# The PLASTIC Framework and Tools for Testing Service-Oriented Applications

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
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.** The emergence of the Service Oriented Architecture (SOA) is changing the way in which software applications are developed. A service-oriented application consists of the dynamic composition of autonomous services independently developed by different organizations and deployed on heterogenous networks. Therefore, validation of SOA poses several new challenges, without offering any discount for the more traditional testing problems. In this chapter we overview the PLASTIC validation framework in which different techniques can be combined for the verification of both functional and extra-functional properties, spanning over both off-line and on-line testing stages. The former stage concerns development time testing, at which services are exercised in a simulated environment. The latter foresees the monitoring of a service live usage, to dynamically reveal possible deviations from the expected behaviour. Some techniques and tools which fit within the outlined framework are presented.

## 1 Introduction

A widely used approach to validation in industrial software development is *testing*, which consists of observing the behavior of a program under some controlled executions [7]. Indeed, testing provides a feasible and effective strategy to check that a software implementation conforms to the specifications, or to evaluate its dependability and performance.

In the years, many different methods for test selection and execution have been proposed. As new software paradigms emerge, testers have to take into account many new features that in most cases make existing testing techniques no more sufficient or sometimes not even applicable. Therefore, testing methods and tools need to be continuously adapted and empowered to face the exigencies posed by the evolution of the development process and programming approaches.

The latest shift in software development is the *Service-oriented Architecture (SOA)*. All leading IT vendors, including IBM, SAP, Microsoft, Oracle, have already moved towards service-centric. Service-based technology promises to easily integrate software components deployed across distributed networks by different providers. Its great appeal derives from the announced flexibility and interoperability among heterogeneous platforms and operating systems, that is achieved by means of *loose coupling*, *implementation neutrality* and *flexible configurability* [22].

Loose coupling among services means that the mutual dependencies are minimized and maintained exclusively through standardized interfaces. The latter feature ensures implementation neutrality, in that the internal implementation details of a service must be totally uninfluential: a service must be seen as a "black box". But the most compelling feature of SOA is that its configuration can *change dynamically as needed and without loss of correctness* [22].

Unfortunately, these very same features that make SOA highly attractive to vendors and integrators, pose new difficult challenges to testers. Simplifying in a sentence, testing a program (or a subprogram) amounts at anticipating at development time a relevant and comprehensive sample of the potential program invocations in operation. Hence traditional testing presupposes that before a product is released, someone from the development organization or a third party is sufficiently acquainted with the intended system behavior and can control its configuration, so to select and launch an adequate executions sample (the test suite).

It is evident that in service integration such an assumption does not hold anymore: according to the SOA paradigm, services can discover each other at run-time and can select the partner to interact with based on parameters that are only defined at run-time. Therefore, new approaches and means for testing must be sought.

Interesting challenges for validation also stem from the great relevance of extra-functional requirements in SOA. As services are used in a pervasive way, in fact, ensuring adequate levels of their provided Quality of Service (QoS) becomes as important as guaranteeing their proper functional behavior. Therefore, an additional task that testers need to accomplish concerns the advance evaluation of the QoS characteristics of the System Under Test (SUT).

All the above issues are attracting great interest from researchers in industry and academia, and several techniques and tools have been proposed for SOA testing. A broad survey of such approaches is provided in Chapter 4 [9] of this book, and we refer to it for related work discussion. In this chapter we specifically focus on the framework for SOA testing developed in the European Project PLASTIC (Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication) [12].

PLASTIC aims at facilitating the cost/effective development of adaptable context-aware services, with a special emphasis on enforcing service dependability. The project has been inspired by the vision that service development platforms for B3G (Beyond 3rd Generation) networks will be effective and successful only if the services they deliver are adaptive and offer Quality of Service (QoS) guarantees to users despite the uncontrolled open wireless environment. This vision is pursued via a development paradigm based on Service Level Agreements (SLAs) and resource-aware programming. The project, started in 2006 and now approaching its conclusion, has developed a SOA platform integrating:

– A development environment leveraging model-driven engineering for the rigorous development of SLA- and resource-aware services, which may be deployed on the various networked nodes, including handheld devices;
– A middleware leveraging multi-radio devices and multi-network environments for applications and services run on mobile devices, further enabling context-aware and secure discovery and access to such services;
– A validation framework enabling *off-line* and *on-line* testing of networked services, encompassing functional and extra-functional properties.

This chapter is meant as a brief summary of the PLASTIC validation framework and is not exhaustive of all research results and tools achieved in the project. Further information of the other PLASTIC tools and approaches can be found in the project deliverables and publications [12]. Indeed, another chapter in this volume [24] broadly discusses challenges for the software of the future, and in particular it illustrates possible solutions that have been investigated within PLASTIC. In this chapter we expand the discussion to challenges and solutions concerning testing activities.

Focusing on the validation framework, the above distinction between off-line and on-line approaches concerns the stage and the context in which the testing is carried on. In off-line testing, the system is still undergoing development or, in a wider-ranging view, it is already completed but not available for use yet. Hence, off-line validation implies a more traditional view that a system is tested in the laboratory, within an environment that reproduces or simulates possible real interacting situations. The advantage of off-line validation activities is that these are performed while no customer is using the service, thus avoiding undesired side-effects. On the other hand, on-line approaches concern a set of increasingly used techniques to monitor the system in its real working context after its deployment. A possible scenario is that while the end-users are using the service, real data are collected and sent back to the developers, e.g., for determining whether the service behavior is correct or for performing extra-functional analyses. Hence, on-line "test cases" consist of actual usage scenarios. Another scenario can be that the development organization performs on-line validation activities on a fielded service, with selected test cases, possibly during idle times.

In the following of this chapter we first provide (in Sect. 2) a brief overview of the whole PLASTIC validation framework. In Sect. 3 we provide some background information to the presented tools, and outline an example scenario on which the tools application is illustrated. Then, in Sect. 4 we present in more detail a set of testing tools proposed for testing a service before it is published, namely the tool Jambition (Sect. 4.1) for functional off-line testing, and the tool Puppet (Sect. 4.2) for the generation of a testbed respecting both functional and extra-functional specifications. Sect. 5 presents and discusses the Audition framework (Sect. 5.1) that supports on-line testing before a service is published. Conclusions are drawn in Sect. 6.

## 2 The PLASTIC Validation Framework

The PLASTIC project aims at enabling the development and deployment of adaptable robust services for B3G networks. For this purpose, it has developed a comprehensive platform integrating both adequate software methodologies and tools, and the supporting middleware. The project has devoted special concern to equip the platform with suitable validation technology. A team of several partners has contributed with different approaches and tools for service analysis, testing and evaluation. Such approaches and tools are intended for usage at different phases of the lifetime of a service.

With reference to Fig. 1 below, a service life-cycle can include the following steps: after being developed, it must be installed and deployed for being made accessible to potential customers. To facilitate discovery, a service provider can then explicitly request to be registered with a registry (such as the UDDI [28]). We distinguish between the act of submitting the request for being included in the register, which we indicate as the "Admission" stage, and the actual inclusion of the service in the registry (publication), after which the service is made publicly accessible. Finally, the service enters live usage.

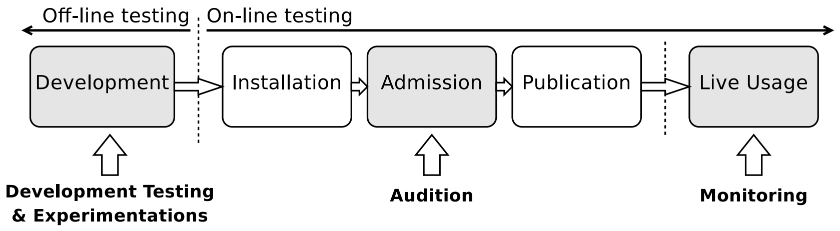


**Fig. 1.** PLASTIC Testing stages

Considering this service life-cycle model, three testing stages have been identified in the PLASTIC validation framework. They are shown in the figure with arrows pointing to the related life-cycle stages (Development, Admission, Live Usage), and include:

– Development Testing and Experimentation;
– Audition;
– Monitoring.

In principle, it is assumed that the three stages proceed in sequential order, since a service is first developed, then deployed, then used. There are however clear relationships among the stages. In particular the results of the analyses conducted off-line may be used to guide the on-line validation activities. Moreover, the results from analysis during the on-line validation might provide a feedback to the service developer or to the service integrator, highlighting necessary or desirable evolutions for the services, and therefore might be used for off-line validation of successive enhanced versions of services.

Having made clear the context for the three introduced stages for validation in PLASTIC, we describe in the following of this section the integrated framework in which all the PLASTIC developed approaches and tools fit together. An overall picture of the PLASTIC Validation Framework[1] is illustrated below in Fig. 2. Given the broad variety of PLASTIC target applications, this validation framework is not conceived as a fixed methodology, but rather as a set of techniques/tools that can be used alternatively, or in combination, depending on the constraints and exigencies of the considered application/scenario.

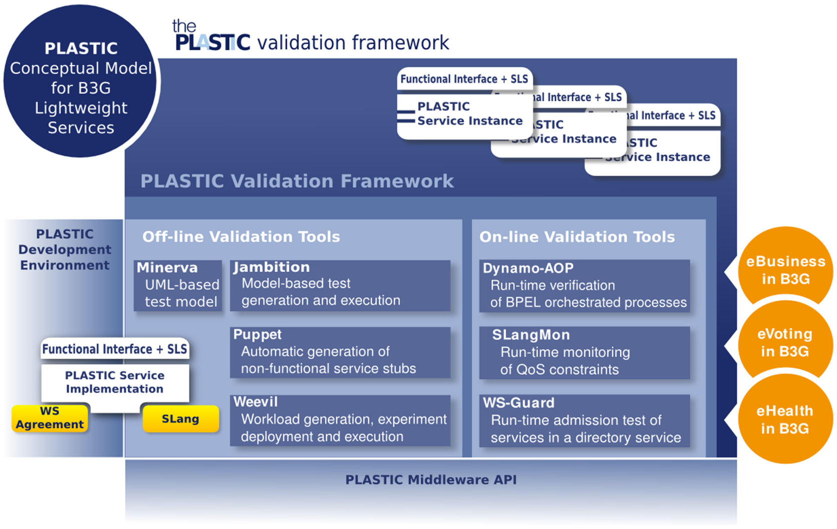

PLASTIC Conceptual Model for B3G Lightweight Services
the PLASTIC validation framework
Functional Interface + SLS
PLASTIC Service Instance
PLASTIC Validation Framework
PLASTIC Development Environment
Off-line Validation Tools
Minerva UML-based test model
Jambition Model-based test generation and execution
Puppet Automatic generation of non-functional service stubs
Weevil Workload generation, experiment deployment and execution
On-line Validation Tools
Dynamo-AOP Run-time verification of BPEL orchestrated processes
SLangMon Run-time monitoring of QoS constraints
WS-Guard Run-time admission test of services in a directory service
PLASTIC Service Implementation
WS Agreement
SLang
eBusiness in B3G
eVoting in B3G
eHealth in B3G
PLASTIC Middleware API

**Fig. 2.** PLASTIC Validation Framework

## 2.1 Development-Time Testing

As shown in Fig. 2, the PLASTIC off-line testing tools include: JAMBITION (with its library MINERVA), PUPPET and WEEVIL.

The JAMBITION tool relies on a model-based testing approach originating from a sound and well-established formal testing theory. The key idea of this tool is to exploit as much as possible the behavioral description often available for deployed services to automatically derive a conformance test suite for a service under development. Due to the extreme dynamicity of the service domain, many authors have suggested to augment the service WSDL description with operational specifications in order to characterize services in a richer way. JAMBITION

[1] All the PLASTIC test tools can be freely downloaded at [35].

assumes that such specifications are available in the Symbolic Transition System formalism, as introduced in Sect. 3.2, or through its UML interface, implemented via the Minerva library. The Jambition tool is illustrated in Sect. 4.1.

As services are discovered and integrated only at run-time, it is difficult – if at all possible – to have advance guarantees on service behavior. This is particularly true when extra-functional properties are considered. Nevertheless, developers need tools and techniques to assess the quality of a service before its final deployment. Services in general may invoke other services in order to carry out the computation requested by the clients. If this invocation is directed to a service that does not refer to stateful resources, then for testing purposes it is possible to use existing and already running services. Conversely, if the invoked service accesses stateful resources, this option must be ruled out and the required services have to be simulated. Within PLASTIC the problems of reproducing predictable run-time environment is addressed providing two different tools, Puppet and Weevil, which allow the developers to mock up different live usage scenarios.

In particular, Puppet supports the automated derivation of the elements necessary to recreate a predictable "live" environment that is suitable for the evaluation of extra-functional properties. Puppet allows testers to automatically generate the required services in such a way that they yield the "correct functional and extra-functional behavior" with respect to a given specification. As in the case of Jambition, we assume that the functional specification is given by means of Symbolic Transition System formalism. Concerning the extra-functional specification, we assume that it is based on the WS-Agreement language, as discussed in Sect. 3.3. How Puppet works is further discussed in Sect. 3.3.

Finally, Weevil is meant to ease the reproduction of distributed experimental environments. In particular it permits to recreate expected workload to stimulate the service under test, to remotely deploy the various element required by the experiment, and to collect data during the experiment. Weevil is not further discussed in this chapter, we refer to the PLASTIC project site [12] for more information and for getting the tool.

## 2.2 Admission Testing

The SOA foresees the existence of a service broker that is used by services to search and obtain references to each other. The idea of Admission testing is to have the service undergo a preliminary testing stage (also referred to as an audition) whose results will decide the actual registration of the service in the directory.

The intuition of Admission testing is that the quality of registered services can be increased by granting the registration only to those services that pass the audition testing phase. At the same time this should provide better confidence in the fact that services will interact in a correct way even if they discover each other at run-time.

Admission testing clearly raises issues regarding the invocations to fully-operating services (as opposed to services being auditioned). This may be particularly dangerous if the services invoked are related to stateful resources. In order to avoid side effects resulting from invocations fired in the process of auditing a service, suitable countermeasures must be taken. Sect. 5.1 introduces the issues behind admission testing, while Sect. 5.2 presents WS-Guard, which is an implementation of a directory service conforming the UDDI specification that permits to test services before their registration.

### 2.3 Live Usage Verification

Difficulties in applying verification techniques before live usage suggest to extend the verification phase till run-time. The idea is to add suitable mechanisms to the SUT and the middleware so as to detect violations with respect to the expected behavior of services.

Within PLASTIC two different activities, aiming at the development of monitoring mechanisms, have been activated. The first of these approaches, called Dynamo-AOP, focuses on functional behavior of orchestrated services, and provides support to augment orchestrating services with checks, in order to verify that the orchestrated services behave as expected.

Another approach in this category, called SLAngMon, supports the monitoring and logging of extra-functional properties for running services. In particular, SLAngMon implements a mechanism to automatically generate on-line checkers of Service Level Agreements (SLAs). The approach is founded on the timed automata theory.

Dynamo-AOP and SLangMon are not further discussed in this chapter, but are both extensively documented and made available in the PLASTIC web page.

## 3 Modelling Service Properties

In this section we introduce the modelling notations adopted by the tools presented in the chapter. The notations are exemplified on a simple case study, presented below.

### 3.1 An Example Service Scenario

We will exemplify the several testing approaches and tools on a common case study. It is a simplified variant of the scenario presented in [2], 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 the warehouse service to check if the requested quantity is in stock. The information provided by the warehouse is then returned to the customer service. If satisfied with the provided quote, the customer can proceed with the order. We will also look at advanced interactions like supplier authentication and bonus accounting. We can realistically assume that the three services are implemented and provided

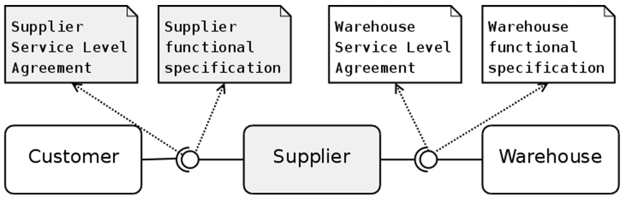

Supplier Service Level Agreement
Supplier functional specification
Warehouse Service Level Agreement
Warehouse functional specification
Customer
Supplier
Warehouse

**Fig. 3.** The Customer-Supplier-Warehouse Case Study

by different stakeholders, and that their interactions are governed by functional specifications under agreed levels of QoS, as shown in Fig. 3.

## 3.2 Modeling Functional Behavior

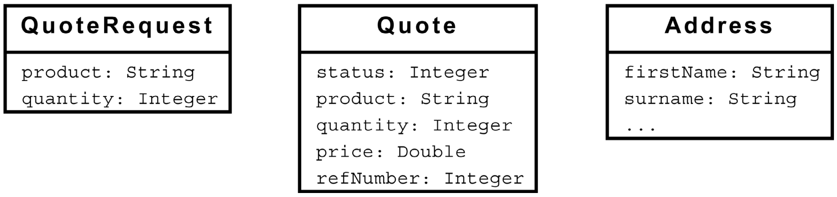
The functional behavior of a stateful software entity like a class, component, or service, is commonly modelled using a state machine. There are many flavors of such machines defined. For model-based testing purposes there are two main classes of relevant models - *Labelled Transition Systems* (LTSs) [8] and *Mealy Machines* [26] (often just referred to as *Finite State Machines* (FSMs)). Most common model-based testing theories are based on either the LTS or the FSM model. One important feature of these models is their simplicity - labels on transitions correspond to basic actions like for instance PUSHING THE BUTTON, or INVOKING OPERATION SUCC(41) ON THE SERVICE. Whereas this simplicity is helpful in defining testing theories and algorithms, it is often hindering for modeling real-world systems. Instead of using just basic actions, one would like to use concepts known from programming languages like variables, conditional branching, etc. Such concepts are sometimes referred to as *symbolic concepts.* One prominent symbolic model is the UML 2.0 state machine [29]. To use the accessible and broadly accepted model of an UML state machine together with the precise and well-defined testing theories, one has to define a mapping from UML state machines to LTSs or FSMs.

A model which is somewhat similar to UML state machines is a *Symbolic Transition System* (STS). STSs are a well studied formalism in modeling and testing of reactive systems [16], and they can be mapped to LTSs. Still, also STSs could sound unfamiliar and difficult for practitioners. But since STSs are close to UML state machines we developed a library called MINERVA [35], which transforms UML state machines modelled with MAGICDRAW [19] – a commercial UML modeling tool – into STS representations understood by the tools we will present later. Thus, a developer can use MAGICDRAW to model the functionality of service interfaces in the common formalism of UML state machines. We do not describe this transformation here, but present instead directly the STS formalism. Finally, the important notion of a *testing relation*, which precisely defines when a system conforms to its specification, is introduced.

**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 [10]. Secondly, there are the dynamic constituents like states, and transitions 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 [16]. Instead, we exemplify the concepts in the setting relevant for this paper.

We assume that data types in the WSDL are specified via XML Schema types, as commonly done. For our example scenario we first have a closer look at the warehouse service. Firstly, we need some complex types to represent quote requests, quotes, and addresses. We depict them in form of a class diagram:



The next table lists the operations we assume to be present in the WSDL of the warehouse:

| Operation | Input Message | Output Message |
|---|---|---|
| checkAvail | ?checkAvail(r : QuoteRequest) | !checkAvail(q : Quote) |
| auth | ?auth(pw : String) | !auth(q : Quote) |
| cancelTransact | ?cancelTransact(ref : Integer) | — |
| orderShipment | ?orderShipment(ref : Integer, adr : Address) | — |

Since the two operations `cancelTransact` and `orderShipment` do not have an output message, they are, in the WSDL jargon, *oneway* operations. The other operations have an input and an output message - they are *request-response* operations. Figure 4 shows an STS specifying the warehouse service. Initially, the warehouse is in state `1`. Now a user of the service (in our example the supplier) can invoke the `checkAvail` operation by sending the `?checkAvail` message. This corresponds to the transition from state `1` to state `2`. The guard of the transition [between square brackets] restricts the attribute `quantity` of parameter `r` (which is of type `QuoteRequest`) to be greater than zero. After the transition has fired, the requested quote object `r` is saved in the variable `qr` (which is

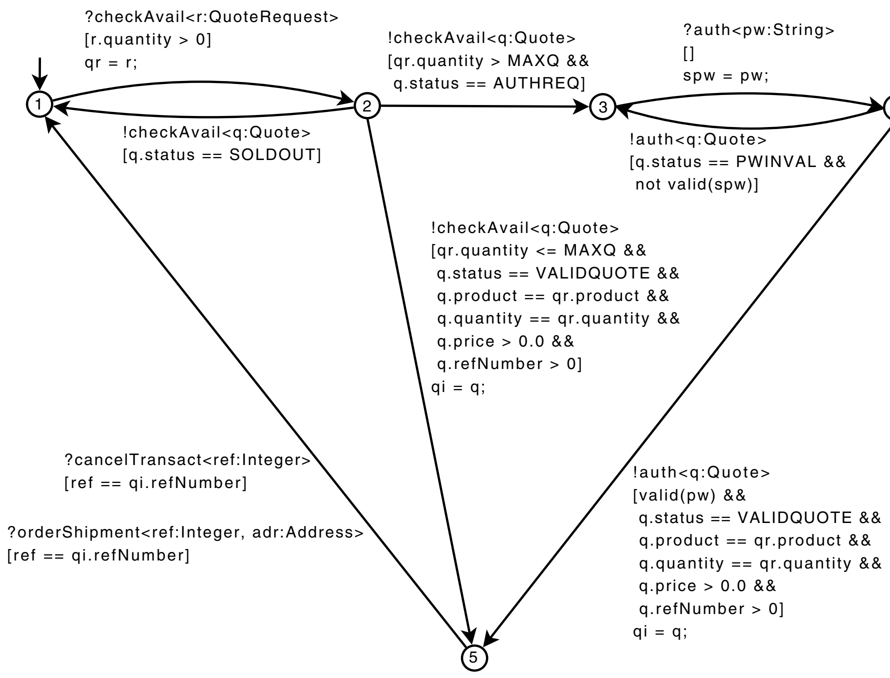


**Fig. 4.** The Warehouse STS

also of type `QuoteRequest`) via the update statement `qr = r;`. 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 in stock with the requested quantity. In this case a `Quote` object is returned with the status attribute being `SOLDOUT` (transition from state `2` to state `1`). Secondly, if the product is in stock and the requested quantity is less than or equal some limit `MAXQ`, a `Quote` object is returned with status `VALIDQUOTE`, the same `product` and `quantity` as being requested, and a `price` and `refNumber` greater than zero (transition from state `2` to state `5`). We save here the issued quote `q` in the variable `qi`. Thirdly, if the requested quantity exceeds `MAXQ`, a quote is returned with status `AUTHREQ` (transition from state `2` to state `3`). This informs the supplier to provide a password string via the `auth` operation (transition from state `3` to state `4`). If the password is invalid, a quote with status `PWINVAL` is returned (transition from state `4` to state `3`), and the user has to invoke the `auth` operation again. Given a valid password, a valid quote is returned (transition from state `4` to state `5`). 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`. 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

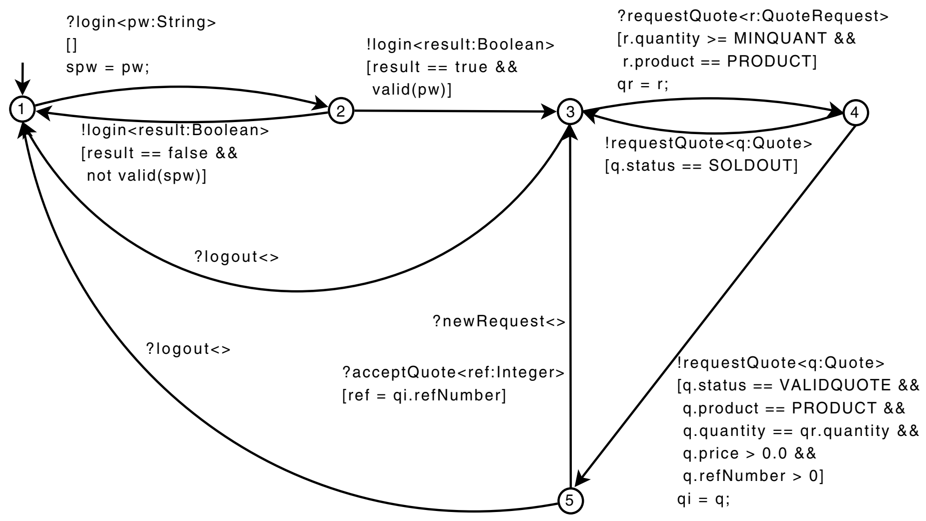


**Fig. 5.** The Supplier STS

address must be provided as a second parameter to the `?orderShipment` message (both messages are labelled at the transition from state `5` to state `1`).

Next we have a look at the STS specifying the supplier service, see Fig. 5. Its WSDL also specifies the `QuoteRequest` and `Quote` complex types, as shown above. The operations are as follows:

| **Operation** | **Input Message** | **Output Message** |
|---|---|---|
| login | ?login(pw : String) | !login(result : Boolean) |
| requestQuote | ?requestQuote(r : QuoteRequest) | !requestQuote(q : Quote) |
| acceptQuote | ?acceptQuote(ref : Integer) | — |
| newRequest | ?newRequest() | — |
| logout | ?logout() | — |

The supplier interface is relevant for the customer service to request quotes at the supplier. After a customer service has passed the login procedure (transitions between states `1`, `2`, and `3`), a quote can be requested (transition from state `3` to state `4`). This supplier is specified to deal only with one specific product, represented by the constant `PRODUCT`. Only quote requests for this product are allowed by the guard `r.product == PRODUCT`. Furthermore, the requested quantity has to be at least `MINQUANT`. Also the supplier uses the quote status `SOLDOUT` to indicate to the customer that it could not reach a warehouse with the product in stock (transition from state `4` to state `3`). If, instead, a warehouse could be reached with the product in stock, the corresponding quote is returned to the customer (transition from state `4` to state `5`). Finally, the customer can either accept the quote, ignore the quote and request a new quote (both via transition from state `5` to state `3`), or end the transaction (transition from state `5` to state `1`).

**Testing Relations.** A *testing relation* precisely defines when a SUT conforms to its specification. Even though SUTs are not formal models, but physical systems, one can define this relation formally. The trick is to assume that the SUT can be represented by some formal model (like an LTS or FSM). Having this assumption, one can reason about the SUT by reasoning about the formal model it represents. This assumption is referred to as the *testing hypothesis.* Albeit this formal model is just assumed to exists, but not known for a given SUT, one can define a testing relation by relating the formal models representing SUTs with formal models representing the specifications. The gain of this effort is, that one can unambiguously express what a testing algorithm is testing for, since the notions of *passing* or *failing* a test case are formally defined. Furthermore, the testing algorithm itself can be proven to be sound and complete for a given testing relation, see also [36].

For the tools to be presented later in this paper two testing relations are specifically important – ioco [36] and eco [14]. Both relations originate from the domain of reactive systems. Such a reactive system is a more complex system than the services we deal with here. The main difference is that we assume services to be *passive.* What does that mean? Every service provides some operations via their WSDL to potential users, who can connect to the service and invoke those operations. If it is a *request-response* operation, the service will send a response message back, but it will never send a message to the user without being requested beforehand. Even though the WSDL allows in principle to specify *active* services via *solicit-response* and *notification* operations, such services are not in common use since they do not easily map to current programming paradigms and service deployment infrastructures. To overcome some issues related to the lack of active services, techniques like *asynchronous access* via *callback handlers* [27] are used.

Due to the restriction to passive services the testing relations simplify, concepts like *quiescence* [36] are not relevant here. We do not formally define the relations, but give instead their intuitive meaning, and some hints to implementation issues. For the precise definitions please refer to the cited papers.

ioco tests the provided interface of an SUT. For passive services it simplifies to the requirement: *If the Web Service produces a response message $x$ after some specified trace $\sigma$, then the STS specification can also produce response message $x$ after $\sigma$.* In other words, each observed response message must be allowed by the STS specification. For instance, a ioco-tester for the warehouse would play the role of the supplier, and test if the warehouse behaves conforming to its STS specification, given in Fig. 4. It requires that the requested quantity must be greater zero (transition from state `1` to state `2`). Since this is an input message, it is under the control of the ioco-tester, which has to take care to respect this requirement by constructing a quote request with a positive quantity, and invoking the `checkAvail` operation with it. Which exact positive quantity is chosen, is at the discretion of the tester. We see here two tasks an automatic test tool must perform:

– construct input data which respects the given guard
– select a concrete input value in case of multiple solutions

Both problems can be difficult especially when dealing with symbolic models. We will come back to this when presenting the specific tools.

The STS does not specify what should happen when a zero or negative quantity is requested, we call this a *partial* specification since there are *underspecified inputs*[2]. Since it is not specified, it is also not tested for, the default interpretation is that everything is allowed after non-specified inputs.

After having requested the quote, the warehouse must response a `Quote` object. Here, the tester will receive the quote and check if it matches one of the three cases specified (transitions from state `2` to states `1`, `3`, and `5`). A potential failure here is for instance a returned quote having status `AUTHREQ`, even though the requested quantity was less or equal `MAXQ`. Or, the returned quote has status `VALIDQUOTE`, but deals with a different product or quantity than requested – and so on. We will come back to ioco-based testing when explaining the JAMBITION tool in Sect. 4.1 and the WS-GUARD tool in Sect. 5.2.

ioco aims at testing if a service does conform to its interface specification. The question is: *Does the service give the specified responses?* The motivation of *eco* is to answer the question: *Is the service correctly invoked by other services?* The situation is somewhat dual to the ioco case. The starting point in both cases is an STS specification of some service $S$. A ioco-tester takes the STS, plays the role of a *user of* $S$, and generates requests to $S$ as explained above. An eco tester, instead, takes the STS, plays the role of $S$ *itself*, and checks if a user of $S$ respects the STS specification. Again in simple words *eco* means here that each operation invocation to $S$ must be allowed by its STS specification.

Taking again the warehouse STS specification given in Fig. 4, an eco-tester plays the role of the warehouse, and in doing so it can test if the supplier, while using the warehouse, does respect the STS specification. Initially, the STS is in state `1`. The only allowed call here is `checkAvail` with a quantity greater zero. If the eco-tester receives a different call from the supplier, it will alert a detected failure of the supplier. If the call is correct, it moves to state `2`. Now the tester can decide 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`. We see here another choice a test tool has to make:

– choose a transition in case of multiple options

This choice, together with the choice of a concrete input value (see above) does affect the way the state space of the specification and the SUT is covered. Since specification- and code-coverage are basically the only means to measure test effectiveness, several approaches exist for making these choices. From simple random choices to sophisticated techniques based, for instance, on symbolic

[2] Underspecification of inputs has important consequences for the *compositionality* of ioco and its interpretation of non-determinism. This is out of the scope of this paper, please refer to [38].

execution of the model and/or the source-code of the SUT, exist, see for instance [31,34].

Let us further assume that the `eco`-tester decides to check the requested quantity and that the quantity is greater `MAXQ`. It then constructs a quote with status `AUTHREQ`, returns it to the supplier, and moves to state `3`. Now it waits for the supplier to invoke the `auth` operation. Assuming that the supplier does provide a valid password here, the `eco`-tester moves the STS to state `4` and sees that the password is valid. Next it constructs a `Quote` object with status `VALIDQUOTE`. Also here the tester has many choices, every solution to the guard on the transition from state `4` to state `5` corresponds to a possible quote. After having made that choice the tester moves the STS to state `5`. Now it waits again for the supplier to either cancel the transaction, or order the shipment. And so on.

One main observation here is, that an *eco*-tester for a service $S$ does exactly what we demand from a functionally correct *stub* for $S$. It accepts invocations and always returns responses which are allowed by the given STS specification. We will come back to `eco`-based testing when explaining the Puppet tool in Sect. 4.2 and the WS-Guard tool in Sect. 5.2.

## 3.3 Modeling Extra-functional Behavior

In recent years both industry and academia have shown a great interest on expressing and modeling extra-functional properties by means of machine-readable artifacts. Specifically, several proposals on specification languages for SLAs exist (e.g. [23], [32], [17]).

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 [33]. QoS guarantees are usually defined only as a provider constraint, and do not include any kinds of events that the client may experience, for example due to the mobility of the devices or traffic congestion problems.

Nevertheless, in some scenarios it would be interesting to deal with the QoS perceived by the clients rather than the QoS offered by the services. Within Puppet, we refer to a QoS testbed generator that can take into account also how the mobility of the devices hosting the services can affect the QoS provided at the service port (see Sect. 4.2).

In the rest of the section, we describe WS-Agreement [17], one widely used notation in modeling extra-functional behavior in the Web Service communities.

WS-Agreement [17] is a language defined by the Global Grid Forum aiming at providing a standard layer to build agreement-driven SOAs. The main ingredients 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.

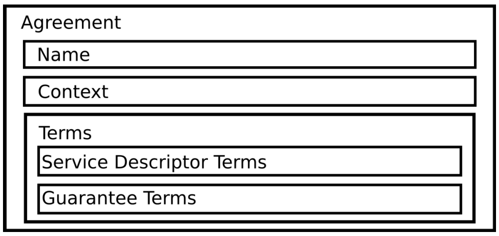


**Fig. 6.** WS-Agreement Structure

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.

As introduced in Sect. 2.1, within PUPPET we use the QoS properties contained in an agreement specification in order to automatically derive the elements necessary to recreate a testbed that is suitable for predicting the extra-functional properties of the SUT.

Specifically, for each concept in the WS-Agreement (i.e., `SLO`, `Qualifying Condition`, `Service Scope`) we define an interpretation of it by means of a given operational semantics. Clearly, 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.

Precisely, such operational semantics is defined as a mapping from the declarative XML descriptions of the supported QoS properties to composable Java code segments. Such segments are then injected into the stubs composing the testbed in order to emulate the extra-functional behavior. 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 [12]. Nevertheless, also other definitions can be adopted (e.g. as in [30]).

```
...
<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. 7.** SLO Mapping for Latency

*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 can be simulated generating *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. 7.A reports the XML code for a maximum latency declaration of $25000msec$ normally distributed, and Fig. 7.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. 7. 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. Within PUPPET, we map 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. 8. 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.

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

```
...
<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. 8.** SLO Mapping for Reliability

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 transformation function can wrap the simulating code 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.

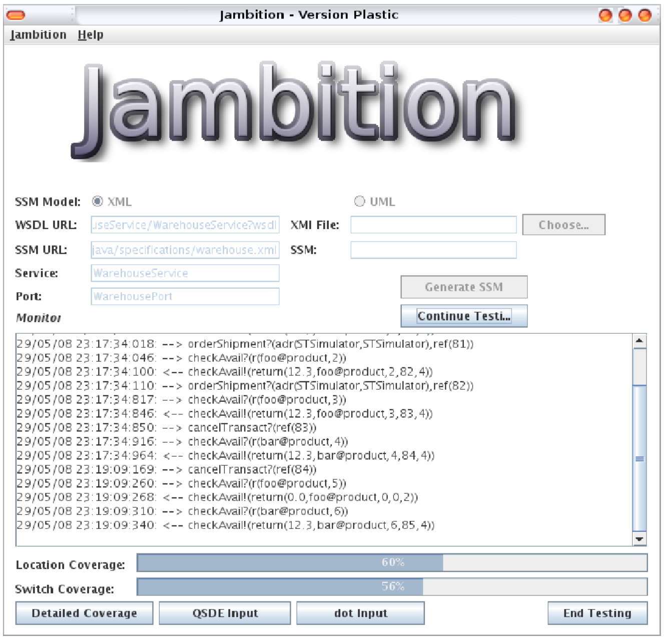
## 4 Off-Line Testing Tools

In this section two tools for off-line testing will be introduced: Jambition and Puppet. Whereas the former is a model-based functional testing tool, the latter is an automatic generator for service stubs respecting both a functional- and an extra-functional specification. The underlying models and theories have been explained in the preceding Sect. 3.

### 4.1 Jambition

Jambition [35] is a Java tool which automatically tests Web Services based on STS specifications, Fig. 9 shows a screen-shot. The underlying testing relation is ioco. Both STSs and the ioco relation have been introduced in Sect. 3.2.

The testing approach of Jambition is *random* and *on-the-fly*. This basically means that out of the set of specified input actions one input is chosen randomly, and then given to the service (i.e., an operation is invoked). Next, the returned



SSM Model: [✓ XML] UML
WSDL URL: useService/WarehouseService?wsdl
XMI File:
SSM URL: java/specifications/warehouse.xml
SSM:
Service: WarehouseService
Port: WarehousePort
Monitor

```
29/05/08 23:17:34:018: --> orderShipment?(adr(STSimulator,STSimulator),ref(81))
29/05/08 23:17:34:046: --> checkAvail?(r(foo@product,2))
29/05/08 23:17:34:100: <-- checkAvail!(return(12.3,foo@product,2,82,4))
29/05/08 23:17:34:110: --> orderShipment?(adr(STSimulator,STSimulator),ref(82))
29/05/08 23:17:34:817: --> checkAvail?(r(foo@product,3))
29/05/08 23:17:34:846: <-- checkAvail!(return(12.3,foo@product,3,83,4))
29/05/08 23:17:34:850: --> cancelTransact?(ref(83))
29/05/08 23:17:34:916: --> checkAvail?(r(bar@product,4))
29/05/08 23:17:34:964: <-- checkAvail!(return(12.3,bar@product,4,84,4))
29/05/08 23:19:09:169: --> cancelTransact?(ref(84))
29/05/08 23:19:09:260: --> checkAvail?(r(foo@product,5))
29/05/08 23:19:09:268: <-- checkAvail!(return(0.0,foo@product,0,0,2))
29/05/08 23:19:09:310: --> checkAvail?(r(bar@product,6))
29/05/08 23:19:09:340: <-- checkAvail!(return(12.3,bar@product,6,85,4))
```

Location Coverage: 60%
Switch Coverage: 56%

**Fig. 9.** The Jambition Testing Tool

message (if any) is received from the service. If that output message is not allowed by the STS, a failure is reported. Otherwise the next input is chosen – and so on. The *on-the-fly* approach differs from more classical testing techniques by not firstly generating a set of test cases, which are subsequently executed on the system. Instead, the test case generation, -execution, and -assessment happen in lockstep. So doing has the advantage of allaying the state space explosion problem faced by several conventional model-based testing techniques. The rationale here is that a test case developed beforehand has to consider all possible outputs the system might return, whereas the *on-the-fly* tester directly observes the specific output, and can guide the testing accordingly. Another cause of state space explosion is the transformation of symbolic models like STSs into semantic models like Labeled Transition Systems (LTSs). Several tools like TorX [37] and TGV [25] do this step to apply test algorithms which are defined on LTSs. Jambition also solves this issue by skipping this transformation step. Instead, its test algorithm, which is proven sound and complete for ioco, is dealing directly with the STS, see [15] for details. We have seen in Sect. 3.2 that a test tool has to perform three tasks:

1. construct input data which respects the given guard
2. select a concrete input value in case of multiple solutions
3. choose a transition in case of multiple options

To deal with the first task, JAMBITION consults the constraint solver of GNU Prolog [18] via a socket connection. That solver can compute solutions to constraints expressed over finite domain variables, which have a domain in the range [0..max_integer]. Since Web Services do not only deal with integer data, it would be quite restrictive to only allow integer message parameters. Fortunately, several types can be mapped to integers, so that constraint solving is still possible with them. In its current version, JAMBITION supports the *simple types* `Integer`, `Boolean`, `String`, and `Enumeration`. Furthermore, there is an experimental support for `Double` values having a fixed number of decimal places (to express for instance prices of products, as used in our example case study). Such "fake" doubles can also be mapped to integers. To express the STS transition guards, the most usual operators known from common programming languages can be used for integer- and boolean expressions. For enumerations and strings the only supported operator is (in)equality, see the manual for details [35]. Sometimes Web Services also deal with *complex types*, which store a sequence of data objects of arbitrary types, either simple or complex. Such a complex type is for instance used to represent `struct` data known from `C`, or objects of classes in OO languages. When explaining the warehouse service in Sect. 3.2 we have already seen three examples of complex types - *QuoteRequest*, *Quote*, and *Address*. JAMBITION also allows complex types, but not in a recursive manner, meaning that a complex type must not have a field of its own type. This excludes recursive types like *lists* and *trees*.

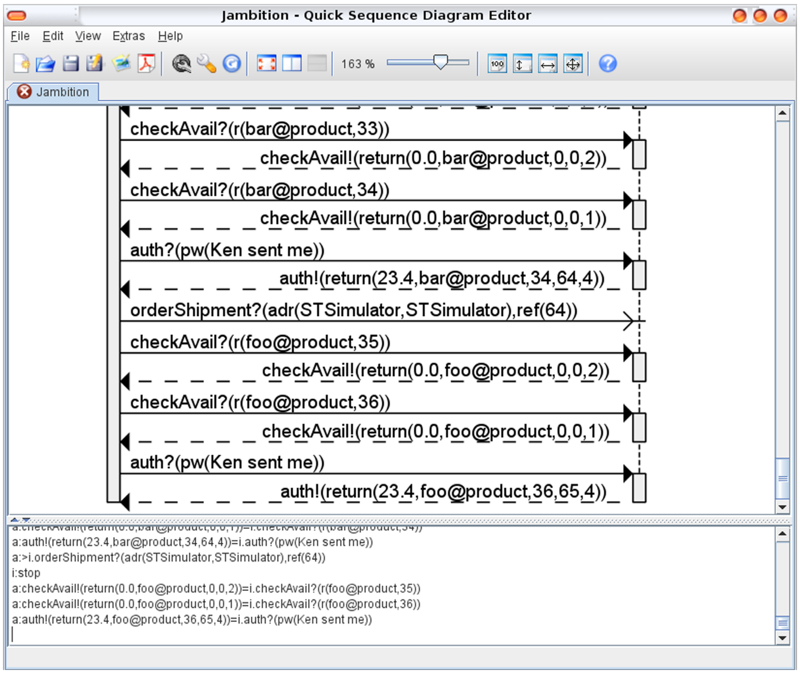
To deal with the second task, JAMBITION either selects a random value in case the input is not constrained by a guard. If it is constrained, four heuristics can be applied:

– **min:** choose the smallest solution
– **max:** choose the greatest solution
– **middle:** choose the solution in the middle
– **random:** choose a random solution

For instance, in Fig. 4 the transition from state `1` to state `2` requires `r.quantity > 0`. If we decide to choose the smallest solution, we get `r.quantity = 1`. But if `MAXQ` is greater `0`, always choosing `r.quantity = 1` will have the consequence that states `3` and `4` will never be reached. Thus, choosing a random solution is commonly good practice. To deal with the third task, a purely random choice is made. Being more sophisticated in these respects is one of our main future work goals.

To visualize the ongoing testing process, and to understand a reported failure, JAMBITION can display the messages exchanged with the service while being tested in real-time via the QUICK SEQUENCE DIAGRAM EDITOR [20], an external Java open-source visualizer for UML sequence diagrams. Figure 10 shows an excerpt of a sequence diagram representing the message exchange between JAMBITION and a warehouse service, being tested based on the STS specification from Fig. 4.

The left lifeline corresponds to JAMBITION, the right one to the warehouse service. The topmost message seen, sent from JAMBITION to the warehouse,


Jambition - Quick Sequence Diagram Editor
File Edit View Extras Help
163 %
Jambition

```
checkAvail?(r(bar@product,33))
checkAvail!(return(0.0,bar@product,0,0,2))
checkAvail?(r(bar@product,34))
checkAvail!(return(0.0,bar@product,0,0,1))
auth?(pw(Ken sent me))
auth!(return(23.4,bar@product,34,64,4))
orderShipment?(adr(STSimulator,STSimulator),ref(64))
checkAvail?(r(foo@product,35))
checkAvail!(return(0.0,foo@product,0,0,2))
checkAvail?(r(foo@product,36))
checkAvail!(return(0.0,foo@product,0,0,1))
auth?(pw(Ken sent me))
auth!(return(23.4,foo@product,36,65,4))
```

```
a:checkAvail!(return(0.0,bar@product,0,0,1))=i.checkAvail?(r(bar@product,34))
a:auth!(return(23.4,bar@product,34,64,4))=i.auth?(pw(Ken sent me))
a:>i.orderShipment?(adr(STSimulator,STSimulator),ref(64))
i:stop
a:checkAvail!(return(0.0,foo@product,0,0,2))=i.checkAvail?(r(foo@product,35))
a:checkAvail!(return(0.0,foo@product,0,0,1))=i.checkAvail?(r(foo@product,36))
a:auth!(return(23.4,foo@product,36,65,4))=i.auth?(pw(Ken sent me))
```

**Fig. 10.** The Quick Sequence Diagram Editor

invokes the `checkAvail` operation with the `QuoteRequest` object `r`. The `product` attribute of `r` is `bar`, and the `quantity` equals `33`. The corresponding STS transition goes from state `1` to state `2`. Since this is a *request-response* operation, the warehouse sends back a response message, depicted as the following return message (for technical reasons the returned `Quote` object is called `return`, not `q`, as in the STS). The last field of the offered quote is the `status` attribute, being `2`, which is the encoding of `SOLDOUT`. `Jambition` receives the return message and moves the STS back to state `1`. Now it again has to construct a quote request, this time it chooses a quantity of `34`, also for the `bar` product, and the returned `status` is `1`, which corresponds to `AUTHREQ` (transition from state `2` to `3`. Next, `Jambition` sends a password string ("`Ken sent me`") via the `auth` operation (transition from state `3` to `4`). Since the password string is not constrained, the probability that `Jambition` randomly chooses the right one is negligible. To still make it pick the right one once in a while, extra options can be set. The chosen password is correct in this case, and the warehouse returns a quote with `status` `4`, meaning `VALIDQUOTE`. Here `Jambition` has to check the guard on the transition from state `4` to state `5`, which is true, the warehouse behaves as specified: the `product` is `bar`, the `quantity` equals `34`, the `price` is `23.4 > 0.0`, and the `refNumber` is `64 > 0`. Finally, `Jambition` decides to order the shipment via the *oneway* operation `orderShipment`, depicted as an asynchronous message. And so on.

Furthermore, JAMBITION displays the achieved state- and transition coverage of the STS, see Fig. 9 (JAMBITION calls states *locations* and transitions *switches*).

## 4.2 PUPPET

In this section, we introduce PUPPET illustrating first the main characteristics of the approach and then the logical architecture of the implemented tool. The idea of PUPPET is general and could be applied to any instantiation of the SOA. However, the current implementation focuses on the Web Services technology.

In SOAs, services collectively interact to execute a unit of programming logic [2]. Service composition allows for the definition of complex applications at higher levels of abstractions. Nevertheless, since services are always part of a larger aggregation, their executions often rely on the interaction with other/external services.

In such a cooperating scenario, let us consider the example of a service provider who develops a composite service (i.e., the SUT), which is intended to interact with several other existing services (e.g. the supplier in the example at Sect. 3.1). In general, we can suppose that the service provider needs to test the implementation of the SUT, but he/she does not own or control the externally invoked services: for example interactions may have a cost that is not affordable for testing purposes, or the external services are being developed in parallel with the SUT.

The approach proposed by PUPPET is to automatically derive stubs for the externally accessed services `Si` from published functional and extra-functional specifications of the external services. PUPPET generates an environment (the services stubs) within which the composite service can be run and tested (see Fig. 11).

While various kinds of testbed can be generated according to the purposes of the validation activities, PUPPET aims specifically at providing a testbed for reliable estimation of the exposed QoS properties of the SUT. Concerning the externally accessed services, PUPPET is able to automatically derive stub services that expose a QoS behavior conforming to the extra-functional specifications such as agreements among the interacting services.

Once the QoS tested is generated, the service provider may test the SUT by deploying it on the real machine used at run-time. This would help in providing realistic QoS measures preventing the problem of recreating a fake deployment platform; in particular, the QoS evaluations will also take into account the other applications running on the same machine that compete for resources with the service under test (it is worth noting that handling this case would be extremely difficult using analytical techniques).

The stubs developed thus far include the set of operations they export and the emulation code for the extra-functional behaviors as specified in the WS-Agreement. Moreover, PUPPET includes a module to link each stub with code emulating the supposed functional behavior [5]. This module is optional, in the sense that is anyhow possible to skip it and still generate working stubs that only emulate the extra-functional behavior of the real services.

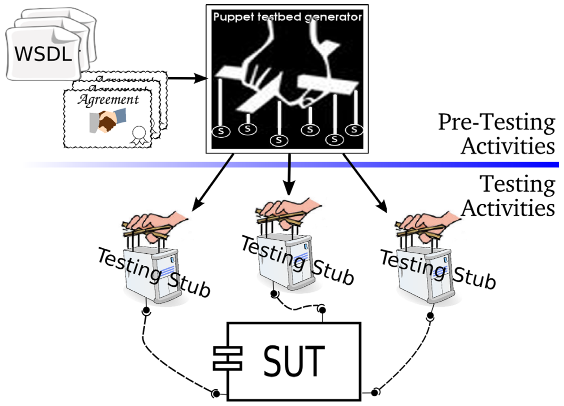


**Fig. 11.** General idea of PUPPET

The functional behavior of a service is modeled by means of the STS models as described in Sect. 3.2. PUPPET inserts into the stubs parametric code able to wrap an STS simulator we have developed [21]. The simulator simulates the STS according to the eco testing relation. Specifically, for each invocation to a service the stub can call the STS simulator package, choose one of the possible functionally correct results, and send it back to answer the service client request. The STS simulator can keep track of the symbolic states in which the STS can currently be. Thus, to supply the emulation of the functional behavior, PUPPET would demand that the external services carry on the STS specification corresponding to their provided interface.

In the following at Sect. 4.3 we will show an example on how the eco testing relation enhances the ability of the testbed in revealing extra-functional bugs in the SUT. Also, note that both the specification, and simulation of the STS are subject to the same restrictions as the ones given for the JAMBITION tool in Sect. 4.1 since the STS simulator is also used by JAMBITION as its underlying engine.

Latest works on PUPPET concerns the module that finally plugs into the obtained stubs the emulation of the mobility. The detailed description of this module is given in [6].

In the end, PUPPET generates service stubs that can be used by the testers in order to mock-up the deploying environment they would emulate. The architecture of PUPPET is structured in layered modules, whereby each module plays a specific role within the stub generation process. The detailed description of the architecture is reported in [5].

## 4.3 Combined Functional and Extra-functional Testing Mode

The two off-line approaches presented can be fruitfully combined, as functional testing can be influenced by extra-functional properties and vice-versa. We have previously discussed this combined approach in [5]. Below, we provide two examples extracted from [5], referring again to the customer, supplier, warehouse scenario introduced in Sect. 3.1.

**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.

As described in [5], a testbed that emulates the QoS features respecting also the functional protocols on the one hand it gives a more realistic model of the deployment environment; on the other hand it can reveal extra-functional leaks that can be closely related to functional values.

For example, 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 $40000 msec$. As defined in the agreement with the warehouse shown in Fig. 7, each interaction between the warehouse and the supplier service can take up to $25000 msec$. Take also into account, as described in Sect. 3.2, 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 (the arc from `state 2` to `state 3` in Fig. 4), together with the response to the provided password (the arc from `state 4` to `state 5` in Fig. 4), may in the worst case sum up to $50000 msec$, 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 interdependency, exist. Thus, in a mere extra-functional testbed this extra functional failure can easily be invisible.

**Detecting Functional Failures.** Similarly to what discussed above, in [5] it is shown how the extra-functional correctness of the stubs, can reveal further functional issues of the services under test.

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. 12. 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. 8. The supplier service is not prepared

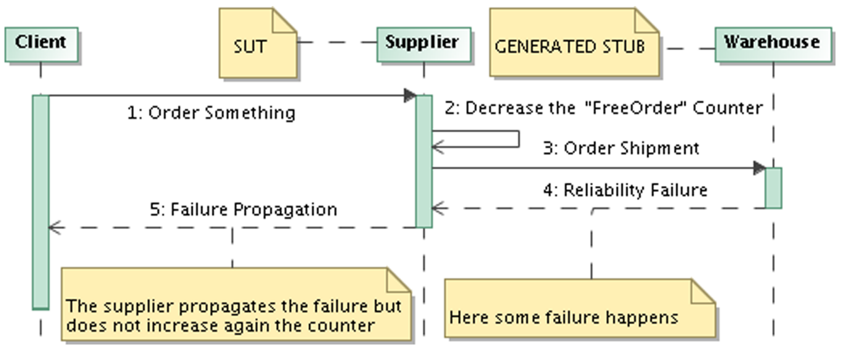

Client
SUT
Supplier
GENERATED STUB
Warehouse
1: Order Something
2: Decrease the "FreeOrder" Counter
3: Order Shipment
4: Reliability Failure
5: Failure Propagation
The supplier propagates the failure but does not increase again the counter
Here some failure happens

**Fig. 12.** Functional Fault Revealed by a Reliability Constraint

to deal functionally properties 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 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.

## 5 On-Line Testing Approaches

This section discuss on-line web-services testing strategies. Run-time testing can be a quite dangerous activity in particular when it involves stateful resources. Therefore in some cases run-time testing of services is not a valid option being monitoring a possible alternative solution for run-time verification. Obviously the drawback in this case is that observed fault are "real" i.e. they really exist on the running system. As a result monitoring approaches have to be combined with recovery strategies. In this paper we limit our discussion to testing strategies and in particular in this section with respect to the framework illustrated in Sect. 2 we discuss a suitable approach for the admission testing phase. The interested reader can refer to [4] for approaches to run-time monitoring.

### 5.1 The Audition Framework

The basic idea behind the audition framework is to test a service when it asks for registration within a directory service. Then in case the service fails to show the required behavior the registration in the directory is not granted. In this sense we called the framework "Audition", as if the service undergoes a monitored trial before being put "on stage".

It is worth noting that from a scientific point of view the implementation of the framework does not really introduce novel testing approaches. On the

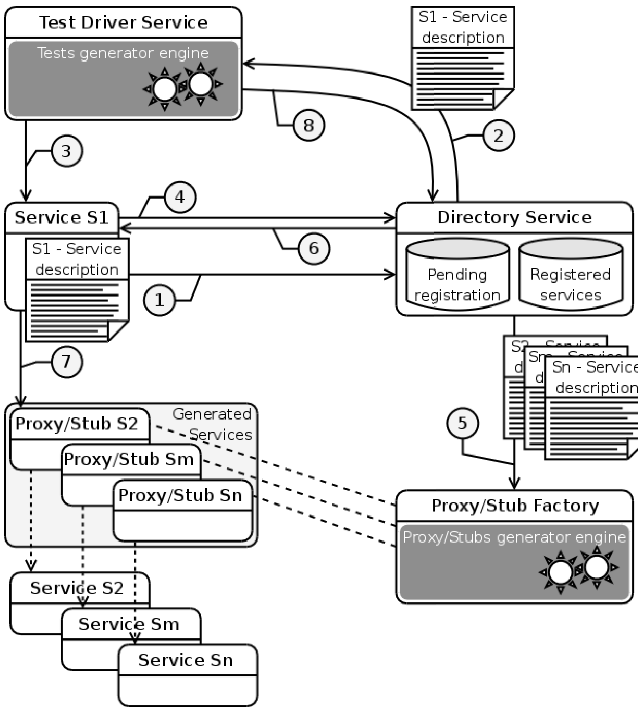


**Fig. 13.** The Audition Framework

contrary one of its target is just to reuse complex software tools (such as test generators) in a new context trying to take advantage of the new opportunities provided by the service oriented paradigm, such as for instance the existence of a “trusted party” corresponding to the service broker.

Nevertheless in order to automatically derive test cases for services asking for registration the framework requires the use of an increased service information model that should provide some description in a computer readable format of the service expected behavior. Such information model has to be provided to the service registry when a service asks for being included in the registry, and according to the framework the behavioral description has to be suitable for automatic test case derivation. Certainly this request has importance consequences on the applicability of the framework on a real setting. Nevertheless slightly different configuration of the framework can be derived for instance relaxing the part on automatic derivation of test cases with the usage of predefined and static test suites stored in the registry. This section will only discuss the framework when an automatic test generator is available. In particular Figure 13 shows the main elements of the framework. The figure intends to provide just a logical view, i.e., the arrows have not to be directly interpreted as invocations on methods provided by one of the elements. Instead they generally represent a logical step in the process and point to the elements that will take the responsibility of carrying on the associated task.

The process subsumed by the framework is activated by a request made by a service asking for being listed within the registry and is structured in eight main steps (the numbers in the list below correspond to the numbers in Figure 13):

1. a service S1 asks the registry (directory service in the figure) to be published among the services available to accept invocations. Contextually, S1 provides information concerning both the syntax (WSDL in the framework of the web service related technology) and a behavioral description of the offered service (expressing the protocol that a possible client should follow to correctly interact with the service). The behavioral description format has to be suitable for automatic test case generation.
2. the registry service stores S1 provided information marking them as "pending registration". At the same time S1 related information are sent to a Testing Driver Service (TDS). The provided behavioral description has to correspond to the one expected by the specific TDS. It is logically possible to accept different behavioral description for a service (contract based, automata based etc.); nevertheless for each accepted description a TDS able to automatically derive test cases from such a description has to be identified.
3. the TDS starts to make invocations on S1, acting as the driver of the test session, checking if the service behaves accordingly to the specification.
4. during the audition, unless S1 is a basic service, i.e. a service that does not require to cooperate with other services to fulfill its task, S1 will query the registry for references to other services necessary to complete the provision of its own service. Indeed S1 could use other services without asking to the registry since the references are hard coded in S1 definition. From the point of view of the framework in this case S1 is not different from a "basic service" since also at run-time it will continue to use the statically bound services.
5. the registry checks if the service asking for external references is in a pending state or not. If not, references for the required service description file and its relative access point are provided. Instead in case the service is in a pending state the registry provides the information, such as the interface and the behavioral description for the requested service to a Proxy/Stub Service factory. This Service starting from the syntactic and behavioral description is able to derive proxy or stubs for the requested service. In case of a proxy generation the generated service will implement the same interface of the "proxied" service, but at the same time it will check if the invocations made by S1 are in accordance to the ones defined in the specification and then expected by the invoked service. In case no errors are discovered the invocation is redirected to the real implementation of the service.

   In some cases a testing invocation to a running service may not be an option, since it would result in permanent effects on a stateful resource. In such cases, in order to completely implement the framework, the factory has to be able to generate service stubs. Obviously this will increase the complexity of the framework and asks for the provisioning of service description models

suitable for the automatic generation of service stubs. An STS specification could be for instance a model suitable for automatic stub derivation.
6. for each inquiry request made by S1 the registry service returns a binding reference to a Proxy/Stub version of the requested service.
7. on the base of the reference provided by the registry, S1 starts to make invocations on the Proxy/Stub versions of the required services in order to fulfill a request made by the test driver service. As a consequence the Proxy/Stub version of the service checks the content and the order of any invocation made by S1; In case a violation to the specification for the invoked service is detected, the Proxy/Stub informs the registry service that S1 is not suitable for being registered. As a consequence the directory service removes from the pending entries the service currently under test, and denies the registration;
8. finally in case one of the invocation made by the TDS results in the detection of an error the registry is informed. As for the previous case the registration will be denied.

**Considerations on the Framework.** The availability of a registry enhanced with testing capabilities, granting the registration only to "good" services, should reduce the risk of run-time failures and run-time interoperability mismatches. As described above, in our vision a service asking for registration will undergo two different kinds of check before being registered. The first concerns the ability of the service of behaving according to its specification and the second of being able to correctly interact with required services. Nevertheless some issues have to be considered in particular to derive a real implementation of the service and to better understand the applicability of the framework itself.

A first note concerns the reduced control over a service implementation by a third party such as the tester. In a SOA setting each organization has full control over the implementation of exposed services. This would mean that a service implementation could be changed by the organization to which it belongs, after its registration has been granted by the registry, and without informing the registry that otherwise would start another testing campaign. As a result a non-tested service will be considered as registered. Main consequence of this lack of control is that the framework can be fruitfully applied only within a semi-open environment, i.e. in an environment in which the participating organizations are known and interested in collaborating with the registry in order to guarantee that no "bad" services will enter the "stage".

Another relevant request posed by the framework concerns the fact that each interaction with the registry has to permit the identification of the sender. This constraint directly derives from the fact that the registry has to recognize the status of the registration for the invoking service when it asks for references to external services. At the same time it is worth mentioning that a service asking for registration has to know that it will undergo a testing session. Therefore during the testing session, and until the registration is not confirmed, the invocations should not lead to permanent effects.

A final interesting note concerns the automatic generation of stubs and proxies. Stubs intend to simulate the behavior of an invoked service. However the automatic generation of a service stub asks for the storing in the registry of a complex service model such as for instance the one discussed in Sect. 3.2. Indeed in case the registered service model does not permit the automatic derivation of a suitable stub the framework foresees the generation of proxy services instead of stub services. A proxy service will check and log incoming invocations with respect to the model and then it will redirect the invocation to a real implementation of the service in order to generate a meaningful answer. However this option is only acceptable in case the invoked service does not refer to a stateful resource; or otherwise in case the platform provides specific support for run-time testing purpose.

Next section shows a partial implementation of the framework that we derived within the Web Service domain.

## 5.2 WS-Guard

This section describes some relevant detail of a real implementation of an enhanced version of a UDDI registry able to apply the "audition phases". The resulting registry shows a standard UDDI web service interface and has been called WS-Guard(Guaranteeing Uddi Audition at Registration and Discovery) [11].

The first decision to take concerns the different technologies can be used for SOA. In developing WS-Guard we decided to use Apache related technologies and in particular the Axis2 SOAP container. This permitted us to easily derive, using the WSDL2Java tool, skeleton classes from the WSDL definitions for the two interfaces foreseen by the UDDI 2.0 specification : one for publishing services and the other for inquiring registered services.

Concerning the registry, we adopt an open source version of a UDDI registry that is provided by the Apache foundation under the project called jUDDI [13]. jUDDI consists of a set of servlets that are able to handle SOAP messages formatted according to the message format defined by the UDDI specification. To store the information related to registered services jUDDI requires to be linked to a suitable database, being MySQL one of the possible options (and the one we took).

Figure 14 describes the basic elements necessary to set-up a UDDI server using jUDDI.

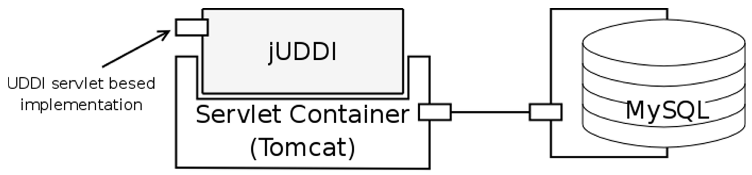


**Fig. 14.** jUDDI environment setting and technology

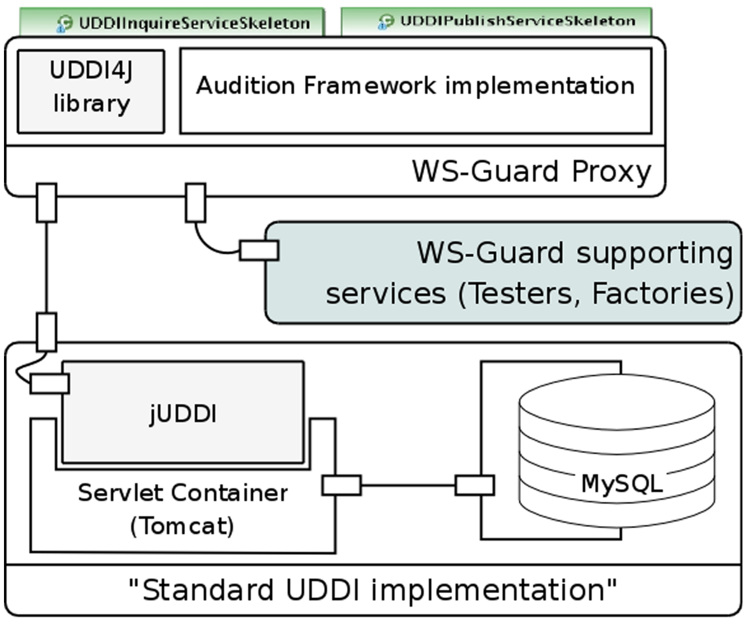

UDDIInquireServiceSkeleton
UDDIPublishServiceSkeleton
UDDI4J library
Audition Framework implementation
WS-Guard Proxy
WS-Guard supporting services (Testers, Factories)
jUDDI
Servlet Container (Tomcat)
MySQL
"Standard UDDI implementation"

**Fig. 15.** WS-GUARD service logical structure

Another tool that can be usefully adopted in this setting is UDDI4J [1]. This is a java based library providing an API that permits to directly interact with a remote servlet based implementation of the UDDI specification. As a result using UDDI4J it is possible to interact with a jUDDI server just making local calls to suitable objects derived from the UDDI4J library.

The availability of all these tools permitted to us to drastically reduce the required implementation effort, making also natural the choice of implementing our enhanced "UDDI server" as a Proxy service. This means that we were able to completely decouple the "audition enhancements" from a standard UDDI implementation. So within WS-Guard the functionalities required by the audition framework are enclosed in the skeletons derived by the UDDI WSDL specification using the WSDL2Java tool. At the same time the skeletons act as proxy implementation for a real UDDI implementation derived using jUDDI. Within the proxy the interactions with jUDDI are defined through the use of classes made available by the UDDI4J library.

Figure 15 shows the logical structure of the implementation we derived. The picture reports the main elements of the WS-Guard implementation and in particular it shows a supporting services element whose implementation is detailed in the next subsection.

## 5.3 WS-Guard Usage: Issues and Solutions

In this section we illustrate the various implementation choices we took to develop the WS-Guard registry and how the various issues posed by the framework have been solved. For the sake of presentation we illustrate the various steps reusing the scenario highlighted in Sect. 3.1 assuming that a supplier service provider wants to register its service on a audition enhanced directory service.

**Modeling and Testing Web Services.** WS-Guard requires that a service asking for registration provides a specification of its external behavior (also a reference to an URL from which such specification can be retrieved). The formalism we have adopted for such step is the one described in Sect. 3.2. In order to store such information we had to change the data structure used by UDDI and the WSDL-UDDI mapping in order to be able to associate a WSDL description to an STS specification. So with reference to the Supplier service it is required that the service provides a reference to a suitable representation of the STS shown in Figure 5. Such model will be then stored in the registry.

Having services represented as STSs, we could exploit the JAmbition "engine" illustrated in 4.1 in order to implement a tester service. So JAmbition has been wrapped within a web service that is then invoked by our WS-Guard proxy service after receiving a publish request. WS-Guard provides the tester with the endpoint of the service to be tested and the corresponding STS representation.

**Web Services Identification.** The Audition framework foresees the ability of recognizing if a service performing an inquiry corresponds to a service under test. In such a case the registry returns a reference to a proxy version of the requested service instead of a direct reference.

To uniquely identify a Web service we decided to choose the service endpoint reference as an identifier. To gather the endpoint reference, we mandate the usage of messages compliant to the WS-Addressing specification [39] when interacting with the registry. Indeed this specification provides a means to the receiver to identify the service endpoint from which a Web service message request comes from. Moreover Axis 2 provides mechanisms to support communications and message generation according to this standard.

**Pending Data Structures.** A Java HashSet data structure is used to store identification information of services under test (pending state). The data structure is maintained in memory in order to make faster the check that have to be carried on for each incoming inquiry request in order to verify the status of the sender. All the other service related information are directly inserted in the database and marked as pending. This is necessary in order to avoid that the service endpoint is returned as result of inquiry made by already registered services. So with reference to the example, the supplier service reference will not be returned to any other service until it is marked as pending. All the entries related to a service under audition will be deleted in case the service does not overtake the audition phase.

**Discovery of Services and Generation of Proxies.** One key point of the Audition process is the replies to inquires made by services in a pending state. Going to our example this is the case of the supplier that needs to access to retrieve and access to warehouse services. According to the framework in this case WS-GUARD must returns, to the supplier service, a reference to a proxy version of the warehouse and not a direct access point. Moreover the proxy service should be able to identify wrong calls made by the supplier on the warehouse. To do this we automatically generate proxies that can check received invocations against the STS defined for the requested service (warehouse in our case). In case of an error the registry is then informed.

Finally to cope with stateful services WS-Guard make the assumption that services are deployed in two copies. One copy is the regular service accessible at the endpoint `http://www.mycompany.com/service` and that have been registered in the registry. The second copy is made available only for testing purpose at `http://www.mycompany.com/service_test`. In order to avoid dangerous usage of stateful services WS-Guard generate proxies that only interact with services at endpoints with extension "`_test`".

## 6 Conclusions

This chapter has overviewed several issues and recent results in the field of SOA testing, with a special focus on the PLASTIC project, which is also discussed in Chapter 1 [24].

We have presented several new testing methods facing the exigencies of flexibility posed by SOAs. Such flexibility is mainly expressed in terms of the dynamic interactions among "black-box" software, provided by different organizations. We have also addressed the evaluation of extra-functional properties, which are of outmost importance in pervasive SOAs.

Summarizing, the described PLASTIC framework spans over the whole service life-cycle, covering with a coherent set of tools both off-line and on-line stages, and addressing both functional and QoS concerns. Given the broad variety of PLASTIC applications, the validation framework is not conceived as a fixed methodology, but rather as a set of techniques/tools that can be used alternatively, or in combination, depending on the constraints/requirements on each considered application/scenario.

Although verification and validation of SOA is a very active research topic, as shown by the many works surveyed in Chapter 4 [9], solutions that can be found in the literature generally address a specific limited objective. The concerted effort for service validation in PLASTIC provided the opportunity for developing a consistent methodology which is unique in terms of comprehensiveness and flexibility. The platform integrates several approaches that can be applied during the whole service life-cycle: after being developed, when published on a new environment, and during the actual live usage.

In particular: Jambition, Puppet, and Weevil allow service developers to rigorously test a service (using the original STS model) before deployment in a realistic reproduction of the deployment context (as opposed to testing in the real environment or to manually mocking it). SlangMon and Dynamo-AOP support monitoring against the defined properties with an improved efficiency with respect to existing solutions, and directly deriving the monitor from the SLA contracts.

Further work on experimentation on realistic testbeds and on real applications is required. At the current stage, a set of prototype tools have been released and are publicly available for download, but their usage on PLASTIC industrial case studies is still ongoing. Usage of the tools requires some adaptation/modeling effort and a major open issue is the lack of realistic testbeds on which to perform experiments that can provide realistic validation.

## Acknowledgement

This paper reports about the work carried on in the 32 months European STREP IST-26955 PLASTIC. All the partners in the project have in different measures and ways inspired our ideas and collaborated with us. The PLASTIC validation framework has been mainly developed within Workpackage 4, and we would like to thank the many colleagues who contributed to it, and in particular Domenico Bianculli, Franco Raimondi, Antonino Sabetta and Alexander Wolf. Section 2 of this paper is an excerpt of joint discussions and writings for the WP4 deliverables. Lars Frantzen is further supported by the Marie Curie Network TAROT (MRTN-CT-2004-505121) and by the Netherlands Organization for Scientific Research (NWO) under project STRESS.

## References

1. UDDI4J (accessed on June 3rd, 2008), `http://uddi4j.sourceforge.net/`
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services–Concepts, Architectures and Applications. Springer, Heidelberg (2004)
3. Basha, S.J., Irani, R.: AXIS: the next generation of Java SOAP. Wrox Press (2002)
4. Bertolino, A., Bianculli, D., De Angelis, G., Frantzen, L., Kiss, Z.G., Ghezzi, C., Polini, A., Raimondi, F., Sabetta, A., Toffetti Carughi, G., Wolf, A.: Test Framework: Assessment and Revision. Technical Report Deliverable D4.3, PLASTIC Consortium. IST STREP Project (May 2008)
5. Bertolino, A., De Angelis, G., Frantzen, L., Polini, A.: Model-based Generation of Testbeds for Web Services. In: Proc. of the 20th IFIP Int. Conference on Testing of Communicating Systems (TESTCOM 2008). LNCS. Springer, Heidelberg (2008)
6. Bertolino, A., De Angelis, G., Lonetti, L., Sabetta, A.: Let The Puppets Move! Automated Testbed Generation for Service-oriented Mobile Applications. In: Proc. of the 34rd EUROMICRO CONFERENCE on Software Engineering and Advanced Applications. IEEE, Los Alamitos (2008)
7. Bertolino, A., Marchetti, E.: A brief essay on software testing. In: Thayer, R.H., Christensen, M.J. (eds.) Software Engineering, 3rd edn. Development process, vol. 1, pp. 393–411. Wiley-IEEE Computer Society Press (2005)

8. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
9. Canfora, G., Di Penta, M.: Service Oriented Architectures Testing: A Survey. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006–2008, University of Salerno, Italy. LNCS, vol. 5413, pp. 78–105. Springer, Heidelberg (2009)
10. Christensen, E., et al.: Web Service Definition Language (WSDL) ver. 1.1 (2001), `http://www.w3.org/TR/wsdl/`
11. Ciotti, F.: Ws-guard - enhancing uddi registries with on-line testing capabilities. Master's thesis, Department of Computer Science, University of Pisa (April 2007)
12. PLASTIC european project homepage, `http://www.ist-plastic.org`
13. Apache Foundation. JUDDI (accessed on June 3rd, 2008), `http://ws.apache.org/juddi/`
14. Frantzen, L., Tretmans, J.: Model-Based Testing of Environmental Conformance of Components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 1–25. Springer, Heidelberg (2007)
15. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)
16. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
17. Global Grid Forum. Web Services Agreement Specification (WS–Agreement), version 2005/09 edn. (September 2005)
18. GNU Prolog homepage, `http://www.gprolog.org/`
19. MagicDraw homepage, `http://www.magicdraw.com`
20. Quick Sequence Diagram Editor homepage, `http://sdedit.sourceforge.net/`
21. STSimulator homepage, `http://www.cs.ru.nl/~lf/tools/stsimulator/`
22. Huhns, M.N., Singh, M.P.: Service-Oriented Computing: Key Concepts and Principles. IEEE Internet Computing 9(1), 75–81 (2005)
23. IBM. WSLA: Web Service Level Agreements, version: 1.0 revision: wsla-2003/01/28 edn. (2003)
24. Inverardi, P., Tivoli, M.: The future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006–2008, University of Salerno, Italy. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009)
25. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: IDPT 2002. Society for Design and Process Science (2002)
26. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. Proceedings of the IEEE 84, 1090–1126 (1996)
27. NetBeans Tutorial on Asynchronous JAX-WS Web Service Client End-to-End Scenario, `http://www.netbeans.org/kb/55/websvc-jax-ws-asynch.html`
28. OASIS consortium. Universal Description, Discovery, and Integration (UDDI) (accessed on June 3rd, 2008), `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec`
29. Object Management Group. UML 2.0 Superstructure Specification, ptc/03-08-02 edn. Adopted Specification
30. Sahner, R.A., Trivedi, K.S., Puliafito, A.: Performance and Reliability Analysis of Computer Systems An Example-Based Approach Using the SHARPE Software Package. Kluwer Academic Publishers, Dordrecht (1995)

31. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
32. Skene, J., Lamanna, D.D., Emmerich, W.: Precise Service Level Agreements. In: Proc. of ICSE 2004, pp. 179–188. IEEE Computer Society Press, Los Alamitos (2004)
33. Skene, J., Skene, A., Crampton, J., Emmerich, W.: The Monitorability of Service-Level Agreements for Application-Service Provision. In: Proc. of WOSP 2007, pp. 3–14. ACM Press, New York (2007)
34. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
35. PLASTIC tools homepage, `http://plastic.isti.cnr.it/wiki/tools`
36. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
37. Tretmans, J., Brinksma, E.: TorX : Automated Model Based Testing. In: Hartman, A., Dussa-Zieger, K. (eds.) First European Conference on Model-Driven Software Engineering, December 11-12 2003, Imbuss, Möhrendorf, Germany (2003)
38. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional Testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
39. W3C. WS-Addressing (accessed on June 3rd, 2008), `http://www.w3.org/Submission/ws-addressing/`