A QoS Test-Bed Generator for Web Services

Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini

Istituto di Scienza e Tecnologie dell'Informazione - CNR
Via Moruzzi 1, 56124 Pisa - Italy
{antonia.bertolino, guglielmo.deangelis,
andrea.polini}@isti.cnr.it

Abstract. In the last years both industry and academia have shown a great interest in ensuring consistent cooperation for business-critical services, with contractually agreed levels of Quality of Service. Service Level Agreement specifications as well as techniques for their evaluation are nowadays irremissible assets. This paper presents Puppet (Pick UP Performance Evaluation Test-bed), an approach and a tool for the automatic generation of test-beds to empirically evaluate the QoS features of a Web Service under development. Specifically, the generation exploits the information about the coordinating scenario (be it choreography or orchestration), the service description (WSDL) and the specification of the agreements (WS-Agreement).

1 Introduction

The attractive promise of the Service Oriented Architecture (SOA) paradigm is to enable the dynamic integration between applications belonging to different enterprises and globally distributed across heterogeneous networks. Within the SOA paradigm, the most concrete technology is today realized by the Web Services (WSs).

Although WSs constitute a quite new drift in software application development, research in this technology has already evolved through a few stages. Initially the focus was in mechanisms which could allow for the loose interconnection among services independently developed and implemented on different machines. Such vision can only be achieved through the disciplined usage of standard notations and protocols, and in fact the WS domain is characterized by a strong boost toward standardization. Thus key achievements at this stage have been the establishment of common service descriptions, the definition of open service directories for storing and retrieving such descriptions, and the enactment of dynamic discovering and binding mechanisms. The technology for basic WS interconnection is now well established, with WSDL, SOAP and UDDI being just the most representative elements.

Nevertheless, the need soon arose of allowing for more complex scenarios, beyond simple point-to-point interactions [1]. The standardization of adequate mechanisms for services composition and interaction constituted thus the next stage, which is still very active. Two directions currently lead the scene within two different, but related, contexts [1]: the first aims at defining the composition of services (referred to as *orchestration*), the second at describing how related services should cooperate to perform a given

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 17–31, 2007.

task (*choreography*). Both interpretations of service integration provide a means to describe interacting scenarios. On one side, orchestration approaches foresee the availability of an execution engine that, by executing the code defining the orchestration, reproduces the specified interactions; as a fact, this need limits the applicability of orchestration to those cases in which a governing organization is in charge for defining the business process. In contrast, the choreography approach foresees the availability of a specification of the interactions to which the various services must conform, but it does not introduce *per se* any mechanism for forcing such interactions. Currently, the most significant proposals concerning the specification of WS orchestration and choreography are represented by the Business Process Execution Language (WSBPEL) [18] and the Web Services Choreography Description Language (WS-CDL) [22], respectively.

Eventually, the openness of the environment characterizing the SOA paradigm naturally led to the pursuit of mechanisms for specifying the provided levels of Quality of Service (QoS) and establishing an agreement on them, in line with the widely accepted idea that service delivery cannot just focus on functional aspects, and ignore QoS-related properties.

Indeed, not only for Service Oriented systems, but for many other kinds of enterprise applications [6] [20], communication networks and embedded systems [4], solutions that do not put adequate consideration of non functional aspects [17] are no longer acceptable. Correspondingly, in recent years much research has been devoted to methodologies for QoS evaluation, including *predictive* and *empirical* techniques [13]. Predictive approaches are crucial during the design and the development of a software system, to shape the quality of the final product [20]: they perform analytical QoS evaluation, based on suitable models, such as Petri Nets or Queueing Networks. But increasingly modern applications are deployed over complex platforms (i.e., the middleware), which introduce many factors influencing the QoS and not always easy to model in advance. In such cases, empirical approaches, i.e., evaluating the QoS via run-time measurement, could help smoothing platform-dependent noise. However, such approaches require the development of expensive and time consuming prototypes [15], on which representative benchmarks of the system in operation can be run.

In this last direction, however, when computer-processable specifications exist, and *code-factories* can be used to automatically generate a running prototype from a given specification, there is large room for the adoption of empirical approaches. In particular, and this is the position we take in this work, given the high availability of standardized computer processable information, WSs and related technologies [2,10,18,22,14] yield very promising opportunities for the application of empirical approaches to QoS evaluation.

According to this intuition, in this paper we introduce an approach, called Puppet (Pick UP Performance Evaluation Test-bed), which realizes the automatic derivation of test-beds for evaluating the desired QoS characteristics for a service under development, before it is deployed. In particular, we are interested in assessing that a specific service implementation can afford the required level of QoS (e.g., latency and reliability) when playing one of the roles in a specified choreography or when used in composition with other services (orchestration). To this purpose, Puppet relies on the availability of the QoS specification of both the service under evaluation and the interacting services. Such

assumption is in line with the increasing adoption of formal contracts to establish the mutual obligations among the involved parties and the guaranteed QoS parameters, which is referred to as the Service Level Agreement (SLA) for WSs.

In the next section we provide a basic background on the emerging languages for the definition of SLAs. Then, in Sec. 3 we illustrate the general scenario in which the Puppet tool should be employed. Successively, in Sec. 4 we describe the approach and its logical architecture. An exploratory example is presented in Sec. 5 while related work is summed up in Sec. 6. In Sec. 7 we draw conclusions and hint at future work.

2 Specification of Service Level Agreements

An important ingredient of the SOA paradigm is the QoS level agreement specifications among interacting services.

Traditionally, agreements were expressed informally, not in machine-readable form. In software engineering quite basic notions of agreements were established by means of Interface Description Languages [17]. Concerning the WS technology, Service Level Agreements (SLAs) represent instead one of the most interesting and actively pursued issues. SLAs aim at ensuring a consistent cooperation for business-critical services. Relevant experiences in this direction are certainly represented by work around Web Service Level Agreement (WSLA) [14] or SLAng [19].

The approach introduced in this paper has been conceived to be as independent as possible of a specific SLA language. Indeed, concerning the goal of the work, any SLA languages predicating on the concepts we are considering are equivalent. However, when it comes to developing a specific implementation of our conceptual environment, we obviously need to consider a specific technology. Hence, in the remainder of the paper, we will focus on a proof-of-concept development carried on using the WS-Agreement language [10]. To make the paper self-contained, we report below the background notions behind its current proposal.

WS-Agreement is a language defined by the Global Grid Forum (GGF) aiming at providing a standard layer to build agreement-driven SOAs. The main assets of the language concern the specification of domain-independent elements of a simple contracting process. Such generic definitions can be augmented with domain-specific concepts.

As shown in Fig. 1, the top-level structure of a WS-Agreements offer is expressed by means of a XML document which comprises the agreement descriptive information, the context it refers to and the definition of the agreement items.

The Context element is used to describe the involved parties and other aspects of an agreement not representing obligations of parties, such as its expiration date. An agreement can be defined for one or more contexts.

The defined consensus or obligations of a party core in a WS-Agreement specification are expressed by means of Terms. Special elements (e.g., AND/OR/XOR operators) can be used to combine terms, via the specification of alternative branches or the nesting within the terms of agreement.

The obligation terms are organized in two logical parts. The first specifies the involved services by means of the Service Description Terms. Such part primarily describes the functional aspects of a service that will be delivered under an agreement. A

```
<wsag:Agreement
  AgreementId=xsd:string>
  <wsag:Name>
    xs:NCName
  </wsag:Name>
  <wsag:AgreementContext>
    wsag:AgreementContextType
  </wsag:AgreementContext>
  <wsag:Terms>
    wsag:TermCompositorType
  </wsag:Terms>
</wsag:Agreement>
```

Fig. 1. WS–Agreement Structure

term for the service description is defined by means of its name, and the name of the service which it refers to. In some case, a domain-specific description of the service may be conditional to specific runtime constraints. A special kind of Service Description Terms is the *Service Reference*, which defines a pointer to a description of a service, rather than describing it explicitly into the agreement. The second part of the terms definition specifies measurable guarantees associated with the other terms in the agreement and that can be fulfilled or violated. A Guarantee Term definition consists of the obliged party (i.e, *Service Consumer*, *Service Provider*), the list of services this guarantee applies to (Service Scope), a boolean expression that defines under which condition the guarantee applies (Qualifying Condition), the actual assertion that have to be guaranteed over the service (Service Level Objective) and a set of business-related values (Business Value List) of the described agreement (i.e., importance, penalties, preferences). In general, the information contained into the fields of a Guarantee Term are expressed by means of domain-specific languages.

3 A WS Development and Evaluation Scenario

As explained in the Introduction, initially WSs were intended for loose and basic interactions, but soon the need for mechanisms to describe more complex integration of services emerged. The basic assumption of our approach is that such a description indeed exists. This is not an unrealistic assumption, as the global definition of applications resulting from the dynamic integration of unrelated services is seen as one of the most relevant factors at the basis of the take-off of the Service Oriented paradigm.

Our view[1] is that the integration of services offers major guarantees, and will be fostered, by the existence of predefined choreographies and the definition of orchestration. Given a definition of the integration of different services, based on choreography or orchestration, our objective is to provide a tool to support the QoS evaluation of services to be integrated, but still under development. The scenario we envisage is depicted in Fig. 2.

[1] This view is shared within the EU FP6 Strep n.26955 - PLASTIC, see at http://www.ist-plastic.org

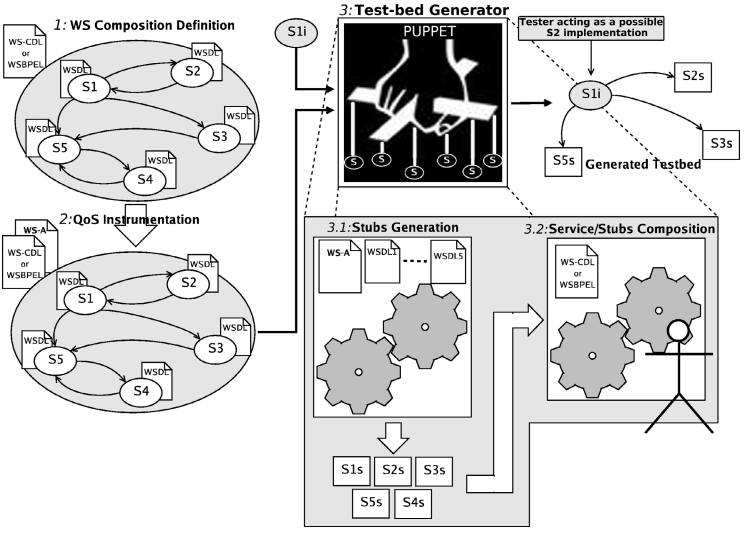

Fig. 2. The Puppet approach and supporting tool

The first step in this process, referred to as "1: WS Composition Definition" in Fig. 2, is indeed the specification of service integration, in terms of WS-CDL or WSBPEL, and of a set of WSDL descriptions defining the interfaces of the services involved in the interaction.

The process continues with the annotation of the composition with QoS attributes for each service involved in the integration. In Fig. 2 this step is referred to as "2: QoS instrumentation". For this step we distinguish between the case of a choreography and that of an orchestration. In the former case, the organization that released the specification of the composition is in charge of augmenting the specification with QoS attributes. Developers of services will take such a specification as a reference for their implementation, expecting that their required services do the same. As a consequence each developer of a service is interested in evaluating that it actually can provide the required service according to the specified QoS. If this is not true we can expect that no other service in the given choreography will agree on binding to such service.

In the case of an orchestration, the derivation of QoS values for the services to be integrated is quite different. In this case the service that will be the subject of the validation is actually the orchestrated service. The developer company is interested in deriving a competitive service in terms of provided QoS. In this case the parameters for the QoS could not have been defined by someone else, i.e. a "standard" body as in the case of a choreography, but should be derived by the QoS defined by similar services possibly registered within a directory. Moreover in order to have a reliable picture of the final run-time environment the developer should also consider the QoS defined for the services that will be composed in the orchestration. Therefore retrieving from the directory

service the QoS specification for similar services, and the QoS provided by the composed services it is possible to get a quite trustable picture of the run-time conditions for the case of orchestrating services.

Summarizing, Puppet assumes the availability of a WSDL specification for each service, a definition of a composition in terms of WS-CDL or WSBPEL, and a WS-A description for the services in the composition/coordination. At this point, Puppet can automatically generate a test-bed to validate the implementation of a service before its deployment in the target environment (referred as "3: Test-bed Generator" in Fig. 2). The test-bed will consist of fake versions of the used services and of the possible clients. Such services are successively composed by the tester in order to reproduce different runt-time conditions for the service under evaluation, as discussed in the following.

As illustrated in Fig. 2, step 3 consists of two different phases. The first one is the generation of the stubs simulating the non functional behavior of the services in the composition, and referred as "3.1: Stubs Generation" in Fig. 2. The second one, referred as "3.2: Service/Stubs Composition" in Fig. 2, foresees the composition of the implementation of a service, called "S1i" in Fig. 2, with the services with which it will interact. The next paragraphs provide a short introduction to both phases, that will be successively described in detail in Sec. 4.

The generation of the stubs proceeds through two successive steps (not detailed in the figure). In the first one a skeleton of the stubs is generated starting from the WSDL description. At this time the generated skeletons contain no behavior. Hence, in the second step the implementation is "filled" with some behavior that by construction fulfills the required non functional properties, for the corresponding service. The necessary information is retrieved from the WS-A specification and used to apply automatic code transformation according to rules that we have defined and we describe in Sec.4. At the end of the "Stubs Generation" phase, a set of stubs providing the services specified in the composition according to the desired properties is available.

In turn Puppet permits also to derive stubs simulating the behaviour of possible clients for the service under evaluation as "S2" in Fig.2. To generate such stubs Puppet considers the part of the WS-A document defining the constraints among the client and the service. Possible constraints can be for instance the number of invocations within a time frame, or a minimum time between successive invocations, and similar ones.

The "Service/Stubs Composition" phase implements the final setting of the test-bed. Goal of this step is to derive a complete environment in which to test the service. To this purpose, Puppet composes the service under test, "S1i" in Fig. 2, with the required services and according to the composition specified in the choreography or in the orchestration. At the same time client stubs are composed, by the tester, to derive a meaningful workload for the service under evaluation. Currently this phase requires the assistance of a human agent, as illustrated by the presence of a stick man in Fig. 2, that has to hand-code the composition in the service stubs. Nevertheless we are working on further automatizing the process on the base of the forthcoming final WS-CDL specification and WS-BPEL specification.

Concluding, the final product provided by the Puppet tool is an environment for the QoS validation of a composite service. The evaluation will require the development of a tester that integrates the client stubs generated by Puppet, in order to reproduce

meaningful workloads. Such a tool will have to verify that the properties specified in the QoS document for the service under evaluation are fulfilled. In Fig. 2 also such a tool is shown, nevertheless how this component can be derived is not within the scope of our work and we refer to the literature on the argument [8] for possible approaches.

4 Description of the Approach

The inspiring idea behind Puppet is that the technologies introduced within the WS infrastructure make it possible to automatically generate a test-bed environment for a service. The generated environment can then be used to test if the specified QoS properties (e.g. performance) will be respected by the service under development after its deployment in the final environment.

Specifically, the generation exploits the information about the coordinating scenario (be it choreography or orchestration), the service description (WSDL) and the specification of the agreements that the involved *roles* will abide (see Sec. 3). Tools and techniques for the automatic generation of service skeletons, taking as input the WSDL descriptions, are already available and well known in the Web Services communities [2]. Nevertheless such tools only generate an empty implementation of a service and do not add any logic to the service operations. Puppet exploits and improve those solutions by processing the empty implementation of a service operation and augmenting it with fake code resulting from the appropriate transformation of the SLA specification.

4.1 Skeleton Generation Process

In Puppet we automatically generate service stubs whose behaviors are derived from the terms defined in a WS-Agreement document. The UML Component Diagram in Fig. 3 outlines the architecture we propose. In the picture we directly refer to Apache-Tomcat/Axis [2] as the technology used for the derivation of the various intermediate artifacts needed for the derivation and deployment of the generated services stub. Nevertheless, the approach is not bound to a particular technology and other solutions are possible: the only requirement is to identify the corresponding tools in the chosen platform with respect to Apache-Tomcat/Axis. Any Web Services platform provides in fact some WSDL compiler permitting to automatically derive the different harness needed both for the deployment of the service and for enabling service clients to invoke the published service operations [1].

More specifically, the generation of a QoS stub service simulator for the service S1 in Fig. 3 undergoes three main phases: service skeleton structure definition, QoS behavior generation, service stub deployment.

The first step in the process is directly performed exploiting the Apache-Axis *WSDL-2Java* utility [2]. Such tool, taking as input a WSDL description of a service, generates a collection of Java classes and interfaces according to the abstract part of the specification. Thus, for each binding a service skeleton structure will be automatically defined and released. At the same time the tool generates both a deployment and an undeployment descriptors. Such descriptors can be identified by the extension WSDD (Web Service Deployment Descriptor) [2]. The deployment specification represents the contact

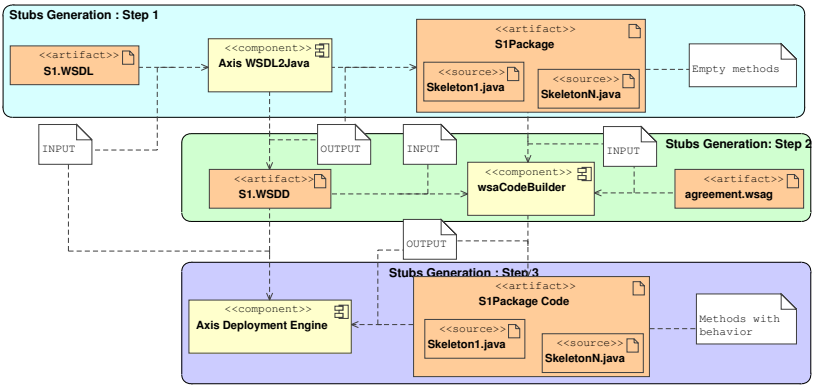

Fig. 3. Puppet Test-bed Generator Logical Architecture

point between the abstract definition of the service, expressed into the WSDL, and the corresponding concrete implementation of the endpoints coded into the Java skeletons.

Thus far, no behaviors are coded into the skeletons, but only the operations they export are derived. According to a set of specified transformation rules from WS-Agreement to Java and the relations in the deployment WSDD file, the *wsCodeBuilder* then generates the simulation code and inserts it into the proper operations. At the time of writing a running implementation of the *wsCodeBuilder* unit has been developed.

The last step of the process concerns the deployment of the services simulating the selected QoS. The deployment descriptor coming from the first phase and the new version of the skeletons are then used as input for the *Axis Deployment Engine*.

For the sake of completeness, it is important to remark that the generation of meaningful stubs would require to also handle their return values. Such values could in fact be part of a parametric QoS specification or influence the behavior of the tested service. The current implementation of Puppet does not provide a general solution, but returns values arbitrarily chosen among a set previously built for each possible data type. If no behavioral contract has been defined for the service this should be considered correct by the service under test. Nevertheless, within the WS community there is a great boost toward the definition of functional contracts for services [12] [5] [3]. To manage such situations, future releases of Puppet will integrate mechanisms that enable the instrumentation of stub services with code that returns conforming value with respect to the associated functional contract [21] (a related problem is that the determination of the return values must be low effort-intensive not to invalidate the evaluated QoS).

As above mentioned, the current version of the tool needs human support for the setting of the choreography. This means that the binding among services under development and stubs is manually derived from the WS-CDL or WSBPEL.

4.2 Matching WS-Agreement Statements to Java Code

Puppet can handle those QoS constraints that can be simulated by means of a parameterizable portion of code. The approach implemented in Puppet requires then that for

Table 1. Service Level Objective Mapping for Latency

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:Latency>
   <puppet:TagDelay>    1000
   </puppet:TagDelay>
   <puppet:Ditribution> normal
   </puppet:Distribution>
  </puppet:Latency>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
try{
 Random rnd = new Random();
 float val = rnd.nextFloat();
 int sleepingPeriod = Math.round(val*1000);
 Thread.sleep(sleepingPeriod);
}
catch (InterruptedException e)  {}
...
```

each concept in a SLA language a precise mapping must be provided. This is clearly a quite complex task; nevertheless given a specific language and a possible interpretation of the corresponding statements, it has to be done only once and for all. Currently we have defined a simple language for SLA that can predicate over several QoS characteristics (latency, reliability, workload) and have set a precise mapping for this concepts, defined in XML format, to composable Java code segments.

The mapping between the XML statements of the WS-A and the Java code has been specified in a parametric format that is instantiated each time one occurrence of the pattern appears. The examples reported in the remainder of this section show the transformations we have defined and encapsulated in Puppet.

In particular, conditions on latency can be simulated introducing *delay* instructions into the operation bodies of the services skeletons. For each Guarantee Term in a WS-Agreement document, information concerning the maximum service latency is defined as a Service Level Objective according to a prescribed syntax. The example in Tab. 1 reports the XML code for a maximum latency declaration of $1000mSec$ normally distributed and the correspondent Java code that Puppet will automatically generate.

Table 2. Service Level Objective Mapping for Reliability

```
...
<wsag:ServiceLevelObjective>
  <puppet:PuppetRoot>
   <puppet:Reliability>
    <puppet:TagRate>   99.50
    </puppet:TagRate>
    <puppet:Window>    2000
    </puppet:Window>
   </puppet:Reliability>
  </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
if (this.possibleFailureInWindow()){
 Random rnd = new Random();
 float val = rnd.nextFloat()*100;
 if ( val>99.50f) {
  String fCode = "Server.NoService";
  String fString="No target service to invoke!"
  org.apache.axis.AxisFault fault = new
     AxisFault(fCode,fString,"",null);
  this.incNumberOfFailure(); throw fault;  }
}
...
```

Constraints on services reliability can be declared by means of a percentage index into the Service Level Objective of a Guarantee Term. Such kind of QoS can be reproduced introducing code that simulates a service container failure. Thus, given the size of a sliding window defining the time interval within which reliability is measured, the generated stub for a service will raise remote exceptions according to the specified rate in the window. The XML code in Tab. 2 provides an example of the transformation for

reliability constraint description, assuming that the Apache-Tomcat/Axis [2] platform is used.

The case of workload can be simulated by equipping the generated skeletons with client-side code for the automatic invocation of the service under evaluation. Currently Puppet permits to describe the maximum number of requests that the clients can deliver to the service in a given period (i.e. WinSize in Tab. 3). The generation process augments the stubs with a private method for the remote invocation (i.e. invokeService in Tab. 3) and an exported public method that triggers the emulation request stream. The transformation in Tab. 3 reports the code for the trigger method. The information required for the instantiation of parameters such as the target endpoint is obtained from the part of the guarantee term concerning the scoping aspects discussed below.

According to what described in Sec. 2, a guarantee in a WS-Agreement document can be enforced under an optional condition. Such additional constraints are usually defined in terms of accomplishments that a service consumer must meet: for example the latency of a service can depend on the time or on the day in which the request is delivered. In these cases, the transformation function wraps the simulating behavior code-lines obtained from the Service Level Objective part with a conditional statement (see Tab. 4.a).

As mentioned, the scope for a guarantee term describes the list of services to which it applies. In particular, Tab. 4.b points out how to apply the term to a sub-set of the service operations. In this case, for each listed service, the transformation function adds the behavior previously obtained from the Service Level Objective and Qualifying Condition parts only to those operations declared in the scope.

5 Working Example

This section illustrates the application of Puppet to derive a test-bed to verify the QoS characteristics of one service when interacting with other services. The case study considered refers to an on-line booking of flights. Several clients access a Travel Operator

Table 3. Service Level Objective Mapping for Workload Generator

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:Workload>
   <puppet:NRequest>
    20
   </puppet:NRequest>
   <puppet:WinSize>
    60000
   </puppet:WinSize>
  </puppet:Workload>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
public void generateTraffic ()
 throws MalformedURLException,RemoteException{
  Random rnd = new Random();
  int sleepPeriod;
  String endpoint="http://myhost/axis/services/";
  String service="client";
  String method="planJourney";
  int winSize=60000;
  for (int i=0; i<20; i++){
     this.invokeService(endpoint,service,method);
   sleepPeriod = rnd.nextInt(winSize);
   try {
    this.sleep(sleepPeriod);
   } catch (InterruptedException e) {}
   winSize = winSize - sleepPeriod;
  }
}
...
```

Table 4. More Mappings

```
...
<wsag:QualifyingCondition>
 <puppet:PuppetRoot>
  <puppet:Condition
    operator="Not">
   <puppet:Var> interFly </puppet:Var>
  </puppet:Condition>
 </puppet:PuppetRoot>
</wsag:QualifyingCondition>
...
```

```
...
if (!interFly){
  try{
     ...
}
...
```

a) Qualifying Condition

```
...
<wsag:ServiceScope wsag:ServiceName="ABS1">
 <puppet:PuppetRoot>
  <puppet:Operation>checkFlight</puppet:Operation>
 </puppet:PuppetRoot>
</wsag:ServiceScope>
...
```

b) Service Scope

Service (TOS) that in turn accesses different airline booking services (ABSs) to check the availability of a route for the required journeys. In the general case, different ABSs can provide the same functionality according to different QoS specifications.

In the scenario depicted in Fig. 4, clients invoking the TOS service have some restrictions due the maximum number of invocations that can be generated within a time frame. In particular it is supposed that clients cannot generate a workload higher than 20 invocations each 60 seconds. Furthermore, the TOS can interact with two different airline booking services providing different reliability and latency QoS agreements, in particular when invoked with request for intercontinental route.

In the example one the airline booking service (ABS_1) assures a latency of 15 seconds for checking seat availability on a specified flight. The company guarantees service replies reliable up to 99.5% of the requests per day. On the other hand the second airline service (ABS_2) declares to provide the same service in 10 seconds and a realiability of 98% every day. However, it is supposed that the company offering ABS_2 does not directly operate on intercontinental flights. If an international flight operation is required, ABS_2 could need to contact other airline partners providing segments of the selected journey. In these cases, the declared latency agreement reduces to 20 seconds.

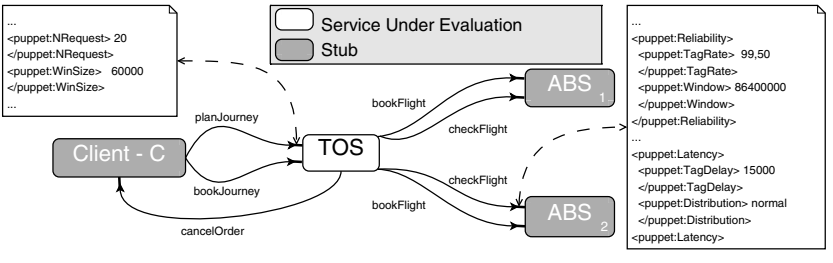

Fig. 4. Example Scenario

Table 5. ABS_2:*checkFlight* Generated Code

```
...
/** RELIABILITY EMULATOR CODE
 */
if (this.possibleFailureInWindow()){
 Random rnd = new Random();
 float val = rnd.nextFloat()*100;
 if ( val>98.00f){
  String fCode="Server.NoService";
  String fString =
    "No target service to invoke!"
  org.apache.axis.AxisFault fault=new
    AxisFault(fCode,fString,"",null);
  this.incNumberOfFailure();
  throw fault;
 }
}
/**QUALIFYING COND. GENERATED CODE*/
if (interFly){
  try{
   Random rnd = new Random();
   float val = rnd.nextFloat();
   int sleepingPeriod = Math.round(val*20000);
   Thread.sleep(sleepingPeriod);
  }
  catch (InterruptedException e) {}
}
/** QUALIFYING CONDITION GENERATED CODE */
if (!interFly){
/** LANTENCY EMULATOR CODE */
 try{
  Random rnd = new Random();
  float val = rnd.nextFloat();
  int sleepingPeriod = Math.round(val*10000);
  Thread.sleep(sleepingPeriod);
 }
 catch (InterruptedException e) {}
}
...
```

Starting from this description Puppet is able to generate a set of stubs for the services interacting with the service under test. In particular Table 5 shows the derived source code for the method checkFlight provided by ABS_2 starting from the corresponding QoS values specified in Fig. 4 formatted according to the WS-Agreement specification as shown in Section 4.

Table 3 instead shows the code generated for the client stub. The generated client defines a method, to be used by the tester, that will raise the specified number of invocations within the specified time frame.

In the idea of Puppet the tester should combine all the generated services in different configurations, reproducing real run-time scenarios. Then launching the scenario it is possible to check if the specification of QoS defined for the Service under evaluation are respected and under which conditions. For instance the developer can try to understand which is the maximum number of clients that can be introduced in the scenario while still being able to abide by the specified QoS.

6 Related Work

Some years ago the authors of [9] recognized that the combination of testing and QoS evaluation, performance in particular, was a seldom explored path in software engineering research. Today the situation has changed and some interesting work starts to appear on this topic. The application of testing for QoS evaluation requires to solve two main problems. The first one is the derivation and simulation of an environment faithfully reproducing the final execution conditions. The second problem is the derivation of a testing suite representative of the real usage scenario, with particular reference to the specific property to be assessed.

This work is mainly related to the first point. Nevertheless the application of empirical approaches to QoS evaluation asks to solve both problems cited above. For works on how to automatically derive test cases, to be used for the successive evaluation of a system in a simulated environment, we refer to the literature, e.g., [9,16,7].

With reference to the generation of test-beds for the validation of WS QoS, we did not find many works. With reference to the area of Component-based software the work presented in [7] shows some similarities with what we propose here. Nevertheless the two approaches are quite different in their motivations and hypotheses. In [7] starting from the hypothesis that the middleware strongly influences the performance behavior of a deployed CBS, the authors generate an environment for the evaluation of the architecture of the system under development. The generated environment then is not intended to be used for the evaluation of a single real component implementation. Instead our target here is to develop a test-bed for the evaluation of a real implementation of a service. Moreover in [7] the stubs are directly derived starting from the architectural definition, no descriptions of QoS are considered (that work was mainly aimed at early performance evaluation). In our work instead, thanks to the availability of the QoS specification (such as the WS-A document), we can generate stubs behaving in accordance to what is defined in the corresponding WS-A document.

A work that has much in common with what we propose here is [11], in which a performance test-bed generator, in the domain of SOA, is presented. The approach proposed is structured in several steps. First, the service under development is described as a composition of services. Then, from this compositional model a collection of service stubs is generated. At this point for each service a description of the load that will be generated by possible clients is defined. From such descriptions, clients simulating the defined load are automatically developed. Finally, clients are executed in order to stress the service composition. The main differences with our proposal are twofold: on the one hand, the system in [11] makes no use of any kind of contract or agreement specification, differently from the approach we propose which is heavily based on the SLA. On the other hand, in [11] the service under development reacts to external stimuli generated by some clients according to the given load model. In our approach the service under development is considered plugged in a well specified choreography. Since none of the choreography members implementation is supposed to be available, our approach automatically builds an environment according to the specification of the coordination scenario.

Finally a main stream of SOA literature is devoted today to runtime evaluation of QoS by means of monitoring approaches. We do not tackle such related work here for size limitations.

7 Conclusions and Future Work

The paper proposed the Puppet approach for the automatic generation of test-beds to empirically pre-validate the QoS of a composite WS before it is deployed. To be applicable the approach requires the availability of the specification of the composition in which the service under evaluation will be inserted. We also assume that the composition is augmented with QoS properties, for instance expressed as a WS-A specification. QoS annotations of WS are not state-of-practice nowadays, nevertheless the relevance of this topic is raising fast in this domain. This is due to the fact that the SOA paradigm aims at removing the barrier among different organisations that can then directly cooperate. In a such scenario the reduced control over the required services certainly asks

for the introduction of agreements concerning the quality of non functional properties in addition to the functional behaviour.

Puppet requires first to define a precise mapping of the terms used for specifying QoS properties to simple parameterized Java code. This step gives a sort of semantic to each type of terms and is done once and for all. In some sense, it implicitly defines the simplest instance of a service providing the specified QoS. More complex definitions of QoS properties are obtained by composing terms of the QoS specification language. Thus, the translation function composes simple transformations according to the rules for the composition. In such a manner Puppet is able to generate service stubs according to composite QoS properties. These stubs can be used to validate possible real implementations of an under development service participating to the same coordination scenario.

The approach we are working on seems promising, nevertheless some issues remain open. Particularly interesting seems the generation of stubs that permit to return meaningful values without introducing complex code that could undermine the realization of stubs behaving in accordance to a specified QoS property.

The approach has been shown to be applicable and a tool is currently under finalization as an Eclipse plug-in. As described the tool will provide stubs mimicking real services, according to the corresponding QoS definition. To carry on reliable experiments the developer will have to distribute the stubs among various machines trying to reproduce as much as possible the final deployment environment. Another important factor strongly influencing the evaluation will be the definition of workload actually representing the final deployment condition. We are working on adding support for these steps that currently rely on human intervention to solve some technical issues. Nevertheless it is important to stress that the reproduction of a representative environment will never be achieved in a completely automated way.

Acknowledgements

The authors wish to thank Giovanni Possemato for his important contribution to the implementation of the tools enabling the proposed approach.

G. De Angelis PhD grant is sponsored by Ericsson Lab Italy in the framework of the PISATEL initiative (http://www1.isti.cnr.it/ERI/)

This work is partially supported by the PLASTIC Project (EU FP6 Strep No. 26955).

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services–Concepts, Architectures and Applications. Springer–Verlag, Heidelberg (2004)
2. Apache Software Foundation. Axis User's Guide. `http://ws.apache.org/axis/java/user-guide.html`.
3. Baresi, L., Ghezzi, C., Guinea, S.: Smart Monitors for Composed Services. In: ICSOC 2004. Proc. 2nd Int. Conf. on Service Oriented Computing, pp. 193–202. ACM Press, New York (2004)

4. Bertolino, A., Bonivento, A., De Angelis, G., Sangiovanni Vincentelli, A.: Modeling and Early Performance Estimation for Network Processor Applications. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, Springer, Heidelberg (2006)
5. Bertolino, A., Frantzen, L., Polini, A., Tretmans, J.: Audition of Web Services for Testing Conformance to Open Specified Protocols. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, Springer, Heidelberg (2006)
6. Bertolino, A., Mirandola, R.: Software Performance Engineering of Component–Based Systems. In: WOSP 2004, pp. 238–242. ACM Press, New York (2004)
7. Denaro, G., Polini, A., Emmerich, W.: Early Performance Testing of Distributed Software Applications. In: WOSP 2004, pp. 94–103. ACM Press, New York (2004)
8. Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G.: Realistic Load Testing of Web Applications. In: Proc. Conf. on Software Maintenance and Reengineering, pp. 57–70. IEEE Computer Society Press, Los Alamitos (2006)
9. Weyuker, E., Vokolos, F.: Experience with performance testing of software systems: Issues, and approach, and case study. IEEE Transaction on Software Engneering 26(12), 1147–1156 (2000)
10. Global Grid Forum: Web Services Agreement Specification (WS–Agreement), version 2005/09 (edn.) (September 2005)
11. Grundy, J., Hosking, J., Li, L., Liu, N.: Performance Engineering of Service Compositions. In: SOSE 2006. Proc. Int. Workshop on Service–Oriented Software Engineering, pp. 26–32. ACM Press, New York (2006)
12. Heckel, R., Lohmann, M.: Towards Contract–based Testing of Web Services. Electronic Notes in Theoretical Computer Science 116, 145–156 (2005)
13. Hrischuk, C.E., Rolia, J.A., Woodside, C.M.: Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In: MASCOTS 1995. Proc. 3rd Int. Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, pp. 399–409. IEEE Computer Society Press, Los Alamitos (1995)
14. IBM. WSLA: Web Service Level Agreements, version: 1.0 revision: wsla-, 2003/01/28 edn. (2003)
15. Liu, Y., Gorton, I.: Accuracy of Performance Prediction for EJB Applications: A Statistical Analysis. In: Gschwind, T., Mascolo, C. (eds.) SEM 2004. LNCS, vol. 3437, pp. 185–198. Springer, Heidelberg (2005)
16. Liu, Y., Gorton, I., Liu, A., Jiang, N., Chen, S.: Designing a test suite for empirically-based middleware performance prediction. In: CRPIT '02: Proc. 4th Int. Conf. on Tools Pacific, pp. 123–130. ACS (2002)
17. Ludwig, H.: WS-Agreement Concepts and Use – Agreement-Based Service-Oriented Architectures. Technical report, IBM (May 2006)
18. OASIS. Web Services Business Process Execution Language (WSBPEL) 2.0 (December 2005) `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.
19. Skene, J., Lamanna, D.D., Emmerich, W.: Precise Service Level Agreements. In: Proc. 26th Int. Conf. on Software Engineering (ICSE 2004), pp. 179–188 (2004)
20. Smith, C.U., Williams, L.: Performance Solutions: A practical Guide To Creating Responsive, Scalable Software. Addison–Wesley, London, UK (2001)
21. Tkachuk, O., Rajan, S.P.: Application of Automated Environment Generation to Commercial Software. In: ISSTA 2006. Proc. ACM Int. Symp. on Sw Testing and Analysis, pp. 203–214. ACM Press, New York (2006)
22. W3C. Web Services Choreography Description Language (WS–CDL) 1.0 (November 2005) `http://www.w3.org/TR/ws-cdl-10/`