Automatic Generation of Test-beds for Pre-Deployment QoS Evaluation of Web Services*

Antonia Bertolino, Guglielmo De Angelis, Andrea Polini
Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
via Moruzzi 1 – 56124 Pisa, Italy
{antonia.bertolino, guglielmo.deangelis, andrea.polini}@isti.cnr.it

ABSTRACT

In last years both industry and academia have shown a great interest on issues concerning consistent cooperation for business-critical services. Service Level Agreement (SLA) specifications as well as techniques for their treatment are nowadays irremissible assets.The availability of many computer processable information make the Web Services framework a valid candidate to experiment with new ideas. This paper proposes Puppet (Pick UP Performance Evaluation Test-bed), an approach for the automatic generation of test-beds to empirically evaluate different 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-Agreements) that the *roles* will abide. The approach is supported by a proof-of-concept tool to validate the feasibility of the idea.

1. INTRODUCTION

The attractive promise of the Service Oriented Architecture (SOA) paradigm is to make real the integration between applications belonging to different enterprises. Web Services related technologies are today the most concrete example enabling such paradigm.

The evolution of Web Services technology has undergone different stages. Initially, the focus was in providing mechanisms for the interaction of service implementations deployed on different machines. Key concepts were service descriptions, service directory definition for storing and retrieving such descriptions, and dynamic services discovering and binding. Then, the need arose for a further level of abstraction to be introduced in the framework, in order to describe possible complex interaction scenarios. In this case, it is assumed that services can either belong to the same organization or to different ones. At last, the openness of the environment characterizing the SOA paradigm naturally led to the pursuit of mechanisms for defining Quality of Service (QoS) level agreement specifications. This is today a quite active research topic (Sec. 2 provides more details on such technologies). In line with the widely accepted idea nowadays that an effective software design process cannot only focus on functional aspects, and ignore QoS-related properties. For Service Oriented systems, as well as for many other kind of complex enterprise applications [7] [29], communication networks and embedded systems [5], it is certainly no longer possible to propose solutions without adequate consideration of their non functional aspects [22].

In recent years, great efforts have been spent in deriving methodologies for the elicitation of non functional requirements, in defining expressive annotation methods, and in the development of methodologies for early performance analysis of software systems. Such studies can be grouped in two broad and orthogonal classes of techniques that combine software design, software development and performance engineering, and are generally referred as *predictive* techniques or *empirical* techniques [16]. The basic ideas of the predictive techniques is that performance cannot be retrofitted: it must be designed into software since the beginning [29]. Thus performance prediction guides the design, providing hints on whether the proposed software solution is likely to meet the desired performance goals.

On the contrary, empirical techniques are applied to running software. Such evaluation can be conducted on the final implementation or on a prototype. The objective is carrying on some tests and comparing the observed QoS characteristics with the expected ones. In case the system shows diverging QoS properties, the software needs to be modified.

Predictive approaches play a very important role during the design and the development of a generic software system. In fact, their use can have a drastic impact and produce great benefits on the quality of the final product. Nevertheless, modern applications are deployed over complex platform (i.e. middleware) introducing external factors not always easy to model. In such contexts, empirical approaches could be a more reliable approach providing more realistic estimates. However, the testing of non-functional properties needs to address issues quite different from those concerned when doing functional validation. In the general case, such approaches require the development of expensive and time consuming prototypes [19], on which representative benchmarks of the system in operation can be run.

Fortunately, much progress has been done in this direction. Specifically, computer-processable specifications can be used in order to describe in detail both the software under evaluation and the expected environment. Today mechanisms acting as *code-factories* are often available. Such tools can automatically generate a run-

*Published as Technical Report at ISTI/CNR

ning prototype from a given specification. In our view when these technologies can be assumed to be applied there is a large room for the development of empirical approaches for the evaluation of a system within the context where it will be used.

In particular, in the domain of Web Services, and related technologies [2, 32, 24, 13, 25, 33, 31, 17], Service Level Agreements (SLAs) are used to explicitly define the services and the customer relations in term of the evaluation criteria, the promised QoS and the penalties assigned if the agreed SLA is not met. Therefore, this domain certainly yields very promising opportunities for the application of empirical approaches for QoS evaluation.

According to this intuition, in this paper we propose an approach, called Puppet, for the automatic derivation of test-beds permitting to evaluate different QoS characteristics for a service under development and before its final deployment. In particular we are interested in assessing that a specific service implementation can afford the required level of QoS (e.g., latency and availability) defined in a corresponding QoS specification for a composition of services (choreography/orchestration) in which the Service Under Evaluation will play one of the role.

The problem of software performance testing has already been acknowledged as an important and tough one in the past [12]. Nevertheless, the pervasiveness of software-intensive systems relying on Web Services, and the great emphasis put today on QoS by the competition between different providers make these topics at the top of the developer's concerns. Moreover, with the advances in the SLA specification technology, means to evaluate the QoS parameters that a service can publish before it is deployed become an indispensable tool for a service developer. Certainly, it becomes vital to be able to agree on levels of performance which are realistic and feasible.

Our approach provides a first step in this direction , because it automatically generates a test-bed which mimics the intended services choreography or orchestration, in which the empirical evaluation can be carried on. Such evaluation can provide crucial information to the developer of the service in order to improve it if some QoS lacks are identified. In the long term the evaluation could be associated to the registration of the service within a directory service. In this perspective the test-bed will be automatically generated and the service tested before its registration. If the required QoS level are not respected, the directory service could refuse the registration, as already foreseen for the case of behavioral evaluation in [8].

The paper is organized as follows. In the next section we provide the necessary background on the Web Services specification technologies, with particular focus on the proposed SLA languages. Then, in Sec. 3 we illustrate the scenarios in which the Puppet tool could be usefully employed. In Sec. 4, we describe the approach and its logical architecture. An example is briefly presented in Sec. 5 while related work are summed up in Sec. 6. At last, in Sec. 7 we draw conclusions and hint at future work.

2. SUPPORTING STANDARDS OVERVIEW

The WS domain is characterized by a strong boost toward standardization. As stated in the Introduction, WS-related specifications belong to different types. For the purpose of illustration we have identified three different groups, which are going to be described in the following subsection.

2.1 WS-related Types of Specification

A first group of specifications include technologies such as the Web Service Description Language (WSDL) [31], the Simple Object Access Protocol (SOAP) [32] and the Universal Directory and Discovery Services (UDDI) [24]. Taken altogether the above three technologies permit the interaction among services deployed in different machines, the storing and successive retrieving of service description and reference, and the dynamic discovery and binding of services.

Of particular relevance to our approach is the Web Service Description Language (WDSL). WSDL is a XML-based language permitting to describe both services and their provided operations [1]. WSDL refers to a service as a set of endpoints interoperating with others by means of message exchanging. In particular, the description of a service consists of an abstract definition of the operations and the messages that will be referred. Thus, the operations are instantiated as concrete endpoints setting some implementation oriented information such as the specific protocol used (e.g., SOAP, HTTP GET, MIME), the addresses at which the operations are made accessible to the clients and so on. However, the WSDL description does not provide any information on how the service has been implemented: it provides only the syntactical information necessary to correctly invoke the system, no assumption on the expected behavior is provided.

Further levels of specification are necessary to describe complex scenarios among interacting services [1]. A second group of specifications is relative to the composition and interaction of services. The identification of adequate mechanisms for doing so is probably the most relevant and interesting ongoing activity within the WS community. In the last years, mainly two approaches seem to lead the scene within two different, but related, context. The first is aiming at providing an instrument for defining orchestration of services, the second instead is intended to describe choreography of web services [1]. Both interpretations provide a means to describe interacting scenarios. Orchestration approaches foresee the availability of an execution engine that, by executing the orchestration code, will reproduce the specified interactions. Clearly the necessity of introducing an execution engine brings some limitations to applicability of the approach, in particular to the case of services belonging to foreign organizations. Such engine will in fact constitute a centralization point that requires a highly trustable location in general not easy to identify.

On the other hand, the choreography approach foresees the availability of a specification of the interactions to which the different services must conform. However it does not introduce any mechanisms for forcing such interactions. As a result, the confidence on correct executions of the described scenarios can only be based on extensive test and verification activities, performed before the deployment of a service. At the same time choreography approaches foster the insertion of monitoring mechanisms within the WS middleware. Such mechanisms can warn the services participating to the choreography if some exceptional conditions arise.

Currently, the most significant proposals concerning the specification of Web Services orchestrations and/or choreographies are represented by the Business Process Execution Language (WSBPEL) [25] and Web Services Choreography Description Language (WS-CDL) [33].

Finally another important ingredient to the success of the SOA

paradigm is the emergence of languages for defining QoS level agreement specifications among interacting services. Even in this context the situation is rapidly moving and different but non exclusive proposals are emerging such as WS-Agreement (WS-A) [13], Web Service Level Agreement (WSLA) [17] and SLAng (Service Level Agreement Language) [28]. Given the centrality of such type of specification in our work, a more extensive discussion is provided below in Sec.2.2.

It is worth noting that the very basic ingredient of the WS technologies is the adoption of the eXtensible Mark-up Language (XML) as the "lingua franca". All the specification languages cited above are in fact based on a XML syntax. Therefore, the description of service composition using such languages permits to describe software engineering concepts in an easily computer processable format. In this sense a choreography or an orchestration strongly recall concepts emerged within the Software Architecture research domain [27]. In this context, the interactions of different components have been described through the definition of roles and glue. It is certainly not a coincidence that the term *role* is used with a similar meaning in the WS-CDL specification. Similar considerations hold for the QoS specification and the contract-based engineering approach [23].

2.2 Service Level Agreement Languages

Traditionally, agreements have been not machine-readable. In software engineering only basic notion of agreements have been experimented by means of Interface Description Languages [21] [22]. In recent years both industry and academia have shown a great interest on this topic. Concerning the Web Services technologies, Service Level Agreements (SLAs) represent one of the most interesting and active issues. SLAs aim at ensuring a consistent cooperation for business-critical services.

To make the paper self-contained, we report below the background notions behind the currently prevailing proposals for agreement specification in SOA. In the remainder of the paper, we will focus on the experiments carried on using one of them (WS-Agreement).

WS-Agreement: WS-Agreement is a specification defined by the Global Grid Forum (GGF) aiming at providing a standard layer to build agreement-driven SOAs [13]. The main assets of the language concern the specification for domain-independent elements of a simple contracting process. Such generic definitions can be extended by domain-specific concepts. In an agreement not every type of content is directly defined. Specifically, the syntax defines statements in order to describe concepts and terms of a generic agreement as top level entities [21]. The use of special construct wraps and integrates the definition related to a specific agreement term.

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 (see Fig. 1).

The Context element is used to describe the involved parties and other content of an agreement not representing obligations of parties, such as 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

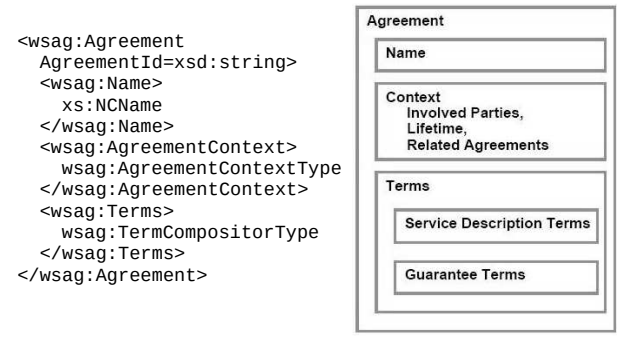

Figure 1: WS–Agreement Structure

compositor elements (e.g., AND/OR/XOR operators) can be used to combine terms enabling the specification of alternative branches and nesting within the terms of agreement.

The obligations of parties 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 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 latter part of the terms definition defines measurable guarantee 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.

WSLA: The Web Service Level Agreement (WSLA) is an XML specification of performance constraints associated with the provision of a Web Service [17]. In particular, the language allows to specify agreements between a service provider and a customer, by defining the obligations of the parties involved.

A WSLA document comprises three main parts: the Parties, the Service Definitions and the Obligations (see Tab. 1). The Parties section defines the actors involved in the agreement. Specifically, this section may contain both the service providers and consumers as well as third-part entities involved in the service monitoring.

The second part of a WSLA document encompasses references to the service operations, the definitions of the observable services' parameters and the definition of metrics. The parameters can be used in assertions in order to guarantee service agreements. The metrics specify how to measure or compute the parameter values. A metric description also includes which party is in charge of measuring. The WSLA language supports constructions for metrics

```
<xsd:complexType name="WSLAType">
  <xsd:sequence>
    <xsd:element name="Parties"
        type="wsla:PartiesType"/>
    <xsd:element name="ServiceDefinition"
        type="wsla:ServiceDefinitionType"
        maxOccurs="unbounded"/>
    <xsd:element name="Obligations"
        type="wsla:ObligationsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="SLA" type="wsla:WSLAType"/>
```

Table 1: WSLA Types Definition

composition. In this manner metrics at different level of abstraction can be expressed in an agreement (e.g., resource metrics, business metrics).

The Obligations part represents the core of the WSLA specification. Here, the arrangement between parties regarding the course of the services is specified: both the constraint and the guarantee terms. WSLA defines a service level objective as a commitment to maintain a particular state of the service in a given period. The state of a service is intended as a value assignment to the service parameters. An action guarantee in an obligation expresses a commitment to perform a particular activity if a given precondition is met. In both cases the service provider as well as the service customer could play as obliged party. In particular, the execution of an action guarantee can also be delegated to a third-party.

SLAng: The last language for service level agreement presented in this section is SLAng [18]. SLAng defines a set of SLAs corresponding to the different kinds of interaction with a service. In particular, the main classification splits the agreements in *vertical* and *horizontal* ones. The former subset refers to a service providing infrastructure support for a client, while the latter to the case in which the client subcontracts part of its functionality to a service of the same type [28]. Examples of vertical SLA are the ones between service provider and host or between a host and storage service provider. On the other hand, a horizontal agreement is contracted between a service and an Application Service Provision.

The SLAng syntax was formerly defined using XML Schema [18]. However, in [28] the authors propose a UML-based specification of the semantics of the language. Since the OMG had meanwhile specified the UML QoS Profile [26] to represent services and SLAs, the authors have reused such specification by defining a QoS catalogue for SLAng.

In our approach we carried on experiments referring to the WS-Agreement, so in the following of this paper we only show examples using such kind of descriptions. We chose WS-Agreement since it can be considered as an evolution of WSLA. In particular, IBM that is the owner of the WSLA project is now heavily involved into the *Grid Resource Allocation Agreement Protocol* working group for the WS-Agreement specification [21] [13]. Concerning SLAng, it represents a valid alternative to the Global Grid Forum's proposal especially because of its UML-based specification. Furthermore, it incorporates in a more natural way relations among the entities involved in the SOA paradigm. However, currently its use is still mainly limited within academia.

3. A WS DEVELOPMENT SCENARIO

In this section we illustrate the application of our proposed approach, called Puppet (Pick-UP Performance Evaluation Test-bed) to automatize the derivation of test-beds. We also discuss the assumptions on the development process that are at the base of the approach. Details on the different steps and how they have been implemented into a prototype tool will then be provided in Sec. 4.

As explained in Sec.2, initially Web Services technologies were intended to merely provide support for the binding and communications between services, but soon the need of mechanisms for describing the integration of services emerged. The basic assumption of our approach is that such a description indeed exists. In our view this is not an unrealistic assumption, as the definition of integrated services will be one of the most relevant factors to facilitate the take-off of the Service Oriented paradigm based on WS[1].

Even though the flexible assembly at run-time of services a priori unknown to each other and without a predefined plan might be an interesting possibility for some applications, our view is that the development of services offers major guarantees, and will be fostered, by the existence of predefined choreographies or orchestrations. Such specifications can be released by relevant organizations within a particular domain, or even standard bodies, in the case of a choreography, or more simply by a company in the case of an orchestration. So what we imagine is, for instance, that for the booking of a flight and for all the related business process, an alliance of airline companies will define a set of choreographies. Such specifications have then to be used by the developers implementing the different services involved, if they want to claim compliance, and in any case for ensuring interoperability.

The first step of our process, referred to as "WS Composition Definition" in Fig. 2, assumes the availability of a specification describing the integration of different services, in terms of WS-CDL or WSBPEL, and of a set of WSDL descriptions defining the interfaces of the services involved in the interaction. Clearly in the case of WSBPEL the organization will be actually the owner of the environment in which the services execute.

The second step in the process is the annotation of the composition with QoS attributes for each service involved in the integration. In Fig. 2 this step is referred to as "QoS instrumentation". We can distinguish between a choreography or an orchestration. In the former case, the organization that released the composition specification is in charge of augmenting the specification with QoS attributes, for instance adding information concerning costs for each service in a prefigured composition. Developers of services will take such a specification as a reference for their implementation, expecting the same for required services. Therefore the binding, static or dynamic, to services foreseen in the choreography, will be done only considering the available services that can provide a QoS level respecting the agreement specification. As consequence each developer of a service is interested in evaluating that its service actually can provide the required service at the given cost. If this will not be true we can expect that no other service, acting one of the role in the given choreography, will agree on binding to such service, with clear consequences on the relevance of the service that will not be used.

In the case of an orchestration, the company defining it will use it to retrieve external services that fulfill the required non functional

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

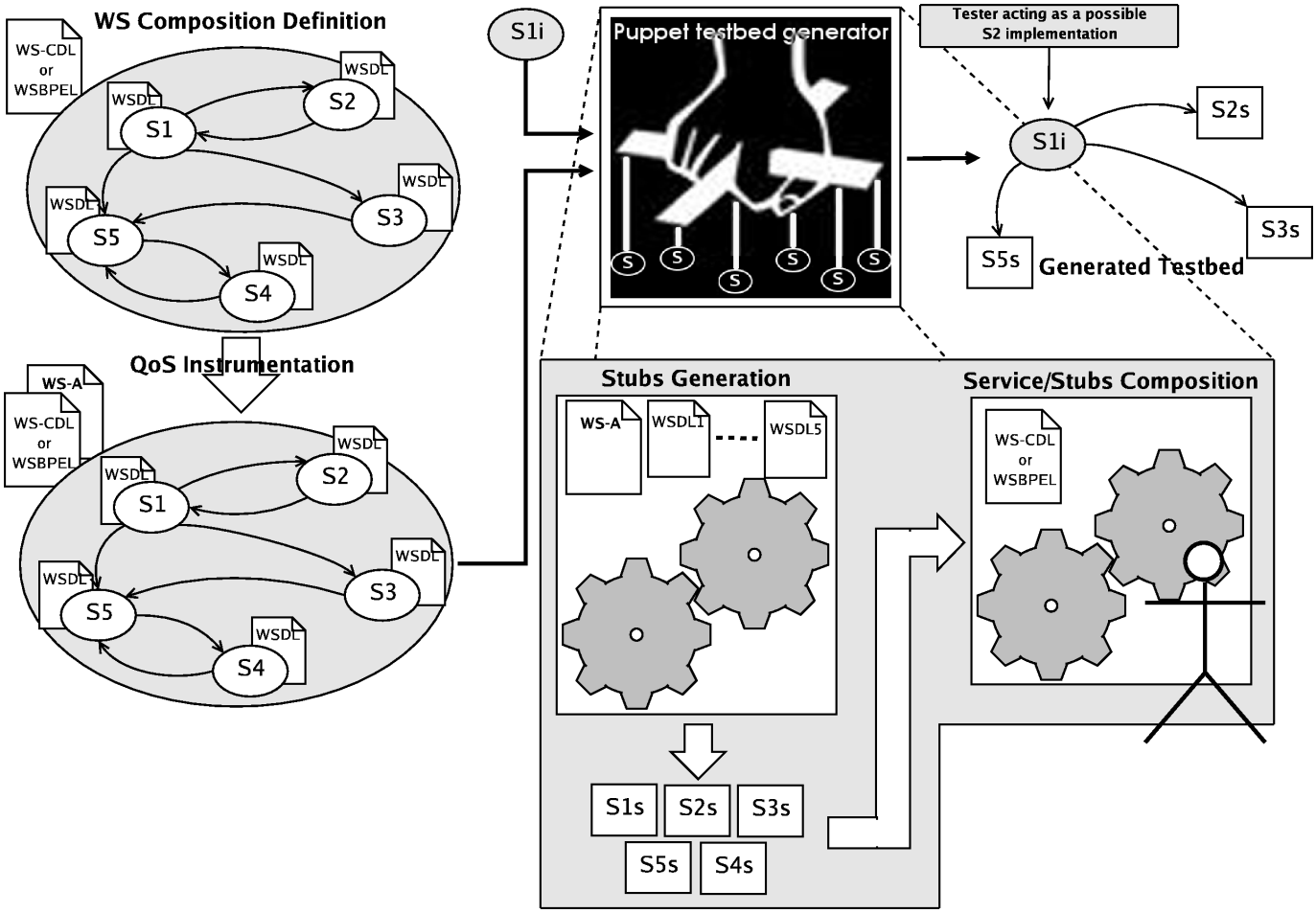

Figure 2: The Puppet approach and supporting tool

properties. In this case the developer of a service willing to sell a service implementation has to assess that, when inserted in the defined orchestration, it is actually able to process a request according to the requirements specified by the company.

Summarizing, we assume 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. At this point, goal of the tool we developed is to automatically generate a test-bed to validate the implementation of a service before its deployment in the target environment.

As illustrated in Fig. 2, the generation of the test-bed proceeds through two different phases. The first one is the generation of the stubs simulating the non functional behavior of the services in the composition; the second one, instead, foresees the composition of the implementation of a service, called "S1i" in Fig. 2, with the services with which it will interact. Both phases will be technically described in detail in Sec. 4 and will be also shortly described here to give a complete overview of the approach.

The generation of the stubs consists in turn of two successive substeps. In the first one a skeleton of the stubs is generated starting from the WSDL description. The thus generated skeletons contain no behavior. Hence, in the second substep the implementation is "filled" with some behavior that will fulfill the required non functional properties for the service corresponding to the stub. This step is carried on retrieving the information from the WS-A and applying automatic code transformation according to rules that we have defined and that are described in Sec.4. At the end of the first phase, a set of stubs providing the services specified in the composition according to the desired properties are available.

The second phase implemented in Puppet is the setting of the testbed. 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 service and according to the composition specified in the choreography or in the orchestration. Currently this phase requires the assistance of a human agent, as illustrated by the presence of a stick man in Fig. 2; nevertheless we are now working on implementing a complete automatic process on the base of the forthcoming final WS-CDL specification.

The final result presented by the Puppet tool is an environment for the evaluation of "S1i". The evaluation, to be carried on, will then require the availability of a tester. In Fig. 2 also such a tool is reported. Nevertheless this component is not within the scope of our work and we refer to the literature on the argument [10] for possible approaches. Such a tool will have to verify that the properties specified in the QoS document, such as a WS-A document, are fulfilled,

in addition to traditional functional testing.

4. DESCRIPTION OF THE APPROACH

The main idea behind the Puppet is that the set of technologies introduced within the Web Service 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 *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 adding to it the lines of code resulting from the transformation of the service agreements specification. The approach can handle those QoS constraints that can be simulated by means of a parameterizable portion of code. The mapping between XML definitions and Java code is defined once in a parametric format and instantiated each time in the occurrences of the pattern appears. In particular, the examples reported in this paper show the transformations we have defined and that are encapsulated in Puppet.

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:TagDelay>
    1000
  </puppet:TagDelay>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
try{
 Thread.sleep(1000);
}
catch
 (InteruptedExcepiton e)
  {}
...
```

Table 2: Service Level Objective Mappaing for Latency

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 service latency are defined as a Service Level Objective according to a prescribed syntax. The example in Tab. 2 reports a latency declaration of $1000mSec$ in XML code and the correspondent Java code that Puppet will automatically generate.

Even though in the examples we refer to constant delays, in general it is possible to handle and generate transformation rules for more complex constraints. Indeed, by declaring the parameters that characterize a distribution in a Service Level Objective, it is possible to implement a transformation function that collects such data and instantiates the delays according to the desired distribution.

On the other hand, constraints on services availability 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 the generated stub service will raise remote exceptions according to the specified rate. Specifically, the XML code in Tab. 3 expresses that a service should be available for the 98% of the invocations. Otherwise, an excepiton would be thrown. Assuming that the Apache-Tomcat/Axis [3] [2] platform is used, the correspondent generated Java code simulates a service invocation fault.

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:TagAvail>
   98.00
  </puppet:TagAvail>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
Random rnd = new Random();
float val = rnd.nextFloat()*100;
if ( val>98.00f){
  String faultCode = "Server.NoService";
  String faultString = "No target service to invoke!"
  org.apache.axis.AxisFault
    fault = new AxisFault(faultCode,faultString,"",null);
  throw fault;
}
...
```

Table 3: Service Level Objective Mappaing for Availability

According to what described in Sec. 2.2, 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). Note that in the current implementation the external factors in the condition are explicitly denoted by means of a variable name.

```
...
<wsag:QualifyingCondition>
 <puppet:PuppetRoot>
  <puppet:Condition
    operator="Equal">
   <puppet:Variable>
    x
   </puppet:Variable>
   <puppet:Value>
    10
   </puppet:Value>
  </puppet:Condition>
  <puppet:Condition
    operator="Lower">
   <puppet:Variable>
    y
   </puppet:Variable>
   <puppet:Value>
    100
   </puppet:Value>
  </puppet:Condition>
 </puppet:PuppetRoot>
</wsag:QualifyingCondition>
...
```

```
...
if ((x==10)&&(y<100)){
  try{
      ...
}
...
```

Table 4: Qualifying Condition Mapping

As mentioned, the scope for a guarantee term describes the list of services to which it applies. In particular, Tab. 5 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.

A WS-Agreement consists of zero or more guarantee terms grouped using the logical grouping compositors. We consider such group as a boolean expression where each guarantee term represents a variable. Hence, a test bed for a selected testing scenario can be ob-

tained providing a variable assignment that satisfies the formula. Correspondingly, for the derived stub Puppet will generate a skeleton that contains lines of code only for those terms corresponding to variables with assigned value equal to true.

```
...
<wsag:ServiceScope wsag:ServiceName="AddressBook">
 <puppet:PuppetRoot>
  <puppet:Operation>addEntry</puppet:Operation>
 </puppet:PuppetRoot>
</wsag:ServiceScope>
...
```

Table 5: Service Scope Mapping

The UML Component Diagram [11] in Fig. 3 outlines the architecture we propose in order to automatically generate service stubs whose behaviors are derived from the terms defined in a WS-Agreement document. In the picture we directly refer Apache-Tomcat/Axis 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. In such case the only requirement is to identify the corresponding tool 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:

1. service skeleton structure definition
2. QoS behavior generation
3. service stub deployment

The first step in the process is directly performed exploiting the Apache-Axis *WSDL2Java* 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, that are based on a XML syntax and can be identified by the extension WSDD (Web Service Deployment Descriptor) [2]. The deployment specification represents the contact 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, into the skeletons no behaviors are coded. Only the operation they exports are derived. According to the above transformation rules from WS-Agreement to Java and the relations in the deployment WSDD file, the *wsCodeBuilder* then generates the simulation code and insert 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 and are then used as input for the *Axis Deployment Engine*.

For the sake of completeness, is important to remark that the generation of meaningful stubs would require to consider also the returning value. Such values could be part of a parametric QoS specification or influence the behavior of the tested service. The current implementation of Puppet does not treat this problem and 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 contract for a service [15] [6] [4]. To correctly manage such situation Puppet will have to integrate mechanisms that enable the instrumentation of stub services with code that can return correct value with respect to the associated functional contract [30]. Obviously the selection of the value to return must be lowly intensive in order to not invalidate possible QoS parameter such as expected performance.

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

5. WORKING EXAMPLE

This section illustrates an application of the proposed approach. In particular, the example used in [1] will be considered, in simplified version to make illustration clearer. We populate the example with sample QoS data.

The referred business process is composed by three kinds of entities (see 4): the customers, the suppliers and the warehouses. Specifically, we consider the interactions among a customer (C1), two suppliers (S1,S2) and a *cloud* of warehouses. All the entities are referred to as Web Services.

The interface exported by the suppliers defines two operations: *requestQuote*, and *orderGoods*. The warehouses export shipment management operations (i.e., *checkShipment*, *defineShipment*, etc.), while the customer defines the *cancelOrder* one.

The coordination scenario describing the example defines that a customer can engage a supplier in order to check some product availability. If the selected supplier can provide the required quantity, the customer can proceed with the order. A supplier interacts with both the customer and the warehouses. In particular, it interacts with a customer canceling a previously confirmed order, and with a warehouse to start or check the status of a shipment.

We assume that S1 and S2 provide the same functionality, but the QoS agreements between C and each of them are supposed to be different. In particular, S1 is a higher performance supplier able to accomplish both the search request and the order commitment within 30 seconds. The availability rate defines the load that a supplier will serve. S1 guarantees to attend to the 94% of the requests. On the other hand, *requestQuote* on S2 takes 60 seconds, and the *orderGoods* operation takes 45 seconds. The availability rate of S2 is equal to 96% on the Sundays and to 99% the other days of the week. App. A reports an extract of the WS-Agreement describing example.[2]

In our application example, we suppose that the Web Service im-

[2]Note that in the current version the weekdays are coded as integers, with string “Sunday” corresponding to the value of “7”

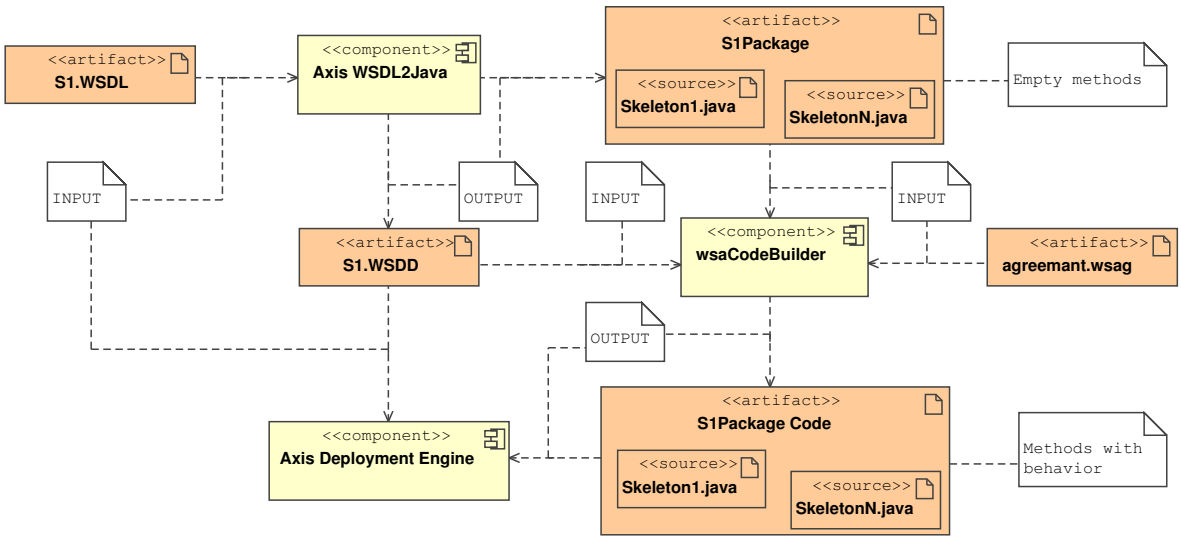

Figure 3: Puppet Test-bed Generator Logical Architecture

plementing the customer is under development. According to the given choreography, a testing environment for the customer part requires to build skeletons for both the suppliers. The warehouse skeletons are not required since there are no interaction with the customers.

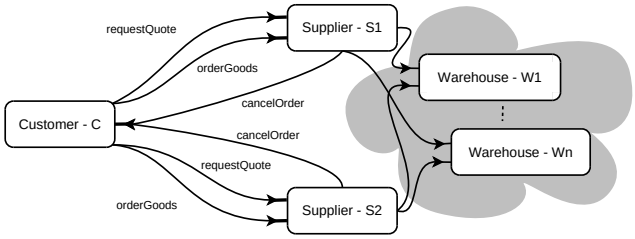

Figure 4: Example Scenario

According to steps described in Sec. 4, the WSDL descriptions for the suppliers S1 and S2 and the WS-Agreement document are used in order to automatically generate both the services skeleton structure and their QoS behaviors. Specifically, in this case only those scenarios that satisfy the guarantee formulas are involved to generate portions of source code for any guarantee term. Tab. 6 reports for instance what is generated for the S2's *requestQuote* operation.

The generated source codes for both the services S1 and S2 can be compiled as normal stand-alone applications. Then, exploiting the information in the WSDD descriptor, S1 and S2 are deployed by means of the Axis Engine. At this point, a simulated environment for the customer part of the coordination scenario is available. Running C or a derived prototype it is possible to test if the current implementation meets the agreements expressed in the WS-Agreement document instrumented within the choreography.

6. RELATED WORK

Some years ago the authors of [12] recognized that the combination of testing and QoS evaluation, performance in particular, was a seldom explored path. Today the situation is changing and some

```
...
if (currentDay==7){
 Random rnd = new Random();
 float val = rnd.nextFloat()*100;
 if ( val>96.00f){
  String faultCode = "Server.NoService";
  String faultString = "No target service to invoke!"
  org.apache.axis.AxisFault
     fault = new AxisFault(faultCode,faultString,"",null);
  throw fault;
 }
}
if (currentDay!=7){
 Random rnd = new Random();
 float val = rnd.nextFloat()*100;
 if ( val>99.00f){
  String faultCode = "Server.NoService";
  String faultString = "No target service to invoke!"
  org.apache.axis.AxisFault
     fault = new AxisFault(faultCode,faultString,"",null);
  throw fault;
 }
}
try{
 Thread.sleep(60000);
}
catch
 (InteruptedExcepiton e)
  {}

...
```

Table 6: S2:*requestQuote* Generated Code

interesting work has been recently published on the topic. The application of testing to 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 to be solved is the derivation of a testing suite representative of the real usage scenario, with particular reference to the specific property to be assessed.

Our original work is mainly related to the first point, such as the generation of a testing environment. Nevertheless the application of empirical approach 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 referred to the works presented in [12, 20, 9].

With reference to the generation of test-bed for the early evaluation of Web Services QoS indeed we did not find many works. With reference to the area of Component-based software the work presented in [9] shows some similarities with what we propose here. Nevertheless the two approaches are quite different in motivation and hypothesis. In [9] starting from the hypothesis that complex middlewares 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 [9] the stubs are directly derived starting from the architectural definition, no description of QoS are considered (such work was mainly interested in early performance evaluation). Our work instead, thanks to the availability of QoS specification (such as WS-A document), generates stubs behaving in accordance to what is defined in the corresponding WS-A document.

Indeed the work that has much in common with what we propose here is certainly [14]. In that work a performance test-bed generator, in the domain of SOA, is presented. The approach proposed is structured in several steps. At first, the service under development is described as a composition of services. Then, from this compositional models a collection of service stubs are 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 in the last step, 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 [14] makes no use of any kind of contract or agreement specification. Differently, the approach we propose is heavily based on the use SLA. In particular, we refers to WS-Agreement specification [13]. On the other hand, in [14] the service under development reacts to external stimuli generated by the clients according to the given load model. In our approach the service under development is considered plugged in a well specified choreography. Here, it is the service under development that invokes the other *roles*. Since no one of the choreography members implementation is supposed to be available, our approach automatically build an environment according to the specification of the coordination scenario.

7. CONCLUSIONS AND FUTURE WORK

The paper proposes an approach for the automatic generation of test-beds intended to be used to verify QoS properties of web services implementation before their deployment. To be applicable the approach requires the availability of precise specification describing the composition of services in which the developed service should be inserted. We assume that such composition is also augmented with QoS properties, in particular the current implementation of the developed tool assumes the availability of a WS-A specification for the whole composition.

The approach we propose requires to define a precise mapping of terms in the language used for specifying QoS properties and simple parametrized Java code. This step gives a sort of semantic to each kind of terms. Furthermore, in some sense, it implicitly defines the most simple instance of a service providing the specified QoS. Complex definition of QoS properties are obtained composing terms of the QoS specification language. Thus, the translation function will compose simple transformations according to rule for the composition. In such manner Puppet is able to generate service stubs according to different QoS properties. Hence, stubs can be used to validate possible real implementations of an under development service participating to the same coordination scenario. Currently, we have implemented a first proof-of-concept of the tool.

The approach we are working on seems promising, nevertheless we still need to investigate some issues. Particularly interesting seems the generation of stubs that permit to return meaningful values without introducing complex code that could bias the obtainment of stub behaving in accordance to a specified QoS property.

Finally the current implementation of the Puppet tool is not able to automatically handle all the phases described in this paper. In particular the setting of the environment currently requires some human intervention. We are working to make also this step completely automatic.

8. ACKNOWLEDGMENT

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 n.26955).

9. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services–Concepts, Architectures and Applications*. Springer–Verlag, 2004.

[2] Apache Software Foundation. *Axis User's Guide*. http://ws.apache.org/axis/java/user-guide.html.

[3] Apache Software Foundation. Tomcat. http://tomcat.apache.org/.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC 2004)*, pages 193–202. ACM Press, 2004.

[5] A. Bertolino, A. Bonivento, G. De Angelis, and A. Sangiovanni Vincentelli. Modeling and Early Performance Estimation for Network Processor Applications. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006)*. Springer–Verlag, 2006. to–apper.

[6] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in LNCS. Springer-Verlag, 2006.

[7] A. Bertolino and R. Mirandola. Software Performance Engineering of Component–Based Systems. In *Proc. of the 4th International Workshop on Software and Performance (WOSP 2004)*, pages 238–242. ACM Press, 2004.

[8] A. Bertolino and A. Polini. The Audition Framework for Testing Web Services Interoperability. In *Proc.of 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO–SEAA 2005)*, pages 134–142. IEEE Computer Society, 2005.

[9] G. Denaro, A. Polini, and W. Emmerich. Early Performance Testing of Distributed Software Applications. In *Proc. of the 4th International Workshop on Software and Performance (WOSP 2004)*, pages 94–103. ACM Press, 2004.

[10] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic Load Testing ofWeb Applications. In *Proc. of the Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 57–70. IEEE Computer Society, 2006.

[11] H.E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit*. Wiley Publishing Inc., 2004.

[12] E.Weyuker and F. Vokolos. Experience with performance testing of software systems: Issues, and approach, and case study. *IEEE Transaction on Software Engneering*, 26(12):1147–1156, December 2000.

[13] Global Grid Forum. *Web Services Agreement Specification (WS–Agreement)*, version 2005/09 edition, September 2005.

[14] J. Grundy, J. Hosking, L. Li, and N. Liu. Performance Engineering of Service Compositions. In *Proc. of the 2006 International Workshop on Service–Oriented Software Engineering (SOSE 2006)*, pages 26–32. ACM Press, 2006.

[15] R. Heckel and M. Lohmann. Towards Contract–based Testing of Web Services. *Electronic Notes in Theoretical Computer Science*, 116:145–156, 2005.

[16] C.E. Hrischuk, J.A. Rolia, and C.M. Woodside. Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In Patrick W. Dowd and Erol Gelenbe, editors, *Proc. of the 3rd International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS 1995)*, pages 399–409. IEEE Computer Society, 1995.

[17] IBM. *WSLA: Web Service Level Agreements*, version: 1.0 revision: wsla-2003/01/28 edition, January 2003.

[18] D.D. Lamanna, J. Skene, and W. Emmerich. SLAng: A Language for Defining Service Level Agreements. In *Proc. of 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, pages 100–106. IEEE Computer Society, 2003.

[19] Y. Liu and I. Gorton. Accuracy of Performance Prediction for EJB Applications: A Statistical Analysis. In *Proc. of Software Engineering and Middleware (SEM 2004)*, volume LNCS 3437, pages 185–198. Springer, 2004.

[20] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. In *CRPIT '02: Proc. of the Fortieth International Conference on Tools Pacific*, pages 123–130. ACS, 2002.

[21] H. Ludwig. WS-Agreement Concepts and Use – Agreement-Based Service-Oriented Architectures. Technical report, IBM, May 2006.

[22] H. Ludwig, A. Dan, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreents. In *Proc. of Service-Oriented Computing - ICSOC 2004, Second International Conference*, pages 65–74. ACM, 2004.

[23] B. Meyer. *Object–Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[24] OASIS. *Universal Description Discovery & Integration (UDDI) 3.0.2*, October 2004. http://uddi.org/pubs/uddi_v3.htm.

[25] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, December 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[26] OMG. *UML Profile for Modeling QoS and FT Characteristics and Mechanisms Specification*, OMG Available Specification – formal/06-05-02 edition, May 2006.

[27] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[28] J. Skene, D.D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. of 26th International Conference on Software Engineering (ICSE 2004)*, pages 179–188. IEEE Computer Society Press, 2004.

[29] C.U. Smith and L. Williams. *Performance Solutions: A practical Guide To Creating Responsive, Scalable Software*. Addison–Wesley, 2001.

[30] O. Tkachuk and S.P. Rajan. Application of Automated Environment Generation to Commercial Software. In *Proc. of the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 203–214. ACM Press, 2006.

[31] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001. http://www.w3.org/TR/wsdl.

[32] W3C. *Simple Object Access Protocol (SOAP) 1.2*, June 2003. http://www.w3.org/TR/soap12.

[33] W3C. *Web Services Choreography Description Language (WS–CDL) 1.0*, November 2005. http://www.w3.org/TR/ws-cdl-10/.

APPENDIX
A. WS-AGREEMENT CODE

In the following an extract of the WS-Agreement describing the working example in Sec. 5 is reported.

```
<wsag:AgreementOffer ... wsag:AgreementId="exampleWSAG">
 ...
 <wsag:Terms>
  <wsag:All>
   <wsag:ServiceDescriptionTerm ... wsag:ServiceName="C"/>
   <wsag:ServiceDescriptionTerm ... wsag:ServiceName="S1"/>
   <wsag:ServiceDescriptionTerm ... wsag:ServiceName="S2"/>
   ...
   <wsag:All>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S1">
      <puppet:PuppetRoot>
       <puppet:Operation>requestQuote</puppet:Operation>
       <puppet:Operation>orderGoods</puppet:Operation>
```

```
      </puppet:PuppetRoot>
     </wsag:ServiceScope>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagDelay>30000</puppet:TagDelay>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S1"/>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagAvail>94.00</puppet:TagAvail>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
     ...
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S2">
      <puppet:PuppetRoot>
       <puppet:Operation>requestQuote</puppet:Operation>
      </puppet:PuppetRoot>
     </wsag:ServiceScope>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagDelay>60000</puppet:TagDelay>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
     ...
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S2">
      <puppet:PuppetRoot>
       <puppet:Operation>orderGoods</puppet:Operation>
      </puppet:PuppetRoot>
     </wsag:ServiceScope>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagDelay>45000</puppet:TagDelay>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
     ...
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S2"/>
     <wsag:QualifyingCondition>
      <puppet:PuppetRoot>
       <puppet:Condition operator="Equal">
        <puppet:Variable>currentDay</puppet:Variable>
        <puppet:Value>7</puppet:Value>
       </puppet:Condition>
      </puppet:PuppetRoot>
     </wsag:QualifyingCondition>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagAvail>96.00</puppet:TagAvail>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
     ...
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Obligated="ServiceProvider">
     <wsag:ServiceScope wsag:ServiceName="S2"/>
     <wsag:QualifyingCondition>
      <puppet:PuppetRoot>
       <puppet:Condition operator="Different">
        <puppet:Variable>currentDay</puppet:Variable>
        <puppet:Value>7</puppet:Value>
       </puppet:Condition>
      </puppet:PuppetRoot>
     </wsag:QualifyingCondition>
     <wsag:ServiceLevelObjective>
      <puppet:PuppetRoot>
       <puppet:TagAvail>99.00</puppet:TagAvail>
      </puppet:PuppetRoot>
     </wsag:ServiceLevelObjective>
     ...
    </wsag:GuaranteeTerm>
   </wsag:All>
  </wsag:All>
 </wsag:Terms>
</wsag:AgreementOffer>
```