PLASTIC

IST STREP Project

Deliverable D4.3

Test Framework:
Assessment and Revision

http://www.ist-plastic.org

INRIA SIEMENS Telefónica UCL UNIVERSITÀ DELLA SVIZZERA ITALIANA VIRTUAL TRIP Ltd Internet Technologies PRAGMA CONSULTORES

Project Number	:	IST-26955
Project Title	:	PLASTIC
Deliverable Type	:	Prototype, Report

Deliverable Number	:	D4.3
Title of Deliverable	:	Test Framework:
Assessment and Revision	:	
Nature of Deliverable	:	Prototype,Report
Dissemination Level	:	Public
Internal Document Number	:	D4.3V.0.3
Contractual Delivery Date	:	29 February 2008
Actual Delivery Date	:	28 February 2008
Contributing WPs	:	WP4
Editor(s)	:	Antonia Bertolino
Author(s)	:	Antonia Bertolino, Domenico Bianculli, Guglielmo De Angelis, Lars Frantzen, Zsolt Gere Kiss, Carlo Ghezzi, Andrea Polini, Franco Raimondi, Antonino Sabetta, Giovanni Toffetti Carughi, Alexander Wolf
Reviewer(s)	:	Paola Inverardi

Abstract

This document is Deliverable D4.3 of PLASTIC Work Package 4 (WP4), titled: *Test Framework: Assessment and Revision*. It provides a detailed description of the most up-to-date version of the tools developed within WP4 along with guidelines for installing and using them.

A first version of the tools with a preliminary accompanying description had been firstly released in Deliverable D4.2: *Test Framework: Prototype Implementation*, which can be considered superseded by this Deliverable. We refer throughout to WP4 Deliverable D4.1: *Test Framework Specification and Architecture*, which provided a comprehensive specification of the proposed PLASTIC validation framework, with state-of-the-art overview and justification for the adopted techniques. Although some of the basic principles underlying the framework are summoned up for completeness, the present deliverable must anyhow be considered together with D4.1.

The PLASTIC validation framework is organised around two main phases, respectively called *off-line* and *on-line*. Off-line validation concerns validation at development time. In this phase services are tested in a fake/simulated environment that reproduces functional and/or extra-functional run-time conditions. The tools made available for this stage are:

1. JAMBITION (see Chapter 2): this is a model-based testing tool that allows to automatically derive and execute invocation sequences on a service, checking whether the responses conform to a given specification, expressed as a Service State Machine (SSM). Jambition is based on a model based testing engine called Ambition , defined in D4.1, of which Jambition is the Java front-end. To facilitate the usage of Jambition, a library called MINERVA (see Chapter 3) has been embedded into the tool. Minerva permits to model SSMs via an UML modelling tool.

2. PUPPET (see Chapter 4): this is a tool to automatically generate stubs implementing external services invoked by the service under development. The mock services generated by Puppet exhibit a correct behaviour with respect to given non-functional properties. At the same time Puppet can generate stubs making invocations on the service under evaluation according to certain workload profiles.

3. WEEVIL (see Chapter 5): it consists of a synthetic-workload generator coupled with an environment for managing the deployment and execution of experiments. Weevil is intended to facilitate experimentation activities for distributed systems by providing engineers with a flexible, configurable, automated and, thus, repeatable process for evaluating their software on a networked testbed.

On-line validation foresees testing of a service when it is ready for deployment and final usage. In particular, the PLASTIC validation framework supports validation during *Live Usage*, i.e., service behaviours are observed during real execution to reveal possible deviations from the expected behaviour. Also on-line validation can cover both functional and extra-functional properties. Tools developed to support this phase are:

1. DYNAMO-AOP (see Chapter 6): it is a framework for monitoring functional properties of external services which a BPEL process interacts with, to realize a composite service.

2. SLANGMON (see Chapter 7): it permits to dynamically detect violations of non-functional properties specified in SLAng (the PLASTIC language to specify service level specifications and service level agreements, developed within Work Package 2). Events related to the non-functional characteristics are logged and possibly used to redeem controversy.

Keyword List

On-line testing, Off-line testing, Model-Based Testing, QoS testing, Monitoring, Testbed Harness

Document History

Version	Type of change	Author(s)
0.1	Proposed Outline	CNR (with all)
0.2	First Complete Release for internal review	CNR (with all)
0.3	Final Version	CNR (with all)
0.4	Final Version, some typos adjusted, added section on novelty in Intro chapter(AB)	CNR (with all)

Contents

List of Figures ... 9

List of Tables ... 11

1 Introduction ... 12

1.1 PLASTIC validation stages ... 12
1.2 Testing challenges and opportunities ... 13
1.3 Development-time testing ... 14
1.4 Admission testing ... 15
1.5 Live-usage verification ... 15
1.6 PLASTIC validation framework novelty ... 16
1.7 Tool download ... 16

2 Jambition ... 17

2.1 Jambition Overview ... 17
2.2 Technical info ... 18
2.3 Deployment ... 19
2.3.1 Install ... 20
2.3.2 Configure ... 21
2.4 Tutorial ... 22
2.4.1 Web Service Description Language ... 22
2.4.2 Service State Machines ... 24
2.4.3 The Warehouse Example ... 28
2.5 Appendix ... 37
2.5.1 The Dumont Grammar in BNF ... 37

3 Minerva ... 39

3.1 Minerva Overview ... 39
3.2 Technical info ... 39
3.3 Deployment ... 39
3.3.1 Install ... 39
3.3.2 Configure/Usage ... 40
3.4 Tutorial ... 40
3.4.1 Creating the project ... 40
3.4.2 Modeling the types and data structures ... 41
3.4.3 Creating the service description ... 44

3.4.4 Creating the Service State Machine ... 46
3.4.5 Location variables ... 49
3.4.6 Exporting the SSM ... 50

4 Puppet ... 51

4.1 Puppet Overview ... 51
4.2 Technical info ... 51
4.3 Deployment ... 52
4.3.1 Install ... 52
4.3.2 Configure ... 53
4.3.3 Usage ... 54
4.4 Tutorial ... 54
4.4.1 Terms in the Agreement and Generation Process ... 54
4.4.2 The Syntax for the Terms in the WS-Agreement Contracts ... 54
4.4.3 Writing an Agreement ... 56
4.4.4 Functional Behavior with Jambition ... 59
4.4.5 Example ... 63
4.5 Appendix ... 67
4.5.1 WS Agreement ... 67

5 Weevil ... 75

5.1 Weevil Overview ... 75
5.2 Technical info ... 75
5.3 Deployment ... 76
5.3.1 Install ... 76
5.3.2 Configure ... 76
5.4 Tutorial ... 78
5.4.1 Workload ... 78
5.4.2 Experiment ... 80

6 DynamoAOP ... 82

6.1 DynamoAOP overview ... 82
6.2 Technical info ... 82
6.3 Deployment ... 83
6.3.1 Install ... 83
6.3.2 Configure/Usage ... 84
6.4 Tutorial ... 84
6.4.1 WS-CoL ... 84

6.4.2 Demo . . . 87
6.5 Appendix . . . 88
6.5.1 WS-CoL grammar . . . 88
6.5.2 Architecture . . . 90

7 SLAngMon . . . 92
7.1 SLAngMon Overview . . . 92
7.2 Technical info . . . 92
7.3 Deployment . . . 93
7.3.1 Install . . . 93
7.4 Tutorial . . . 94
7.4.1 Demo . . . 96
7.5 Appendix . . . 99
7.5.1 Structure of the source code . . . 99
7.5.2 FAQ . . . 99
7.5.3 An SLA in SLAng (XMI representation) . . . 99

8 Conclusions and ongoing improvements . . . 101

Bibliography . . . 103

List of Figures

Figure 1.1: PLASTIC Validation Framework 12

Figure 1.2: PLASTIC Testing stages 13

Figure 2.1: Jambition Concepts 19

Figure 2.2: Jambition Inputs 20

Figure 2.3: Jambition Initial Screenshot 21

Figure 2.4: Jambition Preferences Screenshot 22

Figure 2.5: The Warehouse SSM 33

Figure 3.1: Creating a new project 40

Figure 3.2: The basic types and data structures 41

Figure 3.3: Creating an enumeration 42

Figure 3.4: Applying a stereotype 43

Figure 3.5: The warehouse types diagram 44

Figure 3.6: The service 44

Figure 3.7: The service diagram 45

Figure 3.8: The service state machine diagram 46

Figure 3.9: Setting initial state 47

Figure 3.10: Selecting an operation 48

Figure 3.11: Editing a guard 48

Figure 3.12: The SSM diagram in MagicDraw 49

Figure 3.13: Location variables in the SSM 50

Figure 4.1: Expressions in the Qualifying Condition 55

Figure 4.2: AND in the Qualifying Condition 55

Figure 4.3: OR in the Qualifying Condition ... 56

Figure 4.4: NOT in the Qualifying Condition ... 56

Figure 4.5: Operators in the Qualifying Condition ... 57

Figure 4.6: Extra-Functional Properties in the Service Level Objective ... 58

Figure 4.7: Defining the Scope of a Term ... 58

Figure 4.8: Scenario 3 Deployment Diagram ... 64

Figure 4.9: Scenario 3 Sequence Diagram ... 66

Figure 5.1: Weevil Experimentation Process ... 77

Figure 5.2: An example of workload ... 78

Figure 5.3: Simulation-Based Workload Generation ... 79

Figure 5.4: Workload Scenario Conceptual Model ... 80

Figure 6.1: Process `PizzaDeliveryCompany` deployed successfully.. ... 87

Figure 6.2: The result of inserting two monitoring rules. ... 88

Figure 6.3: Output console of the monitored process ... 88

Figure 6.4: Monitoring rules modified with the “Dynamo Supervision Manager” ... 89

Figure 6.5: Dynamo-AOP components architecture ... 91

Figure 7.1: Plugin screenshot for SLAngMon: generation of checkers. ... 95

Figure 7.2: Service description diagram for the eHealth scenario ... 96

Figure 7.3: Use case service description diagram for the eHealth scenario ... 96

Figure 7.4: Latency clause for e-Health Provider and Patient ... 97

Figure 7.5: Excerpts from the automatically generated Java code for the Axis handler ... 98

List of Tables

Table 4.1: Enabling the code generation in PUPPET 54

Table 4.2: QoS Properties 67

1 Introduction

The PLASTIC project aims at enabling the development and deployment of mobile adaptable robust services for Beyond 3G (B3G) networks, by providing a comprehensive platform integrating both adequate software methodologies and tools, and the supporting middleware. Within PLASTIC, the goal of Work Package 4 is to investigate, develop, and integrate with the PLASTIC platform a suitable validation technology for the B3G computing domain.

We recall in this chapter the main concepts of the conceived validation framework and then in the following chapters we provide in detail a description of the tools that have been developed. In particular this chapter shortly illustrates the novel aspects of the proposed approaches and the key reasons why each of them is suitable for testing in the B3G and service-oriented domain.

1.1 PLASTIC validation stages

An overall picture of the PLASTIC Validation Framework is illustrated below in Figure 1.1.

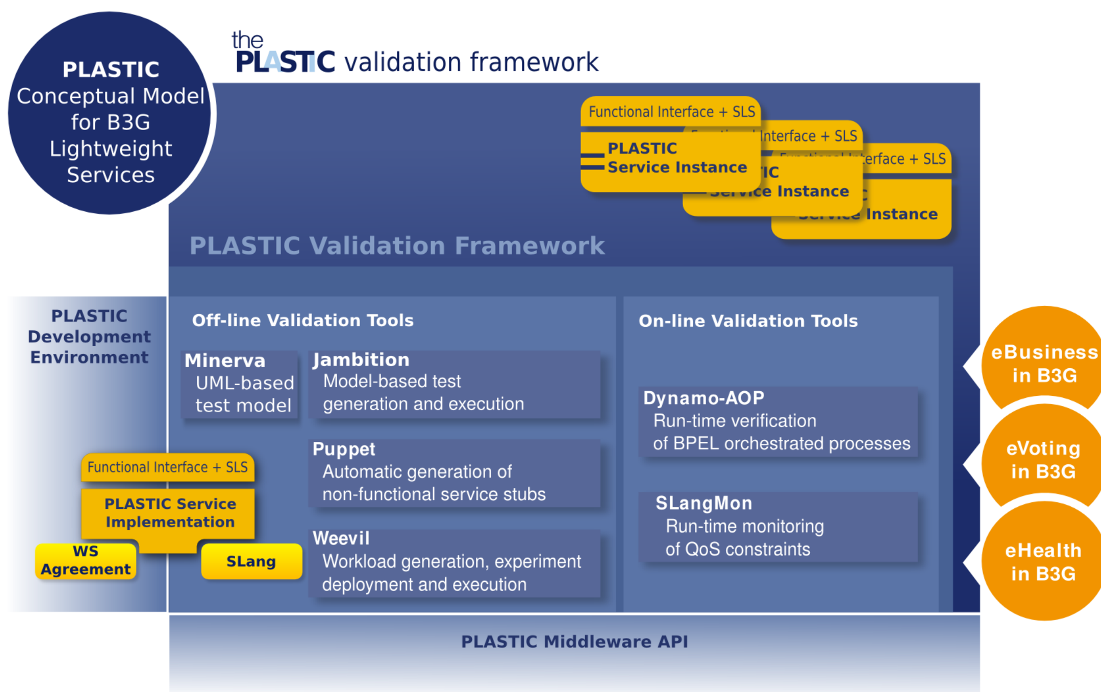

Figure 1.1: PLASTIC Validation Framework

An in-depth discussion of the state-of-the-art and of the approaches proposed has been given in Deliverable D4.1, produced in the first year. We have decomposed the problem into more easily solvable subproblems, by identifying a set of distinct validation stages and by developing a set of testing techniques for each stage. Figure 1.2 below illustrates the different testing stages that make up the PLASTIC testing framework.

Development-time testing refers to activities that are carried out by service developers before the services are deployed; this is the more conventional testing stage. On-line testing refers to monitoring and testing performed after deployment, which is further subdivided between testing at admission stage, and (passive) testing during live-usage.

In the second year we have then developed and deployed the tools supporting the specified approaches. In particular, in Deliverable D4.2 (released in Month 18) we provided a first release of

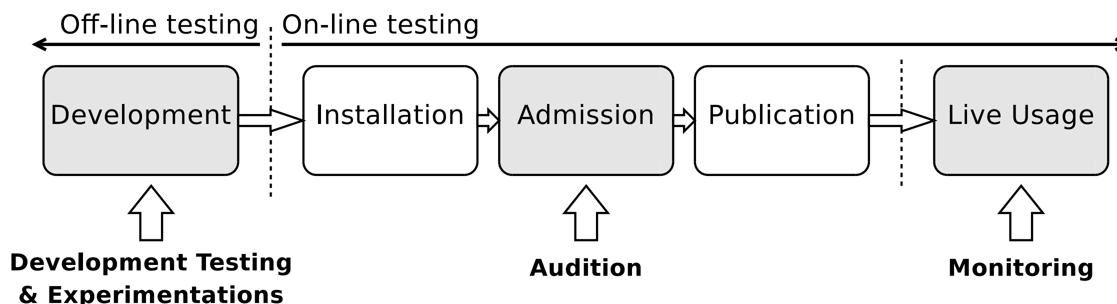

Figure 1.2: PLASTIC Testing stages

the tools implemented, with a preliminary description of their offered functionality. In the present document we release a more mature version of the validation framework, with assessment and revision of the deployed testing tools.

The implementation of the validation framework follows the first year review, in which a general recommendation to the whole PLASTIC consortium consisted into avoiding ”spreading project resources too thinly over too many ambitious sub-areas” and considering to ”keep focus and ensure thorough depths in the work performed”. To address such recommendation, we have critically revisited the D4.1 methodology specification, and have focused the implementation effort of the second year towards the most promising approaches (from feedback with the Consortium). From the approaches originally proposed in D4.1, we have thus dropped from further revision the Ws-Guard tool, implementing the Audition approach, and we have also dropped from further consideration alternative approaches to on-line testing which had been originally investigated in D4.1 (OSLAC) as well as alternative approach to off-line testing (simulation-based). Nevertheless, as a quite preliminary version of Ws-Guard tool had already been developed before the review, we deemed it useful to still make it publicly available in D4.2 repository for interested researchers.

The next sections provide some detail on the techniques developed within PLASTIC, according to the strategy shown in Figure 1.2, highlighting how the challenges are tackled and the opportunities are exploited.

1.2 Testing challenges and opportunities

As said, service-oriented systems and in particular software developed to be deployed over B3G network presents new challenges to software testers. PLASTIC targets more in particular services developed for mobile devices. In deliverable D4.1 the challenges faced by testers in such domain have been extensively discussed, nevertheless it can be useful to recall here briefly some of them.

The first challenge that testers have to address concerns the fact that in general service integration does not become apparent until run-time. According to the SOA paradigm, services can discover each other in dynamic way and they can select the partner to interact with based on parameters that are only defined at run-time. Therefore, it is difficult to anticipate at testing time which services will interact with a given service under development. This is particularly true for mobile devices. Besides, even if a service could be statically bound to an external service, the lifecycle of the latter will generally under the control of a different organisation, who could change its implementation without notice.

Another important issue in the B3G and service-oriented domain is the relevance of extra-functional characteristics, which are central to the PLASTIC approach. Service clients have no control on the services they use and on the underlying platform, but at the same time require that certain requirements on the service extra-functional properties are met. This raises the neces-

sity of languages to define agreements among parties (this issue in PLASTIC is being tackled in WorkPackage 2). Also, developers need techniques and approaches to evaluate extra-functional properties of systems built using web-services before the real system is actually deployed. Finally it is necessary to develop mechanisms to check whether the agreements are fulfilled at run-time.

Besides the new challenges that must be faced within this new development paradigm, new opportunities can be envisaged as well.

An extraordinary opportunity is the general increased availability of service specification in a machine-readable format. At the current state of art, only the specification of syntax pertaining to the services undergo standardised approaches, see e.g. the adoption of WSDL specifications for Web Services. Nevertheless, many researchers suggest that syntactical published information should be complemented with behavioural specifications. As a natural consequence, such specifications could be fruitfully applied in order to derive test cases automatically. We apply such methodology, for example, by using the SSM specifications to derive functional test cases (see the tool Jambition).

Finally the introduction of another intermediate phase in the software lifecycle, i.e., the *registration* phase, augments the opportunities for testing by providing the possibility of an additional testing phase in the whole validation strategy, which is what we outlined with the Audition [8] stage.

1.3 Development-time testing

As remarked in the previous section, in a B3G service-oriented setting, where services are discovered and integrated only at run-time, it is difficult – if at all possible – to establish in advance appropriate guarantees on the behaviour of composite services. 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.

Moreover, 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 it is possible to use existing real services for testing purposes. Conversely, if an invoked service accesses stateful resources, this option must be ruled out and the required services have to be simulated. Within PLASTIC the problem of reproducing predictable run-time environment is addressed providing two different tools, Puppet and Weevil, which allow the developers to reproduce different live usage scenarios.

In particular Puppet can be used to to automatically derive the elements necessary to recreate a predictable "live" environment that is suitable for the evaluation of extra-functional properties. Puppet allows testers to generate automatically the required services in such a way that they show a "correct extra-functional behaviour" with respect to a given specification. Chapter 4 discusses how to install and use Puppet.

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. The usage of Weevil is illustrated in Chapter 5.

Finally a testing tool targeting design-time functional testing of PLASTIC applications, called Jambition, is used to automatically derive test suites for services under development. The concrete usage of Jambition is illustrated in Chapter 2. The key idea of this tool is to exploit as much as possible the behavioural description often available for deployed services. The extreme dynamicity of the service domain suggested to augment service with operational specification in order to characterize services in a richer way. Jambition exploits this kind of information in order to derive test cases that are suitable for service evaluation. Jambition assumes that such specifications are

available as Service State Machine (SSM), for which a sound theoretical foundation is also being developed. We understand, though, that they could sound unfamiliar and difficult for practitioners. However, SSMs can be seen as a formal semantics for a variant of UML 2.0 state machines [19]. In integrated way with the development environment produced by WP2, we have then also developed an automatic mapping of the output generated by the MagicDraw UML modeling tool [18] to an XML format describing a corresponding SSM. The mapping is performed by the Jambition embedded library Minerva which is described in Chapter 3. Thus, a developer can use this visual tool to model the functionality of service interfaces in the common formalism of UML 2.0 state machines.

1.4 Admission testing

As discussed in Section 1.2 one of the main challenges of testing is to test the service in the environment where it will operate at run-time. At the same time, the Service-Oriented Architecture (SOA) foresees the existence of a service broker that is used by services to search and obtain references to each other. The idea of the Admission testing is to have the service undergo a preliminary testing stage (also referred to as audition) whose results will decide the actual registration of the service in the directory.

The intuition of the Admission testing is that the quality of registered services can be increased by granting the registration into the directory 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 auditioning a service, suitable countermeasures must be taken.

Following the recommendations coming from the first year review, plans for the subsequent period of experimentation of the PLASTIC platform do not include however admission testing. D4.2 includes however the release of a directory service, called WS-Guard, conforming the UDDI specification that implements the Audition idea and permits to test services before their registration.

1.5 Live-usage verification

Difficulties in applying verification techniques before live usage, suggested to extend the verification phase till run-time. Within workpackge 4 two different activities, aiming at the development of monitoring mechanisms, have been activated. The idea is to add suitable mechanisms to the platform so as to detect violations with respect to the expected behaviour of services.

The first of these approaches, called Dynamo-AOP, focuses on functional behaviour of orchestrated services, and provides support to augment orchestrating services with checks, in order to verify that the orchestrated services behave as expected. Chapter 6 describes how to install and use Dynamo-AOP.

Another approach in this category supports the monitoring and logging of extra-functional properties for running services. SLAngMon implements a mechanism to parse Service Level Agreement specifications defined in SLAng automatically generates the code of efficient checkers, whose operation is based on timed automata theory. Chapter 7 is devoted to the description of how to set and use SLAngMon.

1.6 PLASTIC validation framework novelty

Summarising, the described PLASTIC framework spans over the whole service lifecycle, covering with a coherent set of tools both off-line and on-line stages, and addressing both functional and QoS concerns. Although verification and validation of SOA is a very active research topic, solutions that can be found in the literature address a specific limited objective. The WP4 concerted effort for service validation in PLASTIC provided the opportunity for developing a consistent matrix methodology which is unique in terms of comprehensiveness and flexibility. The contribution does not consist only in the combination of different techniques though: Jambition, Puppet, and Weevil, allow service developers to rigorously test a service (using the original SSM 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 improving in efficiency with respect to existing solutions and directly deriving the monitor from the SLA contracts.

1.7 Tool download

All the prototypes described in this document can be freely downloaded from:

http://plastic.isti.cnr.it

2 Jambition

2.1 Jambition Overview

Every Web Service provides a set of operations to its potential users. To know which operations are available, an interface specification is needed. Commonly, Web Service interfaces are specified in the *Web Service Description Language (WSDL)*.

For instance, a Web Service representing a warehouse may offer operations to check the availability of products, and to order such products. In the first case, an object representing a quote request is sent, and an object represeting a quote is returned. The corresponding WSDL file makes the signature of these operations public. This information is sufficient to connect to the Web Service, and to invoke the operations, but it does not give any kind of semantic information. For instance, the warehouse may only accept quote-requests of a certain quantity. Or it may only allow to order products when the availability has been requested beforehand. Or it may guarantee that every offered quote deals with the same quantity as the requested quote has indicated. WSDL files are not intended to provide such kind of information.

One natural way to extend a Web Service description in this direction is to use state machines. Jambition uses a dedicated variant of state machines which is especially useful for Model-Based Testing – *Service State Machines (SSM)*. Such a state machine can be used to express constrains on the data as it is passed via the operations, and it gives a legal ordering of the invocations of operations. Hence, properties like the ones stated above for the warehouse can be expressed via SSMs.

Such an SSM model can suit several needs. For instance, it gives a specification of the dynamic aspects of the Web Service invocations. A user of the Web Service knows which operations are allowed to be invoked at what point in time. She also knows the restrictions on the data to be sent and received. SSMs are a valuable means to extend Web Service specifications.

Furthermore, an SSM can be used to automatically test a Web Service. This is what Jambition does. It takes a WSDL and a SSM specification of a Web Service as input. Based on these it fully automatically generates invocations to the Web Service, receives the returned messages, and checks if this data is conforming to the SSM specification.

Following the warehouse example, Jambition will respect the protocol as it is encoded in the SSM, for instance that it will always check the availability before making an order. It will also respect the data constraints, for instance making orders with a quantity of at least 3 products. With respect to testing, Jambition will receive the responses of the operation calls it makes, and then check if also the warehouse respects its constraints on the data. For instance, Jambition will check if the warehouse makes only offers for the requested products.

If Jambition spots a failure it will report so and stop the testing. While no failure is found, it does a random walk through the SSM, meaning that when there are several inputs specified, it chooses one randomly. With respect to the parameters of operations, Jambition only chooses data values which respect the constraints (like a quantity $\geq$ 3). Usually the first solution of the constraint is chosen (like quantity $=$ 3).

To visualize and to keep track of how the test proceeds, Jambition offers a GUI which monitors the testing events. It is also possible to log the ongoing testing to rotating log files. Furthermore, Jambition can be connected to the open source tool `Quick Sequence Diagram Editor`, which displays the communication between Jambition and the Web Service in real time as an UML sequence diagram.

The execution of Jambition on a Web Service corresponds to the automatic generation and execution of thousands of test cases within minutes. The only thing needed in addition to the WSDL

is the SSM specification. Having that, Jambition can be a great help in testing the functionality of Web Services.

Note, that as a result of recent efforts, the testing engine of Jambition has been integrated into the Puppet tool. Please refer to chapter 4 and section 8 for further details.

2.2 Technical info

Provider CNR

Introduction Jambition is a prototype tool to automatically test the functionality of stateful Web Services based on Service State Machine (SSM) specifications. SSMs are similar to UML state machines, but tailored to the domain of Web Services. They have a precise semantics and are especially suited to perform Model-Based Testing. SSMs specify a single port type. The Web Service to be tested is assumed to be passive, meaning that only `request-response` and `one-way` operations are defined in the corresponding WSDL. Jambition supports a subset of the XML-Schema data types.

Development status Version 301107 is available for download

Intended audience Developers of stateful services

License GPLv3 (open source) with some exceptions to include libraries

Language Java 6, GNU Prolog 1.3.0

Environment (set-up) Hardware: no special requirements
Software: Java 6 Runtime Environment, treeSolver, dot(optional), Quick Sequence Diagram Editor (optional)

Platform Java 6 Runtime Environment. treeSolver is available for Linux and Windows. Other operating systems are supported by recompiling. Ideally, the Web Service to be tested should be deployed on a Glassfish server. Other applications servers may demand minor adaptions of Jambition.

Download The official version can be retrieved at http://plastic.isti.cnr.it/download/tools

Tasks SSMs can in principle also be used to model the communication between several Web Services. To do so they have to involve the message flow at several ports. Testing based on such multi-port SSM would allow to test more complex scenarios like coordinated and composed Web Services.

Many Web Services use lists as data types. Lists correspond to an unbounded sequence in XML, which is not supported by the current version. Supporting lists is another eligible future step.

With respect to the testing itself, the purely random approach might not always be satisfactory. Specific coverage criteria can be conceived.

The constraint solving can be more flexible and general. For instance, dealing with negative integers can be achieved by changing the underlying constraint solver. Also, a “first solution found” approach is sometimes too restrictive.

Bugs N/A

Patches N/A

Contact Lars Frantzen<lars.frantzen@isti.cnr.it>

2.3 Deployment

The relevant concepts of Jambition are depicted in Fig. 2.1. The structural aspects (data types,

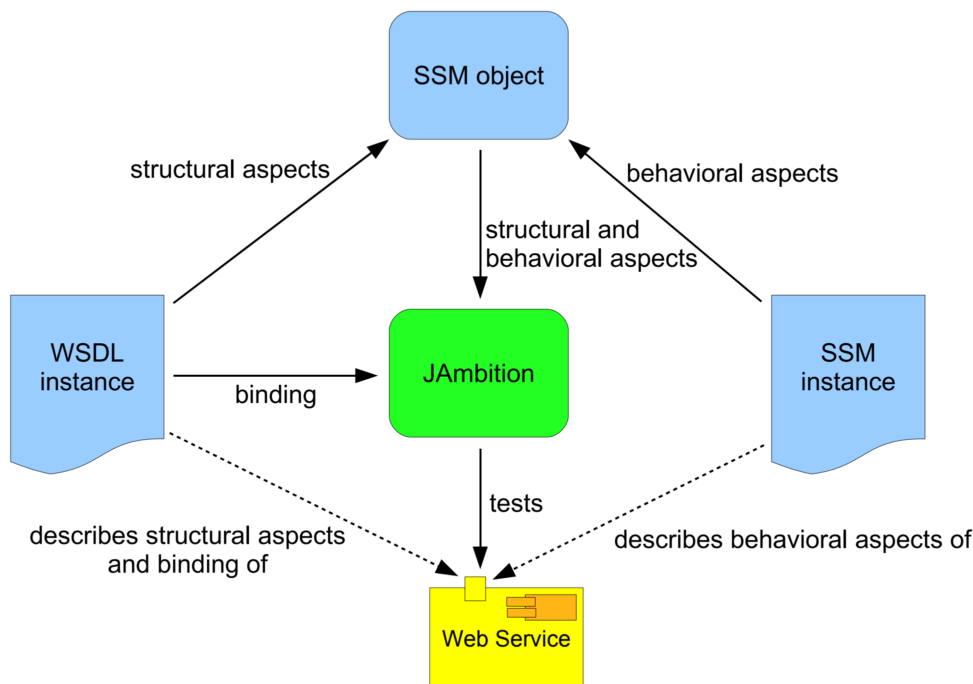

Figure 2.1: Jambition Concepts

messages, operations, port types) of the Web Service to be tested are described in a WSDL instance. The behavioral aspects (states of the Web Service, ordering of invocations, constraints on the data flow) are described in an SSM instance. Both aspects are combined together in an SSM object. This object is the main reference of Jambition to do the automatic testing. Furthermore, the binding information of the WSDL file (service, port) is used to physically connect to the Web Service.
Concretely, Jambition needs four inputs, see Fig. 2.2.

1. The URL of the WSDL instance
2. The service name of the Web Service to be tested as being given in the WSDL instance (since a single WSDL can define several services)
3. The port name of the Web Service to be tested as being given in the WSDL instance (since a single WSDL can define several ports per service)
4. The URL of the SSM instance

Based on these inputs, the SSM object is generated and the testing can be started.

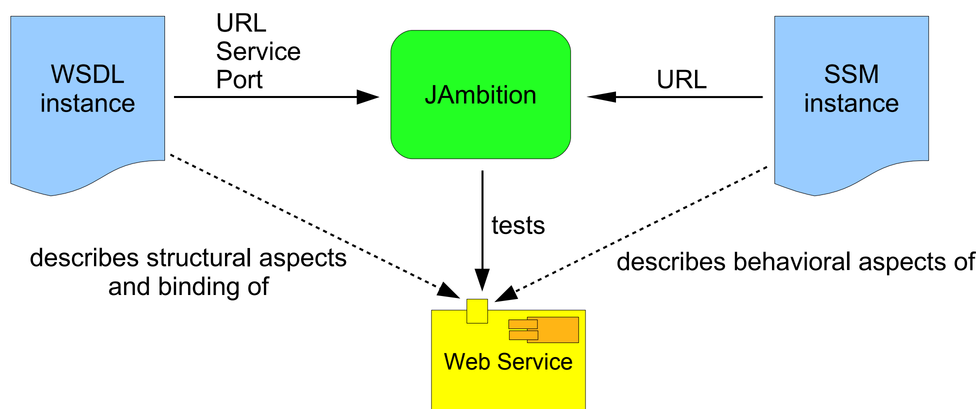

Figure 2.2: Jambition Inputs

2.3.1 Install

Jambition is written in Java 6, hence it needs a Java 6 JRE installed. It comes as a `Jambition.zip` archive. Within the archive there is the `Jambition` directory. Just extract this directory to a desired place. Within the directory there is the `Jambition.jar` archive. A simple double click should start Jambition on all common operating systems.

Jambition needs access to a constraint solver to do the testing. It uses GNU Prolog for this purpose. It is necessary to install and run the `treeSolver` program, which opens a socket connection to the constraint solver. `treeSolver` can be freely downloaded here:

http://www.cs.ru.nl/~lf/tools/treesolver/

The web page provides executables for Windows and Linux. Also the source code is available, so it can be compiled for several other operating systems, see http://www.gprolog.org/#platform. The `treesolver` gets as only input the socket-port to open. Before starting Jambition, invoke `treeSolver` for instance with the port number `60002`. You should get something like:

```
user@pc-plastic:~$ treeSolver 60002
treeSolver V180607
By Lars Frantzen (lars@frantzen.info)
A socket interface to the constraint solver of GNU Prolog
GNU Prolog is copyright (C) Daniel Diaz
Listening on pc-plastic port 60002
```

A tool which is not mandatory for Jambition, but useful to visualize the testing process, is the `Quick Sequence Diagram Editor - sdedit`. It is written in Java and can be freely downloaded here:

sdedit.sourceforge.net/

After `sdedit` is launched, turn on its real-time server by choosing in the `File` menu `Start/stop RT server`. For the socket-port accept the suggestion `60001`.

Another tool which is not mandatory for Jambition, but useful to visualize the SSM, is `dot`. It is part of the `Graphviz` toolsuite. Binaries for Windows and Linux can be freely downloaded here:

http://www.graphviz.org/

Now Jambition can be launched, see the screenshot in Fig. 2.3. The first thing to tell Jambition is where to find the WSDL and SSM specifications of the Web Service to test. The following sections will give further details about these files.

Figure 2.3: Jambition Initial Screenshot

2.3.2 Configure

Via the `Jambition` menu the `Preferences` window can be displayed, see the screenshot in Fig. 2.4. We explain the options from top to bottom.

At the top the socket of the `treeSolver` can be modified. The default values here are port `60002` on the local host. The option `Always New Inputs` is experimental, it means that whenever Jambition has to find a solution for a constraint, it tries to find a new solution. For instance, the constraint $\texttt{quantity} \geq 3$ will always be solved by choosing $\texttt{quantity} = 3$, since this is the first solution found. This may not always be desirable, turning this option on will find the solutions 3,4,5,et cetera. But this generates very huge and inefficient constraints, so this option should be treated with care.

Next the `sdedit` socket can be modified. The `Enabled` option allows to turn on and off the usage of `sdedit`. This means that Jambition does not send the testing messages to the `sdedit` tool, which can nicely display those in form of a sequence diagram. Still, it is possible to generate the input for `sdedit` later, for instance when a failure was found. To do so it is necessary to turn

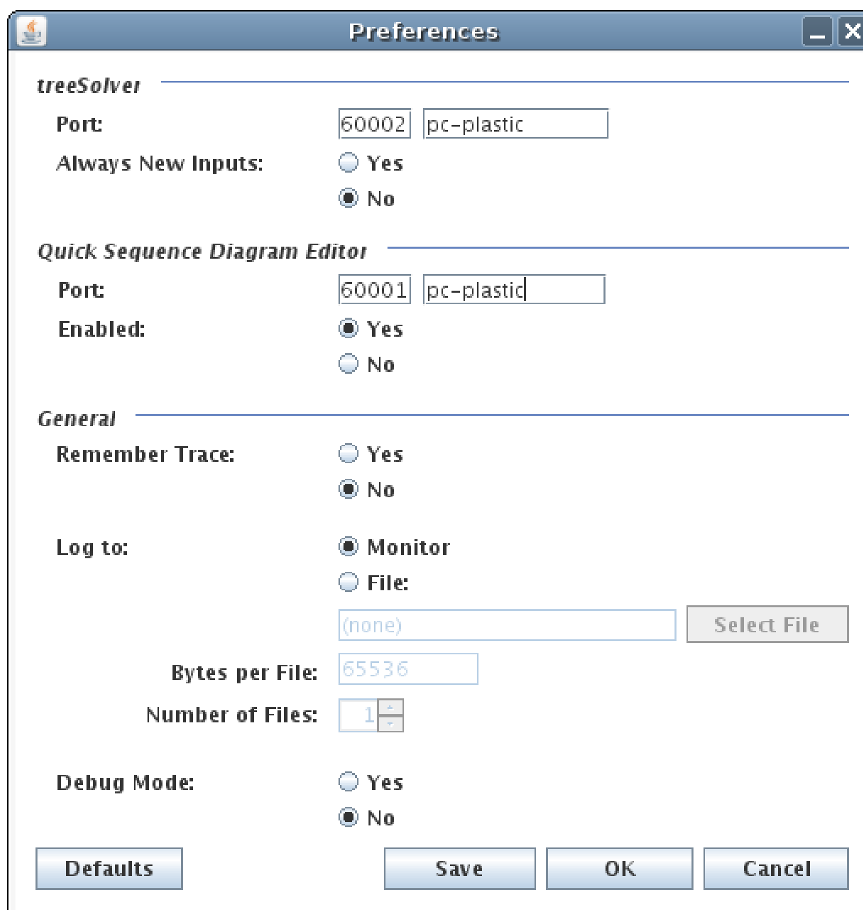

Figure 2.4: Jambition Preferences Screenshot

on the `Remember Trace` option.

Jambition logs per default to its `Monitor` window. This can be turned off. Note that logging to the `Monitor` increasingly consumes memory since the window remembers all test events. Instead, or additionally, it is possible to log to rotating log files. The name of the file, the bytes per file, and the number of files can be set.

Finally, the `Debug Mode` can be turned on or off. This means that additional information is logged, for instance the exchanged SOAP messages. All options can be reset to its default values, and saved.

2.4 Tutorial

2.4.1 Web Service Description Language

The structural aspects (data types, messages, operations, port types) of the Web Service to be tested are described in a WSDL instance. Also the binding information is encoded here. Normally, WSDL files are automatically generated by the IDE which is used to develop the Web Service.

To define the data types of the message parameters, commonly the data types of XML Schema are used in the WSDL.

2.4.1.1 XML Schema Simple Types

Jambition supports the following simple types:

- xs:boolean
- xs:integer
- xs:string
- xs:enumeration (a restriction on xs:string)
- xs:double

Two limitations are important to note. Firstly, Jambition can only deal with positive integers. This is due to the fact that GNU Prolog (which is used by the `treeSolver`) can only solve constraints over positive integers.

Secondly, the support for doubles is very limited. This is due to the fact that constraint solving over double variables is not feasible in the way Jambition needs it. What Jambition does is supporting double variables with a fixed precision. For instance, doubles with a fixed precision of two positions after the decimal point can be used to model amounts of money. Such "fake" doubles can be mapped internally to integers. But this mapping easily leads to large integers, which may be too large again for the constraint solver. In its current version Jambition assumes a fixed precision of two positions after the decimal point.

2.4.1.2 XML Schema Complex Types

Complex types are used to represent classes of Object-Oriented languages in the WSDL. For instance, given a Java method which returns an object of class c. The class c has two fields. If this method is exposed as a Web Service method, the corresponding WSDL can define a complex type for c which is a sequence of two elements representing the two fields. Jambition supports such complex types.

Jambition does not support elements in a sequence which are of the kind `maxOccurs = "unbounded"`. Such elements are used to model, for instance, lists.

2.4.1.3 WSDL Operations

Four kinds of operations can be defined in a WSDL:

- request-response
- one-way
- solicit-response
- notification

Jambition does only support the first two kinds — request-response and one-way. We call such a service a passive service since it does never send messages actively, only after being requested.

2.4.1.4 SOAP Binding

The XML parameter mapping is defined by the `soap:body use` and the `soap:binding style` attributes. Jambition assumes that:

- `use = "encoded"` or
- `use = "literal"` and `style = "document"` and each message has exactly one part called `parameters`.

2.4.2 Service State Machines

The behavioral aspects (states of the Web Service, ordering of invocations, constraints on the data flow) are described in an SSM instance. An SSM is a state machine, consisting of typed variables, states, and transitions between the states. Every transition consists of these elements:

1. The state where the transition starts.
2. The name of a WSDL operation.
3. The kind of the corresponding message. This can be either `input` or `output` for request-response operations, and is always `input` for one-way operations.
4. A guard which restricts the conditions under which the transition can be executed. The language of the guard is described below.
5. An update of the variables.

Every SSM has an initial state.

2.4.2.1 The Language of the Guards

There are five kind of *literals*, i.e. constants of a certain type:

- **Integer literal** – a numberstring (e.g. 3423432)
- **Boolean literal** – `true` or `false`
- **Double literal** – numberstring followed by a dot '.' and optional another numberstring (e.g. 23.2324)
- **String literal** – string between double quotes, e.g., "Hello World"
- **Enumeration literal** – a string of the form *element@enumname*, where *enumname* is the name of an enumeration type, and *element* one of its elements, e.g. `BOOK@product`

An *identifier* is a name of a variable. If a variable has a complex type, then one can refer to the fields by the dot-notation, e.g. `q.product`. Every literal and identifier are expressions which have a corresponding type, i.e. one of integer, boolean, float, string, complex or enumeration. One can combine literals and identifiers via *operations*, yielding more complex expressions, which again have a type. Next we mention the operations currently supported by the parser. By *number* we mean integers or doubles. Note that here by writing for instance *boolean* or *number* we refer to any expression of this type.

First we mention the operations which lead to an expression of type boolean:

- `==` compares booleans, numbers, strings and enumeration instances for equality
- `!=` compares booleans, numbers, strings and enumeration instances for inequality
- `<` compares numbers for being *less than*
- `<=` compares numbers for being *less than or equal*
- `>` compares numbers for being *greater than*
- `>=` compares numbers for being *greater than or equal*
- `&&` the logical *and* of two booleans
- `||` the logical *or* of two booleans
- `!` the logical *not* of a boolean

Next we mention the operations which lead to expressions of type integer or double:

- `++` increments (+1) an integer
- `--` decrements (-1) an integer
- `+` adds numbers or strings (i.e., a concatenation of two strings)
- `-` subtracts two numbers
- `*` multiplies two numbers
- `%` The remainder operator for numbers
- `/` divides two numbers

A guard is an expression of type boolean. Sometimes one wants to express an empty guard, meaning an expression which has the logical value *always true*. This cannot be done by giving an empty string, nor by writing just the boolean literal `true`, one has to write `true == true` here. One also cannot just mention the name of a variable of type boolean like `p` or `!p`, it is necessary to write `p == true` or `p == false`.

2.4.2.2 Variable Updates

There is another operator we have not mentioned yet, it is the assignment operator '='. Here we assign a value to a variable. An update can consist of several such assignments, but one can also have no assignment at all by giving the empty string “”, meaning that all variables remain unchanged. All assignments must be terminated by a semicolon `;`.

2.4.2.3 Designing SSM in MagicDraw

Please refer to the Minerva chapter 3.

2.4.2.4 Defining SSM directly in XML

Jambition expects an instance of an XML schema for SSM called `SP-SSM`. After giving the the full schema it is further explained below.

An XML Schema for SSM

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace=
            "http://frantzen.info/testing/ssmsimulator/schema/"
            xmlns:tns="http://frantzen.info/testing/ssmsimulator/schema/"
            elementFormDefault="qualified">

    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A schema for Singleport Symbolic State Machines (SP-SSM).
            Version: 010807
            Copyright (C) 2007 Lars Frantzen
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="spssm" type="tns:SPssm"></xsd:element>
    <xsd:complexType name="SPssm">
        <xsd:sequence>
            <xsd:element name="wsdlURI" type="xsd:anyURI"/>
            <xsd:element name="porttype" type="xsd:token"/>
            <xsd:element name="states">
                <xsd:complexType>
                    <xsd:sequence maxOccurs="unbounded">
                        <xsd:element name="state" type="xsd:token"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="initialState" type="xsd:token"/>
            <xsd:element name="variables">
                <xsd:complexType>
                    <xsd:sequence maxOccurs="unbounded">
                        <xsd:element name="var" type="tns:Variable"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="switches">
                <xsd:complexType>
                    <xsd:sequence maxOccurs="unbounded">
                        <xsd:element name="switch" type="tns:Switch"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Variable">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:token"/>
            <xsd:element name="type" type="xsd:token"/>
        </xsd:sequence>
```

```
    </xsd:complexType>
    <xsd:complexType name="Switch">
        <xsd:sequence maxOccurs="1">
            <xsd:element name="from" type="xsd:token"/>
            <xsd:element name="operationName" type="xsd:token"/>
            <xsd:element name="kind" type="tns:Kind"/>
            <xsd:element name="restriction" type="xsd:string"/>
            <xsd:element name="update" type="xsd:string"/>
            <xsd:element name="to" type="xsd:token"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="Kind">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="input"/>
            <xsd:enumeration value="output"/>
            <xsd:enumeration value="unobservable"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Explanation of the Schema
We describe its elements:

- <wsdlURI> Every SP-SSM instance specifies a port-type which is given in an external WSDL file. This external file is referenced by this tag via an URI. For Jambition, this tag has no relevance.
- <porttype> Here the specific port-type from the referenced WSDL is named (there could be several in the WSDL). For Jambition, this tag has no relevance.
- <states> Here all the state-names are listed.
- <initialState> The name of the initial state is given here.
- <variables> Here all the variables are listed.
  - <name> Every variable has a name.
  - <type> Every variable has a type. This is either one of the XML schema simple types like `xs:boolean`, or the name of a complex type or an enumeration. In case of a complex type or an enumeration the namespace indicator must be the same as the target namespace of the referenced WSDL.
- <switches> Here the SSM transitions, called switches, are given.
  - <from> This is the state-name where the switch starts.
  - <operationName> This is the name of the operation (as defined in the WSDL) where the switch-message belongs to.
  - <kind> This is he kind of the message which is transported by this switch. For every operation this can be either `input` or `output`. A special third kind is `unobservable` which denotes an internal message which is not observable at the interface.
  - <restriction> This is guard.

– `<update>` This is the variable update.
– `<to>` This is the state-name where the switch leads to.

Note that especially when dealing with `<restriction>` and `<update>` elements there may be the need to map special characters like '&' or '<' to a different representation in the schema instances. The next section gives an example instance of this schema.

2.4.3 The Warehouse Example

In this section a simple example is given which demonstrates the usage of Jambition. We follow here again the warehouse scenario. We have seen that Jambition needs as input the WSDL file, and an SSM specification, of the Web Service to be tested. Since the WSDL file is usually generated automatically by some programming language specific tool, we start this example by showing the warehouse operations as Java methods. Note that for Jambition it does not matter how the WSDL was generated, by a Java specific tool, another language's tool, or manually.

2.4.3.1 The Warehouse Web Service in Java

We use here the JAX-WS [1] implementation to handle Web Services in Java. We assume the familiarity of the reader with the JAX-WS (or a similar) package, and will not explain its usage further.

The warehouse service shall offer three operations, which appear in Java as methods:

- `Quote checkAvail(QuoteRequest r)`
  This operation allows a user of the warehouse to request a quote for a given product and quantity. This is realized via a parameter of type `QuoteRequest` having the attributes `Product` and `quantity`. We assume the class `Product` to be a simple enumeration, and `quantity` to be an integer. As a return value, this operation shall send a `Quote` object, which consists also of a `Product` and `quantity`; additionally it has the attributes `price` of type double, and `refNumber` of type integer.

- `void cancelTransact(int ref)`
  This operation shall be invoked if the user rejects a quote offered. To do so, the user sends as a parameter the reference number which has been issued in a corresponding quote (see attribute `refNumber` of the `Quote` object in operation `checkAvail` above).

- `void orderShipment(int ref, Address adr)`
  This operation shall be invoked if the user accepts a quote offered. To do so, the user sends also here as a parameter the reference number which has been issued in a corresponding quote, and a shipment `Address` object.

Next we show the signatures of the corresponding Java methods:

```
@WebMethod
public Quote checkAvail(@WebParam(name = "r") QuoteRequest r)

@WebMethod
@Oneway
public void cancelTransact(@WebParam(name = "ref") int ref)
```

[1] Available at https://jax-ws.dev.java.net/.

```
@WebMethod
@Oneway
public void orderShipment(@WebParam(name = "ref") int ref,
                          @WebParam(name = "adr") Address adr)
```

The auxiliary classes and their attributes are as follows:

```
public class QuoteRequest {
    Product product;
    int quantity;
    ...
}

public enum Product {
    foo,
    bar
}

public class Quote {
    Product product;
    int quantity;
    double price;
    int refNumber;
    ...
}

public class Address {
    String firstName;
    String lastName;
    ...
}
```

2.4.3.2 The Warehouse WSDL

We assume here that the reader has a basic understanding of WSDL files and XML Schema. We have created the Java-files in a package called `services`. Next we generate the corresponding WSDL for the warehouse Web Service, which is one of the input files of Jambition. Important for the further modeling of the service are the operation names, and the names of the operation parameters as they appear in the WSDL. The operations names usually match exactly the Java method names. The parameter names should also match the parameter names given in the Java code (in case of JAX-WS this name can be explicitly set via the `@WebParam` annotation, see above). A special case is the return parameter. In Java this parameter has no name on its own, just a type. For instance, the method `checkAvail` returns a `Quote` object, but this return parameter has no extra name. In the WSDL, also these returned parameters have names.

Let us check the WSDL generated by JAX-WS. We only give the relevant parts, here. First we check the definition of the messages and operations. In a WSDL, every operation consists of messages. Input parameters are modeled by input messages, and the returned parameter is modeled via an output message. In case of a Java `void` operation just an input message is present. In the WSDL this looks like this:

```
  <message name="checkAvail">
```

```
    <part name="parameters" element="tns:checkAvail"/>
  </message>
  <message name="checkAvailResponse">
    <part name="parameters" element="tns:checkAvailResponse"/>
  </message>
  <message name="cancelTransact">
    <part name="parameters" element="tns:cancelTransact"/>
  </message>
  <message name="orderShipment">
    <part name="parameters" element="tns:orderShipment"/>
  </message>

  <portType name="Warehouse">
    <operation name="checkAvail">
      <input message="tns:checkAvail"/>
      <output message="tns:checkAvailResponse"/>
    </operation>
    <operation name="cancelTransact">
      <input message="tns:cancelTransact"/>
    </operation>
    <operation name="orderShipment">
      <input message="tns:orderShipment"/>
    </operation>
  </portType>
```

We see here two things. First of all, the names of the operations equal the names of the Java methods (the `<operation name=XYZ">` attribute). Secondly, the definition of the message parameters refers to XSD elements (e.g. `tns:checkAvail`). To find these elements, we next check the definition of the types:

```
  <types>
    <xsd:schema>
      <xsd:import namespace="http://services/"
      schemaLocation="http://localhost:8080/Fuppet/WarehouseService/...
      WEB-INF/wsdl/WarehouseService_schema1.xsd"
      [...]
    </xsd:schema>
  </types>
```

Here we see that the WSDL refers to an external XML Schema file which defines the types (the `schemaLocation`). We check that file, next:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://services/"
    xmlns:tns="http://services/" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="cancelTransact" type="tns:cancelTransact"/>
  <xs:element name="checkAvail" type="tns:checkAvail"/>
  <xs:element name="checkAvailResponse" type="tns:checkAvailResponse"/>
  <xs:element name="orderShipment" type="tns:orderShipment"/>
```

```
<xs:complexType name="checkAvail">
  <xs:sequence>
    <xs:element name="r" type="tns:quoteRequest" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="checkAvailResponse">
  <xs:sequence>
    <xs:element name="return" type="tns:quote" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="orderShipment">
  <xs:sequence>
    <xs:element name="ref" type="xs:int"/>
    <xs:element name="adr" type="tns:address" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="cancelTransact">
  <xs:sequence>
    <xs:element name="ref" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="quoteRequest">
  <xs:sequence>
    <xs:element name="product" type="tns:product" minOccurs="0"/>
    <xs:element name="quantity" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="quote">
  <xs:sequence>
    <xs:element name="price" type="xs:double"/>
    <xs:element name="product" type="tns:product" minOccurs="0"/>
    <xs:element name="quantity" type="xs:int"/>
    <xs:element name="refNumber" type="xs:int"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="address">
  <xs:sequence>
    <xs:element name="firstName" type="xs:string" minOccurs="0"/>
    <xs:element name="lastName" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="product">
  <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="bar"/>
      <xs:enumeration value="foo"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

We find here the names of the parameters, and their corresponding XML-Schema types. Summarized, we get:

- Operation `checkAvail`
  Input message parameters: `r` of type `quoteRequest`
  Output message parameters: `return` of type `quote`
- Operation `cancelTransact`
  Input message parameters: `ref` of type `xs:int`
- Operation `orderShipment`
  Input message parameters: `ref` of type `xs:int` and `adr` of type `address`

The corresponding complex types are:

- `quoteRequest`
  Elements: `product` of type `product` and `quantity` of type `xs:int`
- `quote`
  Elements: `price` of type `xs:double`, `product` of type `product`, `quantity` of type `xs:int`, and `refNumber` of type `xs:int`
- `address`
  Elements: `firstName` of type `xs:string` and `lastName` of type `xs:string`

This is very close to what we have written in the Java code. Still, it is necessary to check the concrete names and types, since this is crucial for the definition of the SSM, as we will see next.

Note: this information can also be automatically extracted and displayed by the *showWSDL* tool. It is part of the Jambition distribution (see the `showWSDL` directory).

2.4.3.3 The Warehouse SSM

Now comes the part where we specify the dynamic aspects of the warehouse via an SSM. Remember that an SSM consists of static constituents like types, messages, parameters, and operations. This information is already present in the WSDL file, as we have seen in the last section. Still to be modeled are the dynamic constituents like states, and transitions between the states. SSMs can be seen as a dynamic extension of a WSDL. They specify the legal ordering of the message flow at the service port, together with constraints on the data exchanged via message parameters.

A SSM can store information in SSM-specific variables. Every SSM 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. Take the WSDL operation `checkAvail`. This request-response operations has an input message with parameter `r` of complex type `quoteRequest`, and an output message `return` of complex type `quote`. In the SSM we model the call of the

checkAvail operation by two succeeding transitions, the first representing the input message, and the second the output message.

Figure 2.5 shows the SSM we want to use for this example, which specifies the warehouse service. What we specify here is the session protocol a user of the warehouse service has to follow.

Initially, the warehouse is in state 1. Now a user of the warehouse can invoke the checkAvail

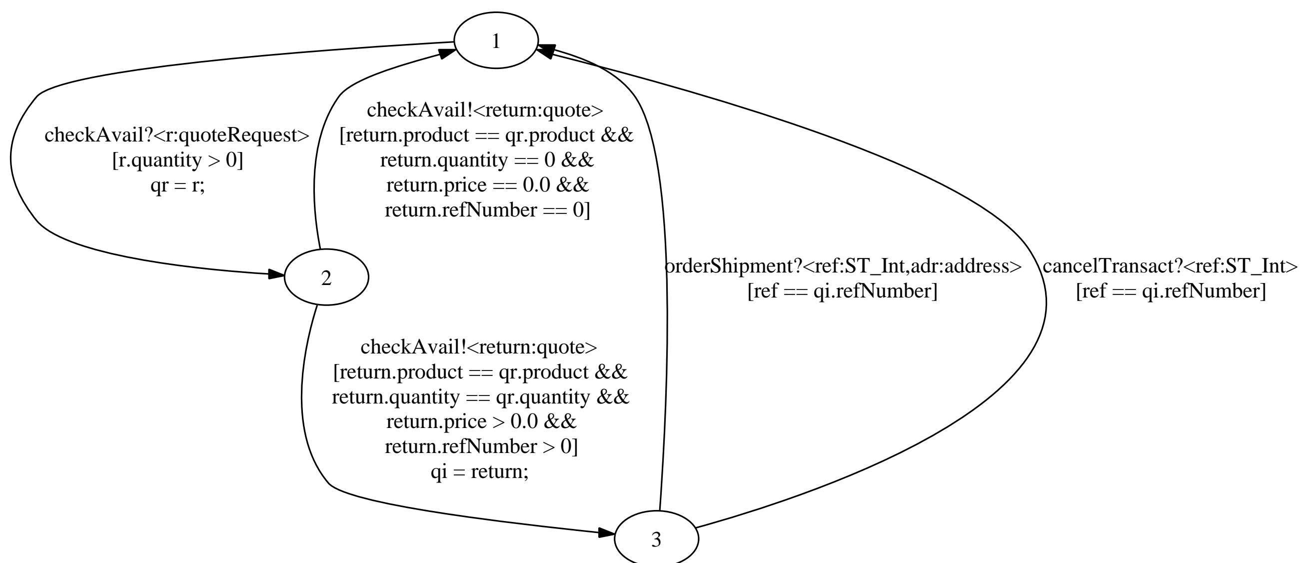

Figure 2.5: The Warehouse SSM

operation by sending the corresponding input message with parameter r of complex type quoteRequest. This corresponds to the transition from state 1 to state 2. The first line of the label is checkAvail?<r:QuoteRequest>. This states that this transition refers to the checkAvail operation. The appended question mark indicates that here the input message is modeled. The guard of the transition r.quantity > 0 restricts the attribute quantity of parameter r to be greater than zero. After the transition has fired, r is saved in the SSM-specific variable qr (which must also be of type QuoteRequest). Next, the warehouse has to return a Quote object via the return parameter return. Two things can happen. Firstly, the requested product may not be on stock with the requested quantity. In this case a Quote object is returned with a zero quantity, refNumber and price (transition from 2 to 1). Additionally it is ensured that the returned product attribute is the same as the requested one. This is achieved by relating the returned attribute product of return with the saved requested attribute product of qr: return.product == qr.product. Secondly, if the product is on stock, a Quote object is returned with the same quantity as being requested, and a price and refNumber greater than zero (transition from 2 to 3). We save the issued quote in the SSM-specific variable qi. Not that the appended exclamation mark indicates that these two transitions model possible output messages of the checkAvail operation.

Now again two things can happen. Either the user of the warehouse decides to reject the quote. He/she invokes the one-way operation cancelTransact by sending the message ?cancelTransact (rightmost transition 3 to 1). Here he/she must refer to the correct issued reference number refNumber. Or he/she decides to accept the quote. In this case, in addition to the correct reference number, an address must be provided as a second parameter (leftmost transition 3 to 1).

Note that in the picture the XML Schema type xs:int appears as ST_Int. This is due to an

automatic mapping of the XML Schema types to internal SSM types when the WSDL is read by Jambition.

2.4.3.4 Denoting the SSM in XML

We denote the SSM now directly in XML, meaning we create an instance of the `SP-SSM` schema which denotes exactly the SSM from Fig. 2.5:

```
<?xml version="1.0" encoding="UTF-8"?>

<spssm xmlns='http://frantzen.info/testing/ssmsimulator/schema/'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:package='http://services/'
       xsi:schemaLocation='http://frantzen.info/testing/ssmsimulator/schema/
         file:///home/lf/NetBeans/Audition/.../SP-SSM.xsd'>

    <wsdlURI>not needed</wsdlURI>
    <porttype>Warehouse</porttype>
    <states>
        <state>1</state>
        <state>2</state>
        <state>3</state>
    </states>
    <initialState>1</initialState>
    <variables>
        <var>
            <name>qr</name>
            <type>package:quoteRequest</type>
        </var>
        <var>
            <name>qi</name>
            <type>package:quote</type>
        </var>
    </variables>
    <switches>
        <switch>
            <from>1</from>
            <operationName>checkAvail</operationName>
            <kind>input</kind>
            <restriction>r.quantity > 0</restriction>
            <update>qr = r;</update>
            <to>2</to>
        </switch>
        <switch>
            <from>2</from>
            <operationName>checkAvail</operationName>
            <kind>output</kind>
            <restriction>return.product == qr.product &amp;&amp;
             return.quantity == qr.quantity &amp;&amp;
             return.price > 0.0 &amp;&amp; return.refNumber > 0</restriction>
```

```
            <update>qi = return;</update>
            <to>3</to>
        </switch>
        <switch>
            <from>2</from>
            <operationName>checkAvail</operationName>
            <kind>output</kind>
            <restriction>return.product == qr.product &amp;&amp;
             return.quantity == 0 &amp;&amp; return.price == 0.0
             &amp;&amp; return.refNumber == 0</restriction>
            <update></update>
            <to>1</to>
        </switch>
        <switch>
            <from>3</from>
            <operationName>cancelTransact</operationName>
            <kind>input</kind>
            <restriction>ref == qi.refNumber</restriction>
            <update></update>
            <to>1</to>
        </switch>
        <switch>
            <from>3</from>
            <operationName>orderShipment</operationName>
            <kind>input</kind>
            <restriction>ref == qi.refNumber</restriction>
            <update></update>
            <to>1</to>
        </switch>
    </switches>
</spssm>
```

2.4.3.5 Modeling the SSM in MagicDraw

Please refer to the Minerva chapter 3.

2.4.3.6 Starting Jambition, treeSolver, sdedit, and Checking the Preferences

We now have the necessary specifications ready – the WSDL and the SSM. We assume that the corresponding (stateful) warehouse Web Service is successfully deployed. We start Jambition and see the main GUI (see Fig. 2.3). In the `WSDL URL` textfield we enter the URL of the WSDL, and in the `SSM URL` textfield the URL of the SSM. Not that in case of a local file one can use the `file://PathToFile` notation. Since a WSDL can in principle define several services and ports, it is additionally necessary to name the specific service and port to be tested. If unsure what to put here, consult here the WSDL. For our example the relevant WSDL-part looks like this:

```
<service name="WarehouseService">
  <port name="WarehousePort" binding="tns:WarehousePortBinding">
    [...]
  </port>
</service>
```

Hence, we put `WarehouseService` in the `Service` textfield, and `WarehousePort` in the `Port` textfield.

Next we check the preferences by choosing `Preferences` from the `Jambition` menu. For all preferences options we keep the default values. Make sure that the `treeSolver` is running under the port given. Also start `sdedit` and make sure that its real-time server is running on the right port (via its `File` menu item `Start/stop RT server...`). Close the preferences.

2.4.3.7 Generating the SSM Object

Now we are ready to generate the SSM object. We do so by pressing the `Generate SSM` button. After a short while a little notification window should report `SSM successfully generated!`. The Monitor of the Jambition GUI should report something similar to:

```
08/10/07 11:54:10:161: treeSolver is bound to local port 59132
08/10/07 11:54:10:283: treeSolver sends: treeSolver V180607 ready
08/10/07 11:54:10:285: qsdeServer is bound to local port 56311
08/10/07 11:54:10:451: STSimululator ready.
```

The `treeSolver` reports something like `Connected with 127.0.0.1`, and `sdedit` shows in a new tab called `Jambition` the two actors `Jambition` and `SUT` (which is the Web Service Under Test).

2.4.3.8 Using `dot` to Visualize the SSM

Note that after having successfully generated the SSM object, the button `dot Input` is enabled. Pressing this button outputs the generated SSM in a textual format understood by the `dot` tool from the `Graphviz` toolsuite. For our example, we get this output in the Jambition Monitor:

```
digraph STS {
1 -> 2 [label = "checkAvail?<r:quoteRequest[product:product{bar,foo},
        quantity:ST_Int]>\n[r.quantity > 0]\nqr = r;"];
2 -> 1 [label = "checkAvail!<return:quote[price:ST_PseudoPosDouble,
        product:product{bar,foo},quantity:ST_Int,refNumber:ST_Int]>\n
       [return.product == qr.product &&\n return.quantity == 0 &&\n
        return.price == 0.0 &&\n return.refNumber == 0]\n"];
2 -> 3 [label = "checkAvail!<return:quote[price:ST_PseudoPosDouble,
        product:product{bar,foo},quantity:ST_Int,refNumber:ST_Int]>\n
        [return.product == qr.product &&\n return.quantity == qr.quantity
        &&\n return.price > 0.0 &&\n return.refNumber > 0]\nqi = return;"];
3 -> 1 [label = "orderShipment?<ref:ST_Int,adr:address[firstName:ST_String,
        lastName:ST_String]>\n[ref == qi.refNumber]\n"];
3 -> 1 [label = "cancelTransact?<ref:ST_Int>\n[ref == qi.refNumber]\n"];
}
```

Now one can copy-paste this into a text editor and save it as a file. Running the `dot` tool on it generates a graph representing the SSM. Figure 2.5 shows a slightly simplified version of the generated graph where the details of the complex types have been removed.

2.4.3.9 Testing the Warehouse Service

Finally we are in the position to test if the warehouse service does conform to the SSM specification we have developed. Let us recapitulate what we have done so far. The initial assumption

in this running example was that we have developed a warehouse Web Service in the Java programming language. To access this service the `JAX-WS` package automatically generates a corresponding WSDL file which specifies the operations available at the service's port, and its physical address (also called the *endpoint*) of the service.

To specify dynamic aspects of the service we have additionally modeled an SSM. This SSM specifies the legal ordering in which the warehouse operations must be invoked, and restricts the data which is exchanged via the operations. On the one hand, the SSM specifies a protocol for a potential user of the warehouse. For instance, it postulates that firstly the `checkAvail` operation must be called with a positive quantity. On the other hand, the SSM specifies a protocol to which the warehouse service itself must conform. For instance, the quote returned by the `checkAvail` operation must always deal with the same product that being requested.

In other words, the SSM deals with two actors. One is the user of the warehouse invoking its operations. The other is the warehouse itself receiving and responding to the operation calls. When testing the warehouse, Jambition is playing the role of the user of the warehouse. Initially, being in state `1`, the only specified call is `checkAvail`. To invoke this operation, a parameter of type `quoteRequest` must be constructed. The guard additionally restricts the parameter `quantity` to be greater zero (see transition from state `1` to `2`). Jambition will randomly choose a product (`foo` or `bar`), and a quantity greater zero. With this `quoteRequest` parameter it then invokes the `checkAvail` operation. Now, being in state `2`, Jambition receives the quote object returned by the warehouse. The SSM dictates, that this quote has to have a zero quantity, price, and reference number (transition from `2` to `1`), or the same quantity as requested and a positive price and reference number (transition from `2` to `3`). For both transitions must hold that the quote's product must be the same as the requested product. Jambition checks now, which of the two cases holds for the received quote, and moves to the respective next state (`1` or `3`). If none holds, a failure has been detected. For instance, this is the case if the returned quote deals with a different product than requested. Or, if the quantity and price are zero, but the reference number is not.

Assuming the quote was correct and Jambition moved to state `3`, then it will next send either a `cancelTransact` or an `orderShipment` to the warehouse (taking care of using the correct reference number). And so on.

While no failure is found, the testing continues. If enabled, `sdedit` shows the messages exchanged in realtime. The user can halt the testing by pressing `Stop Testing`. Doing so enables the `Detailed Coverage` button. Pressing it detailed coverage information of the SSM are displayed (which state and which transition has been visited how often up to now). The testing can be continued by pressing `Continue Testing`, or ended by pressing `End Testing`.

2.5 Appendix

2.5.1 The Dumont Grammar in BNF

```
     SwitchRestriction ::= BooleanExpression <EOF>
         UpdateMapping ::= ( VarAssignment )* <EOF>
         VarAssignment ::= Id "=" TermExpression ";"
     BooleanExpression ::= LogicalOrExpression ( "&&" LogicalOrExpression )*
   LogicalOrExpression ::= LogicalNotExpression ( "||" LogicalNotExpression )*
  LogicalNotExpression ::= "!" LogicalNotExpression
                         |  TermExpression TermEqualityExpression
                         |  "(" BooleanExpression ")"
TermEqualityExpression ::= "==" TermExpression
                         |  "!=" TermExpression
```

```
                        |  "<=" TermExpression
                        |  "<" TermExpression
                        |  ">=" TermExpression
                        |  ">" TermExpression
        TermExpression ::= TermAddExpression
     TermAddExpression ::= TermSubtractExpression
                           ( "+" TermSubtractExpression )*
TermSubtractExpression ::= TermMultExpression ( "-" TermMultExpression )*
    TermMultExpression ::= TermDivExpression ( "*" TermDivExpression )*
     TermDivExpression ::= TermModExpression ( "/" TermModExpression )*
     TermModExpression ::= TermUnaryExpression ( "%" TermUnaryExpression )*
   TermUnaryExpression ::= "++" TermUnaryExpression
                        |  "--" TermUnaryExpression
                        |  "(" TermExpression ")"
                        |  PrimaryTermExpression
 PrimaryTermExpression ::= Literal
                        |  Id
                  Name ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
                    Id ::= Name
               Literal ::= ( <INTEGER_LITERAL> )
                        |  ( <BOOLEAN_LITERAL> )
                        |  ( <STRING_LITERAL> )
                        |  ( <ENUMERATION_LITERAL> )
```

3 Minerva

3.1 Minerva Overview

Minerva is a tool for using SSM models designed in MagicDraw [18] in the testing environment of Jambition. It reads such models and converts them to an internal representation, suitable to be used by Jambition to test a web service. It also contains `showWSDL` and other library utilities called by Jambition in order to accomplish its validation and testing activities.

3.2 Technical info

Provider 4D Soft Ltd.

Introduction Minerva is a library containing the following services:

- `showWSDL` This is a tool which parses a WSDL document and outputs its structure in a human readable format.
- `getSSMFromWSDL`
  This command takes a WSDL document and an SSM XML description and creates an internal representation of the SSM, passing it to Jambition for testing a web service.
- `getSSMFromUML`
  This call takes an XMI file, containing an UML representation of an SSM model and generates an internal SSM used further by Jambition for testing and validating a web service.

Development status The currently available version is 2.0.

Intended audience Software developers who wish to validate and functionally test their services designed conform to the SP SSM model and modelled with the MagicDraw UML tool.

License This software is open source. GPL version 3 license is used.

Language Java

Environment (set-up) Minerva runs in a Java environment, so a Java 5 or higher Runtime Environment is needed in order to use it. The libraries used are open-source and packaged with the installation. Because it is a library for Jambition, a Jambition installation is also needed. There are no special hardware or software requirements beside this.

Platform Java Runtime Environment 5 or later.

Download http://plastic.isti.cnr.it/download/tools

Documents Javadoc API, this guide.

Contact Zsolt G. Kiss, zsolt.kiss@4dsoft.hu

3.3 Deployment

3.3.1 Install

To install Minerva, unzip the archive `showWSDL.zip` in a directory of choice, referred later as `$MINERVA_HOME` in this document.

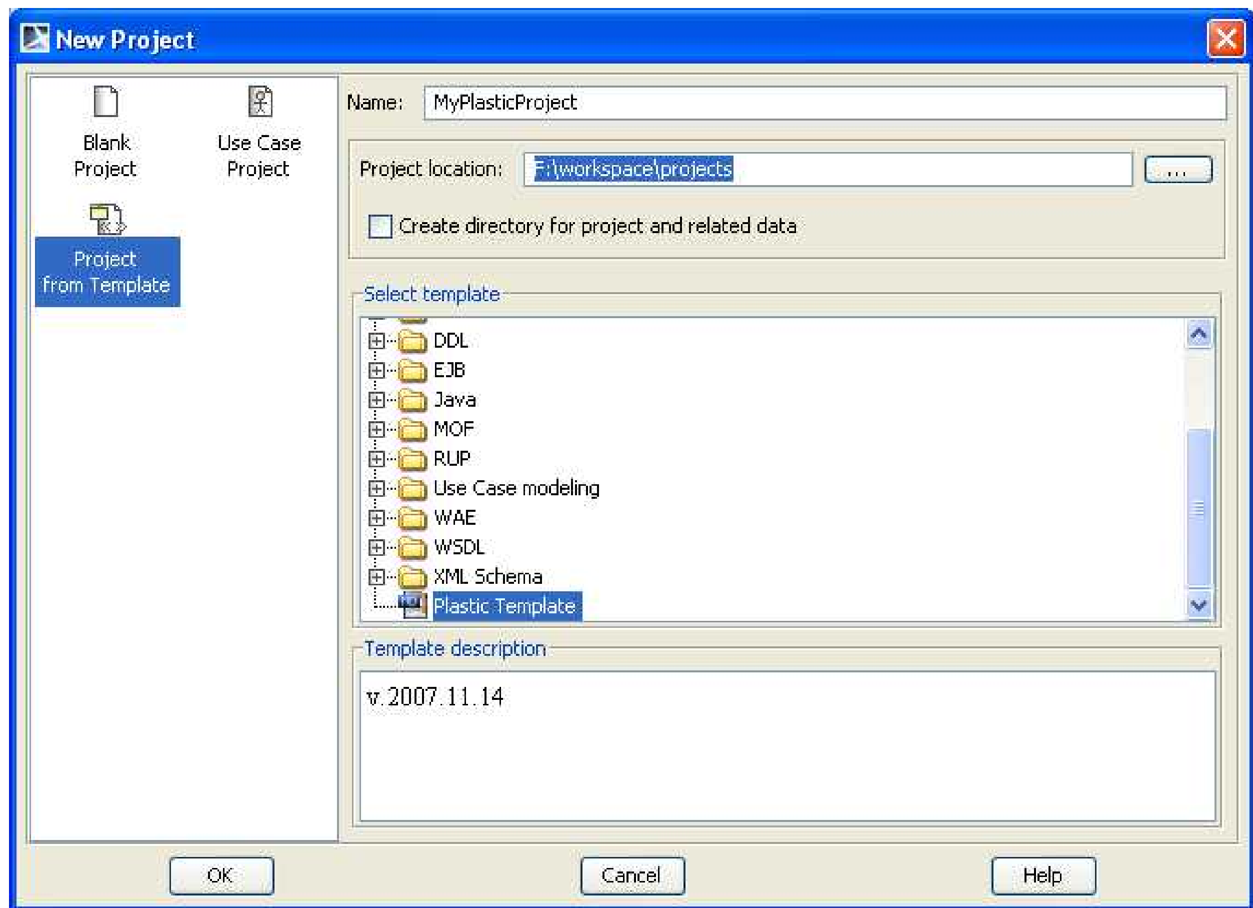

Figure 3.1: Creating a new project

3.3.2 Configure/Usage

The `java` (version ≥ 5.0) should be in the PATH, other configuration is not needed. The used libraries are contained in the package.

ShowWSDLElements tool usage:

1. Change directory to `$MINERVA_HOME`
2. Type `java -jar showWSDL.jar <wsdl-uri>`

3.4 Tutorial

We'll follow the steps of modeling the Warehouse service, described in the Jambition chapter, with an SSM, using MagicDraw.

3.4.1 Creating the project

First we create a PLASTIC project.

- From the *File menu*, choose *New Project*

The *New Project* dialog box opens.

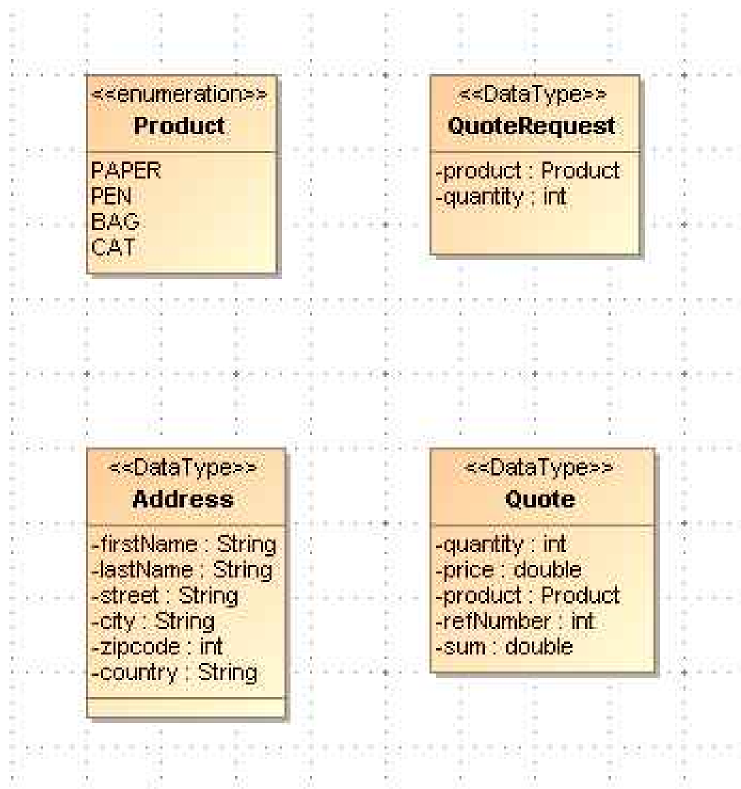

Figure 3.2: The basic types and data structures

1. Select the *Project from Template* icon.
2. Specify the name (e.g. *Warehouse*) in the *Name* text box.
3. Choose the *Project Location*.
4. Select *PLASTIC Template* from the templates available and click *Ok*(fig. 3.1).

Click on the <Top Level Service Name> in the Containment browser and rename it to *Warehouse* (choose Rename from the context menu, obtained by right-clicking the name.)

The created PLASTIC project contains the necessary module imports (PLASTIC Profile and UML Standard Profile) and a package structure already based on the PLASTIC Conceptual Model, containing the 5 basic views. For the SSM we are interested mainly on the Service View, which consists of a Structural View and a Behavioral View.

Next we need to model the composite and other user-defined types which will be used through the project.

3.4.2 Modeling the types and data structures

The Warehouse example contains the following user-defined types: Product, Address, Quote and QuoteRequest. These are shown in Fig. 3.2.

We model these types in a class diagram:

1. From the elements browser pane on the left choose, then right-click “Warehouse — 002.Service View — Structural View”.
2. Select New Diagram — Class Diagram, and specify a name (*WarehouseTypes*)

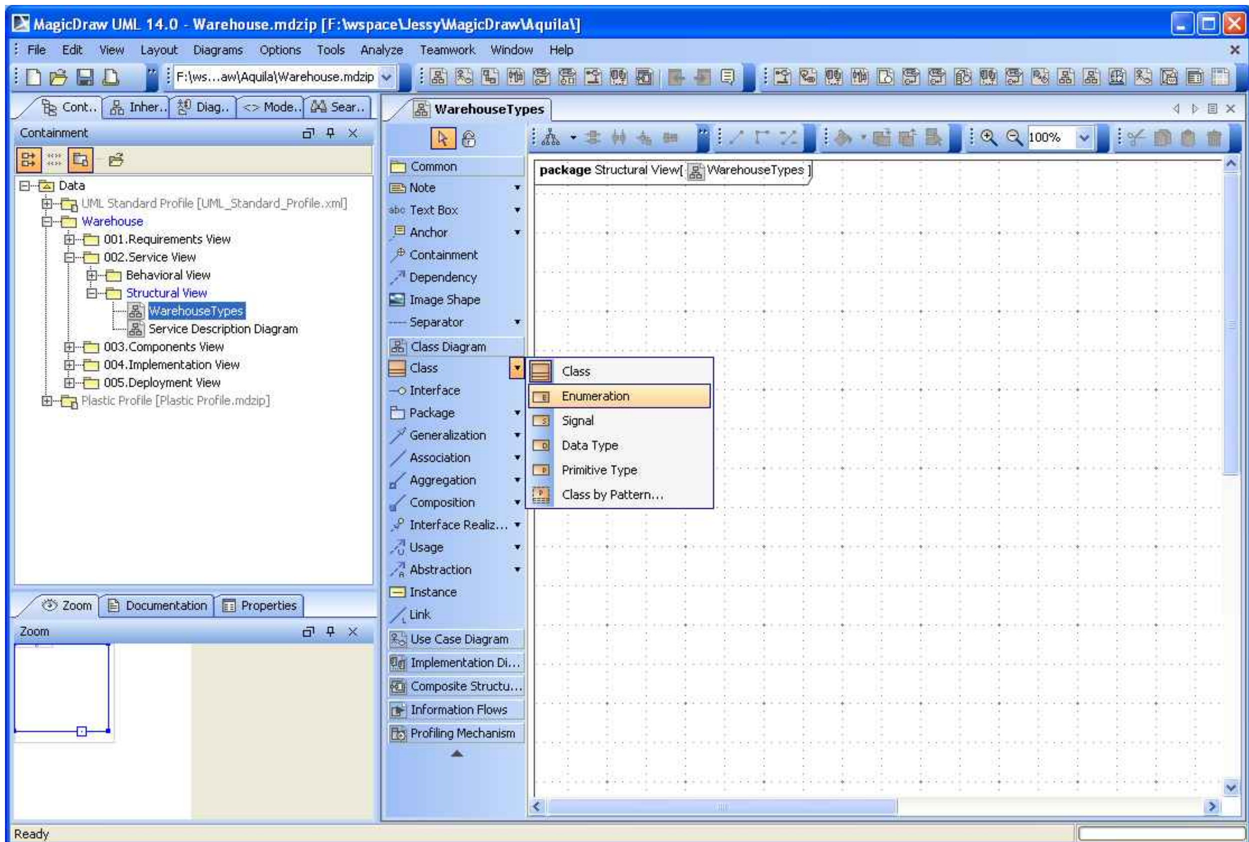

Figure 3.3: Creating an enumeration

3. Add the type elements to the diagram, using the Class and Enumeration shape buttons on the diagram toolbar, found in the "Class Diagram" button group.

First create the enumeration called Product. On the Class Diagram toolbar click the arrow found at the right side of the Class button, then select Enumeration (fig. 3.3):

Create the enumeration by clicking on the canvas. Then, open the specification dialog and set the name ("Product") and the desired values. For the values, click "Enumeration Literals" on the left pane of the specification dialog, then press the "Create" button for each String literal you want to add. Specifying a name for a literal is enough.

Next create the classes similarly. For each class specify the Name, Applied Stereotype and the class Attributes.

To set the stereotype, click on the Value cell of the Applied Stereotype line in the specification dialog. Then click the small rectangle button on the right side, labelled [. . .]. This will open a list box containing all stereotypes which can be applied to the element. Typing "d" the list will be filtered to the stereotypes starting with "d" – in our case the list will contain only one element, "DataType", defined in the PLASTIC Profile. Select this element with the check box and click Apply (fig. 3.4).

Next define the class attributes. Select Attributes in the left browser tree of the specification dialog, then hit Create.

The following data should be specified for each attribute:

- Name
- Type

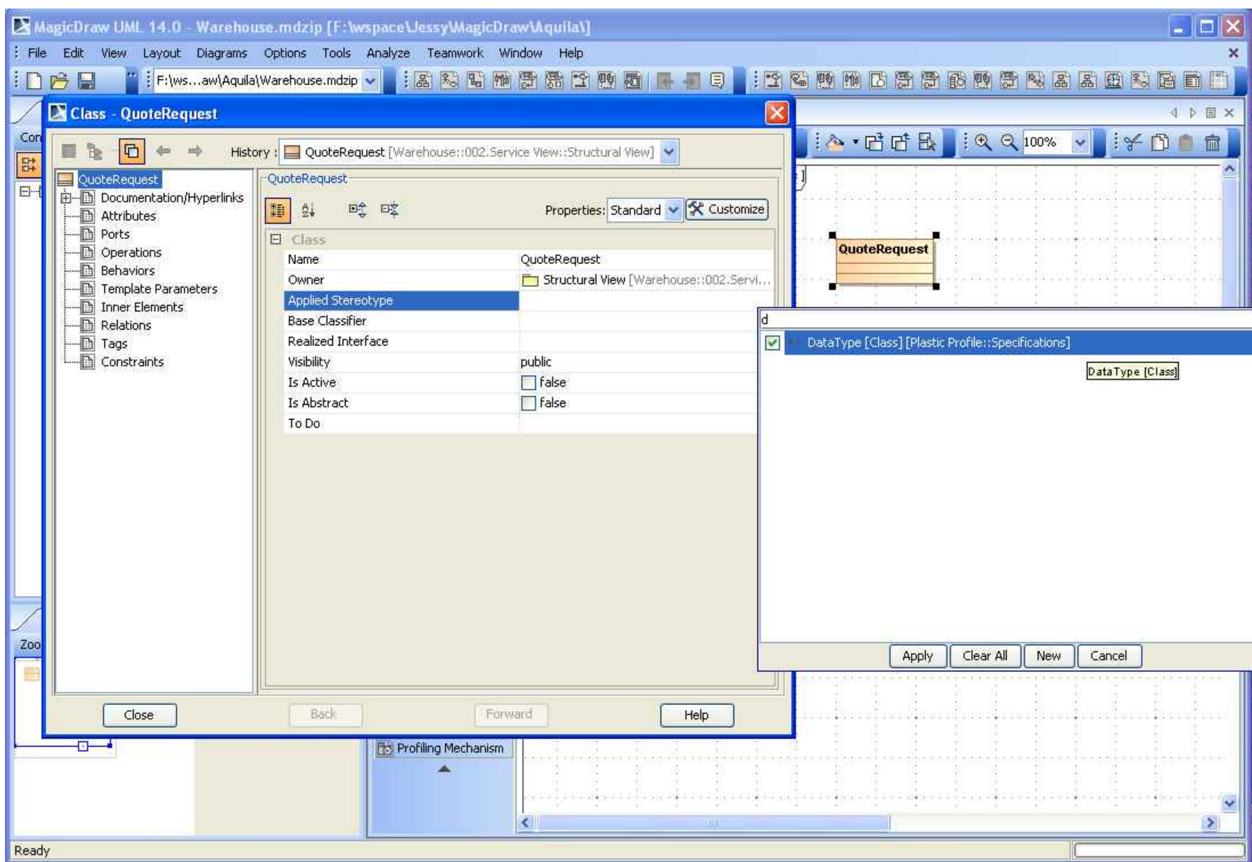

Figure 3.4: Applying a stereotype

For the type, there is also a context-sensitive list box, which filters by the first entered characters. For the primitive types, use the [UML Standard Profile::MagicDraw Profile::datatypes] package.

After defining the classes needed by the Warehouse project, the class diagram will look like in fig. 3.5.

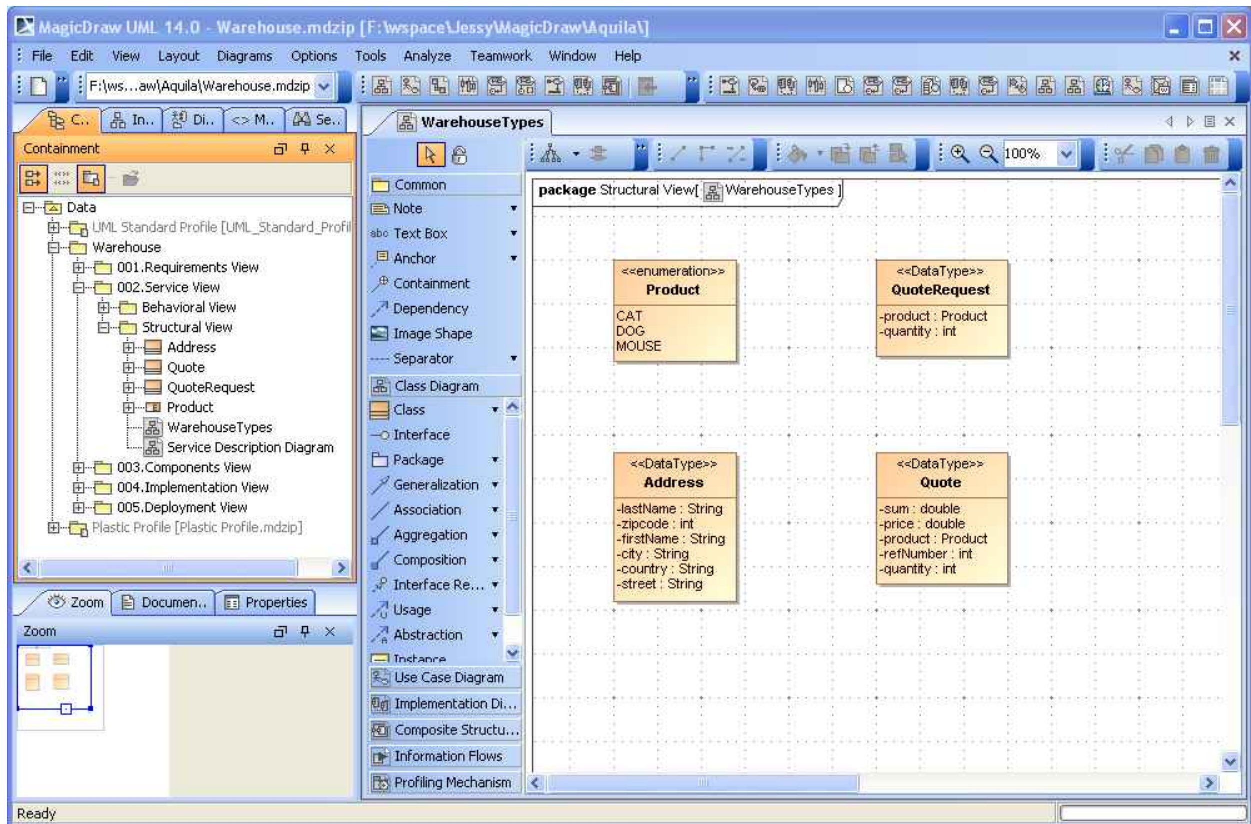

Figure 3.5: The warehouse types diagram

The next step is the specification of services. For this, the Service Description Diagram will be used.

3.4.3 Creating the service description

Next, we create the service (fig. 3.6):

Figure 3.6: The service

In the Service Structural View there is already an empty Service Description Diagram, created by the PLASTIC template. Double-click it in the elements browser to open the diagram for editing. Select the "Service Description" button in the diagram toolbar and click in the canvas to create

a new Service Description (this is the service interface.) Open its specification dialog and enter “Warehouse” for the name. Then choose Operations on the left browser pane and add the necessary operations.

For each operation we should define the following:

- Name
- Kind (*REQUEST_RESPONSE, ONE_WAY, NOTIFICATION or SOLICIT_RESPONSE*)
- Parameters

For each parameter, we should specify:

- its name
- type (primitive, or user-defined, as discussed above) and
- direction (in, out, inout or return)

Note: you have to define a parameter for the return value, which has to be unique between all parameter names (e.g.: rq_return, and not just $return$)

After defining the service description, the service diagram will look like in fig. 3.7.

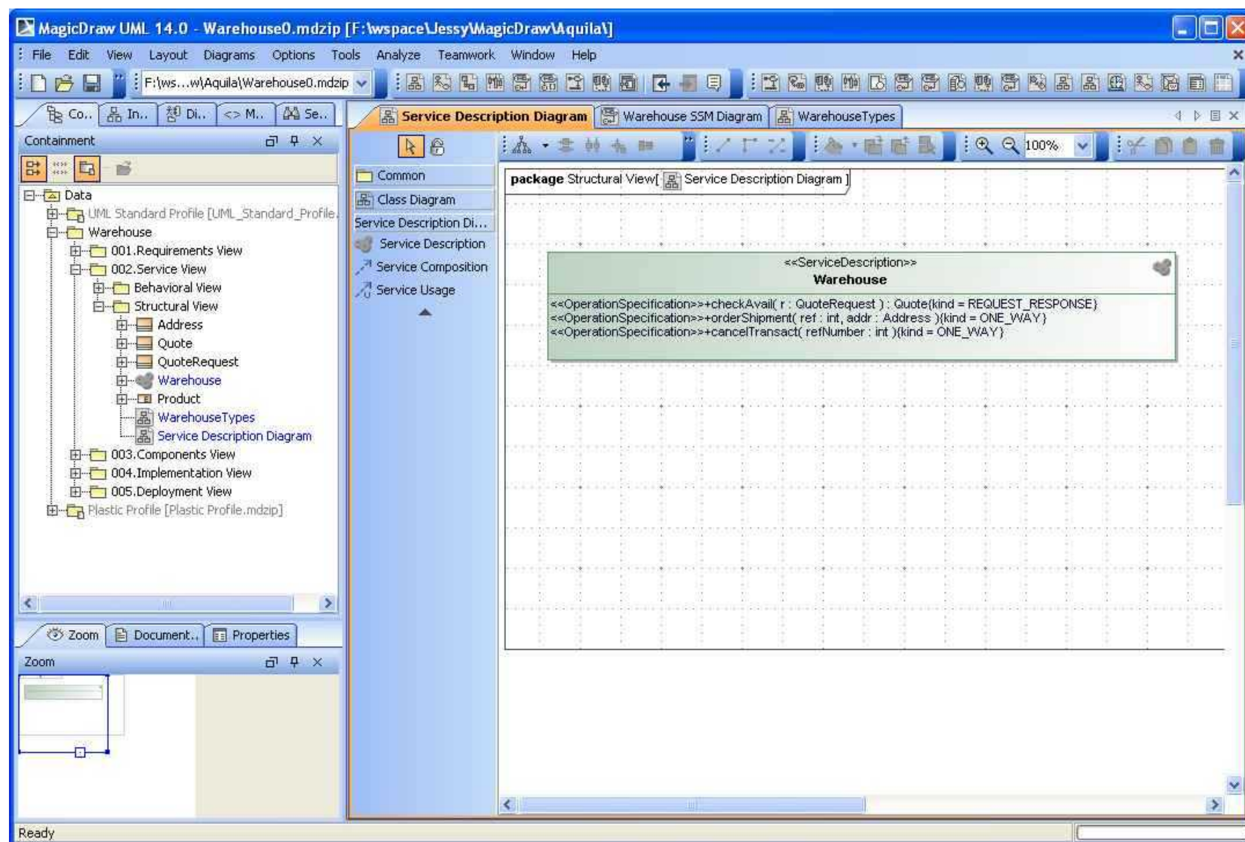

Figure 3.7: The service diagram

Then we can move on specifying the services dynamic behavior, with the SSM.

3.4.4 Creating the Service State Machine

Now follows the most interesting part of the design, specifying the SSM. We define this by drawing an SSM diagram, which will look like in fig. 3.8.

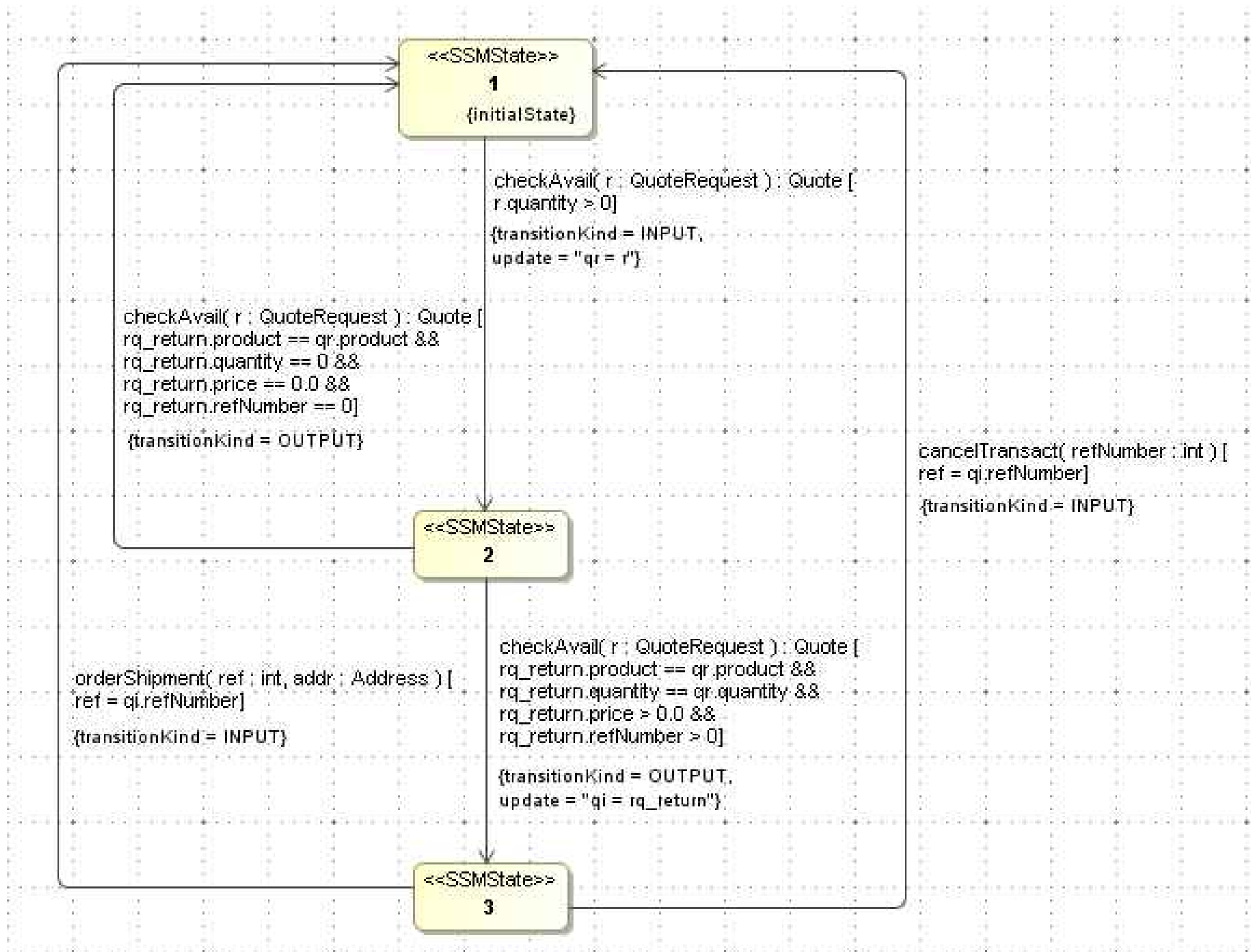

Figure 3.8: The service state machine diagram

We will create an SSM for our Warehouse service.

1. From the elements tree browser pane on the left choose, then right-click “Warehouse — 002.Service View — Behavioral View”.
2. Select New Diagram — JSSM Diagram.

At this point, MagicDraw has done the following: it created an SSM called “Untitled1” and an SSM Diagram underneath, as a child element, called also “Untitled1”. To rename the elements, right-click on each of them and choose Rename from the context menu.

Double-clicking on the newly created diagram name will open the diagram window. The toolbar will have a group called “JSSM Diagram” with 2 shape buttons: State `<<SSMState>>` and Transition `<<SSMTransition>>`. We create the diagram using these element types.

The SSMState is a very simple element. We specify its name (usually a number) and if it is the initial state, we set its initialState property to true. This property can be found under Tags, under the `<<SSMState>>` group, listed at the bottom. Select the tag (“initilaState”), then press the “Create Value” button. After double-clicking the Value check-box on the right the value is set (fig. 3.9).

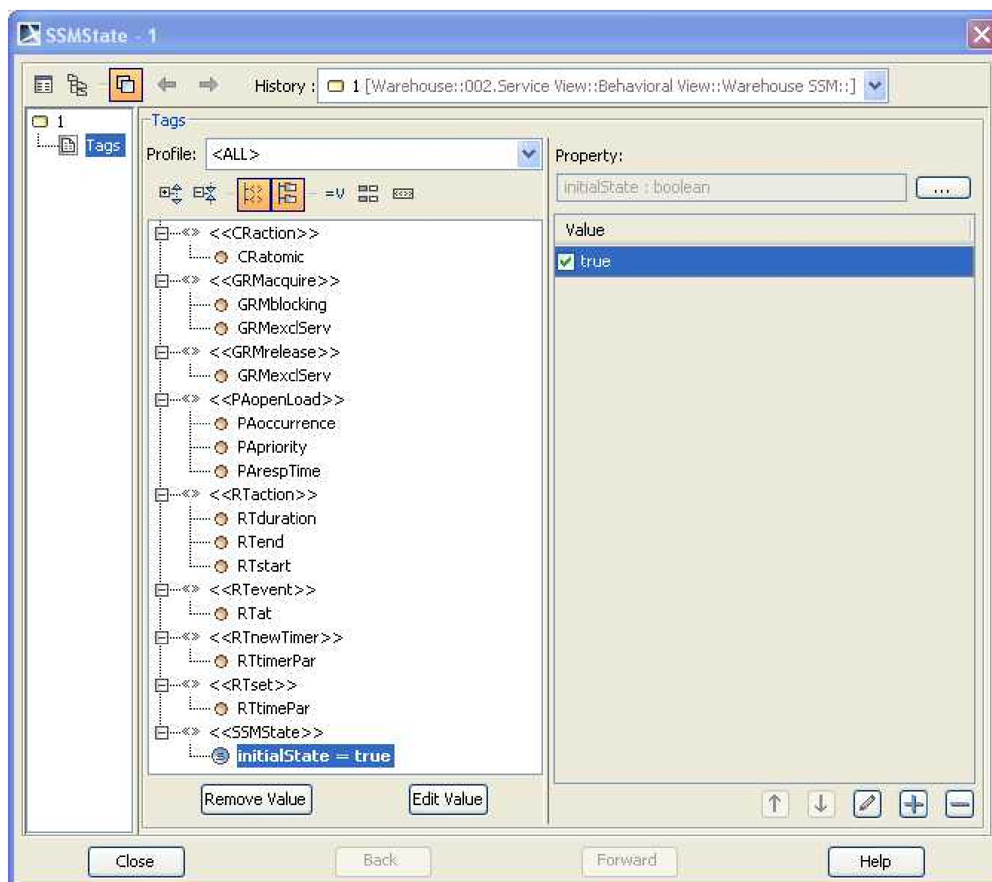

Figure 3.9: Setting initial state

For the other states it is not necessary to deal with this, because the default value is false.

In our example we have 3 states. Let us model these first, then create the state transitions using SSMTransition elements.

After creating a transition, select and delete the label `<<SSMTransition>>` which appears next to it. For each state transition (called "switch" in SSM language) the following information should be entered:

- **Operation** *(in the Trigger group)*

  To see this property, first choose an EventType of CallEvent. This selection will show the Operation property in the Trigger group. Click on the small dialog button ([...]) to open the operation value dialog. On the tree which appears, select the operation checkAvail() from the Warehouse service (fig. 3.10).

- **TransitionKind** *(in the SSMTransition group)*

  This is the message kind. For the first transition (from $1 \rightarrow 2$), choose INPUT.

- **Guard**

  This is the guard of the transition ("$r.quantity > 0$" for the $1 \rightarrow 2$ transition). By opening the Edit dialog (with the right-most button), you can write a large guard using a text box. Choose Language = English. For better display, it is recommended to format the text using new-lines, which are stripped out during the analysis. In fig. 3.11 you see the guard format of the transition from state $2 \rightarrow 3$.

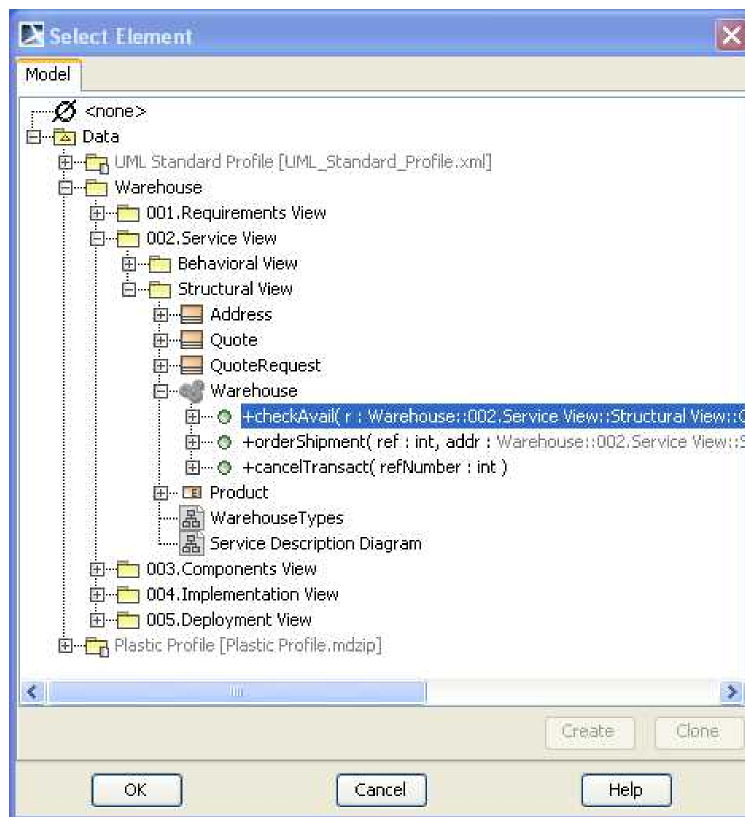

Figure 3.10: Selecting an operation

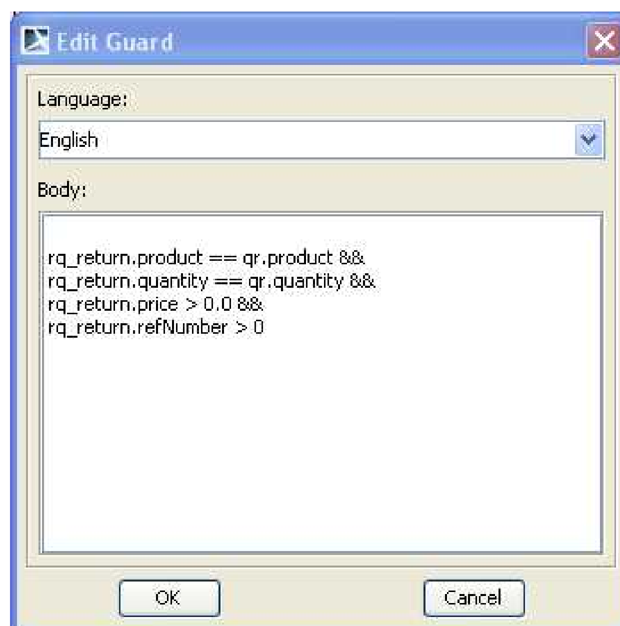

Figure 3.11: Editing a guard

- **Update**

  Represents the switch update command, usually a simple assignment. For the $1 \rightarrow 2$ transition, it is “$qr = r$”.

  Create all transitions in the same manner. For a better visual experience, you can arrange the label positions in a convenient way. For some transitions (e.g. $2 \rightarrow 1$ and $3 \rightarrow 1$) some manual formatting of the arrows is also desirable, in order to produce a decent-looking diagram. After the SSM is modeled, its diagram would look like similar to fig. 3.12.

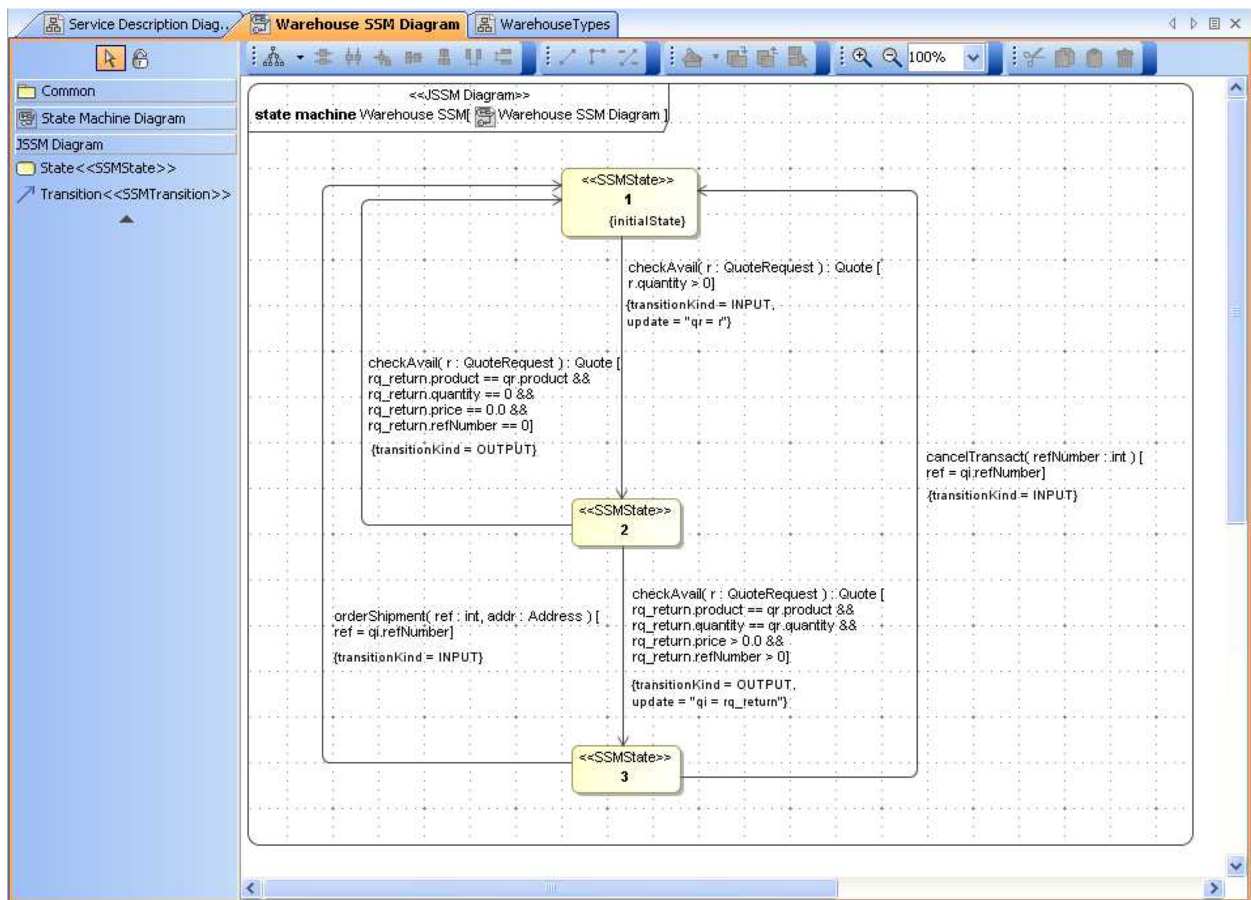

Figure 3.12: The SSM diagram in MagicDraw

3.4.5 Location variables

When setting the guard and update statements, you might have noticed that there were used some new variables, which were not operation parameters. These are the so-called location variables, which record the state of a conversation.

Location variables have to be defined as attributes of the SSM. In the containment browser right-click the SSM object and select **Specification.** In the specification dialog, on the left pane select **Attributes**. Then create each variable with the **Create** button. A name and a type should be specified. The type can be selected from a dialog, like all other parameter types.

In our case we have 2 location variables, qr and qi (fig. 3.13).

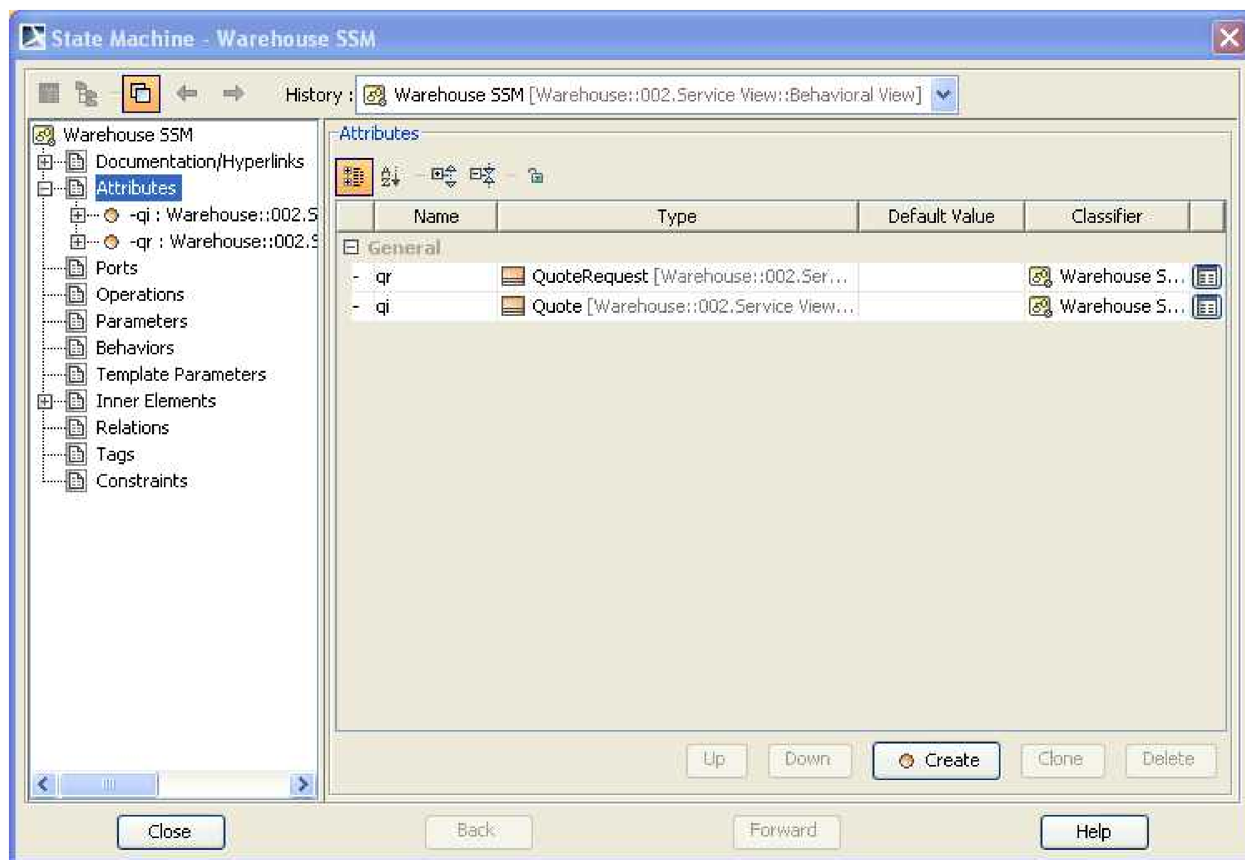

Figure 3.13: Location variables in the SSM

3.4.6 Exporting the SSM

After having created the model, we have to export it for further use. Because there will be one file exported for each (sub)module, the Warehouse project will be exported in 17 files.

Create a directory when the output will be stored, then select **File** | **Export** | **EMF UML2 (v1.x) XMI**. After selecting the output directory, the project will be exported.

4 Puppet

This chapter is the user's guide for PUPPET . Please refer to [7] for the detailed description of the whole approach, the architectural description of the proposed implementation, the tools and the standard that have been used, or for any kind of motivation of the work.

4.1 Puppet Overview

To ensure consistent cooperation for business-critical services, with contractually agreed levels of Quality of Service, SLA specifications as well as techniques for their evaluation are nowadays irremissible assets. Puppet (Pick UP Performance Evaluation Test-bed) is an original approach developed within PLASTIC for the automatic generation of test-beds to empirically evaluate the QoS features of a Web Service under development. Specifically, the generation exploits the information about the coordinating scenario, the service description (WSDL) and the specification of the agreements (WS-Agreement).

As described in [7, 6, 5], PUPPET was originally designed in order to automatically generate stubs conforming only to SLA behaviors ignoring functional aspects. In other words, if invoked the stub were able to provided good QoS values but the responses were not built to be semantically meaningful (i.e. always returning constant values). However, in the general case extra-functional aspects are tightly coupled with functional characteristics. The current version of PUPPET integrates the emulation of the functional specifications as part of the generated testbed. The obtained environment can expose not only the specified extra-functional parameters but also meaningful functional behavior. Specifically, PUPPET generates stubs for Web Services which respect both an extra-functional contract expressed via a Service Level Agreement (SLA), and a functional contract modeled via a Service State Machine (SSM, see Chapter 2).

4.2 Technical info

Provider CNR

Introduction PUPPET is a tool for the automatic generation of test-beds to empirically evaluate the QoS features of a Web Service under development. The stubs generated with PUPPET conform both the extra-functional contract expressed via a Service Level Agreement (SLA), and to the functional contract modeled via a state machine.

Development status Version PuppetD4.3

Intended audience Developers who intend to test a PLASTIC service in composition with 3rd party Web Services

License Open source under GPLv3 with some exceptions to include libs

Language Java, XML

Environment (set-up) In the following the required software and hardware :

Hardware: No specific hardware is required.

Software: The following JAR library are required in order to compile and to launch PUPPET :

Apache Axis 1.4 Libs : axis-ant.jar, axis.jar, commons-discovery.jar, commons-logging.jar, jaxrpc.jar, jsr173_1.0_api.jar, log4j-1.2.8.jar, wsdl4j.jar, saaj.jar – availabe at http://ws.apache.org/axis.

Apache XMLBeans Libs : xbean.jar – availabe at http://xmlbeans.apache.org.

String Template Libs : stringtemplate.jar – availabe at http://www.stringtemplate.org.

INI4J Libs : ini4j.jar, ini4j-compat.jar – availabe at http://ini4j.sourceforge.net.

Jambition Libs : SSMSimulator.jar, minerva.jar (see Chapter 2)

Other Libs : antlr-2.7.7.jar, xercesImpl.jar, xmlsec.jar, mail.jar, activation.jar, java-getopt-1.0.13.jar

Please note that **PuppetD4.3.tgz** archive includes a version of these libraries in the directory **PuppetD4.3/externalLibs**.

Platform Java jdk1.6 or later

Download Download the official version of PUPPET in PLASTIC at http://plastic.isti.cnr.it/download/tools

Documents Related documents on the approach, the architectural description, and the implementation of PUPPET are [7, 6, 5].

Tasks N/A

Bugs N/A

Patches N/A

Contact guglielmo.deangelis@isti.cnr.it, andrea.polini@isti.cnr.it

4.3 Deployment

4.3.1 Install

Unzip the archive **PuppetD4.3.tgz** and configure the environmental CLASSPATH with the required libs indicated above.

The directory structure is the following:

- PuppetD4.3
  - doc
  - example
  - externalLibs
  - puppet.jar
  - puppetLibs
  - puppet.sh
  - runPuppet.bat
  - src
  - xml

The directory **PuppetD4.3/doc** contains this manual. The directory **PuppetD4.3/xml** contains the definitions mapping of the WS-Agreement statements into the Java code. The directories **PuppetD4.3/externalLibs** and **PuppetD4.3/puppetLibs** contains the libraries required by PUPPET in order to run. As your preference, you would append the name of the **.jar** files in **PuppetD4.3/puppetLibs** and **PuppetD4.3/externalLibs** to the Java CLASSPATH variable. **PuppetD4.3/puppet.sh** and **PuppetD4.3/runPuppet.bat** are exacutable batch scripts that set the Java CLASSPATH variable and run PUPPET on a given input file.

4.3.2 Configure

PUPPET generates stubs for web services according to what defined in a given configuration file. The configuration file is supposed to compile with the standard INI File Format. In such file, PUPPET looks for the section named **[mainSection]**. PUPPET loads its parameters as specified in the configuration held by this section.

The parameters that could be specified into the input configuration file are:

wsdlPath : It is the path to the directory holding the WSDL specifications of all the services whose emulators would be generated by means of PUPPET . Required.

targetPath : It is the path to the directory where PUPPET will dump the generated stubs. Required.

JarPath : It is the path to the required JAR file libraries listed above.[1]. Required.

wsaFilename : It is the name of the WS-Agreement file describing the agreements among the considered services. If it is not specified, PUPPET would look for a file named: **agreement.xml**. Optional

trueTermsFilename : It is the name of the file holding the terms of the agreement that have to be considered as fulfilled. As described in Sec 4.4.1, for each term in the agreement, PUPPET will generate code that emulates an extra functional behavior if and only if the term is fulfilled. If this file name is not specified, PUPPET would look for a file named: **gtTrueItemList.xml**. Optional

wsaPath : It is the path to the directory holding the WS-Agreement file. Required.

trueTermsPath : It is the path to the directory holding the *trueTermsFilename*. If it is not specified, PUPPET would assign to this parameter the path to the directory holding the WS-Agreement file (*wsaPath*). Optional.

qcMappingFilename : It is the path to the file holding the template mapping of the Qualifying Conditions in WS-Agreement on to the the Java code that will be generated. PUPPET already includes a predefined mapping file. Even though it is possible to change this mapping, we strongly discourage from changing it. Optional.

sloMappingFilename : It is the path to the file holding the template mapping of the Service Level Objectives in WS-Agreement on to the Java code that will be generated. PUPPET already includes a predefined mapping file. Even though it is possible to change this mapping, we strongly discourage from changing it. Optional.

ambitionMode : If it is set to "**on**" enables the emulation of the functional behavior with Jambition. By default it is set to "**off**". Optional.

[1] In the future releases it would be deprecated

```
<wsag:AgreementOffer xsi:schemaLocation="..." xmlns:wsag="...">
...
    <wsag:GuaranteeTerm wsag:Name="WarehouseGT"
     wsag:Obligated="ServiceProvider">
      ...
    </wsag:GuaranteeTerm>
...
</wsag:AgreementOffer>
```

```
<tns:TrueGTList xmlns:tns="..." xmlns:xsi="..." xsi:schemaLocation="...">
...
 <tns:GTItemName>WarehouseGT</tns:GTItemName>
...
</tns:TrueGTList>
```

Table 4.1: Enabling the code generation in PUPPET

4.3.3 Usage

Let us assume that the variable **CLASSPATH** of the JVM you are executing includes both the JAR files listed in the item **Tools** above, and those contained in **PuppetD4.3/puppetLibs.** PUPPET usage is:

java -cp $CLASSPATH:puppet.jar puppet.Puppet <IniConfigurationFile>

An alternative way to run PUPPET is executing the batch scripts **PuppetD4.3/puppet.sh** and **PuppetD4.3/runPuppet.bat** [2] on a given <IniConfigurationFile>.

4.4 Tutorial

4.4.1 Terms in the Agreement and Generation Process

In WS-Agreement [17], an agreement specification is composed by one or more terms. These terms are grouped in a logic formula by means logic connectors (**All** – logic AND, **OneOrMore** – logic OR, and **ExactlyOne** – logic XOR).

Different scenarios could be considered defining a set of terms in the agreement that are assumed fulfilled. This set of terms enables in PUPPET the generation of the Java code emulating the extra functional behavior. PUPPET loads the list of these terms parsing an XML file. The value of each element in the XML file refers to the name of a term in the WS-Agreement specification. The example in Table 4.1 shows how to enable the code generation for the term named **WarehouseGT** in the agreement specification.

4.4.2 The Syntax for the Terms in the WS-Agreement Contracts

This section describes the domain specific syntax adopted in order to instantiate the generic contents defined by the Terms in WS-Agreement. The section is organized in tree main parts: Section 4.4.2.1 introduces the syntax that PUPPET uses in order to specify under which conditions a Term is applicable. Section 4.4.2.2 introduces the syntax that PUPPET uses in order to specify the extra-functional property the Term predicates about. Section 4.4.2.3 introduces the syntax that PUPPET uses in order to limit the scope of a Term only to some operations among all the ones that a Service exports.

[2] Respectivelly under Unix-like and Windows operating systems

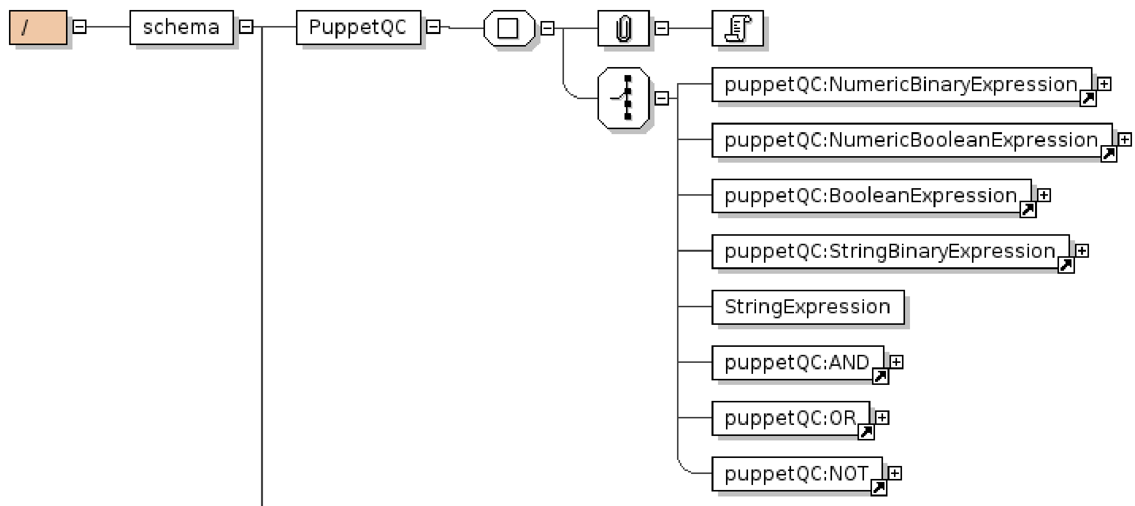

Figure 4.1: Expressions in the Qualifying Condition

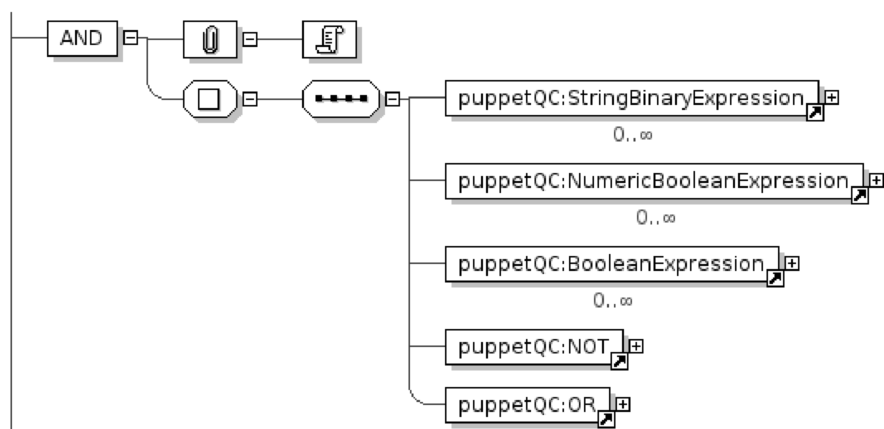

Figure 4.2: AND in the Qualifying Condition

4.4.2.1 Qualifying Conditions

In WS-Agreement the Qualifying Conditions of a Term may appear to express a precondition under which a Term holds [17]. In PUPPET , a Qualifying Condition can be formulated in terms of atomic expressions typed as Numeric, Boolean, or String (see Figure 4.1). The atomic expressions can be combined by means of the Boolean operators: AND (see 4.2), OR (see 4.3), and NOT (see 4.4).

Figure 4.5 depicts the elements that can be used in order to construct an atomic expressions. Also, for each operation type, it shows the operators supported in this release.

4.4.2.2 Service Level Objective

The specification of WS-Agreement defines the Service Level Objective (of type xsd:anyType), as the element expressing the condition that must be met to satisfy the guarantee Term [17].

This version of PUPPET handles constraints on the maximum admissible response time (i.e. service latency) and constraints on reliability (see Figure 4.6).

The time elapsed by a service when invoked (latency) is defined specifying the maximum admissible response time and a probability function describing how the delays are distributed. In this version, is it possible to define delays that are normally distributed or that follow the Poisson's law.

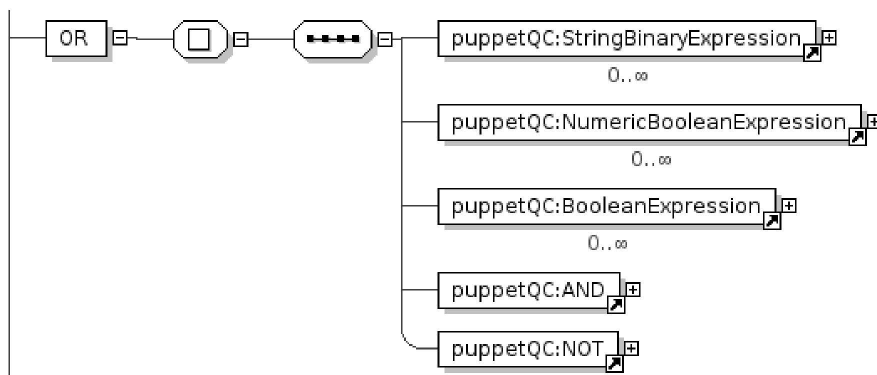

Figure 4.3: OR in the Qualifying Condition

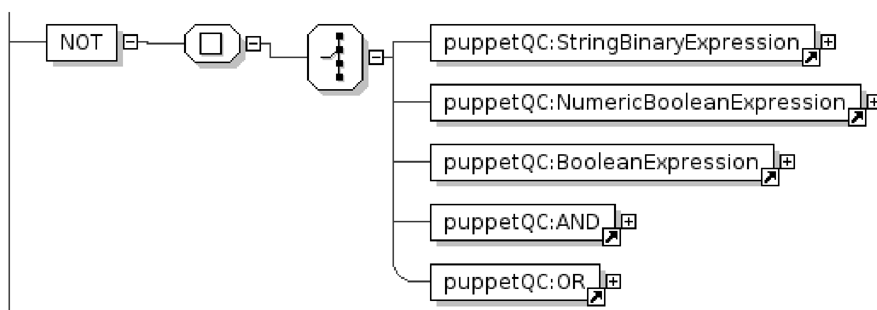

Figure 4.4: NOT in the Qualifying Condition

The constraints on the reliability of a Service are defined in terms of the maximum number of failure (**ReliabilityPerc** in Figure 4.6) in a given window of time. Also in this case, this release offer to describe the distribution of the failures in the window either as normal or as Poisson's.

4.4.2.3 Scope

The scope of a Term describes to what service element specifically a term applies. For example, a term might only apply to one operation of a Web service at a particular end point. According to the specification of WS-Agreement [17], the scope of a Term contains a **ServiceName** attribute and any other XML structure describing a sub-structure of a service to which the scope applies.

In this version of PUPPET it is possible to define the list of the operations affected by a specific Term as depicted in Figure 4.7. If a Term does not specify any scope, PUPPET would generate the emulation of the extra-functional property in all the operations exported by the service the Term refers to.

4.4.3 Writing an Agreement

In PLASTIC there are two possible way to write WS-Agreement specification for PUPPET . The former is to write it directly according to the indications given in [7]. The latter is to exploit the PLASTIC's editor of SLA as explained in the following.

The PLASTIC conceptual model [16] defines the reference SLA concepts adopted the in the project. This means that the specific implementations of the various environments should consider to manage at least the QoS annotations expressed in [16] and then refined in D1.2 [12].

According to the conceptual model, [10] defines with SLAng [20] an abstract syntax for the agreements. Such syntax would be instantiated in several concrete syntax. Each concrete syntax

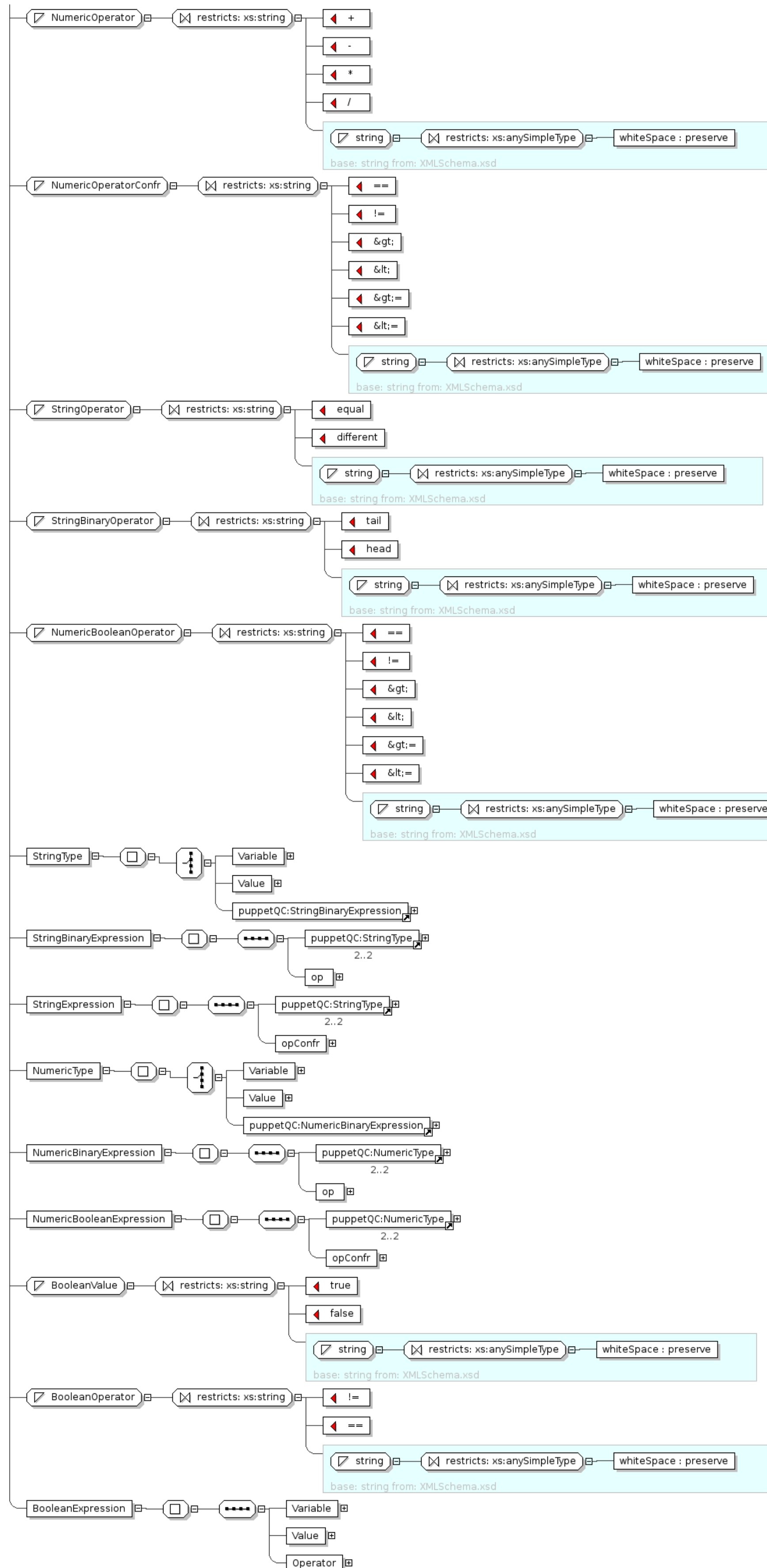

Figure 4.5: Operators in the Qualifying Condition

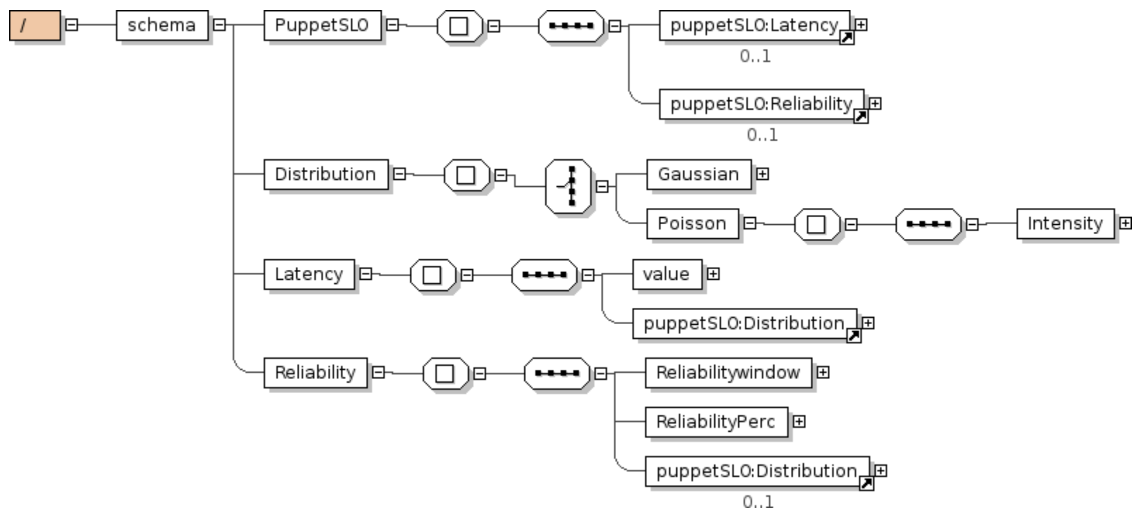

Figure 4.6: Extra-Functional Properties in the Service Level Objective

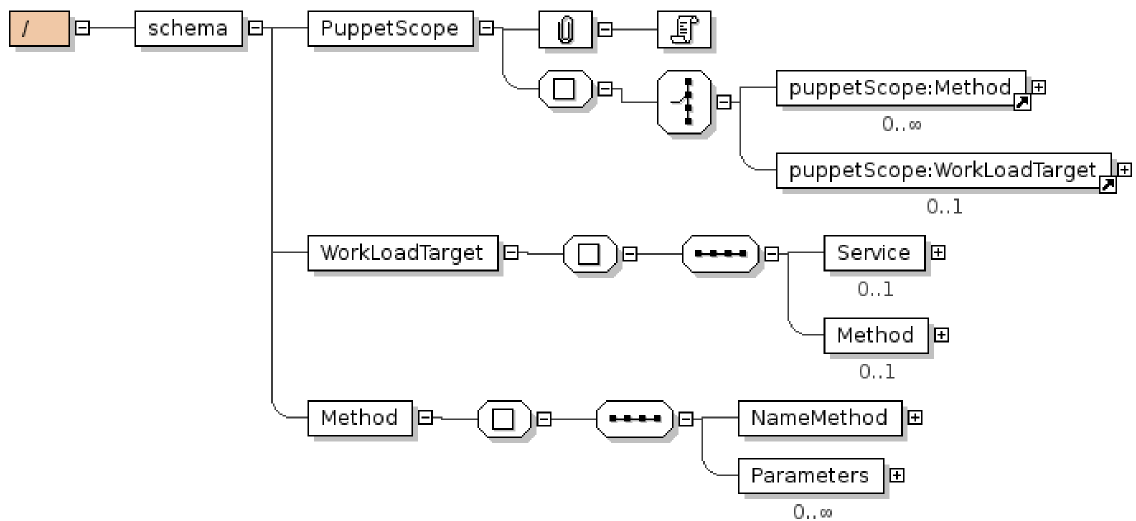

Figure 4.7: Defining the Scope of a Term

refers to a given kind of specification. For example in [10] the SLAng concepts were expressed using the HUTN (Human-Usable Textual Notation) as a concrete syntax.

The concrete syntax of SLAng could also refer to other languages for SLA specification. In that sense, a WS-Agreement specification could be seen as a concrete instantiation of the SLAng's abstract syntax. Note that such association is valid under the assumption that the two specifications predicate on the same kind of concepts.

In deliverable D2.2, the consortium presents a tool support for SLAng. It is an Eclipse-based editor for SLAng, in the form of an Eclipse plugin. The joint work between WP2 and WP4 developed an extension to the SLAng editor including a syntactic translation engine that generates WS-Agreement specification. The output produced by the plugin extension of the SLAng editor could be used as input for PUPPET .

4.4.4 Functional Behavior with Jambition

The integrated work of the team developing PUPPET and the team developing Jambition (see Chapter 2) in WP4 included in PUPPET (version PuppetD4.3) the features to generate stubs whose behavior conforms to both extra-functional contracts and a functional specifications.

As reported in Chapter 2, the functional behavior of a service in Jambition is modeled using a state machine called *Service State Machine* (SSM).

Enabling the **ambitionMode** in the INI configuration as specified in Section 4.3.2, PUPPET would include in the generated stubs the code emulating the functional behavior.

Specifically, the **ambitionMode** flag enables the inclusion in the source code of the stub of facilities used for the emulation of the functional behavior in Jambition. Listing 4.1 shows the definition of the Service State Machine (SSM) (line 4), the simulator that browses the SSM in order to emulates the correct functional behavior (line 9), and a private utility method (lines 11-30).

```
1  /*
2   * The SSM object.
3   */
4  private info.frantzen.testing.ssmsimulator.ssm.ServiceStateMachine aMbItIoNssm;
5
6  /*
7   * The simulator is generated
8   */
9  private info.frantzen.testing.ssmsimulator.SSMSimulator aMbItIoNsim;
10
11 private info.frantzen.testing.ssmsimulator.ssm.Message aMbItIoNfindSSMMessage(
12       info.frantzen.testing.ssmsimulator.ssm.ServiceStateMachine ssm,
13       info.frantzen.testing.ssmsimulator.ssm.MessageKind kind,
14       info.frantzen.testing.ssmsimulator.ssm.Operation op)
15       throws Exception {
16    java.util.HashSet<info.frantzen.testing.ssmsimulator.ssm.Message> messages = ssm
17          .getMessages();
18    for (java.util.Iterator it = messages.iterator(); it.hasNext();) {
19       info.frantzen.testing.ssmsimulator.ssm.Message m = (info.frantzen.testing.ssmsimulator.
            ssm.Message) it
20             .next();
21       if (m.getKind() != info.frantzen.testing.ssmsimulator.ssm.MessageKind.UNOBSERVABLE) {
22          if ((m.getKind() == kind) && (m.getOperation().equals(op))) {
23             return m;
24          }
25       }
26    }
27    throw new Exception(
28          "Cannot find the input SSM message belonging to the Operation "
29                + op.getName() + "!");
30 }
```

Listing 4.1: Attributes and Local operation included in the code

For each stub, the body of a default constructor is generated. In addition those line required in order to instantiate and to initialize both the simulator, and the SSM object are generated. Please note that once the stub is generated, some parameters required by the simulator have to be manually set by the user. Referring to Listing 4.2:

- at line 49 set the URL of the WSDL the Web Service that is going to be emulated exports
- at line 53 set the name of the Web Service that is going to be emulated as reported on the WSDL
- at line 57 set the port of the Web Service that is going to be emulated as reported on the WSDL
- at line 61 set the URL of the file with the specification of the SSM
- at line 76 set the URL of the treeSolver that the simulator uses in order to generate meaningful functional values
- at line 77 set the port of the treeSolver that the simulator uses in order to generate meaningful functional values

```
        /*
         * To initialise the Simulator, the following items are needed:
         */

        /*
         * The URL of the WSDL file
         */
        java.net.URL aMbItIoNWSDLUrl = new java.net.URL("Put␣here␣the␣URL␣of␣the␣Service's␣WSDL");
        /*
         * The name of the WSDL-Service
         */
        String aMbItIoNservice = "Put␣here␣the␣name␣of␣the␣Service␣as␣in␣the␣WSDL";
        /*
         * The name of the WSDL-Port
         */
        String aMbItIoNport = "Put␣here␣the␣port␣of␣the␣Service␣as␣in␣the␣WSDL";
        /*
         * The URL of the SSM Schema Instance
         */
        java.net.URL aMbItIoNSSMUrl = new java.net.URL("Put␣here␣the␣URL␣of␣the␣SSM␣Schema␣Instance
            ");

        /*
         * Now we can generate the SSM object. To do so, we use Zsolt's "Minerva" library
         */
        aMbItIoNssm = hu.soft4d.jessi.ssm.SSMHandler.generateSSM(
              aMbItIoNWSDLUrl, aMbItIoNSSMUrl, aMbItIoNservice, aMbItIoNport);
        /*
         * Before we can use the SSM in the simulator, the parsers have to be attached
         * to the switches
         */
        aMbItIoNssm.attachParsersToSwitches();
        /*
         * Next we generate the socket to the treeSolver.
         */
        String aMbItIoNsolverHost = "Put␣here␣the␣URL␣of␣the␣Solver";
        int aMbItIoNsolverPort = "Put␣here␣the␣Port␣the␣Solver";
        java.net.Socket aMbItIoNsolverSocket = new java.net.Socket(
              aMbItIoNsolverHost, aMbItIoNsolverPort);
        /*
         * The treeSolver sends a welcome message, we remove it from the stream
         */
        new java.io.BufferedReader(new java.io.InputStreamReader(
              aMbItIoNsolverSocket.getInputStream())).readLine();
        /*
```

```
45          * The simulator can use an external tool to display sequence diagrams
46          * of the messages exchanged. // I skip this here since this takes extra
47          * ressources/
48          */
49
50         /*
51          * The simulator needs a logger to log to
52          */
53         java.util.logging.Logger aMbItIoNlogger = java.util.logging.Logger
54                .getLogger("");
55
56         /* The simulator is generated */
57
58         aMbItIoNsim = new info.frantzen.testing.ssmsimulator.SSMSimulator(
59               aMbItIoNssm, aMbItIoNsolverSocket, aMbItIoNlogger);
60
61         /*
62          * If Double variables are used we assume this models money
63          * (experimental). In any case, do this:
64          */
65         info.frantzen.testing.ssmsimulator.types.ST_PseudoPosDouble.postPointLength = 2;
66         /*
67          * Now the Simulator is ready.
68          * ----------------------------------------------------------------------------
69          */
```

Listing 4.2: Code Included Into the Default Class Constructor

For each operation exported by the Web Service, PUPPET include in the correspondent method body the code emulating the functional behavior. Listing 4.3 shows the code line instantiating the local variables used in order to interact with the simulator. Note that both the name and the type of the operation match with the parameters generated at line 279.

```
1    public void orderShipment(int ref, services.Address adr) throws java.rmi.RemoteException {
2        long aMbItIoNinvocationTime = 0;
3        try {
4            aMbItIoNinvocationTime = System.currentTimeMillis();
5            /*
6             * Code Generated for Integration with Ambition
7             */
8            info.frantzen.testing.ssmsimulator.ssm.Operation aMbItIoNoperation = new info.frantzen.
                 testing.ssmsimulator.ssm.Operation("orderShipment", info.frantzen.testing.
                 ssmsimulator.ssm.OperationKind.ONEWAY);
9            info.frantzen.testing.ssmsimulator.ssm.Message aMbItIoNmessage = aMbItIoNfindSSMMessage(
                  aMbItIoNssm, info.frantzen.testing.ssmsimulator.ssm.MessageKind.INPUT,
                 aMbItIoNoperation);
10           info.frantzen.testing.ssmsimulator.ssm.Valuation aMbItIoNvaluation = new info.frantzen.
                 testing.ssmsimulator.ssm.Valuation();
11           java.util.ArrayList<info.frantzen.testing.ssmsimulator.ssm.InteractionVariable>
                 aMbItIoNmessageType = aMbItIoNmessage.getType();
12           java.util.Iterator aMbItIoNit = aMbItIoNmessageType.iterator();
13           info.frantzen.testing.ssmsimulator.ssm.InteractionVariable aMbItIoNvar;
```

Listing 4.3: Local Variables Configuration

Listings 4.4 shows an example of the set of lines generated for each parameter that any operation exports.

```
1            /*
2             * Generated Parameter 0
3             */
4            aMbItIoNvar = (info.frantzen.testing.ssmsimulator.ssm.InteractionVariable) aMbItIoNit.
                 next();
5            info.frantzen.testing.ssmsimulator.types.TypeInstance refInstance = new info.frantzen.
                 testing.ssmsimulator.types.ST_PosIntInstance(ref);
6            aMbItIoNvaluation.addSingleValuation(aMbItIoNvar.getName(), refInstance);
7
8            /*
9             * Generated Parameter 1
```

```
10          */
11          aMbItIoNvar = (info.frantzen.testing.ssmsimulator.ssm.InteractionVariable) aMbItIoNit.
                next();
12          Object[] aMbItIoNparameterValues = new Object[2];
13          aMbItIoNparameterValues[0] = new info.frantzen.testing.ssmsimulator.types.
                ST_StringInstance(adr.getFirstName());
14          aMbItIoNparameterValues[1] = new info.frantzen.testing.ssmsimulator.types.
                ST_StringInstance(adr.getLastName());
15          info.frantzen.testing.ssmsimulator.types.TypeInstance adrInstance = new info.frantzen.
                testing.ssmsimulator.types.ComplexTypeInstance(aMbItIoNparameterValues);
16          aMbItIoNvaluation.addSingleValuation(aMbItIoNvar.getName(),adrInstance);
```

Listing 4.4: The Generation of the Operation's Parameters

In the end, the last piece of code that Listing 4.5 shows concerns the interrogation to the functional simulator. Note that here the simulator can potentially spot a failure, namely when this message is not specified in the SSM. Here the generated stub is thus able to do also functional testing.

```
1           /*
2            * The valuation is ready, we can construct an instantiated message
3            */
4           info.frantzen.testing.ssmsimulator.ssm.InstantiatedMessage aMbItIoNim = new info.
                frantzen.testing.ssmsimulator.ssm.InstantiatedMessage( aMbItIoNmessage,
                aMbItIoNvaluation);
5
6           /*
7            * This instantiated message can now be given to the simulator. Note
8            * that here the simulator can potentially spot a failure, namely
9            * when this message is not specified in the SSM! In that sense,
10           * here we do testing.
11           */
12          aMbItIoNsim.processInstantiatedMessage(aMbItIoNim);
13       } catch (Exception genericException) {
14          throw new java.rmi.RemoteException(genericException.getMessage());
15       }
```

Listing 4.5: The Generation of the Operation's Parameters

In case the operation has to return a meaningful value, an additional set of code lines is added to the body of the operation. Specifically, when the simulator knows the input, it is possible to query it for a correct response. First we ask the simulator for all currently activated output transitions (line 158 at Listing 4.6). Out of all possible output switches, we randomly choose one and check if it has a solution. If yes, we take it. If not, we choose randomly the next one (lines 165-180 at Listing 4.6)).

```
1           /*
2            * Ok, the simulator knows the input. Now we need a functionally
3            * correct response to this call. We first ask the simulator for all
4            * currently activated output transitions.
5            */
6           java.util.ArrayList aMbItIoNoutputs = new java.util.ArrayList(aMbItIoNsim.
                getCurrentOutputSwitches());
7
8           /*
9            * Out of all possible output switches, we randomly choose one and
10           * check if it has a solution. If yes, we take it. If not, we choose
11           * randomly the next one.
12           */
13          boolean aMbItIoNnoSolutionFound = true;
14          info.frantzen.testing.ssmsimulator.ssm.InstantiatedMessage aMbItIoNnextOutput = null;
15          java.util.Random aMbItIoNrandom = new java.util.Random();
16          while (!aMbItIoNoutputs.isEmpty() && aMbItIoNnoSolutionFound) {
17             info.frantzen.testing.ssmsimulator.ssm.Switch aMbItIoNcandidate = (info.frantzen.
                   testing.ssmsimulator.ssm.Switch) aMbItIoNoutputs.get(aMbItIoNrandom.nextInt(
                   aMbItIoNoutputs.size()));
18
19             /*
```

```
20              * try to find a solution, if yes, fine, if not, remove the
21              * candidate
22              */
23             aMbItIoNnextOutput = aMbItIoNsim.findSolution(aMbItIoNcandidate);
24             if (aMbItIoNnextOutput == null)
25                aMbItIoNoutputs.remove(aMbItIoNcandidate);
26             else
27                aMbItIoNnoSolutionFound = false;
28          }
29          if (aMbItIoNnextOutput == null)
30             throw new Exception("Failure␣in␣SSM!␣No␣output␣for␣synchronous␣input␣specified!");
31          /*
32           * Ok, we have now a feasible and functionally correct output:
33           * nextOutput Before we send this output to the Service out there,
34           * we tell so to the simulator:
35           */
36          aMbItIoNsim.processInstantiatedMessageNoBackup(aMbItIoNnextOutput);
37
38          /*
39           * What is left to do, is to map this instantiated message back to a
40           * real returnValue.
41           */
42          info.frantzen.testing.ssmsimulator.ssm.Message aMbItIoNreturnMessage =
                aMbItIoNnextOutput.getMessage();
43          String aMbItIoNreturnVarName = ((info.frantzen.testing.ssmsimulator.ssm.
                InteractionVariable) aMbItIoNreturnMessage.getType().iterator().next()).getName();
44          info.frantzen.testing.ssmsimulator.ssm.Valuation aMbItIoNreturnValuation =
                aMbItIoNnextOutput.getValuation();
45          info.frantzen.testing.ssmsimulator.types.TypeInstance aMbItIoNreturnInstance =
                aMbItIoNreturnValuation.getSingleInstance(aMbItIoNreturnVarName);
46          String[] aMbItIoNarrayRepresentation = aMbItIoNreturnInstance.toString().split(",");
47          aMbItIoNreturnValue = new services.Quote((Double.valueOf(aMbItIoNarrayRepresentation[0])
                .doubleValue()),aMbItIoNarrayRepresentation[1],(Integer.valueOf(
                aMbItIoNarrayRepresentation[2]).intValue()),(Integer.valueOf(
                aMbItIoNarrayRepresentation[3]).intValue()));
48          /*
49           * Now send the returnValue back to the calling service. That's it.
50           */
```

Listing 4.6: The Generation of the Meaningful Return Value

Once a feasible and functionally correct output is found, before sending it to the calling Service, the simulator has to store the output we choose (line 188 at Listing 4.6). Thus, what is left to do, is to map the output message back to a real return value (line 194-199 at Listing 4.6).

For the sake of completeness, in the appendix is reported the Java source code emulating a Warehouse Web Service (see the example in Chapter 2). The whole code of the stub was automatically generated by Puppet with the **ambitionMode** enabled.

4.4.5 Example

In this section, an example scenario on how to describe a WS-Agreement specification for the eHealth application domain is presented. The scenario is inspired to the eHealth scenarios proposed by the PLASTIC industrial partners. We remind that the WS Agreement file could also be automatically generated by means of the SlangMon editor provided by the PLASTIC Platform (see [10] for details).

4.4.5.1 Scenario Description

The scenario is structured as depicted in Figure 4.8. Five web services are involved in this example:

WSPlastic : Is the Web Service that interfaces the current eHealth system with the new PLASTIC

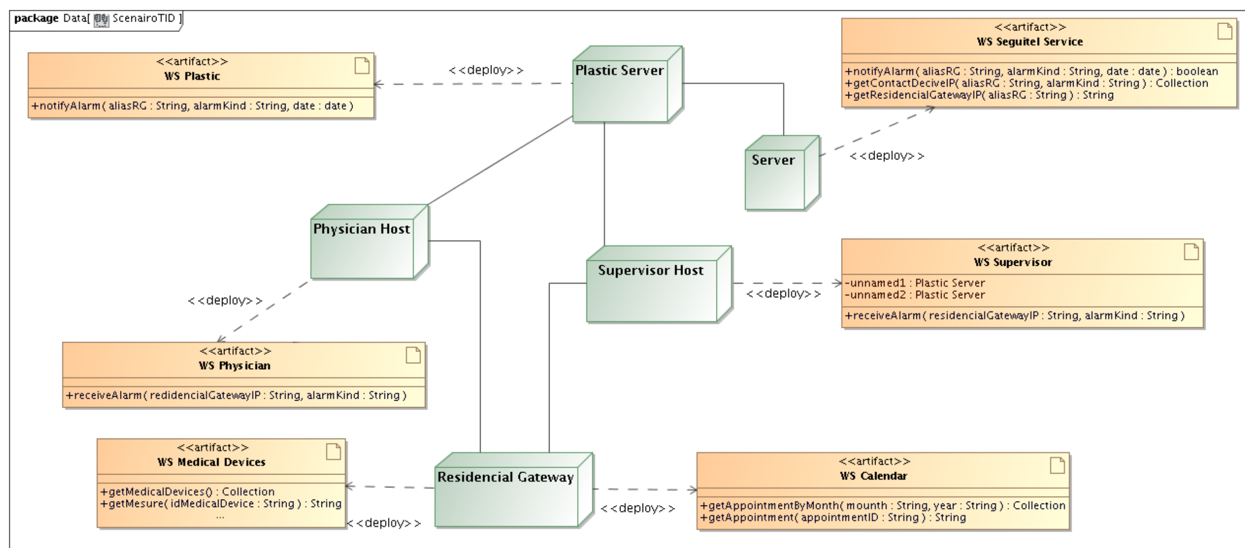

Figure 4.8: Scenario 3 Deployment Diagram

environment. In this scenario description we only refer to one of the possible operations and feature that it exports. Specifically the operation **notifyAlarm** is aimed at both collect and process the alarm messages coming from the eHealth part of the application that runs on PLASTIC. It is invoked when an alarm is raised. It takes as input the name of the Residential Gateway where the alarm comes from, the kind of the alarm and the date when it was raised.

In the scenario, WSPlastic represents the new service that have to be tested. Puppet here is used to automatically build the portion of the system that interacts with the service under test (i.e. WSPlastic)

WSSeguitelService : This Web Service represents the current eHealth application. It exports the following operations: **notifyAlarm**, **getResidentialGatewayIP**, **getContactDeciveIP**. **notifyAlarm**, takes as input the name of the Residential Gateway where the alarm comes from, the kind of the alarm and the date when it was raised. Both the other two exported operations take as input the logic name of a Residential Gateway. Thus **getResidentialGatewayIP** returns the IP address associated with the input label while **getContactDeciveIP** the list of IP addresses that should be contacted in case of alarm.

WSDoctor : It is the service deployed on the device that the doctor uses to interface with the eHealth system. The hosting device could be either a usual wired device as a PC or a mobile and wireless device such as a smart phone. The Web Service exports the **receiveAlarm** operation. Such operation takes as input both the IP address of the Residential Gateway where the alarm was raised and the kind of the alarm.

WSSupervisor : It is a service deployed on the device that a supervisor uses to interface with the eHealth system. The supervisor of a patient is the person that could assist the patient for non critical situation. In those cases that are classified as non critical, some kind of alarm could be forwarded to the supervisor instead of the doctor.

As for the doctor, also here the hosting device could be either a usual wired device as a PC or a mobile and wireless device such as a smart phone. The Web Service exports the **receiveAlarm** operation. Such operation takes as input both the IP address of the Residential Gateway where the alarm was raised and the kind of the alarm.

WSMedicalDevice : Each Residential Gateway controls several hardware medical devices. Each medical device is identified on the Residential Gateway by means of a unique identification code.

This Web Service interfaces the medical devices hosted by the Residential Gateway on the PLASTIC Network. It exports two operations. **getMedicalDevices** returns a collection of the controlled medical devices. To control an eHealth parameter monitored by a medical device, the web service has to be invoked on the **getMeasure** operation providing the id code of the medical device as input.

WSCalendar : Each Residential Gateway holds the lists of the periodic appointments that a supervisor plans with the patient. This Web Service interfaces the PLASTIC Network to this feature exporting the methods : **getAppointmentByMonth**, and **getAppointment**. The former gives information on the appointments already scheduled in a given month of a year. The latter is used to create a new one.

In this scenario, the Web Services described above are supposed to be deployed on different and distributed platforms. Specifically, WSPlastic is deployed on a **PlasticServer**, while the WSSeguitelService is supposed to run on the **eHealthServer**. Nevertheless, in principle the two server could be also the same.

4.4.5.2 Actors Interactions in the Scenario

This section provides a brief description on how the Web Services described in Section 4.4.5.1 interact. Figure 4.9 reports a UML Sequence Diagram describing these interactions.

When a alarm is notified to **WSPlastic**, as first step it forwards the event to **WSSequitelService** invoking the **notifyAlarm** method. The PLASTIC Web Service also invokes **getResidentialGatewayIP** obtaining the IP address of the Residential Gateway where the alarm was raised. Thus, it gets the list of the IP addresses that must be contacted invoking **getContactDeciveIP** on **WSSequitelService**. The obtained list depends on the kind of the received alarm.

Due to the kind of contact list the **WSPlastic** starts to invoke the appropriate Web Service. The Web Services to invoke are supposed to be deployed and reachable by means of the address list. This phase will continue until any confirmation of the handled alarm is obtained either by the doctor or the supervisor.

In the following, the case where the alarm kind is an emergency (i.e. “EMERGENCY”) is described. **WSPlastic** extracts an IP address from the address list. Then, it invokes **receiveAlarm** on **WSDoctor** using the IP address as endpoint. If the target Web Service decides to accept the alarm handling **WSPlastic** ends considering the problem solved. On the other hand, if **WSDoctor** does not accept to handle the notification or due to a QoS agreement violation (see Section 4.5.1), **WSPlastic** proceeds by extracting the next endpoint of the target Web Service.

When the doctor agrees on handle the alarm, he/she contacts **WSMedicalDevices** deployed on the referred Residential Gateway. Then, **WSDoctor** collects the list of the medical devices controlled by the Residential Gateway. The monitoring of the patient parameters is performed invoking the **getMeasure** method on those devices that are considered important for the clinical status.

In the following, the case where the alarm kind is not critical (i.e. “NOT CONFIRMATION”). **WSPlastic** extracts an IP address from the address list. Then, it invokes **receiveAlarm** on **WSSupervisor** using the IP address as endpoint. If the target Web Service decides to accept the alarm handling **WSPlastic** ends considering the problem solved. On the other hand, if **WSSupervisor** does not accept to handle the notification or due to a QoS agreement violation (see Section 4.5.1), **WSPlastic** proceeds extracting the next endpoint of the target Web Service.

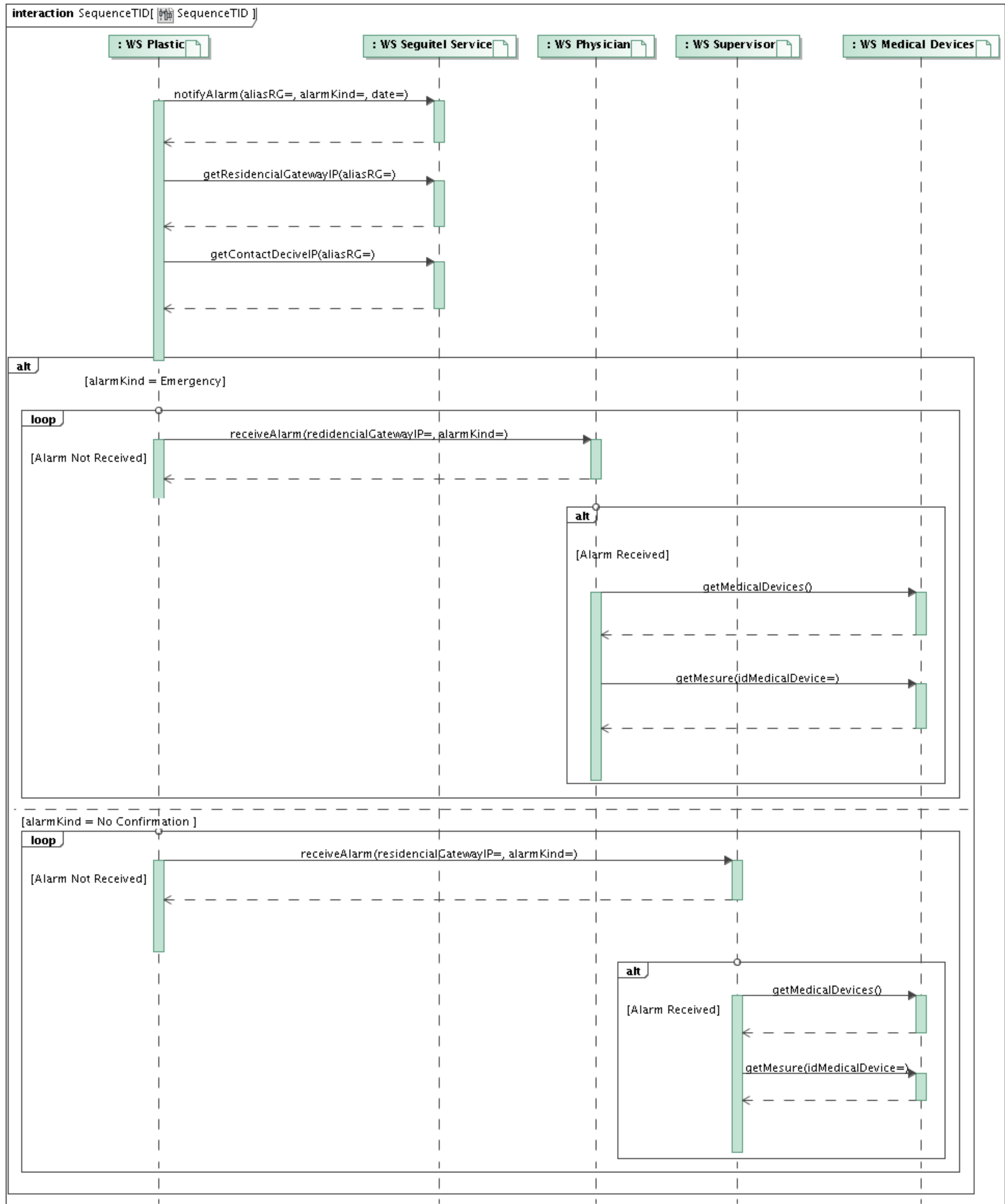

Figure 4.9: Scenario 3 Sequence Diagram

When the supervisor agrees on handle the alarm, he/she contacts **WSCalendar** deployed on the referred Residential Gateway. In the end, the **WSSupervisor** queries the Residential Gateway to schedule an appointment with the patient.

4.4.5.3 QoS Properties Definition

As described in Section 4.4.5.1, the Web Services considered in this scenario could be deployed on different machines. In particular, both **WSDoctor** and **WSSupervisor** could be deployed on a usual wired device as a PC or a mobile and wireless device such as a smart phone.

	Latency (msec)	**Reliability**		**Conditions**
WSSeguitelService:getContactDeciveIP	2000			alarmKind="Emergency"
WSSeguitelService:getContactDeciveIP	1000			alarmKind="No Confirmation"
WSDoctor:receiveAlarm	6000	WinSize Max Fails in Win	30000 5	deployedOn="MobileNode"
WSDoctor:receiveAlarm	2000	WinSize Max Fails in Win	30000 1	deployedOn="WiredServer"
WSSupervisor:receiveAlarm	10000	WinSize Max Fails in Win	30000 5	deployedOn="MobileNode"
WSSupervisor:receiveAlarm	6000	WinSize Max Fails in Win	30000 1	deployedOn="WiredServer"
WSMedicalDevice:getMeasure	3000			idMedicalDevice="device_1"
WSMedicalDevice:getMeasure	10000			idMedicalDevice="device_2"

Table 4.2: QoS Properties

Is it clear that the QoS properties of the service could depend on where the service in actually deployed. For example, if a the Web Service is deployed on a wireless node it is not always given that it is possible to reach it. On the other hand, if a Web Service is deployed on a standard PC it operates at higher performances than one deployed on a smart phone.

Moreover, a method can behave differently depending on the parameters it receives. For example, the processing time of **getMeasure** on **WSMedicalDevice** directly depends on the kind of measurement that is performed and the medical device that is used.

The QoS levels admitted in the scenario here considered are formalized in an agreement. Table 4.2[3] reports a short version of the extra functional properties that are supposed to be respected in the scenario. Time are given in milliseconds. In the appendix below, the listings reporting the complete version of the XML document expressing the agreement are reported.

4.5 Appendix

In the following we report the specification in WS-Agreement expressing the conditions and the constraints reported in Tab. 4.2. For the sake of completeness, the WSDL specifications of the services described in this chapter can be found in the appendix of the PUPPET User Manual available at http://plastic.isti.cnr.it/wiki/doku.php/tools. Also, Please refer to the same document for any information concerning the Java source code emulating a Warehouse Web Service.

4.5.1 WS Agreement

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:AgreementOffer xsi:schemaLocation="http://www.ggf.org/namespaces/ws-agreement"
```

[3] The values in this table has to be considered just as an example.

```
3                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4                   xmlns:xs="http://www.w3.org/2001/XMLSchema"
5                   xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-agreement"
6                   xmlns:puppetScope="http://setest0.isti.cnr.it/puppetScope"
7                   xmlns:puppetSLO="http://setest0.isti.cnr.it/puppetSLO"
8                   xmlns:puppetQC="http://setest0.isti.cnr.it/puppetQC"
9                   xmlns:ns="http://setest0.isti.cnr.it/puppet">
10    <wsag:Name>Telefonica_Example_Scenario_3</wsag:Name>
11
12    <wsag:Context />
13
14    <wsag:Terms>
15      <wsag:All>
16        <wsag:GuaranteeTerm wsag:Name="ContactDeciveIP-Term1"
17          wsag:Obligated="ServiceProvider">
18          <wsag:ServiceScope wsag:ServiceName="WSSeguitelService">
19            <puppetScope:PuppetScope>
20              <puppetScope:Method>
21  <!--              <NameMethod>getContactDeciveIP</NameMethod> -->
22                <NameMethod>getConnectedDeviceIP</NameMethod>
23              </puppetScope:Method>
24            </puppetScope:PuppetScope>
25          </wsag:ServiceScope>
26
27          <wsag:QualifyingCondition>
28            <puppetQC:PuppetQC>
29              <puppetQC:StringBinaryExpression>
30                <puppetQC:StringType>
31                  <Variable>alarmGender</Variable>
32                </puppetQC:StringType>
33
34                <op>equal</op>
35
36                <puppetQC:StringType>
37                  <Value>Emergency</Value>
38                </puppetQC:StringType>
39              </puppetQC:StringBinaryExpression>
40            </puppetQC:PuppetQC>
41          </wsag:QualifyingCondition>
42
43          <wsag:ServiceLevelObjective>
44            <puppetSLO:PuppetSLO>
45              <puppetSLO:Latency>
46                <value>2000</value>
47
48                  <puppetSLO:Distribution>
49                    <Gaussian>10</Gaussian>
50                  </puppetSLO:Distribution>
51              </puppetSLO:Latency>
52            </puppetSLO:PuppetSLO>
53          </wsag:ServiceLevelObjective>
54
55          <wsag:BusinessValueList>
56            <wsag:Penalty>
57              <wsag:AssessmentInterval>
58                <wsag:Count />
59              </wsag:AssessmentInterval>
60
61              <wsag:ValueExpression> 2 </wsag:ValueExpression>
62            </wsag:Penalty>
63          </wsag:BusinessValueList>
64        </wsag:GuaranteeTerm>
65        <wsag:GuaranteeTerm wsag:Name="ContactDeciveIP-Term2"
66          wsag:Obligated="ServiceProvider">
67          <wsag:ServiceScope wsag:ServiceName="WSSeguitelService">
68            <puppetScope:PuppetScope>
69              <puppetScope:Method>
70  <!--              <NameMethod>getContactDeciveIP</NameMethod> -->
71                <NameMethod>getConnectedDeviceIP</NameMethod>
72              </puppetScope:Method>
73            </puppetScope:PuppetScope>
```

```
74        </wsag:ServiceScope>
75
76        <wsag:QualifyingCondition>
77          <puppetQC:PuppetQC>
78            <puppetQC:StringBinaryExpression>
79              <puppetQC:StringType>
80                <Variable>alarmGender</Variable>
81              </puppetQC:StringType>
82
83              <op>equal</op>
84
85              <puppetQC:StringType>
86                <Value>No Confirmation</Value>
87              </puppetQC:StringType>
88            </puppetQC:StringBinaryExpression>
89          </puppetQC:PuppetQC>
90        </wsag:QualifyingCondition>
91
92        <wsag:ServiceLevelObjective>
93          <puppetSLO:PuppetSLO>
94            <puppetSLO:Latency>
95              <value>1000</value>
96
97                <puppetSLO:Distribution>
98                  <Gaussian>10</Gaussian>
99                </puppetSLO:Distribution>
100           </puppetSLO:Latency>
101         </puppetSLO:PuppetSLO>
102       </wsag:ServiceLevelObjective>
103
104       <wsag:BusinessValueList>
105         <wsag:Penalty>
106           <wsag:AssessmentInterval>
107             <wsag:Count />
108           </wsag:AssessmentInterval>
109
110           <wsag:ValueExpression> 2 </wsag:ValueExpression>
111         </wsag:Penalty>
112       </wsag:BusinessValueList>
113     </wsag:GuaranteeTerm>
114     <wsag:ExactlyOne>
115       <wsag:GuaranteeTerm wsag:Name="AlarmDoctor-Term1"
116         wsag:Obligated="ServiceProvider">
117         <wsag:ServiceScope wsag:ServiceName="WSDoctor">
118           <puppetScope:PuppetScope>
119             <puppetScope:Method>
120               <NameMethod>receiveAlarm</NameMethod>
121             </puppetScope:Method>
122           </puppetScope:PuppetScope>
123         </wsag:ServiceScope>
124
125         <wsag:QualifyingCondition>
126           <puppetQC:PuppetQC>
127             <puppetQC:StringBinaryExpression>
128               <puppetQC:StringType>
129                 <Variable>deployedOn</Variable>
130               </puppetQC:StringType>
131
132               <op>equal</op>
133
134               <puppetQC:StringType>
135                 <Value>MobileNode</Value>
136               </puppetQC:StringType>
137             </puppetQC:StringBinaryExpression>
138           </puppetQC:PuppetQC>
139         </wsag:QualifyingCondition>
140
141         <wsag:ServiceLevelObjective>
142           <puppetSLO:PuppetSLO>
143             <puppetSLO:Latency>
144               <value>6000</value>
```

```
145              <puppetSLO:Distribution>
146                 <Gaussian>10</Gaussian>
147              </puppetSLO:Distribution>
148            </puppetSLO:Latency>
149            <puppetSLO:Reliability>
150              <Reliabilitywindow>30000</Reliabilitywindow>
151              <ReliabilityPerc>5</ReliabilityPerc>
152              <puppetSLO:Distribution>
153                <Gaussian>100</Gaussian>
154              </puppetSLO:Distribution>
155            </puppetSLO:Reliability>
156          </puppetSLO:PuppetSLO>
157        </wsag:ServiceLevelObjective>
158
159        <wsag:BusinessValueList>
160          <wsag:Penalty>
161            <wsag:AssessmentInterval>
162              <wsag:Count />
163            </wsag:AssessmentInterval>
164
165            <wsag:ValueExpression> 2 </wsag:ValueExpression>
166          </wsag:Penalty>
167        </wsag:BusinessValueList>
168      </wsag:GuaranteeTerm>
169      <wsag:GuaranteeTerm wsag:Name="AlarmDoctor-Term2"
170        wsag:Obligated="ServiceProvider">
171        <wsag:ServiceScope wsag:ServiceName="WSDoctor">
172          <puppetScope:PuppetScope>
173            <puppetScope:Method>
174              <NameMethod>receiveAlarm</NameMethod>
175            </puppetScope:Method>
176          </puppetScope:PuppetScope>
177        </wsag:ServiceScope>
178
179        <wsag:QualifyingCondition>
180          <puppetQC:PuppetQC>
181            <puppetQC:StringBinaryExpression>
182              <puppetQC:StringType>
183                <Variable>deployedOn</Variable>
184              </puppetQC:StringType>
185
186              <op>equal</op>
187
188              <puppetQC:StringType>
189                <Value>WiredServer</Value>
190              </puppetQC:StringType>
191            </puppetQC:StringBinaryExpression>
192          </puppetQC:PuppetQC>
193        </wsag:QualifyingCondition>
194
195        <wsag:ServiceLevelObjective>
196          <puppetSLO:PuppetSLO>
197            <puppetSLO:Latency>
198              <value>2000</value>
199                <puppetSLO:Distribution>
200                  <Gaussian>10</Gaussian>
201                </puppetSLO:Distribution>
202            </puppetSLO:Latency>
203            <puppetSLO:Reliability>
204              <Reliabilitywindow>30000</Reliabilitywindow>
205
206              <ReliabilityPerc>1</ReliabilityPerc>
207
208              <puppetSLO:Distribution>
209                <Gaussian>100</Gaussian>
210              </puppetSLO:Distribution>
211            </puppetSLO:Reliability>
212          </puppetSLO:PuppetSLO>
213        </wsag:ServiceLevelObjective>
214
215        <wsag:BusinessValueList>
```

```
216          <wsag:Penalty>
217            <wsag:AssessmentInterval>
218              <wsag:Count />
219            </wsag:AssessmentInterval>
220
221            <wsag:ValueExpression> 2 </wsag:ValueExpression>
222          </wsag:Penalty>
223        </wsag:BusinessValueList>
224      </wsag:GuaranteeTerm>
225    </wsag:ExactlyOne>
226    <wsag:ExactlyOne>
227      <wsag:GuaranteeTerm wsag:Name="AlarmSupervisor-Term1"
228        wsag:Obligated="ServiceProvider">
229        <wsag:ServiceScope wsag:ServiceName="WSSupervisor">
230          <puppetScope:PuppetScope>
231            <puppetScope:Method>
232              <NameMethod>receiveAlarm</NameMethod>
233            </puppetScope:Method>
234          </puppetScope:PuppetScope>
235        </wsag:ServiceScope>
236
237        <wsag:QualifyingCondition>
238          <puppetQC:PuppetQC>
239            <puppetQC:StringBinaryExpression>
240              <puppetQC:StringType>
241                <Variable>deployedOn</Variable>
242              </puppetQC:StringType>
243
244              <op>equal</op>
245
246              <puppetQC:StringType>
247                <Value>MobileNode</Value>
248              </puppetQC:StringType>
249            </puppetQC:StringBinaryExpression>
250          </puppetQC:PuppetQC>
251        </wsag:QualifyingCondition>
252
253        <wsag:ServiceLevelObjective>
254          <puppetSLO:PuppetSLO>
255            <puppetSLO:Latency>
256              <value>10000</value>
257
258                <puppetSLO:Distribution>
259                  <Gaussian>10</Gaussian>
260                </puppetSLO:Distribution>
261            </puppetSLO:Latency>
262            <puppetSLO:Reliability>
263              <Reliabilitywindow>30000</Reliabilitywindow>
264
265              <ReliabilityPerc>5</ReliabilityPerc>
266
267              <puppetSLO:Distribution>
268                <Gaussian>100</Gaussian>
269              </puppetSLO:Distribution>
270            </puppetSLO:Reliability>
271          </puppetSLO:PuppetSLO>
272        </wsag:ServiceLevelObjective>
273
274        <wsag:BusinessValueList>
275          <wsag:Penalty>
276            <wsag:AssessmentInterval>
277              <wsag:Count />
278            </wsag:AssessmentInterval>
279
280            <wsag:ValueExpression> 2 </wsag:ValueExpression>
281          </wsag:Penalty>
282        </wsag:BusinessValueList>
283      </wsag:GuaranteeTerm>
284      <wsag:GuaranteeTerm wsag:Name="AlarmSupervisor-Term2"
285        wsag:Obligated="ServiceProvider">
286        <wsag:ServiceScope wsag:ServiceName="WSSupervisor">
```

```
      <puppetScope:PuppetScope>
        <puppetScope:Method>
          <NameMethod>receiveAlarm</NameMethod>
        </puppetScope:Method>
      </puppetScope:PuppetScope>
    </wsag:ServiceScope>

    <wsag:QualifyingCondition>
      <puppetQC:PuppetQC>
        <puppetQC:StringBinaryExpression>
          <puppetQC:StringType>
            <Variable>deployedOn</Variable>
          </puppetQC:StringType>

          <op>equal</op>

          <puppetQC:StringType>
            <Value>WiredServer</Value>
          </puppetQC:StringType>
        </puppetQC:StringBinaryExpression>
      </puppetQC:PuppetQC>
    </wsag:QualifyingCondition>

    <wsag:ServiceLevelObjective>
      <puppetSLO:PuppetSLO>
        <puppetSLO:Latency>
          <value>6000</value>

            <puppetSLO:Distribution>
              <Gaussian>10</Gaussian>
            </puppetSLO:Distribution>
        </puppetSLO:Latency>
        <puppetSLO:Reliability>
          <Reliabilitywindow>30000</Reliabilitywindow>

          <ReliabilityPerc>1</ReliabilityPerc>

          <puppetSLO:Distribution>
            <Gaussian>100</Gaussian>
          </puppetSLO:Distribution>
        </puppetSLO:Reliability>
      </puppetSLO:PuppetSLO>
    </wsag:ServiceLevelObjective>

    <wsag:BusinessValueList>
      <wsag:Penalty>
        <wsag:AssessmentInterval>
          <wsag:Count />
        </wsag:AssessmentInterval>

        <wsag:ValueExpression> 2 </wsag:ValueExpression>
      </wsag:Penalty>
    </wsag:BusinessValueList>
  </wsag:GuaranteeTerm>
</wsag:ExactlyOne>
<wsag:GuaranteeTerm wsag:Name="MedicalDevice-Term1"
  wsag:Obligated="ServiceProvider">
  <wsag:ServiceScope wsag:ServiceName="WSMedicalDevice">
    <puppetScope:PuppetScope>
      <puppetScope:Method>
        <NameMethod>getMeasure</NameMethod>
      </puppetScope:Method>
    </puppetScope:PuppetScope>
  </wsag:ServiceScope>

  <wsag:QualifyingCondition>
    <puppetQC:PuppetQC>
      <puppetQC:StringBinaryExpression>
        <puppetQC:StringType>
          <Variable>idMedicalDevice</Variable>
        </puppetQC:StringType>
```

```

            <op>equal</op>

            <puppetQC:StringType>
              <Value>device_1</Value>
            </puppetQC:StringType>
          </puppetQC:StringBinaryExpression>
        </puppetQC:PuppetQC>
      </wsag:QualifyingCondition>

      <wsag:ServiceLevelObjective>
        <puppetSLO:PuppetSLO>
          <puppetSLO:Latency>
            <value>3000</value>

              <puppetSLO:Distribution>
                <Gaussian>10</Gaussian>
              </puppetSLO:Distribution>
          </puppetSLO:Latency>
        </puppetSLO:PuppetSLO>
      </wsag:ServiceLevelObjective>

      <wsag:BusinessValueList>
        <wsag:Penalty>
          <wsag:AssessmentInterval>
            <wsag:Count />
          </wsag:AssessmentInterval>

          <wsag:ValueExpression> 2 </wsag:ValueExpression>
        </wsag:Penalty>
      </wsag:BusinessValueList>
    </wsag:GuaranteeTerm>
    <wsag:GuaranteeTerm wsag:Name="MedicalDevice-Term2"
      wsag:Obligated="ServiceProvider">
      <wsag:ServiceScope wsag:ServiceName="WSMedicalDevice">
        <puppetScope:PuppetScope>
          <puppetScope:Method>
            <NameMethod>getMeasure</NameMethod>
          </puppetScope:Method>
        </puppetScope:PuppetScope>
      </wsag:ServiceScope>

      <wsag:QualifyingCondition>
        <puppetQC:PuppetQC>
          <puppetQC:StringBinaryExpression>
            <puppetQC:StringType>
              <Variable>idMedicalDevice</Variable>
            </puppetQC:StringType>

            <op>equal</op>

            <puppetQC:StringType>
              <Value>device_2</Value>
            </puppetQC:StringType>
          </puppetQC:StringBinaryExpression>
        </puppetQC:PuppetQC>
      </wsag:QualifyingCondition>

      <wsag:ServiceLevelObjective>
        <puppetSLO:PuppetSLO>
          <puppetSLO:Latency>
            <value>10000</value>

              <puppetSLO:Distribution>
                <Gaussian>10</Gaussian>
              </puppetSLO:Distribution>
          </puppetSLO:Latency>
        </puppetSLO:PuppetSLO>
      </wsag:ServiceLevelObjective>

      <wsag:BusinessValueList>
```

```
429         <wsag:Penalty>
430           <wsag:AssessmentInterval>
431             <wsag:Count />
432           </wsag:AssessmentInterval>
433
434           <wsag:ValueExpression> 2 </wsag:ValueExpression>
435         </wsag:Penalty>
436       </wsag:BusinessValueList>
437     </wsag:GuaranteeTerm>
438
439   </wsag:All>
440  </wsag:Terms>
441 </wsag:AgreementOffer>
```

5 Weevil

5.1 Weevil Overview

Weevil simplifies running repeatable experiments on distributed systems by modelling the system under evaluation (SUE), the test environment (testbed), and the experiment workload. Two executables control an experiment:

1. *weevilgen* uses a simulator to automatically generate workloads given a simple example implementation of the system clients (actors).

2. *weevil*, given a configuration file representing the experiment model, will :

   (a) create start and stop scripts for each component
   (b) deploy the experiment binaries across the testbed
   (c) start experiment components and actors
   (d) collect component logs
   (e) clean the testbed

5.2 Technical info

Provider USI

Introduction Software to simplify the execution and collection of results of experiments on distributed testbeds.

Development status The complete software is available for download, current Version is 1.2.4

Intended audience Developers who intend to test a distributed system on a real environment

License OSS, both GPL 2.0 and 3.0 apply.

Language M4, C++, Shell Script, Make

Sw operating language Shell, SSH, (Java)

Environment (set-up) The following components need to be installed in order to compile and run Weevil:

- SSim discrete-event library[1]

Platform N/A

Download Weevil is available from the WP4 tools repository web site at URL:

`http://plastic.isti.cnr.it/download/tools`

Documents see WP4 tools repository web site and deliverables D4.1 and D4.2

Tasks N/A

Bugs N/A

Patches N/A

Contact `toffettg@lu.unisi.ch`

[1] available from http://www.inf.unisi.ch/faculty/carzaniga/ssim/index.html

5.3 Deployment

5.3.1 Install

Weevil is distributed as source code, as such the typical “configure” and “make” procedure has to be executed for installation.

The simplest way to compile the package is:

1. ‘cd’ to the directory containing the package’s source code, cd to the ‘build’ directory and type ‘../configure’ to configure the package for your system.

   The execution of weevil needs the SSim discrete-event library. So you need to have the library installed somewhere, make sure to pass the right path to the –with-ssim= option in the ‘configure’ command. Here is an example:

   ../configure –with-ssim=/usr/lib/ssim-1.5.0

2. Type ‘make’ to compile the package.

3. Type ‘make install’ to install.

By default, ‘make install’ will install the package’s files in ‘/usr/local/bin’, ‘/usr/local/man’, etc. You can specify an installation prefix other than ‘/usr/local’ by giving ‘configure’ the option ‘–prefix=PATH’. Additional configuration information is contained in the “readme” file distributed with the source code.

5.3.2 Configure

Weevil directly supports experiment deployment and execution. The overall process is depicted in Figure 5.1. Actions are represented by rectangles and are labeled by circled numbers. Input and output data for those actions are represented by ovals. Dark ovals represent input models provided by the engineer. White ovals represent control scripts and data files generated by Weevil. The cross-hatched ovals represent data generated by the SUE during an experiment. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts.

The following steps need to be taken to setup and conduct an experiment.

5.3.2.1 Weevil Configuration (Experiment Registration)

Create a name for the experiment (refered to as <*experiment name*> below), like ”`experiment`” in the siena example; add the name into ”*weevil.conf*” file after ”`weevil_EXPERIMENTS:=`”. You could have more than one experiment registered in ”*weevil.conf*” separated with space. To start a new line, \ is needed to end the current line. Weevil only conducts experiments that have been registered in ”*weevil.conf*”.

For example, in file “*weevil.conf*” of the “plastic-cbr” example we defined one experiment:

```
weevil_EXPERIMENTS  := MyExperiment
```

For each experiment, Weevil must be provided with a workload file (refer to Section 5.3.2.2) and an experiment configuration file named “<*experiment name*>*.m4*” including all the experiment configurations as illustrated in Section 5.3.2.3.

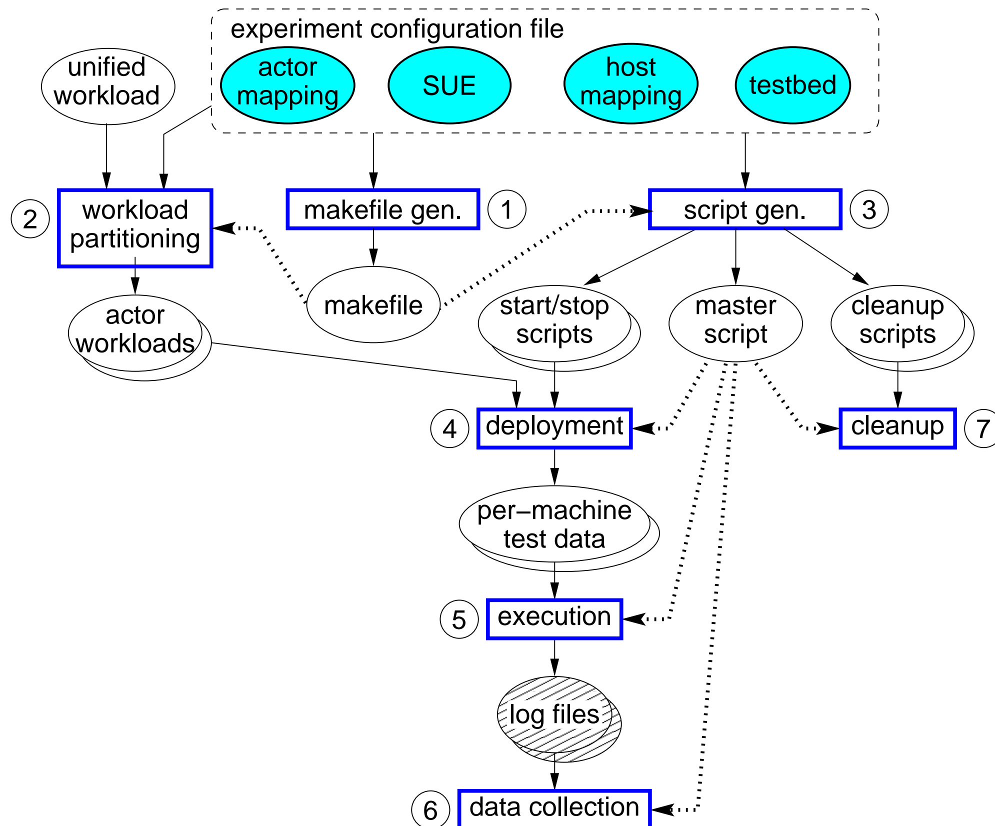

Figure 5.1: Weevil Experimentation Process

5.3.2.2 Workload File

You can generate the workload file with *weevilgen*. Of course, you can always use workloads from other workload generators or just use a real trace as the workload. But please note that the workload should be made up of workload lines with the following format:

```
event(<time stamp>, <workload process ID>, `<event content>')
```

5.3.2.3 Experiment Configurations

These configurations are represented by the dark ovals along the top of Figure 5.1. They are programmed in GNU m4 by parameterizing Weevil-defined declaration macros (Please refer to the "Weevil's Experiment Environment Declaration Macros"[2]) to instantiate elements of two conceptual models (SUE and testbed) and necessary mappings (Please refer to the "Weevil's Experiment Environment Conceptual Models"[3]). In other words, these declaration macros will define a set of macros serving as properties of an experiment. (Please refer to the "Weevil's Experiment Environment Declaration Macros" and the "Some Other Weevil-Defined Property Macros"[4]) for these property macros.) The order of the declarations does not matter. A macro can be used before it is defined as long as it is defined somewhere in the experiment configuration file. Weevil supports the engineer during this activity by performing extensive checks on the syntax and consistency of

[2]see doc/weevilexec/defs.html in the software package
[3]see doc/weevilexec/model.html in the software package
[4]see doc/weevilexec/macros.html in the software package

the configurations and by providing detailed error messages about any problems it encounters. Please refer to the user manual for a complete description of the available experiment configuration macros.

5.4 Tutorial

As a tutorial, in this section we use the “Siena” example distributed with the software package, which is also the introductory example in the Weevil manual. Please refer to the manual and the example files for complete reference. A more complex example (“plastic-cbr”) is included in the package distribution: the aim there is to test the content-based routing component of the PLASTIC middleware upon which PLASTIC-enabled services will be running. Additionally, Weevil is being used to test the context engine component of the PLASTIC middleware both as a standalone application accessed through SOAP-enabled actors, and as an integrated system in the context of the PLASTIC eBusiness scenario.

The aim of the example in this section is to introduce the concepts and models needed to specify an experiment run. In the test, 3 instances of a Siena server (a distributed publish-subscribe system) are started and three small Java applications (*Actors*) are used to impersonate clients subscribing predicates and publishing notifications.

5.4.1 Workload

Two automatically generated workload files are included:

- MyWorkload.wkld
- wkld_sequential.wkld

The file MyWorkload.wkld consists of a list of events to be executed by the actor components, a brief extract from it is shown in Figure 5.2.

```
event(100,C1,`SUB(0, `"i tedious < 91"')',,)
event(102,C3,`SUB(0, `"i tedious > 55"')',,)
event(103,C2,`PUB(0, `"i hedge 95 & i tedious 63"')',,)
```

Figure 5.2: An example of workload

Each event is composed of a relative execution time w.r.t. the experiment start, an identifier of the actor that will enact the event, and an application specific command that has to be interpreted by the actor. For instance, in our example of Figure 5.2, the first line states that at time T=100, actor C1 is to subscribe with predicate “tedious <91”.

Both workload files can be used as is for experiment runs, and provide an example of how to specify a generic workload. They have been automatially generated from MyWorkload.m4 and wkld_sequential.m4 using the command “weevilgen”. You might want to postpone the study of workload generation and skip to Section 5.4.2 to concentrate first on actual experiment runs.

5.4.1.1 Simulation-Based Workload Generation

Weevil’s simulation-based workload generation process supported by the package *weevilgen* is illustrated in Figure 5.3.

In detail, the following steps need to be taken.

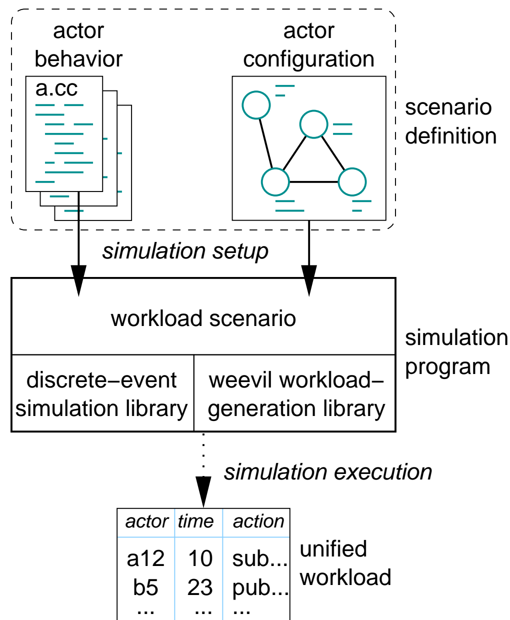

Figure 5.3: Simulation-Based Workload Generation

Weevilgen Configuration (Workload Registration) Create a name for the workload (refered to as <*workload name*> below), like "`MyWorkload`" in the Siena example; add the name into "*weevil.conf*" file after "`weevilgen_WORKLOADS:=`". You could have more than one workload registered in "*weevil.conf*" separated with space, as indicated in the apachesquid example. To start a new line, \ is needed to end the current line. Weevilgen only generates workloads that have been registered in "*weevil.conf*".

Then, to generate each workload, Weevil must be provided with programmed actor behavior models and an actor configuration file named "<*workload name*>*.m4*" including all the actor configurations.

Actor Behavior Model Programming One or more types of actor behaviors could be programmed in C++ with the support of Weevil's workload-generation library and SSim simulation library, and may therefore execute arbitrary functions and maintain arbitrary state. SSim is a discrete-event simulation library that supports message communication between simulated processes, so actor behavior programs may specify interactions with other actors. Weevil's workload-generation library extends the SSim's library by providing a workload-output method and convenient methods for dealing with processes by ids. Please refer to the *Weevil's workload-generation library* and the *SSim's simulation library* in the reference manual when programming your actor behavior model.

Actor Configurations After programming the actor behavior models, you can populate a scenario consisting of many actor instances specified in the actor configuration file named "<*workload name*>*.m4*". The actor behavior models and the actor configurations make up a workload scenario definition.

Figure 5.4 shows the portion of the Weevil conceptual model concerning the definition of workload scenarios. You need to parameterize a set of GNU m4 declaration macros in the actor configuration file. The order of the declarations does not matter. A macro can be used before it is defined as long as it is defined somewhere in the actor configuration file. Weevil supports the engineer during this activity by performing extensive checks on the syntax and consistency of the

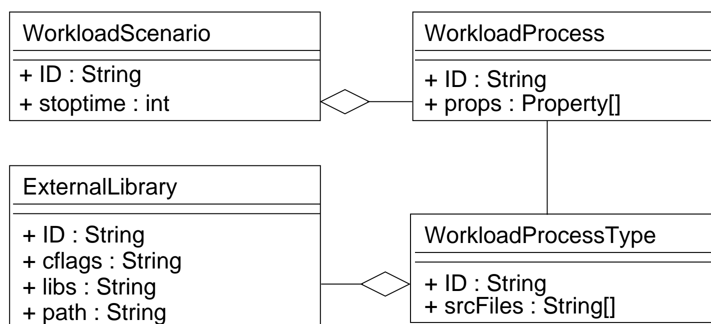

Figure 5.4: Workload Scenario Conceptual Model

configurations and by providing detailed error messages about any problems it encounters.

Workload generation When a workload has been registered in the *weevil.conf* file, actor behaviours have been implemented, and actor configurations have been provided, the command *weevilgen* can be used to transform the actors configuration file into a complete workload.

For instance, in the “Siena” example directory type: “weevilgen gen-MyWorkload”.

5.4.2 Experiment

The configuration file for the the actual experiment run is “experiment.m4”. It specifies the different features of an experiment such as: the testbed, the workload, the number and software for components and actors. Once more, we warmly recommend you read the Weevil manual (or the related conference papers) to get confident with the concepts and terminology.

The experiment starts 3 instances of a Siena server and uses three actors to impersonate the workload and provide stimuli to the servers.

Before running the experiment you should configure it to your particular setup. In particular, you should change the *WVL_SYS_Host* definitions to match hosts you can access via ssh.

Also we recommend you setup the testbed so that you can use your public key to ssh to remote hosts. Run ssh-add to open your keystore before launching the experiment run

To run the experiment:

1. ”cd” to the experiment directory
2. type ”weevil setup-experiment”
3. If the experiment is set up correctly type ”weevil run-experiment”

While the experiment is running you can follow the experiment evolution in a different console with the command:

```
tail -f experiment.runlog
```

The typical output of the Siena experiment is reported below.

```
deploying experiment files...
The deploy copy time is X seconds
starting S0...
starting S1...
```

```
starting S2...
monitoring hosts...
communicating with each host on the testbed to
      estimate the test start time...
The deploy time is 10 seconds
The test will start at 11/30/2007 16:4:59
The deploy and execution time is 23 seconds
killing processes...
stopping system components...
copying logs back...
```

Server logs and error output will be copied to the local directory, actors output is in weevil-parallel/actorstartoutput/*/ActorID.out

6 DynamoAOP

6.1 DynamoAOP overview

Dynamo-AOP is a framework for monitoring functional properties of external services which a BPEL [2] process interacts with, to realize a composite service. It is based on the conceptual model proposed in [4], but, with respect to the original design, its architecture is based on the dynamic aspectization of the BPEL engine executing the monitored service compositions, achieved by using AspectJ [13] as an AOP [14] language.

6.2 Technical info

Provider USI

Introduction Framework for monitoring functional properties of external services with which a BPEL process interacts

Development status Prototype completed

Intended audience Service aggregators/providers that describe service compositions in BPEL

License GPLv3 (open source)

Language Java, AspectJ

Environment (set-up) The following components need to be installed in order to run Dynamo-AOP:

- the Apache Tomcat servlet container (assumed to be installed in the directory `$TOMCAT-DIR` and running on port `7080`), available at http://tomcat.apache.org/
- the JBoss application server (assumed to be installed in the directory `$JBOSS-DIR` and running on port `8080`), available at http://www.jboss.org/
- the ActiveBPEL BPEL engine, available at http://www.active-endpoints.com
- (optional) a mail server, to support the `notify` recovery strategy.

The current version of Dynamo-AOP has been tested using Apache Tomcat ver. 5.5.23, JBoss Application Server ver 4.2, ActiveBPEL ver 2.1.
The following libraries are also required:

- ANTLRv2, available at http://www.antlr2.org/
- Apache Axis, available at http://ws.apache.org/axis/
- Apache XMLBeans, available at http://xmlbeans.apache.org/
- Apache Xerces 2, available at http://xerces.apache.org/xerces2-j/index.html
- Castor, available at http://www.castor.org
- Jakarta Commons Discovery, available at http://commons.apache.org/discovery/
- Jakarta Commons Logging, available at http://commons.apache.org/logging/
- CommonJ Timer and Work Manager for Application Servers, available at http://dev2dev.bea.com/wlplatform/commonj/twm.html

- Javamail, available at http://java.sun.com/products/javamail/
- JAXP, available at https://jaxb.dev.java.net/
- Jaxen, available at http://jaxen.org/
- JAX-RPC, available at https://jax-rpc.dev.java.net/
- JBoss EJB3, available at http://labs.jboss.com/jbossejb3/
- JBoss Web Services, available at http://labs.jboss.com/jbossws/
- Saxon XSLT processor, available at http://saxon.sourceforge.net/
- StAX, available at http://stax.codehaus.org/
- Sun Java Streaming XML Parser (JSR 173), available at http://java.sun.com/webservices/docs/1.5/sjsxp/ReleaseNotes.html
- WSDL for Java API, available at http://sourceforge.net/projects/wsdl4j
- XML Pull Parser (XPP), available at http://www.extreme.indiana.edu/xgws/xsoap/xpp/
- XSUL, available at http://www.extreme.indiana.edu/xgws/xsul/

Platform Java 5

Download both sources and binaries are available at http://plastic.isti.cnr.it/download/tools

Documents see WP4 tools repository web site and deliverables D4.1 and D4.2

Tasks N/A

Bugs N/A

Patches N/A

Contact domenico.bianculli@lu.unisi.ch

6.3 Deployment

6.3.1 Install

To actually install the Dynamo-AOP monitoring framework you have to:

- copy from the Dynamo-AOP distribution `ConfigurationManager.jar`, `HistoricalVariable.jar`, `MonitorLogger.jar` in the directory `$JBOSS-DIR/server/default/deploy`.
- copy from the Dynamo-AOP distribution `ae_rtbpel.jar`, in the directory `$TOMCAT-DIR/shared/libs`, overwriting the original ActiveBPEL file.
- copy from the Dynamo-AOP distribution `invoker.jar` in the directory `$TOMCAT-DIR/common/libs`.
- install a copy of `antlr-2.7.6.jar`, `antlrdebug_1.0.0.jar`, `jaxb-api.jar`, `jaxb-impl.jar`, `jaxb-xjc.jar`, `jaxb1-impl.jar`, `mail.jar`, `jsr173_api.jar`, `saxon8-dom.jar`, `saxon8-jdom.jar`, `saxon8-sql.jar`, `saxon8-xom.jar`, `saxon8-xpath.jar`, `saxon8.jar`, `xbean_path.jar`, `xbean.jar`, `xmlpublic.jar`, `aspectjrt.jar`, `xpp3-1.1.3.4.M.jar`, `xsul-2.0.9_2.jar`, `stax-1.1.1.jar` in the directory `$TOMCAT-DIR/common/libs`.

- install a copy of `axis.jar`, `commons-logging.jar`, `commons-discovery-0.2.jar`, `jaxrpc.jar`, `saaj.jar`, `wsdl4j-1.5.1.jar`, `castor-0.9.6-xml.jar`, `commonj-twm.jar`, `jaxen-1.1-beta-8.jar`, `ssaj-api.jar`, `xercesImpl.jar`, `saxon8-dom.jar` in the directory `$TOMCAT-DIR/shared/libs`.

6.3.2 Configure/Usage

1. launch the JBoss application server using the command `$JBOSS-DIR/bin/run.sh` and wait for the completion of the starting phase; check that JBoss is running by typing in your browser http://localhost:8080: you should see the start page of the application server.
2. launch the Apache Tomcat servlet container using the command `$TOMCAT-DIR/bin/catalina.sh run` and wait until the message "ActiveBPEL In-Memory Configuration Started" is displayed on the console. Check that Tomcat is running by typing in your browser http://localhost:7080. To check that ActiveBPEL is running, visit http://localhost:7080/BpelAdmin: you should see the ActiveBPEL administration tool. On it, click on Configuration and uncheck the box labelled "Validate Input/Output messages against schema".
3. deploy your BPEL process, as illustrated in ActiveBPEL user guide.
4. configure monitoring for the deployed process. This step assumes an interaction with the ConfigurationManager, which — in the first prototype of Dynamo-AOP — is done by directly accessing the API of the component, as shown in the "Dynamo Supervision Manager" application, also distributed within the framework (see next section).

6.4 Tutorial

In this section we will first review WSCoL, the language used to specify monitoring properties; then, we will describe the demo application bundled with Dynamo-AOP.

6.4.1 WS-CoL

WSCoL (Web Service Constraint Language) is the language used inside Dynamo-AOP to define monitoring properties; it is based on JML [15], with some conceptual and syntactical differences due to the adaption to the world of web services. Its main features are:

- Allowing to define and predicate on variables containing the data originating both within and outside the monitored BPEL process, and to retrieve data previously stored in a storage component.
- Using predefined variable functions for data manipulation.
- Using typical boolean, relational and arithmetic operators.
- Predicating on sets of variables through the use of the universal and existential quantifiers, and aggregate operators.

WSCoL allows to attach monitoring properties to the activities of a BPEL process that interact with external services. Properties can be pre- and post-conditions. Indeed, one can attach a pre-condition to an *invoke* activity, and a post-condition to an *invoke*, a *receive* and a *pick* activity.
All monitoring properties have three parameters:

- *priority*: it represents the “importance” of the rule and can be an integer ranging from 1 to 5. Each process can then define a threshold value that makes monitoring properties active or not, allowing for dynamically changing the amount of activities performed for monitoring the process.
- *validity*: it defines time constraints on *when* a monitoring properties should be considered.
- *trusted providers*: it’s a list of service providers for which monitoring is not necessary.

In its simplest definition, a monitoring property states relationships that must hold between variables. WSCoL supports three kinds of variables:

- *internal*: an internal variable corresponds to a datum that originates within the process being monitored. Usually, WSCoL internal variables define a form of *data extraction* from complex BPEL variables, by using XPATH [21] expressions. For example, to extract the value of a sub-element `easting` from the sub-element `start` of the element `parameters` of a complex BPEL variable named `getRouteIn`, one can use the notation `$(getRouteIn/parameters/start/easting)`, i.e. by concatenating the variable name prefixed by a dollar sign, with the XPATH expression matching the value of interest.
- *external*: An external variable indicates a monitoring datum that originates outside of the process in execution, such as a contextual datum. WSCoL assumes that the data source of an external variable can be queried through a web service interface and provides a function to invoke it: `(return<X> (W, O, Ins, Out))` where
  - `X` indicates the XSD type returned by the data source web service; it can be `Int`, `String`, `Bool`.
  - `W` represents the location of the WSDL of the data source web service.
  - `O` represents the name of the web method of the data source web service.
  - `Ins` represents the input message that one has to send when calling the data source web service.
  - `Out` represents an XPATH expression indicating the data extraction to apply to the output message returned from the data source web method, to get the desired value.
- *historical*: Historical variables consist of monitoring data that are related to previous activations of the Dynamo-AOP framework, related either to other processes or previous steps in the same process. Historical variables are defining using the `store` construct, as follows: `store $east_historical=($getRouteIn/parameters/start/easting);`.
  Previously stored historical variables can be retrieved using the `retrieve` function
  `(retrieve(pID,uID,iID,kID,type,alias,n))`
  where `pID` identifies the process family, `uID` is the user-id of the user who run the processes, `iID` identifies the instance within the process family, `kID` identifies an *invoke* activity in the process, `type` specifies if the historical variables was stored in a pre- or post-condition; `alias` is the name of the variable used in the `store` operation, `n` is the maximum number of results that should be returned by the query.

Moreover, variables may be aliased. This feature is provided both for allowing for less verbose expressions and for referring multiple times to a variable, whose value has been collected only once. Aliases are defined using the `let` keyword as shown below:
`let $east=($getRouteIn/parameters/start/easting);`.

Variables can be manipulated using data-type specific functions, invoked on the variable using the dot notation. Numeric functions include `abs()`, `ceiling()`, `floor()` and `round()`. The

available string functions are: `compare(string)`, `replace(pattern, replace)`, `substring(start, len)`, `length()`, `contains(string)`, `startsWith(string)`, `endsWith(string)`. For example, to get the length of the string value referred by the alias `east`, one can write `$east.length()`.

WSCoL allows to use universal and existential quantifiers to express constraints over sets of value. The syntax is: `(quantifier $alias_name in range_def; constraint_def)` where `quantifier` can be `forall` or `exists`. A universal quantifier indicates a parametric constraint that must be true for each and every value (at least one, in the case of the existential quantifier) the parameter can assume in a given range. `alias_name` and `range_def` defines respectively a variable alias and a finite range for the values it can assume; `constraint_def` defines the parametric constraint that must hold.

Aggregate operators allows to define assertions on set of values. The syntax is
`(operator $alias_name in range_def; assertion_def)`
where `operator` is one among `max`, `min`, `avg`, `sum`, `product`; `range_def` is a variable that returns a set of values, and `alias_name` is an alias that can be used as a parameter in `assertion_def`.

When a violation of a property is detected, some sort of recovery action should be performed. In our framework we have not investigated any specific recovery strategy but, by simplifying the work presented in [11], we just provides three simple primitives: `ignore`, which ignores the violation and allows the process to continue, `halt`, which stops the execution of the process, and `notify(msg, addr)`, which sends an email with the text `msg` to the recipient `addr`. These strategies can be structured using an `if-elseif-else` statement and/or boolean operators. For example, if a value is found to have a value below a certain threshold, we might want to differentiate the recovery strategy on the basis of the difference with respect to the threshold, as shown below:

```
if ($hRes < 80;)
   { halt() and notify("Low resolution, dba@localhost)}
elseif ($hres > 80; && $hRes < 120;)
  {ignore ()  and notify("Medium resolution, dba@localhost)}
else {ignore()}
```

The complete grammar of WSCoL is listed in the apppendix of this section.
Let's now see some examples of WSCoL properties.

```
Location: /process/sequence/invoke[@name='InvokeMap']
Supervision priority: 2
Monitoring rule:
   let $hRes=returnInt(
   'http://127.0.0.1:8080/ImageVerifierServiceBean?wsdl',
   'getHRes',
   $MapResponse/result,
   /Response/result);
   $hRes<= 150;
Recovering rule: {ignore()}
```

In this example, a property is associated to the *invoke* activity named *InvokeMap*. The `getHRes` web method of an external data collector service `ImageVerifierServiceBean` is called by passing it — as a parameter — the `result` element of the internal variable `MapResponse`. Once the service sends back its return message, the desired value is obtain through data extraction (`/Response/result`). This value is assigned to the `hRes` variable (actually, an alias), by means of the `let` statement. The actual property then checks if this variable is less than or equal to 150.

The recovery rule of this property is to ignore the fault and continue with the execution of the process. Moreover, a priority of 2 is associated with this property. The meaning is that every time the process is executed with a global priority of 2 or less the property is verified. On the other hand, if the global process priority is higher than or equal to 3 then the property is ignored.
Another sample property is

```
Location: /process/sequence/invoke[@name='InvokeGPS']
Supervision priority: 4
Monitoring rule:
   ($CoordResponse/result/easting).length==7) &&
   ($CoordResponse/result/easting).endsWith('E');
Recovering rule: {halt()}
```

The above property states that the `easting` element (supposed to be of type string) of the `result` element of the `CoordResponse` internal variable, must be seven characters long and that it must end with the character 'E'. The recovery rule just makes the process execution terminate. Moreover, a priority of 4 is associated to this property.

6.4.2 Demo

In the software distribution, you will find a complete web application that can be used to see how the monitoring framework works.

The application contains a BPEL process, `PizzaDeliveryCompany`, which you should deploy in the BPEL engine; at the end of the deployment you should see the process listed in the Deployed Processes section of the ActiveBPEL control panel, as shown in Figure 6.1. Before using the

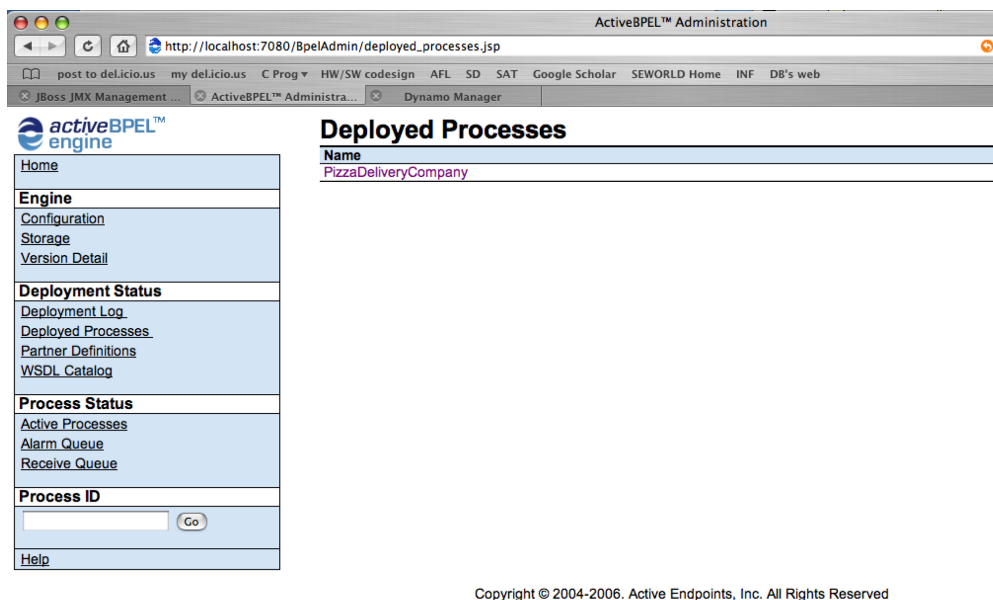

Figure 6.1: Process `PizzaDeliveryCompany` deployed successfully.

demo application, you should deploy some web services in the application server, by copying all the `*.jar` file from the `demo` directory to `$JBOSS-DIR/server/default/deploy`. In the same directory, you should also copy `dynamo.war`, which contains the "Dynamo Supervision Manager", a web application developed to to control the behavior of the monitoring framework; it can be reached at `http://localhost:8080/dynamo`.

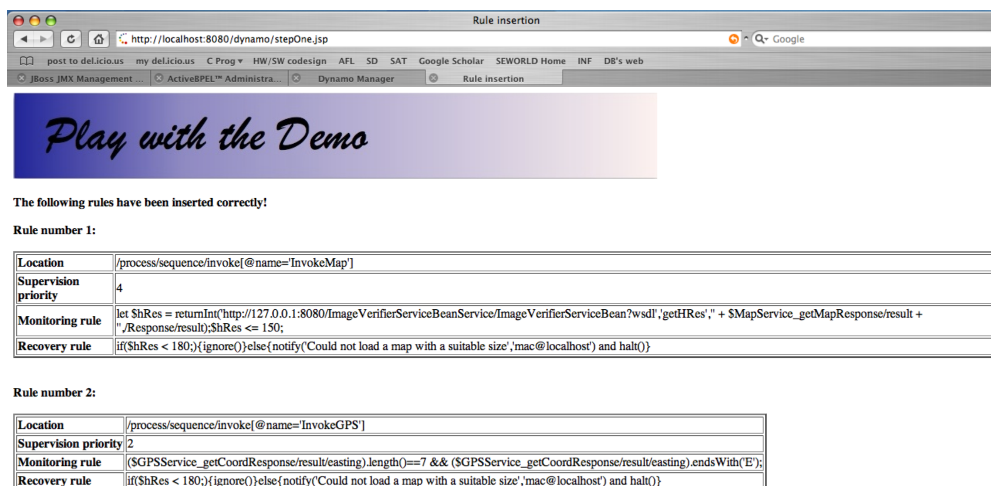

Figure 6.2: The result of inserting two monitoring rules.

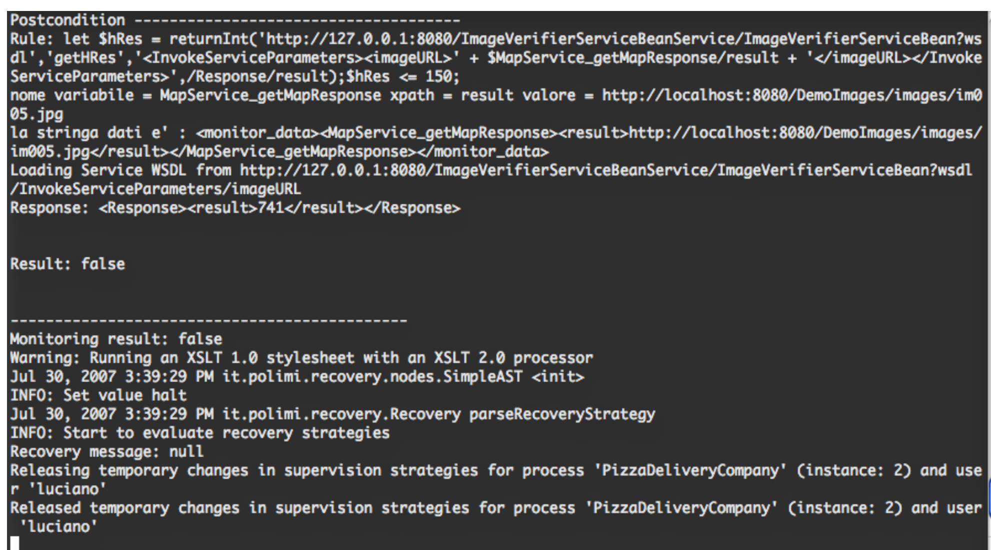

Figure 6.3: Output console of the monitored process

The main page of this application contains two links: one to set the properties of the monitored processes and the other to access the Dynamo-AOP demo, also available directly at http://localhost:8080/dynamo/DemoManager.jsp. This page contains the various steps through which you can interact with the BPEL process. The execution of step #1 will attach some monitoring rules to the process, as shown in Figure 6.2. The execution of the monitor process can then be monitored on the output console of Tomcat, as shown in Figure 6.3. The structure and the priority of the rules attached to a monitored process can be modified using the "Dynamo Supervision Manager", as shown in Figure 6.4.

6.5 Appendix

6.5.1 WS-CoL grammar

⟨analyzer⟩ ::⇒ ⟨rules⟩ | ⟨recovery⟩
⟨recovery⟩ ::⇒ ⟨complete-strategy⟩ | strategy

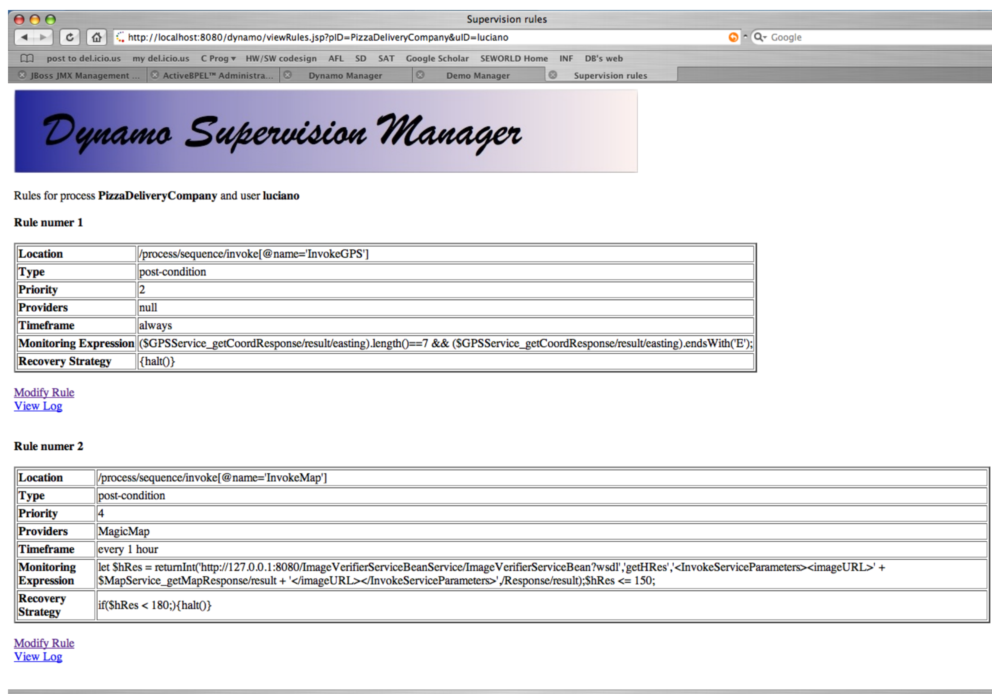

Figure 6.4: Monitoring rules modified with the "Dynamo Supervision Manager"

⟨complete-strategy⟩ ::⇒ ⟨ifStrategy⟩ ⟨elseIfStrategy⟩* ⟨elseStrategy⟩$^?$

⟨ifStrategy⟩ ::⇒ **if** ⟨condition⟩ ⟨strategy⟩

⟨elseIfStrategy⟩ ::⇒ **elseif** ⟨condition⟩ ⟨strategy⟩

⟨elseStrategy⟩ ::⇒ **else** ⟨strategy⟩

⟨condition⟩ ::⇒ **(** ⟨rules⟩ **)**

⟨strategy⟩ ::⇒ { ⟨steps⟩ }

⟨steps⟩ ::⇒ ⟨rec-step⟩ (**or** ⟨rec-step⟩)*

⟨rec-step⟩ ::⇒ ⟨actions⟩

⟨actions⟩ ::⇒ action (**and** ⟨action⟩)*

⟨action⟩ ::⇒ ⟨identifier⟩ **(** ⟨list⟩$^?$ **)**

⟨rules⟩ ::⇒ ⟨sub-rule⟩ ((**==>** | **<==** | **<==>**) ⟨sub-rule⟩)* **;**

⟨sub-rule⟩ ::⇒ ⟨and-expression⟩ (|| ⟨and-expression⟩)*

⟨and-expression⟩ ::⇒ ⟨equals-expression⟩ (**&&** ⟨equals-expression⟩)*

⟨equals-expression⟩ ::⇒ ⟨relational-expression⟩ ((**==** | **!=**) ⟨relational-expression⟩)$^?$

⟨relational-expression⟩ ::⇒ ⟨operator-expression⟩ ((> | >**=** | < | <**=**) ⟨operator-expression⟩)$^?$

⟨operator-expression⟩ ::⇒ ⟨basic-expression⟩ ((**+** | **-** | ***** | / | **%**) ⟨basic-expression⟩)*

⟨basic-expression⟩ ::⇒ ⟨dot-expression⟩ | ⟨variable⟩ | ⟨exists⟩ | ⟨forall⟩ | ⟨let⟩ | ⟨store⟩ | ⟨avg⟩ | ⟨min⟩ | ⟨max⟩ | ⟨sum⟩ | ⟨product⟩ | **true** | **false** | ⟨NUMBER⟩ | ⟨string-value⟩

⟨forall⟩ ::⇒ **(forall $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨exists⟩ ::⇒ **(exists $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨dot-expression⟩ ::⇒ ⟨variable⟩ **.** ⟨identifier⟩ **(** ⟨list⟩$^?$ **)**

⟨let⟩ ::⇒ **let $** ⟨identifier⟩ **=** ⟨sub-rule⟩

⟨store⟩ ::⇒ **store $** ⟨identifier⟩ **=** ⟨sub-rule⟩

⟨avg⟩ ::⇒ **(avg $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨sum⟩ ::⇒ **(sum $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨min⟩ ::⇒ **(min $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨max⟩ ::⇒ **(max $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨product⟩ ::⇒ **(product $** ⟨identifier⟩ **in** ⟨variable⟩ **;** ⟨sub-rule⟩ **)**

⟨variable⟩ ::⇒ **(** (⟨ivar⟩ | ⟨evar⟩ | ⟨hvar⟩) **)** | (⟨ivar⟩ | ⟨evar⟩ | ⟨hvar⟩)

⟨ivar⟩ ::⇒ **$** ⟨identifier⟩ ⟨xpath-expression⟩$^?$

⟨evar⟩ ::⇒ ⟨returnType⟩ **(** ⟨string-value⟩ **,** ⟨string-value⟩ **,** ⟨string-value⟩ **,** ⟨xpath-expression⟩ **)**

⟨returnType⟩ ::⇒ **returnInt** | **returnBool** | **returnString**

⟨hvar⟩ ::⇒ **retrieve (** ⟨string-value⟩ (**,** ⟨string-value⟩)$^?$ (**,** ⟨NUMBER⟩) **,** ⟨xpath-expression⟩ **,** ⟨NUMBER⟩ **, $** ⟨identifier⟩ (**,** ⟨NUMBER⟩) **)**

⟨alias⟩ ::⇒ **$** ⟨identifier⟩

⟨list⟩ ::⇒ ⟨sub-rule⟩ (**,** ⟨sub-rule⟩)*

⟨string-value⟩ ::⇒ ⟨sub-string-value⟩ (**+** ⟨sub-string-value⟩)*

⟨sub-string-value⟩ ::⇒ ⟨identifier⟩ | ⟨literal⟩ | ⟨variable⟩

⟨xpath-expression⟩ ::⇒ ⟨union-expression⟩

⟨location-path⟩ ::⇒ ⟨absolute-location-path⟩ | ⟨relative-location-path⟩

⟨absolute-location-path⟩ ::⇒ (/ | //) (⟨i-relative-location-path⟩ | ε)

⟨relative-location-path⟩ ::⇒ ⟨i-relative-location-path⟩

⟨i-relative-location-path⟩ ::⇒ ⟨step⟩ ((/ | //) ⟨step⟩)*

⟨step⟩ ::⇒ ((⟨axis⟩ | ε) ((((⟨identifier⟩ **:**)$^?$ (⟨identifier⟩ | *****))) | ⟨special-step⟩) ⟨predicates⟩*) | ⟨abbr-step⟩ ⟨predicates⟩*

⟨special-step⟩ ::⇒ **processing-instruction (** ⟨identifier⟩$^?$ **)** | (**comment** | **text** | **node**) **( )**

⟨axis⟩ ::⇒ ⟨identifier⟩ **::** | **@**

⟨predicate⟩ ::⇒ { ⟨predicate-expr⟩ }

⟨predicate-expr⟩ ::⇒ ⟨expr⟩

⟨expr⟩ ::⇒ ⟨or-expr⟩

⟨primary-expr⟩ ::⇒ ⟨variable-reference⟩ | **(** ⟨expr⟩ **)** | ⟨literal⟩ | ⟨number⟩ | ⟨function-call⟩

⟨literal⟩ ::⇒ ⟨LITERAL⟩

⟨number⟩ ::⇒ ⟨NUMBER⟩

⟨variable-reference⟩ ::⇒ **$**⟨identifier⟩

⟨function-call⟩ ::⇒ ⟨identifier⟩ **(** ⟨arg-list⟩$^?$ **)**

⟨arg-list⟩ ::⇒ ⟨argument⟩ (**,** ⟨argument⟩)*

⟨argument⟩ ::⇒ ⟨expr⟩

⟨union-expr⟩ ::⇒ ⟨path-expr⟩ (| ⟨path-expr⟩)*

⟨path-expression⟩ ::⇒ ⟨location-path⟩ | ⟨filter-expr⟩ ⟨absolute-location-path⟩$^?$

⟨filter-expr⟩ ::⇒ ⟨primary-expr⟩ ⟨predicate⟩$^?$

⟨or-expr⟩ ::⇒ ⟨and-expr⟩ (**or** ⟨and-expr⟩)*

⟨and-expr⟩ ::⇒ ⟨equality-expr⟩ (**and** ⟨equality-expr⟩)*

⟨equality-expr⟩ ::⇒ ⟨relational-expr⟩ ((**=** | **!=**) ⟨relational-expr⟩)$^?$

⟨relational-expr⟩ ::⇒ ⟨additive-expr⟩ ((< | <**=** | > | >**=**) ⟨additive-expr⟩)$^?$

⟨additive-expr⟩ ::⇒ ⟨mult-expr⟩ ((**+** | **-**) ⟨mult-expr⟩)$^?$

⟨mult-expr⟩ ::⇒ ⟨unary-expr⟩ ((***** | /) ⟨unary-expr⟩)$^?$

⟨unary-expr⟩ ::⇒ ⟨union-expr⟩ | **-** ⟨unary-expr⟩

6.5.2 Architecture

Figure 6.5 depicts the Dynamo-AOP monitoring framework, by illustrating the dependencies existing between the various components and the technologies used in the implementation. The Configuration Manager is a storage component for all the properties that have to be monitored. The ActiveBPEL engine is a modified version of ActiveBPEL [1] in which we embed monitoring. This is achieved by following an aspect oriented programming approach [14]. The engine is a Java program in which we weave the cross-cutting monitoring features via AspectJ [13]. ActiveBPEL works by creating an internal tree representation of the process being executed. In this tree, each node represents a single BPEL activity in the process definition, and is an appropriate extension of the `AEActivityDefinition` class. Each node contains the information necessary to perform the particular activity it is associated with. At run time, the tree is visited and the definition classes are used by the engine to instantiate appropriate `AEActivityImpl` classes, all of which implement a common interface. Amongst other things, this interface provides an `execute` method where the activity's primary action is performed. For example, a *scope* activity will set up its internal variables, while an *invoke* activity will perform the appropriate external invocation. To perform

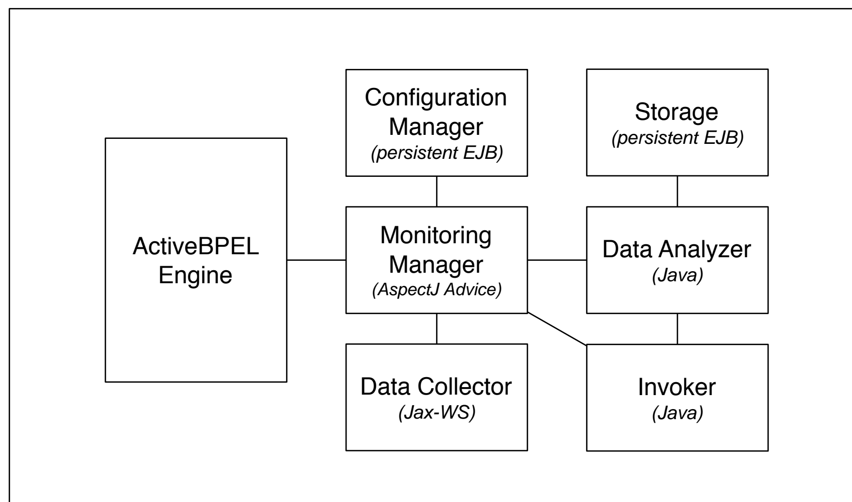

Figure 6.5: Dynamo-AOP components architecture

monitoring, we intercept the process after the `execute` method is called for the various BPEL activities. These are the points where the Monitoring Manager (implemented as an AspectJ advice) is activated. Its main responsibility is data collection, both from within the process and from the outside world, through the Data Collector. The collected data, together with the monitored property, are sent to the Data Analyzer, which first retrieves external data and/or historical variables by calling the Invoker and the Storage, respectively, and then analyzes the property, passing the result of the evaluation back to the Monitoring Manager. At this point, before returning control to the process, the Monitoring Manager executes the recovery code if a monitoring rule has been violated.

7 SLAngMon

7.1 SLAngMon Overview

SLAngMon is an Eclipse plugin to generate Java monitors for Service Level Agreements defined using the SLAng language. We refer to Deliverable 2.1 for a detailed description of SLAng.

SLAngMon implements the automata-based technique presented in D4.1 to monitor agreements by introducing AXIS handlers to intercept messages exchanged and to check their conformance with respect to the established SLAs.

SLAngMon is integrated into the SLA editor described in D2.2 and D2.3; we refer to these documents for a detailed description of how to create and manage a Service Level Agreement in SLAng.

7.2 Technical info

Provider University College London.

Introduction SLAngMon is a tool for the automatic generation of monitors from Service Level Agreements.

Development status The Eclipse plug-in is available for download as part of the SLAng

Intended audience Developers who intend to implement monitors for Service Level Agreements.

License Mozilla Public Licence v.1.1 (MPL).

Language Java, EMOF/OCL, ant.

Environment (set-up) The following components are needed to compile SLAngMon:

- Eclipse 3.2 or greater.
- UCL MDA tools, available from http://uclmda.sourceforgege.net

Platform Eclipse. Operating System(s): all

Download The tool is available as part of the SLAng editor plug-in, available from: http://www-c.inria.fr/plastic/workpackage/wp2

Documents See D2.1 and D2.2 for a description of SLAng and its editor (Eclipse plug-in). Further documentation is available in D4.1 and D4.2 and from the website.

Tasks A binary distribution to avoid the compilation process and dependency problems is currently under development; please contact f.raimondi@cs.ucl.ac.uk for further information.

Bugs N/A

Patches N/A

Contact f.raimondi@cs.ucl.ac.uk

7.3 Deployment

7.3.1 Install

These are the steps required to install SLAngMon:

1. Open Eclipse.
2. Add the following CVS repositories:

   ```
   Host: uclmda.cvs.sourceforge.net
   CVS path: /cvsroot/uclmda
   User: anonymous (no password)
   ```

   In HEAD, checkout the following: UCL, GTL, EMOFOCL2, EMOFOCLPlugin
3. Compile UCL:
   - Rename project.properties.example in project.properties and edit
   - Execute build.xml
4. Compile GTL:
   - You need to obtain gnu.regexp-1.1.4.tar.gz, unzip, and place it in the lib/ directory (the archive can be obtained by looking for it in Google).
   - Execute build.xml
5. Compile EMOFOCL2
   - Rename project.properties.example in project.properties and edit
   - Execute build.xml
6. Compile EMOFOCLPlugin
   - Rename project.properties.example in project.properties and edit
   - Execute build.xml
7. Create a new general project with name SLAng
   - Unzip SLAng.zip and import everything into new project SLAng
   - Edit project.properties
   - Run ant target ”clean”
   - Run ant target ”all”
8. Create a new general project with name SLAngTA
   - Unzip SLAngTA.zip and import everything into new project SLAngTA
   - Edit project.properties
   - Run ”all”

The plugin is at this point compiled and ready to be used.

7.4 Tutorial

SLAngMon output includes

- Java classes implementing the automata corresponding to checkers for SLAs.
- Java classes extending AXIS Basic Handlers, which should be invoked by appropriately modifying the chain of invocations.

For the purposes of this document, it is assumed that the reader is familiar with the following technologies:

- Apache, Tomcat, and AXIS.
- Eclipse and Eclipse plugins (with Ant builds)
- Meta-modelling, EMOF and OCL: please see Deliverables 6.2 and 2.1 for further details.

Usage. After compiling the software as described in the previous section, right-click on SLAngTA, Run As → Run... Run as Eclipse application. A new Eclipse instance should start. In this new instance, create a new General project. Then, create a new file, for instance `test.slangtaxmi` (notice: it is important to use `.slangtaxmi` extensions to activate the plugin). As an example, complete an InputThroughput clause, right click, and choose a destination directory. An example of the plugin at this point is depicted in Figure 7.1. This will generate a directory structure. The actual checker (the automaton) is generated in server/tautils. The handler for AXIS is generated in server/slamonitor

The generated java files can be compiled using the appropriate classpath for AXIS libraries. Copy the files and their container directories in `$AXIS_HOME/classes`

To install the handler, modify the file `$AXIS_HOME/server-config.wsdd`, for instance:

```
<handler name="slamonitor" type="java:slamonitor.SLAMonitor">
  <parameter name="filename" value="SLAMonitor.log"/>
  <parameter name="wsdlURL" value="/axis/SLAMonitorService-impl.wsdl"/>
  <parameter name="scope" value="Session"/>
  <parameter name="serviceName" value="SLAMonitorService"/>
  <parameter name="namespace"
     value="http://tempuri.org/wsdl/2001/12/SOAPMonitorService-impl.wsdl"/>
  <parameter name="portName" value="Demo"/>
</handler>
```

Finally, add the handler to you service (in `server-config.wsdd`):

```
<service name="MyService" provider="java:RPC">
    <requestFlow>
       <handler type="slamonitor"/>
    </requestFlow>
    <responseFlow>
       <handler type="slamonitor"/>
    </responseFlow>
[...]
</service>
```

The handler is now fully operational. Violations of the SLA clause are reported in the file `SLAMonitor.log`.

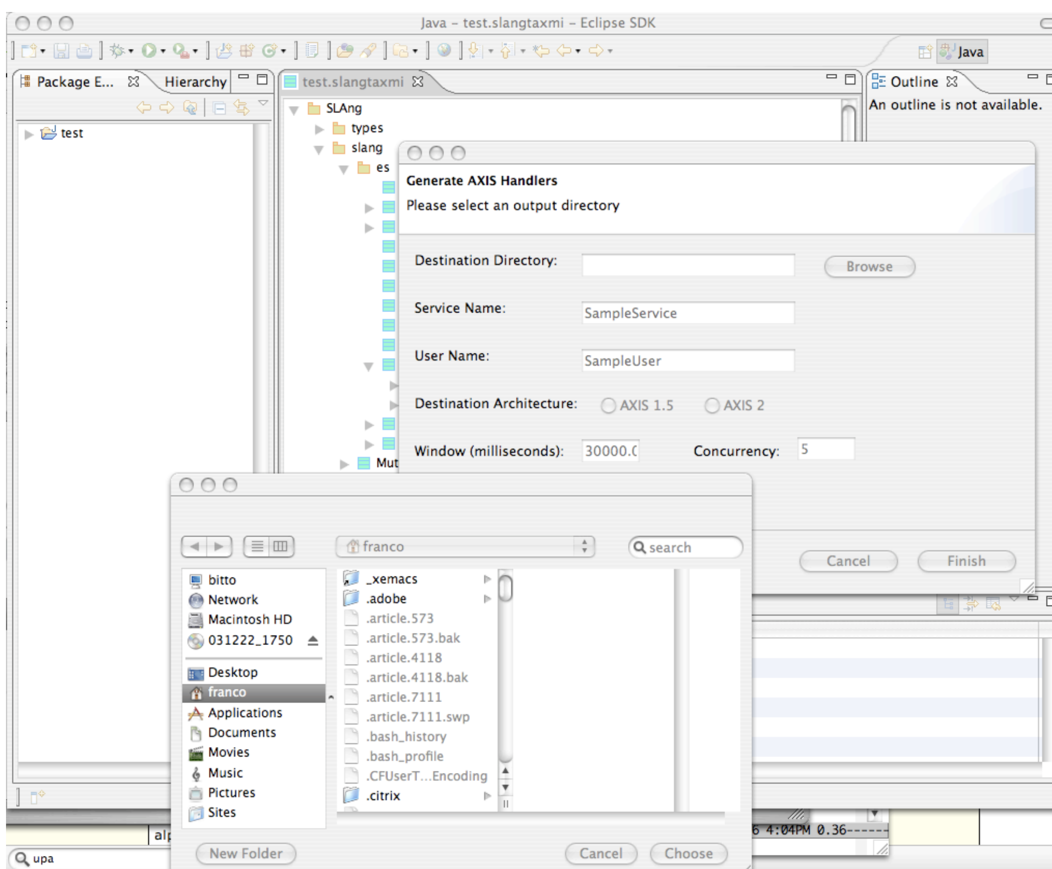

Figure 7.1: Plugin screenshot for SLAngMon: generation of checkers.

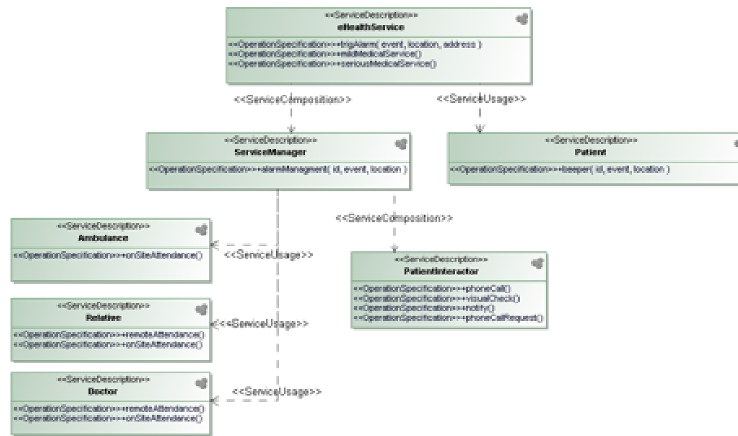

Figure 7.2: Service description diagram for the eHealth scenario

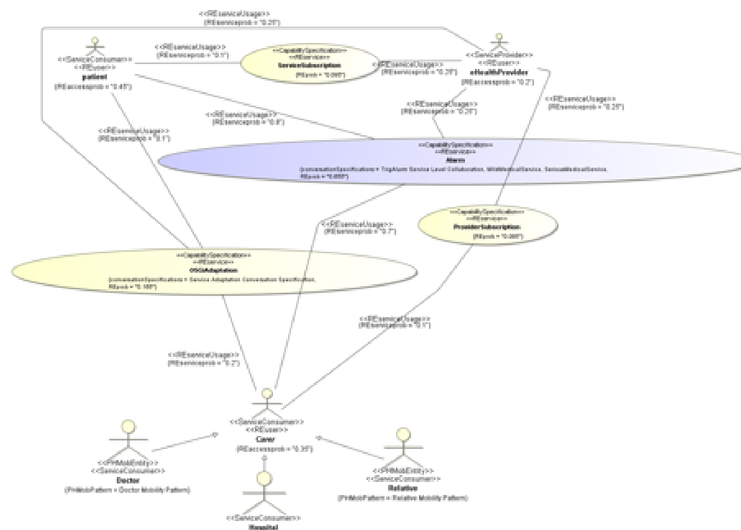

Figure 7.3: Use case service description diagram for the eHealth scenario

7.4.1 Demo

In this section we present a short demo on how to generate and use the monitors for an SLA between a patient and the eHealth service provider in the eHealth scenario. We refer to Deliverable 2.1 for a detailed description of the scenario.

For the purposes of this demo, we only briefly review the scenario in Figure 7.2 and 7.3.

A number of actors are involved in the scenario:

- A patient
- A number of "carers", such as a Doctor, a Relative, the Ambulance.
- A e-Health service provider, responsible for the communication between patients and carers.

Figure 7.4 depicts part of an SLA between the e-Health Provider and the Patient. The XMI code corresponding to this SLA in SLAng is presented in Section 7.5.3. In particular, in this example a latency clause is expanded for the operation `trigAlarm` invoked by the Patient and provided by the e-Health Provider: it is required that the maximum response time for this kind of requests is 10

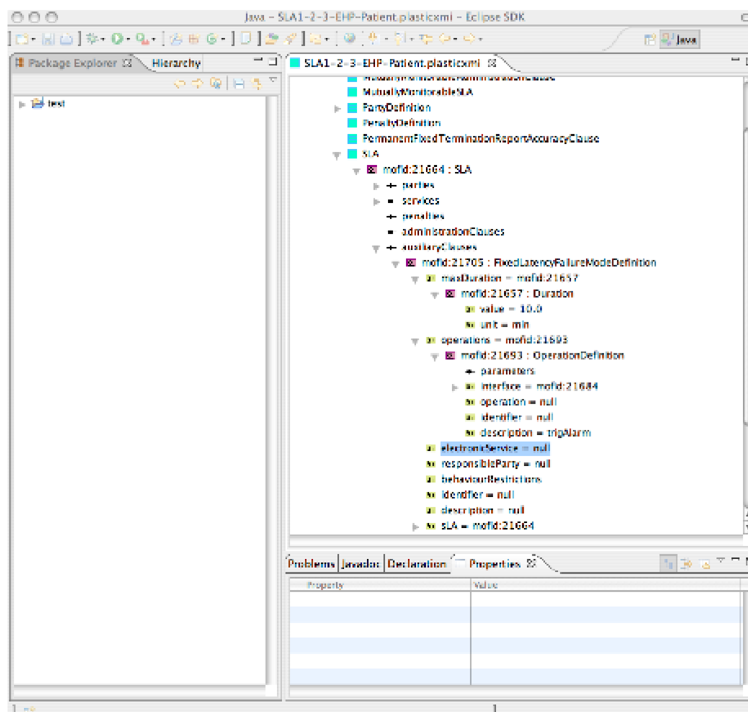

Figure 7.4: Latency clause for e-Health Provider and Patient

minutes: in this time window the e-Health provider has to evaluate how serious is the alarm (false alarm, medium, high).

By right-clicking on the clause in the SLAng editor, a menu as the one depicted in Figure 7.1 appears, and from here it is possible to generate Java files for monitoring this requirement. Excerpts of the Java code for the handler are reported in Figure 7.5.

This code can be compiled without modification by including the relevant AXIS libraries, e.g.:

```
javac -cp $AXISCLASSPATH SLAMonitor.java
```

where `$AXISCLASSPATH` is an environment variable containing the appropriate references to AXIS libraries. The generated .class file is then placed in the directory structure of the AXIS server, as described in the Tutorial above. When Tomcat is restarted, the appropriate service is monitored for its latency and violations are reported to a text file in the directory structure of AXIS, named `SLAMonitor.log` when default values are used. The lines of this file have the form:

```
Calling latency checker at time 1196198201762
No transition found, resetting at T = 1196198201765
```

These two lines mean that the latency checker was invoked with the values reported (in millisecond). “No transition found” means that no violation occurred (indeed the response was only 3 milliseconds in the case of this sample service). A violation line would report

```
VIOLATION DETECTED at T = 1196198201765
```

As mentioned at the beginning of this section, the behaviour in case of violations can be modified easily, both in the automatic generator inside the editor, or even after the generation of the Java code. We refer to Section 7.5.1 for an overview of the code structure.

```
/* Excerpts from the automatically generated AXIS handler */

public class SLAMonitor extends BasicHandler {

    /* private variables declared here [...] */

    public void invoke(MessageContext msgContext) throws AxisFault
    {

        try {

            /* Set-up the environment [...] */

            /* Read the messages: */
            Message reqmsg = msgContext.getRequestMessage();
            Message respmsg = msgContext.getResponseMessage();

            /* After a number of checks, call the appropriate timed
               automaton and make a transition:
            */
            ra.makeTrans(datemillisec, "request",writer);

            /* Here handle violations etc */

        } catch (Exception e) {
            throw AxisFault.makeFault(e);
        }
    }
}
```

Figure 7.5: Excerpts from the automatically generated Java code for the Axis handler

7.5 Appendix

7.5.1 Structure of the source code

This is a brief overview of the structure of the project:

- SLAng is defined in the file `src/uk/ac/ucl/cs/slangta/specification/slang.emof`. By modifying this file it is possible to define / modify SLAng clauses to define custom SLAs
- The SLA editor with the SLAngMon plugin is defined in src/uk/ac/ucl/cs/slangta/editor/SLAngTAEditor.java.
- The actual pluging to generate the checkers is defined by the files in src/uk/ac/ucl/cs/slangta/editor/action and src/uk/ac/ucl/cs/slangta/editor/tautils (this is the automata-related code)

By modifying any of the previous entities, SLAngMon can be easily customized to suit many applications scenarios.

7.5.2 FAQ

- **How do I create an SLA in SLAng using the editor?** Please see D2.1 and D2.2 for the description of the SLAng language and its associated editor.

7.5.3 An SLA in SLAng (XMI representation)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <XMI xmlns:SLAng="file:////Users/franco/college/mysoft/e321ws/SLAng-plastic/gen/uk/ac/ucl/cs/
       slangta/specification/slangta.emofxmi" xmi.version="1.2">
3  <!--This document is in XMI format according to the OMG XML Metadata Interchange (XMI)
       Specification v.1.2, OMG Document formal/02-01-01 (http://www.omg.org/)-->
4  <XMI.header>
5  <XMI.metamodel href="file:////Users/franco/college/mysoft/e321ws/SLAng-plastic/gen/uk/ac/ucl/cs/
       slangta/specification/slangta.emofxmi"/>
6  </XMI.header>
7  <XMI.content>
8  <SLAng:Duration unit="min" value="10.0" xmi.id="mofid:21774"/>
9  <SLAng:Duration unit="min" value="30.0" xmi.id="mofid:21808"/>
10 <SLAng:Duration unit="hr" value="4.0" xmi.id="mofid:21839"/>
11 <SLAng:PartyDefinition description="Patient" identifier="Patient" sLA="mofid:21779" xmi.id="
       mofid:21775"/>
12 <SLAng:PartyDefinition description="EHP" identifier="EHP" sLA="mofid:21779" xmi.id="mofid:21777"/
       >
13 <SLAng:SLA description="SLA␣for␣trigAlarm" xmi.id="mofid:21779">
14 <SLAng:SLA.auxiliaryClauses>
15 <SLAng:AuxiliaryClause xmi.idref="mofid:21796"/>
16 <SLAng:AuxiliaryClause xmi.idref="mofid:21803"/>
17 </SLAng:SLA.auxiliaryClauses>
18 <SLAng:SLA.services>
19 <SLAng:ServiceDefinition xmi.idref="mofid:21784"/>
20 </SLAng:SLA.services>
21 <SLAng:SLA.parties>
22 <SLAng:PartyDefinition xmi.idref="mofid:21777"/>
23 <SLAng:PartyDefinition xmi.idref="mofid:21775"/>
24 </SLAng:SLA.parties>
25 </SLAng:SLA>
26 <SLAng:ElectronicServiceClientDefinition owner="mofid:21775" xmi.id="mofid:21781"/>
27 <SLAng:ElectronicServiceDefinition client="mofid:21775" description="eHealthService" provider="
       mofid:21777" sLA="mofid:21779" xmi.id="mofid:21784">
28 <SLAng:ElectronicServiceDefinition.interfaces>
29 <SLAng:ElectronicServiceInterfaceDefinition xmi.idref="mofid:21787"/>
30 </SLAng:ElectronicServiceDefinition.interfaces>
31 </SLAng:ElectronicServiceDefinition>
32 <SLAng:ElectronicServiceInterfaceDefinition xmi.id="mofid:21787">
```

```
33 <SLAng:ElectronicServiceInterfaceDefinition.operations>
34 <SLAng:OperationDefinition xmi.idref="mofid:21792"/>
35 <SLAng:OperationDefinition xmi.idref="mofid:21790"/>
36 <SLAng:OperationDefinition xmi.idref="mofid:21794"/>
37 </SLAng:ElectronicServiceInterfaceDefinition.operations>
38 </SLAng:ElectronicServiceInterfaceDefinition>
39 <SLAng:OperationDefinition description="trigAlarm" interface="mofid:21787" xmi.id="mofid:21790"/>
40 <SLAng:OperationDefinition description="Mild␣Medical␣Service␣(Remote)" identifier="MMS" interface
       ="mofid:21787" xmi.id="mofid:21792"/>
41 <SLAng:OperationDefinition description="Serious␣Medical␣Service␣(on-site)" identifier="SMS"
       interface="mofid:21787" xmi.id="mofid:21794"/>
42 <SLAng:FixedLatencyFailureModeDefinition maxDuration="mofid:21774" operations="mofid:21790" sLA="
       mofid:21779" xmi.id="mofid:21796"/>
43 <SLAng:FixedLatencyFailureModeDefinition maxDuration="mofid:21808" operations="mofid:21794" sLA="
       mofid:21779" xmi.id="mofid:21803"/>
44 <SLAng:FixedLatencyFailureModeDefinition maxDuration="mofid:21839" operations="mofid:21792" xmi.
       id="mofid:21834"/>
45 <SLAng:Administration xmi.id="mofid:21801"/>
46 </XMI.content>
47 </XMI>
```

8 Conclusions and ongoing improvements

The previous chapters presented the technical details of the tools released within the WP4 Validation Framework. Currently the tools are being experienced within the project (in particular in the context of Work Package 5: Integration and Evaluation), and undergo continuous improvements. In this chapter we draw conclusions, and lay down some further considerations and results of ongoing work at the moment of writing.

The objective we pursue in WP4 is the development of a validation strategy for PLASTIC services, including both off-line and on-line approaches, covering both functional and extra-functional (QoS) properties. This results in a validation matrix methodology. Therefore, the validation framework is not committed to the instantiation of one fixed testing environment, but rather is conceived in the more general sense of an open flexible validation process, which spans over the development, deployment, and provision of PLASTIC services. This has been extensively described in Deliverable D4.1, which provided a detailed view on the proposed PLASTIC validation framework, with state-of-the-art overview and justification for the adopted techniques. The leading objective of the second year of activity has been to implement the envisaged framework into a working validation platform. During the last twelve months, WP4 has thus focused on the implementation and assessment of the specified PLASTIC validation framework. The set of tools deployed in this deliverable includes:

- WEEVIL A synthetic-workload generator coupled with an environment for managing the deployment and execution of experiments
- JAMBITION+MINERVA A model-based testing tool that automatically derives and executes invocation sequences on a service, checking whether the responses conform to a given specification, expressed as a Service State Machine (SSM). The embedded Minerva library permits to model SSMs via an UML modelling tool
- PUPPET A tool for the automatic generation of test-beds to empirically evaluate the QoS features of a Web Service under development.
- SLANGMON An Eclipse Plugin for the creation of SLAs using SLAng and the generation of AXIS handlers that monitor at run time the fulfillment of specified SLAs
- DYNAMO-AOP A framework for monitoring functional properties of external services which a BPEL process interacts with, to realize a composite service

Concerning some relevant results and ongoing work, as said in Chapter 4, the current version of PUPPET integrates the emulation of the functional specifications as part of the generated testbed. Therefore the automatically obtained stubs can expose not only the specified extra-functional parameters but also meaningful functional behavior. The earlier version of PUPPET (see D4.2) derived stubs exposing a SLA conforming behaviour, but did not consider functional aspects (i.e., the stubs provided good QoS values but the responses were not built to be semantically meaningful). However, we have since realized that in the general case extra-functional aspects are tightly coupled with functional characteristics. For instance ordering a particular good will require more or less time depending on characteristics of the specific good given that different warehouses have to be contacted. Indeed, stubs that only realize the desired QoS properties ignoring the functional specification, or vice versa consider the environment functionality only, ignoring the extra-functional properties, can be insufficient to raise failures potentially caused by a combination of causes. To the best of our knowledge, no such framework supporting in integrated way the functional and extra-functional validation of composite services exists. To gain confidence in

the implementation environment, we will need to carry out some experiments to verify if the obtained stubs really behave as their protocols dictate. This is certainly within the scope of WP5 experimentation.

WS-Guard is one of the tools that has been cut from the PLASTIC project after the first year review. Following the recommendation to avoid spreading effort towards too many directions, this tool was dropped. The main motivation behind this decision concerned the fact that the case studies proposed by the industrial partners did not show the necessity of having a registry augmented with testing capabilities. The WS-Guard registry developed until this decision remains available as a proof-of-concept implementation ready for download from the PLASTIC web site. The current version of the WS-Guard registry has been tested with some small examples; nevertheless it is a proof-of-concept version and at the moment its configuration requires some effort. In particular references to other services, such as Jambition, are currently hard-coded within the registry source code. In the near future, our objective is to continue with the development solving the issues that have not been completely addressed so far. Besides the technical issues mentioned above we mainly refer here to the generation of stubs for services to be discovered by the service under audition. This is a relevant and difficult problem in particular when the involved services refer to stateful resources. Finally another interesting future development concerns the study of the applicability of the audition testing phase also to other publish/discover protocols.

Concerning the Dynamo-AOP framework, the main limitation of the current version is that it monitors only functional properties of stateless services; on the basis of this consideration, we envision two different directions of improvement. The first one will concern the kind of properties that can be monitored: the specification language should be extended to allow for expressing also extra-functional properties of services; an initial attempt to define such a specification language is described in [3]. Moreover, this extension should also be integrated with the existing approaches on monitoring QoS properties, developed within WP4. The second extension will focus on extending both the monitoring framework and the specification language to support monitoring conversational web services, a specific class of stateful services. This extension will be based on the work described in [9]. Other improvements will deal with the usability of the monitoring framework: mainly, we plan to develop a graphical user interface, blended within the ActiveBPEL control panel, to configure the parameters and the monitored properties of a process.

Overall, we believe that the proposed matrix validation methodology provides an important advancement to the state-of-art. Indeed, while some partial approaches are elsewhere proposed, to our knowledge no such a comprehensive validation methodology for SOA existed. To fill the matrix cells, we have then released several tools to suitably address the exigencies of PLASTIC. At the time of writing the tools have been relased to PLASTIC partners, and they are being experimented in the adopted scenarios, in various configurations. This is already a first confirmation of the good flexibility and deployability of the proposed matrix framework. We are receiving comments, requests of change and error reports of which we will make treasure in further improving the tools and releasing the final version of the framework by the end of the project.

Bibliography

[1] Active Endpoints. Activebpel engine architecture. http://www.activebpel.org/docs/architecture.html, 2007.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, May 2003.

[3] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, December 2007.

[4] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC 2005: Proceedings of the 3rd International Conference on Service Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.

[5] A. Bertolino, G. D. Angelis, and A. Polini. A QoS Test-bed Generator for Web Services. In *Proc. of the 7th International Conference on Web Engineering 2007 (ICWE 2007)*, volume LNCS series, Como, Italy, 2007. Springer Verlag.

[6] A. Bertolino, G. D. Angelis, and A. Polini. Automatic Generation of Test-beds for Pre-Deployment QoS Evaluation of Web Services. In *Proc. of the 6th International Workshop on Software and Performance (WOSP 2007)*, Buenos Aires, Argentina, 2007. ACM.

[7] A. Bertolino, D. Bianculli, A. Carzaniga, G. De Angelis, I. Forgacs, L. Frantzen, Z. Gere, C. Ghezzi, A. Polini, F. Raimondi, A. Sabetta, and A. Wolf. Test Framework Specification and Architecture. Technical Report Deliverable D4.1, PLASTIC Consortium, March 2007. IST STREP Project.

[8] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, LNCS 3938, 2004.

[9] D. Bianculli and C. Ghezzi. Monitoring conversational web services. In *Proceedings of the 2nd International Workshop on Service-Oriented Software Engineering (IW-SOSWE'07), co-located with ESEC/FSE 2007*, pages 15–21, New York, NY, USA, September 2007. ACM Press.

[10] W. Emmerich, F. Raimondi, J. Skene, V. Cortellessa, P. Inverardi, M. Tivoli, D. D. Ruscio, M.Autili, R. Mirandola, V. Grassi, A. Sabetta, J. Gonzales, P. Mazzoleni, and S. Tai. SLA language and analysis techniques for adaptable and resource-aware components. Technical Report Deliverable D2.1, PLASTIC Consortium, March 2007. IST STREP Project.

[11] S. Guinea. *Dynamo: a Framework for the Supervision of Web Service Compositions*. PhD thesis, Politecnico di Milano, 2007.

[12] P. Inverardi, V. Cortellessa, A. Di Marco, M. Autili, et al. Formal description of the PLASTIC conceptual model and of its relationship with the PLASTIC platform toolset. Technical Report Deliverable D1.2, PLASTIC Consortium, March 2008. IST STREP Project.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[16] F. Liotopoulos, S. Tai, J. Sairamesh, H. Eikerling, J. Gonzalez, J. Barra, M. Jazayeri, J. Wuttke, P. Inverardi, V. Cortellessa, A. Di Marco, and M. Autili. Scenarios, Requirements and initial Conceptual Model. Technical Report Deliverable D1.1, PLASTIC Consortium, June 2006. IST STREP Project.

[17] H. Ludwig. WS-Agreement Concepts and Use - Agreement-Based Service-Oriented Architectures. Technical Report RC23949, IBM, May 2006.

[18] No Magic Inc. MagicDraw. http://www.magicdraw.com.

[19] Object Management Group. *UML 2.0 Superstructure Specification*, ptc/03-08-02 edition. Adopted Specification.

[20] J. Skene and W. Emmerich. Engineering runtime requirements: monitoring systems using MDA technologies. In *Trustworthy Global Computing, International Symposium, TGC 2005, Revised Selected Papers*, volume 3705 of *LNCS*, pages 319–333. springer, 2005.

[21] W3C. XML path language (XPATH). on-line at: http://www.w3.org/TR/xpath, 1999.