Model-Based Generation of Testbeds for Web Services

Antonia Bertolino[1], Guglielmo De Angelis[1], Lars Frantzen[1,2], and Andrea Polini[1,3]

[1] Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy
{antonia.bertolino,guglielmo.deangelis}@isti.cnr.it
[2] Institute for Computing and Information Sciences (ICIS)
Radboud University Nijmegen – The Netherlands
lf@cs.ru.nl
[3] Department of Mathematics and Computer Science
University of Camerino – Italy
andrea.polini@unicam.it

Abstract. A Web Service is commonly not an independent software entity, but plays a role in some business process. Hence, it depends on the services provided by external Web Services, to provide its own service. While developing and testing a Web Service, such external services are not always available, or their usage comes along with unwanted side effects like, e.g., utilization fees or database modifications. We present a model-based approach to generate stubs for Web Services which respect both an extra-functional contract expressed via a Service Level Agreement (SLA), and a functional contract modeled via a state machine. These stubs allow a developer to set up a testbed over the target platform, in which the extra-functional and functional behavior of a Web Service under development can be tested before its publication.

1 Introduction

The emergence of the Service Oriented Architecture (SOA) paradigm is changing the way in which software applications are developed [17]. Two trends, seemingly in contradiction with each other, are witnessed. On the developer's side, we face a drop in control capability: a service-oriented application consists of the dynamic composition of autonomous services. Therefore, a service-oriented application commonly depends on the services provided by external, autonomous services, and its development process is not anymore under the full control of a single stakeholder/organization.

On the consumer's side, instead, users (i.e., the service clients), who can choose among many available services, are becoming more and more exigent concerning the properties expected from a service, and ask for precise specifications of the offered service functionality and performance. Based on those they will make their "purchase" decision. Such specifications establish a *contract* between the service provider and the client.

Thus, a service provider needs to manage and reconcile these contrasting trends: binding client requirements against limited control capability. A key approach to solve this conflict stays in propagating the formalization of contracts among all the services invoked by a composite service. The interdependency with other services is laid on precise specifications of service interfaces, which also establish contracts on the interactions between services.

A contract can deal with functional and extra-functional properties. When dealing with the former, beside the mere signatures of the provided operations, the offered functionality of services is often subject to given conditions, like e.g. *always invoke the authentication operation providing a valid password before invoking the booking operation.* This is especially true for stateful services. Such contracts are prevalently specified via state machines. Extra-functional contracts, i.e., the delivered Quality of Service (QoS) levels, are made explicit in Service Level Agreements (SLAs in the following) [21]. SLAs also provide the basis on which the cost of using the service is quantified, and the penalties, in case the contract is violated, are defined.

When developing a new service, its interaction with the external services it uses must be tested, to validate that it obeys the functional contracts in place, and to evaluate its offered quality level, which is affected by the quality levels of the invoked services. In an ideal case, all the external services are available, and can be arbitrarily accessed at development time for testing purposes. Unfortunately, this ideal case is seldom offered. Commonly, at least some of the external services are either not available at all (for instance simply not implemented, yet), or their usage comes along with unwanted side-effects (for instance utilization fees or database modifications). But what is usually available are the contracts of the external service interfaces. The availability of models specifying extra-functional behavior for the interacting services could also suggest the application of analytical techniques [16] to derive the needed extra-functional properties. As discussed in [8], when interaction happens through complex middlewares, such an option is not always feasible, since the modeling of such infrastructures is particularly difficult and error prone.

At the core of the approach presented in this paper is a model-based stub generator, called Puppet [23]. We assume that for each external service not available or suited for testing, both a functional contract in terms of a state machine, and an extra-functional contract in terms of an SLA, are present. Based on these two contracts, Puppet automatically generates a stub for the external service, which respects the contracts. For the functionality this means that the stub behaves conforming to the functional contract (for instance, it never violates any guard of the corresponding state machine). Furthermore, when someone is using the stub, it is able to detect a violation of its functional contract by that user. Respecting the extra-functional contract means that the stub meets the quality levels specified in the corresponding SLA.

Using this generator the developer of a new composite service can replace all external services which are not fully available at development time with the stubs generated by Puppet. The automatically obtained stubs make up a testbed in

which the service under development can be tested within the target platform before it is published. Carrying on the testing also entails the identification of a suitable test suite. This is a challenging issue, but outside the scope of the present paper (though we hint at possible directions to pursue in future work).

In a preliminary work [4] we described how stubs respecting an extra-functional contract can be generated, but in that work the generated stubs did not consider functional aspects (i.e., they provided a correct quality-related behavior, but the messages were not built to be semantically meaningful). This paper's original contribution is the integration of functional contracts as part of the generated testbed. We will motivate (see Sect. 5) that functionality and extra-functionality are not independent from each other. Having functionally correct stubs can reveal extra-functional behavior of the service under development, which may not be observable in a purely extra-functional testbed. Vice versa, having stubs respecting given quality levels may disclose functional behavior which does not show up in a purely functional testbed. Thus, here the whole is more than the sum of its parts; combining both kinds of contracts strictly increases the testing power of the testbed compared to taking both a purely extra-functional and a purely functional testbed. While related proposals exist for functional verification or extra-functional evaluation of services (see Sect. 6), to the best of our knowledge, no such framework supports an integration of both aspects.

The next section provides an overview of PUPPET, and introduces the case study we will use throughout the paper to illustrate the approach. Section 3 explains how the SLAs are modeled and simulated, while Sect. 4 provides the background for functional modeling, and introduces the formal testing relation we exploit to simulate correct functional behavior of the generated stubs. Section 5 illustrates the testing power of the testbed generated by PUPPET. Finally, related work is overviewed in Sect. 6, while conclusions and further interesting features, that PUPPET could offer in future work, are given in Sect. 7.

2 Overview

We demonstrate our approach by referring to an exemplary case study, which is introduced in Sect. 2.1. Section 2.2 presents the logical approach of the PUPPET stub generator.

2.1 Motivating Scenario

We have implemented a simplified version of the scenario presented in [1], in which three services (the customer, the supplier, and the warehouse) cooperate to achieve the task of a trade. The customer service is interested in buying a certain amount of a given product, and queries the supplier service for a quote for the product of interest. Having received the request, the supplier queries one or more warehouse services to check if the requested quantity is in stock. The information provided by the warehouses is then collected by the supplier service, and returned to the customer service. If satisfied with one of the provided quotes, the customer

can then proceed with the order. The case study also handles further interactions, namely supplier authentication and bonus accounting schemes, which will be discussed in Sect. 4 and Sect. 5.

We can realistically assume that the three services are implemented and provided by different stakeholders, and that their interactions are governed by message exchange protocols under agreed levels of QoS, as shown in Fig. 1.

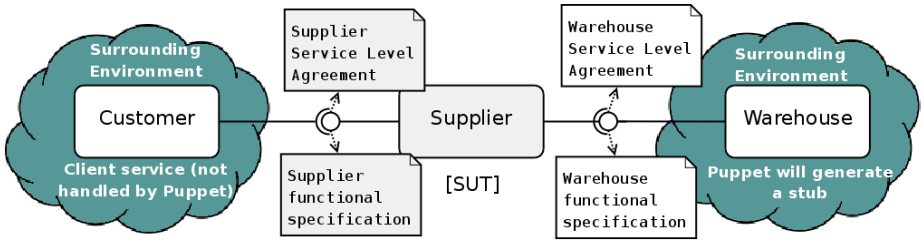


Fig. 1. The Customer-Supplier-Warehouse Case Study

For illustration purposes, we put ourselves in the role of the *developer of a supplier service*, which requires a testbed to a) test the compliance of the supplier service with the functional contracts of the warehouse services, and b) needs to derive reliable values for the QoS of the supplier service by taking into account the QoS of the given warehouse services. The latter is particularly important if the developer him/herself needs to publish a contract for potential customers, which may involve the provision of the services under agreed level of QoS parameters (such as for instance throughput, latency, reliability).

The behavior of the supplier (both functional and extra-functional) depends on the behavior of the warehouse services going to be accessed at run-time. The problem the developer faces here is that he/she does not want to invoke the real warehouse services during development time for testing purposes (e.g., really buying goods).

To address this issue, the PUPPET tool can automatically generate stubs for the warehouses that do not only reproduce the specified functional behavior encoded in corresponding state machines, but also perform according to the QoS parameters defined in the warehouses SLAs. The idea is, that such a stub can then be deployed on a Web Service platform (such as Axis [3]), potentially reproducing different distribution settings, and being accessed during testing by the supplier service for taking experimental measures. For simplicity the considered case study only deals with one warehouse service to be simulated; in general PUPPET can handle an arbitrary number of services, generating one stub for each externally accessed service.

2.2 Logical Approach of PUPPET

PUPPET generates stubs which exhibit meaningful extra-functional and functional behaviour. The generation is structured in three main steps. The first step defines the stub structure by converting the abstract part of a WSDL [6] representation into a collection of Java classes and interfaces. This transformation is performed exploiting the Apache-Axis *WSDL2Java* utility [3]. The second generation step extends the stub with the code simulating the extra-functional behavior. We assume that the desired QoS properties of the service to be stubbed are expressed according to a WS-Agreement specification [13]. Section 3 will show how the WS-Agreement statements are mapped to parametric portions of Java code. The final step of the generation process inserts into the stub parametric code able to emulate the intended functionality. Our assumption is that a functional specification of the service to be stubbed is available (possibly derived from a global view such as a choreography specification) via a state machine. Such a specification defines which are the correct values expected for the service invocation parameters, what is the legal message ordering accepted by the service, and, what are the characteristics of an answer to be provided by the service. In particular we adopt the *Symbolic Transition System* (STS) notation (see Sect. 4). Having all desired stubs generated by PUPPET, the developer is able to mock-up the environment by deploying the generated stubs on any Axis platform.

Note that the generated functional and the extra-functional parts may interoperate at run-time. As described in the following, PUPPET uses pre-defined Java patterns in order to emulate the functional and extra-functional properties. PUPPET combines the generated patterns according to a fixed order. Defining a new pattern or changing an existing one has to be validated with respect to both the possible dependencies with the other patterns, and the order in which PUPPET combines them. For example, as detailed in Sect. 3.2, the definition of the Java pattern dealing with latency constraints takes into account the temporal dependency on other computational tasks.

3 Model-Based Extra-Functionality

This section firstly introduces the language we use to express SLAs (Sect. 3.1), and then shows how the referred extra-functional properties are processed by PUPPET (Sect. 3.2). Generally speaking, SLAs describe the agreements that a service commits to accomplish when processing a request from a client, starting from the moment it receives the request until the moment it replies [22]. QoS properties are defined only as a provider constraint, and do not include any kinds of events that the client may experience, for example network failures or traffic congestion problems. We point at ways to consider network issues in Sect. 6.

3.1 WS-Agreement

WS-Agreement [13] is a language defined by the Global Grid Forum aiming at providing a standard layer to build agreement-driven SOAs. The main in-

gredients 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. The top-level structure of a WS-Agreement is expressed via an XML document comprising the agreement descriptive information, the context it refers to and the definition of the agreement items. It includes the involved parties as well as other aspects 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, organized in two logical parts. The `Service Description Terms` part specifies the involved services. It describes the reference 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. Such guarantees can be fulfilled or violated. A `Guarantee Term` definition consists of the obliged parties (i.e, *Service Consumer* and *Service Provider*), the list of services this guarantee applies to (`Service Scope`), a boolean expression that defines the condition under which the guarantee applies (`Qualifying Condition`), the actual assertions that have to be guaranteed over the service (`Service Level Objective` - SLO), and a set of business-related values (`Business Value List`) of the described agreement (i.e., importance, penalties, preferences). In general, the information contained in the fields of a `Guarantee Term` is expressed by means of domain-specific languages.

3.2 Defining Extra-functional Annotations

The approach implemented in PUPPET associates concepts in the WS-Agreement (i.e., `SLO`, `Qualifying Condition`, `Service Scope`) with an interpretation by means of a given operational semantics. This can be a quite complex and effort-prone task, but given a specific language and an intended interpretation of the concepts, it has to be done only once and for all.

```
...
<wsag:ServiceLevelObjective>
 <puppetSLO:PuppetSLO>
  <puppetSLO:Latency>
   <value>25000</value>
    <puppetSLO:Distribution>
      <Gaussian>10</Gaussian>
    </puppetSLO:Distribution>
  </puppetSLO:Latency>
 </puppetSLO:PuppetSLO>
</wsag:ServiceLevelObjective>
...
```

–A–

```
...
Density D = new Density();
long funcElapsedTime = puppet.ambition.Naturals.asNatural
     (aMbItIoNinvocationTime - System.currentTimeMillis()
     );
long maxSleepingPeriod = 25000 - funcElapsedTime;
Double sleepingPeriod = D.gaussian(maxSleepingPeriod,10);
try {
  Thread.sleep(sleepValue.longValue());
} catch (InterruptedException e) {}
...
```

–B–

Fig. 2. SLO Mapping for Latency

Precisely, PUPPET defines a mapping from the declarative XML descriptions of the supported QoS properties to composable Java code segments. The mapping is specified in a parametric format that is instantiated each time one occurrence of the concept appears. Within the scope of this paper, we deal with two QoS properties: latency and reliability. The remainder of this section introduces their characteristics. Please note that the specifications of such QoS properties conform to the definitions adopted within the PLASTIC Project [9]. Nevertheless, also other definitions can be adopted (e.g. as in [20]) .

Latency is defined as a server-side constraint, and does not concern (just ignores) other kinds of delays that the client may experience, for example due to network failures or traffic congestion problems. Conditions on latency are simulated in PUPPET by introducing *delay* instructions into the operation bodies of the services stubs. For each `Guarantee Term` in a WS-Agreement document, information concerning the maximum service latency is defined as a `Service Level Objective`. As an example, Fig. 2.A reports the XML code for a maximum latency declaration of $25000msec$ normally distributed, and Fig. 2.B shows the corresponding Java code that is automatically generated by PUPPET.

When dealing with latency constraints, PUPPET also has to deal with other computational tasks, like generating a functionally correct return message, taking care of reliability constraints, etc. Since these tasks also consume time, PUPPET has to adapt the generated latency sleeping period. For example, consider that the term in Fig. 2.A comes in combination with some functional computation statements. If at run time these computations take $2sec$, the delay of the service is adjusted to the range of $[0 \div 23000]msec$. In case the calculation of the functionally correct return message takes more than what is allowed by the latency constraint, the stub raises an exception and has failed its purpose. Since SLA latency constraints for services are commonly in the order of seconds, the computational tasks needed to generate the return messages only miss such deadlines in quite rare cases.

Reliability constraints are declared in the `Service Level Objective` of a `Guarantee Term`, stating the maximal admissible number of failures a service can raise in a given time window. Such kinds of QoS attributes can be reproduced introducing code that simulates a service failure. PUPPET realizes a reliability failure via an exception raised by the platform hosting the Web Service stub. An example of the PUPPET transformation for reliability constraints is shown in Fig. 3. Part A shows the XML code specifying a maximum allowed number of three failures over an observation window of 2 minutes; part B gives the corresponding Java translation, assuming that the Apache-Tomcat/Axis [3] platform is used.

As described in Sect. 3.1, a guarantee in a WS-Agreement document could also be stated under an optional condition expressed by means of some `Qualifying Condition` elements. Usually such optional constraints are defined in terms of accomplishments that the service consumer must meet. For instance, the latency of a service may depend on the value of some parameters provided at run-time. In these cases, the PUPPET transformation function wraps the simulating code

```
...
<wsag:ServiceLevelObjective>
 <puppetSLO:PuppetSLO>
  <puppetSLO:Reliability>
   <Reliabilitywindow>
    120000
   </Reliabilitywindow>
   <MaxFailures>
    3
   </MaxFailures>
   <puppetSLO:Distribution>
    <Gaussian>
     10
    </Gaussian>
   </puppetSLO:Distribution>
  </puppetSLO:Reliability>
 </puppetSLO:PuppetSLO>
</wsag:ServiceLevelObjective>
...
```

–A–

```
...
long winSize = 120000;
int maxFault = 3;
long currentTimeStamp = System.currentTimeMillis();
for (int i=0; i<faultBuffer.size();i++){
  if (currentTimeStamp - faultBuffer.get(i) >= winSize){
    faultBuffer.remove(i);
  }
}
if (faultBuffer.size() < maxFault){
  Density d = new Density();
  double dv = d.gaussian(100);
  if (dv > 50) {
    String fCode = "Server.NoService";
    String fString = "PUPPET EXCEPTION : No target
        service to invoke!";
    org.apache.axis.AxisFault fault = new org.apache.
        axis.AxisFault(fCode, fString, "", null);
    aMbItIoNsim.undo();
    faultBuffer.add(currentTimeStamp);
    throw fault;
  }
}
...
```

–B–

Fig. 3. SLO Mapping for Reliability

obtained from the `Service Level Objective` part within a conditional statement. As mentioned, the scope for a guarantee term describes the list of services to which it applies. In these cases, for each listed service, the transformation function adds the behavior obtained from the `Service Level Objective` and `Qualifying Condition` transformations only to those operations declared in the scope.

4 Model-Based Functionality

The functional behavior of a service is modeled using an automata model called *Symbolic Transition System* (STS). STSs are a well studied formalism in modeling and testing of reactive systems [12]. We understand, though, that they could sound unfamiliar and difficult for practitioners. However, STSs can be seen as a formal semantics for a variant of UML 2.0 state machines [18]. We have developed a library called MINERVA [23], which transforms the output generated by MAGICDRAW (http://www.magicdraw.com) – a commercial UML modeling tool – into an STS representation understood by PUPPET. Thus, a developer can use this visual tool to model the functionality of service interfaces in the common formalism of UML 2.0 state machines. We do not describe this mapping here, but present instead directly the STS formalism. We will use a dedicated testing relation called *eco* [11] which is specifically appropriate for the creation of stubs like the ones we are dealing with in our setting. This section introduces STSs and the *eco* relation.

4.1 Symbolic Transition Systems

In our setting, STSs specify the functional aspects of a service interface. Firstly, there are the static constituents like types, messages, parameters, and operations. This information is commonly denoted in the WSDL [6]. Secondly, there are the dynamic constituents like states, and transitions (also called arcs) between the states. STSs can be seen as a dynamic extension of a WSDL. They specify the legal ordering of the message flow at the service interface, together with constraints on the data exchanged via message parameters (called *parts* in the WSDL).

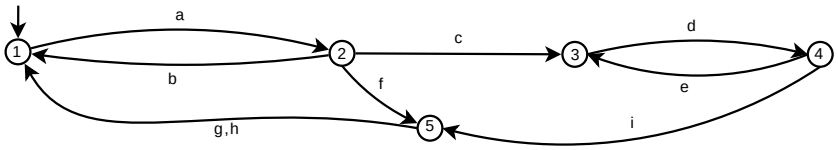
An STS can store information in STS-specific variables. Every STS transition corresponds to either a message sent to the service (input), or a message sent from the service (output). Furthermore, a transition can be guarded by a logical expression. After a transition has fired, the values of the variables can be updated. Due to its extent and generality we do not give here the formal definition of STSs, which can be found in [12]. Instead, we exemplify the concepts in the setting relevant for this paper.

We assume here that data types in the WSDL are specified via XML Schema types, as commonly done. Let us consider a WSDL operation `checkAvail` with an input message `?checkAvail` and an output message `!checkAvail`. The input message has a part `r` of type `QuoteRequest`; the output message has a part `q` of type `Quote`. The `QuoteRequest` type is a complex type sequence with the elements `product` of type `String` and `quantity` of type `Integer`. The `Quote` type is a complex type sequence with the elements `status` of type `Integer`, `product` of type `String`, `quantity` of type `Integer`, `price` of type `double`, and `refNumber` of type `Integer`. This WSDL operation could for instance correspond to a Java method with signature: `Quote checkAvail(QuoteRequest r)`, together with the classes `Quote` and `QuoteRequest`. A message in an STS corresponds to a message in the WSDL. Hence, we model the call of the `checkAvail` operation in the STS by two succeeding transitions. The first one with message `?checkAvail(r:QuoteRequest)` represents the operation invocation, the second one represents the returned value via the `!checkAvail(q:Quote)` message.

Regarding the case study, which we introduced in Sect. 2.1, the `checkAvail` operation is one of the four operations offered in the WSDL specification of the warehouse service. The remaining three operations are `auth`, `cancelTransact`, and `orderShipment`. The `auth` operation has an input message `?auth(pw:String)` and an output message `!auth(q:Quote)`. Just an input message exists for the `cancelTransact` and `orderShipment` operations, no value is returned via an output message. The input message of the `cancelTransact` operation is called `?cancelTransact`, and has a part `ref` of type `Integer`. The input message of the `orderShipment` operation is called `?orderShipment`, and has a part `ref` of type `Integer`, and a part `adr` of type `Address`. The `Address` type is a complex type sequence with elements necessary for identifying an address (the concrete elements are not relevant, here).

Figure 4 shows an STS specifying the warehouse service. Initially, the warehouse is in state `1`. Now a user of the service (in our case study the Service Under

Test (SUT), i.e. the supplier) can invoke the `checkAvail` operation by sending the `?checkAvail` message. This corresponds to the transition a from state `1` to state `2`. The guard of the transition restricts the attribute `quantity` of parameter `r` to be greater than zero. After the transition has fired, `r` is saved in the variable `qr` (which is also of type `QuoteRequest`). Next, the warehouse has to return a `Quote` object via the return parameter `q`. Three things can happen. Firstly, the requested product may not be on stock with the requested quantity. In this case a `Quote` object is returned with the status attribute being `SOLDOUT` (transition b). Secondly, if the product is on stock and the requested quantity is less than or equal some limit `MAXQ`, a `Quote` object is returned with status `VALIDQUOTE`, the same `quantity` as being requested, and a `price` and `refNumber` greater than zero (transition f). We save here the issued quote in the variable `qi`. Thirdly, if the requested quantity exceeds `MAXQ`, a quote is returned with status `AUTHREQ` (transition c). This informs the user to provide a password string via the the `auth` operation (transition d). If the password in invalid, a quote with status `PWINVAL` is returned (transition e), and the user has to invoke the `auth` operation again. Given a valid password, a valid quote is returned (transition i).



ARC	MESSAGE	GUARD	UPDATE
a	`?checkAvail<r:QuoteRequest>`	`r.quantity > 0`	`qr = r;`
b	`!checkAvail<q:Quote>`	`q.status == SOLDOUT`	
c	`!checkAvail<q:Quote>`	`qr.quantity > MAXQ && q.status == AUTHREQ`	
d	`?auth<pw:String>`		`spw = pw;`
e	`!auth<q:Quote>`	`q.status == PWINVAL && !valid(spw)`	
f	`!checkAvail<q:Quote>`	`qr.quantity <= MAXQ && q.price > 0.0 && q.status == VALIDQUOTE && q.refNumber > 0 && q.product == qr.product && q.quantity == qr.quantity`	`qi = q;`
g	`?cancelTransact<ref:Integer>`	`ref == qi.refNumber`	
h	`?orderShipment<ref:Integer, adr:Address>`	`ref == qi.refNumber`	
i	`!auth<q:Quote>`	`valid(pw) && q.status == VALIDQUOTE && q.product == qr.product && q.price > 0.0 && q.quantity == qr.quantity && q.refNumber > 0`	`qi = q;`

Fig. 4. The Provided Interface of the Warehouse as an STS.

Being in state `5`, again two things can happen. Either the user of the service decides to reject the quote. He/she invokes the one-way operation `cancelTransact` by sending the message `?cancelTransact` (transition g). Here he/she must refer to the correct issued reference number `refNumber`. Or he/she decides to accept the quote. In this case, in addition to the correct reference number, an address must be provided as a second parameter to the `?orderShipment` message (transition h).

4.2 Environmental Conformance

We show now how the above specification can be used to generate functionally correct responses within a generated stub. In model-based testing, a testing relation formally defines when a model representing the SUT conforms to a model constituting its specification [24]. A testing relation for the formalism of Labeled Transition Systems (LTS) is *eco* [11]. Since STSs have an underlying LTS semantics, we can use *eco* also for STSs.

The motivation of *eco* is to define what it means for a component C to correctly invoke a requested, or environmental, component E. Given a provided interface specification of E in terms of an LTS, an *eco* compliance checker for E plays the role of E. While doing so it checks if C respects the specification of E when invoking it.

The main observation here is, that an *eco* compliance checker for the component E does exactly what we demand from a functionally correct stub for E. Taking the warehouse STS specification given in Fig. 4, a generated *eco* compliance checker (from hereon simply called *stub*) will play the role of the warehouse, and in doing so it can test if the supplier, while using the warehouse, does respect the STS specification. To render more precisely what that means, we show an example walk by the stub trough the STS, next. Initially, the stub is in state `1`. The only allowed call here is `checkAvail` with a quantity greater zero. If it receives a different call, it will alert a detected failure of the user. If the call is correct, it moves to state `2`. Now the stub can decide nondeterministically if it either returns a quote with status `SOLDOUT` (back to state `1`), or if it checks the quantity and proceeds to state `3` or `5`. This choice is made randomly, but may also be made according to certain coverage criteria, or other heuristics. Assuming, for instance, that the stub decides to check the quantity and that the quantity is greater `MAXQ`. It then constructs a quote with status `AUTHREQ`, returns it to the user, and moves to state `3`. Now it waits for the user in invoke the `auth` operation. Assuming that the user does provide a valid password here, the stub moves to state `4` and sees that the password is valid. Next it constructs a `Quote` object with status `VALIDQUOTE`. Also here the stub has many choices, every solution to the guard on transition i corresponds to a possible quote. The stub will choose one solution (again randomly, or according to some heuristics), return the corresponding quote to the user, and move to state `5`. Now it waits again for the user to either cancel the transaction, or order the shipment. And so on. Using this approach ensures that the generated stubs give always functionally correct answers, and can detect incorrect invocations from interacting components.

5 Verifying and Utilizing the Testbed

We have shown so far the approach to generate stubs for Web Services which respect both a given SLA contract, and a functional contract in terms of a state machine. This section firstly presents in Sect. 5.1 results of experiments which we carried out to verify the stubs themselves, i.e., we tested here if the stubs

really behave as their protocols dictate, to gain confidence in the implementation environment. Having the stubs deployed, the developer can test the service under development in this testbed. Sections 5.2 and 5.3 exemplify and summarize the testing power of the generated testbed. We refer again to the customer, supplier, warehouse scenario introduced in Sect. 2.

5.1 Verifying the Generated Stubs

We have to verify that a stub which is generated by PUPPET both respects its SLA, and its STS specification. To verify the former we specified several SLAs for the warehouse service, and checked if the behavior of the generated stubs is in line with what their SLA dictates. In order to verify the latency exhibited by the stubs, we instrumented the service container of the warehouses with a simple performance monitor logger, as described in [3]. To obtain meaningful measures, we iterated each interaction scenario over 1000 executions. We always verified that the mean response time elapsed was in line with the one dictated by the SLA. To monitor reliability constraints, we associated the agreement in Fig. 3 with the operation `orderShipment` of the warehouse, and developed a supplier which repeats any invocation of `orderShipment` when a remote exception was thrown by the warehouse. Furthermore, we instrumented the supplier with logging code, counting how many invocations it executed until the shipment of the order succeeded. As in the case of latency, we monitored the reliability constraints over 1000 executions, and verified that they were always respected.

To verify the functional conformance we generated several warehouse stubs with PUPPET based on (variations of) the state machine specification from Fig. 4. To verify that they functionally conform to their specification, we automatically tested them with the model-based Web-Service testing tool JAMBITION [23], which is especially suited for this purpose since it also takes STSs as its specification formalism. No failures were observed after several hours of automatic testing. Secondly, we developed a supplier service which behaves conforming to the warehouse specification from Fig. 4, and made several non-conforming mutants of it, like *requesting a quote with a zero quantity*, or *ordering goods without having received a valid quote beforehand*. All mutants were immediately detected by the warehouse stub.

5.2 Detecting Extra-functional Failures

This case refers to the task of the developer to derive reliable values for the quality levels of the newly developed service by taking into account the QoS of the external services. Having merely extra-functional correct stubs gives a testbed as the one we described in [4]. The added value for the approach of this paper comes into play when taking into account also the functionality. The main advantage of a functional stub here is, that it has a notion of state. In other words, it forces the user to respect the functional protocol. By so doing, the user automatically walks through the specified transactions, like *ordering products after having authenticated*. For instance, the enforcement of its functional protocol by

the warehouse stub may reveal failures in the extra-functional behavior of the supplier. Going in detail, let us assume that the supplier has to meet a given SLA on latency regarding the interactions with its clients, namely processing each request within $40000msec$. As defined in the agreement with the warehouse shown in Fig. 2, each interaction between the warehouse and the supplier service can take up to $25000msec$. Take also into account, as described in Sect. 4, that the warehouse service requires an additional authentication step in case the product quantity exceeds `MAXQ` (see Fig. 4).

A potential extra-functional failure here is, that when the authentication of the supplier is required, the time needed by the supplier to fulfill a client request may violate its SLA. Even if the first password provided is correct, the response of the warehouses to the availability request (arc c), together with the response to the provided password (arc i), may in the worst case sum up to $50000msec$, which respects for each invocation the SLA exposed by the warehouse, but breaks the supposed SLA between the supplier and the client. Given a warehouse stub which does not have any notion of the functional protocol might never notice the necessity of authentication for a supplier. Each request is considered stand-alone, and no relation to previous or following requests, including data interdependencies, exist. Thus, in a mere extra-functional testbed this extra functional failure can easily be invisible.

5.3 Detecting Functional Failures

We have already shown above (end of Sect. 5.1), that the generated stubs are able to detect functional violations of their contracts by the user. This refers to the task of testing the compliance of a service under development with the functional contracts of external services, and is on its own already of high value for the developer.

Taking into account also the extra-functional correctness of the stubs, can reveal further functional issues of their users. For instance, the warehouse stub generated by PUPPET shows an extra-functional behavior which can potentially affect the functional state of the supplier (its user). To exemplify this, assume a supplier offering a special *welcome discount* to new clients for their first five purchases. Furthermore, let us consider that the supplier behaves as depicted in Fig. 5. For a given client, the supplier associates a counter `FreeOrder`, initially being five, which is decremented each time the client places an order. To fulfill the order request, the supplier invokes the `orderShipment` operation of the warehouse stub. In case a reliability failure occurs now, this is propagated to the client. Let us recall that the interactions between the supplier and the warehouse service is governed by an SLA containing a reliability clause as specified in Fig. 3. The supplier service is not prepared to deal functionally properly with such a reliability failure in the sense that it does not increase again the `FreeOrder` counter to its original value. This is necessary since the warehouse does not process the order due to its reliability failure - the products cannot be purchased by the client of the supplier. As a consequence, each reliability failure reduces the number of discounts by one, even though no goods have been

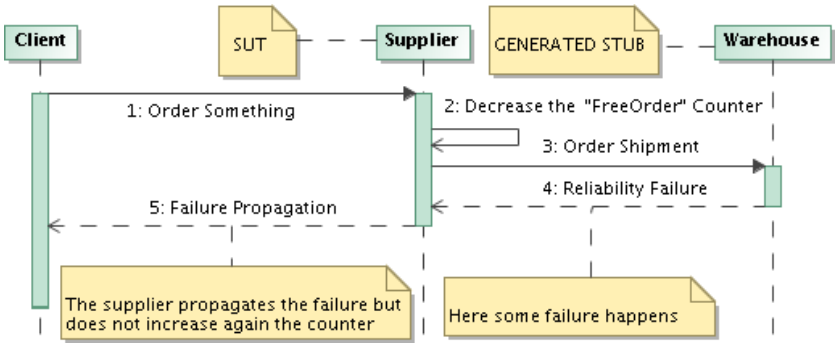


Fig. 5. Functional Fault Revealed by a Reliability Constraint

purchased by the client. Such kinds of functional failures cannot be discovered using a testbed that only reproduces functionally correct behavior, ignoring the extra-functional specifications.

6 Related Work

An early interesting work exploring QoS testing is Grabowski and Walker [14], although they did not consider QoS relations among input and output, but only temporal relations among two streams, and did not target the automatic generation of testbeds. However, they provide an interesting classification of QoS requirements. In [10], Weyuker and Vokolos highlight the potential of using testing and synthetic workload to assess the performance behavior of complex distributed applications. Denaro and coauthors [8] provide a framework for early evaluation of performance characteristics of a distributed application, taking into account the influence of complex middleware.

Concerning our goal, i.e., testbed generation for deriving or validating the contracts of composite services, without accessing the invoked real services, to the best of our knowledge no similar approaches exist that can simulate both the functional and extra-functional behavior of the surrounding environment. Ramsokul and coauthors [19] discuss a testing strategy and provide a conformance relation to assess the cooperation among services against a globally specified protocol (they do not address SLA constraints). Closer to our approach are some QoS testbeds. Zhu and coauthors [26] develop an interesting model-driven approach to the generation of benchmarks reproducing clients' workload for evaluating the QoS provided by a Web Service. Grundy and coauthors [15] propose a performance test-bed generator which has much in common with what we propose here, in that a collection of service stubs is generated from a compositional model, and clients simulating a defined workload are also automatically developed. However, in the cited works, no contract or agreement specification is assumed, and the surrounding environment is not simulated. We believe that

the above works could be used to complement PUPPET, providing the workload to solicit our generated stubs.

Another interesting issue is how the network affects the QoS. As said, the provider of a service is responsible of the agreed SLAs at his/her port. Possible network problems are not his/her business, so the customer should handle such issues with the network provider [22]. Nevertheless the QoS perceived by the user of a service is also affected by the network. Therefore, to reproduce likely distribution settings and network delay distributions, the tester should use a network in which it is possible to control the introduced delay. In this respect an interesting tool is *Weevil* [25], that supports the creation of a synthetic workload for distributed software and can reproduce realistic stub distribution over worldwide distributed platforms.

7 Conclusions and Future Work

It is well known that testing amounts for the most part of its effort to coding, and great part of this coding effort is needed for making test cases executable. Automation of the setup of the testing environment is routinely addressed in conventional software development. Such need also affects, or is even exacerbated, in the testing of service-oriented systems, as the provision of a service commonly depends on other surrounding services.

We have presented the PUPPET environment for the automatic generation of stubs simulating the behavior of external services invoked by a service under test, which encapsulates both their functional specifications and their contractual SLAs. To achieve this, we assume that we have access to an extra-functional specification (SLA), and a functional specification (STS), of the surrounding services. Using these specifications, the PUPPET tool automatically generates for each surrounding service a stub which respects the functional and extra-functional properties. These stubs constitute an environment which realistically simulates the runtime environment. The service developer can test the SUT within it, and obtain realistic QoS measures, without having the need to access the real surrounding services, that might not be available, or could generate unwanted side effects. We have illustrated why considering both functional and extra-functional aspects for the stub generation is a crucial means to raise potential failures and obtain realistic estimates for the QoS attributes.

The integration of the extra-functional and functional contracts is currently done in pragmatic way, aiming at a proof-of-concept of the presented approach. Further investigations are necessary to come up with a theoretical foundation of such an integration. One promising candidate here is the combination of functional models like STSs with concepts known from timed automata [2,5].

Having in place the PUPPET testbed, a suitable test suite must be generated. This task is out of the scope of PUPPET, but the framework already gives indications on how this could be achieved. Obtaining reliable values demands for the execution of many tests, thus test generation here is a strong candidate for automation. Model-based testing (MBT) has precisely this goal, i.e., the au-

tomatic generation, execution, and evaluation of test cases based on a (usually functional) formal model of the SUT. A natural choice would be to keep the model already used by PUPPET, namely STSs, also for functionally modeling the provided service of the SUT. We could use some existing STS-based MBT tools for doing so (like [7,23]). Future research has to explore how to precisely combine functional MBT of the SUT with an effective derivation of the corresponding QoS properties. Also here the promising candidate is the integration with timed automata models.

Finally the experiments we conducted so far have been aimed at verifying that the generated stubs are correct with respect to the (functional and extra-functional) specifications. As a next step, we are currently involved in further experiments directed at validating PUPPET in practice. Specifically, in collaboration with the industrial partners of the PLASTIC project [9], we are using the stubs generated by PUPPET on wider and more complex case studies.

8 Acknowledgements

This research is carried on within the European Project PLASTIC (IST STREP 026955), and supported by the Marie Curie RTN TAROT (MRTN-CT-2004-505121) and the Netherlands Organisation for Scientific Research (NWO) under project STRESS.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services–Concepts, Architectures and Applications*. Springer, 2004.
2. R. Alur. Timed Automata. In *Computer Aided Verification*, pages 8–22, 1999.
3. S. Jeelani Basha and Romin Irani. *AXIS: the next generation of Java SOAP*. Wrox Press, 2002.
4. A. Bertolino, G. De Angelis, and A. Polini. A QoS Test-bed Generator for Web Services. In *Proc. of ICWE 2007*, number 4607 in LNCS. Springer, 2007.
5. L. B. Briones and E. Brinksma. A Test Generation Framework for quiescent Real-Time Systems. In J. Grabowski and B. Nielsen, editors, *Proc. of FATES 2004*. Springer, 2004.
6. E. Christensen et al. Web Service Definition Language (WSDL) ver. 1.1. http://www.w3.org/TR/wsdl/, 2001.
7. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a Symbolic Test Generation tool. In *Proc. of TACAS 2002*, number 2280 in LNCS. Springer, 2002.
8. G. Denaro, A. Polini, and W. Emmerich. Early Performance Testing of Distributed Software Applications. In *Proc. of WOSP 2004*, pages 94–103. ACM Press, 2004.
9. PLASTIC european project homepage. http://www.ist-plastic.org.
10. 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.
11. L. Frantzen and J. Tretmans. Model-Based Testing of Environmental Conformance of Components. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.P. de Roever, editors, *Proc. of FMCO 2006*, number 4709 in LNCS, pages 1–25. Springer, 2007.

12. L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Proc. of FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.
13. Global Grid Forum. *Web Services Agreement Specification (WS–Agreement)*, version 2005/09 edition, September 2005.
14. J. Grabowski and T. Walker. Testing Quality-of-Service Aspects in Multimedia Applications. In *Proc. of PROMS 1995*, 1995.
15. J. Grundy, J. Hosking, L. Li, and N. Liu. Performance Engineering of Service Compositions. In *Proc. of IW-SOSE 2006*, pages 26–32. ACM Press, 2006.
16. Boudewijn R. Haverkort and Ignas G. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation*, 25(1):17–40, 1996.
17. M.N. Huhns and M.P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005.
18. Object Management Group. *UML 2.0 Superstructure Specification*, ptc/03-08-02 edition. Adopted Specification.
19. P. Ramsokul, A. Sowmya, and S. Ramesh. A Test Bed for Web Services Protocols. In *Proc. of ICIW 2007*, pages 16–21, 2007.
20. Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and Reliability Analysis of Computer Systems An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, Nov 1995.
21. J. Skene, D.D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. of ICSE 2004*, pages 179–188. IEEE Computer Society Press, 2004.
22. J. Skene, A. Skene, J. Crampton, and W. Emmerich. The Monitorability of Service-Level Agreements for Application-Service Provision. In *Proc. of WOSP 2007*, pages 3–14. ACM Press, 2007.
23. PLASTIC tools homepage. http://plastic.isti.cnr.it/wiki/doku.php/tools.
24. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
25. Y. Wang, M.J. Rutherford, A. Carzaniga, and A.L. Wolf. Automating Experimentation on Distributed Testbeds. In *ASE 2005*, pages 164 – 173. ACM, 2005.
26. L. Zhu, I. Gorton, Y. Liu, and N.B. Bui. Model Driven Benchmark Generation for Web Services. In *Proc. IW-SOSE 2006*, pages 33–39. ACM Press, 2006.