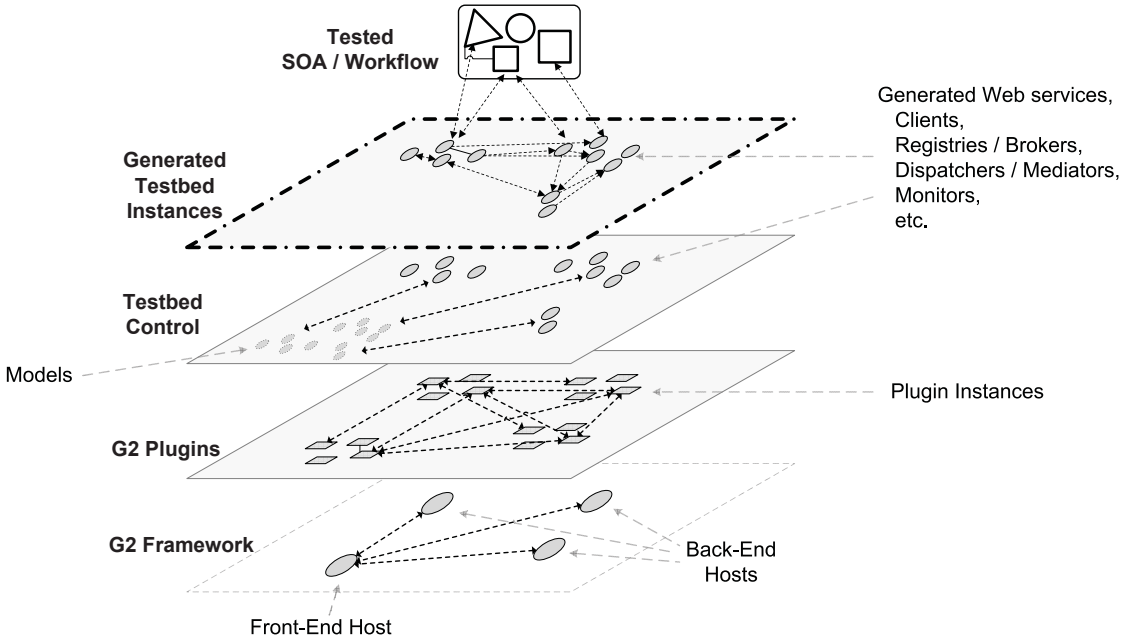


Figure 4.3: Interactions within G2 layers

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

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

4.2.2 Exploitation of Groovy Features

G2 derives a great deal of its flexibility and extensibility from Groovy [24]. Groovy is a dynamic programming language for the Java Virtual Machine, providing modern features such as dynamic typing, closures, and support for meta programming. Also, it has a compact syntax and can be used as an easy-to-read scripting language.

G2 uses Groovy's dynamic `Expando` type as a base class for model types. This allows to expand the model (ergo the generated testbed) on-the-fly and to observe changes, which facilitates automatic front-end/back-end synchronization. Moreover, by intercepting model manipulation

requests, plugin developers can customize the handling of these (e.g., to log everything) or can even restrict the model's expandability.

Internally, model objects are realized as flexible hash maps and entire testbed models are constructed by aggregating these, e.g., by attaching a `WsOperation` instance to the corresponding list inside a `WebService`'s map. However, aggregating model objects by hand is rather cumbersome and inefficient, especially for complex testbeds. As a solution, we use Groovy's *Builder* support which helps to create nested data structures in an intuitive manner. The following sample demonstrates the convenience by comparing the modeling of Web services with and without builders:

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

// usage of model builder for generating the same WS model
def s2 = webservice.build {

  TestService() {

    binding= "doc,lit"
    tags= ["test"]

    SayHello(name: String, response: String) {
      return "hello $name"
    }
  }
}[0]
```

Listing 4.1: Modeling Web services with and without Closures

The resulting model and its internal nested map structure look as shown in the following snippet. The Web service model comprises the model of the operation in the corresponding variable and the operation covers its data types. Keeping models in such map structures benefits the serialization of testbed models which can be mapped easily into XML.

```
type         : WebService
id           : ad8f20ae-ea4e-4909-8c4b-d1c10668167e
name         : TestService
bindingUse   : LITERAL
bindingStyle: DOCUMENT
behavior     : return "hello $name"
operations   : <List>
                  (
```

```
type      : WsOperation
id        : f00d74f9-3bb0-4969-8161-a139b2a7cb26
name      : SayHello
paramTypes   : <Map>
        {
        name      :
                  type=DataType
                  id=86056f9e-0c25-46f1-ac5d-3b9e542e89dd
                  name=String
                  javaClassName=java.lang.String
        returnType :
                  type=DataType
                  id=86056f9e-0c25-46f1-ac5d-3b9e542e89dd
                  name=String
                  javaClassName=java.lang.String
        }
)
```

Listing 4.2: Internal structure of a Web service model

By default, model types allow to encapsulate executable code (e.g., operation behavior in previous sample) in order to program the testbed and to implement fine-grained customizations. For this purpose, G2 is using Groovy *closures* [25] which are executed on top of the shared runtime environment (introduced in previous section). The environment provides access to all registered entities inside the testbed (plugins, deployed instances, shared data, etc.) and offers introspection via model reflection. For example, the next snippet defines a simple operation behavior that determines the number of available Web service models and invokes the `Logger` plugin. During execution, Groovy would resolve the aliases `webservice` and `log`, and provide access to the referenced instances.

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

As the structures of testbed models can get quite complex, we use *GPath* expressions [26] to offer simplified access via compact queries. In the following example, the query checks whether there exist local services which are tagged with `"test"` and which contain an operation named `"SayHello"`. Such queries are applied to select and manipulate specific parts of the model with one single command.

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

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

4.2.3 Extensible Generation of Testbed Instances

Because of its generic nature, which provides a high level of extensibility, the G2 framework outsources the generation of testbed elements to the plugins. This means each plugin that introduces a model type into the schema is also handling the generation of deployable instances out of it. G2 does also not predefine a strict methodology for how the instances must be generated, but rather provides supporting features. This might raise the false impression that we are just providing the base framework and leave the tricky part to the plugin developers. The truth is that we kept the framework generic on purpose, in order to have a basic grounding for future research on testbed generation, which might also include non-SOA domains. At the time of publishing G2, we had developed a set of plugins covering basic SOA:

- `WebServiceGenerator` creating SOAP Web services
- `WebServiceInvoker` calling remote SOAP services, both generated and preexisting ones (e.g., 3rd-party .NET-based)
- `CallInterceptor` processing SOAP calls on a message level (e.g., for fault injection or logging)
- `DataPropagator` providing automated replication of variables/macros/functions among back-end hosts in order to establish a homogeneous shared runtime environment
- `QOSEmulator` emulating Quality of Service properties
- `SimpleRegistry` for global service lookups
- `ClientGenerator` seeding testbeds with standalone clients (e.g., for bootstrapping testbed activities)

Meanwhile this set of plugins got extended with new ones, e.g., the fault injection plugins presented in Chapter 5. Of these plugins, however, the `WebServiceGenerator` plays a major role and, therefore, serves as a good example for demonstrating the testbed generation process. We have reused selected parts of the generation code from G1 but we were able to simplify it significantly by using Groovy features. Basically, the process comprises the following steps:

1. Recursive analysis of the `WebService` model to determine used customization plugins and message types.
2. Translation of message types (`DataType` models) to Java classes that represent the XSD-based data structures (using `xjc`, the Java XML Binding Compiler).
3. Automatic generation of Java/Groovy source code implementing the modeled Web service.
4. Compilation of sources using Groovy's built-in compiler.

5. Forwarding the generation of customizations to corresponding plugins. E.g., to `CallInterceptor` which extends the Web service's instance with additional code for processing SOAP messages.

6. Deployment of completed Web service instance at local Apache CXF [4] endpoint.

7. Subscription to model changes for automatic adaptation of deployed Web service instance.

The following Java/Groovy code demonstrates the generated source code for the Web service modeled in Listing 4.1 on Page 47. Similar as done in G1, the model is mapped into a JAX-WS-annotated class. However, one of the big differences is the handling of functional behavior which got hard-coded in G1. In G2, the Web service is accessing its model at invocation (e.g., in Line 41) in order to retrieve the Closure object containing the behavior specification, bind some variables to its runtime environment (Lines 44-48), and executes it (Line 49). This flexible retrieval of behavior code allows to perform hot updates on running Web services, just by replacing Closure variables, and, this way, updating their functionality without the need of redeploying them. Of course, if one wants to update a service's signature (e.g., change an operation's name) this would require to regenerate and redeploy the service anew.

```
1  import javax.jws.*;
2  import javax.xml.ws.Holder;
3  import javax.jws.soap.*;
4  import javax.xml.ws.soap.Addressing;
5  import javax.annotation.Resource;

7  @WebService(name = "TestService", targetNamespace = "http://www.vitalab.
       tuwien.ac.at/Genesis2/generated/TestService")
8  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.
       LITERAL, parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
9  interface ITestService {

11   @WebMethod
12   @WebResult(name="response")
13   public void SayHello(@WebParam(name="name") String name, @WebParam(name="
         result") String result) throws Exception;
14 }

16 @WebService(name = "TestService", portName = "TestServicePort",
       targetNamespace = "http://www.vitalab.tuwien.ac.at/Genesis2/generated/
       TestService", endpointInterface = "ITestService")
17 class TestService implements ITestService {

19   @Resource
20   WebServiceContext wsc0;
21   WebService ws0=AModelElement.getMap(WebService).get("ad8f20ae-ea4e-4909-8
         c4b-d1c10668167e");

23   public TestService() throws Exception {
24     Closure depl0=ws0.getOnDeploy();
25     if (depl0) {
26       ws0.getPlugin().prepareHookClosure(wsc0,ws0,depl0).call();
```

```
27      }
28    }

30    public void onUndeploy() {
31      Closure undepl0=ws0.getOnUndeploy();
32      if (undepl0) {
33        ws0.getPlugin().prepareHookClosure(wsc0,ws0,undepl0).call();
34      }
35    }

37    @WebMethod
38    @WebResult(name="response")
39    public String SayHello(@WebParam(name="name") java.lang.String name) throws
          Exception {
40      def result0;
41      WsOperation op0=ws0.getOperation("SayHello");
42      groovy.lang.Closure be0=op0.getBehavior();
43      if (be0) {
44        be0=ws0.getPlugin().prepareOperationClosure(wsc0,ws0,op0);
45        ClosureDelegateDispatcher.setVariable(be0,"name",name);
46        ClosureDelegateDispatcher.setVariable(be0,"result",result);
47        ClosureDelegateDispatcher.registerConstant(be0,"args",[name,result]);
48        ClosureDelegateDispatcher.registerConstant(be0,"argNames",["name","
              result"]);
49        result0=be0.call();
50      }
51      return result0;
52    }
53  }
```

Listing 4.3: Generated Java/Groovy code of a Web service

In general, the whole generation procedure depends completely on the plugin's functional purpose and is not restricted by the framework. For instance, in contrast to the `WebService-Generator` that actually generates source code, the `CallInterceptor` translates the modeled functionality into Apache CXF Features [6] and binds them to service and client instances, the `ClientGenerator` simply implements programmable threads, and the `QOSEmulator` does not generate any deployable elements but works in the background by slowing down and hampering the interactions within the testbed.

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

4.2.4 Multicast Testbed Control

A drawback of G1 was that testbed manipulations had to be done in a point-to-point manner, updating one Web service at a time. This was an issue for controlling large-scale testbeds, such as the one used in the VRESCO project [123] consisting of up to 10000 services. To overcome this issue, G2 supports multicast-based manipulations. This feature is inspired by multicast network communication, where a single transmitted packet can reach an arbitrary large number of destination hosts with the help of replicating routers.

To provide similar efficiency, G2 uses filter closures which specify the destination of a change request and reduces the number of request messages. In detail, G2 applies the filter at the local testbed model to get the resulting set of designated elements and checks at which back-end hosts these are deployed. Then it wraps the change request, including the filter, and sends it to the involved hosts. Eventually, the hosts unwrap it, run the filter locally, and perform the changes on each matched testbed element. This way, G2 reduces the number of request messages to the number of involved back-end hosts, which significantly improves efficiency. The following snippet shows a sample multicast manipulation. It addresses Web services matching a namespace and performs a set of modifications on them, e.g., appending a new operation and setting model properties.

```
def newOp=wsoperation.create("newOperation")
newOp.behavior={  // do something ... }

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

Listing 4.4: Sample command for multicast-like update

4.3 Practical Application

We demonstrate the application of G2 in a scenario for generating QoS testbeds in order to give a better understanding of the presented concepts and also to give an impression about the intuitiveness of G2's script-based control. Our scenario covers the creation of a rather simple testbed for testing the QoS monitor [141] used in the VRESCO project. The monitor performs periodical checks for determining a Web service's execution time, latency, throughput, availabiltiy, robustness, and other QoS properties. Most of the monitoring is done in a non-intrusive manner, while for some checks local sensors need to be deployed at the service. For verifying the monitor's correct functionality, runtime tests must be performed on a testbed of generated Web services simulating QoS properties. Furthermore, the QoS properties must be controllable during test execution and the Web services must support the application of local sensors. For the sake of simplicity we omitted some testbed features, such as registration of generated services

at a broker, and replaced the usage of the `QoSEmulator`. Instead, we just simulate processing time and failure rate via simple delaying and throwing exceptions at the Web operations. However, for demonstration purposes, we have included some additional features, such as nested invocations, dynamic replacement of functionality, and generation of active clients. For setting up the testbed, we are using the plugins `WebServiceGenerator`, `WebServiceInvoker`, `CallInterceptor`, `ClientGenerator`, `SimpleRegistry`, and `DataPropagator`, which establish the model schema depicted in Figure 4.1. We divided the scenario into three parts: in the first step we generate the service-based testbed, then we generate clients invoking the testbed's services, and, finally, show how the running testbed can be altered at runtime.

Listing 4.5 covers the specification of the services. First, a set of 10 back-end hosts (which must be already running at the specified location) is referenced and the service's message types are imported from an XSD file. In Line 11, the `DataPropagator` plugin is invoked, via its alias `prop`, to bind a global function/closure to the shared runtime environment. The function simply chooses a random element out of a list and will be used to deploy components at random back-end hosts. The testbed itself comprises 100 simple worker services and, in addition, 20 delegators that dispatch invocations to the workers. In Lines 19 to 35, the worker services are built. For each we declare variables for controlling the simulation of QoS (`delay` and `failureRate`) , and add a tag for distinction (`"worker"`). For the worker's Web service operation `Process` we specify its I/O message types and customize its behavior with simple code for simulating delay and failure rate, controlled via the service's variables. Later, the composite delegator services are created in a similar manner (Lines 38-65), but contain nested service invocations and a user-defined customization encapsulated in the `processError()` function. Furthermore, a header argument is specified (`neededResults: hdr(int)`), which means that it is declared as part of the SOAP header instead of the body. In Line 52 the `SimpleRegistry` is queried to get a list of references to worker services. Of these random ones are picked and invoked (Line 55) in sequence, until the required number of correct responses has been reached. On faults, the customizable error handling routine named `processError()` is called. Eventually, the delegator service returns a list of responses. At the end of the script, the testbed is generated by deploying the modeled Web services on random hosts.

```
1  // reference 10 back-end hosts
2  1.upto(10) { n->
3    host.create("192.168.1.$n",8080)
4  }
5  hostlist=[]

7  // load message type definitions from XSD file
8  prop.inType=datatype.create("/path/to/sample.xsd","test")
9  prop.outType=inType // same input and output type

11 prop.randomListItem={ lst->
12   lst=lst.flatten() // workaround for nested lists
13   pos=new Random().nextInt(lst.size())
14   return lst[pos]
15 }

17 def serviceList=webservice.build {
```

```
19   1.upto(20) { i-> //create Service1 .. Service100
20   "Service$i"() {

22     delay=0
23     failureRate=0.0
24     tags=["worker"]

26     //Web service operation "Process"
27     Process(input: inType, response: outType) {

29       Thread.sleep(delay)
30       if (new Random().nextFloat()<failureRate) {
31         throw new Exception("sorry!")
32       }
33       return outType.newInstance()
34     }
35   }
36 }

38   1.upto(5) { i-> //create 20 delegator services
39   "CompositeService$i"() {

41     tags=["delegator","composite"]

43     processError={} //initially empty function

45     //Web service operation "Delegate"
46     Delegate(input: inType, neededResults: hdr(int), response: arrayOf(
           outType)) {

48       def gotResults=0
49       def result=[]

51       while (gotResults<neededResults) {
52         def refs=registry.get{"worker" in it.tags}
53         def ref=randomListItem(refs)
54         try {
55           def resp=ref.Process(input).response
56           result+=outType.newInstance().assign(resp)
57           gotResults++
58         } catch (e) {
59           processError(e)
60         }
61       }
62       return result
63     }
64   }
65 }
66 }

68 serviceList.each { s-> //deploy at random hosts
69   s.deployAt(randomListItem(hostlist))
```

```
70 }
```

Listing 4.5: 'Generation of Web services for task delegation example'

Though, in this state the testbed contains only passive services awaiting invocations. In order to make it "alive", by generating activity, Listing 4.6 specifies and deploys clients which invoke random delegator services in 5 second intervals.

```
1  def initClient=client.create()

3  initClient.run=true //boolean flag 'run'

5  initClient.code={ //client code as closure
6    while (run) {
7      Thread.sleep(5000) //every 5 seconds
8      def refs=registry.get{"delegator" in it.tags}
9      def r=randomListItem(refs) //pick random
10     def arg=inType.newInstance()
11     println r.Delegate(arg, 3) //initiate delegation
12   }
13 }

15 initClient.deployAt(hostlist) //run everywhere
```

Listing 4.6: 'Generation of clients invoking delegator Web services'

Finally, Listing 4.7 demonstrates how running testbeds can be altered at runtime. At first, a call interceptor is created, which can be, for instance, used to place the QoS sensors. In our example we simply print out the message content to `stdout`. We make use of G2's multicast updates and enhance all delegator services by appending the interceptor to the service model. In the same request we replace the (formerly empty) `processError()` routine and instruct the services to report errors to a 3rd-party Web service. At the back-end, the `WebServiceGenerator` plug-ins will detect the change request and automatically adapt the addressed services. Furthermore, by making use of G2's immediate synchronization of models with running testbed instances, the simulation of QoS is altered on the fly by changing the corresponding parameter variables of worker services in a random manner. In the end, the clients are shut down by changing their `run` flag.

```
1  def pi=callinterceptor.create()
2  pi.hooks=[in:"RECEIVE", out :"SEND"] //where to bind
3  pi.code={ ctx->
4    println "MESSAGE = ${ctx.soapMsg}" //just print the msg object
5  }

7  webservice(i:pi) { s->
8    "delegator" in s.tags
9  } { s->
10   s.interceptors+=i //attach to author services
11   s.processError= { e->
12     def url="http://somehost.com/reportError?WSDL"
13     def reportWs=wsreference.create(url)
```

```
14      reportWs.Report(my.webservice.name, e.message)
15    }
16  }

18  int cycles=100

20  while (--cycles>0) {
21    Thread.sleep(3000) //every 3 seconds

23    def workers=webservice.get{"worker" in it.tags}
24    def w=randomListItem(workers)
25    w.delay=new Random().nextInt(5*1000) //0 - 5sec
26    w.failureRate=new Random().nextFloat() //0.0 - 1.0
27  }

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

Listing 4.7: 'On-the-fly manipulation/extension of running testbed'

In this scenario we have tried to cover as many key features of G2 as possible, to demonstrate the simplicity of our scripting interface. We have used builders to create nested model structures (`WebService`→`WsOperation`→`DataType`), designed Web services and clients with parameterizable behavior, customized behavior with closures, applied plugins (e.g., call interceptors and service invokers), performed a multicast manipulation request, and steered the running testbed via parameters. The generated testbed consists of interconnected Web services and active clients calling them. To facilitate proper testing of the QoS monitor [141], it would require to simulate not only processing time and fault rate, but also scalability, throughput, and other properties which we have skipped in this chapter but will be explained in detail in the next one. In any case, we believe that the presented scenario helps to understand how G2 is used and gives a good impression about its capabilities.

4.4 Discussion of Shortcomings and Solutions

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

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

plugins which generate basic SOA artifacts, such as services, clients, and registries and, therefore, provide all the functionality one expects of a testbed generator.

Moreover, in the introduction of this chapter we said that SOA comprises more than just Web services, but also clients, service buses, mediators, workflow engines, etc. But looking at the list of plugins which we developed (see Section 4.2.3), it becomes evident that they do not cover all these components. This is partially true, as this chapter presents the current state of our work at the time of publishing G2. Meanwhile, several extensions and other component generator plugins have been developed that are presented in the following chapters.

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

CHAPTER 5

Generating Fault Injection Testbeds for SOA

Published in:

Programmable Fault Injection Testbeds for Complex SOA.

Juszczyk L., Dustdar S. (2010).

8th International Conference on Service-Oriented Computing (ICSOC'10), 07. - 10. December 2010, San Francisco, USA.

and in

Testbeds for Emulating Dependability Issues of Mobile Web Services.

Juszczyk L., Dustdar S. (2010)

1st International Workshop on Engineering Mobile Service Oriented Systems (EMSOS). 6th IEEE World Congress on Services (SERVICES'10), 5. - 10. July 2010, Miami, USA.

Outline. The more complex a service-oriented system gets, the more error-prone it becomes. Faults can happen at every SOA component, at various levels, and have severe effects on the whole SOA system. As in any distributed system, fault handling mechanisms can mitigate the effects of faults and guarantee a certain level of availability. And these mechanisms must be tested too. In this chapter we introduce techniques for generating testbeds which expose faulty behavior, also referred to as fault injection testbeds. By applying these, engineers can evaluate the fault handling routines of their systems in various scenarios in order to develop more robust SOAs.

5.1 Motivation

Modern SOA systems comprise stand-alone and composite Web services, clients, brokers and registries, workflow engines, monitors, governance systems, message dispatchers, service buses, and other components. 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. 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, however, 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 [140] and [116], faults do also 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. 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 [92, 98, 129] and is out of the scope of our research. 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 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. Again, 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. 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 [131]. 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.

In the current chapter we apply G2 and extend it with plugins 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.*

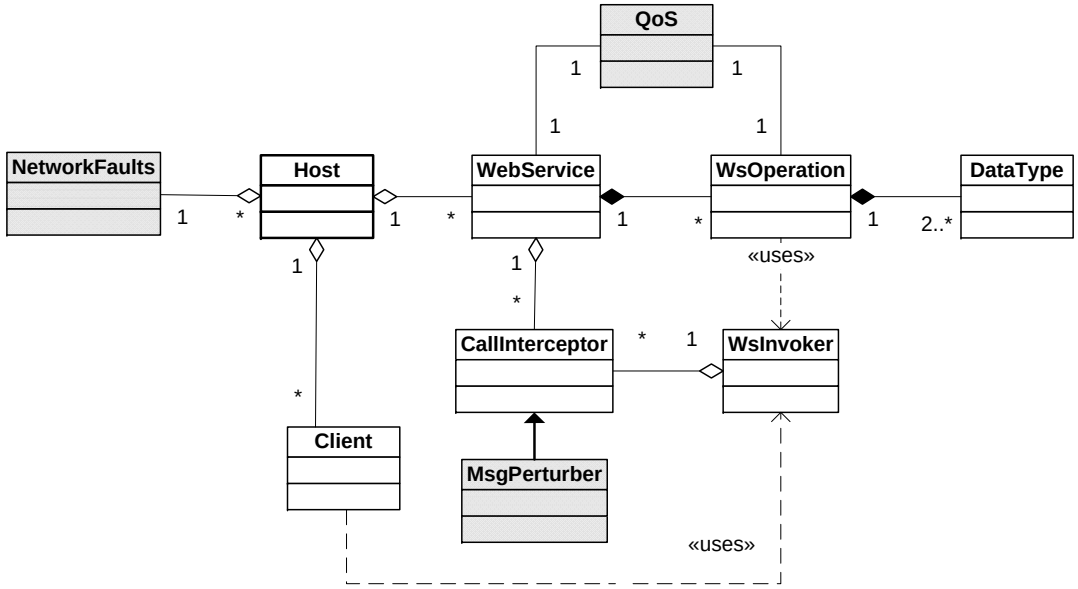


Figure 5.1: Extended G2 testbed model schema for fault injection

5.2 Programmable Multi-level Fault Injection

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 [72]. 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 into these. 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. In 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 5.1 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.

5.2.1 Message Faults

SOAP Web services are using WSDL documents [56] 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*, e.g., an address String in a field where a name is expected),
- which violate the message's XML schema definition [64] (= *high-level syntax errors*, e.g., surplus XML attributes in a tag),
- or which even do not represent a correct XML document at all (= *low-level syntax errors*, e.g., no closing tags present).

Depending on the degree of corruption, fault handling mechanisms can be applied to allow the integration of faulty components into a SOA [131]. 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 [7]. The interceptors provide means to intervene at all phases within the chain of modules that process SOAP messages. Taking the processing of incoming request messages for example, it starts with reading in the byte stream and building an XML tree structure from this. Next, the pure XML is transformed into objects which represent SOAP messages, including headers and body elements. Then, the SOAP elements are unmarshalled into Java objects representing the message types. As last step, the SOAP call is passed to the corresponding method of the class/object implementing the Web service. Obviously, for outgoing messages this chain is being processed in a reserve manner.

For injecting faults, it is necessary to place a module into the chain, which corrupts the message. Even though all corruption could be done in the very first phase of an incoming chain,

Phase	**Functions**
RECEIVE	Transport level processing
(PRE/USER/POST)_STREAM	Stream level processing/transformations
READ	This is where header reading typically occurs
(PRE/USER/POST)_PROTOCOL	Protocol processing, such as JAX-WS SOAP handlers
UNMARSHAL	Unmarshalling of the request
(PRE/USER/POST)_LOGICAL	Processing of the umarshalled request
PRE_INVOKE	Pre invocation actions
INVOKE	Invocation of the service
POST_INVOKE	Invocation of the outgoing chain if there is one

Table 5.1: Ingoing SOAP processing phases in Apache CXF.

Phase	**Functions**
SETUP	Any set up for the following phases
(PRE/USER/POST)_LOGICAL	Processing of objects about to marshalled
PREPARE_SEND	Opening of the connection
PRE_STREAM	
PRE_PROTOCOL	Misc protocol actions
WRITE	Writing of the protocol message, e.g., the SOAP Envelope
MARSHAL	Marshalling of the objects
(USER/POST)_PROTOCOL	Processing of the protocol message
(USER/POST)_STREAM	Processing of the byte level message
SEND	Final sending of message and closing of transport stream

Table 5.2: Outgoing SOAP processing phases in Apache CXF.

or the last of an outgoing one, where the message is available as a byte stream or string, it makes more sense to place the corrupting code deeper in the chain. Tables 5.1 and 5.2 (taken from [7]) display the individual phases which are supported by CXF and where SOAP processing modules can be placed. For semantic-level corruption we place the code into the phases "PRE_LOGICAL" and "MARSHAL" as it gives us access to the Java objects representing the messages data, which can be altered. For syntax errors we apply the corruption at the phases "RECEIVE" and "PRE_STREAM" and either perturb the byte stream directly (for low-level syntax) or at the parsed XML tree (for high-level syntax errors).

We have built the perturbation mechanism upon the visitor pattern [132]. Perturbation code, wrapped in visitor objects, is propagated recursively along the XML tree and/or the unmarshalled Java objects and has full read/write access for performing arbitrary 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 code shows an interceptor that is programmed to assign random values to all integer fields named Price.

Moreover, it deletes all postcodes for matching addresses.

```
def valuePert = msgperturber.create("args") //pert. data values

valuePert.code = { it ->
    if (it.name=="Price" && it.type==int) { //get by name and type
        it.value*=new Random().nextInt()
    } else if (it.name=="Address" && it.value.country=="Austria") { //by val
        it.value.postcode=null
    }
}
```

For high-level syntax manipulation (next code snippet), 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 sample code, the visitor is looking for nodes which have children named `country` and appends a new child which violates the message's XSD definition.

```
def xmlPert = msgperturber.create("dom") //pert. XML structure

xmlPert.code = { node ->
    if (node.children.any { c-> c.name=="Country" }) {
        Node newChild = node.appendNode("NotInXSD")
        newChild.attributes.someAtt="123"
    }
}
```

The result of the last sample 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.

```
def bytePert = msgperturber.create("bytes") //pert. msg bytes

bytePert.code = { str ->
    str.replaceFirst("</","<") //remove closing tag from XML doc
}
```

Finally, the interceptors must be attached to a Web service model in order to be deployed at the generated service instance and start injecting faults into its request and response messages.

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

To illustrate the effect of message corruption consider the SOAP message examples in Listings 5.5 and 5.6. The first one shows the correct message before the corruption, the second one shows the altered result. The effects reside in Line 5, where the price has been changed, in Lines 8-13 where the post code has been removed and a new element (`NotInXSD`) has been attached. Moreover, in Line 4 the closing tag has been changed replaced with an opening one, which renders the XML document invalid.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
<soap:Body xmlns:ns="http://bookorder.com/ws">
  <ns:Order>
    <ns:Product>Book ABC</ns:Product>
    <ns:Price>30</ns:Price>
    <ns:Customer>
      <ns:Name>Lukasz Juszczyk</ns:Name>
      <ns:Address>
        <ns:Street>XYZ-Strasse 315</ns:Street>
        <ns:Postcode>1234</ns:Postcode>
        <ns:City>Vienna</ns:City>
        <ns:Country>Austria</ns:Country>
      </ns:Address>
    </ns:Customer>
  </ns:Order>
</soap:Body>
</soap:Envelope>
```

Listing 5.5: Original SOAP Message

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
<soap:Body xmlns:ns="http://bookorder.com/ws">
  <ns:Order>
    <ns:Product>Book ABC<ns:Product>
    <ns:Price>180</ns:Price>
    <ns:Customer>
      <ns:Name>Lukasz Juszczyk</ns:Name>
      <ns:Address>
        <ns:Street>XYZ-Strasse 315</ns:Street>
        <ns:City>Vienna</ns:City>
        <ns:Country>Austria</ns:Country>
        <ns:NotInXSD someAtt="123"/>
      </ns:Address>
    </ns:Customer>
  </ns:Order>
</soap:Body>
</soap:Envelope>
```

Listing 5.6: Corrupted SOAP Message

5.2.2 Service Execution Faults

Service execution faults usually result in degraded Quality of Service (QoS) [122]. 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 [141], comprising *processing time, scalability, throughput, error rate* 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 at the back-end and intercepts their invocations in order to simulate QoS. The `QoSEmulator` emulates service execution faults as follows:

- *Availability* is emulated by deploying and undeploying of Web services, in given intervals, according to a probability value. For instance, an availability of 0.9 means that in a regular interval (e.g., once a minute) the service will be deployed with a probability of 90% and will be unavailable with 10% probability.
- *Processing time, throughput,* and *scalability* only slow down the processing of a service request and do not cause any faults in the strict sense. Usually, these three qualities cohere with each others, in terms of that each one has an effect on the others. For instance, a weak scalability or throughput affects the general processing time, and vice versa. We have chosen a rather simple model for specifying the effects of these qualities, which, however, can be replaced with a more complex one easily. We specify processing time as the minimum time the service needs to process an invocation. Parallel invocations are all processed at the defined processing time, unless their number exceeds the throughput parameter. In that case, the processing time will be increased according to the scalability penalty, e.g., a penalty of 2.0 means that the processing time will grow quadratically. Internally, delays are simulated by applying an interceptor which, however, does not alter the SOAP message but just delays the processing.
- *Error rate* is quite simple and specifies the probability that the service's response message will be replaced with a SOAP fault, done in an interceptor placed in the outgoing chain.

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 closures, 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 define the processing time according to a statistical distribution function, supported via the *Java Distribution Functions* library (distlib) [34].

The following listings contain 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.

```
def svcQos = qos.create()

svcQos.availability = {
  if (new Date().getHours() < 8) { //from 0 to 7 AM
    return 99/100 //set high availability of 99%
  } else {
    return 90/100 //otherwise, set lower availability rate
  }
}
```

For the service operation, the processing time is derived from a beta distribution (alias `dist`) while throughput and error rate are assigned with constant values. The effect of assigning a closure to the processing time is, that at every invocation of the service/operation, the `QoSEmulator` plugin checks the parameter by executing the closure and delays the execution according to the return value.

```
def opQos = qos.create()

opQos.processingTime = {
  dist.beta.random(5000,1,null) //beta distribution
}
opQos.throughput = 10/60 //restrict to 10 invocations per minute
opQos.errorRate = 15/100 //15% of invocations will fail with exceptions
```

At the end, the models are bound to the service and its operations.

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

service.operations.grep { o->
  o.returnType!=null //and to all 2-way operations (= returning something)
}.each {
  o.qos=opQoS
}
```

5.2.3 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* [48] and *Web Service Dynamic Discovery* [58], 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) [39] (with *netem* module [41]) 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. Similar to the previously presented plugins, engineers create fault models but now they 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.

The presented next listings 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.

```
def nf = networkfaults.create()

nf.loss.value = 2/100 //2% packet loss
nf.duplicate.value = 1/100 //1% packet duplication
nf.delay.value = 100 //100ms
nf.delay.variation = 20 //20ms of variation
nf.delay.distribution = "normal" //normal distribution

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

The modelled faults are translated into tc commands that are applied on the back-end host. The first block of commands sets up the necessary handler chains of tc for incoming and outgoing traffic and binds them to port 8182, which is the port at which the back-end instances are hosted. The second block contains the actual packet corruption commands which are derived from the model.

```
tc qdisc add dev lo handle 1: root htb
tc class add dev lo parent 1: classid 1:1 htb rate 1000Mbps
tc qdisc add dev lo parent 1:1 handle 10: netem
tc filter add dev lo protocol ip prio 1 u32 match ip sport 8182 0xffff flowid
    1:1
tc qdisc add dev lo ingress
tc filter add dev lo parent ffff: protocol ip u32 match ip dport 8182 0xffff
   flowid 1:1 action mirred egress redirect dev ifb0
tc qdisc add dev ifb0 root netem

tc qdisc change dev lo parent 1:1 handle 10: netem duplicate 1.0% loss 2.0%
   delay 100ms 20ms distribution normal
tc qdisc change dev ifb0 root netem duplicate 1.0% loss 2.0% delay 100ms 20ms
    distribution normal
```

And, eventually, when the fault injection must be stopped, the tc queues are deleted.

```
tc qdisc del dev lo handle 1: root htb
tc qdisc del dev lo ingress
tc qdisc del dev ifb0 root netem
```

Unfortunately, due to the static parameterization of tc, it is not possible to provide similar programmability of fault models via closures, as it is the case for the service execution faults. Though, tc and netem provide a powerful facility to analyze the effects of low-level network faults on SOA communication.

5.3 Practical Application

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 mecha-

nisms. 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 also offer mechanisms for importing external specifications of SOAs and their components into G2 models (e.g., from BPEL process definitions [17] and WSDL documents, as presented in the next chapter). 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.

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. 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 have not developed any tool support so far. 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. We regarded this problem as out of scope of our current research - and as possible future work - and we concentrated on how to generate the testbeds.

In Section 4.2.4 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 5.13 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.

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

host { h-> //filter
    h.location =~ /192.168.1./
} { h-> //command
    netFaultModel.attachTo(h)
}
```

Listing 5.13: 'Injecting faults to hosts and Web services'

CHAPTER 6

Towards Automation of Testbed Generation

Published in:

Automating the Generation of Web Service Testbeds using AOP

Juszczyk L., Dustdar S. (2011).

9th European Conference on Web Services (ECOWS'11), 14. - 16. September 2011, Lugano, Switzerland.

Outline. In the last chapters we have shown how the concepts of GENESIS have evolved and how the framework can be extended, e.g., in order to generate fault injection testbeds. In these works, however, the whole specification of the testbeds, including composition, structure, and functional behavior, had to be provided by the engineer/testers via the scripting language(s). The current chapter presents our approach towards more automation of the specification process. We apply techniques of aspect-oriented programming (AOP) in order to inspect Java-based SOA systems at runtime, to detect invocations of remote Web services, and to generated replica services for these automatically. Engineers are only required to specify customizations to the replicas. As a result, we can automate parts of the the testbed generation and accelerate the whole process.

6.1 Motivation

SOA's principles are grounded on modularization of functionality into services and on providing these to clients for on-demand usage. Of course, the idea of software modularization was not invented with SOA but has been applied since decades. But what SOA propagates is not only to

compose systems out of a set of modules, referred to as services, but also to integrate external services (e.g., from other companies/organizations) into a system. This is supported by the open character of Web service-based SOA, that uses open standards for communication (SOAP [47]), interface descriptions (WSDL [56]), and for the numerous WS-* extensions which are public [166]. The benefits are obvious: faster software development due to reuse and the ability to choose dynamically among available services depending on their quality, just to list to most prominent ones. External services can be integrated easily by analyzing their WSDL descriptions, making sure that client and service agree on communication details, such as available operations and exchanged message types.

Though, in spite of the benefits derived from flexible reuse, engineers of outsourcing systems are facing several problems. Integration of functionality provided by external services turns an SOA vulnerable, as the services may become unavailable or may suffer from degraded quality of service. Hence, it is a potential risk for the dependability of an outsourcing SOA system. To address this issue, such systems should undergo rigorous tests, in order to make sure that they are able to handle faults properly and that there will be no bad surprises once they are deployed. Unfortunately, the whole testing procedure becomes problematic as invocations of external services often cost money or because their providers have policies which restrict trial invocations.

In our previous chapters we argued that this dilemma can be mitigated by using testbeds that emulate external SOA infrastructures. Engineers had to write specification scripts that describe the environment to be emulated, including the topology as well as functional behavior of it. In the current chapter we simplify this process. We evolve our approach towards an automated generation of testbeds and reduce the input of SOA engineers significantly. We apply AspectJ [15], an AOP extension for the Java VM, for inspecting Java-based SOA systems at runtime in order to detect invocations of external Web services. On detection, our system analyzes the remote services, generates replicas of these, and deploys them within a testbed. Eventually, the invocation is redirected to the replica transparently, leaving the original service untouched. The result of this procedure is that external SOA infrastructures can be emulated on-the-fly and their replicas act as a testbed for the engineer. Compared to our previous work, the current paper brings forward the concept of testbed generation towards more automation, requiring less specification input from the engineer. It improves the practical applicability and accelerates the testing process.

6.2 Automated Generation of SOA Sandboxes

In a nutshell, our approach is based on monitoring running Java-based SOA systems, detecting calls of external Web services, and redirecting the calls to automatically generated replicas. Figure 6.1 depicts our approach which consists of the following steps:

1. Detection of Web service calls in the SOA system, by intercepting WSDL retrieval code in the Java runtime.

2. Analysis of the WSDL document and generation of a replica model at the front-end.

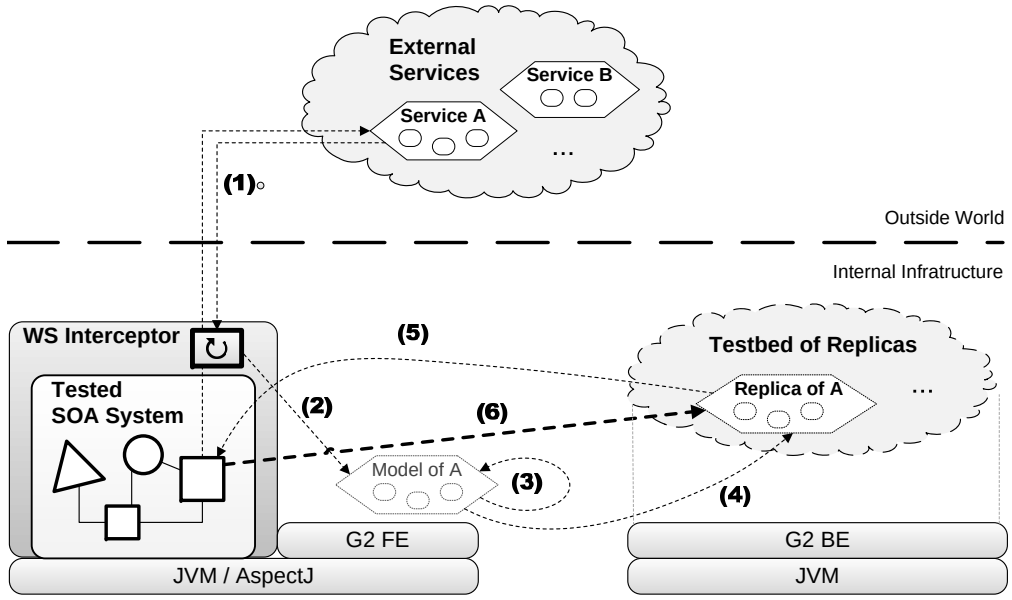


Figure 6.1: Interception of Web service calls and generation of replicas

3. Rule-based customizations of the replica model.

4. Deployment of the replica instance, from the model, at the testbed infrastructure (back-end).

5. Forwarding of the replica WSDL document, instead of the original WSDL, to the calling SOA system.

6. Actual Web service invocation, redirected to the replica.

In the next sections we explain the concepts behind each step and give an overview about the extendability and programmability of the testbeds.

6.2.1 AOP-based Interception of Web service Invocations

Aspect-oriented programming (AOP) is a paradigm which aims at increasing software modularity by allowing the separation of cross-cutting concerns. In particular, AOP allows developers to alter the behavior of a system (in terms of enhancing but also replacing functionality) at runtime by specifying pointcuts and join points (intercepted points/functions in a program) and executing advices (new/additional behavior) on them. For example, a developer can define aspects for enhancing a program with logging functionality or replace certain parts of it in order to observe the effects. Up to date, AOP has gained high popularity among software engineers and numerous programming languages provide support for it. However in this work we have concentrated on

Java and on its AOP extension called AspectJ [15], due to Java's importance for Web (service) engineering.

In our approach we apply AOP for intercepting Web service invocations, create replica services in a testbed, and redirect the invocations to these, as depicted in Figure 6.1. To be precise, we do not wait until the invocation is being executed and the SOAP request message is ready to be sent but we intercept this procedure earlier, namely at the retrieval of WSDL documents, as this is the first step of a typical invocation procedure [124]: WSDLs are retrieved from a registry and then passed to a client generator which creates the corresponding invocation stubs. Here we intervene in the program flow and create the replica at the very first invocation of the service. Before the client generator processes the WSDL we analyze it in order to create a replica of the service in a dedicated testbed infrastructure (replica creation is explained in the next section). The next step is to replace transparently the original WSDL with the one of the replica, in order to make the client generator create stubs which point to the new endpoint and, eventually, to direct all invocations to it. This way, the original service is only contacted for retrieving its description but all communication is actually done with its replica in the testbed.

The interception of WSDL retrieval is realized via aspects which detect calls of the retrieval routines in the SOA system and by altering their execution. Depending on which Web service framework the system is using (e.g., Apache Axis 2 [3], Apache CXF [4], or GlassFish [21]) different aspects must be applied in order to match the corresponding API functions. The following code snippet shows a simple aspect which covers the popular WSDL4J [55] library/framework, used by various workflow engines. At first a pointcut is defined which catches calls of the method `WSDLReader.readWSDL(String)`. If this pointcut is matched at runtime, the following advice will be executed. It passes the URI to the `Sandbox` utility class that initiates the replication of the service in the background and returns the URI to the replica's WSDL. Finally, the intercepted method is called, however, the URI of the original WSDL is replaced with the one of the replica.

```
public aspect InterceptorAspect {

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

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

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

Listing 6.1: 'Declaration of WSDL interceptor aspect'

We take the next Java code snippet, which reads a WSDL and creates a client stub, as an example for demonstrating the effects of such an aspect. The code basically comprises three steps: the retrieval of the service's URI, e.g., from a registry/broker, the retrieval of the WSDL located at the URI, and the generation of a client stub from this WSDL.

```
String uri= ... // get from somewhere
WSDLReader wr=new WSDLReader();
Definition wsdl=wr.readWSDL(uri); // <-- will be intercepted by aspect
Client client=generateFrom(wsdl);
```

On applying the `InterceptorAscpect`, AspectJ detects the calling of the `readWSDL()` method and changes the code to the following.

```
String uri ... // get from somewhere
WSDLReader wr=new WSDLReader();
// <changes caused by aspect>
String repURI=Sandbox.createReplica(uri);
Definition wsdl=wr.readWSDL(repURI);
// </changes caused by aspect>
Client client=generateFrom(wsdl);
```

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

6.2.2 On-the-fly Generation of Service Replicas

After a WSDL retrieval has been intercepted, the process of replicating the described Web service is initiated, which comprises the following three main steps:

1. the WSDL of the remote Web service is analyzed and a basic model of a replica service is created,
2. the model is subject to user-defined customizations, and
3. the final model is transferred to a back-end host and a Web service instance is generated in the testbed.

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

6.2.2.1 Creation of a Replica Model

For creating the model of a replica service, our tool retrieves the original service's WSDL document in order to analyze the interface and to clone it in the model. Even though the analysis of WSDL documents could be done in a "raw" manner by processing the document directly, for instance by using the WSDL4J library [55], we apply instead the `wsimport` utility of JAX-WS [32] for convenience. `wsimport` takes WSDL's as input and generates corresponding Java stubs, imports referenced XML schemas automatically, checks for WS-I [60] compatibility, and performs various other necessary steps which would otherwise have to be done manually. Our tool takes the generated stubs and compiles them which provides us a binary representation of the service's interface. This is then analyzed via object reflection techniques plus by checking the corresponding Java annotations (`@WebService, @WebMethod, @WebParam`, etc.). Taking for example, the WSDL document in Listing 3.3 on Page 31, the generated stub source would look as follows.

```
package at.ac.tuwien.vitalab.generatedservice.bookservice;

import javax.jws.*;
import javax.xml.ws.*;
// import ...

@WebService(name = "BookService", targetNamespace = "http://vitalab.tuwien.ac
    .at/generatedService/BookService")
@XmlSeeAlso({
  ObjectFactory.class
})

public interface BookService {

  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(localName = "getISBN",
    targetNamespace = "http://vitalab.tuwien.ac.at/generatedService/
        BookService",
    className = "at.ac.tuwien.vitalab.generatedservice.bookservice.GetISBN")
  @ResponseWrapper(localName = "getISBNResponse",
    targetNamespace = "http://vitalab.tuwien.ac.at/generatedService/
        BookService",
    className = "at.ac.tuwien.vitalab.generatedservice.bookservice.
        GetISBNResponse")
  public String getISBN(@WebParam(name = "bookName", targetNamespace = "")
      String bookName) throws Exception_Exception;

}
```

Listing 6.4: JAX-WS Web service interface generated from WSDL

By analyzing the interface and the attached annotations, we can easily determine the service's operation signatures, message types, binding information, and all the other required data for modelling a replica, and we translate this data into the model representation. The next step is to perform user-defined customizations to the model.

6.2.2.2 Customization, Extensibility, and Programmability

The result of the first step is a model of a replica service which clones the original service's interface, however, does not provide any (proper) functionality. On an invocation it delivers response messages which are syntactically correct, according to the XML schema definition (XSD) [64] in the WSDL document, but simply fills these messages with randomized data, which makes them semantically meaningless. Therefore, at this stage the replica can be regarded as a pure dummy or mock-up service. For some test cases mock-ups are perfectly sufficient, for instance if the engineer wants to determine how many parallel service invocations his/her tested SOA system can handle, regardless of the response message content. But if the test cases become more sophisticated and message content does matter, or if the services must expose certain functional or non-functional behavior, then the replica models must be customized in order to meet these requirements and to expose the desired behavior. This is a part of the replication process that we cannot automate, as requirements vary with every tested SOA system and the purpose of the particular test case. Furthermore, it is impossible to replicate a remote service's functionality automatically and to provide 100% realism, as explained later in Section 6.3.1. But what we can do is to support engineers in performing the necessary customizations in order to replicate the behavior *as good as possible and/or required for the tests*! For this purpose we make use of G2's extensibility via plugins and, again, apply Groovy scripts for specifying rules for how generated replica services are customized.

The following Listing demonstrates two sample rules for customizing replica services with fault injection (as shown in previous chapter) plus for assigning functional behavior to the service's operations. The rules are wrapped in a Groovy closure which takes the service model as parameter (`svc`). The first rule matches services by their namespace (Line 4) and augments them with a QoS model that simulates changing availability. In Lines 7-17 the QoS model is instantiated, bound to the service, and the availability behavior is programmed. The second rule matches services by their declared name plus by whether they contain a certain operation. In case of a match, the operation's behavior is programmed to return prepared responses from a repository containing recorded data for replaying.

```
1  serviceCustomizer = { svc ->
2
3    // customize service which has "infosys" in its NS
4    if (svc.namespace =~ /infosys.ac.at/) {
5
6      // instantiate QOS model and attach to service
7      def svcQos = qos.create()
8      svc.qos = scvQoS
9
10     // program availability behavior model
11     svcQos.availability = {
12       if (new Date().getHours() < 8) { // till 8 AM
13         return 99/100 //set high availability of 99%
14       } else {
15         return 90/100 //otherwise, set lower rate
16       }
17     }
```

```
18    }

20    // customize service with a particular operation
21    if (svc.name == "CustomerSVC" && "GetCustomerData" in svc.operations.name)
          {

23      // program behavior of operation via closure
24      svc.getOperation("GetCustomerData").behavior = {
25        def response = rrRepo.get(request)
26        return response
27      }
28    }

30  }
```

Listing 6.5: 'Customization Rule for Replica Services'

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

6.2.2.3 Generating Web service Instances in the Back-end

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

```
hostList = []

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

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

Listing 6.6: 'Host Picker Rules'

The process of generating Web service instances out of the models comprises the serialization of the model, transferring it to the G2 instance at the choses back-end host, and, finally, the translation of the model into a running and deployable Web service. This procedure has been explained in Chapter 4.

All in all, the result of the whole replication process is a running Web service that clones the original service's interface, behaves according to the engineer's customizations, and can be used for testing purposes as a replacement of the original service.

6.3 Evaluation

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

In our evaluation we have applied our approach on intercepting dynamic binding and invocation of Web services. We agree with [113] that dynamic binding combined with message-based interactions is the proper way to implement SOA communication. SOA systems should be able to adapt to their environment and redirect invocations to "better suited" Web services at runtime, instead of being tightly bound to particular services. This requires that the invocation stubs, that handle the communication with the remote services, are not hard-coded, but are being generated dynamically out of the services' WSDL descriptions. In our evaluation we have used three different Web service frameworks/libraries which are capable of dynamic binding:

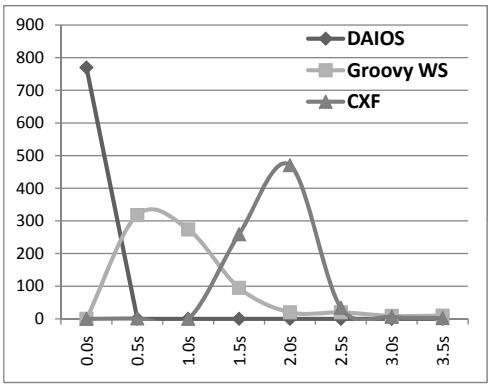
1. *Apache CXF* [5] is a rich framework, supporting SOAP [47], ReST [97], WS-* extenstions, etc. Implements the JAX-WS API [32] for Web service development.

2. The *Groovy SOAP Module* [27] provides SOAP support for the Groovy language [24]. Strongly focused on simplicity and usage convenience, less on feature support.

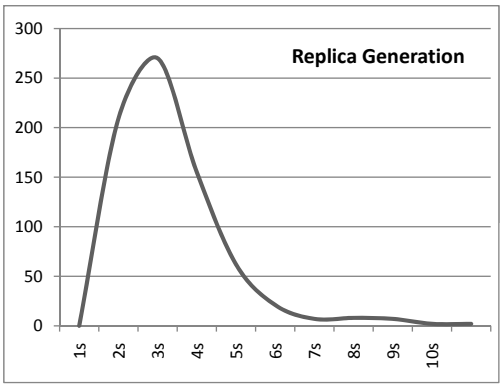
3. *DAIOS* [113] aims at supporting truly dynamic interactions between clients and services. Provides composition of request messages automatically via similarity metrics between request data and WSDL contract, improving loose coupling this way.

For testing these client generators, we took the QWS dataset [69] that is basically a collection of WSDL documents of public SOAP Web services. In total it contains 2505 WSDLs, however, we were not able to use all of these for our purposes. First of all, 31% of the documents are not WS-I-compatible as they use RPC/encoded bindings or other WSDL styles which have been regarded as deprecated [60]. Furthermore, 15% of the remaining WSDLs were corrupted, for instance by referencing not existing XSD definitions or by having an invalid document structure. Moreover, we also removed 12% of valid WSDLs as they were referencing remote artifacts which had to be downloaded with sometimes significant latency which delayed and, therefore, distorted the whole processing time of the client generators. The remaining WSDLs were considered as suitable for executing our evaluation.

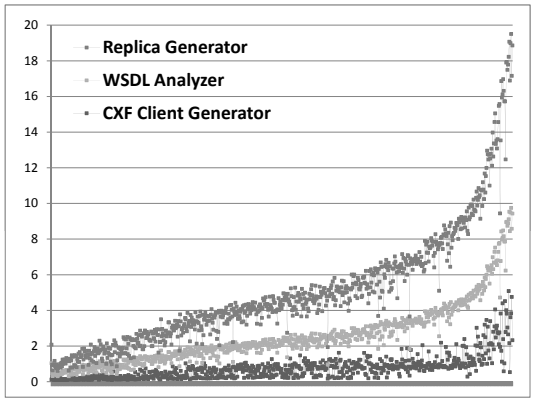
To determine the performance intrusiveness, we calculated how fast a client generator actually generates the corresponding stub code and how much this procedure is delayed if we apply



(a) Processing time distribution of tested client generators



(b) Processing time distribution of Web service replica generator



(c) Accumulated processing time of intercepted client generator

the testbed generator aspects. We did not measure the time for eventually invoking the services on purpose, for two reasons: 1) the main load is caused during the analysis of the WSDL and the generation of the stubs, and only minimal load is caused during the actual invocation, and 2) the invocation is mainly delayed by the processing time of the remote Web service (referred to as QoS) and does not depend on the client generator at all.

Figure 6.2a visualizes the distribution of processing times for the three testes client generators, while Figure 6.2b shows the processing time of the replica generator for comparison. The performance of the client generators differs significantly, due to their level of (pre-)processing of Web service calls. For instance, DAIOS does not generate and compile code into Java classes but simply analyzes the content of the WSDL document. This makes DAIOS, by far, the fastest client generator. Groovy SOAP and Apache CXF, however, take the WSDL document as input for generating stub classes and for translating message types into Java classes. Even though this provides several benefits, it comes at the cost of performance. For the set of WSDL document of the QWS dataset we measured an average processing time of 12 msec for DAIOS, 871 msec

for Groovy SOAP, and 1622 msec for CXF[1]. For the replica generator we measured an average processing time of 4617 msec which is significantly slower than the client generators. As a consequence, if our testbed generator approach is applied on a SOA system that uses dynamic binding, the following delays will occur for the generation of clients: DAIOS-based calls would be slowed down by a factor of 384 - which is, however, exceptional as DAIOS performs only restricted processing of the WSDL -, for Groovy SOAP the slow down factor would by 6.3, and for CXF it would be 3.8. Of course, this delay happens only *once*, namely when the client stub gets generated during the first invocation of the service. Each subsequent call will not be delayed, as the replica needs to be created only once.

For a better visualization of the delay caused by replication, we measured the performance of the individual steps of a Web service call interception: a) of the client generator, b) of the WSDL analyzer, and c) of the actual replica generator. Figure 6.2c displays the results. Again we performed the tests on the QWS dataset, sorted the WSDLs according to their size/complexity, and this time we used only Apache CXF as the base client generator. The bottom graph displays the performance of CXF, which is quite constant except for the very complex WSDL's. The middle graph shows the accumulated performance of the WSDL analyzer, in addition to CXF' processing time. On the top, the graph displays the sum of all three modules, which constitutes the actual processing time for intercepting a Web service call plus for generating the corresponding replica service. The results show that approximately 30% of the time is consumed for analyzing the remote WSDL and creating the model, 44% is spent on generating a running replica Web service out of it, while the generation of the local client stubs takes only 26% of the time.

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

6.3.1 Discussion

In our previous chapters we assumed that testbeds are being generated *before* the test runs, mostly by specifying all details via G2's scripting language and deploying the testbed on a backend. In the current one we propagate the generation testbeds *on-thy-fly*, which is in general the

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

exact opposite. In our opinion on-the-fly generation makes sense if the testbed composition is not known a-priori or a generation of the complete testbed infrastructure is not reasonable, e.g., because it could get too large-scale even though only a few services of it would be actually invoked. For such scenarios on-the-fly generation is more efficient and, therefore, preferable.

Regarding the degree of realism of the testbeds it is safe to say that replication of functional behavior, without having access to the remote Web service's code, is not possible. Let's consider the example of a complex and stateful remote service which performs sophisticated calculations, uses a data base system as data source, interacts with some legacy components, etc. There is no way of replicating its functional behavior in 100% if one has only access to the Web service's interface description. There are means to record and playback Web service interactions, e.g., [49, 83], but still the replication of functionality is only limited to the recorded data. That is something we have to live with. But what is a SOA software engineer supposed to do, if he/she is developing a system that must interact with such a complex service, but he/she is not allowed to use this service for testing purposes? Basically, his/her hands are bound and the best solution is to use a replica of the service in a testbed. And for this replica, it is up to the developer to implement a "realistic" behavior according to the requirements of the test run. Often it is sufficient to emulate just QoS properties (e.g., by taking over the collected QoS data from the QWS dataset), or to implement a rudimentary clone of the services functionality, etc. It all depends on how much is known about the original service and how much of it must be replicated. We cannot solve this problem in a *fully automated* manner, but we can support developers by replicating the service's interface and by providing means for assigning functional behavior to them. It is the task of the testing engineer to assign *sufficiently realistic* behavior to the replicas. We have no possibility to unburden him/her from that.

CHAPTER 7

Large-scale Testbeds in the Cloud

Published in:

CAGE: Customizable Large-scale SOA Testbeds in the Cloud.

Juszczyk L., Schall D., Mietzner R., Dustdar S., Leymann F. (2010).

6th International Workshop on Engineering Service-Oriented Applications (WESOA). 8th International Conference on Service-Oriented Computing (ICSOC'10), 07. - 10. December 2010, San Francisco, USA.

Outline. If implemented properly, SOA systems can be very scalable and interact with an arbitrary numbers of clients, services, and other external components. A problem which appears if one must develop such a system, is how to test it in realistic scenarios with (tens or hundreds of) thousands of participating components. Especially, if the designated runtime environment is not available at development time and must be, again, emulated. First of all, how can engineers establish such large-scale testbeds in a convenient manner, but the second problem is where to host such testbeds? This short chapter deals with this problem and presents the CAGE framework, which is a combination of Cafe framework, a system for provisioning of distributed systems in Cloud infrastructures, and GENESIS 2.

7.1 Motivation

Service-oriented computing (SOC) provides a high level of flexibility and scalability which benefits the realization of large-scale and complex distributed systems [135]. By applying asynchronous communication 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. However, engineers that develop complex systems that 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 resources.

In this chapter we present a step towards solving these issues. We introduce CAGE, a framework and methodology for emulating SOA environments (for utilization as testbeds) and for deploying these automatically in the cloud. Our approach combines G2 with *Cafe* [128], a framework for provisioning distributed systems 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 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

7.1.1 Scenario: Large-scale SOA Testbed Infrastructures

SOA's well-known features of dynamic binding and adaptivity make 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 (MTurk) [2], 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 the previous chapters and we have explained how testbed instances are generated in a distributed back-end. Of course, for large-scale testbeds an adequate back-end hardware infrastructure is required in order to be able to host all generated SOA components, which can get very costly. 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 of 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.

7.2 Applying Cafe

CAGE derives it functionality from combining two base frameworks: Cafe and GENESIS2. Cafe provides support for an automated provision of component-based applications into the cloud, while G2 makes use of the infrastructure provided by Cafe and generates customizable and dynamic SOA testbeds.

Cafe[1] [128] 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, e.g., what subsystems it consists of, whereas the variability model describes variability of the application, e.g., configuration details. 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 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 [1], 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 [36] 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

[1] **C**omposite **A**pplication **F**ram**e**work

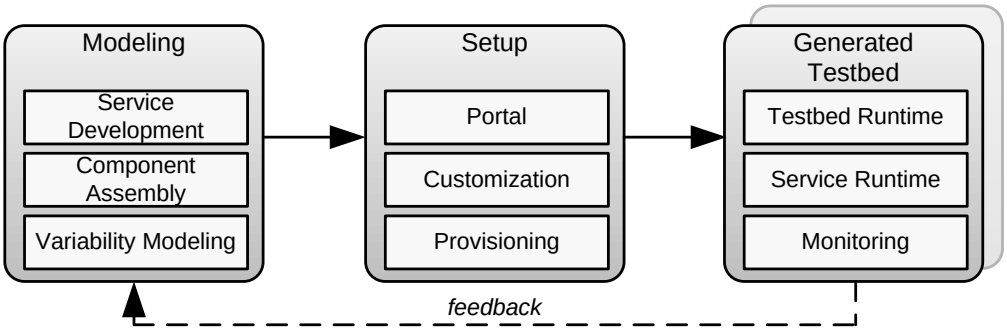


Figure 7.1: Overall CAGE approach and architecture.

component. Application template developers use internal and provider supplied components and their deployment relations to describe their application.

7.3 The CAGE Framework

7.3.1 CAGE Methodology and Roles

The CAGE methodology specifies how the framework can be used to generate large-scale customizable SOA testbeds and how to deploy them on-demand in cloud-based infrastructures. The methodology comprises three steps (modelling, setup, & execution) that are performed by two different roles (testbed engineer & tester) during the establishment of CAGE testbeds. Figure 7.1 depicts a high-level overview of the three steps and the CAGE tools supporting them.

1. *Modeling.* At first, in the modeling step, the testbed engineer creates a specification/model of the testbed. Building upon the G2 framework, this specification can comprise diverse types of SOA components that can be augmented with functional as well as non-functional properties, plus also 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 functional aspects that can be tested based on the modeled testbed.

2. *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 shown in Chapter 5. Customization is accomplished by binding points of variability, i.e., by selecting one of multiple alternatives or by entering values for a point of variability.

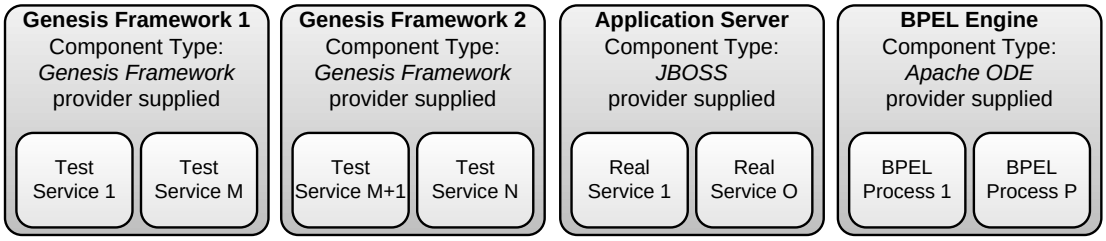


Figure 7.2: Component assembly example

3. *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. 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 Figure 7.1.

7.3.2 Modeling Testbeds

Figure 7.2 shows a component assembly example to illustrate various CAGE concepts. Component assembly in CAGE closely follows the Cafe approach [128] where modeling of composite application templates is centered around components (see application meta model in Figure 7.3). 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 [36] and Apache ODE process engine [9] shown in Figure 7.2 which allow to deploy components of implementation type Java enterprise edition (JEE) application and BPEL process on them, respectively.

As a result, the testbed engineer can compose a testbed by combining middleware components including the G2 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.

In Cafe, application templates are annotated with a variability model that is specified using the Cafe *variability meta-model* (see Figure 7.4). Such a variability model contains a set of *variability points* that specify possible configuration *alternatives* (see Figure 7.5). 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.

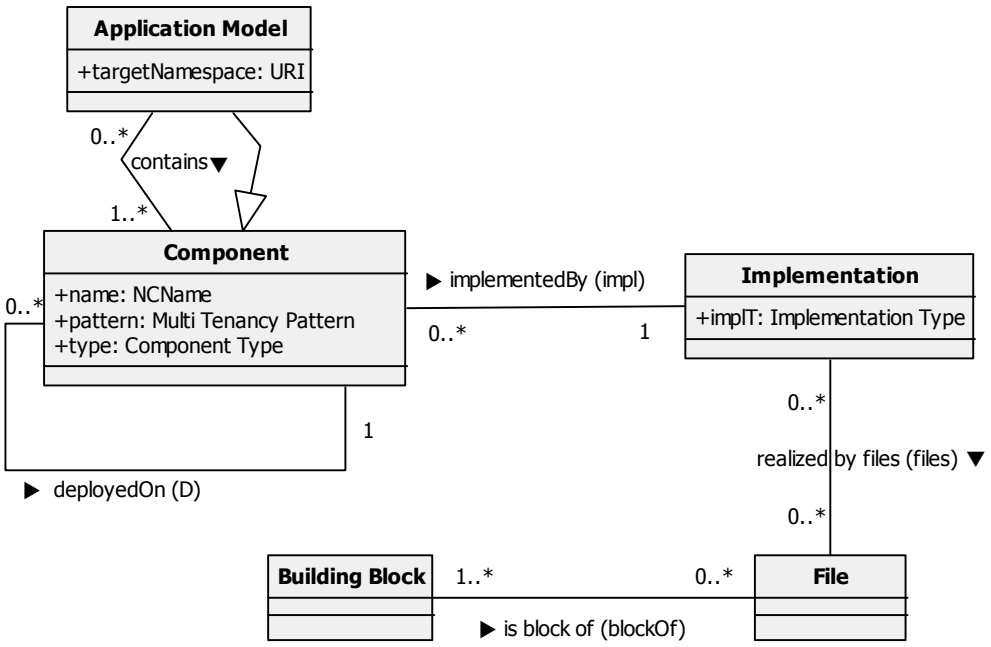


Figure 7.3: Cafe application meta model (from R. Mietzner [125]).

- *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. 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 variability that must be filled during provisioning by the provisioning environment. Variability points can also express functional and 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.

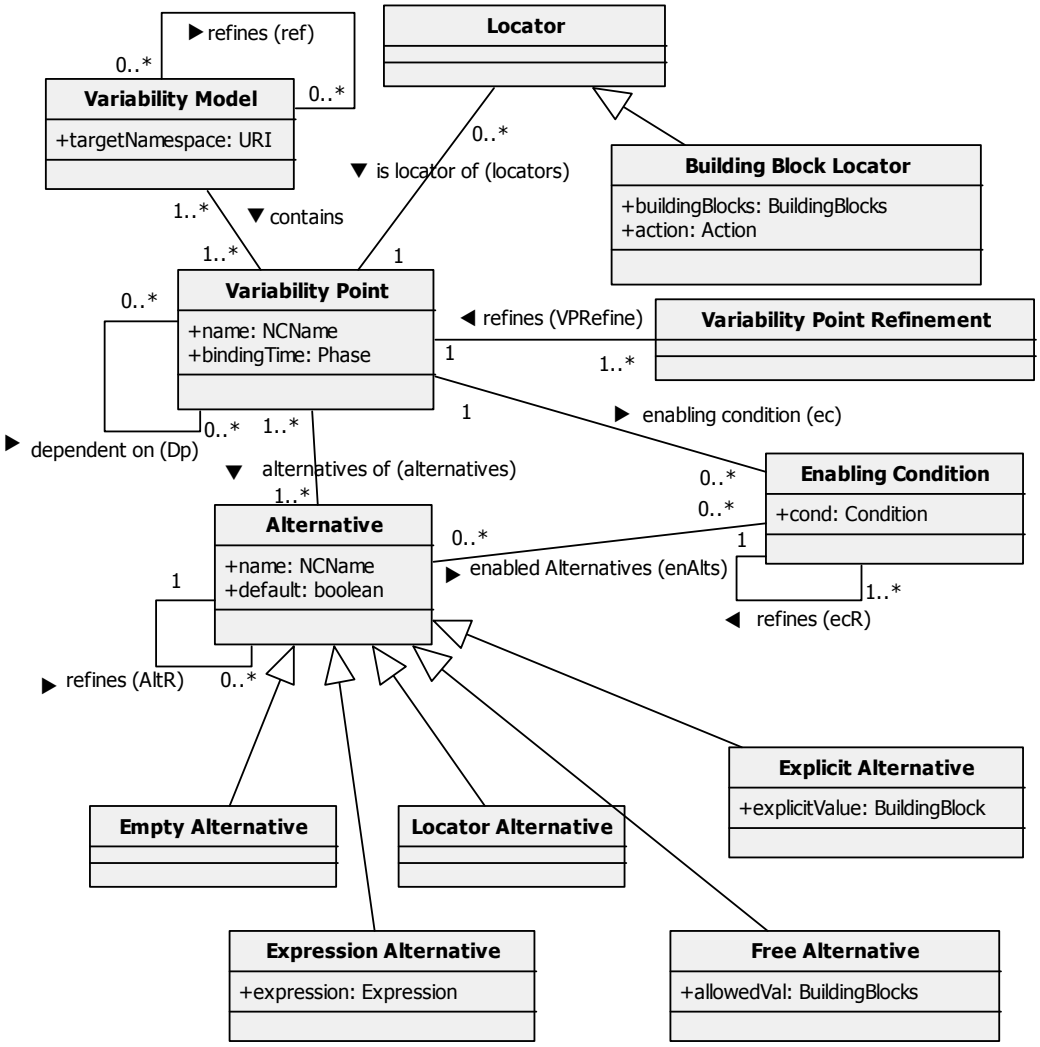


Figure 7.4: Cafe variability meta model (from R. Mietzner [125]).

7.3.3 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 [126]. 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

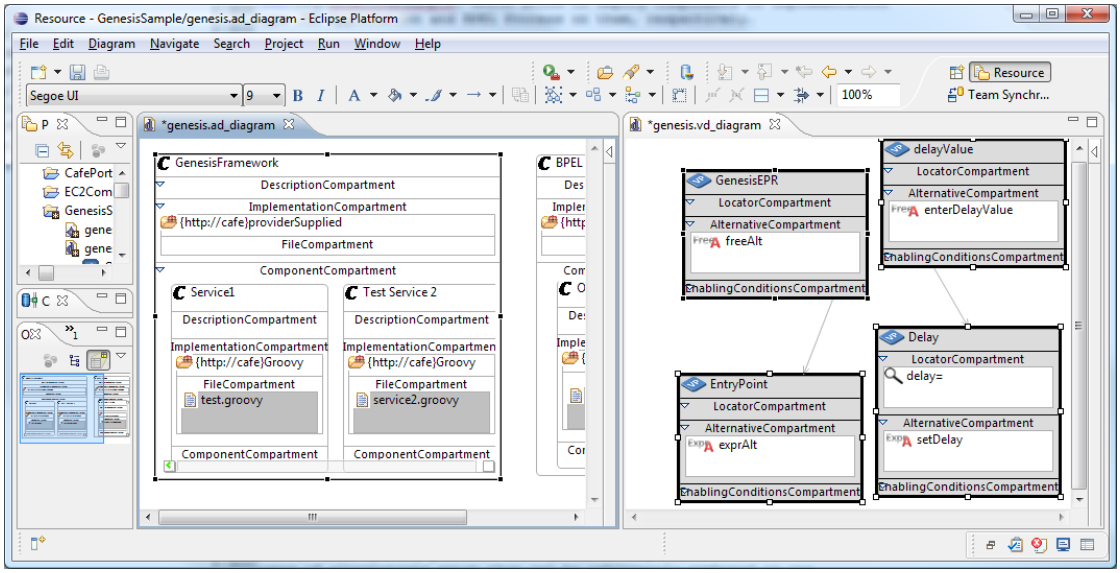


Figure 7.5: CAGE modeler: testbed components on the left, variability model on the right

testing), the provisioning infrastructure sets up the necessary components and configures them as described in the next section.

7.3.4 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* [127]. For example G2 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 G2 testbed template script which contains placeholders (`BACKENDHOSTS`, `QOS_DEF_RESPONSETIME`, and `QOS_DEF_AVAILABILITY`)

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
```

7.4 Practical Application

CAGE has been created from a merge of two already existing frameworks, Cafe and G2, in order to combine their features and to facilitate a convenient set up of testbeds which not only comprise generated components but also already existing systems, e.g., application servers and process engines. Via variability points we established links between these components, in order to create coherent testbeds, and were able to model and deploy these in a user-friendly manner, where Cafe was handling most of the configuration details during the set up, e.g., allocation of cloud instances or resolving of inter-component links.

However, CAGE has been so far only used as a proof of concept implementation and has not (yet) been applied in any real projects, which would have been necessary to evaluate its usefulness for engineers/testers. Therefore, the purpose of this chapter was to give a high-level overview of the concepts and of how engineers/testers can potentially benefit from applying our framework.

CHAPTER 8

Programming Evolvable Web Services

Published in:

Programming Evolvable Web Services.

Treiber M., **Juszczyk L.**, Schall D., Dustdar S. (2010).

2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS). 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), 2. - 8. May 2010, Cape Town, South Africa.

Outline. In this chapter we do not focus anymore on pure testbed generation but we apply the GENESIS programming model for engineering evolvable and adaptable Web services. Combining the intuitiveness of the scripting language with the ability to perform hot updates makes it possible to develop services which can be adapted at runtime in a convenient manner. We present a user-centric approach supporting automatic mechanisms for adaptation and foremost a programming model to reduce the burden of reconfiguration, update, and customization of service-based applications.

As this work was done before we developed G2, it is still built upon the concepts and the implementation of the first GENESIS framework.

8.1 Motivation

Recent research on SOA increasingly addressed the need to design systems in order to make them adaptable on changing requirements and environmental constraints [134]. The requirement of engineering systems towards being more robust, flexible, and ultimately self-adapting holds numerous challenges for developers. One way to address these challenges is to build loosely-coupled systems out of modular and dynamically replaceable services. The system can adapt,

for instance, by switching over to "better suited" services on-the-fly, if the situation requires this. However, not only the composite systems, invoking other services, need to be adaptive but also the services themselves in order to evolve, to improve their quality, and to be prepared for the selection among equivalent competitors [154]. In general, services can evolve in various manners: by improving their non-functional properties (e.g., performance) or their functional behavior (e.g., delivering better quality responses), by relocating to new hosting environments, by adapting their own interfaces, by improving their description, etc. The big challenge is, though, how to perform these changes upon running services without hampering their availability and usability. In this chapter we present a simple programming model and methodology for engineering evolvable Web services and for adapting them at runtime.

In this work we focused particularly on the developers perspective as significant evolutions of services during the life cycle [70, 133], like code refactoring or the implementation of new algorithms do not happen automatically, but rather require the developer in the loop who conducts these changes [154]. Developers require an answer to the question of *how to implement* services and their changes. Consequently, developers need the support from the service infrastructure to be able to modify services in an efficient manner. Furthermore, an adequate programming abstraction is required that allows developers to change the implementation of services according to changing requirements.

Current approaches offer limited support for developers regarding the modification of services. They treat Web services either as components or resources [97], focus on interface descriptions [52, 56] or describe services with semantic techniques [114]. The support for direct modifications of Web services in the deployment environment is limited, approaches like chain of adapters [109] intercept and transform messages but do not support the developer in modifying deployed Web services.

In this chapter, we apply the GENESIS programming model as a basis for implementing evolvable Web services.

8.1.1 Application Scenarios

The presented application scenarios are motivated by the need to create service-based applications which may be subject to changing requirements. In the first scenario we discuss an example where distributed organizations form virtual organizations (VO) to collaborate on joint tasks/projects. The second case highlights the need to provide customized services based on the service consumers preferences.

- *VO formation and collaboration:* The global scale and distribution of companies have changed the economy and dynamics of businesses. In recent years, companies and individuals have started to form virtual organizations (VO) to harvest business opportunities which single partners cannot realize on their own due to missing expertise or resources. VOs are established by creating connections between individual partners. Web services and SOA are the ideal technical framework to automate the formation process as well as interactions within VOs. Since VOs form and dissolve for the timespan of a specific

collaboration, it is desirable to create *tailored services* supporting the needed interactions between partners in an easy manner. Thus, developers are required to create a set of services enabling collaborations. Moreover, tailored services prevent unauthorized access to services or operations that should remain invisible to the collaboration partner.

- *Provisioning of custom services:* Web services have undergone fundamental changes. Users demand for personalization and context-awareness when using services. Services may be adapted based on the users' context information by offering extended features. Also, personalization plays an increasing role as the number of available services increases. A simple example is a Web portal where personalized views are created based on user preferences. Web service providers, for example hosting services in cloud environments, may want to create services offering a set of features (operations) targeting a specific consumer and/or community. These custom services can be created by selecting and aggregating available operations from a repository.

Both scenarios demand for flexibility and adaptivity of services. A system satisfying the needs of the presented application cases is not designed, developed, and deployed in a top-down manner, but rather changes and evolves over time. While research in the semantic Web services community focuses on automatic adaptation of services (e.g., mediation of messages, goal driven compositions, etc.), our approach focuses on the developer's perspective. The fundamental question we attempt to address in this work is: how can developers efficiently adapt systems while maintaining the system's availability?

8.1.2 Adaptation in Service-oriented Systems

In the following, we discuss various possibilities for adapting SOA's. We distinguish between *(i) manual adaptation* performed by developers and *(ii) automatic (self-)adaptation*. The latter has recently received considerable attention from the research community while the former is not sufficiently supported by existing WS toolkits and frameworks. Figure 8.1 provides an overview of adaptation in SOA.

The development process can be viewed from various starting points. The *Provider* may start to implement services based on market demands and innovations of competitors. We make no assumption about the role of the provider with regards to development aspects of services. From the Web services point of view, functional capabilities of services can be described through well-defined interfaces, whereas non-functional characteristics, for example service metering, costs, and QoS attributes [133], can be modeled using policy frameworks.

Providers typically offer a set of services to a number of *Consumers*. As mentioned in our motivating application scenarios, consumers may have different preferences and requirements, thereby demanding for customizations of services. The provider is responsible for hosting services in an appropriate *Infrastructure* to satisfy consumer demands in terms of functional capabilities and QoS.

Autonomic adaptation (see self-referential arrow) may need to be performed to satisfy QoS guarantees, service availability, to name a few and has received considerable attention (e.g.,

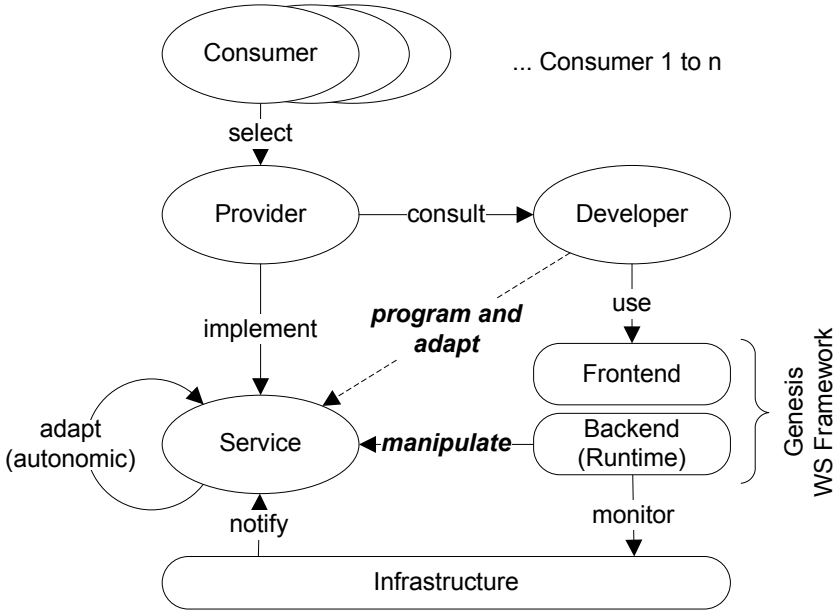


Figure 8.1: Overview of adaptive SOA

see [130]). For example, such self-adaptation actions may be triggered by notifying services about infrastructure events.

Here we address adaptations performed by the *Developer*. The developer is in charge of programming services and implementing adaptations (dotted arrow pointing from Developer to Service in Figure 8.1). A set of tools are needed for this purpose, which are depicted as *Frontend* and *Backend*, both provided by GENESIS. The developer performs the logical action 'program and adapt' using the frontend comprising tools, APIs, user shell, etc. The backend is deployed in the infrastructure — potentially multiple backend instances to achieve scalability. Also, the infrastructure is monitored by the backend to receive information about deployed resources or load conditions which can be propagated back to the frontend to assist the developer when adapting services. Automatic adaptations could potentially be triggered by the backend, although this is not within the scope of this work.

In this chapter, we highlight the following novel key contributions:

1. A developer-centric approach for programming and adapting Web services via a scripting language.
2. A simple and intuitive programming model assisting developers in adaptation actions such as service migration, replication, or even refactoring of multiple distributed services at the same time.
3. Extensibility and flexibility in managing services through behavior modules, realized as GENESIS plugins.

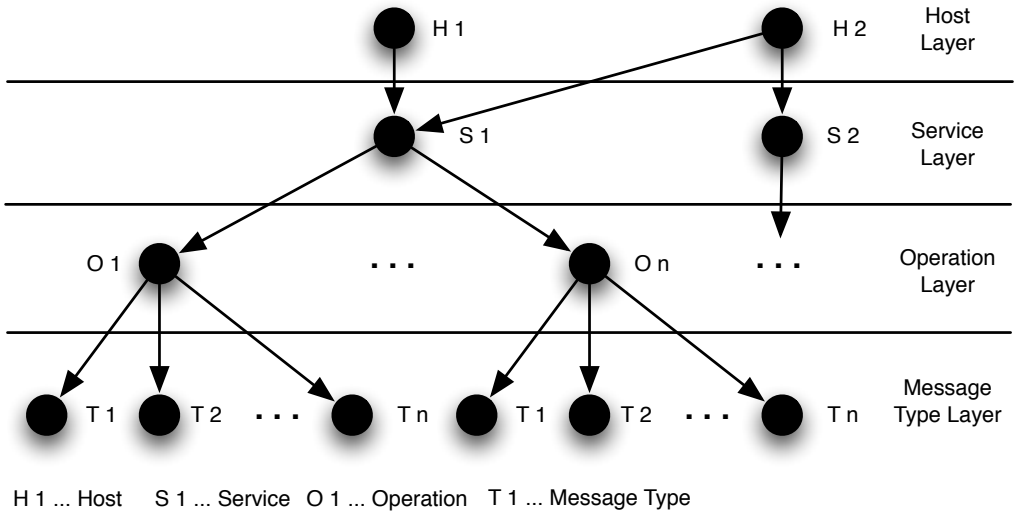


Figure 8.2: Programming abstraction for Web services derived from WSDL.

8.2 Programming Model

The proposed programming model differs from the usual methodology for Web service development. In order to make Web services adaptable, in the sense of altering the service's interface and behavior at runtime, we regard it as useful to encapsulate its implementation into serializeable and composable building blocks. Today, the most common way of developing Web services is to create data/code objects representing the services, with class methods implementing the service's operations. The main drawback of this approach is the tight binding of operations to services (which means, methods to objects) making it impossible to perform adaptations on a structural level without recompiling and redeploying the whole service. This poses a hard limitation for flexibility and adaptivity.

Our approach, derived from the GENESIS programming model, splits the Web service model, according to the structure in WSDL documents, into 4 main layers of element types, namely `Host`, `Service`, `Operation`, and `MessageType`.

The `Host` type references a remote host and provides access to its deployed services, in order to de-/re-/install these. The `Service` type represents the general specification of a Web service, with all its global dependencies and properties. `Operations` encapsulate the behavior of the service and `MessageTypes` define the schemas of the exchanged messages, which also includes headers.

Our methodology comprises the usage of an API at the front-end in order to control a set of back-end hosts. The main features of the API deal with creating and manipulating Web service models and for deploying them on the back-end hosts. At the back-end, a runtime environment handles the transformation of models into service instances and supports the basic CRUD (`Create`, `Read`, `Update`, & `Delete`) operations for manipulating them. Furthermore, our

approach supports the usage of pluggable behavior modules which augment the Web services with arbitrary functionality, e.g., access to data bases or registration at UDDI brokers. Table 8.1 summarizes the main effects of CRUD on the model.

Model	Modification
Host	**Create:** Bootstrapping of remote host (requires Cloud-like host manipulations, e.g., performed by Cafe [128]) **Read:** Retrieve model of deployed Web services (e.g., for migration) **Update:** • Update of host-global behavior modules • Setting host-global properties **Delete:** Host shutdown
Service	**Create:** Service deployment for creation / replication / migration **Read:** Read service configuration and metadata **Update:** • of service behavior modules • of service properties (e.g. URL) **Delete:** Service undeployment
Operation	**Create:** Addition of operation **Update:** • of operation code • of operation properties (e.g. binding) **Delete:** Removal of operation
MessageType	**Create:** Addition of headers and/or message types **Update:** • of request/response types and headers • of header processing code **Delete:** Removal of headers and/or message types
Behavior Module	**Create:** Deployment of modules for extending service functionality **Update:** • Replacement of modules • Steering of pluggable functions via parameter manipulation **Delete:** Undeployment of modules

Table 8.1: Modifications on model types and behavior modules.

8.2.1 Script-based Web Service Programming

As explained in Chapter 3, GENESIS provides an API for modeling services, which can be also integrated into scripting environments such as the Bean Scripting Framework (BSF) [31]. In this work, we are using sample snippets written in Jython [38], a BSF implementation of the Python language, to demonstrate the usage of our programming model. The following sample shows the creation and deployment of a simple Web service.

```
# create host reference
h = Host("http://example.net:8080/services")

# create service model
s = Service("SampleService")

# create dummy method in Jython
def sign(par):
    signatureStr = "Not implemented yet"
    return signatureStr

# bind method to service operation
o = Operation("SignData")
o.setBehavior(sign)

# create XSD-based message types
t = MessageType("types.xsd", "dataTypeName")

# attach message types to operation
o.addInputType("param", t)
o.setOutputType("string") % type for sign() response

# attach operation to service
s.addOperation(o)

# attach service to host and deploy
s.deployAt(h)
```

Listing 8.1: Sample definition of Web service

By executing the script code, the Web service is deployed at `http://example.net:8080/services/SampleService`. The functionality of the service is defined by binding a native Jython method (`sign()`) to a Web service operation, in order to encapsulate the operation's behavior for remote execution.

Deployed Web services can be adapted by importing their model from a host, performing changes to it, and redeploying it again. The next snippet demonstrates this feature and shows how to extend services with behavior modules.

```
# import service model from remote host
h = Host("http://example.net:8080/services")
s = h.getService("SampleService") # get by name
o = s.getOperation("SignData")

# create new header message type
ct = MessageType("types.xsd", "credentials")
ct.setHeader(true)

o.addInputType("creds",ct)

# uses "auth" and "gpg" plugins
def newSign(par,creds):
```

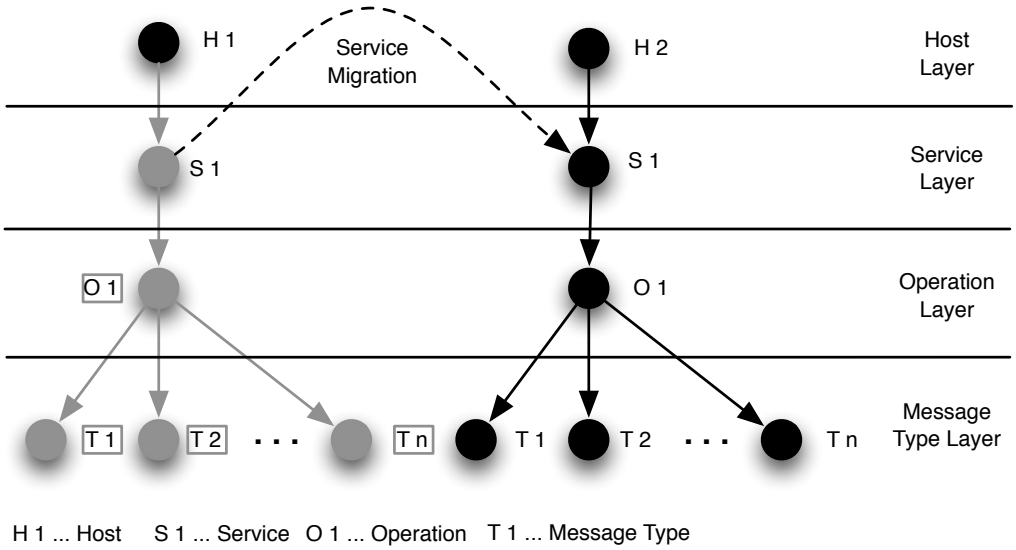


Figure 8.3: Migration of service S1 from host H1 to host H2

```
    auth.check(creds) # exception on failure
    signatureStr = gpg.sign(par)
    return signatureStr

# replace operation behavior
o.setBehavior(newSign)

# install plugins at host using global aliased "auth" and "gpg"
h.usePlugin("auth.jar")
s.usePlugin("gpg.jar")

# redeployment at remote host
s.update()
```

Listing 8.2: Usage of behavior modules (plugins)

Apart from simple adaptations, which change a service's interface and/or behavior, changes can be also performed at a higher level, by combining API methods into composite ones, for instance to migrate or replicate services to other hosts (see Figure 8.3).

```
def replicateService(serviceName, fromHost, toHost)
    # import model from source host
    s = fromHost.getService(serviceName)
    if s is None:
        raise Exception("Unknown service")
    # deploy at new one
    s.deployAt(toHost)
    return s
```

```
def migrateService(serviceName,fromHost,toHost)
    s = replicateService(serviceName,fromHost,toHost)
    # remove after successful deployment
    s.undeployFrom(fromHost)

def migrateHost(fromHost,toHost)
    # iterate through deployed plugins and services
    for p in fromHost.getPluginModules():
        toHost.usePlugin(p)
    for s in fromHost.getServices():
        migrateService(s.getName(),fromHost,toHost)
```

Listing 8.3: Combined commands for replicating and migrating services

During deployment at back-end hosts, models of Web services are translated into running instances of these. For this purpose our system serializes service models, including the code blocks of the operations and all referenced pluggable modules, and transfers them to the designated hosts. The translation process itself comprises the analysis of the Web service model and the generation of Java code which implements the intented behavior. This procedure got explained in detail in Chapter 3.

8.2.2 Extending Services with Behavior Modules

For our programming model particular priority has been put on simplicity, allowing developers to set up Web services quickly and to perform adaptations in a convenient manner.

However, this came at the cost of sacrificing the ability to create complex Web services due to the encaplusation of Web service operations to single code blocks, e.g., to Jython methods like in the previous samples. To overcome this limitation *to a certain degree*, we are using pluggable behavior modules which can provide arbitrary functionality and which can be accessed by the operations. Our framework supports developers by providing an abstract Java class for the modules, which takes care of binding them to the runtime environment via alias names. Modules can bei either registered at the host level, for globally visibility, or at the service level, for restricted visibility to particular services. We have developed a set of behavior modules which are frequently needed in the SOA domain, e.g., a service invoker for calling remote services and a simple workflow engine for executing nested BPEL processes. Basically, we adopted the repository of already available plugins provided by GENESIS, which are listed in Section 3.3.2.

Being based on the programming model of the first version of GENESIS (as G2 had not been available yet at that time), the presented approach suffers from several restrictions regarding fine-granular programmability of service behavior. In GENESIS plugins were orchestrated via aligning calls according to a simple grammar, while G2 has been designed to specify service behavior in Groovy closures that referenced plugins and executed them on top of the shared runtime environment. However, in this work we improved the plugin concept of GENESIS and adapted an approach which was more flexible, yet, not as advanced as the one of G2. We use Jython procedures to implement behavior and to align plugin invocations, which is superior to the restricted possibilities of GENESIS (sequential, parallel alignment and try/catch blocks).

Engineers are able to program their own behavior routines by reusing functionality encapsulated in behavior modules (plugins), e.g., as done in Listing 8.2 (`newSign()`). Yet, it it missing several features which were introduced later with G2, such as code blocks as model parameters (for exchanging sub-routines of operations) or a shared runtime environment, used for data exchange among pluings and/or services. Therefore, we regard the approach presented in this chapter as an intermediary step between GENESIS and G2, where we had already experienced the limitations of GENESIS and wanted to overcome them as best as possible by only improving the framework and without rewriting it from scratch.

8.3 Discussion

The adaptation of services requires careful considerations. In the following we discuss the strengths and the limitations of our proposed programming model and its current implementation.

8.3.1 Strengths

The flexibility of our model has several positive implications for the service developer who benefits from the following features:

- *Simplicity and intuitivity:* Web services can be created in a simple and intuitive manner. For example, the developer is able to modify services on the host layer (e.g., replication, migration) with the same programming primitives like on the service layer when changing the operations of a service.
- *Modularity:* As a unit of reuse, behavior modules allow developers to encapsulate arbitrary functionality and reuse these modules with different services in the GENESIS runtime environment.
- *Run time service adaptation:* The modification of services at run time is supported by the GENESIS based prototype infrastructure. The GENESIS environment manages the (re-)deployment of services and the modifications in a transparent way for the developer. Our prototype implementation supports developers by hiding the complexity of service modifications and keeps the available services in sync with the abstract programming model.
- *Consistency between Model and Service:* The gap between the abstract model that describes a service and its implementation is very narrow. When manipulating the service model, the developer directly changes the implementation of the service and vice versa. Thus, we lay the foundation for automated service modifications which can be caused other than by the developer.

8.3.2 Limitations

Using a flexible programming model, we encounter a set of challenges that are of importance and are not fully addressed in our current version of the programming model. Limitations at this stage include:

- *Support for Stateful Services:* An issue that has strong impact on service adaptations is state [117]. Services which have internal state that influences the execution result of a service must be treated in a manner that does not corrupt the state of the service during the adaptation process. A straightforward approach is to allow service modifications only in ground states when no invocation is active and the internal state of a service can be persisted and then later recovered to restart the service. This obviously limits applicable modifications, because (i) adaptations only can take place during a certain time interval and (ii) modifications that change the service semantics (e.g., the removal of an operation) are not applicable if the service is not in a ground state. In the present implementation, we do not support the complex manipulations of stateful services which require knowledge of meta information like active transactions.

- *Dependency Model:* Dependencies of services manifest themselves in different dimensions [153]. We can roughly distinguish between internal and external service dependencies. Examples of internal dependencies are the use of behavior modules, external dependencies can be observed as links to databases or libraries. These dependencies need to be taken into account, before a service is adapted. An approach to model dependencies is the use of manifest files that simply lists required resources of Web services that must be available for a service to function properly. In our programming model, we encounter these types of dependencies on all four layers. For example, the migration of an operation from one service to another might require the availability of a certain behavior module at the target service. So far, we do not provide active support to manage dependencies on the infrastructure level.

- *Security and User Model:* Security concerns are not addressed with our current prototype implementation. Basically, each user that has access to the GENESIS framework can modify each service at any given moment. Related to the security model are considerations about the user model. We do not support a dedicated user model in the prototype version.

- *Eventing Model - Event Propagation:* The causes that trigger adaptations can originate from different sources. For example, the observation that a service is operating near its pre-defined maximum throughput of 50 requests per second, might trigger a duplication of the service to another host to handle the load in order to fulfill the requirements of customers. Manually triggered adaptions, include for example internal code changes due to optimization of algorithms are triggered by the developer. The needed infrastructure is not yet implemented subject to possible future work.

CHAPTER 9

Summary, Conclusions, and Outlook

In this thesis we have presented the results of our research on *techniques for generating testbed infrastructures for SOA*. We have analyzed the state of the art of SOA software engineering, outlined a lack of support for testing complex service-oriented systems, investigated techniques and approaches for solving this issue, and, consequently, we propose our results in this dissertation.

Our work has been concentrated on testing complex systems which operate in (potentially large-scale) environments of SOA components, such as services, clients and workflow engines. As often such environments are not available during the process of developing the complex system, this puts a burden on the engineers/testers. The question appears of *how to perform tests if the designated destination environment is not available?* In this thesis we argue that this problem can be solved by emulating the missing environments in order to have so called testbeds for the developed system. Though, this important task has not been given enough attention by the research community, as only a small number of articles have been published on this topic. Furthermore, these were all focused on solving very specific problems, which means on performing specific kinds of tests. Support for generating generic SOA testbeds was not available at all and engineers/testers had to set up their required testbeds manually, which was a time-consuming task. We regarded this situation as unacceptable and concentrated our research on developing *novel techniques for generating testbeds of arbitrary composition, structure, topology, and behavior*, ergo which are customizable to the requirements of the tested system.

We achieved our goal in several steps which were presented in this thesis' chapters. At first, we came up with the GENESIS framework which was focused on generating testbeds consisting of customizable Web services. We evolved our concepts towards more flexibility, extendability, and towards supporting the generation of arbitrary SOA components, not only Web services. The result was GENESIS2, in short, G2. Then, we developed techniques for augmenting G2 with fault injection in order to achieve more realism within the testbeds. Later, we automated parts of the testbed generation process by monitoring SOA systems and creating testbed for these automatically. And, finally, we combined G2 with the Cafe framework for deploying large-scale testbeds in cloud-based infrastructures.

In a nutshell, the conceptual contribution of this thesis comprises

- *a flexible scripting language for specifying and programming of testbeds,*
- *techniques for generating deployable testbed instances from the specifications,*
- *a testbed model providing high extendability and customizability via plugins,*
- *and an approach to automate the process with AOP.*

Furthermore, we contributed to the research community by *implementing the developed concepts in a prototype* and by *releasing it as open-source*.

9.1 Outlook and Possible Future Work

GENESIS and G2 were the first available *generic purpose* testbed generators that can be extended and customized to the requirements of the tested system and the needs of the engineers/testers/users. Based on this extendability they provide a solid grounding for research on testing of SOA systems. In our group, we applied our framework for emulating faults and QoS in testbeds [123], for emulating human-provided services [143, 146], for evaluating self-healing techniques in SOA [138], for evaluating monitoring mechanisms in business processes [112], etc. Of course these projects do not cover all aspects of SOA testing. We identified the following challenges to be solved yet and potential evolutions of our work, which will be, hopefully, handled in future projects:

- We envision a better integration of the software engineering process, the testing process, and the testbed generation. We argue that it is necessary to provide means to derive testbed specifications automatically from software models, without the need to specify all details manually. Furthermore, the set up and execution of test runs, as well as the evaluation of the test results, should be more supported and automated. At our group we have started to work on these issues in the project *Audit 4 SOA*.
- One of the main shortcomings of the current state of our work is the limited support for emulating (or replicating) the functional behavior of external Web services. We expect engineers/testers to know about the service and to replicate its functionality via the programmability of our script-based approach. We think that this process could be improved by having not only a public description of the services interface (wrapped in WSDL documents) but also a functional one, which would make it possible to generate functional replicas in an automated manner. Of course, this is not a trivial task and requires thorough investigations.
- In the scope of socially-enhanced SOA, where human workers are part of SOA-based systems, the emulation of human-provided services (HPS) becomes very important. In our previous works we emulated selected properties of HPS, such changing availability and

throughput of tasks. However, human workers expose a significantly different behavior compared to software services. This is not only limited to QoS and performance issues, but also affects the quality of the responses, task delegation behavior, etc. We regard it as necessary to invest more effort into emulating realistic HPS in order to be able to test their application in an SOA.

- There exist several WS-* specifications, such as WS-Transaction and WS-Notification, which handle and control complex interactions among services. To test the execution engines implementing these specifications, it would be necessary to generate testbeds of Web services which are able to participate in such interactions and to enhance them with corresponding behavior models in order to execute realistic scenarios.

- Last but not least, we see the need for emulating of ReST-based testbeds, due to the growing importance of ReSTful services for the Web community, e.g., within service mashups (= simple Web-based workflows). As ReST is based on different concepts compared to SOAP services and also is applied in a different manner, this would require to come up with novel concepts for emulating ReSTful service environments.

This was only a list of selected (possible) future works/projects that we regard as most important. In general we see a big potential for successor projects to our work, due to the extensible nature of the framework, the open-source availability of the prototype implementation, plus due to the urgent need of sophisticated testing solutions. We hope that GENESIS will have a strong impact on the future development of research on SOA testing.

Bibliography

[1] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[2] Amazon Mechanical Turk. http://www.mturk.com/.

[3] Apache Axis 2. http://axis.apache.org/axis2/.

[4] Apache CXF. http://cxf.apache.org/.

[5] Apache CXF Dynamic Clients. http://cxf.apache.org/docs/dynamic-clients.html.

[6] Apache CXF Features. http://cxf.apache.org/docs/features.html.

[7] Apache CXF Interceptors. http://cxf.apache.org/docs/interceptors.html.

[8] Apache Muse. http://ws.apache.org/muse/.

[9] Apache Orchestration Director Engine (ODE). http://ode.apache.org/.

[10] Apache ServiceMix. http://servicemix.apache.org/home.html.

[11] Apache ServiceMix BeanFlow. http://servicemix.apache.org/beanflow.html.

[12] Apache Tuscany CORBA Binding. http://tuscany.apache.org/sca-java-bindingcorba.html.

[13] Apache Tuscany RMI Binding. http://tuscany.apache.org/sca-java-bindingrmi.html.

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

[15] AspectJ. http://www.eclipse.org/aspectj/.

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

[17] Business Process Execution Language for Web services (WS-BPEL). http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[18] Electronic Business using XML (ebXML). http://www.ebxml.org/.

[19] Extensible Markup Language (XML). http://www.w3.org/XML/.

[20] GENESIS Prototype. http://www.infosys.tuwien.ac.at/prototype/Genesis/.

[21] GlassFish. http://http://glassfish.java.net/.

[22] Globus Toolkit. http://www.globus.org/toolkit/.

[23] Google SOAP API. http://code.google.com/intl/de-DE/apis/soapsearch/.

[24] Groovy. http://groovy.codehaus.org/.

[25] Groovy Closure. http://groovy.codehaus.org/Closures.

[26] Groovy GPath Expressions. http://groovy.codehaus.org/GPath.

[27] Groovy SOAP Web Services. http://groovy.codehaus.org/Groovy%2BSOAP.

[28] Hypertext Transfer Protocol (HTTP). http://www.w3.org/Protocols/.

[29] IBM Websphere. http://www-01.ibm.com/software/websphere/.

[30] IBM Websphere Process Server. http://www-01.ibm.com/software/integration/wps/.

[31] Jakarta Bean Scripting Framework (BSF). http://jakarta.apache.org/bsf/.

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

[33] Java Compiler Compiler (JavaCC). https://javacc.dev.java.net/.

[34] Java Distribution Functions library. http://statdistlib.sourceforge.net.

[35] Java Message Service (JMS). http://www.oracle.com/technetwork/java/jms/.

[36] JBoss. http://www.jboss.org.

[37] JBoss Enterprise Service Bus. http://www.jboss.org/jbossesb/.

[38] Jython. http://www.jython.org.

[39] Linux Advanced Routing & Traffic Control (tc). http://lartc.org.

[40] Microsoft Distributed Component Object Model (DCOM). http://www.microsoft.com/com/default.mspx.

[41] netem - Network Emulator. http://www.linuxfoundation.org/en/Net:Netem.

[42] ns-2 Network Simulator. http://isi.edu/nsnam/ns/.

[43] Oracle BPEL Process Manager. http://www.oracle.com/technetwork/middleware/bpel/overview/.

[44] PUPPET Prototype. http://labsewiki.isti.cnr.it/labse/tools/puppet/public/main.

[45] SOA Principles. http://www.soaprinciples.org.

[46] SOABench Prototype. http://code.google.com/p/soabench/.

[47] SOAP. http://www.w3.org/TR/soap/.

[48] SOAP Over UDP. http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.html.

[49] soapUI Web Service Testware. http://www.soapui.org/.

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

[51] Vienna Runtime Enviroment for Service-oriented Computing (VRESCO). http://www.infosys.tuwien.ac.at/prototypes/VRESCo/.

[52] Web Application Description Language (WADL). https://wadl.dev.java.net/wadl20090202.pdf.

[53] Web Service Agreement. http://www.ogf.org/documents/GFD.107.pdf.

[54] Web Services Choreography Description Language (WS-CDL). http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[55] Web Services Description Language for Java (WSDL4J). http://sourceforge.net/projects/wsdl4j/.

[56] Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl.

[57] Web Services Distributed Management (WSDM). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm#overview.

[58] Web Services Dynamic Discovery. http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf.

[59] Web Services Human Task. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf.

[60] Web Services Interoperability (WS-I). http://www.ws-i.org/.

[61] Web Services Notification. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[62] Web Services Transaction. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[63] WS-BPEL Extension for People (BPEL4People). http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf.

[64] XML Schema Definition (XSD). http://www.w3.org/TR/xmlschema-0/.

[65] *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2008, September 3-5, 2008, Parma, Italy* (2008), IEEE.

[66] *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008* (2008), ACM.

[67] *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011* (2011), ACM.

[68] AL-MASRI, E., AND MAHMOUD, Q. H. Discovering the best web service. In *WWW* (2007), ACM, pp. 1257–1258.

[69] AL-MASRI, E., AND MAHMOUD, Q. H. Qos-based discovery and ranking of web services. In *ICCCN* (2007), IEEE, pp. 529–534.

[70] ANDRIKOPOULOS, V., BENBERNOU, S., AND PAPAZOGLOU, M. Managing the evolution of service specifications. *Advanced Information Systems Engineering* (2008), 359–374.

[71] AVERSTEGGE, M. Contract based, non-invasive, black-box testing of web services. In *ICSOC* (2010), vol. 6470 of *Lecture Notes in Computer Science*, pp. 695–698.

[72] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. E. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput. 1*, 1 (2004), 11–33.

[73] BARESI, L., BIANCULLI, D., GUINEA, S., AND SPOLETINI, P. Keep it small, keep it real: Efficient run-time verification of web service compositions. In *FMOODS/FORTE* (2009), vol. 5522 of *Lecture Notes in Computer Science*, Springer, pp. 26–40.

[74] BARROS, M. D., SHIAU, J., SHANG, C., GIDEWALL, K., SHI, H., AND FORSMANN, J. Web services wind tunnel: On performance testing large-scale stateful web services. In *DSN* (2007), IEEE Computer Society, pp. 612–617.

[75] BASILI, V. R., AND PERRICONE, B. T. Software errors and complexity: An empirical investigation. *Commun. ACM 27*, 1 (1984), 42–52.

[76] BENATALLAH, B., HACID, M.-S., LÉGER, A., REY, C., AND TOUMANI, F. On automating web services discovery. *VLDB J. 14*, 1 (2005), 84–96.

[77] BERTOLINO, A., ANGELIS, G. D., FRANTZEN, L., AND POLINI, A. Model-based generation of testbeds for web services. In *TestCom/FATES* (2008), vol. 5047 of *Lecture Notes in Computer Science*, Springer, pp. 266–282.

[78] BERTOLINO, A., ANGELIS, G. D., LONETTI, F., AND SABETTA, A. Let the puppets move! automated testbed generation for service-oriented mobile applications. In *EUROMICRO-SEAA* [65], pp. 321–328.

[79] BERTOLINO, A., ANGELIS, G. D., AND POLINI, A. Automatic generation of test-beds for pre-deployment qos evaluation of web services. In *WOSP* (2007), ACM, pp. 137–140.

[80] BERTOLINO, A., ANGELIS, G. D., AND POLINI, A. A qos test-bed generator for web services. In *ICWE* (2007), vol. 4607 of *Lecture Notes in Computer Science*, Springer, pp. 17–31.

[81] BERTOLINO, A., GAO, J., MARCHETTI, E., AND POLINI, A. Automatic test data generation for xml schema-based partition testing. In *AST* (2007), IEEE, pp. 10–16.

[82] BERTOLINO, A., AND POLINI, A. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA* (2005), IEEE Computer Society, pp. 134–142.

[83] BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND PUCELLA, R. Tulafale: A security tool for web services. In *FMCO* (2003), vol. 3188 of *Lecture Notes in Computer Science*, Springer, pp. 197–222.

[84] BIANCULLI, D., BINDER, W., AND DRAGO, M. L. Automated performance assessment for service-oriented middleware: a case study on bpel engines. In *WWW* (2010), ACM, pp. 141–150.

[85] BIANCULLI, D., BINDER, W., AND DRAGO, M. L. Soabench: performance evaluation of service-oriented middleware made easy. In *ICSE (2)* (2010), ACM, pp. 301–302.

[86] BIRMAN, K. P. Like it or not, web services are distributed objects. *Commun. ACM 47*, 12 (2004), 60–62.

[87] BOZKURT, M., HARMAN, M., AND HASSOUN, Y. Testing web services: A survey. Tech. Rep. TR-10-01, Department of Computer Science, King's College London, January 2010.

[88] CANFORA, G., AND PENTA, M. D. Testing services and service-centric systems: challenges and opportunities. *IT Professional 8*, 2 (2006), 10–17.

[89] CANFORA, G., AND PENTA, M. D. Service-oriented architectures testing: A survey. In *ISSSE* (2008), vol. 5413 of *Lecture Notes in Computer Science*, Springer, pp. 78–105.

[90] DAI, G., BAI, X., WANG, Y., AND DAI, F. Contract-based testing for web services. In *COMPSAC (1)* (2007), IEEE Computer Society, pp. 517–526.

[91] DENARO, G., PEZZÈ, M., TOSI, D., AND SCHILLING, D. Towards self-adaptive service-oriented architectures. In *TAV-WEB* (2006), ACM, pp. 10–16.

[92] DIALANI, V., MILES, S., MOREAU, L., ROURE, D. D., AND LUCK, M. Transparent fault tolerance for web services based architectures. In *Euro-Par* (2002), vol. 2400 of *Lecture Notes in Computer Science*, Springer, pp. 889–898.

[93] DOMINGUE, J., FENSEL, D., AND GONZÁLEZ-CABERO, R. Soa4all, enabling the soa revolution on a world wide scale. In *ICSC* (2008), IEEE Computer Society, pp. 530–537.

[94] EL-REWINI, H., AND HALANG, W. The engineering of complex distributed computer systems. *IEEE Concurrency 05*, 4 (1997), 30–31.

[95] ELLIMS, M., BRIDGES, J., AND INCE, D. C. Unit testing in practice. In *ISSRE* (2004), IEEE Computer Society, pp. 3–13.

[96] ERL, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[97] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[98] GOESCHKA, K. M., FROIHOFER, L., AND 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* (2008), IEEE Computer Society, pp. D4–D9.

[99] HALIMA, R. B., DRIRA, K., AND JMAIEL, M. A qos-oriented reconfigurable middleware for self-healing web services. In *ICWS* (2008), IEEE Computer Society, pp. 104–111.

[100] HECKEL, R., AND LOHMANN, M. Towards contract-based testing of web services. *Electr. Notes Theor. Comput. Sci. 116* (2005), 145–156.

[101] HOLANDA, H. J. A., BARROSO, G. C., AND DE BARROS SERRA, A. Spews: A framework for the performance analysis of web services orchestrated with bpel4ws. In *ICIW* (2009), IEEE Computer Society, pp. 363–369.

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

[103] HUANG, H., TSAI, W.-T., PAUL, R. A., AND CHEN, Y. Automated model checking and testing for composite web services. In *ISORC* (2005), IEEE Computer Society, pp. 300–307.

[104] HUHNS, M. N., AND SINGH, M. P. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing 9*, 1 (2005), 75–81.

[105] HUMMER, W., RAZ, O., SHEHORY, O., LEITNER, P., AND DUSTDAR, S. Test coverage of data-centric dynamic compositions in service-based systems. In *ICST* (2011), IEEE Computer Society, pp. 40–49.

[106] IBM. WS-* Standards. `http://www.ibm.com/developerworks/webservices/standards/`.

[107] JOE GREGORIO. Do we need WADL? `http://bitworking.org/news/193/Do-we-need-WADL`.

[108] JORGENSEN, P. C., AND ERICKSON, C. Object-oriented integration testing. *Commun. ACM 37*, 9 (1994), 30–38.

[109] KAMINSKI, P., LITOIU, M., AND MÜLLER, H. A. A design technique for evolving web services. In *CASCON* (2006), IBM, pp. 303–317.

[110] KERNER, S. M. Lots of REST For Ruby on Rails 2.0. `http://www.internetnews.com/dev-news/article.php/3715981`.

[111] KHAN, T. A., AND HECKEL, R. On model-based regression testing of web-services using dependency analysis of visual contracts. In *FASE* (2011), vol. 6603 of *Lecture Notes in Computer Science*, Springer, pp. 341–355.

[112] KHAZANKIN, R., AND DUSTDAR, S. Providence: A framework for private data propagation control in service-oriented systems. In *ServiceWave* (2010), vol. 6481 of *Lecture Notes in Computer Science*, Springer, pp. 76–87.

[113] LEITNER, P., ROSENBERG, F., AND DUSTDAR, S. Daios: Efficient dynamic web service invocation. *IEEE Internet Computing 13*, 3 (2009), 72–80.

[114] LI, K., VERMA, K., MULYE, R., RABBANI, R., MILLER, J. A., AND SHETH, A. P. Designing semantic web processes: The wsdl-s approach. In *Semantic Web Services, Processes and Applications* (2006), vol. 3 of *Semantic Web And Beyond Computing for Human Experience*, Springer, pp. 161–193.

[115] LI, Z. J., ZHU, J., ZHANG, L.-J., AND MITSUMORI, N. M. Towards a practical and effective method for web services test case generation. In *AST* (2009), IEEE, pp. 106–114.

[116] LOOKER, N., MUNRO, M., AND XU, J. Simulating errors in web services. *International Journal of Simulation Systems 5*, 5 (2004), 29–37.

[117] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The smart way to migrate replicated stateful services. In *EuroSys* (2006), ACM, pp. 103–115.

[118] LÜBKE, D. Unit testing bpel compositions. In *Test and Analysis of Web Services*. Springer, 2007, pp. 149–171.

[119] MA, C., DU, C., ZHANG, T., HU, F., AND CAI, X. Wsdl-based automated test data generation for web service. In *CSSE (2)* (2008), IEEE Computer Society, pp. 731–737.

[120] MARTIN, E., BASU, S., AND XIE, T. Websob: A tool for robustness testing of web services. In *ICSE Companion* (2007), IEEE Computer Society, pp. 65–66.

[121] MCILRAITH, S. A., SON, T. C., AND ZENG, H. Semantic web services. *IEEE Intelligent Systems 16*, 2 (2001), 46–53.

[122] MENASCÉ, D. A. Qos issues in web services. *IEEE Internet Computing 6*, 6 (2002), 72–75.

[123] MICHLMAYR, A., ROSENBERG, F., LEITNER, P., AND DUSTDAR, S. End-to-end support for qos-aware service selection, binding, and mediation in vresco. *IEEE T. Services Computing 3*, 3 (2010), 193–205.

[124] MICHLMAYR, A., ROSENBERG, F., PLATZER, C., TREIBER, M., AND DUSTDAR, S. Towards recovering the broken soa triangle: a software engineering perspective. In *IW-SOSWE* (2007), ACM, pp. 22–28.

[125] MIETZNER, R. *A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing*. PhD dissertation, University of Stuttgart, 2010.

[126] MIETZNER, R., AND LEYMANN, F. Generation of bpel customization processes for saas applications from variability descriptors. In *IEEE SCC (2)* (2008), IEEE Computer Society, pp. 359–366.

[127] MIETZNER, R., AND 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* (2008), IEEE Computer Society, pp. 3–10.

[128] MIETZNER, R., UNGER, T., AND LEYMANN, F. Cafe: A generic configurable customizable composite cloud application framework. In *OTM Conferences (1)* (2009), vol. 5870 of *Lecture Notes in Computer Science*, Springer, pp. 357–364.

[129] MODAFFERI, S., MUSSI, E., AND PERNICI, B. Sh-bpel: a self-healing plug-in for ws-bpel engines. In *MW4SOC* (2006), vol. 184 of *ACM International Conference Proceeding Series*, ACM, pp. 48–53.

[130] MOSER, O., ROSENBERG, F., AND DUSTDAR, S. Non-intrusive monitoring and service adaptation for ws-bpel. In *WWW* [66], pp. 815–824.

[131] OFFUTT, J., AND XU, W. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes 29*, 5 (2004), 1–10.

[132] PALSBERG, J., AND JAY, C. B. The essence of the visitor pattern. In *COMPSAC* (1998), IEEE Computer Society, pp. 9–15.

[133] PAPAZOGLOU, M. P. The challenges of service evolution. In *CAiSE* (2008), vol. 5074 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[134] PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S., AND LEYMANN, F. Service-oriented computing: State of the art and research challenges. *Computer 40*, 11 (nov. 2007), 38 –45.

[135] PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S., AND LEYMANN, F. Service-oriented computing: a research roadmap. *Int. J. Cooperative Inf. Syst. 17*, 2 (2008), 223–255.

[136] PAUTASSO, C., ZIMMERMANN, O., AND LEYMANN, F. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW* [66], pp. 805–814.

[137] PEYTON, L., STEPIEN, B., AND SEGUIN, P. Integration testing of composite applications. In *HICSS* (2008), IEEE Computer Society, p. 96.

[138] PSAIER, H., JUSZCZYK, L., SKOPIK, F., SCHALL, D., AND DUSTDAR, S. Runtime behavior monitoring and self-adaptation in service-oriented systems. In *SASO* (2010), IEEE Computer Society, pp. 164–173.

[139] PSAIER, H., SKOPIK, F., SCHALL, D., JUSZCZYK, L., TREIBER, M., AND DUSTDAR, S. A programming model for self-adaptive open enterprise systems. In *MW4SOC* (2010), ACM, pp. 27–32.

[140] REINECKE, P., AND 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, 2008), AMBER.

[141] ROSENBERG, F., PLATZER, C., AND DUSTDAR, S. Bootstrapping performance and dependability attributes of web services. In *ICWS* (2006), IEEE Computer Society, pp. 205–212.

[142] RUSSEL BUTEK. Which style of WSDL should I use? `http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/`.

[143] SCHALL, D., SKOPIK, F., PSAIER, H., AND DUSTDAR, S. Bridging socially-enhanced virtual communities. In *SAC* [67], pp. 792–799.

[144] SCHULTE, R. Research Note SPA-401-069 "Service Oriented' Architectures, Part 2". http://www.gartner.com/DisplayDocument?id=302869.

[145] SCHULTE, W. R., AND NATIS, Y. V. Research Note SPA-401-068 "Service Oriented' Architectures, Part 1". http://www.gartner.com/DisplayDocument?id=302868.

[146] SKOPIK, F., SCHALL, D., AND DUSTDAR, S. Managing social overlay networks in semantic open enterprise systems. In *WIMS* (2011), ACM, p. 50.

[147] SKOPIK, F., SCHALL, D., PSAIER, H., AND DUSTDAR, S. Social formation and interactions in evolving service-oriented communities. In *Proceedings of the 2010 Eighth IEEE European Conference on Web Services* (2010), ECOWS '10, IEEE Computer Society, pp. 27–34.

[148] SKOPIK, F., SCHALL, D., PSAIER, H., AND DUSTDAR, S. Adaptive provisioning of human expertise in service-oriented systems. In *SAC* [67], pp. 1568–1575.

[149] SNEED, H. M., AND HUANG, S. Wsdltest - a tool for testing web services. In *WSE* (2006), IEEE Computer Society, pp. 14–21.

[150] SOA SYSTEMS INC. WS-* Specifications. http://www.whatissoa.com/soaspecs/ws.php.

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

[152] TILLMANN, N., AND DE HALLEUX, J. White-box testing of behavioral web service contracts with pex. In *TAV-WEB* (2008), ACM, pp. 47–48.

[153] TREIBER, M., TRUONG, H. L., AND DUSTDAR, S. On analyzing evolutionary changes of web services. In *ICSOC Workshops* (2008), vol. 5472 of *Lecture Notes in Computer Science*, Springer, pp. 284–297.

[154] TREIBER, M., TRUONG, H. L., AND DUSTDAR, S. Semf - service evolution management framework. [65], pp. 329–336.

[155] TSAI, W.-T., CAO, Z., WEI, X., PAUL, R. A., HUANG, Q., AND SUN, X. Modeling and simulation in service-oriented software development. *Simulation 83*, 1 (2007), 7–32.

[156] TSAI, W.-T., CHEN, Y., CAO, Z., BAI, X., HUANG, H., AND PAUL, R. A. Testing web services using progressive group testing. In *AWCC* (2004), vol. 3309 of *Lecture Notes in Computer Science*, Springer, pp. 314–322.

[157] TSAI, W.-T., CHEN, Y., PAUL, R. A., LIAO, N., AND HUANG, H. Cooperative and group testing in verification of dynamic composite web services. In *COMPSAC Workshops* (2004), IEEE Computer Society, pp. 170–173.

[158] TSAI, W.-T., PAUL, R. A., SONG, W., AND CAO, Z. Coyote: An xml-based framework for web services testing. In *HASE* (2002), IEEE Computer Society, pp. 173–176.

[159] TSAI, W.-T., WEI, X., CHEN, Y., PAUL, R. A., AND XIAO, B. Swiss cheese test case generation for web services testing. *IEICE Transactions 88-D*, 12 (2005), 2691–2698.

[160] VAN DER AALST, W. M. P., AND TER HOFSTEDE, A. H. M. Yawl: yet another workflow language. *Inf. Syst. 30*, 4 (2005), 245–275.

[161] VERMA, K., AND SHETH, A. P. Autonomic web processes. In *ICSOC* (2005), vol. 3826 of *Lecture Notes in Computer Science*, Springer, pp. 1–11.

[162] VOGELS, W. Web services are not distributed objects. *IEEE Internet Computing 7*, 6 (2003), 59–66.

[163] WANG, Y., RUTHERFORD, M. J., CARZANIGA, A., AND WOLF, A. L. Automating experimentation on distributed testbeds. In *ASE* (2005), ACM, pp. 164–173.

[164] WHITE, S. R., HANSON, J. E., WHALLEY, I., CHESS, D. M., AND KEPHART, J. O. An architectural approach to autonomic computing. In *ICAC* (2004), IEEE Computer Society, pp. 2–9.

[165] WIKIPEDIA. Service-oriented Architecture. `http://en.wikipedia.org/wiki/Service-oriented_architecture`.

[166] WIKIPEDIA. WS-* Specifications. `http://en.wikipedia.org/wiki/List_of_web_service_specifications`.

[167] XU, W., OFFUTT, J., AND LUO, J. Testing web services by xml perturbation. In *ISSRE* (2005), IEEE Computer Society, pp. 257–266.

[168] YU, Y., HUANG, N., AND YE, M. Web services interoperability testing based on ontology. In *CIT* (2005), IEEE Computer Society, pp. 1075–1079.

[169] ZHENG, Y., ZHOU, J., AND KRAUSE, P. A model checking based test case generation framework forweb services. In *ITNG* (2007), IEEE Computer Society, pp. 715–722.

[170] ZHOU, X., TSAI, W.-T., WEI, X., CHEN, Y., AND XIAO, B. Pi4soa: A policy infrastructure for verification and control of service collaboration. In *ICEBE* (2006), IEEE Computer Society, pp. 307–314.

APPENDIX A

Code Examples

A.1 JavaCC Grammar Definition for GENESIS Plugin Alignment

```
options {
  STATIC=false;
}

PARSER_BEGIN(PluginOrchestration)

package at.ac.tuwien.vitalab.genesis.server.plugin.ast;

import at.ac.tuwien.vitalab.genesis.server.plugin.*;
import at.ac.tuwien.vitalab.genesis.model.config.*;
import java.util.*;
import java.io.*;

public class PluginOrchestration {

    private ATestbedConfiguration config=null;

    public static void main(String args[]) throws ParseException {
        PluginOrchestration parser = new PluginOrchestration(System.in);
        System.out.println(parser.Statement());
    }

    public static APluginTreeElement parse(String expression,
        ATestbedConfiguration config) throws ParseException {

        expression=dereference(expression,config);

        PluginOrchestration po=new PluginOrchestration(new StringReader(
            expression));
        po.config=config;
```

```
        return po.Statement();
    }

    public static String dereference(String expression, ATestbedConfiguration
         config) throws ParseException {

        PluginOrchestration po=new PluginOrchestration(new StringReader(
            expression));
        po.config=config;

        return po.DereferenceStatement();
    }

}

PARSER_END(PluginOrchestration)

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

// TOKEN definition taken from JavaCC example files
TOKEN :
{
  < BEHAVIOR: <IDENTIFIER> ( "." ( <IDENTIFIER> | <STRING_LITERAL>)  )? >
|
  < IDENTIFIER: <LETTER> (<PART_LETTER>)* >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
|
  < #LETTER:
      [  // all chars for which Character.isIdentifierStart is true
         "A"-"Z",
         "a"-"z"
      ]
  >
|
```

```
  < #PART_LETTER:
      [  // all chars for which Character.isIdentifierPart is true
         "0"-"9",
         "A"-"Z",
         "a"-"z"
      ]
  >
}



APluginTreeElement Statement() :
                                {   APluginTreeElement e;
                                }
{
    [   e=Expression()          {   return e;
                                }
    ]
    <EOF>                       {   return new EmptyElement();
                                }
}



APluginTreeElement Expression() :
                                {   APluginTreeElement result;
                                    Token p;
                                    APluginTreeElement e,f;
                                    Vector list=new Vector();
                                }
{
    (
        "{" e=Expression() "#" f=Expression() "}"       {   result=new
            TryCatchElement(e,f);
                                }
    |
        "[" e=Expression()      {   list.add(e);
                                }
        (
            LOOKAHEAD(2)
            "||" f=Expression() {   list.add(f);
                                }
        ) *
        "]"                     {   result=new ParallelElement(list);
                                }
    |
        "(" e=Expression()      {   list.add(e);
                                }
        (
            LOOKAHEAD(2)
            "->" f=Expression() {   list.add(f);
                                }
        ) *
```

```
        ")"                         {   result=new SequenceElement(list);
                                    }
    |
        p=<BEHAVIOR>                {   result=new PluginCallElement(p.toString()
            );
                                    }
    )                               {   return result;
                                    }
}



String DereferenceStatement() :
                                    {   String s;
                                    }
{
    [   s=DereferenceExpression()       {   return s;
                                    }
    ]
    <EOF>                           {   return "";
                                    }
}



String DereferenceExpression() :
                                    {   String result;
                                        Token p;
                                        String e,f;
                                    }
{
    (
        "{" e=DereferenceExpression() "#" f=DereferenceExpression() "}" {
            result="{"+e+"#"+f+"}";
                                    }
    |
        "[" e=DereferenceExpression()   {   result="["+e;
                                    }
        (
            LOOKAHEAD(2)
            "||" f=DereferenceExpression()      {   result+="||"+f;
                                    }
        ) *
        "]"                         {   result+="]";
                                    }
    |
        "(" e=DereferenceExpression()   {   result="("+e;
                                    }
        (
            LOOKAHEAD(2)
            "->" f=DereferenceExpression() {   result+="->"+f;
                                    }
        ) *
```

```
        ")"                         {   result+=")";
                                    }
    |
        p=<BEHAVIOR>                {   result=AWebServicePlugin.resolveReference
            (p.toString(),config);
                                    }
    )                               {   return result;
                                    }
}
```

A.2 Apache Velocity Template Files for GENESIS

Template for generating Web service code:

```
#set ( $serviceName = $service.getName())
#set ( $serviceNamespace = $service.getNamespace())

/* automatically generated Web service */

package $packagename;

import java.lang.*;
import javax.jws.*;
import javax.jws.soap.*;
import java.util.*;
import at.ac.tuwien.vitalab.genesis.server.*;
import at.ac.tuwien.vitalab.genesis.server.plugin.*;
import at.ac.tuwien.vitalab.genesis.model.*;

@WebService(name = "$serviceName",
            targetNamespace = "$serviceNamespace")
#if ($service.isRPCStyle())
@SOAPBinding(style = SOAPBinding.Style.RPC,
#else
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT,
#end
#if ($service.isLiteralUse())
use = SOAPBinding.Use.LITERAL,
#else
use = SOAPBinding.Use.ENCODED,
#end
#if ($service.isBareParameterStyle())
parameterStyle = SOAPBinding.ParameterStyle.BARE)
#else
parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
#end
public class $serviceName extends AWebService {

public $serviceName() {
}

#foreach ( $operation in $service.getOperations() )
#parse($operationTemplate)

#end
}
```

Template for generating Web service operation code:

```
#set( $input = $operation.getInputTypes() )
#set( $noOfInputs = $input.size() )
#set( $outputTypeName = $operation.getOutputType().getJavaClassName() )
#set( $behaviorTree = $operation.getBehaviorSyntaxTree() )

#if (!$operation.isHook())
#set ( $operationName = $operation.getName() )
@WebMethod
public $outputTypeName $operationName(
#foreach ( $inputType in $input )
   #set( $inputName = $inputType.getName() )
   #set( $inputTypeName = $inputType.getJavaClassName() )
   @WebParam(name="$inputName") $inputTypeName $inputName #if ($noOfInputs !=
        $velocityCount) , #end
#end
) throws Exception
#else
#set( $operationName = $operation.getHookName() )
protected void $operationName() throws Exception
#end
{
   LinkedHashMap<String,Object> arguments=new LinkedHashMap<String,Object>();

#foreach ( $inputType in $input )
#set( $inputName = $inputType.getName() )
   arguments.put("$inputName",$inputName);
#end

   final PluginContext context=new PluginContext(this,"$operation.getName()",
       arguments);

   try {
      callPlugins_$operationName(context);
   } catch (Exception e) {
      throw new Exception("Operation '$operationName' of service '
          $serviceName' threw exception: "+e.getMessage());
   }

#if ( "void" != $outputTypeName )
   if (context.getReturnValue()!=null) {
      return ($outputTypeName)MessageType.deserialize(context.getReturnValue
          (),${outputTypeName}.class);
   }
#if ( $operation.getOutputType().isSimpleType() )
   return $operation.getOutputType().getDefaultJavaExpression();
#else
   return ($outputTypeName)MessageType.createDummyObject(${outputTypeName}.
       class);
#end
#end
}
```

```
private void callPlugins_$operationName(final PluginContext context) throws
    Exception
{
$behaviorTree
}
```

A.3 Sample Configuration of GENESIS

```
<configuration>

  <!-- necessary plugins to simulate QOS processing time and service
      invocations -->
  <plugins>
        at.ac.tuwien.vitalab.genesis.server.plugin.QOSPlugin
        at.ac.tuwien.vitalab.genesis.server.plugin.InvocationPlugin
  </plugins>

  <!-- by default delay service operations by 2 seconds-->
  <defaultparameters
    qos_processingtime="2000"
  />

  <!-- by default we just simulate the delay -->
  <behavior>
    <QOS default="true">
      QOSPlugin.simulateDelay
    </QOS>
  </behavior>

  <schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
      qualified">
    <!-- types can be imported of defined inline -->
    <!-- <import name="SomeData" file="path/data.xsd"/> -->
    <xs:complexType name="somestructure">
      <xs:sequence>
        <xs:element name="a" type="xs:string" fixed="red"/>
        <xs:element name="b" type="xs:integer"/>
        <xs:element name="c" type="xs:boolean"/>
        <xs:element name="d">
          <xs:complexType>
            <xs:choice>
              <xs:element name="x" type="xs:double"/>
              <xs:element name="y" type="xs:string"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </schema>

  <servicetemplates>
    <service name="getAndCheckServiceTemplate">
      <deploy>
        <behavior>
          <!-- empty -->
        </behavior>
      </deploy>
      <undeploy>
        <behavior>
```

```
        <!-- empty -->
      </behavior>
    </undeploy>
    <operation name="getAndCheck" >
      <input>
        <name type="string"/>
      </input>
      <output type="somestructure"/>
      <behavior>
        (
          InvocationPlugin."return=dbService.getData(arg.name)"
        ->
          InvocationPlugin."checkService.checkData(return)"
        )
      </behavior>
    </operation>
  </service>
</servicetemplates>

<environment>
  <host address="http://localhost:8080/WebServices/GeneratorService" >
    <service name="dbService">
      <operation name="getData" >
        <!-- data retrieval takes 5 seconds -->
        <parameter name="qos_processingtime">5000</parameter>
        <input>
          <name type="string"/>
        </input>
        <output type="somestructure"/>
      </operation>
    </service>
    <service name="checkService">
      <operation name="checkData" >
        <!-- checking takes 0.5 seconds -->
        <parameter name="qos_processingtime">500</parameter>
        <input>
          <data type="somestructure"/>
        </input>
        <output type="void"/>
      </operation>
    </service>
  </host>

  <host address="http://localhost:8070/WebServices/GeneratorService" >
    <service name="getAndCheckService1" template="
        getAndCheckServiceTemplate">
    </service>
    <service name="getAndCheckService2" template="
        getAndCheckServiceTemplate">
    </service>
  </host>

  <host address="http://localhost:8060/WebServices/GeneratorService" >
```

```
      <service name="retrievalService">
       <operation name="retrieveAndProcess" >
         <input>
           <name type="string"/>
         </input>
         <output type="boolean"/>
         <behavior>
           {
             (
               [
                 InvocationPlugin."d1=getAndCheckService1.getAndCheck(arg.
                     name)"
               ||
                 InvocationPlugin."d2=getAndCheckService2.getAndCheck(arg.
                     name)"
               ]
             ->
               InvocationPlugin."return=retrievalService.processData(d1,d2)"
             )
           #
             InvocationPlugin."retrievalService.reportError(arg.name)"
           }
         </behavior>
       </operation>
       <operation name="processData" >
         <parameter name="qos_processingtime">1000</parameter>
         <input>
           <data1 type="somestructure"/>
           <data2 type="somestructure"/>
         </input>
         <output type="boolean"/>
       </operation>
       <operation name="reportError" >
         <parameter name="qos_processingtime">0</parameter>
         <input>
           <name type="string"/>
         </input>
         <output type="void"/>
       </operation>
     </service>
   </host>

 </environment>
</configuration>
```

Łukasz Juszczyk

Born: March 21, 1981 in Wrocław, Poland
e-Mail: mail@lukasz.at
Web: http://www.lukasz.at

Education

2006 - 2011 Doctoral Studies
Vienna University of Technology

2004 - 2006 Master Studies "Software-Engineering & Internet-Computing"
Vienna University of Technology

1999 - 2004 Bachelor Studies "Software- & Information-Engineering"
Vienna University of Technology

Published Books

1. Dustdar, S.; Schall, D.; Skopik, F.; **Juszczyk, L.**; Psaier, H. (2011).

   Socially Enhanced Services Computing - Modern Models and Algorithms for Distributed Systems

   Springer, ISBN 978-3-7091-0812-3

Published Papers

1. **Juszczyk L.**, Dustdar S. (2011).

   Automating the Generation of Web Service Testbeds using AOP

   9th IEEE European Conference on Web Services (ECOWS'11)

2. **Juszczyk L.**, Dustdar S. (2010).

   Programmable Fault Injection Testbeds for Complex SOA

   8th International Conference on Service Oriented Computing (ICSOC'10)

3. **Juszczyk L.**, Schall D., Mietzner R., Dustdar S., Leymann F. (2010).

   CAGE: Customizable Large-scale SOA Testbeds in the Cloud

   6th International Workshop on Engineering Service-Oriented Applications (WESOA). 8th International Conference on Service-Oriented Computing (ICSOC'10).

4. Psaier H., Skopik F., Schall D., **Juszczyk L.**, Treiber M., Dustdar S. (2010).

   A Programming Model for Self-Adaptive Open Enterprise Systems

   5th Workshop on Middleware for Service Oriented Computing (MW4SOC). 11th ACM/IFIP/USENIX Middleware Conference

5. Psaier H., **Juszczyk L.**, Skopik F., Schall D., Dustdar S. (2010).

   Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems

   4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10)

6. **Juszczyk L.**, Dustdar S. (2010).

   Script-based Generation of Dynamic Testbeds for SOA

   8th IEEE International Conference on Web Services (ICWS'10)

7. **Juszczyk L.**, Dustdar S. (2010).

   Testbeds for Emulating Dependability Issues of Mobile Web Services

   1st International Workshop on Engineering Mobile Service Oriented Systems (EMSOS). 6th IEEE World Congress on Services (SERVICES'10)

8. Treiber M., **Juszczyk L.**, Schall D., Dustdar S. (2010).

   Programming Evolveable Web Services

   2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS). 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)

9. **Juszczyk L.**, Psaier H., Manzoor A., Dustdar S. (2009).

   Adaptive Query Routing on Distributed Context - The COSINE Framework

   International Workshop on the Role of Services, Ontologies, and Context in Mobile Environments (ROSOC-M). 10th International Conference on Mobile Data Management (MDM'09)

10. Catarci T., de Leoni M., Marrella A., Mecella M., Vetere G., Salvatore B., Dustdar S., **Juszczyk L.**, Manzoor A., Truong H.-L. (2008).

    Pervasive and Peer-to-Peer Software Environments for Supporting Disaster Responses

    IEEE Internet Computing, January 2008.

11. **Juszczyk L.**, Dustdar S. (2008).

    A Middleware for Service-oriented Communication in Mobile Disaster Response Environments

    6th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC). 9th ACM/IFIP/USENIX Middleware Conference

12. Truong H.-L., **Juszczyk L.**, Bashir S., Manzoor A., Dustdar S. (2008).

    Vimoware - a Toolkit for Mobile Web Services and Collaborative Computing

    Special session on Software Architecture for Pervasive Systems, the 34th EUROMICRO Conference on Software Engineering and Advanced Applications

13. **Juszczyk L.**, Truong H.-L., Dustdar S. (2008).

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

    13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08)

14. Dustdar S., **Juszczyk L.** (2007).

    Dynamic Replication and Synchronization of Web Services for High Availability in Mobile Ad-hoc Networks

    Service Oriented Computing and Applications, Springer, Vol 1(1), 2007

15. Truong H.-L., **Juszczyk L.**, Manzoor A., Dustdar S. (2007).

    ESCAPE - An Adaptive Framework for Managing and Providing Context Information in Emergency Situations

    2nd European Conference on Smart Sensing and Context (EuroSSC'07)

16. **Juszczyk L.**, Michlmayr A., Platzer C., Rosenberg F., Urbanec A., Dustdar S. (2007).

    Large Scale Web Service Discovery and Composition using High Performance In-Memory Indexing

    IEEE Joint Conference on E-Commerce Technology and Enterprise Computing, E-Commerce and E-Services (CEC & EEE'07)

17. Catarci T., de Leoni M., De Rosa F., Mecella M., Poggi A., Dustdar S., **Juszczyk L.**, Truong H.-L., Vetere G. (2007).

    The WORKPAD P2P Service-Oriented Infrastructure for Emergency Management

    3rd International Workshop on Collaborative Serviceoriented P2P Information Systems (COPS). WETICE 2007 - 16th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises

18. **Juszczyk L.**, Lazowski J., Dustdar S. (2006).

    Web Service Discovery, Replication, and Synchronization in Ad-Hoc Networks

    IEEE Workshop on Dependability in large-scale service-oriented systems (DILSOS). ARES 2006 - 1st International Conference on Availability, Reliability and Security