Automated Performance Assessment for Service-Oriented Middleware

Domenico Bianculli*
Faculty of Informatics
University of Lugano
Lugano, Switzerland
domenico.bianculli@usi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Lugano, Switzerland
walter.binder@usi.ch

Mauro Luigi Drago
DEEP-SE group - DEI
Politecnico di Milano
Milano, Italy
drago@elet.polimi.it

USI-INF Technical Report 2009/07
November 2009

Abstract

Middleware for Web service compositions, such as BPEL engines, provides the execution environment for services as well as additional functionalities, such as monitoring and self-tuning. Given its role in service provisioning, it is very important to assess the performance of middleware in the context of a Service-oriented Architecture (SOA). This paper presents *SOABench*, a framework for the automatic generation and execution of testbeds for benchmarking middleware for composite Web services and for assessing the performance of existing SOA infrastructures. *SOABench* defines a testbed model characterized by the composite services to execute, the workload to generate, the deployment configuration to use, the performance metrics to gather, the data analyses to perform on them, and the reports to produce. We have validated *SOABench* by benchmarking the performance of different BPEL engines.

1 Introduction

Service-oriented Computing has received lot of attention both by industry, since more and more existing enterprise IT infrastructures have been migrated to SOAs [5], and by academia, which carries on several research projects in the area [17]. The implementation [7] of an SOA requires to put together several components — each one offering a specific functionality — that are aggregated under the name *service-oriented middleware*. Examples of functionality provided by a service-oriented middleware are service registration and discovery, assembly and execution of composite services, service monitoring, and service management. For example, in the context of Web services-based SOAs, some of these functionalities are provided by UDDI [15] registries and BPEL [16] engines.

It is then clear that the correct and efficient realization of an SOA heavily depends on the characteristics of the middleware infrastructure, since the latter represents the environment where services are deployed and executed. All major SOA vendors propose product suites that are actually a bundle of middleware components. Moreover, academic research has targeted specific facets of a service-oriented middleware, like "smart" service discovery, automated service composition, dynamic service rebinding, computation of service reputation, monitoring of functional properties and Service Level Agreements (SLAs), self-tuning, and self-healing.

This scenario reveals a huge amount of engineering activities, both industrial and academic, for the development of service-oriented middleware components. From an engineering point of view, one of the key activities in the development process is the validation of the product; in the case of service-oriented middleware, one validation task consists in assessing the performance delivered by the system. Middleware performance is a critical factor for service-oriented systems, since it affects the global Quality of Service (QoS) perceived by the end-users of the system. Moreover, besides being executed on the developers' site, this

*This work was carried out while the author was an intern at Mission Critical Technologies, Inc., on site at NASA Ames Research Center.

kind of test can also be executed on the stakeholders' site, for example by SOA analysts, when they have to choose a specific middleware component from several alternatives.

Performance assessment of a software component, at a minimum, consists in executing a series of tests, each one with a specific workload for the component, and collecting and aggregating some performance metrics that characterize the system. In the case of distributed systems, and thus also for service-oriented systems, the components can be deployed on different machines over different networks, and may need to be stimulated by different remote clients. This task, when performed manually or with a limited amount of automation, can be cumbersome and error-prone. Furthermore, in the case of distributed systems, the heterogeneity of the platforms and of the communication channels increase the complexity of this task. In this paper we address this problem by presenting *SOABench*, a framework to help SOA engineers and researchers evaluate the performance (in terms of some metrics, such as response time) of the middleware components that handle the execution of service compositions. In particular, we focus on Web service compositions described as BPEL processes, running on execution environments that consist of a BPEL engine and additional components such as service registry or enterprise service buses.

The main goal of this work is to enable further research in the service-oriented computing area, by providing a powerful, ready-to-use framework for automating experimentation with middleware components. Indeed, we foresee the application of *SOABench* for the evaluation and comparison of the performance of different BPEL engines, sporting different optimizations or featuring specific extensions, such as monitoring, reputation reporting, rebinding, self-tuning, or self-healing. Example research questions that *SOABench* helps answering are:

- *Which BPEL engine handles best the workload for given experiments on a given platform?*
- *What is the scalability of a certain BPEL engine, in terms of the maximum number of users and requests it can handle without showing failures?*

These questions can also seek an answer outside academia, for example in enterprise ICT environments, in the context of business decisions making regarding the implementation of SOAs in the enterprise. Additionally, *SOABench* could also be used by developers of service-oriented middleware components, such as BPEL engines, to evaluate the performance and the scalability of their products under various workloads.

The original, scientific contributions of the paper are twofold.

1) We present a flexible framework for automating performance assessment for service-oriented middleware, which automatically generates and executes testbeds for benchmarking middleware components. *SOABench* has been implemented using state-of-the-art technologies and is publicly available[1].

2) We leverage *SOABench* to evaluate performance and scalability of three popular BPEL engines and discuss how the engines react to certain kinds of workload. We show that one of the engines fails even under low workload, making it not suitable for use in production environments.

The rest of the paper is structured as follows. Section 2 gives an overview of *SOABench* and its components. Section 3 illustrates the testbed meta-model we use to characterize the experiments to be run on top of *SOABench*. Section 4 details how the internals of *SOABench* work. We report and comment about our experience in using *SOABench* in Section 5. Section 6 discusses the related work and Section 7 concludes the paper, outlining possible research directions.

2 Overview of SOABench

The *SOABench* framework is composed by two building blocks: a testbed modeling environment and a tool chain. Figure 1 shows the architecture of the framework, as well as how the components interact with each other.

The modeling environment comprises a meta-model, which establishes the concepts necessary to define the tests to run and the testbed to use. The performance engineer can then use it to design a testbed model for an SOA. Such a model is characterized by the composite services to execute, the workload to generate, the deployment configuration to use, the performance metrics to gather, the data analyses to perform on them, and the reports to produce.

[1] `http://code.google.com/p/soabench/`.

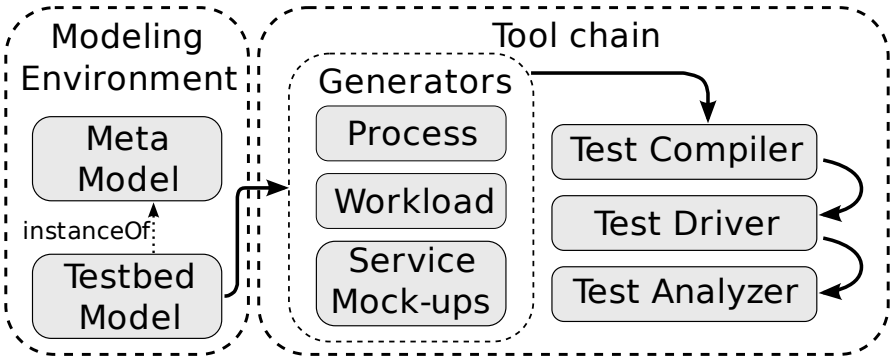


Figure 1: *SOABench* at a glance

After its definition, the model can be passed as input to the tool chain underlying *SOABench*, which includes four main components (see Figure 1). The first component is a set of *generators* for creating executable versions of processes, the mock-ups of the external services they interact with, and the testing clients that create (concurrent) workload by invoking processes. A *compiler* translates the testbed definition into a format understandable by the underlying platform we use for executing experiments. The *testbed driver* steers the experiment execution, and the *analyzer* gathers and processes measurement data, producing as output reports including statistics computed from the measurements.

2.1 Design Goals

Our framework has been designed to meet the following goals:

Technology independence. *SOABench* treats the BPEL engine and the infrastructure it interacts with as a black-box characterized by a generic, common interface, which acts as a wrapper for any BPEL-compliant infrastructure. Since platform dependencies — such as code specific to a certain BPEL engine for deploying test processes — constitute a major barrier to testability [18] for SOAs, we decided to keep them at a minimum degree, and introduced a plug-in mechanism to deal with them. Performance metrics are measured at the testbed infrastructure level, without the need for instrumenting the single components of the infrastructure (e.g., BPEL engines) by inserting profiling code. Technology independence ensures that the testbeds generated by *SOABench* are reusable in as many settings as possible. One direct consequence of this feature is the possibility to use *SOABench* for benchmarking any standard-compliant BPEL engine.

Model-driven approach. By using a model-driven approach we allow engineers to focus on the rationale of experiments and on the specification of the testbeds, since *SOABench* automatically takes care of low-level issues such as the deployment and the execution of test cases.

Repeatability. Repeatability is a key factor during experimentation. In the context of this work, repeatability is required to guarantee a fair comparison of different reference settings and to mitigate uncontrollable factors in the measurement environment. For example, benchmarking different BPEL engines requires to use the same workload, with the external services showing similar QoS behavior at each execution. However, due to complexity of the infrastructure, measurements are not exactly reproducible [8]. To compensate for measurements variances, simulations can be repeated several times and data can be reported by including statistical quantities over several runs, such as the median, the average and standard deviation, or confidence intervals.

Run-time evolution of testbeds. Some experiments require the testbed to evolve at run time, for example to support the addition of new services and process types in the system during experimentation. *SOABench* supports this feature by providing appropriate constructs to define run-time changes for services and processes.

Ability to process data and produce reports. During the execution of experiments, *SOABench* collects logs of run-time data related to the performance metrics specified in the testbed model. The model contains also the description of how to gather and process these data, as well as how to generate reports from them. The automatically produced documentation can be of great help supporting analysts and engineers in taking their decisions. To support extensibility, new metrics and report definitions may be added with a plug-in mechanism.

3 Testbed Model

The definition of the *SOABench* testbed meta-model has been inspired by similar work [19] in the area of (generic) distributed systems, and has been tailored to the service-oriented computing domain by relying on the personal experience of our research groups. The meta-model is depicted in Figure 2; solid lines represent composition associations while dotted boxes correspond to packages of the model.

In the model, the *System Under Test (SUT)* comprises atomic services, represented by the *Service* concept, and composite services, modeled by the *Process* concept; the instances of these two classes can interact with each other. Moreover, the SUT comprises also *Server*s, i.e., generic middleware components, on top of which *Service*s and *Processes* are executed. A *Server* component is deployed on a *Host*, i.e., a network-addressable physical machine. The *SUT* is stimulated with a *TestSuite*, which contains one or more *TestCase*s. For each *TestCase*, one or more *TestingClient*s are defined; a *TestingClient* is a lightweight remote application that executes a *Workload* on the *SUT*. A *TestCase* also includes a *DeployPlan* that allocates *Client*s and *Server*s on the *Host*s. Moreover, each *TestCase* specifies the performance *Metric*s of interest for engineers and which *Statistics* to compute for them.

In the rest of this section, we illustrate the main concepts of the meta-model.

3.1 Service

Atomic services constitute the basis of any SOA. They can be either invoked directly or accessed through a composite service, for which they play the role of external partners. We support the definition of different service types, i.e., different service interfaces represented by WSDL documents. For each service type, there may be a configurable number of instances.

Each instance of a service can be configured in terms of its non-functional attributes, such as *throughput*, *response time*, and *reliability*. These attributes can be described probabilistically, by specifying the associated probabilistic distribution (e.g., normal or uniform distribution) and its parameters.

SOAs are an instantiation of *open-world* software [1]. Indeed, they are characterized by a high level of dynamism; for example, new (instances of) services may become available while previously available (instances of) services may disappear or become unavailable. To support these characteristics, *SOABench* allows for specifying the occurrence of certain events related to a service life time:

- *Service unavailability*. It models the permanent unavailability of a service instance. In contrast to the reliability parameter — which models a temporary failure characterized by an exceptional response to a single request — a *service unavailability* event makes a service instance disappear permanently. This type of event has one parameter, representing the reference to the service instance to which it applies.
- *Availability of a new service*. It models the appearance of a new service instance in a registry. Its parameters are the type (interface) of the new service becoming available and a reference to a UDDI server, to be used for publishing the service.

Our previous work [4, 14] shows that these two kinds of events can be sufficient for experimenting with middleware extensions that deal with services rebinding/tuning or reputation reporting. However, new event types may be defined by extending the meta-model and defining a plug-in for *SOABench*.

3.2 Process

Atomic services can be composed together to realize more complex, added-value composite services, which we call *process*es, a synonym for business processes and service workflows. A process describes the control and the data flow realized among several atomic services. In *SOABench* we support two different definitions of composite services:

- *Black-box*. This kind of process hides the workflow it implements and makes only available the service interface by which it can be invoked. In a model, it is represented with a reference to the WSDL document describing its interface and with the endpoint reference at which it can be reached. Black-box processes are extremely useful to model legacy processes, which are often already deployed in an SOA infrastructure, and thus cannot be directly controlled by *SOABench*.

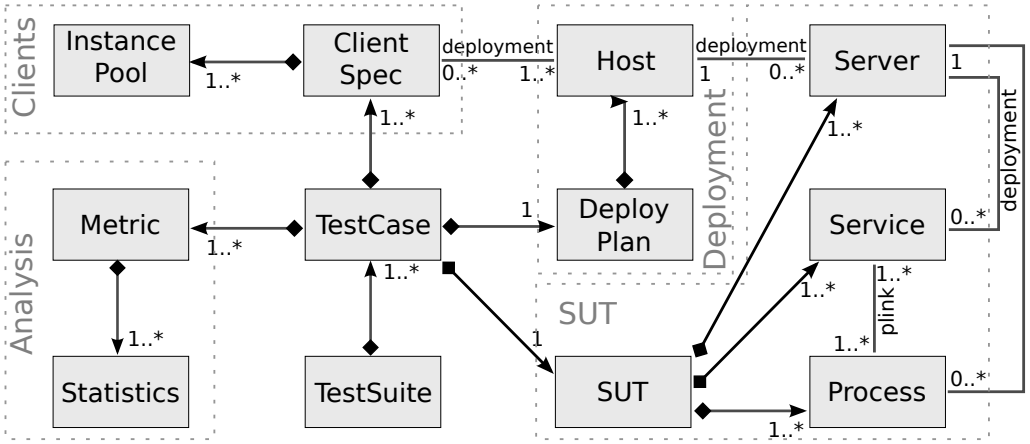


Figure 2: Extract of the meta-model for automated experimentation of SOAs

- *White-box*. For these processes, engineers get access both to their interfaces and to their implementations. The latter are modeled using a standard language (e.g., BPEL), without using any extensions specific to a certain orchestration engine. This requirement guarantees technology independence and relieves engineers of dealing with low-level details, such as deployment descriptors or other middleware-specific artifacts; hence, engineers can concentrate on the process business logic. In a model, a white-box process is represented with a reference to the BPEL file that defines its workflow and with WSDL files defining its own interface and the ones of its partner links; initial bindings for partner links are specified by references to service instances (defined elsewhere in the testbed model). White-box processes are often used to model test processes from scratch, for example to define a benchmarking suite for BPEL engines, like the one used in the experimentation reported in Section 5.

3.3 Client

Services and processes are invoked by service clients, also called testing agents, which execute, for each test case, the workload indicated in the model description. Specifying a workload extensionally, i.e., by specifying each single action to execute for realizing the workflow itself, can be tedious and error prone. To overcome this issue, we decided to adopt intensional definitions for workloads. In this way, workloads can be automatically generated by specifying a set of *client classes* and the number of clients to be generated for each class. Classes are characterized by the following parameters:

- *Process reference*. It is a reference to the process or to the service that should be invoked by the clients of the class.

- *Instance pool*: It is a set of data chunk instances, to be used as parameters when invoking processes. They are particularly useful for invoking black-box processes, since this class of processes may respond in an exceptional way if invoked with empty data [13]. The instances contained in the pool, besides specifying data chunks, are also characterized by the frequency with which they should be selected.

- *Type*. It represents the type of behavior to generate for a client. Using the terminology of Queuing Networks (QNs) [11], clients may execute either *open workloads* or *closed workloads*. In the former case, clients are characterized by an arrival rate, which indicates the interval between two subsequent requests. These clients do not wait for the completion of the requests already issued, before issuing new invocations. The number of requests circulating in the system is not bound *a priori*. *Open workloads* are appropriate to model service-based applications publicly available on the Internet. In the latter case (*closed workloads*), clients are instead characterized by a think time parameter. Clients wait for the completion of the requests already issued, before generating new invocations; between each completion and the subsequent invocation, a client waits for an additional period of time, equal to the think time.
  In both cases, the characterizing parameter (arrival rate for open workloads, think time for closed workloads) can be defined in a probabilistic way.

3.4 Deployment

The *Deployment* package of the meta-model defines the concepts for dealing with the intrinsic distributed nature of service-oriented systems. The distribution is both physical, i.e., at the level of the network infrastructure underlying an SOA, and logical, in terms of the deployment of clients, services and processes on the available physical resources. For this reason, the model is equipped with two distinct concepts, *Host*s and *Server*s.

*Host*s represent physical resources and are characterized by the network address at which they can be reached and by the credentials and protocol (e.g., SSH) to access it.

*Server*s represent middleware components running on resources, such as a BPEL engine. Several *Server*s may reside on a single *Host*. To support extensibility, we provide the ability to define new *Server Type*s with a plug-in mechanism. Each *Server Type* plug-in comes with the declaration of the model and a collection of scripts to start/stop the server and to compile/deploy services and processes on it. The actual deployment of services and processes onto *Server*s and of *Client*s onto *Host*s is defined in a *Deploy(ment) Plan*.

3.5 Analysis

The *Analysis* package of the meta-model defines the concepts to indicate the performance metrics of interest for the engineers using *SOABench*, as well as the statistical function to apply on the metrics for producing specific kinds of reports.

The current implementation of *SOABench* provides support for gathering and computing three metrics: a) the response time of processes and services, as measured by testing agents; b) the network traffic generated by a server and c) the number of threads created by a server. Additional metrics may be defined with a plug-in mechanism, by providing the scripts to gather and process monitoring logs.

For each performance metrics, it is possible to indicate the artifact for which it should be computed, e.g., a specific server or process. Furthermore, each metric specifies also which statistical functions to apply to it. Currently, *SOABench* supports basic aggregating functions, such as *maximum*, *minimum* and *average*, plus the possibility to plot and output a chart of the metric values measured during the experiment.

4 SOABench at work

When the *SOABench* tool chain receives a testbed model, it first processes (parses) it and then: 1) generates the mock-ups of the services, the actual service compositions in the form of BPEL processes, and the testing clients; 2) creates the deployment descriptors for the artifacts generated in the previous step and additional, supporting scripts for starting/stopping them; 3) compiles the model of the experiments in order to target the underlying execution platform, and executes the experiments defined in the testbed model; 4) gathers and processes the run-time data and produces a series of reports. This section discusses in detail all these steps.

4.1 Generation of Service Mock-ups

The *Service Mock-ups* generator parses the definitions of external services from the testbed model and subsequently generates the code that implements them. As mentioned in the previous section, services are described in terms of their interfaces, as described in WSDL documents. This means that the only information available for generating a service mock-up is represented by the syntactical interface it should have; no details are provided about its behavior. Although generating service mock-ups from descriptions with limited behavior information has been already dealt with in the literature [10], we decided to not include this functionality. Given the ultimate goal of our work, i.e., evaluating the performance of service-oriented middleware and SOA infrastructures, we are only interested in mocking the QoS characteristics of services.

Therefore, the generator creates the so-called *QoS-aware Service Mock-ups*: they are service mock-ups compliant with the syntactic interface, which ignore input data, produce default output data and manifest a deterministic QoS. Figure 3 outlines the steps for generating this kind of mock-ups.

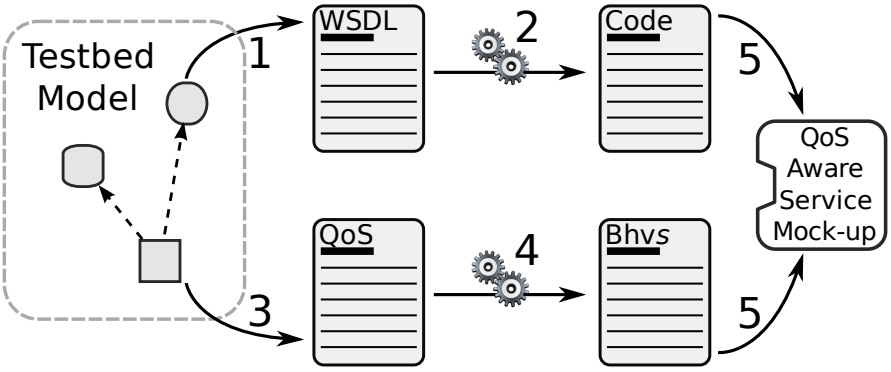


Figure 3: QoS-aware Service Mock-up generation process

We start with the WSDL description of the service, extracted from the testbed model (step 1), which is transformed into a code skeleton with tools such as *Wsdl2Java*[2] (step 2). Since the mock-up will not simulate the real behavior of the service, we can safely ignore to create the input data of the service. On the contrary, output data must comply with the data model specified in the WSDL document accompanying the service. To generate these data, we adopt an approach similar to the one described in [2]. For atomic types we use standard default values, such as the empty string for string types or false for boolean types. Structured types are instead represented as decision trees and a possible path from the root to one leaf is selected as the candidate instance of the chunk of data to return. Regarding QoS properties, they are extracted from the testbed model (step 3) and are then used for decorating (step 4) the skeleton mock-up with some code snippets that simulate non-functional properties, such as response time or reliability.

An important aspect in the design of the *Service Mock-ups* generator is to meet the requirements of *SOABench* in terms of repeatability of the experiments. At the level of service mock-ups, this requirement is met by using deterministic QoS to model their performance. By deterministic QoS, we mean that the same invocation during two runs of the same experiment should exhibit the same performance attributes. To achieve this, we first pre-simulate the behavior of a service; this simulation can be performed by extracting values from the probabilistic distribution specified in the model. The resulting traces are then stored and labeled with Unique Invocation IDentifiers (UIIDs), which represent single invocations of the service for which a behavior (trace) is available. The UIID is used in the mock-up implementation to look up for the pre-computed behavior to exhibit, so that two subsequent invocations with the same identifier will show the same non-functional behavior.

4.2 Generation of Processes

The *Process* generator transforms the technology-independent representation of processes into a format that is specific for the workflow execution engine indicated in the model. Moreover, it also prepares the deployment descriptor specific for the chosen engine, since it is very common that each SOA vendor adopts a non-standard format for defining how the deployment should be done.

This generator considers both the description of the process available in the testbed model and the derived BPEL workflow definition. These descriptions are then used to assemble a deployable artifact (e.g., a Java archive file for Active Endpoints ActiveVOS or a directory tree for Apache ODE) that contains the BPEL workflow, the WSDL documents for the external services and an automatically generated deployment descriptor.

These assemblies are created by build scripts defined for each specific target engine; new process engines may be added with a plug-in mechanism.

4.3 Generation of Testing Clients

The *Client* generator translates the testing client definitions found in the testbed model into a form suitable for execution and deployment. In particular, the output of this component is represented by an executable

[2]Included in the Apache Axis2 framework, `http://ws.apache.org/axis2/`.

program implementing a testing client and by a workload file containing the actions that should be executed by the client.

We provide a default implementation of the executable client program that can a) invoke a service or a process, both synchronously and asynchronously; b) wait for a certain amount of time between subsequent invocations; c) log the result of the invocations into a common format suitable for interpretation at run time.

Regarding the generation of the workload file, we include an additional program to simulate the run-time behavior of a testing client compliant with the *agent class* specified in the testbed model. This program simulates the invocation of a specific external service by randomly selecting data from *instance pools* and by waiting, between two subsequent calls, for the amount of time specified in the *agent class* definition. The output of the simulation is a time-stamped log that records the actions to perform over time, when the experiment takes place. At a higher level of abstraction, a workload definition is an ordered list of actions, each one to be performed at a certain time instant. At run time, testing clients read the log and execute, step by step, each action recorded in it, and thus they reproduce the simulated behavior during an actual experimentation.

4.4 Compilation and Execution

Instead of implementing our own platform for executing the experiments, we decided to use an existing solution, Weevil [19]. As will be described in Section 6, Weevil is a toolkit that can be used to generate distributed testbeds and execute experiments over them. It provides automated script construction, workload generation, experiment deployment, and experiment execution. These tasks are all required in *SOABench* and thus Weevil proved to be one of the best off-the-shelf solutions.

Even though Weevil also allows for generating testbeds, we only used its functionality for executing distributed experiments. The meta-model it defines for describing testbeds is general enough to be used for a wide range of distributed applications, and thus, at the same time, not very accurate for describing specific domains, such as the one of SOAs. This is also the reason for which we designed our meta-model and *SOABench*, at a higher, SOA-aware, level of abstraction.

However, Weevil has its own concepts and representation of software artifacts, which are slightly different from the ones used in *SOABench*. Therefore, an extra conversion step is required before using it; this step is carried out by the *Test Compiler* component of *SOABench*. Many concepts in our meta-model have a direct counterpart in the Weevil meta-model, so their translation is straightforward. This is the case, for example, with the *Host* and the *Testing Actor* concepts. Other concepts, such as *Components*, require extra effort to be translated. For example, each *Server* in our models is translated into a specific Weevil *Component*, for which we also provide the scripts to start up, shutdown and deploy services and processes. A similar transformation is undergone by our *Process* and *Service* concepts.

From a logical point of view, a Weevil experiment is a collection of software artifacts — either *Testing Actor*s or *Component*s — that should be deployed and run on *Host*s. In practice, after translating our model into a Weevil one, we generate the Weevil repository containing all the artifacts that need to be deployed for each *Host* and *Server*. This phase merely assembles all the artifacts created during the *testbed generation* phase with the appropriate scripts for starting and stopping them. At this point, Weevil takes care of running the experiments and gathering data; this last aspect is discussed in more detail in the next subsection.

4.5 Analysis and Reporting

Weevil provides a very limited support for manipulating experimental data, since it only gathers all the output generated by servers and the testing clients, without providing any data processing facility. To overcome this limitation, we included in *SOABench* a dedicated analysis and reporting component.

For each metric defined in our meta-model, we provide scripts to parse the experiment logs and load them into a relational database. After that, logs are processed and the statistics defined in the testbed model are computed.

To facilitate the interpretation of the results, we use the *Eclipse Birt* reporting system[3] to generate

[3]The Eclipse Birt Project, `http://eclipse.org/birt/`.

reports showing the outcomes of the analyses. We currently provide outcomes with different granularity. Coarse grained reports summarize the results of each test case contained in a test suite. Fine grained reports show instead detailed information about each test case. Moreover, for each metric defined in the testbed model, a summary report and a detailed report are generated for each test case.

5 Experience

In order to validate *SOABench*, we used it for benchmarking several BPEL engines, thus answering the research questions posed in Section 1. The BPEL engines we have considered in this benchmark are: a) Active Endpoints ActiveVOS v.5.0.2 Server Edition; b) Apache ODE v.1.3.3 and v.2.0-beta2; c) Red Hat JBoss jBPM-Bpel. In the rest of this section we detail the testbed model we used as well as the adopted testing infrastructure, and we present and discuss the results of our experimentation. For space reasons, the results presented here do not include data about thread usage and network resources consumption. The complete reports about the analyzed metrics are available on line on the web site of *SOABench*.

5.1 Description of the Testbed Model

We have defined four processes, each one characterized by the use of a specific structured activity of the BPEL language. In this way, we can analyze the performance of the BPEL engines with respect to the structural diversity and complexity of the processes they execute. Two processes have been defined in terms of the *flow* activity, while the other two use, respectively, the *sequence* and the *while* activity. The other structured activities defined in the BPEL standard have not been considered since they are subsumed by the ones that we have chosen: the *if* activity is a particular case of a *flow* activity with proper *transitionConditions*; the *repeatUntil* activity can be simulated with a *while* activity; the *forEach* activity can be simulated with the proper combination of *flow* and *sequence* activities.

Each process interacts with one partner service, which is modeled using a *QoS Aware Service Mock-up*. To minimize the impact of this kind of external services on the evaluation, we set their response time to zero.

Each of the processes invokes the operation made available by its partner service five times. How the five invocations are executed depends on the structure of the process. The process that uses a *sequence* activity (hereafter called "Sequential") executes five different *invoke* activities in a row. The process that uses the *while* activity (hereafter called "While") contains one *while* activity, whose body contains an *invoke* activity; the loop condition makes sure the loop is executed exactly for five times. The first process that uses a *flow* activity is called "FlowNoDep" since it does not define any synchronization dependency (expressed with a *link* construct) among the activities defined inside the *flow*. The other process is called just "Flow", since it does define dependencies among the activities that it encloses. We have modeled the dependencies in order to define an execution order that is equivalent to the one realized in the "Sequential" process. In both cases, the *flow* contains five *invoke* activities, all targeting the process partner service. Using two different kinds of *flow* structures in the two *flow* processes allows us to check how well the engines support parallel executions of activities (with the "FlowNoDep" process), and to test the overhead introduced by using a *flow* activity (with the "Flow" process).

Several test cases are defined using the processes described above, and are then assembled in two test suites that model closed type workloads; one suite has a "low" workload, while the other has a "high" workload. The "low" workload is characterized by a number of clients, ranging from 10 to 25, and by a think time that follows a normal distribution, with mean varying from 0.5 to 2 and variance always set to 0.5. The "high" workload adopts the same range for the think time, but varies the number of clients from 50 to 200. A summary of the workloads used in our experiments is shown in Table 1.

5.2 Testing Infrastructure

The middleware components required to execute the workflow processes and the external services have been deployed on two virtual machines, each one sporting three CPU cores at 2.40 GHz in exclusive use (i.e., not shared with other VMs), 4 GiB of memory and running Ubuntu Server 9.04 32 bit as operating system. The hosting environment of the virtual machines is a high-end server with four Dual Core Intel Xeon processors

Table 1: Workload summary

Load Type	Wkl ID	# Clients	Think Time $\sim \mathcal{N}(\mu, \sigma^2)$	
			μ	σ^2
Low	L1	10	1	0.5
	L2	25	2	0.5
	L3	25	1	0.5
	L4	25	0.5	0.5
High	H1	50	1	0.5
	H2	50	0.5	0.5
	H3	100	1	0.5
	H4	200	2	0.5

at 2.40 GHz, 32 GiB of memory, RAID-5 hard disks and VMware ESX as operating system. ActiveVOS and ODE (both versions) run under Apache Tomcat 5.5.25, which runs also Apache Axis2, required to execute the external services; jBPM-Bpel runs under JBoss 4.2.2-GA. The Java Virtual Machine installed on each server is the Sun-JVM-1.5.0.19 Server Edition. Application servers have been configured with the connection timeout set to 30 s and with the maximum size of the threads pool set to 700 threads[4]; these settings should minimize possible saturation issues. Since all the BPEL engines require access to a database, Oracle/Sun MySQL 5.0.75 has been installed on the servers hosting the BPEL engines. Finally, the deployment and the execution of the testing clients have been carried out on five machines, each one equipped with two Intel Xeon processors at 2.40 GHz, 4 GiB of memory, and running Ubuntu Server 9.04 32bit as operating system.

5.3 Active Endpoints ActiveVOS

The benchmarking results of ActiveVOS related to response time are shown in Table 2. The engine passed successfully all the "low" load test cases and exhibited some scalability issues only while executing the H3 test case, characterized by 100 concurrent clients. Further analysis indicates that this value represents its scalability limit. When the number of concurrent clients becomes greater than this threshold, the number of requests that fail increases rapidly; for example, during the H4 test case, the server saturated immediately and experiments were stopped manually. It needs to be said that the failures manifested were only related to time-outs in replying to clients, indicating that the server did not respond within the 30 s threshold; neither exceptions nor application error messages were logged during the experiments.

Figure 4 includes two plots that outline the trend of the response time during the "FlowNoDep" and the "Flow" experiments. For readability reasons, the plots show the trend for a limited time window over the entire execution of the test cases. The most interesting points — enclosed in a dotted circle — correspond to invocations that failed because the response time of the process exceeded the 30 s threshold. The data contained in Figure 4 and Table 2 allow us also to comment about how the engine deals with processes containing *flow* activities, with and without dependencies. As expected, in "low" load situations we can confidently say that *flow* activities with no dependencies (i.e., with all inner activities executed in parallel) have a better performance (almost twice faster) than the ones with dependencies (i.e., with a serialization of some of the inner activities).

This is not the case for "high" load test cases, where the performance gain becomes negligible: "FlowNoDep" is outperformed during the execution of H1 and H2, while the contrary has been measured during the H3 experiment. Moreover, the failure rate of "FlowNoDep" is greater than the rate of "Flow". We feel confident that the motivation for such results resides in the high number of concurrent requests handled by the Axis2 server, which hosts the external services. This is supported by the measured number of threads for this component: during the execution of "FlowNoDep" test cases, it is twice than the value measured during the execution of "Flow" test cases.

[4]This limit has been reached only while executing the heaviest test cases.

Table 2: ActiveVOS Response Time Results

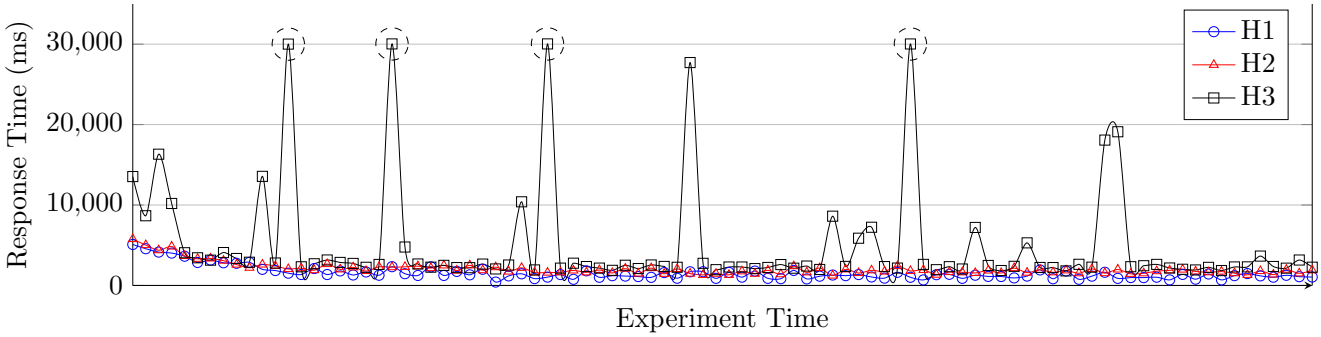
Test	Wkl ID	Max [s]	Min [s]	Avg [s]	Failure rate (%)
Sequential	L1	3.967	0.549	0.622	0
	L2	5.733	0.546	0.641	0
	L3	4.682	0.542	0.709	0
	L4	5.633	0.542	0.832	0
	H1	10.199	0.540	1.116	0
	H2	6.838	0.540	1.564	0
	H3	30.000	0.542	4.356	12.8
FlowNoDep	L1	3.997	0.129	0.215	0
	L2	6.969	0.126	0.268	0
	L3	5.246	0.126	0.418	0
	L4	5.280	0.125	0.753	0
	H1	10.443	0.124	1.526	0
	H2	10.126	0.340	2.070	0
	H3	30.000	0.127	3.522	11.6
Flow	L1	2.882	0.549	0.611	0
	L2	5.004	0.545	0.656	0
	L3	5.595	0.542	0.759	0
	L4	5.542	0.542	0.914	0
	H1	8.040	0.560	1.404	0
	H2	7.417	0.543	1.872	0
	H3	30.000	0.552	4.673	7.0
While	L1	3.510	0.550	0.613	0
	L2	5.027	0.546	0.634	0
	L3	5.428	0.541	0.706	0
	L4	5.380	0.542	0.820	0
	H1	6.448	0.542	1.102	0
	H2	8.115	0.563	1.524	0
	H3	30.000	0.542	4.020	13.4

Finally, when comparing the response time of the "Sequential" test case with the one of the "Flow" test case, one may notice that the former is 5% faster [9] than the latter. Such performance gain is ascribable to the overhead due to the creation of the additional threads required to execute a *flow* activity, and to check for the synchronization conditions.

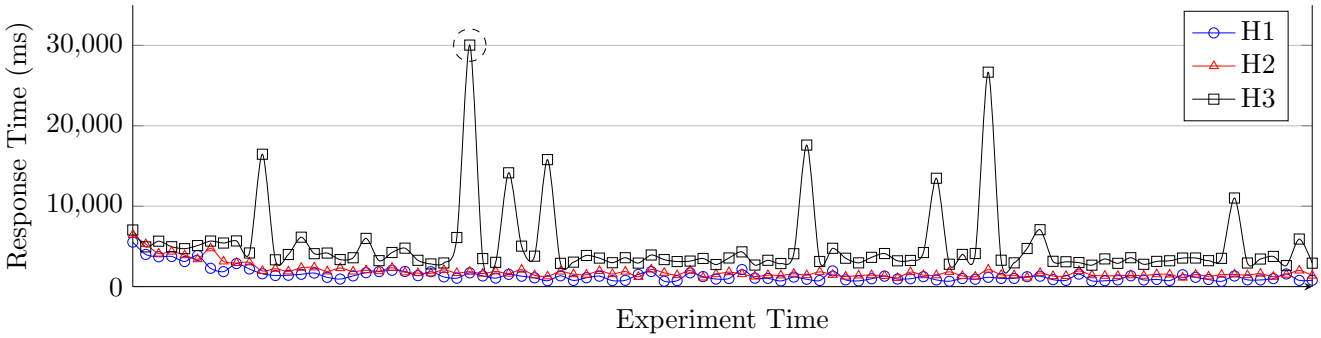
5.4 Red Hat JBoss jBPM

Table 3 reports the statistics of the response time for the jBPM engine. With respect to ActiveVOS, it exhibits worse response times. Moreover, the engine exhibits scalability problems even during the execution of test cases characterized by a "low" workload; failures where registered with only 25 concurrent clients. In the case of "high" workload test cases, the engine reached saturation after only 200 invocations, therefore it was not possible to continue the experiments; this is the reason why Table 3 contains only data related to "low" workload test cases.

We have investigated the possible causes of these failures and we observed that, when the server starts to saturate, the (numerous) exceptions that are thrown are about deadlock conditions and rollback operations in the database. These exceptions are ascribable to the Java Message Service (JMS) infrastructure of JBoss, which is used by jBPM to handle exchange of events and messages during the execution of the process.



(a) FlowNoDep - an excerpt of the experiments



(b) Flow - an excerpt of the experiments

Figure 4: ActiveVOS High Load Experimentation Results

In turn, JMS relies on the transactional support of the JBoss JavaEE container. At the JMS level, the delivery of a message may fail, for example because the receiver did not respond quickly enough, or because a deadlock was detected in the messages database. In all these cases, the transaction, within which the delivery operation is enclosed, is aborted, rollback actions are taken by the database, and the operation being executed is rescheduled in the future. This behavior confuses the engine, which reaches a saturation stage very quickly. In such a state, lot of requests accumulate, leading to a time-out failure, and the engine becomes incapable of serving both old and new process invocations.

Another evidence of this behavior is shown in Figure 5, which depicts the trend of the response time during "Sequential" experiments with a "low" workload; in the figure, the points of interest are surrounded with a dotted ellipse. After the first peaks, which correspond to timed-out requests, the server does not respond normally, and continues to stay in a saturation condition.

5.5 Apache ODE

We do not include any result for this engine since all test cases have failed, for both the versions we considered.

The stable version (ODE 1.3.3) failed to handle the test case with the lowest load defined in our benchmark suite, that is, the one with 10 clients and the think time taken from a $\mathcal{N}(1, 0.5)$ distribution. The engine hung and threw an exception after a low number of requests (about 30 invocations of the workflow process) from each client. This behavior is due to a known bug[5], which has been corrected in the beta version of the server.

However, also the beta version (ODE 2.0-beta2) could not handle the same workload, resulting in a similar, faulty behavior: after about 75 requests from each client, the server hung and started to throw some exceptions. By analyzing the stack trace, we have been able to identify the point where the failure occurs; it is located in ODE and not in the code of the application container. We have also discovered that, when the server starts to saturate, old process instances are never canceled; moreover, in the underlying database, they remain marked as active forever.

[5] `https://issues.apache.org/jira/browse/ODE-647`.

Table 3: jBPM Response Time Results

Test	Wkl ID	Max [s]	Min [s]	Avg [s]	Failure rate (%)
Sequential	L1	4.031	0.607	0.905	0
	L2	30.025	0.947	5.908	7.1
	L3	30.000	0.933	6.659	32.7
	L4	30.032	0.636	6.808	23
FlowNoDep	L1	8.261	0.721	2.135	0
	L2	28.612	1.481	7.976	0.1
	L3	30.000	1.500	8.357	0.9
	L4	30.000	1.318	9.111	1.6
Flow	L1	6.947	0.711	2.334	0
	L2	30.000	1.087	8.365	0.04
	L3	30.000	1.605	8.906	0.06
	L4	30.000	1.434	9.500	1.9
While	L1	4.573	0.622	1.034	0
	L2	30.000	0.633	4.487	0.08
	L3	30.000	1.008	5.455	0.03
	L4	30.000	0.964	5.676	0.07

5.6 Discussion

The results of the experiments performed with *SOABench* allow us to answer the two research questions raised in Section 1.

For the given experiments, on the given platform, ActiveVOS has been the engine with the lowest response time, under different workloads. jBPM has higher response times and can only deal with "low" workloads. Figure 4 displays the comparison between these two engines, in terms of response time.

In terms of scalability, ActiveVOS has a threshold equal to 100 clients, before starting to experience significant delays in replying to them, while jBPM supports at most 25 concurrent users.

As said above, no results can be provided for Apache ODE. We regret to admit that both the stable and the development version are only prototypes, which should not be used in production systems.

Before executing the experiments, we expected good results for ActiveVOS, given the industrial quality of this middleware. However, we we were surprised by the fact that, although jBPM is developed by the Red Hat company, its poor performance makes it not usable in real settings. We try to explain this by saying that its software house pushes (and develops products for) another language, alternative to BPEL, called jPDL.

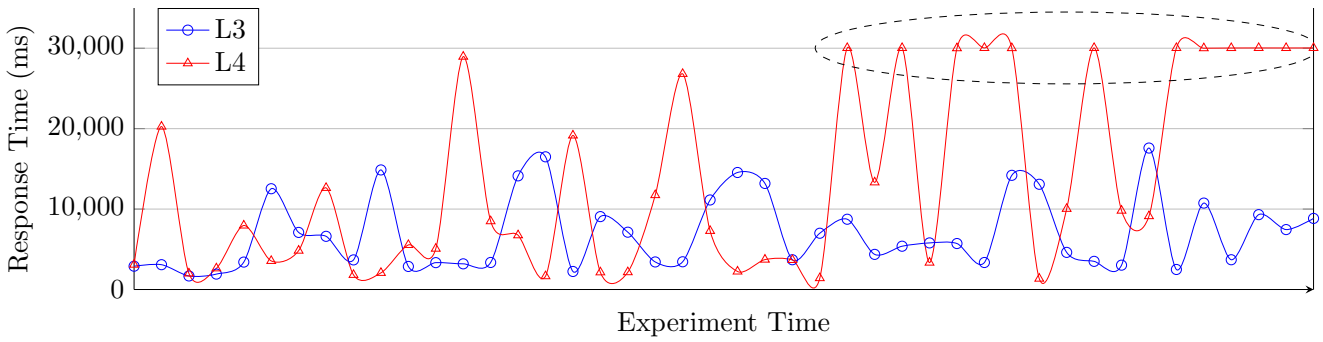


Figure 5: jBPM Sequential Test Case Results

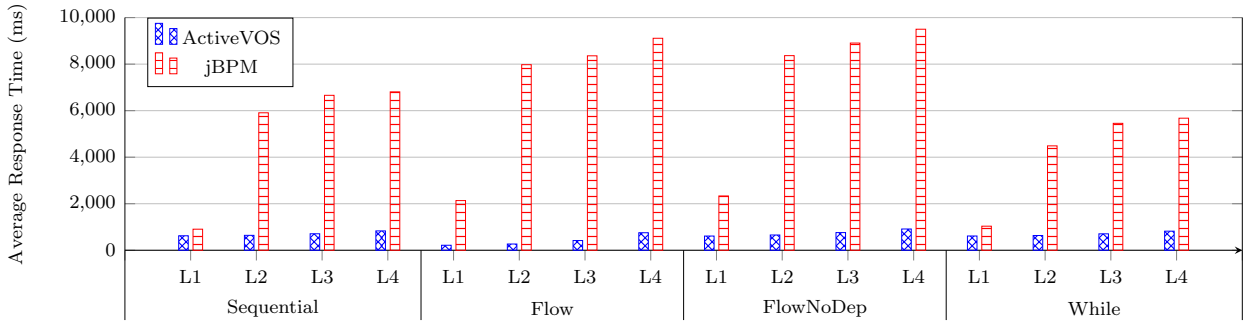


Figure 6: Comparison of the average response time between ActiveVOS and jBPM

5.7 Threats to Validity

Our empirical study may be subject to the following threats:

1. *The processes used in the experiment were very simple.* We used four BPEL processes, each one being defined in terms of one structured BPEL activity. Although such kind of processes may be good for an initial study, e.g., to identify how differently a BPEL engine deals with each of them, we believe that using more complex processes, implementing real business logics, would probably lead to more accurate results.

2. *Network latency is not modeled and thus taken into consideration during the experiments.* The current implementation of *SOABench* is not capable of handling and mitigating network latency problems. We have tried to reduce the network latency *a priori*, by conducting the experiments in a controlled environment, with a dedicated network infrastructure. No other traffic than the one generated by experiments was allowed through the network. Moreover, network communication among the virtual machines hosted on the same server has been handled by VMware through memory sharing. Therefore, we are confident that the impact of latency on the presented results is minor.

3. *VMware has been used to run the server hosting the SOA middleware.* Experiments should be re-run on physical machines to deduce the impact of virtualization on the results. However, VMware ESX is being currently used by companies in industrial settings, so we feel confident that its impact may be ignored.

4. *Impact of other middleware components.* In the experiments, we did not measure the impact of additional middleware components used by the evaluated engines. More than one instance of the Axis2 service container could have been used to reduce saturation impact of external services. Moreover, the DBMS used during the experiments could have been replaced with a different one, with better performance, such as Oracle Database or PostgreSQL.

5. *Java Virtual Machine version.* We had to use a rather old Java Virtual Machine for the experiments, since jBPM does not run on newer versions. More advanced compiler optimizations provided by more recent virtual machines could affect the outcome of our study.

6 Related Work

To the best of our knowledge, the work presented here is the first available framework for automating performance assessment for service-oriented middleware, such as BPEL engines and their extensions. Related work in the area shifts the focus of the experimentation towards the services running in an SOA, dealing with testbed generation for Web services.

For example, Genesis [10] is a testbed generator for complex Web services that provides support for generating and deploying the services, for simulating QoS metrics and steering the service execution by modifying execution parameters and QoS parameters using a plug-in mechanism. Genesis can be seen as a

complement to *SOABench*, since it generates a testbed of services that could execute when benchmarking service-oriented middleware. However, Genesis lacks some features, such as the generation of events (e.g., publication of a new service in a registry, which may trigger dynamic rebinding), accurate data analysis, and report generation.

Puppet [3] is a model-based generator of Web service stubs, which can be used before the actual deployment of a service, in order to test its behavior when interacting with externals services that are not available for testing. The automatically generated stub manifests the functional contract of an external service encoded as a state machine and its non-functional contract, as specified by an SLA written in WS-Agreement. Because the tool support provided by Puppet is limited to stub generation, deployment, test execution, and data analysis tasks cannot be automated.

Outside the realm of service-oriented computing, there have been many proposals for automating experimentation for distributed systems. For example, Weevil [19] is a framework that provides model-based configuration of testbeds, automated script construction, workload generation, experiment deployment and execution. Splay [12] sports features similar to Weevil and adds important extensions such as a specialized language for coding distributed algorithms and resource isolation in a sandbox environment. The design of these two systems emphasizes their intrinsic nature to support experimentation with distributed algorithms, which however has different requirements than performance assessment of (service-oriented) middleware, making their adoption impracticable in the service-oriented computing domain. For this reason, we have picked one of these systems (Weevil) and built *SOABench* on top of it, providing domain-specific abstractions for service-based systems.

The goal of promoting research in service-oriented computing by providing a framework for automating experimentation is also shared by the WorldTravel testbed [6]. The WorldTravel testbed represents a single service-oriented application, which operates on real data and can be invoked in order to experiment with SOA technologies. In contrast, the focus of our research is on automating performance assessment for service-oriented middleware in a wide range of possible scenarios.

7 Conclusion and Future Work

In this paper we have presented *SOABench*, a framework for assessing the performance of middleware components in service-oriented environments, by means of automated experimentation. In *SOABench*, testbeds are represented with a conceptual model tailored to the domain of SOAs, which captures the processes and the services composing the system, the workload to generate, and the performance metrics to compute. The experiments' deployment/execution/data reporting cycle is then automatically performed by *SOABench*, allowing engineers to focus on the analysis data returned by the testbed.

As case study, we have evaluated the performance of different BPEL engines using a new benchmark suite based on *SOABench*. We have explored the scalability limits of each engine and used the gathered data to investigate critical parts in the engines' implementations that were causing performance problems under high workload.

Regarding future work, we plan to extend the modeling capabilities for workloads and service mock-ups, by means of Queuing Networks models, to support more aspects such as network congestion or request bursts. Furthermore, we intend to improve the configuration management of the middleware, by allowing for automatically exploring the space of the configuration parameters and finding the best tuning. Finally, we also plan to progressively add support for other middleware components and/or vendors (e.g., for Oracle SOA Suite), for other performance metrics, as well as for different analysis methods.

Acknowledgments

This work has been partially supported by the European Community under the IDEAS-ERC grant agreement no. 227977-SMScom and the grant agreement no. EU-FP7-215483-S-Cube; by the Swiss SNF under grant agreements no. 125604 and 125337-CLAVOS.

References

[1] L. Baresi, E. D. Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[2] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based testing tool for web services. In *Proc. of ICST '09*, pages 326–335. IEEE Computer Society, 2009.

[3] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *Proc. of TESTCOM/FATES 2008*, volume 5047 of *LNCS*, pages 266–282. Springer, 2008.

[4] D. Bianculli, W. Binder, L. Drago, and C. Ghezzi. Transparent reputation management for composite web services. In *Proc. of ICSW 2008*, pages 621–628. IEEE Computer Society, 2008.

[5] J. Bosch, S. Friedrichs, S. Jung, J. Helbig, and A. Scherdin. Service orientation in the enterprise. *IEEE Computer*, 40(11):51–56, 2007.

[6] P. Budny, S. Govindharaj, and K. Schwan. Worldtravel: A testbed for service-oriented applications. In *Proc. of ICSOC 2008*, volume 5364 of *LNCS*, pages 438–452. Springer, 2008.

[7] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, 2005.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. of OOPSLA'07*, pages 57–76. ACM, 2007.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1992.

[10] L. Juszczyk, H.-L. Truong, and S. Dustdar. GENESIS - a framework for automatic generation and steering of testbeds of complex web services. In *Proc. of ICECCS 2008*, pages 131–140. IEEE Computer Society, 2008.

[11] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, 1984.

[12] L. Leonini, E. Rivière, and P. Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proc. of NSDI'09*, pages 185–198. USENIX Association, 2009.

[13] E. Martin, S. Basu, and T. Xie. Automated testing and response analysis of web services. In *Proc. of ICSW 2007*, pages 647–654, 2007.

[14] A. Mosincat and W. Binder. Transparent runtime adaptability for bpel processes. In *Proc. of ICSOC 2008*, volume 5364 of *LNCS*, pages 241–255. Springer, 2008.

[15] OASIS. UDDI specifications, 2004.

[16] OASIS. Web Service Business Process Execution Language Version 2.0 Specification, 2007.

[17] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, 2007.

[18] W. T. Tsai, J. Gao, X. Wei, and Y. Chen. Testability of software in service-oriented architecture. In *COMPSAC '06*, pages 163–170. IEEE Computer Society, 2006.

[19] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proc. of ASE '05*, pages 164–173. ACM, 2005.