PLASTIC

IST STREP Project

Deliverable D4.1

Test Framework Specification and Architecture

http://www.ist-plastic.org

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

Deliverable Number	:	D4.1
Title of Deliverable	:	Test Framework Specification and Architecture
Nature of Deliverable	:	
Dissemination level	:	
Internal Document Number	:	
Contractual Delivery Date	:	
Actual Delivery Date	:	
Contributing WPs	:	WP4
Editor(s)	:	Antonia Bertolino, Domenico Bianculli, Istvan Forgacs, Andrea Polini
Author(s)	:	Antonia Bertolino, Domenico Bianculli, Antonio Carzaniga, Guglielmo De Angelis, Istvan Forgacs, Lars Frantzen, Zsolt Gere, Carlo Ghezzi, Andrea Polini, Franco Raimondi, Antonino Sabetta, Alexander Wolf
Reviewer(s)	:	Sebastian Elbaum, Paola Inverardi, Alexander Wolf

Abstract

This document is the first deliverable of PLASTIC Work Package 4 (WP4): *Service Validation Methodology and Tools*. It provides the high-level specifications of the "Test framework specification and architecture" aimed at validating the PLASTIC applications. Generally speaking, the validation of B3G applications requires novel advanced technologies to face the high complexity of the systems considered, in terms of distribution, mobility, heterogeneity, dynamism, context-awareness, and the stringent requirements for quality of service and dependability. Given the blend of functional and extra-functional properties that need to be validated, in WP4 we aim at a *holistic approach* which includes analytical and empirical techniques to be employed both during development and at runtime.

The framework has been organized around three main stages that we have identified as suitable for the validation of B3G services. A first main distinction concerns whether the services are tested in a fake/simulated environment before deployment, or rather after deployment, by validating their behaviour while they execute in the real environment. We name the two kinds of testing as *off-line* and *on-line*, respectively, and we claim that the validation of PLASTIC applications should stem from a suitable combination of both these kinds of testing. In particular, in addition to off-line testing, we adopt on-line testing because the high run-time dynamism of service-based systems makes it impossible to foresee all possible contexts in which the services will be used. Both off-line and on-line validation should consider the functional properties and the extra-functional properties.

With regard to off-line validation, we propose advanced techniques such as simulation-based testing and model-based testing on symbolic state machine, and introduce support for distributed experimentation and automated test harness generation. In relation to on-line validation, we introduce a further distinction between i) on-line testing before publication, by which a service undergoes a sort of qualification exam, called Audition, before it can be "officially" recognized as having adequate quality for being included in the PLASTIC environment, and ii) monitoring of the service behaviour during real usage, in which case we provide for an aspect-oriented approach to monitoring service compositions, as well as for the verification that the contractually agreed extra-functional specifications are fulfilled.

As a result of the rich variety of challenges, in WP4 we introduce a set of technologies. The proposed techniques used altogether provide a powerful multilateral approach to the validation of B3G applications. As their usage however requires some effort and implies the provision of adequate basic technology, they have been conceived so that each can be used independently from the others, and also they can be variously combined to obtain a more comprehensive approach. An integrated overview of the resulting architecture, in terms of a unified process, is given in the concluding part of this document.

Keyword list

Audition, Distributed experimentation, Model-based testing, Monitoring, Off-line testing, On-line testing, QoS, Simulation-based testing

Document History

Version	Type of change	Author(s)
0.1	Proposed outline	CNR
0.2	Proposed outline with updates from partners	CNR
0.3	New outline coming out from Athens discussions – Includes deadline for contributions	CNR (with USI and 4DS)
0.4	First draft version with state of the art and some other initial contributions. Warning: probably in this version some cross-references of references went lost in editing and are wrong	CNR, USI, 4DS
0.5	Second draft version – Antonia's revision of V4.0 and addition of Sec. 3.4	CNR
0.6	Dec. 11, 2006: Third draft version – newly added sections 1.2.1.4 (Satoh's flying emulator), 3.2, 5.1, 5.2, 5.3, and modified and expanded sections:, 1.2.2, 1.2.3, 2.5.1.1, 3.4, 4.2, plus of course the list of References. In this version notably still missing: Audition architecture, General framework (whole Chapter 6), and cross references still giving troubles (Words problems, to be adjusted in a more mature version) NB: Finally AB separated former Chapter 1 into two chapters, therefore all above references to Section numbering do not apply anymore to current version.	CNR, USI, 4DS
0.7	Chapter1 from 1 to 6 Finalized. Complete overview of all individual contribution, circulated within WP4 for QUICK feedback and review.	CNR, USI, 4DS, UCL
1.0	First complete version for review: checked all cross-references, proof checking, etc… Added chapters 7, 8, and front-matters material	CNR
1.1	Revised version after Paola and Alex reviews	CNR
1.2	Carefully re-corrected all cross-references lost for Word problems	CNR, USI; 4DS

Document Review

Date	Version	Reviewer	Comment
Jan07	V0.9	Sebastian Elabum	Suggested some reorganization, found weaknesses
Feb07	V1.0	Paola Inverardi	Checked coherence and integration
Feb07	V1.0	Alexander Wolf	Proofed

Table of Contents

Table of Acronyms 9

List of Figures 11

List of Tables 13

1 Introduction 14

1.1 Terminology 15

1.1.1 Services vs. Web services 15
1.1.2 Functional Validation vs. QoS validation 15
1.1.3 Off-line vs. On-line 15
1.1.4 Deployment 16
1.1.5 Testing vs. Monitoring 16

1.2 Scope and Context 16

1.3 Problems and Objectives 17

1.3.1 SOA Testing 17
1.3.2 QoS Evaluation 18
1.3.3 Context-awareness and Adaptability 19

1.4 Roadmap 20

2 State of the Art Overview 21

2.1 Testing of networked systems 21

2.1.1 Distributed Testing Frameworks and Testbeds 21
2.1.2 Early-phase Performance Evaluation 22
2.1.3 Distributed Component Unit Testing 22
2.1.4 Testing of Concurrent Systems 22
2.1.5 Testing Software Distributed on Mobile Terminals 23

2.2 Web Services Testing 23

2.2.1 Involved Actors 24
2.2.2 Approaches to Off-line WS Testing 24
2.2.3 Testing by Enhancing the Directory and Discovery Service 25
2.2.4 WS-I Initiative 26

2.3 Runtime Monitoring 26

2.3.1 Assertions Monitoring 27
2.3.2 Requirements Monitoring 27
2.3.3 Monitoring for Planning 27
2.3.4 Interactions Monitoring 27
2.3.5 Logical Architectures for Monitoring 27
2.3.6 Some Industrial Proposals 28

2.4 QoS Validation of Networked Services 28

2.4.1 Predictive vs. Empirical Techniques 28
2.4.2 SLA-based Monitoring 29
2.5 Existing Tools for Web Service Testing 30
2.5.1 Parasoft SOATest 32
2.5.2 Mindreef SOAPScope 34
2.5.3 PushToTest TestMaker 37
2.5.4 Summary Table 39
2.5.5 Discussion 40
2.6 Summary 40
3 Context Setting and Background 41
3.1 Introduction 41
3.2 Problems and Objectives 41
3.3 Adopted PLASTIC Validation Methodology 43
3.4 Modelling 45
3.4.1 Behaviour Models 45
3.4.1.1 Service State Machines (SSMs) 45
3.4.1.2 Simulations 50
3.4.2 QoS Annotations 52
3.5 Summary 54
4 Off-line Validation Stage 55
4.1 Introduction 55
4.2 Experiments on Networked Services 56
4.2.1 Workload Generation 58
4.2.2 Deployment and Execution 59
4.2.2.1 Configuration Modeling 60
4.2.2.2 Setup and Script Generation 62
4.2.2.3 Deployment and Execution 63
4.3 Simulation-based Testing 63
4.3.1 Simulation Environment 65
4.3.1.1 Process Model 65
4.3.1.2 Communication 65
4.3.1.3 Messages 66
4.3.1.4 Components 66
4.3.2 Configuration 66
4.3.3 Simulation Execution 66
4.3.4 Aggregation 67
4.3.5 Fault-Based Analyses 67
4.3.6 Usage Scenarios 68
4.3.6.1 Conventional 68
4.3.6.2 Boosting 68
4.3.6.3 Ranking 69
4.4 Model-based testing 69

4.4.1 A Taxonomy of Web Service Scenarios ... 70
4.4.2 The AMBITION tool ... 72
4.4.2.1 On-The-Fly Testing ... 73
4.4.2.2 The On-The-Fly Testing Algorithm ... 73
4.4.2.3 Open Issues ... 75
4.4.3 The JESSI tool ... 76
4.5 QoS Evaluation ... 77
4.5.1 Open issues ... 81
4.6 Summary ... 81
5 On-line pre-publication stage: Audition ... 82
5.1 Introduction ... 82
5.2 Scenarios for Audition ... 82
5.2.1 Audition Testing Targets ... 83
5.2.2 Kind of Checks Used During the Audition Testing Phase ... 85
5.2.3 Putting in Relation Service Testing Scenarios and Check Typologies ... 86
5.2.4 Using Audition ... 88
5.2.5 Requirements Issues ... 89
5.3 Audition Architecture ... 90
5.4 Summary ... 92
6 On-line Live-usage Validation Stage ... 93
6.1 Introduction ... 93
6.2 Dynamic Monitoring of WS Composition ... 93
6.3 SLA Monitoring Based on Formal Language ... 95
6.3.1 Preliminaries ... 96
6.3.1.1 Timed Automata ... 96
6.3.1.2 TCTL, a logic for real time ... 97
6.3.2 Counting events and real time constraints ... 97
6.3.2.1 Verification of traces ... 97
6.3.3 From SLAng to real-time verification ... 98
6.3.4 A methodology for on-line monitoring ... 100
6.4 Qos Monitoring ... 101
6.4.1 The Monitoring Framework ... 101
6.4.2 Dynamic Loggers ... 102
6.4.3 Platform Observer ... 103
6.4.4 Data Collection Controller ... 104
6.4.5 SLA Parser ... 104
6.4.6 SLA Analyzer ... 104
6.5 Summary ... 105
7 Validation Framework Architecture ... 106
7.1 Introduction ... 106

7.2 Integrated Framework Architecture 106
7.3 Risks and Mitigation Actions 111
7.4 Summary 111
8 Conclusions 112
9 References 113

Table of Acronyms

Acronym	Extended Version
AMBITION	Automatic Model-Based Interface Testing In Open Networks
AOP	Aspect Oriented Programming
ASG	Adaptive Services Grid
ATS	Abstract Test Suite
B3G	Beyond Third Generation
BPEL	Business Process Execution Language
CB	Component-based
CBSE	Component-based Software Engineering
COMET	Converged Messaging Technology
CV	Curriculum Vitae
DES	Discrete Event Simulations
GCD	Greatest Common Divisor
GGF	Global Grid Forum
IBC	Implementation Based Conformance
ICT	Information and Communications Technology
JESSI	Java Editor and Simulator for Service Interfaces
LMS	Learning Management Systems
LTS	Labelled Transition Systems
MDT	Model-Based Testing
MADAM	Mobility and Adaptation Enabling Middleware
PUT	Process Under Test
QoS	Quality of Service
SeCSE	Service Centric System Engineering
SLA	Service Level Agreement
SLS	Service Level Specification
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSM	Service State Machine
SUE	Service Under Evaluation/System Under Experimentation
SUT	Software Under Test
TP	Test Process

UBC	Usage Based Conformance
UDDI	Universal Description, Discovery and Integration
WP	Work Package
WS	Web Service
WS-A	Web Service Agreements
WS-I	Web Service Interoperability
WSCL	Web Services Conversation Language
WSDL	Web Services Description Language
WSLA	Web Service Level Agreement

List of Figures

Figure 1: Screenshot from the Parasoft SOATest tool 32
Figure 2: Screenshot from the Mindreef SOAPScope tool 36
Figure 3: Screenshot from the PushToTest TestMaker tool 38
Figure 4: PLASTIC testing stages....43
Figure 5: Exemplary setup of WS 47
Figure 6: The choreography as a Sequence Diagram....47
Figure 7: The choreography as a Coordination-SSM....48
Figure 8: A multiport-SSM specifying the supplier-role....49
Figure 9: WS Agreement document structure 53
Figure 10: Experiment trial automation....57
Figure 11: Simulation-based workload generation 59
Figure 12: Detailed view of trial automation process 60
Figure 13: SUE conceptual model 61
Figure 14:Testbed conceptual model 61
Figure 15: Actor-mapping conceptual model 62
Figure 16: A generic testing process 63
Figure 17: The general MBT approach 69
Figure 18 : Testing a single, passive WS....70
Figure 19 : Testing a single, active WS....70
Figure 20 : Testing a coordinated setup....71
Figure 21: The tester plays the role of the Warehouse 71
Figure 22 : The environment of the customer is tested 72
Figure 23: The JESSI approach....76
Figure 24 : The off-line MBT approach via JESSI and AMBITION 77
Figure 25: Puppet approach and supporting tool 78
Figure 26: Stubs generation logical process 79
Figure 27: Audition components and interactions 86
Figure 28: Audition components implementation....91
Figure 29: Monitoring embedded within the BPEL engine....95
Figure 30: A simple automaton....96
Figure 31: A timed automaton TA1 96
Figure 32: A timed automaton TA2 97
Figure 33: HUTN code for SLAng (excerpts) 99

Figure 34: Timed automaton for the reliability clause 100
Figure 35: A proposed implementation of an automata-based monitor. 100
Figure 36: Architecture of the proposed QoS monitoring framework 102
Figure 37: In-bound and out-bound service requests 103
Figure 38: Example of logging mode switching based of an indicator function 103
Figure 39: The three validation stages in PLASTIC 107
Figure 40: Integrated validation framework 110

List of Tables

Table 1: Features of the examined WS testing tools 39
Table 2: Possible stakeholders for service validation 42
Table 3: PLASTIC proposed solutions 44
Table 4: Example of latency mapping 79
Table 5: Example of mapping for reliability 80
Table 6: Typical metrics captured by logs 102

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 [16].
Service provisioning over B3G distributed computing platforms faces numerous challenges, among which:

- easing the deployment of services on a wide range of infrastructures, from resource-constrained handheld devices to heterogeneous networks of systems;
- making such services resource-aware so that they can most appropriately benefit from networked resources and related services;
- taking care that users of the services always experience the best Quality of Service (QoS) possible according to their specific situation;
- in particular, the QoS may include guaranteeing that services provide adequate dependability, in terms of the specified levels of reliability, safety, security.

Because of the emphasis on the QoS and on the dependability of service behaviour, the validation of B3G applications must consider in equal measure both functional and extra-functional properties, i.e., not only we need to check that the behaviour of service-oriented applications complies with the expected logical sequence of events as defined in the service functional specifications, but also that the services abide to the expected or contractually agreed levels of quality, which are also part of the specifications in form of suitable annotations.

This document is the first deliverable of Work Package 4 (WP4): *Service Validation Methodology and Tools*. It provides the high-level specification of “Test framework specification and architecture” to validate the PLASTIC applications, in terms of the adopted techniques. From the above list of challenges faced by B3G applications, it is evident how their validation requires novel advanced technologies that can face the high complexity of the systems considered. Given the blend of functional and extra-functional properties that need to be validated, in WP4 we aim at a *holistic approach* which includes analytical and empirical techniques to be employed both at development stage and at runtime.

The framework is organized around three main stages that we identified as suitable for the validation of B3G services. A first main distinction concerns whether the services are tested in a fake/simulated environment before deployment, or rather after deployment, by validating their behaviour while they execute in the real environment. We name the two kinds of testing as *off-line* and *on-line*, respectively. We claim that the validation of PLASTIC applications should stem from a suitable combination of both these kinds of testing. In relation to on-line, in WP4 we also introduce a further distinction between i) on-line testing before publication, and ii) monitoring of the service behaviour during real usage. Moreover, both off-line and on-line validation should consider the functional properties and the extra-functional properties.

With regard to off-line validation, to make this more effective for the applications at hand we propose advanced techniques such as simulation-based testing and model-based testing on Service state machine, and introduce support for distributed experimentation and automated test harness generation. By the on-line testing before publication a service undergoes a sort of qualification exam, called Audition, before it can be “officially” recognized as having adequate quality for being included in the PLASTIC environment. Finally, regarding the monitoring of service in live-usage, we provide for an aspect-oriented approach to monitoring service compositions, as well as verification that the contractually agreed extra-functional specifications are fulfilled.

All these considerations give rise to an assorted framework, as described more in detail in Chapter 3, while the three stages with the techniques that we propose for functional and

extra-functional validation are described each in a separate chapter, which correspond to Chapters 4, 5 and 6 respectively.

In the remainder of this Chapter, we first establish the basic terminology adopted in WP4, then in Section 1.2 we define more in detail scope and context of the deliverable and finally in Section 1.3 consider specific problems and objectives for the introduced methodologies and tools. In Section 1.4 a roadmap of the document is outlined.

1.1 Terminology

As the literature is wide and many of the terms we use are overloaded, it is useful to informally introduce the meaning we assume in WP4 for the main terms adopted throughout.

1.1.1 Services vs. Web services

Services, much like components, are intended to be independent building blocks that collectively represent an application environment [36]. Unlike traditional components, services explicitly focus on a description of provided functionalities that can be contractually defined as a mix of syntax, semantics and behavior specifications. As a consequence, in contrast to component integration, which is carried out at assembling time, service integration may be carried out prior or at run-time [9]. In such a context, dynamic discovery and run-time integration become first level concepts of the service oriented proposal.

Web Services (WS) represent a concrete instantiation of the service oriented paradigm. The UDDI consortium defines them as "*self-contained, modular business applications that have open, Internet-oriented, standard-based interfaces*" [107]. In other words, Web Services are components that can be integrated into more complex distributed applications by means of some Web-based open standards (e.g. WSDL, UDDI, HTTP, SOAP, etc.) [2].

Without loss of generality, most of the WP4 tools and experimentation will target WSs, since they are by far the most common kind of services.

1.1.2 Functional Validation vs. QoS validation

As said in the Introduction, the requirements on B3G services involve functional and extra-functional properties. Correspondingly in WP4 we address both functional and QoS validation. Functional testing relies on validating the observed behaviour of a system in terms of the I/O sequences of events. In particular, it checks if the functionalities implemented by the systems conform to those described in the functional specification. On the other hand, QoS validation concerns the activities that focus on checking whether the system under test meets its extra-functional constraints. Specifically, in PLASTIC we assume that such properties could be defined by means of annotations on functional specifications as well as agreements describing the promised quality by/for a service.

1.1.3 Off-line vs. On-line

Within WP4, methods and tools for testing purposes are classified into two main categories: *off-line* and *on-line*.

Off-line validation activities are performed while no user ("paying customer") is using the service. Hence, off-line validation of a system implies that it will be tested in one or more artificially evolving environments that simulate possible real interacting situations. In this case the system is still undergoing development or, in a wider-ranging view, it is already completed but not available for use (deployed) yet.

On the other hand, on-line approaches concern a set of techniques, methodologies and tools to monitor the system after its deployment in one of its real working context. A possible scenario is that while the end-users are using the service, data are being collected about the

scenario under which the service is being used and these data are sent back to the development organization, which then determines (i.e., validates) whether the service behavior is correct (i.e., the "test" passes) or incorrect (i.e., the "test" fails). Hence, the "test cases" consist of actual usage scenarios. Otherwise, the development organization performs on-line validation activities on a fully fielded service, perhaps during idle times.

1.1.4 Deployment

Another important milestone in the lifetime of a service is the deployment. The deployment in general refers to the process whereby a software system is made available to be referred by other software. This process in turn may involve several steps. In WP4, for the purpose of devising an integrated validation framework, deployment is further subdivided into three different steps: installation, admission and publication (see Figure 4 on page 43). From the point of view of the tester, in particular, publication is the step which makes the difference on whether the service is visible to users or not. Before publication we apply the Audition approach (see Chapter 5), after publication we monitor system behaviour (see below).

1.1.5 Testing vs. Monitoring

Software testing is a broad term encompassing a wide spectrum of activities. The primary underlying common goal of these activities is to apply methodologies and techniques designed to make sure that a software system does what it was designed to do and that it does not do anything unintended. However, the idea of pursuing such a goal just by applying an exhaustive exploration of all the possible configuration of the system and of its environment is unrealistic. So, given a generic software system, testing activities focus on observing a sample of its executions.

Monitoring is a runtime activity whose objective is to collect data about a phenomenon and analyze them with respect to certain criteria. Monitoring is tackled in more depth in Chapter 6. Crucial monitoring steps are the elicitation of which data should be collected, how long the observing windows should be, what policies should be applied when a fault is detected, and so on. In some approaches, the middleware is responsible for providing the facilities to disseminate the data collected from the sources to the sinks where the data are analyzed.

The two practices clearly have many commonalities in terms of overall goals and problems. In this sense, we consider monitoring as a testing activity which is applied when the system is *on-line*.

1.2 Scope and Context

WP4 is entirely devoted to the validation of PLASTIC applications. The stated objectives of WP4, according to the Project “Description of Work” document, are the following:

1. To develop a testing methodology for mobile, adaptable component-based services.
2. To develop methodologies, both analytical and empirical, for assessing QoS of mobile, adaptable component-based services.
3. To develop a test framework to be incorporated into the PLASTIC platform.

Given the large variety of possible scenarios of application for the PLASTIC platform (as depicted in the PLASTIC Conceptual Model [92]), and given for each scenario the many types of interoperating services that are involved, the general approach that we have taken towards these objectives is that of not committing the validation framework to one specific test methodology, but rather to conceive it as an open environment which can include differing methods and tools.

In other words, to develop the test framework addressed in the third objective, which is the ultimate goal of WP4, we deem it more useful and farsighted to specify it in the more general

sense of an open flexible validation process, which spans over the development, deployment, and provision of PLASTIC services, rather than in the restricted sense of the instantiation of a specific testing environment.

Hence, in this deliverable, which specifies the "Test framework specification and architecture", we investigate in the broadest way what can be appropriate and promising methodologies and tools, both from the existing literature and newly developed within the PLASTIC project. For the next deliverable of WP4, which will provide the "Test framework prototype implementation and release", we will focus more specifically on the implementation of some instances of this generic process, that will be used for experimentation on the selected scenarios within Work Package 5.

Therefore in the remainder of this document we present a collection of useful approaches, which can either be used in isolation for the validation of a specific application, or, preferably, adopted in a suitable combination. The methods and tools proposed obviously need to fit the conceptual modelling and specification approaches devised in Work Package 1 and Work Package 2, respectively, and regarding the possibility to observe the runtime execution, they must be compatible and interact with the middleware infrastructure being built in Work Package 3. These dependences provide the context for the validation framework, and impose constraints on its implementation. While WP1 has already released the PLASTIC conceptual model, the development activities of WP2, WP3 and WP4 have proceeded in parallel. We have then left open for this deliverable some issues that will be dealt with when the respective results of these three technical WPs will be integrated within the next-coming WP5.

1.3 Problems and Objectives

As said, the validation of PLASTIC applications poses a mix of new and old challenges, whose solution can only be searched by the joint employment of different approaches to testing and analysis.

When investigating a new domain, the first question that should be answered is what is it that makes this domain different from what has been done in the past. Below we address such question considering a few topical issues. What makes the WP4 task very challenging is that all of these issues, and even more, need to be addressed altogether, since all of them characterize B3G applications jointly.

In the following, we present some solutions that we started to investigate within the PLASTIC project. The discussion covers the problems highlighted above and illustrates why the proposed approaches can solve or mitigate these problems.

1.3.1 SOA Testing

As with any other newly introduced technology, the emergence of the Service Oriented Architecture asks for reviewing all software development phases to understand what can be adapted from previous experience and what instead requires the development of new approaches. We are interested here in particular to understand which are the new and more challenging issues with relation to testing when the service oriented paradigm is considered.

In our view, testing services shows many of the challenging issues that testers had to solve in the past. In particular at a first glance many, if not all, the difficulties emerged in the Component-Based testing field seem to be back, maybe in a more subtle form. In fact:

1. service based applications are built by integrating pieces of software in a distributed setting,
2. integrated services are generally developed by different stakeholders,

3. configuration of software applications can change during run-time (in the component-based domain when one or more components change)

At the same time there are additional new features that strongly modify the perspective on testing, even with respect to a component-based application. In particular:

1. the third feature listed above is, in the case of services, pushed to the extreme consequences. In particular services discover each other at run-time and nobody can predict who will be the receiver or the provider of a given service.
2. services developed by different stakeholders are managed and run on different machines owned by different organizations. In the general case nobody has a full control on all the software parts composing a service-based application. This characteristic makes the verification step particularly hard,

The items in the above list probably are the greatest novelties of the service oriented paradigm and probably the biggest challenges that must be overcome for verification purposes.

For many different engineering aspects, and in particular for testing, it is certainly relevant to note that the service oriented paradigm foresees the availability of a description of services in a computer processable format. This is one of the main characteristic of the service oriented paradigm that explicitly foresees a point in which such information can be stored and retrieved. The minimal information model concerning a service should allow the description of the service interface in terms of the signature of each provided method. Within the Web Service specification suite this information can be expressed by a WSDL definition .

Clearly, the description of a service only in terms of the method signature is too poor for any engineering activity. Nevertheless the information model of a service can be extended. For instance Web Service related technologies explicitly support storing such information using UDDI Models [84]. Many different proposals appeared in the literature suggesting solutions and increased information models to describe services. With reference to testing activities the very general idea is to assume the availability of service descriptions suitable for testing purpose. In other words it is useful to describe a service providing information and models suitable for the automatic derivation of test cases. Similar proposals already emerged within the CB software engineering community. Nevertheless, in that context, an explicit and standard support for the management of “metadata” was missing.

Moreover, from the software developer point of view, a service can be obtained from the composition of other services. Indeed, the very notion of a service may span different levels of complexity, since it can refer both to a single service or to a complex service obtained via the composition of other services. While this does not make any difference from the service user viewpoint, who in any case always invokes the service from the externally provided interface and does not (want to) care about what happens behind, clearly from the tester’s point of view the validation becomes much more difficult when the interaction among more services has to be tested. Also for this, we need to refer to some description of how the interaction is assumed to happen, which is different depending on whether the services are orchestrated or they form a choreography.

The techniques proposed in this document all variously assume the availability of augmented service descriptions and discuss methods that can be used, given the availability of such information, to test services. In each case the objective of testing can be different and the same approach can be sometimes used with different contexts and timings.

1.3.2 QoS Evaluation

In PLASTIC the QoS specifications are first-class citizens of the application model, as detailed in [93] and hence the successful provision of services is indivisible from the achievement of the contractually agreed levels of QoS. Therefore, in validating service

adequacy, a key objective is to verify that the delivered QoS conforms to the specified agreement. We need to enhance functional testing techniques with appropriate approaches for QoS evaluation. This task involves two different levels of problems.

At the test planning level, we need to understand which test cases should be selected, whereby each test case also includes the configuration of quality parameters. Hence also the QoS properties to be validated need to be suitably specified, so that we can extend the derivation of test cases, e.g. by use of model-based approaches, to account for extra-functional properties too. The trend in SOA applications is that QoS agreed attributes and levels are specified by means of appropriate annotations which augment the service specifications. The challenge then is: how can we use such annotations to devise an effective validation strategy?

A second level of problems regards configuration and environment. As we will discuss in more detail in the remainder of this document, evaluating by means of testing specified performance attributes such as the response time for a service, or dependability attributes, such as the exposed reliability, involves many technical difficulties since such qualities depend not only on the service under test, but also on the underlying support system and on the network. Therefore, the tough challenge here is how to validate the provided QoS of an application, keeping the aspects that pertain strictly to the observed system separate from the aspects which depend on the underlying platform. The service developer can intervene on the former, but not on the latter, which nonetheless should be accounted for.

The consequent challenge for off-line testing is to provide a testbed in which the underlying platform and network can be simulated in a realistic way: this challenge will be more properly the subject of the next experimentation activities in WP5.

In on-line testing, instead, provided that we insert the probes for measuring in the right places we can measure the point-to-point attributes in real usage. The monitoring process however consumes valuable resources. Therefore, care should be put in devising lightweight approaches which do not heavily impact performance. In fact, one of the problems that we need to explore is how to keep the overhead introduced by data collection and analysis to a minimum. This can be achieved by an intelligent sampling of the introduced probes and by dynamically adjusting the monitoring intensity based on some suitable policy.

1.3.3 Context-awareness and Adaptability

Context-awareness and adaptation are key aspects of PLASTIC applications. Such properties characterize services that can handle mobility and changes in the supporting platforms and networks. In particular, in the scenarios identified for PLASTIC two types of mobility have been foreseen [92]: physical mobility, for services that are accessed from mobile devices, and logical mobility, for mobile business services.

The vision of PLASTIC is that the provided applications can react to context changes, both anticipated and unanticipated, and adapt themselves so to continue to provision services with the guaranteed levels of QoS. The validation of such a requirement would demand that, for each given service under test, we think in advance of all the many changes of context that could happen and observe which is the behaviour of the service under each new context. The complexity of such an endeavour is clearly unaffordable, because not only the parameters to take into account would be too many, but also, as already said, the service reconfiguration may happen dynamically at runtime, and involve scenarios of services interaction that are not foreseeable at the time the service is tested.

In the Deliverable D2.1 [93] the impact of context-awareness and adaptability on to the software engineering process is discussed in more detail. As explained there (see in particular [93] Chapter 2: Roadmap), the process evolves progressively from a conventional one in which the system is frozen towards a “futuristic” one allowing for larger and larger portions of the system to change at runtime.

This is the reason why we have decided to supplement conventional off-line testing with on-line testing. In fact, before the application is deployed we can try to reproduce in laboratory some possible contexts of use, but such trials cannot envisage all potential usages. By testing on-line, instead, we can potentially monitor the executions in their real environment, and in principle detect all potential malfunctions. This becomes particularly relevant to the validation of QoS properties, since by using on-line approaches we can measure really exhibited quality, while in off-line testing any measure will be at the best a realistic simulation.

There are drawbacks in this view, though. In fact, even though we can monitor the behavior at runtime and detect malfunctions, it is late for counteracting, since we identify the failure only after it manifests itself. With on-line testing, we can only intervene for recovering and for preventing malfunctions in future usages of the service. This is why we also need exploit and improve off-line testing so that we can early experiment the service usage under different configurations. In WP4 we will provide support for off-line testing of both functional and QoS properties over distributed testbeds.

A second drawback, already mentioned, is that the monitoring infrastructure could degrade performance. Hence we need to identify appropriate policies for monitoring the dynamic adaptation of services. We need to capture the relevant measures at the right moment, as an example the data collection could be triggered by the occurrence of some specified adaptation events, and the monitoring approach itself could be made adaptive so to vary its weight depending on the available resources.

1.4 Roadmap

This document is organized as follows:

- In Chapter 2 we survey the related work in the areas of interest for WP4. We report on existing approaches and tools, and on relevant ongoing work, both from PLASTIC partners and other groups.
- Chapter 3 provides a deeper introduction to the techniques and tools under development. We set the context for the validation framework, and then also give some background concepts and definitions.
- Chapters 4, 5 and 6 are devoted to the three identified stages for validation of B3G applications, which are off-line, on-line pre-publication, and on-line live usage.
- Chapter 7 finally describes how the presented techniques fit together. The integrated framework consists of an open flexible process in which the developed techniques can be combined in different ways.
- Chapter 8 briefly summarizes conclusions and hints at future steps.

2 State of the Art Overview

In this Chapter we provide a survey of topics related to the validation of B3G services, by reviewing the literature of related work. We have identified several topics which in our opinion altogether make the context for the development of the WP4 framework: i.e., while none of them alone provides a ready solution for the testing of B3G services, each of them should be considered in developing a validation framework for PLASTIC. These topics include: issues relative to the inherent distribution, mobility and concurrency of B3G services (Section 2.1); approaches for the testing of Web Services (Section 2.2); approaches to run-time monitoring (Section 2.3); approaches to QoS validation (Section 2.4); and finally existing support tools for web services testing (Section 2.5).

2.1 Testing of networked systems

Distribution is one inherent characteristic of B3G applications, and in WP4 we need to cope with issues surrounding the testing of networked services. Unfortunately, a major motivating factor of our work to build a framework for validation of PLASTIC applications is the lack of any general-purpose, disciplined, and effective testing method for distributed systems. This being said, a number of studies and research efforts exist, aimed at understanding the problem and at developing methods and tools to improve the state of the art.

Several authors describe their experience and highlight the issues related to testing distributed systems [33], [48], [74]. In particular, Ghosh and Mathur [48] list a number of differentiating factors between distributed and non-distributed software that are representative of those mentioned by other authors:

- distributed systems have a larger scale;
- heterogeneity, monitoring and control are more difficult;
- non-determinism, concurrency, scalability and performance, and partial failure (fault tolerance) contribute complexity.

Of these, tool support for monitoring and controlling distributed tests has received considerable attention. Some tools support primarily functional testing [45], [47], [57], [56]; others are specifically targeted at performance testing [32], [50]. While this is important work, it addresses only the accidental issues associated with distributed system testing, and does not address the more fundamental questions having to do with what and how best to test a networked service.

Several studies concentrate on distributed component-based systems. Gosh and Mathur [48] present an adequacy criterion targeted at covering CORBA component interfaces. Krishnamurthy and Sivilotti also specifically target CORBA components, and present a method for specifying and testing progress properties of components [66]. Williams and Probert apply techniques of pair-wise interaction testing to component-based systems [112], [113]. By its nature, work targeting distributed component systems is restricted to a limited level of distribution, since much of the complexity of developing these systems is handled by the container infrastructure.

We now discuss some of these areas in more detail.

2.1.1 Distributed Testing Frameworks and Testbeds

One focus of testing distributed applications has been the development of testbeds that are appropriate for evaluating large-scale distributed applications.

PlanetLab [89] is a popular global-scale testbed for distributed systems. To use PlanetLab, an engineer interacts with each of the hosts associated with their “slice”, using SSH to

perform the necessary configuration and execution. There are hundreds of PlanetLab nodes spanning the globe. PlanetLab relies on the actual conditions of the global Internet to provide variability and realistic (by definition) conditions.

Another fully featured testbed for distributed systems, Netbed/Emulab [111], is designed to operate on a cluster of computers all occupying the same local network. In order to achieve “realistic” conditions within the cluster, Netbed provides an integrated emulation and simulation environment that supports the use of traffic-shaping techniques. The Netbed management system provides ways for the engineer to distribute software onto the hosts used in their experiment.

Neither of these systems provides explicit support for testing. However, with the integration of some additional management and reporting software, they would provide test engineers with many of the tools they need to exercise a networked service under varying network conditions.

2.1.2 Early-phase Performance Evaluation

Another aspect of distributed application testing that has received significant attention from the research community lately is performance evaluation of systems based on architectural and design specifications. For example, Grundy, Cai, and Liu [51] discuss the SoftArch/MTE system. This system enables the automatic generation of prototype distributed systems based on high-level architectural objectives. These generated systems are deployed on a dedicated testbed and performance metrics are gathered while the system is exercised. The SoftArch/MTE system is targeted at prototype systems created early in the design process.

More recently, Denaro, Polini and Emmerich have presented similar ideas that are targeted at generating test cases for exercising middleware configurations as a surrogate for the functioning system during early phases of development [32].

2.1.3 Distributed Component Unit Testing

In the testing of a complex system, unit testing is intended to verify the most basic program units in isolation before more complicated and comprehensive tests at the integration- and system-level are conducted. Unfortunately, some modules simply cannot be tested in complete isolation since they depend heavily on other modules for a portion of their functionality. For example, in an object-oriented paradigm, if one class delegates part of its core functionality to another class, it is difficult to test this top-level object in isolation. A technique for dealing with this, called “mock object testing”, grew out of the extreme programming movement [83]. In mock object testing, stub classes are created that allow the tester to control the behavior of a dependency so that all expected behavior of the low-level module can be mimicked.

While mock object testing was not specifically developed for networked systems, it has proven quite useful for testing distributed component software where the modules are intended to run inside a container like a servlet engine or CORBA object request broker (ORB). In these types of systems, hooks into the container are usually provided through an object that is passed in at run-time. Testing inside a running container can be time consuming and complicated, so mock versions of the container hooks are created to allow unit testing of these classes without running any container software.

2.1.4 Testing of Concurrent Systems

PLASTIC WP4 validation is also related to a wide spectrum of validation techniques for concurrent systems (e.g., [23], [36], [74], [101]) aiming at the detection of concurrency faults, in particular execution sequences and schedules. In general, these techniques can be

classified in two groups: those that sample over non-deterministic runs, and those that attempt to create specific deterministic runs.

The first class of solutions involves executing the program repeatedly over the same inputs in the hope of exercising a reasonable percentage of the possible synchronization events. The ConTest tool [36], for example, inserts random perturbations around concurrency relate structures in the program to induce interleavings of threads that were not manifested with the original test suite. This approach is relatively inexpensive to put in place, but it cannot guarantee that even a reasonable subset of the interesting scenarios are exercised.

The second class of solutions (e.g., [23], [101]) deterministically replays a chosen set of synchronization sequences. This approach generally requires specific tool support, and its effectiveness is dependent on the tester's selection of sequences.

2.1.5 Testing Software Distributed on Mobile Terminals

A key aspect of B3G applications is the inherent mobility of terminals and devices on which the Software Under Test runs. As B3G networks will incorporate different wireless technologies, software running on mobile terminals must be possibly tested in all the networks to which the terminals could be moved and connected to, since we need to validate how the services can interoperate in the various contexts, and also that the required QoS is provided. This may increase exponentially the difficulty of B3G application development and testing. There does not exist much work in the literature about how the testing for such aspect can be tackled. One author who has addressed specifically this concern and proposed an innovative approach is Ichiro Satoh [97]. Satoh identifies four approaches for testing applications running on mobile terminals: i) test the software while running on the moving terminal; but the terminal could have limited computational resources; ii) test the software on a workstation emulating the moving terminal and physically brought to the real external environment; but it may be difficult to bring the workstation to the LAN where the terminal should move; iii) simulate the external environment also in the workstation; but this simulator is quite complex to achieve; iv) using the workstation emulator and remotely access the actual servers; but there might be protection problems.

Satoh [97] presents a further innovative solution, called the Flying Emulator, in which the physical mobility is emulated through logical mobility, by means of mobile agents. The mobile agents/emulators can travel between computers connected to the networks to which a mobile terminal is also connected, carry the target software, and allow it to access services provided in the same way as if it was moved with and executed on the actual terminal.

Notably, such a solution could be also adopted by Plastic for off-line testing of mobility; for the moment such a solution is not being considered, and we rely on the on-line testing approaches for testing mobility concerns.

2.2 Web Services Testing

Web Services is still an immature discipline in intense need of research by academia and industry. In WP4, we are concerned particularly with testing approaches. Indeed, while nowadays on the practitioner's side WSs are evidently considered a key technology, research in this area does not seem to draw an adequate attention from the testing community. This might be due to the contiguity/overlap with other emerging paradigms, especially with Component-Based Software Engineering (CBSE), or perhaps to the quite technical details that this discipline entails. In this section we give a brief overview of related work in the area of testing of Web Services.

The resemblance of the WS domain to the Component-based (CB) domain makes it possible to reuse some of the results and approaches proposed in testing CB. Nevertheless testing WS is somehow more challenging. In particular the lack of full control over the development

process, which is an important characteristic of the CB domain where components are developed by different stakeholders, is now extended to all phases of the software life-cycle. A service application results from the integration of several services, generally controlled and owned by different organizations. As a result, speaking in general terms in the service domain an application is never under the full control of one organization. Thus, the services composing an application can change at run-time without informing all the other services integrated in the application. Therefore it is true that some results from the CB domain can be reused, nevertheless this “reuse” has to be done carefully to adapt to the difficulties arising when considering the discussed new scenarios.

As happened for CB, software testing of service oriented systems requires a revision of the traditional testing phases. In the PLASTIC vision, three different phases can be identified in the testing of services (as further expanded in the rest of this document). In the first phase, called off-line, the service is tested by its developer trying to anticipate possible usage scenarios. In this phase services required by the service under test are “simulated” by the testing environment. In a second phase, referred to in this document as Audition, the service is tested when it asks for registration to a directory service. During this phase the service will interact with real implementations of the required services reproducing in some way a real scenario of usage. Nevertheless services are generally bound at run-time so it is not always easy to predict which will be the services that will cooperate to complete a particular task. Main consequence of this reduced capability of prediction is the recognition of the necessity in this new domain, of combining testing techniques with other methodologies that permit to continue to check the behavior of a service oriented system also at run-time, i.e. Monitoring. This section provides a general overview of the first two phases, instead next section is focused on the monitoring phase.

2.2.1 Involved Actors

In [22] the authors provide a high level discussion on the opportunities and challenges for services and service-centric systems. In their work the authors identify five main actors that can be concerned with testing of services. According to their classification the five actors are:

1. Developer: an organization that implements the service;
2. Provider: an organization that deploys the service and makes it accessible to possible users;
3. Integrator: an organization that provides services integrating (i.e., orchestrating) services provided by other parties;
4. Third-party: an organization that certifies the quality of a service in different contexts;
5. User: an organization that is interested in using a service.

In testing a service, each of these actors pursues different objectives and can apply different strategies (actually a longer list of stakeholders of WS testing which we foresee is provided in Table 2: Possible stakeholders for service validation, p. 42). It is also noticed that in some cases particular constraints, such as when a piece of software is running on a small device, strongly hinder the execution and the feasibility of a testing session. In the cited paper the authors also discuss the importance of introducing monitoring techniques to validate service interactions at run-time.

2.2.2 Approaches to Off-line WS Testing

As stated above, off-line testing is carried on by the developer, or possibly by the provider of the service, in order to assess that the implementation of the services actually behaves as expected. In the service oriented domain the specification of the behavior could have been defined by the developers themselves or even by some external bodies, as in the case of choreographies. In the latter case the developer has to retrieve the specification of the service s/he wants to implement from the Net. Also the specification of the services that will

be used by the service under development is usually retrieved from the Net using in general some kind of discovery service. Depending on the specification used, testing can be carried on during the off-line phase in different ways. In particular if the specification is defined by the developer itself, the testing phase probably does not show many differences with respect to traditional testing. In this case the developer has to create stubs that will simulate the behavior of the expected accessed services. More interesting seems the possibility of using already developed specifications. For instance in [42] the authors show how the specification of a choreography can be used to test a service implementation. In their approach they automatically derive test cases from a specification, given as a Symbolic Transition System. Another approach targeting a similar objective, using however a different notation, is discussed in [10].

An interesting approach to the off-line unit testing of orchestration description, defined using BPEL, is described in [71]. In the paper the authors provide a testing architecture that permits to recreate an environment useful for testing the BPEL process. In particular the BPEL4WS service (called Process Under Test – PUT – in the paper) will interact with a set of Test Processes (TPs) simulating the behavior of the composed services. The authors discuss advantages and drawbacks of having centralized or distributed implementation of the TPs. Finally the architecture proposed foresees the presence of a Control Process that in some way allows the TP to coordinate with each other in order to have a global control on the invocation sequences.

Other approaches have been proposed posing the focus on the use of orchestration and/or choreographies in order to check before run-time the possible emergence of failures. In particular the use of such information as main input for analysis activities is considered in [45]. The objective of the authors, in this case, is to formally verify, rather than testing, that some undesired situations are not allowed by the collaboration rules defined by a service orchestration definition. To do this, the authors, after having translated the BPEL4 specifications into Promela (a language that can be accepted by the SPIN model checker), apply model checking techniques to verify if specific properties are satisfied. Another approach to model-based analysis of WS composition is discussed in [40]. From the integration and cooperation of WSs, the authors synthesize Finite State Machines and then verify if the obtained result and allowable traces in the model are compatible with that defined by BPEL4-based specification.

2.2.3 Testing by Enhancing the Directory and Discovery Service

The above list of five actors concerned with testing of services does not include the directory and discovery services. Nevertheless some test approaches in the literature suggest to augment the discovery and directory service with testing functionality. The very general idea is that the service can be tested at the time it asks for registration. Main advantages of such an approach is that tests are executed in a real execution environment providing more realistic results. Moreover, following such an idea, only “high quality” certified services will be guaranteed to pass the registration. In a semi-open environment this will provide some extra guarantees to the registered services. Nevertheless, in a completely open environment the discovery service could be cheated by a malicious service provider that after registration modifies the code associated to the end point.

Approaches belonging to this class can be differentiated mainly on the base of the information used to carry on the testing session. The possibility of enhancing the functionality of a UDDI service broker with logic that permits to perform a testing step before registering a service has been firstly proposed in [101] and [105], and subsequently in [52].

This idea is also the basis for the Audition approach which has been adopted within the PLASTIC validation framework. Audition is an extended framework, originally presented in [19] and [17], which permits to test and monitor a Web Service as it interacts with other services. However, the information the Audition approach uses, and the corresponding

derived tests, are very different from those proposed in the cited papers. In particular, while in the cited works testing is used as a means to evaluate the input/output behavior of the WS that is under registration, the Audition framework grants the registration also on the base of the interactions occurred between the WS under registration and the providers of services already registered. For doing this the framework reuses some of the monitoring mechanisms used for run-time checking. More information on how this phase will be carried on within PLASTIC is described in Chapter 5.

With reference to the information that must be provided with the WS description, the authors of [101] foresee that the WS developer provides precise test suites that can be run by the enhanced UDDI. In [52], instead, the authors propose to include Graph Transformation Rules that will enable the automatic derivation of meaningful test cases that can be used to assess the behavior of the WS when running in the "real world". To apply the approach they require that a WS specifically implements interfaces that increase the testability of the WS and that permit to bring the WS in a specific state from which it is possible to apply a specified sequence of tests. The Audition Framework instead foresees the availability of specifications similar to UML state machines [17], called Service State Machines (SSMs) (see Section 3.4.1.1) and does not require that the implementation provides any particular testing interface to increase testability.

Indeed the classification of testing phases presented here does not foresee the possibility of activating testing campaigns during run-time. In some way in the present PLASTIC framework run-time testing (monitoring) is seen as a passive activity. In literature some approaches for run-time active testing have been proposed, for instance in order to select the best service among the registered services providing the same functionality [106]. As for the case of Audition, run-time active testing requires precise support from the platform in order to distinguish testing phases from "normal" phases, in particular when stateful resources are involved in the process. Within the project we are not targeting, at the moment, any approach following such paradigm, and the topic is not discussed further in this document.

2.2.4 WS-I Initiative

Finally it is important to spend a few words on the WS-I initiative [114]. WS-I is basically an industry organization chartered to facilitate WS interoperability. As a main outcome this organization released a specification, called the basic profile, composed of a list of rules that refine and put constraints on the combined use of specifications such as WSDL, UDDI, SOAP. At the same time a testing architecture has been developed to check the use of elements from such specifications. In particular such architecture is composed of a WS-I analyzer and a WS-I monitor, that applying the man-in-the-middle paradigm, intercepts all the messages exchanged by services, and pass them to the analyzer that checks the conformance to the profile. This approach is certainly relevant to assure interoperability among different services, nevertheless it is not concerned with the functional and extra-functional verification of a service implementation.

2.3 Runtime Monitoring

Monitoring is a runtime activity which collects data and transfers them to external functions for elaboration. The definition of data to collect and the functions to apply are given separately, possibly dynamically.

In the rest of this subsection, we survey related work on runtime monitoring, including both academic and industrial approaches. The systems we survey here are first characterized in terms of what they intend to monitor and then on the basis of their logical architectures.

2.3.1 Assertions Monitoring

Baresi et al. [11] propose an approach to monitor dynamic web service compositions, described in BPEL, with respect to contracts expressed as assertions. These are checked at runtime, by invoking an external monitor service. In [12] they extend this work by introducing the concept of monitoring rules, a language (WS-CoL) to specify constrains on execution, the capability of setting the degree of monitoring at run-time and a proxy-based solution to enact the monitoring rules.

A similar approach is proposed in [38] where the BPEL process is made fault tolerant by automatically identifying and monitoring desired services and replacing them upon failure, using a proxy service for discovering and binding equivalent web services that can substitute monitored services.

2.3.2 Requirements Monitoring

A framework for the runtime verification of requirements of service-based software systems is described in [79]. Requirements can be behavioral properties of a service composition, or assumptions on the behavior of the different services composing the system. The first can be automatically extracted from the composite process, expressed in BPEL; the latter are specified by system providers using the event calculus. System events are collected at runtime and stored in an event database; checking is done by an algorithm based on integrity constraint checking in temporal deductive databases.

In [94], requirements are expressed in KAOS [28] and analyzed to identify conditions under which they can be violated. If such conditions correspond to a pattern of events observable at runtime, each of them is assigned to an agent for monitoring. At runtime, an event adaptor translates SOAP messages into events and forwards them to the corresponding monitoring agent.

2.3.3 Monitoring for Planning

Lazovik et al. [68] present a planning architecture based on the concept of a continuous interleaving of planning steps and execution steps. Within this framework, several kinds of properties can be monitored: properties that must be true before transitioning to the next state, invariant properties which should hold for the entire process execution, properties on the evolution of process variables.

A framework for planning the composition and monitoring the execution of BPEL web services is defined in [91]. Planning techniques are exploited to synthesize a monitor of a BPEL process, which detects and signals whether the external partners behave consistently with the specified protocols, by observing interactions with the external services.

2.3.4 Interactions Monitoring

Li et al. [70] present a framework for monitoring the runtime interaction behavior of web services and validating the behavior against their predefined interaction constraints. Monitor intercepts and analyses messages exchanged between a service and its clients, and validates the message sequence against the interaction constraints specifications.

2.3.5 Logical Architectures for Monitoring

Many of the architectures presented above have a monitoring component, e.g. a proxy, which represents a potential bottleneck. Moreover, some of the approaches presented above, require the modification of the workflow process, which could be error-prone and cumbersome.

An alternative architecture for integrating monitoring, property checking, and possible reactions may be based on an aspect-oriented programming approach. Instead of modifying the workflow process, it is possible to dynamically weave the needed actions into the code of the run-time workflow engine. This approach has been explored by Courbis and Finkelstein [26], who solve the problem by using non-standard and proprietary workflow engine and AOP tool.

Another use of AOP for monitoring is described in [108]: a selection mechanism for web services is proposed that allows dynamic switching between services, based on business driven requirements that can change over time. The selection procedure is triggered by a monitoring mechanism which observes criteria such as cost, presence on approved partner list, binding support, quality of service classifications, historical performance and proximity. AOP techniques, in the form of *monitoring aspects*, allow to dynamically insert measurements points in the system and to execute the monitoring logic tailored for a desired property.

2.3.6 Some Industrial Proposals

Cremona [77] is a tool from IBM devised to help clients and providers in the negotiation and life-cycle management of WS-Agreements. It provides a component, the “Status Monitor”, which helps in deciding whether a negotiation proposal should be accepted or refused, on the basis of system’s available resources and the terms of an agreement. Once an agreement has been accepted by the client and the provider, its validity is checked at runtime by a “Compliance Monitor”, which can check for violations as they occur, predict violations that still have to occur, and take corrective actions.

Colombo [27] is a lightweight, optimized middleware for service oriented architectures supporting BPEL, also proposed by IBM. One of its features is the support of declarative service descriptions, such as those expressed using WS-Policy. It intercepts messages before they leave the system or before they are processed, and use a pipe of dedicated policy-specific verifiers to validate messages with respect to a certain policy.

GlassFish [54] is an open-source community implementation of a server for JavaEE 5 applications. It allows for collecting data on response times, throughputs, numbers of request, and message tracing, of the deployed services.

IBM Tivoli Composite Application Manager for SOA [58] uses an event-based collaboration paradigm, implemented through a special purpose integration bus. Special components inside the bus can be used to monitor the behavior and the performance of messages.

2.4 QoS Validation of Networked Services

The term Quality of Service (QoS) refers to the extra-functional requirements, such as latency, reliability, workload, etc, associated with the service specification; these requirements can be subject to an agreement contracted between clients and service providers. In the previous section an overview of existing approaches to run-time monitoring of the functional behaviour of WSs has been provided; here we augment such overview with a focus specifically on monitoring aimed at verifying contractually established QoS properties.

2.4.1 Predictive vs. Empirical Techniques

In recent years, great efforts have been spent in deriving methodologies for the elicitation of extra-functional requirements, in defining expressive annotation methods, and in the development of methodologies for early performance analysis of software systems. Such studies can be grouped in two broad and orthogonal classes of techniques that combine software design, software development and performance engineering, and are generally

referred as predictive techniques and empirical techniques [55], respectively. The basic idea of the predictive techniques is that performance cannot be retrofitted: it must be designed into software since the beginning [99]. Thus performance prediction guides the design, providing hints on whether the proposed software solution is likely to meet the desired performance goals.

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

Both approaches have pros and cons, and they should not be seen as alternative but rather as complementary. Predictive approaches play a very important role during the design and the development of a generic software system. In fact, their use can have a drastic impact and produce great benefits on the quality of the final product. Nevertheless, modern applications are deployed over complex platform (i.e. middleware) introducing external factors not always easy to model. In such contexts, empirical approaches could be a more reliable approach since they can provide more realistic estimates. On the other hand, they require the existence already of a running system.

However, the testing of extra-functional properties needs to address issues quite different from those concerned when doing functional validation. In the general case, such approaches require the development of expensive and time consuming prototypes [73], on which representative benchmarks of the system in operation can be run.

Fortunately, much progress has been done in the direction of providing automated support for the development of test environments. Specifically, computer-readable specifications can be used in order to describe in detail both the software under evaluation and the expected environment. Today mechanisms acting as code-factories are often available. Such tools can automatically generate a running prototype from a given specification. In our view, when these technologies can be assumed to be applied, there is a large room for the development of empirical approaches for the evaluation of a system within the context where it will be used.

2.4.2 SLA-based Monitoring

Keller and Ludwig [62] present the Web Service Level Agreement (WSLA) framework to define and monitor SLAs, focusing on QoS properties such as performance and costs. The monitoring component is made up of two services: the first, the *Measurement Service*, measures parameters defined in the SLA, by probing client invocations or retrieving metrics from internal resources; the second, the *Condition Evaluation Service*, tests the measured values against the thresholds defined in the SLA and, in case of a violation, triggers corrective management actions.

An automated and distributed SLA monitoring engine is proposed in [96]. The monitor acquires data from instrumented processes and by analyzing the execution of activities and messages passing. Collected data are then stored in a high performance database and a data warehouse, to allow the verification of SLAs.

Several other European research projects recognize that it is certainly no longer possible to propose solutions without adequate specification and validation of QoS features, especially in heterogeneous and networked services contexts.

The goal of the Adaptive Services Grid project (ASG) [7] is to develop a proof-of-concept prototype of an open platform for adaptive services discovery, creation, composition, and enactment. ASG explicitly focus on reliable service provisioning with assured quality of service.

SeCSE (Service Centric System Engineering) [98] explicitly asserts that one of the current scientific challenges is focus on moving from simple structural testing to semantic testing. In this sense, the project exploits service specification describing semantics and QoS information in order to guide the test phase proposing tools for the automatic generation and execution of test cases.

The COnverged MEssaging Technology (COMET) [25] consortium is an industry initiative focusing on research around global, open, converged and ubiquitous IP based messaging systems. Among the project's key activities, a special effort is given in the development of models and tools for end-to-end QoS control.

The main objective of INFRAWEBS [60] is to develop an ICT framework consisting of several specific software tools, which enables software and service providers to generate and establish open and extensible development platforms for Semantic Web Service based applications. Also this project includes in the offered features the execution of regression unit tests and QoS tests for running services within the Runtime Subsystem.

The exploitation of QoS-awareness in the context of highly dynamic system is also a key feature of the MADAM (Mobility and Adaptation Enabling Middleware) project [78]. The objectives of the project include the development of an adaptation theory and a set of reusable adaptation strategies and mechanisms to be enacted at run-time. Context monitoring is used as the basis for decision making about adaptation, which is managed to a large extent by generic middleware components.

Supporting the development of innovative context-aware mobile applications will be the main focus the forthcoming MUSIC project [82]. Regarding QoS, the project will assume a rather wide concept of context: beside the typical factors related to service performance and availability, also changes in the user's role, location and environmental conditions (e.g. light or noise) are taken into account.

2.5 Existing Tools for Web Service Testing

In considering testing tools which could be used for the PLASTIC platform, we have focused on frameworks specialized for Web Services. Although WS is not the only paradigm used in the project, this and "traditional" Web applications cover the major part of the applications used by PLASTIC. We do not discuss tools for traditional Web application testing here. They exist for a long time, are better understood, well-settled and extensively described in the literature. Also, they are not very relevant for PLASTIC.

We considered the following features and properties when we compared the WS testing frameworks:

- License

  The majority of Web Service testing tools are commercial. The free/ open-source ones are fewer in number, with less features and not so mature (e.g. serious installation problems.)

- Protocols

  There is a number of protocols and standards at the core of the Web Services, which are considered in all frameworks: HTTP, HTTPS, SOAP, WSDL and UDDI. Only very few other WS-related standards are taken into account, e.g. WS-Policy and WS-Security. Some tools (TestMaker) go beyond this layer and implement testing agents for other protocols as well: SMTP, POP3, IMAP, XML-RPC.

- Functional testing

  This feature denotes the possibility of creating testing scenarios, where Web Service(s) under test interact with other Web Services, test clients or service simulators (stubs). It

may be called use case/ scenario testing, integration testing or something else in other papers.

- Custom scripting

  Almost all testing tools provide a scripting language in which test cases understood (and runnable) by the corresponding framework can be written. Many companies feel the need to come up here with a “just-click-and-go” module (see “User Interface” below).

- User Interface

  The interface presented to the user. It tends to be graphical, some tools (SOATest) seem almost to miss or neglect the command-line driven approach. Some frameworks come with a large collection of code viewers, or some kind of intelligent editors which allow an intuitive view of the documents or scripts involved.

- Simulation

  This means the creation of service stubs, which replace services which are missing or not tested in the current scope.

- Recording

  All tools investigated present a recording feature, where user interactions are automatically recorded and translated into scripting language commands. This is also a feature that test framework developers feel mandatory to implement.

- Governance

  This means enforcing policies on WSDL files, schema files, and SOAP messages by defining rule sets containing policies and best practices that the user wishes to enforce. This allows system architects to define a set of rules they wish to enforce on components of the SOA which in turn can be used by developers and testers during test creation.

  This is somewhat analogous to coding rules and styles. A governance document can include, for example:

  - What subset of WSDL and schema should be used:
    - Follow the WS-I Basic Profile
    - Don’t use ANY
    - Avoid using CHOICE
    - Schema elements must not contain substitution groups
    - etc.
  - They also include sections discussing naming standards such as:
    - All name spaces must conform to a specific pattern
    - Don’t allow more than one definition of “Customer”
    - All names must be less than a given length
    - etc.
  - Specifying restrictions such as a service must not have more than one port

  Governance documents can be conceived as a document file, read and followed by humans, or the rule-sets can be entered and understood by the system, and then automatically verified. Modern products give support for the latter.

- Extra-functional testing

  This - if any - is usually limited to some kind of load, scalability or performance testing.

- Documentation

  We considered the quality of the documentation, which is the first contact of a user with the system. Is it comprehensive, easily browsable, does it have a “Getting started” or tutorial section, etc.

- Extras

  Here we included any other features which were not listed previously in the common area, thus being specific to the tool in question.

Because the WS technology is a new area, there are not many mature tools for WS testing. Some tools (e.g. eTest, Jblitz) claim to test web services, but in reality they are Web application testers. We are going to present 2 commercial and 1 open-source framework. This is not a long list, but it should be enough to get a view of the state of the art of actual solutions and trends in WS testing.

2.5.1 Parasoft SOATest

SOATest from Parasoft is a serious WS testing product, with many features. It aims to verify all aspects of a Web service, from WSDL validation, to unit and functional testing of the client and server, to performance testing. It addresses also some other Web service issues such as interoperability, security, change management, and scalability. It has a Windows native graphical user interface and all its operations are strongly UI-oriented. Most operations are implemented by wizards.

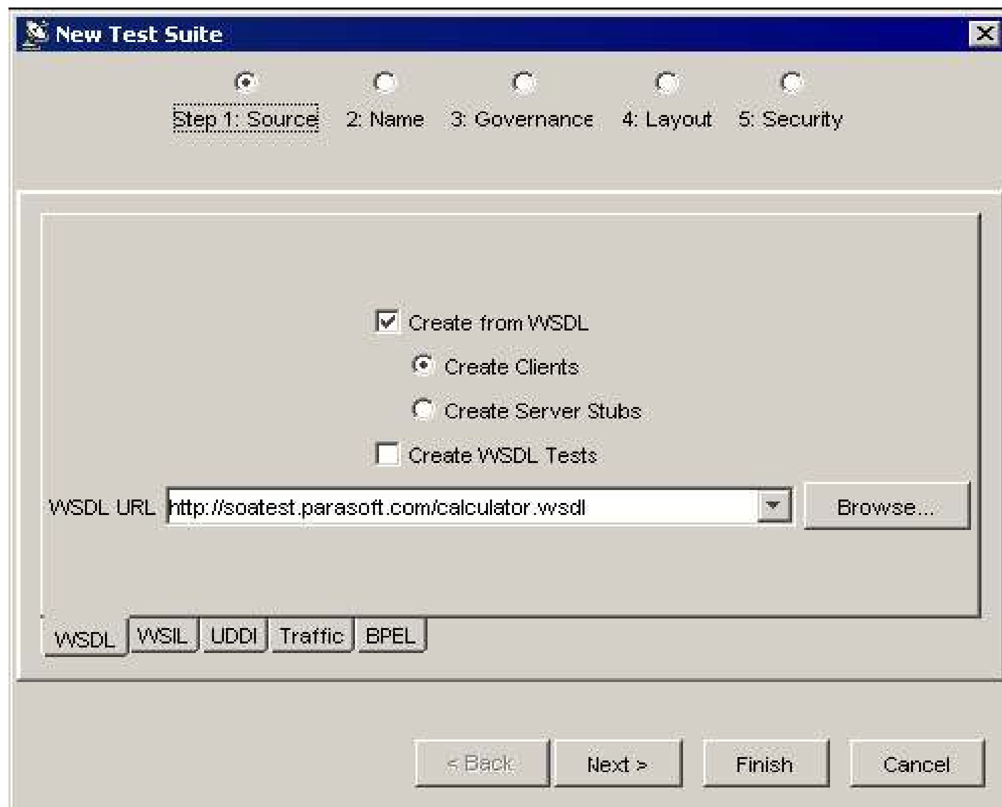


Figure 1: Screenshot from the Parasoft SOATest tool

This tool presents the following features (see the interface in Figure 1):

- **WSDL Tests**

  SOATest can check the WSDL interfaces to ensure that they conform to the schema and pass XML validation tests. Additionally, it performs an interoperability check to verify that the web service will be interoperable with other WS-I compliant services.

- **Unit Tests**

  The program allows the creation by wizard of a test client for each operation defined within a WSDL. These tests can then be moved into separate test suites, creating one test suite for each test case.

  The tests can get their data from external data-sources, e.g. Excel spreadsheets.

  SOAPScope cares of the possibility to generate positive and negative test cases as well, when we send invalid requests to a WS server.

- **Scenario Testing**

  Scenario testing allows creating test cases for a sequence of calls, let's call them conversations. Because there is usually an inherent state of a conversation, holding variable values from one call to another (e.g. the list of goods inserted in the shopping cart so far, or reusing returned values to parameterize the next call), there should be containers in any testing framework which store these state values. In SOATest this is the so-called XML Data Bank. This tool allows the user to extract XML element values and store these values in memory to be used in later tests, or to use external data-sources.

  The creation of scenario tests is a rather complicated, multi-step procedure. Everything is implemented by the UI wizards. The results of a test run are displayed in an html browser.

- **Custom scripting**

  Although SOATest is strongly UI-oriented, you can write test parts or tests in a scripting language. The tool supports Jython (Java enabled Python), Java, or Javascript to be used.

- **Asynchronous Testing**

  SOATest allows the testing of asynchronous web services which use the WS-Addressing, Parlay or SCP protocols.

- **WS-Security**

  SOATest contains security tools and options that fully supports the industry standard WS-Security specification. These are the following:

  - XML Encryption Tool
    Allows to encrypt or decrypt messages using Triple DES, AES 128, AES 192, or AES 256. In WS-Security mode, Binary Security Tokens, X509IssuerSerial, and Key Identifiers are supported.
  - XML Signer Tool
    The XML signer tool allows to digitally sign an entire message or parts of a message. In some cases it may be important to digitally sign parts of a document while encrypting other parts.
  - XML Verifier Tool
    The XML verifier tool allows for the verification of digitally signed documents using a public/private key pair stored within a key store file.

- Key Stores
  The use of key stores in SOAtest allows to encrypt/decrypt and digitally sign documents using public/private key pairs stored in a key store. Key stores in JKS, PKCS12, BKS, and UBER format can be used.
- Username Tokens, SAML Token, or Custom Headers
  SOAtest supports sending custom SOAP Headers and includes templates for Username Tokens and SAML tokens.

◆ **Governance**

You can apply rule-sets to validate newly developed web services, or check other services conformance with given rule-sets. SOAPTest comes with `WSDL.Governance.rs`, which is a collection of industry-wide best practices in WS, but you can use also custom rule sets. These are usually set up by a WS administrator in the WS developer company or organization.

◆ **Load Testing**

SOAtest provides four default load testing scenarios (Bell, Buffer Test,Linear Increase, and Steady Load) or allows you to create your own custom scenario. These scenarios can be created to emulate possible real life scenarios that may occur during normal usage of a web service. Several machines can be clustered to provide a heavy load of a service. The test result statistics can be obtained in a tabular or graphical form.

◆ **Server Stubs**

With these one can mock the behaviour of a server. You can manipulate the response and furthermore, monitor the client's behavior to the server responses. In this way it is possible to isolate the component (service) under test, and to test services not implemented yet. The starting point is "Create from WSDL" and "Create Server Stubs" controls. You can link then datasources to the generated stub.

◆ **Report generation**

SOATest can generate reports in HTML, XML, PDF, and CSV formats.

◆ **Missing features**

There seems to be no recording option for automatic recording of a test case.

◆ **License**

SOATest is a commercial product. It can be obtained for a test/evaluation of 7 days by contacting the developing company.

2.5.2 Mindreef SOAPScope

Mindreef comes with a collaborative platform that allows teams to efficiently work together to design, develop, test, and support services in a service-oriented architecture. SOAPScope includes a server (internally uses Tomcat) and a testing and diagnostic tool which is intended to be used in all phases of the WS development cycle.

User Interface

The interface presented to the user is a web interface (see Figure 2). From it the user can define and use workspaces, which are attached to web services and are used as the testing/debugging infrastructure in the WS life cycle. For development, there are plugins for Eclipse and MS Visual Studio. Almost every object (workspaces, contracts, message invocations) is persisted to database on demand.

Workspaces can be thought of as named containers that are comprised of:

1. Contracts (WSDLs and other associated documents)
2. Messages (sent and received from the Web service)
3. Actions (test scripts)
4. Reactions (simulated operations)
5. Notes (shared with others)

WSDL view (contracts)

WSDL and related documents are called contracts. This emphasizes their interface role. Typically the user imports a contract by giving a WSDL url or file. Then, she can browse through the list of operations defined by this contract and obtain it in several views, selectable by tabs.

The following views are available:

- Overview (shows operations in Pseudocode and lets you invoke them)
- Documentation (a drill-down view of Web service components based on namespace, local name, schema components, and use)
- Files (displays contract components as formatted or raw XML, trees, or graphs)
- Coverage (displays usage statistics)

The Pseudocode and tree views are very useful. There is also some graph view.

After examinig the operations, they can be invoked simply by pushing an “Invoke” button. A form is presented where the user should fill parameter data. A “Send” command button will submit the request and invoke the web service. The response is then presented in the same manner (e.g. Pseudocode).

Figure 2: Screenshot from the Mindreef SOAPScope tool

A message can be resent any times, with different parameters. Invocations are recorded on demand.

Scenarios

To organize the calls in reproductible test cases, SOAPScope uses the notion of actions. Actions are very similar to messages, but you can manage them in a list, introduce constants and variables for recording the conversation state, a.s.o. So actions can be used to build small test scripts to perform positive, negative, boundary testing on a Web service.

Recording

There is a recording feature in the framework, which allows for recording based on the user actions. When running a script back, there is a possibility to pause execution, and to use some debugging facilities.

Simulation

The framework supports the simulation of web service stubs. For this, they introduce the notion of reaction. One can set up responses that the stub will send, delay times, a.s.o. There is quite a good deal of work the implementers put on simulations.

Load testing

The program even permits to invoke actions in a loop with a predefined time delay between calls. However, Mindreef warns that “SOAPscope is not intended to be a load and performance test tool”.

Governance

SOAPScope provides also strong support for governance. It has a prebuilt set of rules and a user can build a custom one. The prebuild set contains Mindreef basic diagnostics and best practices and an interactive version of the WS-I basic profile. There is a very extended support for analysis and design-time governance.

Extras

For those who are accustomed to Eclipse, SOAPScope comes with an Eclipse plugin, which implements some of the web interface functionality. An MS Visual Studio plugin is also available.

2.5.3 PushToTest TestMaker

TestMaker is a framework and utility that builds intelligent test agents which implement users interactions in several environments. This program is developed in Java (so it is multi platform) and the core is available under an open source license.

Protocols

There are handlers available for the following protocols: HTTP, HTTPS, SOAP, XML-RPC, SMTP, POP3, IMAP. It follows that the tool can test web applications, web services and mail systems. The list can be extended by writing new protocol handlers.

The open-source version does not support remote clients for distributed testing, but the software TestNetwork sold by the same company is able to perform remote clients. That is not open-source.

Graphical Interface

The framework comes with a graphical user interface, which includes a syntax-sensible editor, a file browser and a test running toolbar and console (see Figure 3).

Despite of this user interface, to create complex test scenarios it is necessary to write the testing scripts (called agents here) by hand. This is somehow similar to Junit or Htmlunit. The scripting language is Jython.

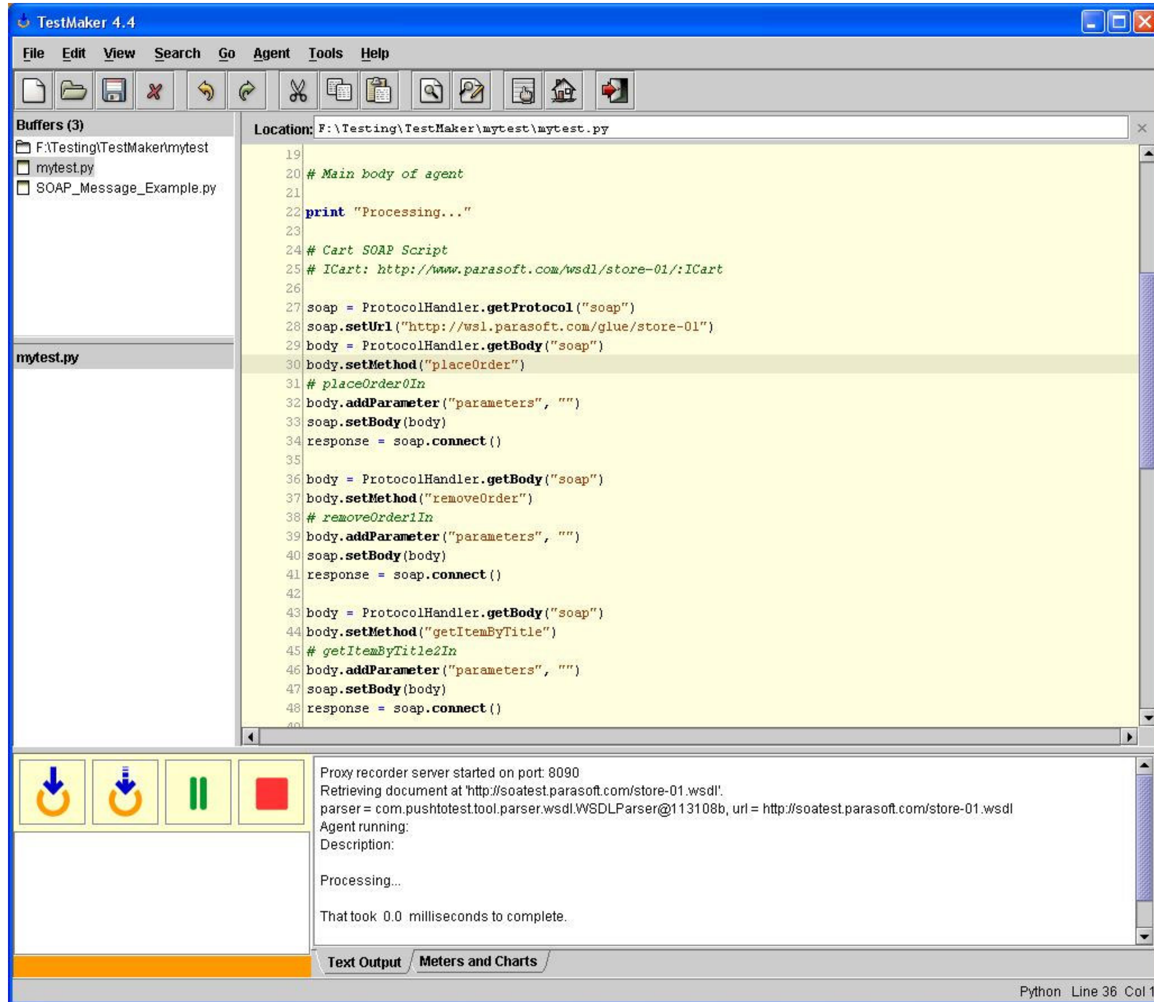


Figure 3: Screenshot from the PushToTest TestMaker tool

Test Agents

TestMaker can generate a test skeleton client from a WSDL file. However, all invocation arguments are empty strings here; they should be filled by the test engineers. To support this, a TOOL library implemented in Java is provided. It provides, between others, datasource objects, for example. To create complex test scenarios, it is necessary to understand Jython and the TOOL library. A number of example agents are provided. The source code is also available.

Recorder

You can set up a proxy in TestMaker which intercepts the http messages you exchange with your browser. This can be used, for example to record an http conversation and to create a test script from it. This is what the Agent Recorder is doing.

Of course, currently this works only for “traditional” web applications, since the Recorder creates an html agent. But this could be extended for SOAP messages as well.

Load test

TestMaker comes with XSTest, a module which allows running and monitoring of multiple agents concurrently. The load scenario should be set up by scripts. After execution, a utility

parses the log files and extracts statistical information. Usually the result code, time to complete, memory footprint, number of concurrent users a.s.o. are recorded.

Conclusion

In summary, this tool is by far not as featured as similar products developed in industry. But it has the advantage of being open-source and extendable by anyone who has software development capacity.

2.5.4 Summary Table

We have collected the main framework features in a comparative table.

	Parasoft SOATest	***Mindreef SOAPScope***	***PushToTest TestMaker***
License	Commercial	Commercial	Open-source core, some parts commercial
Protocols	HTTP, HTTPS, SOAP, WS-Addressing, WS-Security	HTTP, HTTPS, SOAP	HTTP, HTTPS, SOAP, XML-RPC, SMTP, POP3, IMAP
WSDL test	Yes	-	-
Unit test generation	Yes	Yes	yes
Scenario testing	with the user interface	with the user interface	scripting by hand
Scripting language	Jython, Java, or Javascript	-	Jython
User Interface	Graphical (Windows native)	Web (browser), Eclipse, VS Studio plugins	Some graphical (Swing), mainly script-oriented
External datasources	Yes	-	yes, with TOOLS API
Recording	-	Yes	yes
Simulation (server stubs)	Yes	Yes	-
Governance	custom	WS-I Basic Profile, custom	-
WS-Security	XML Encryption, signing, key stores	-	-
Load testing	predefined testing scenarios	Some	In program, commercial distributed environment
Reporting	HTML, XML, PDF, CSV	HTML	TXT
Documentation	Good	Good	distributed, good samples
Extras	WS-Security	Team server, nice message views, IDE plugins	Mail protocols

Table 1: Features of the examined WS testing tools

2.5.5 Discussion

By examining the above table we can derive some conclusions about the current state-of-the-art of service testing in industry. Web service testing is usually expensive: the featured tools are commercial and highly priced. They are also “closed” in the sense that they propose only a given methodology for testing services, for which they develop and provide a good, usually graphical user interface oriented support. In contrast, the open-source programs are less featured but more flexible, the emphasis being on programmability and extensibility.

The above mentioned and similar tools could be used for validating Plastic services, but they have some limitations. First of all, they are mostly optimized for functional testing only. Parasoft SOATest provided built-in tools for some predefined load testing scenarios; TestMaker has a distributed environment available commercially. In the PLASTIC validation framework, we need functional and extra-functional testing. In particular, we deal with QoS testing and validation, which was not found in any of the existing tools we examined.

Another common aspect of the industrial tools is the testing process: it relies on manual test cases, developed by testers, usually based on a deterministic environment setup. In contrast, we focus on diverse service and environment models, which can be developed in earlier phases of the development and enable an automatic testing process. We also consider some non-deterministic environment behaviour. Although testing in this way may still be a preliminary attempt from research, it seems to prospect many advantages compared to traditional techniques. In other terms the proposed PLASTIC validation framework is a research work in progress, which can be of course complemented by the application of existing tools such as the ones surveyed here.

2.6 Summary

In this chapter we have overviewed related work in the areas of interest for PLASTIC WP4 approaches. In particular, we have overviewed problems and technologies related to the testing of mobile distributed concurrent systems, methodologies and existing approaches for runtime monitoring, the issues implied by the validation of QoS, and we have illustrated some existing tools for WS testing. As the extent of the chapter shows, the scope of validating service-oriented application is very broad. In the following chapters, we will build a test and monitoring framework that relies on and complements the surveyed work.

3 Context Setting and Background

3.1 Introduction

Inherently, testing is an activity that asks for the exploitation of the specific peculiarities of the different contexts to which it is applied. This chapter, extending the insights outlined in the introductory chapter, is aimed at providing a general discussion on the application domains that are the objectives of the project investigation, and how validation should be approached within PLASTIC. In particular the next section discusses which are the objectives, the new stakeholders and the new problems that we face in the B3G and service oriented environments. Successively Section 3.3 puts the various approaches that we are developing within WP4 in the context of a possible testing process. Finally Section 3.4 provides some background and introductory material to make easier the comprehension of the following chapters in which the specific solutions proposed are illustrated in detail.

3.2 Problems and Objectives

PLASTIC intends to define and develop instruments that will make the setting of service oriented application within the B3G context easier. Among these instruments particularly relevant are those for verifying that developed software is correct and behaves accordingly to the specifications. The first steps in the definition of such instruments is the identification of possible testing targets. Within the SOA paradigm, three main testing targets have been identified asking for customised testing techniques, i.e.:

1. **Single Services**: different specifications can be considered to assess the implementation of a single service. First possibility is to use a specification defined by the developer of the service. Another opportunity is that the service is actually derived from a specification defined by someone else. In both cases the assumption is that these specifications are available and retrievable from some repository.

2. **Orchestrating Services**: in this case the target of a testing campaign is a service that aims at integrating different services to provide a complex service. The specification for such a kind of service is in general available only to the organisation that developed or deployed it. At the same time using orchestration languages such as BPEL the specification and the implementation coincide. It will not make sense then to apply model-based testing to verify that the implementation conforms to the specification and that the invocations are as specified. What is still relevant instead, is checking that the orchestrating service makes valid invocations on the orchestrated services (with respect to their published interface and specifications). Assuming such services as correct it is possible to verify if the orchestrating service makes wrong assumptions on the used services.

3. **Choreographies**: in this case the objective of the testing phase is to verify that the task specified by the choreography can be actually carried on by the integration of services each one acting one of the different roles foreseen by the choreography. The emergence of a mismatch among expected and actual behaviour highlight that the services integrated are not suitable for the specific purpose.

Within PLASTIC we are mainly interested to the first two categories of scenarios, and in the following some possible solutions for those two cases will be illustrated.

The testing phase is also strongly influenced by the motivation and objective of the stakeholder carrying on the verification phase. In the service-oriented context, the stakeholders possibly interested in testing a service or a set of services are described in the following table.

Service validation stakeholders		
1	**Service Developer**	in order to provide a good quality service the developer tests the service under development to assess the correspondence to the specification. The developer can apply both white and black box approaches. The specification used to derive black-box test cases could be successively made available to service customers.
2	**Service Deployer**	this stakeholder acquires a service from a developer and deploys it on a server, making the service available to possible users. In testing the service the deployer wants to verify that the service actually behaves as agreed with the developer. However, it is not uncommon that the developer and the deployer coincide.
3	**Service Integrator**	aim of the service integrator is to assemble several services to provide to the user a more complex service. Testing is focused on verifying that the assembled services behave as expected and then that the composed service can correctly provide the specified service.
4	**Service User**	when accessing a service the user ultimately wants that his/her expectations are satisfied. Testing could be a useful mean to verify that the service corresponds actually to what he/she needs. Nevertheless as correctly noticed in [22] testing is really an expensive task and really difficult to apply during the discovering and binding of a service, especially when some kind of dynamism is allowed. It is probably more reasonable in this case that the user specifies some parameters concerning the required service and that the service provided by the directory service is in some way "guaranteed" – see next item.
5	**Directory service**	aiming at providing only high quality service to the users, the provider of a directory service can be interested in testing that the services asking for registration are of "good" quality. For this purpose the directory could submit the service under registration to a verification step before granting the registration. Different kinds of specification can be considered for testing purpose. Section 5 discusses in detail the topic providing a possible approach to couple testing to the registration phase.
6	**Standard Body**	release of agreed specifications allowing for choreographies can open interesting scenarios for testing. A choreography allows for specifying high level integration and coordination of abstract services to which possible developers must conform; this would make it possible the establishment of standard organisations interested in developing choreography specifications and at the same time willing to verify that a set of services can actually implement the specified choreography.

Table 2: Possible stakeholders for service validation

Certainly the above listed different stakeholders will use different approaches and techniques to test a service. PLASTIC is developing technologies that in different moments might help all of the stakeholders listed above to verify a defined service, except for the case of the Service User. The rational behind this choice resides in the fact that in the B3G domain the service end users generally rely on computationally limited devices (e.g. cell phones) that are not suitable for carrying on testing activities, and that they certainly have no time or willingness of testing a service before usage. They would like to use high quality services without the necessity of testing them in advance, relying instead on directory services to get high quality services.

The two lists above provided an answer to the questions of “what?” and “who?“ when testing within the service application domain and in particular within PLASTIC. In the introduction we discussed which are the new challenges that we need to solve within the domains considered by PLASTIC and so we answered to the “why new approaches and techniques to testing are required?”. The next section will now provide an introductory overview for the following chapters in which different approaches for the validation of “PLASTIC applications” are specified.

3.3 Adopted PLASTIC Validation Methodology

This section illustrates the different testing approaches that are under development within PLASTIC. What is proposed and discussed in the remainder of this deliverable is not meant to be a closed set of techniques inclusive of “all the necessary” for the testing phase within PLASTIC. The solutions that are proposed in WP4 try to address some of the peculiar testing problems within the B3G and service oriented domain. Nevertheless approaches, techniques and tools for testing developed within other domains could certainly be useful when adapted to the new concepts of this new environment, and then complement the solutions that are proposed here.

The solutions that are under study within PLASTIC cover different aspects of testing and are intended to be applied in different phases of the software lifecycle, as said possibly complemented with other “standard” approaches.

In particular within PLASTIC we classify the lifecycle of software services according to two different main stages: off-line and on-line. The distinction between these two phases has already been discussed in Chapter 1 and mainly relates to the environment in which the software is inserted in a specific moment. Therefore during the off-line stage the software under analysis is inserted in an artificial environment and experimentations will be carried on in such a context. Obviously to have reliable results the reproduced environment will try to simulate as closely as possible the characteristics of a real environment. On the contrary during the on-line stage the software is inserted in a real environment. In such a context the evaluation can certainly be more realistic but at the same time suitable mechanisms are required to avoid that the insertion of instruments for testing and validation purpose within the real environment could influence the behavior of the whole system in terms of provided functionality and QoS.

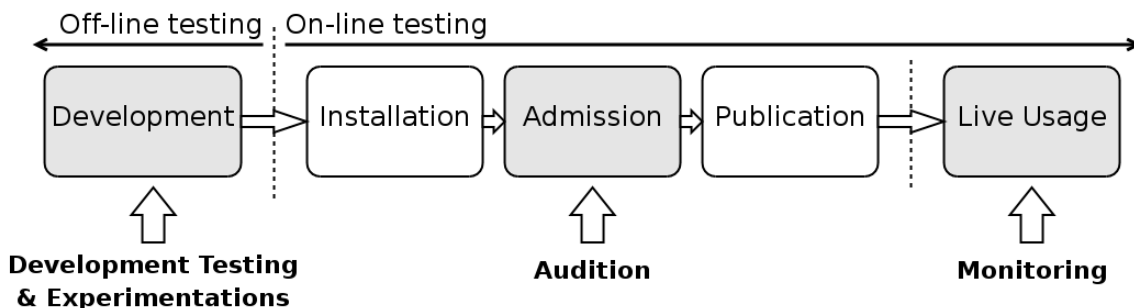


Figure 4: PLASTIC testing stages

Figure 4 shows the two main stages of the software lifecycle, and also identifies the main steps within the on-line stage: From a tester’s viewpoint, it is important to distinguish whether the deployed service is already published and available to the end users, or not yet. As

shown in Figure 4 for testing purpose the on-line phase has been in fact split in two different sub-phases. The first one is related to the deployment of the software and the second one to the real usage of it. In turn the deployment phase, which is targeted to bring the software in a status that permits other services to access it, has been subdivided into three different steps. The first one concerns the installation of the software on a real machine and in general refers to the activities of copying software artefacts in specific locations. The third deployment sub-phase is the publication and is referred to the registration of a software access point within a directory service. If the publication ends successfully the software will be available to other for access. Hence within the On-line stage we identify Installation, Admission, Publication and Live Usage. In grey we show the boxes corresponding to the subphases for which PLASTIC will provide specific solutions. To each one of these boxes a separated chapter is devoted in the remainder of this document and in the next section introductory material is provided. The solutions proposed will focus both on functional and QoS validation aspects. More in detail Table 3 shows the approaches and techniques that will be used in the different sub-phases of Figure 4.

In brief, for the off-line testing stage PLASTIC will rely on two different approaches for functional testing and one approach for QoS evaluation. The first approach to functional testing is called AMBITION. This is a tool that is based on the application of model-based techniques to the service oriented domain. The application of such an approach to testing of services seems particularly appealing given the largely recognized necessity of having richer specifications of services. Nevertheless the application of formal model-based techniques requires the adaptation and extension of the available theory of formal testing, which is a very challenging and demanding effort.

	Functional	**Extra-functional**
Off-line (Development)	Ambition (Model-Based Testing) Simulation based Testing (+Distributed experimentation)	Puppet (+ Distributed experimentation)
On-line (Admission)	Audition (Reuse MBT)	
On-line (Monitoring)	Functional Monitoring of Orchestrated Services	Timed Automata-based Monitoring Adaptive QoS Monitoring

Table 3: PLASTIC proposed solutions

The Simulation based testing approach wants instead to provide a mean for the evaluation and optimization of test suites. Within the B3G and service domain, testing becomes particularly expensive requiring for instance the availability of costly infrastructure. The reduction and minimization of test suites become then particularly relevant.

Puppet is a tool that permits to automatically recreate a reliable test harness for the QoS evaluation of services. It is based on the automatic generation of stubs that show predefined “QoS behaviour”. Chapter 4 will deeply discuss Off-line testing approaches and techniques.

We then notice that both functional and extra-functional validation need to rely on top of an environment allowing for distributed experimentation.

The second subphase of the deployment step in Figure 4 is peculiar of the PLASTIC testing approach. The idea underneath this subphase is to associate a testing session with the registration and publication process. In such a manner the publication of a software end-point will be guaranteed only to high quality software probably reducing the risk of run-time errors. Chapter 5 describes Audition, a specially conceived framework that applies the idea of testing during the publication step.

Finally PLASTIC will provide instruments for the verification of software behaviour during the live-usage sub-phase (see Figure 4). The basic idea in this case is to develop mechanisms that permit to observe the software behaviour during real executions. Observations include functional aspects and also QoS parameters. Chapter 6 is devoted to the description of the approaches proposed for functional and QoS monitoring.

A big issue PLASTIC is facing concerns the introduction of mechanisms for the evaluation of context-awareness and adaptability before the live usage phase. Even though the monitoring of the systems at run-time provides real evidence of the behaviour of a software element also for what concerns adaptability, in general the effects produced by wrong behaviour are not always easy to remove. Indeed the approaches to testing of a service before live usage phase currently defined within PLASTIC do not allow for the prediction of context-awareness or adaptability properties of the service, as trying to tackle such qualities off-line is extremely expensive. Nevertheless it would be certainly helpful to have mechanisms for the verification of context-awareness and adaptability before real usage of a software. In this line PLASTIC is conducting several studies trying to introduce mechanisms for context-awareness verification before the live-usage phase. The general idea that is under study concerns the possibility of defining adaptation policies within a formal model. Such a model should be successively used to derive test cases and to reproduce different contexts. At the moment this deliverable is written, research on this subject is actively pursued, and interactions with both WorkPackage 2 and WorkPackage 3, where mobility and adaptation are being specified and implemented, respectively, are ongoing to tackle this point.

3.4 Modelling

In this section we provide some preliminary background behind the techniques and tools introduced in the following Chapters.

3.4.1 Behaviour Models

Off-line approaches adopted in Plastic follow the predominating approach of model-based testing. In particular, we are considering two different classes of model-based testing: the first, more conventional way to conceive model-based testing, refers to state-based models of expected behaviour, in particular we have adopted Service State Machines (SSMs), as described in the next section; the second class refers to a simulator not only for design purposes but also for guiding test selection.

3.4.1.1 Service State Machines (SSMs)

SSMs are a variant of Symbolic Transition Systems [43], dedicated at specifying Web Services (WSs). Similarly to UML State Machines every SSM consists of states and transitions between the states. Such a machine models "the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events" [61]. The control and data flow is specified as it takes place at the WSDL interfaces, the implementation of the interface methods is regarded as being hidden in a black box. By so doing the SSM model can be exerted for black-box testing of WS.

In its simplest case an SSM specifies a single port[1] of a single WS. In this case only the control and data flow is specified as it takes place at the given port. This view is sufficient if the WS is basically a stand-alone service, meaning that it does not depend on other WS to provide its functionality.

As we anticipated in Section 3.2, scenarios where a WS does depend on other WS can be divided into two classes. First, a bunch of (usually distributed) WSs can communicate which each other following a so called *coordination protocol*. Here the protocol defines at least the control flow, meaning the legal ordering of messages as they are exchanged between the WS. It describes the externally visible behavior and should be publicly available in registries. It usually does not reveal the precise data flow and is therefore not executable. It can be used for monitoring the legal ordering of method invocations, but not for active testing due to the lack of data flow. It serves as a blueprint for implementing a service which has to match the protocol to play a certain role in a coordinated setup. Such a setup is also called a Choreography of WS. To specify an only control-flow oriented choreography we use so called Coordination-SSM [41].

Second, a WS can be implemented via composing other services. Hence such a composition is defined in models which have to be executable. Such a composition can be done via conventional programming languages or by an executable composition language. In the latter case the middleware has to support the composition language by a corresponding composition engine. The Business Process Execution Language for Web Services (BPEL) is such a composition language. A composition model is not publicly available since it reveals the implemented business logic. A composed service is also called an Orchestration.

SSMs can cover the data and control flow between several services and ports, and they are flexible in the way they reveal data information. In one extreme they describe the mere control flow (as a coordination protocol). In the other extreme they additionally describe the complete data flow. But also intermediate cases are possible, allowing the modeler to deal with nondeterminism and incomplete information about the participating services. This is crucial in a SOA setting, where one cannot expect to have access to implementation details of all services. For some only a very general, external description of the offered service might be available, and some services might be totally out of the control of the modeler. Furthermore, giving the complete data flow of a coordinated setup of services might simply be a too complex and time-consuming task. To specify data-enriched scenarios we use so called Multiport-SSM [41].

Summarized, SSMs can be used to model Choreographies with a desired amount of data information, and Orchestrations of services. Within Plastic the latter seems dominating, although the nextcoming integration stage will better evidence which specific scenarios will be of importance.

In Figure 5 an exemplary setup of WS is given as an UML 2.0 component diagram. Such a diagram can be seen as a graphical representation of a WSDL-file. Asynchronous interface operations are denoted in an extra <<signal>> compartment. The diagram describes the static aspects of a procurement scenario adopted from [2] which is briefly summarized next. The Customer WS can connect to the Supplier WS to gather information about the price, availability, delivery dates etc. of goods by means of the requestQuote() operation. Then it places the order by an orderGoods() operation. The Supplier WS checks at the Warehouse WS whether there is a shipment available by a checkShipAvailable() operation. Based on the result the Supplier WS responses the Customer WS a cancelOrder() operation (e.g. he ran out of stock) or confirms the order by a confirmOrder() operation. If the supplier can deliver, the following sequence occurs: the customer pays (makePayment()), the shipment is ordered at the warehouse (orderShipment()), the shipment details are provided to the customer

[1] A port is also called an *EndPoint*, i.e., a concrete binding of an interface to a network address.

(getShipmentDetail()), the customer confirms its order (confirmShipment()) and the warehouse confirms the shipment to the supplier (confirmShipment()).

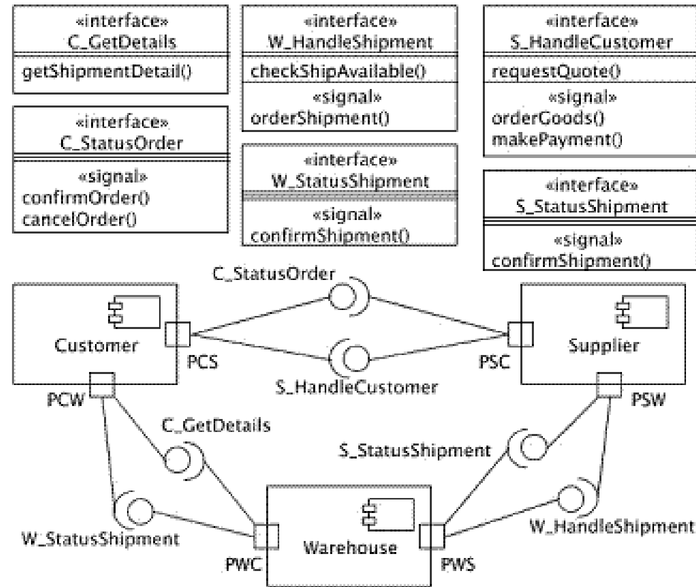


Figure 5: Exemplary setup of WS

SSMs can now be used to specify several aspects of this scenario. For instance one might be interested in the intended Choreography as being described above in plain text. For very simple cases sequence diagrams (as in Figure 6) suffice here:

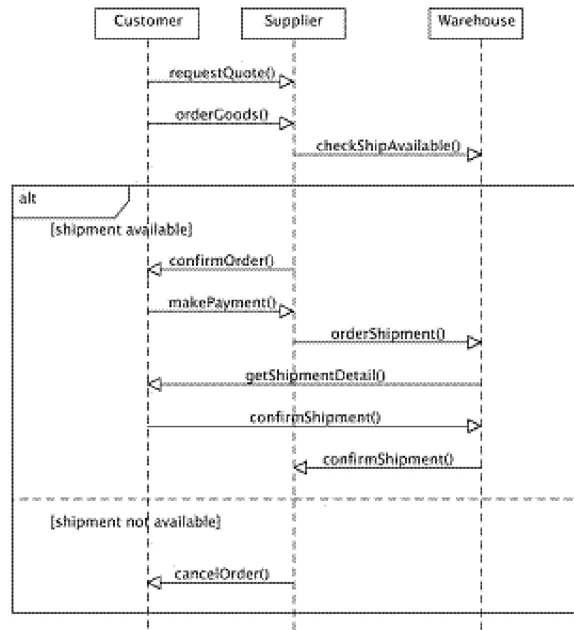


Figure 6: The choreography as a Sequence Diagram

Since the UML 2.0 it is possible to describe alternative behavior in one single sequence diagram via the “alt” conditional execution, however, this stretches the diagram for every described alternative. Also, cycles and complex dependencies cannot be modeled. Hence, sequence diagrams are not suited to specify (complex) alternative behavior; we just present this one here to illustrate the protocol also with a model which might be more familiar to the reader. In Plastic we intend to use instead Coordination-SSM. Every Coordination-SSM is attached to a corresponding Component Diagram. The next picture (Figure 7) is a Coordination-SSM describing exactly the same scenarios as the upper sequence diagram:

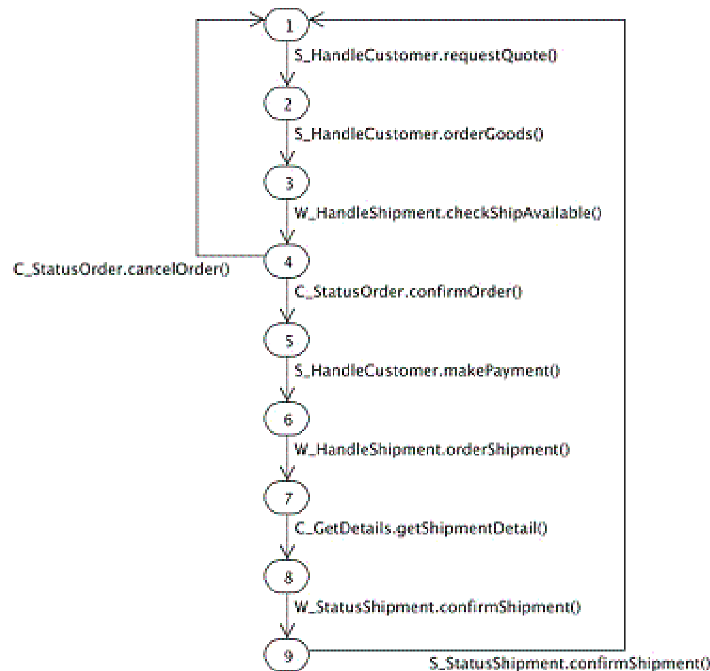


Figure 7: The choreography as a Coordination-SSM

The shown Coordination-SSM consists of nine states. The state labeled by 1 is the initial state. Every label at a transition refers to an interface method which is referenced by writing the common dot-notation interface-name.interface-method(). By giving the interface method the location where the message appears within the coordinated setup is precisely specified. For instance referring to S__HandleCustomer.requestQuote() specifies the method requestQuote() which is sent by the PCS-port of the Customer to the PSC-port of the Supplier via the S_HandleCustomer interface.

Such a Coordination-SSM can be further refined and enriched with data to represent an executable Multiport-SSM. Figure 8 represents such a Multiport-SSM focusing on the Supplier-role:

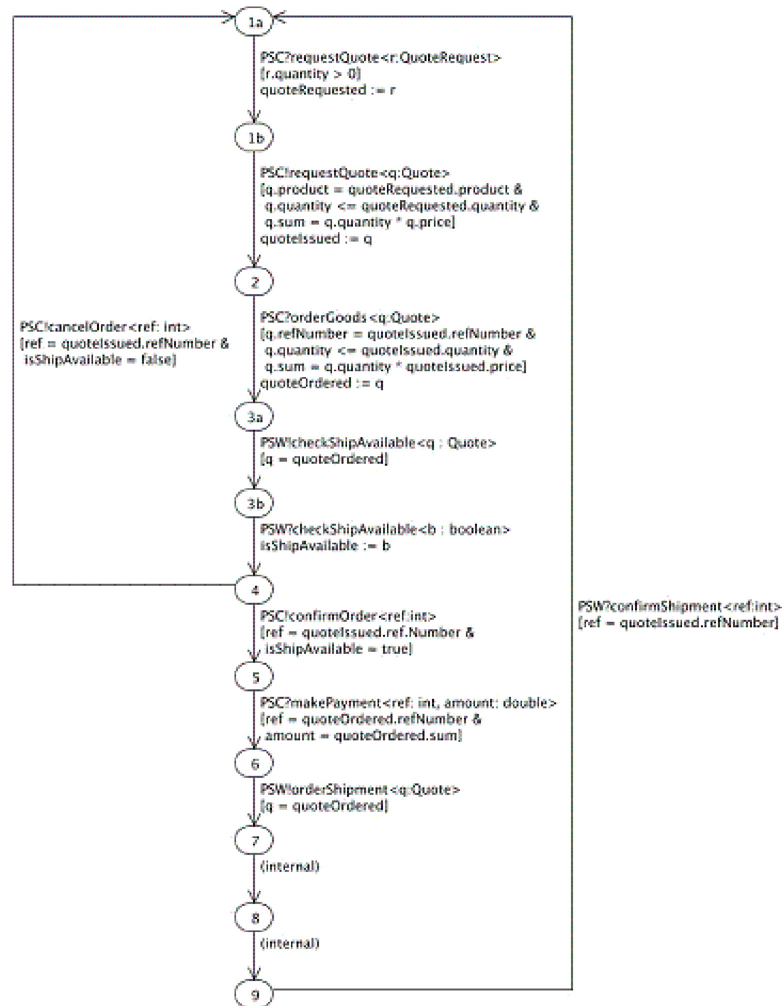


Figure 8: A multiport-SSM specifying the supplier-role

To model the data involved, the SSM-model allows to use advanced transitions dealing with variables and guards. There are global variables which allow the SSM to store values, and message parameters which represent the parameters of the interface messages. Global variables are global to the SSM whereas message parameters are local to the transition where the message occurs. In such a data-enriched Multiport-SSM a transition has five parts:

1. Source state
2. Message Signature — an interface message together with its parameters.
3. Guard — an expression which is evaluated to decide if the transition is eligible to fire. It can deal with message parameters of the message of its transition, and global variables.
4. Variable Update — new values for the global variables can be assigned here.
5. Target state

The shown SSM makes use of the global variables quoteRequested, quoteIssued, quoteOrdered of type Quote, and the boolean variable isShipAvailable. Such global variables are a crucial concept for having a natural and powerful specification model. These concepts sometimes cause confusion when the strict duality between specification models and implementation models is overlooked. Of course, a black box specification must not refer to

the real implementation details like variables which really exist in the implementation. Specification variables like global SSM variables are used to abstractly model the state of the service and have not any kind of semantical relation to real variables from the service.
We exemplify the data concept on the operation requestQuote(r:QuoteRequest):Quote. Since this is a synchronous operation it is modeled as a pair of transitions, one representing the method call going from state 1a to state 1b, and one representing the synchronous return of type Quote going from state 1b to state 2. Looking at the first transition the label mentions first the message signature requestQuote<r:QuoteRequest>. Here we refer with r to the message parameter of type QuoteRequest. Next, the guard [r.quantity>0] constrains the attribute quantity of r to be a positive integer. Remember that message parameters are local to a transition, not global to the whole SSM as global variables are. Hence we have to save the value communicated via r in a global variable quoteRequested. This is done via the assignment quoteRequested:=r. The succeeding transition from state 1b to state 2 corresponds to the returned value of the synchronous requestQuote operation, which is an object of type Quote, referenced by the message parameter q. The guard here ensures that returned quote deals with the same product as mentioned in the requested quote. It further constrains the offered quantity to be less or equal to the requested one, and ensures that the mentioned sum of the quote equals the quantity times price per item. This is done by relating the global variable quoteRequested and the method parameter q. Finally, the offered quote is saved in the global variable quoteIssued. An asynchronous method corresponds to a single transition since here no value is returned, for instance the method orderGoods corresponds do the transition from state 2 to 3a. As usually done, input messages are preceded with an question mark, and output messages are preceded with an exclamation mark.

We will not go into further details here and refer instead to [41] [42]. Semantically SSMs map to Labelled Transition Systems (LTSs). For LTSs well studied testing theories, algorithms and tools exist, see [21]. Within Plastic we are developing a tool which automatically tests WSs based on the introduced SSM models and the algorithm presented in [44].

3.4.1.2 Simulations

Discrete-event simulations (DESs) are commonly used during the design and development of networked systems and services. Traditionally, simulations are used to help understand the behavior and performance of complex systems. Here we are interested in using them to help guide testing [95].

Discrete-event simulations are organized around the abstractions of process and event. Briefly, processes represent the dynamic entities in the system being simulated, while events represent a stimulus applied at a particular time to one of the running processes. When simulating networked services, processes are used to represent the core components of the system, as well as environmental entities such as the network or external services. Events represent messages exchanged by the components and can be thought of as generic structured data types. Virtual time is advanced explicitly by processes to represent "processing time" and advanced implicitly when events are scheduled to occur in the future. To run a simulation, processes are instantiated, initialized, and arranged into a particular configuration that is then executed. A simulation executes until there are no longer events scheduled to occur.

As a brief example, consider a simple client/server application designed to operate in a network environment with unreliable communication. The simulation of this service consists of three process types, Client, Server, and Network, and two event types, Request and Response. The Network process is used as an intermediary through which events between clients and servers are scheduled. Network latency is implemented in the simulation by having the Network process control the scheduling of event deliveries. The unreliable nature of the network is implemented by coding the Network process to probabilistically drop events

by ignoring them. A given configuration might include four process instances: s:Server, c1:Client, c2:Client, and n:Network, communicating using an arbitrary number of Request and Response events.

Clearly, the simulation code of this example can be used to experiment with network latencies and drop rates under different configurations, as a means to predict overall performance, and to evaluate scalability and other properties. But, how can the simulation code be used for testing?

To make the discussion more concrete, we use an example networked service, GCDService, which is a service that computes the greatest common divisor of two integers upon request. GCDclient components construct a request message containing the integers of interest and send it over an unreliable network to a server. A GCDserver component waits for requests and provides responses either by looking up a cached copy of the result from a previous request of the same integers or by computing it directly through the application of Euclid's algorithm.

The natural way to represent a networked service within a DES is to map components of the service to DES processes, and to use events to represent the arrival of messages at the components. This model is used ubiquitously by distributed-system researchers and practitioners. With this basic model, it is relatively straightforward to preserve and represent the fundamental properties of networked services.

- Latency: the behavior of the network can be implemented as a process. Message propagation delays are implemented by altering the scheduled time of message arrival events.
- Separate address spaces: processes in the simulation should not communicate directly (i.e., by invoking methods), but instead use events.
- Partial failure: within a DES environment, processes can stop at any time, but other processes have no direct way of determining this.
- Concurrency: within DES environments, processes execute independently, using simple directives to simulate delays or pauses in processing.

For a developer, the challenge of modeling a networked service is in determining the level of abstraction at which to implement algorithms and behavior. There are no rules about how to proceed, and to a large degree it depends on the intended use of the simulation and experience of the simulation developer. Part of the power of using such a flexible paradigm is that it can easily be tailored to the situation at hand.

For example, with GCDService, the core functionality is provided by Euclid's GCD algorithm, and the distributed functionality related to the caching behavior, sending and receiving messages, is wrapped around it. When modeling this system, the developer must decide whether to include Euclid's algorithm and thus have a fairly complete simulation, or to exclude it to focus on the distributed processing done by the GCDserver component. This decision is influenced by, and impacts the testing process. If the simulation developer knows that Euclid's algorithm is undergoing extensive testing in isolation, they might abstract away this logic in the simulation, and thereby limit the scope of what system-level tests will address. On the other hand, if the developer performing system-level testing is worried that the testing of this module is incomplete at lower levels, they can include the algorithm in the simulation and ensure that its functionality is adequately exercised.

Modeling the network is a key decision that developer must make. There are several possibilities. At one extreme, the developer may model each physical network link that connects components of the service, the routers and switches, and the like. This would be appropriate in situations where the behavior of the service was influenced by individual link properties. At the other extreme, the developer may represent the entire network as a single

process and manipulate delays and drops globally. The DES paradigm is flexible enough to account for either extreme, or virtually anything in between.

To simulate the networked services we consider in this work, we developed a simple simulation environment that encourages a particular programming style and makes certain modeling decisions. Foremost, we represent the network as a single process that has homogeneous drop and delay rates; we do this because our experimental method requires that we can automate the selection of simulation inputs, and more detailed network implementation would make this difficult. We also chose to include the notion of "ports" with communication channels to make the simulation implementation have the flavor of low-level sockets programming. Finally, we adopted a coding style in which message objects (i.e., events) are limited to being structured data types without methods of their own.

3.4.2 QoS Annotations

The openness of the environment characterizing the Service Oriented Architecture (SOA) paradigm naturally led to the pursuit of mechanisms for defining Quality of Service (QoS) level agreement specifications. Nowadays the idea is widely accepted that an effective software design process cannot only focus on functional aspects, ignoring QoS-related properties. For Service Oriented systems, as well as for many other kind of complex enterprise applications [18], communication networks and embedded systems [14] it is certainly no longer possible to propose solutions without adequate consideration of their extra-functional aspects [77].

Nevertheless, traditionally agreements have been not machine-readable. In software engineering only basic notion of agreements have been experimented by means of Interface Description Languages [76][77]. In recent years both industry and academia have shown a great interest on this topic. Concerning the Services Oriented technologies, Service Level Agreements (SLAs) represent one of the most interesting and active issues. SLAs aim at ensuring a consistent cooperation for business-critical services defining contracts between the provider and client of a service and the terms governing their individual and mutual responsibilities with respect to these qualities [100]. Usually a SLA contains a the technical QoS descriptions with the associated metrics. These information are referred as Service Level Specifications (SLSs). In the following a brief description of two languages for service level agreement specification are reported.

WS-Agreement is a specification defined by the Global Grid Forum (GGF) aiming at providing a standard layer to build agreement-driven SOAs [49]. The main assets of the language concern the specification for domain-independent elements of a simple contracting process. Such generic definitions can be extended by domain-specific concepts. In an agreement not every type of content is directly defined. Specifically, the syntax defines statements in order to describe concepts and terms of a generic agreement as top level entities [76]. The use of special construct wraps and integrates the definition related to a specific agreement term.

The top-level structure of a WS-Agreements offer is expressed by means of a XML document which comprises the agreement descriptive information, the context it refers to and the definition of the agreement items (see Figure 9).

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

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

The obligations of parties are organized in two logical parts. The former specifies the involved services by means of the Service Description Terms. Such part primarily describes the functional aspects of a service that will be delivered under an agreement. The latter part of the terms definition defines measurable guarantee associated with the other terms in the agreement and that can be fulfilled or violated. Usually this part refers to defines QoS specifications. The information contained into the fields of a Guarantee Term is expressed by means of domain-specific languages.

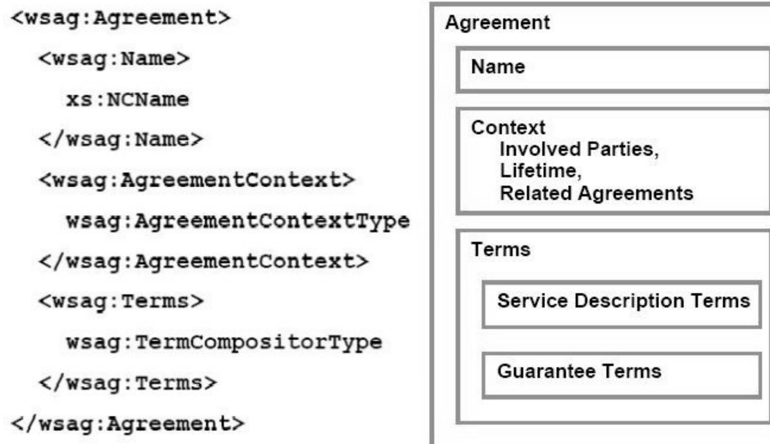


Figure 9: WS Agreement document structure

Another example of language for service level agreement is SLAng [67]. SLAng defines a set of SLAs corresponding to the different kinds of interaction with a service. In particular, the main classification splits the agreements in vertical and horizontal ones. The former subset refers to a service providing infrastructure support for a client, while the latter to the case in which the client subcontracts part of its functionality to a service of the same type [100]. Examples of vertical SLA are the ones between service provider and host or between a host and storage service provider. On the other hand, a horizontal agreement is contracted between a service and an Application Service Provision. The SLAng syntax was formerly defined using XML Schema [67]. However, in [100] the authors propose a UML-based specification of the semantics of the language. Since the OMG had meanwhile specified the UML QoS Profile [86] to represent services and SLAs, the authors have reused such specification by defining a QoS catalogue for SLAng.

For more and detailed information about the adopted language for Service Level Agreement specification, we refer to the deliverable produced by PLASTIC WorkPackage 2 [93].

According to the PLASTIC conceptual model definition [92], SLAng is adopted as the reference SLA language in the project. This means that the specific implementations of the various environments should consider to manage at least QoS annotations expressed in SLAng[2]. Nevertheless, the tools that will be designed to build a QoS validation environments will be proposed as more independent as possible of a specific SLA language.

[2] At the time of writing the QoS elements of the PLASTIC Conceptual Modeling are already identified, but it is not yet defined how the PLASTIC tools will represents/store such information. Thus, if on the one hand it is possible that any of the PLASTIC validation tool could use an *ad-hoc* SLA language, on the other hand it is assumed that the such a language must describe the same concepts on which SLANG predicates with the same

3.5 Summary

This chapter has provided an overview of the context in which the WP4 validation framework will be built, in particular we discussed the objectives, stakeholders and problems that we face for validation in the B3G and service oriented environments. We have also introduced a summary of the overviewed testing process, in which the proposed approaches find their place. We have also provided some background and introductory material useful to the comprehension of the detailed solutions illustrated in the following chapters.

semantics defined by means of PLASTIC Conceptual Model. As consequence, the possible integration of such tools should concern just minor but necessary syntactic translation issue.

4 Off-line Validation Stage

4.1 Introduction

Off-line validation means that the system under evaluation is tested separate from its real working environment, from which it is decoupled. Hence its environment is artificially created, mostly simulated. Simulation plays an important role in off-line testing. We are targeting to validate a distributed system. Therefore traditional testing techniques used for a long-time in industry, developed for component-based, non-distributed architectures have a limited applicability. Traditionally, testing in industry is achieved by developing test suites, which contain test cases, each testing one particular feature of the system under evaluation. The test suites are usually developed by test engineers and written by hand, which makes them costly for large scale or evolving systems. That is why automatic test generation and execution techniques gain more attention in the recent years. In this chapter we are focusing on automatic testing, and will present a few methodologies for achieving this. To automate a validation process, one needs to have input from a higher level of abstraction, which are called models. There can be developed a large variety of models, even for the same system, depending on the system parts or properties in focus. This chapter will expose techniques which use different models, therefore they are fairly independent. Chapter 7 will then deal with integration of these techniques.

Section 4.2 presents the design of a tool called Weevil for automating the process of experimenting with networked services. By experimentation we mean deploying, executing and gathering data on testbeds. To do this in practice, we need, beside the services under experimentation, a testbed where the components of the services will be deployed and executed. Examples of general-purpose testbeds are PlanetLab and EmuLab. The testbed should be configured according to a (quite simple) testbed model. The service under experimentation should also conform to, or wrapped into, a specific conceptual model containing components, start up scripts, and so on. The test engineer then has to map the components to the testbed hosts using a host-mapping model. To stimulate the service (with service calls), so-called actors are configured, which are program segments that mimic the behavior of service users, generating a specific workload. The workload will also be automatically generated from a workload model, specified at design time. Having these in place, control scripts are generated to manage the execution of the experiment.

Section 4.3 presents how to use simulation-based testing in PLASTIC. The basic idea is that the specification, not the implementation, is used as the vehicle for developing test suites. A pool of test cases is generated, for example based on some coverage technique. Then the method selects an adequate test suite from the test pool. To do this, a test data adequacy criterion is selected and, similar to the standard methods, new tests are selected until the criterion has been satisfied. The difference is that we use the specification in the form of a simulation, and not the implementation, to determine the adequacy of a test suite. The applied criteria can be statement, all-branch or definition-use pair coverage. Moreover, we also predict the (relative) effectiveness of candidate test suites using a fault-based (mutation) analysis. The hypothesis is that we can do this sort of analysis (suggesting effective test suites) using a simulation of a service at much lower cost than what would be required to do so on the implementation itself.

In Section 4.4, we propose an approach for model-based testing of WSs. We assume that the system to be evaluated is a setup of Web Services which interact in coordinated scenarios. The static structure of the services and their operations are described in WSDL files. The dynamic, behavioural aspects, that is the possible scenarios of message and control flow are described in another model, called SSM specifications. Such a specification contains the sequence of all possible message invocations in a given state of the system, augmented also with permissible data flow specification. The WSDL and the SSM

specification will be stored in a PLASTIC registry. In the testing/validating phase, a testing engine - AMBITION - is going to take these specifications and then test a given service or a coordinated setup of services "on-the-fly", that is generating test data dynamically, at testing time. Another tool, called JESSI, is meant to help the service developer in creating dynamic SSM specifications using a graphical user interface.

In the last section, an approach for the automatic derivation of test harnesses is presented. These are generated in such a way so as to test/assess that a given system can afford the required level of QoS (e.g., latency, reliability and workload) defined in a corresponding SLA/SLS for a composition of services. The generation of a test harness is done in two phases: first a set of stubs is generated which will simulate the extra-functional behavior of the services in the composition, then the second phase foresees the composition of services. Regarding the SLA/SLS specification, the method is general, but in line with what proposed in PLASTIC WP2, here SLAng will be considered as a reference language for SLA/SLS descriptions. The generation of stubs will be done also as a two-way process: first, an empty stub set is generated from WSDL-s, using common techniques, then these implementations are filled with some behavior that will fulfill the required extra-functional properties for the service corresponding to the stub. As an example, a latency declaration is modeled by introducing a delay in the generated code.

4.2 Experiments on Networked Services

Engineering a highly distributed system such as a B3G networked service is a challenging activity. The difficulties are due in part to the intrinsic complexity of the services and the protocols used to integrate them, and in part to the practical obstacles that one faces in evaluating and tuning alternative designs and implementations. Difficulties of the first kind arise early on in the development process, where analytical methods and simulations can offer valuable guidance to the engineer. By contrast, the latter kind of difficulties are typical of the later stages of development, where only systematic, repeated experimentation with executable prototypes in realistic execution environments can yield accurate results.

Experimentation is an essential tool employed by the developers of all kinds of software systems. It allows them to gain an understanding of various dynamic system properties and to help tune their systems prior to deployment. Testing is, indeed, a form of experimentation. Experimentation with networked services is particularly important and particularly challenging: important in that networked services can exhibit a far greater range of behaviors than simple host-based systems; and challenging in that it can be difficult, costly, and time consuming to reproduce in the laboratory all possible configurations of a networked service, under all possible usage scenarios, and under all possible environmental conditions or operational contexts.

To get valid results, the experimental environment must mimic a networked service's real execution environment, and must be able to gather data from the running service without introducing unreasonable run-time overhead. The large number of parameters that influence a service's behavior often requires many experiments to be configured and repeated easily. Since experiments are conducted on services that are being actively developed, and usage scenarios often change rapidly and unpredictably, the experimentation framework must support efficient refactoring of experiment configurations. Moreover, the scale, heterogeneity, and dynamism of networked services make it difficult to conduct these experiments manually. Thus, assuming an iterative design/evaluation process, our goal is to provide software engineers with tools and methods that allow them to quickly evaluate their design and implementation decisions through automated, repeatable experiments.

At a minimum, experimentation with a networked service consists of deploying and executing its components on some sort of testbed. But in many cases, the size and degree of distribution of that testbed are crucial to the validity of the experimental results. As mentioned

in Section 2.1.1, recent efforts have led to the development of large, general-purpose testbeds such as PlanetLab [89], which is a collection of nearly 700 hosts located at over 330 sites around the world that communicate over the live Internet[3] and Emulab [111], which is a large, local-area cluster of machines offering a configurable and controllable network layer.[4] PlanetLab provides a generic platform where software systems can be deployed and executed in a realistic multi-host environment. It does not, however, provide support for large-scale experimentation activities. Emulab provides several primitive but useful environmental controls, including facilities for traffic generation and shaping, for constructing and modifying a network topology, and for remotely rebooting network hosts. When an engineer wishes to run an experiment using a particular simulated operational context, they configure the routers to act as a network having a given topology, drop rate, congestion, and delay, determine the times at which hosts fail and recover, load the service's components onto the hosts, and then start execution of the components. On the other hand, Emulab suffers from the same shortcoming as PlanetLab: it does not provide the automation features needed to make large-scale experimentation tractable and cost effective.

Experimentation is much more than simple deployment and execution. It involves complex activities such as experimental design, workload generation, data collection, data analysis, and overall experiment management. What is needed is a comprehensive framework to help software engineers manage and automate experiments with networked services performed on distributed testbeds.

Our approach to such a framework is depicted in Figure 10. At a high level, the process for conducting an experiment involves preparation, execution, and analysis. We make use of *model-driven generative techniques* to create scripts that configure, deploy, and manage an individual run within an experiment's suite of runs. We also provide a technique that we refer to as *simulation-based workload generation* for creating the application workloads that embody client usage scenarios.

Going a bit deeper, the engineer first populates several models that capture essential information about a planned experiment run or "trial" (following the terminology of Fenton and Pfleeger [39]). There are separate models for the client activity, the subject service, the testbed, the mapping of the subject service's components to the testbed hosts, and the mapping of the clients to the testbed hosts. The first model is used to drive the simulations that generate the application workloads, while the other models are used to generate control scripts for managing the trial.

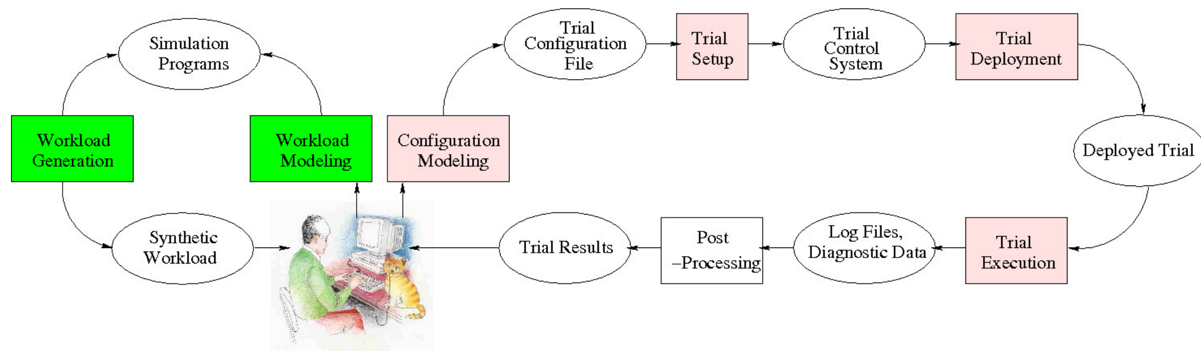


Figure 10: Experiment trial automation

[3] http://www.planet-lab.org/

[4] http://www.emulab.net/

After the workload and scripts have been generated, the engineer begins the trial by executing, on a master host, a master control script that in turn deploys, executes, and coordinates several other slave control scripts. The master script is able to automatically redeploy and re-execute the slave scripts when a testbed host fails during the trial. Each slave script performs three tasks. First it configures and deploys some portion of the subject service's components. It then initiates execution of those components. Finally, after the trial has concluded, it sends logged output and other diagnostic data back to the master host, and performs any necessary cleanup activities.

During the trial, the subject service is stimulated by the execution of service calls at the times and locations dictated by the application workload. The input to the workload generator is an operational definition of a set of *actors* representing clients. For example, if the subject of the experiment is a Web caching service, then an actor would represent a person browsing the Web or a robotic Web crawler. Actors are instantiated as processes within a discrete-event simulation and those processes are then executed in simulation to reveal the actor behaviors. The output of the simulation is a simple, time-ordered trace of actions performed by the actors. The trace is then processed into one or more workloads. The actions in the workload are applied to the system during the trial by *actor programs*, which are themselves deployed and executed by the slave control scripts.

4.2.1 Workload Generation

A common practice in software experimentation is to generate a workload on the basis of a statistical model that abstracts usage patterns of the subject system. This approach is concise and efficient. However, usage patterns often vary widely based on context, and a statistical model offers only limited expressiveness. Also, sufficient data must be available to create an accurate statistical model of an existing behavior.

As a more general, complementary approach, we propose an operational technique that models usage behavior directly. Specifications for an operational model could come from, for example, empirical traces [101], detailed user behavior profiles [80], or an experimentation plan in which specific usage scenarios are described. Our idea is to give software engineers the ability to quickly and easily express their specific system usage scenarios or user behaviors to create a diversity of workloads.

The workload generation process is illustrated in Figure 11. It allows the engineer to model one or more types of actor behaviors as programs written in a common programming language, such as Java or C++, supported by a workload-generation library. The actors may therefore execute arbitrary functions and maintain arbitrary state. After programming the actor behavior types, the engineer populates a scenario consisting of actor instances specified in the actor configuration. The actor behavior types and the actor configuration make up a workload scenario definition. A workload scenario is then translated into an executable simulation program that is linked with the workload-generation library and executed to produce the desired workload.

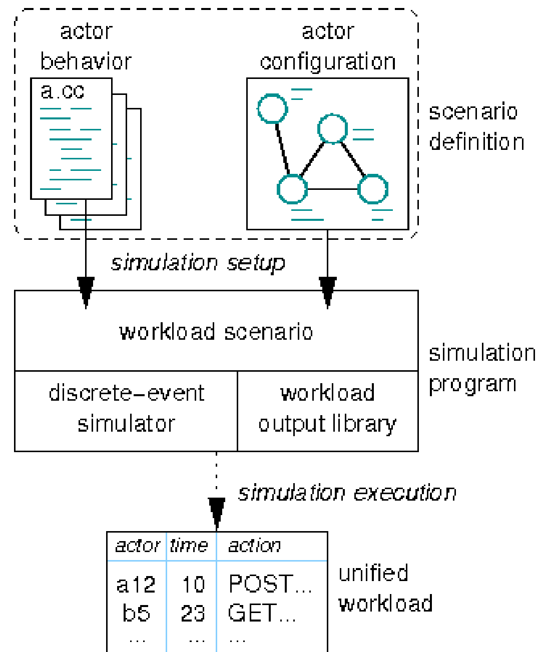


Figure 11: Simulation-based workload generation

The workload consists of all interactions between actors and the subject service, which are recorded by a special output function of the workload-generation library. These interactions represent service calls that are applied to the subject during the actual experiment execution. Thus, we are using a discrete-event simulator to simulate actor behaviors, and capture the service calls made to the subject as a reusable and reconfigurable workload that can be applied in multiple experimental scenarios.

Our motivation for developing the simulation-based workload generator is its inherent flexibility and scalability. It is flexible in two dimensions. First, it can be immediately used to program workload generators based on statistical models. In fact, those generators reduce to scenarios with independent stochastic processes. Second, because it is fully programmable, it offers a natural way to represent complicated actor behaviors at any level of abstraction. It allows for an easy and compact specification of interdependent dynamic client behaviors that may result in complex and interesting workloads for collaborative activities.

Simulation-based workload generation is scalable in the sense that it can seamlessly deal with very complex scenarios, consisting of a multitude of interacting actors, executing over long periods of (virtual) time. In fact, this is precisely what simulation engines are designed to do. This ability to scale up is particularly beneficial because it allows an engineer to produce workloads in which complex collective behaviors emerge from simple individual behaviors.

4.2.2 Deployment and Execution

A more detailed view of the experiment automation process is depicted in Figure 12. 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 generated control scripts and data files. The cross-hatched ovals represent data generated by the subject service during an experiment. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts.

We take an automated, model-based approach, whereby generative techniques are used to transform experiment configuration directives into an experiment management framework. This provides three main advantages over manual approaches: (1) engineers are relieved of the burden of creating and maintaining a large volume of experiment control scripts, and instead must only deal directly with a relatively concise set of configuration parameters; (2) models can be shared among experiments and easily tweaked when experiments must be changed; and (3) the generative capabilities transparently handle much of the complexity brought about by a service's or a testbed's scale and heterogeneity.

4.2.2.1 Configuration Modeling

Each trial requires a workload together with experiment configurations for two primary models: the subject service under experimentation (SUE) and the testbed. Additionally, mappings between the SUE and the testbed, and between actors represented in the workload and the SUE, must be provided. These configuration models are represented by the dark ovals along the top of Figure 12. They can be programmed in a macro language such as GNU m4 [1] by calling declaration macros to instantiate the model elements. In other words, the declaration macros will define a set of property macros serving as properties of an experiment. Other than the reserved properties described here, we can allow service-specific properties to be assigned. All these property macros can be used as parameters in other declaration macros. They are resolved during script generation using macro expansion. The engineer is supported during this activity by performing extensive checks on the syntax and consistency of the configurations, and by providing detailed error messages about any problems encountered.

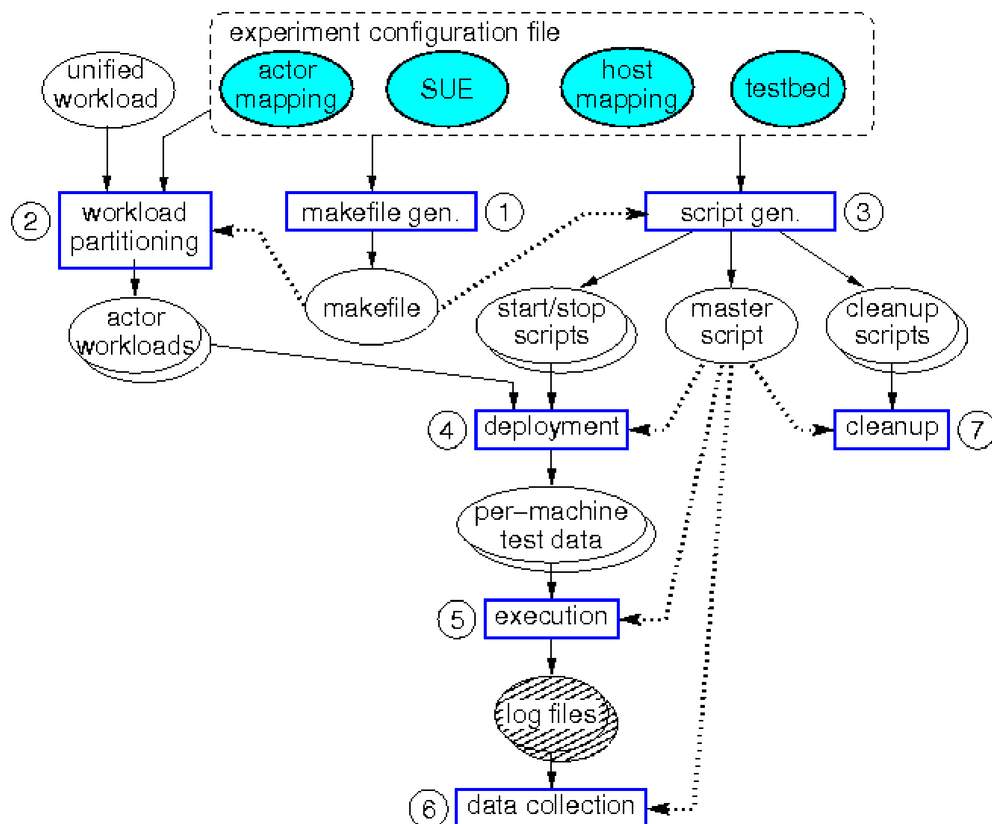


Figure 12: Detailed view of trial automation process

4.2.2.1.1 SUE Model

The conceptual model of a SUE is shown in Figure 13. As this figure shows, a SUE is comprised of typed *Component*s, *Relation*s between them, and an *Order* in which to start up the components. In general, a SUE can consist of different types of components. Each component is declared as an instance of a *ComponentType*. *ComponentType* has *startScript* and *stopScript* attributes, and optionally a *config* attribute that contains the contents of a configuration file. This design allows common attributes to be shared among all components of a type. For different experiments targeting the same networked service, an engineer would typically need to make minor changes to other entities without having to modify *ComponentType* attributes. The attributes *processing* and *output* are used to specify post-processing of experiment output: a *processing* script is first executed on the component log files, and then the *output* of the script is copied back from each component's workspace to the master.

The *Relation*s contained in a SUE model are used to represent any binary associations between components. They are optional and entirely service specific. In general, relations are used in situations where one component references properties of another component for its execution.

Order entities are used to represent the necessary or preferred order in which to start the components. This is optional and entirely service specific, since some SUEs require certain components to be ready before others.

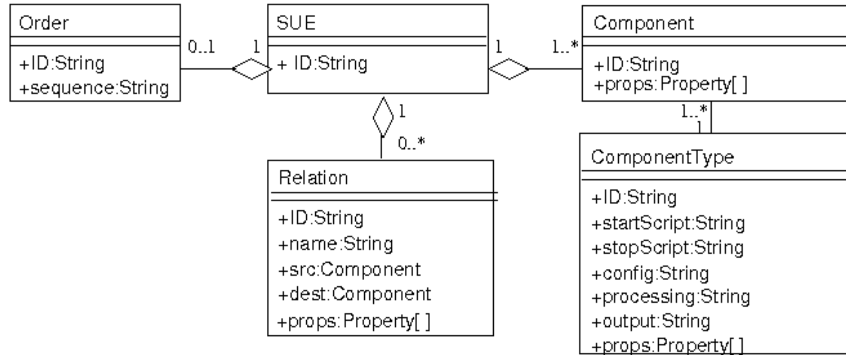


Figure 13: SUE conceptual model

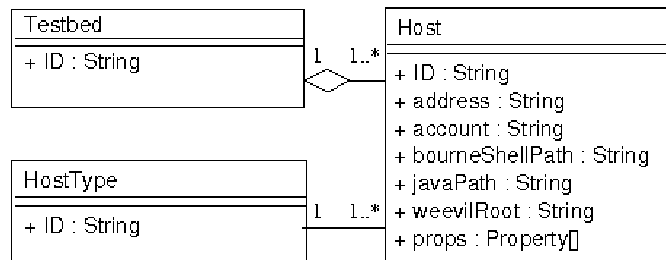


Figure 14:Testbed conceptual model

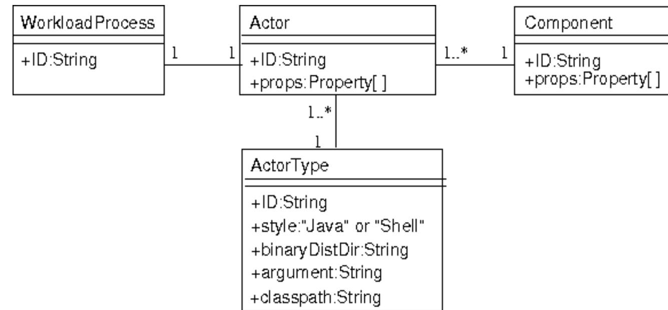


Figure 15: Actor-mapping conceptual model

4.2.2.1.2 Testbed Model

We make minimal assumptions about the nature of the testbed itself. As shown in Figure 14, a *Testbed* has an identifier and a collection of *Host*s. Each host in a testbed is an *account* on a network *address*. In PlanetLab the account is actually the PlanetLab slice name to which the engineer is assigned. Each *Host* is assumed to be capable of running some form of Unix commands (even if the operating system is Windows) and so has the attribute *bourneShellPath* to provide its local path to the program *sh*. The *javaPath* attribute is needed if an actor is implemented in Java, as described below. *weevilRoot* specifies the workspace on the host assigned for the experiment.

To support deployment on heterogeneous testbeds, the *HostType* entity is used to partition the hosts into categories needed for each software binary package.

4.2.2.1.3 Mappings

The two models described above are designed to be largely independent of each other and of the workload to be used during an experiment. This gives the engineer a fair amount of flexibility in composing experiments. The connection among the three is specified using two mappings.

The first mapping associates the SUE and the testbed by simply specifying on which host of the testbed each component of the SUE should reside. The second mapping, shown in Figure 15, associates the workload and the SUE. Each workload process is mapped to a single component through an *Actor* that is declared as an instance of an *ActorType*. *ActorType* represents the implementation of the actors using the same service APIs. An actor is a service-specific program that understands how to stimulate the SUE as dictated by the workload.

4.2.2.2 Setup and Script Generation

Given a set of configuration models, the engineer can now “compile” the experiment configurations into the framework scripts that will be used for experiment deployment and execution. First, the configurations are checked for consistency, and a per-experiment-trial file is generated to control the rest of the process (shown as action 1 in Figure 12). Next, the overall workload is tailored according to the experiment configuration (action 2). The partitioning of the overall workload into per-actor workloads is also performed as part of this action. Finally, a start script, a stop script, and a cleanup script are generated for each component in the SUE, and a master control script is generated to manage the execution of the experiment (action 3).

4.2.2.3 Deployment and Execution

At this point, the engineer can perform an experiment by simply executing the master control script. The master control script deploys the components of the SUE, per-actor workloads, actors, and control scripts to the hosts (action 4), then starts all the components and actors (action 5). By estimating the round-trip time between the master host and testbed hosts, the master script intelligently decides when to have actors begin processing their workloads. The master script waits for all actors to complete processing their workloads and then causes execution of the stop scripts for each of the components. After all the components terminate, post-processing scripts are executed and output is copied back to the master machine (action 6). Finally, the testbed machines are cleaned up as necessary (action 7).

4.3 Simulation-based Testing

Figure 16 depicts a simple and generic testing process. As a first step, the tester selects a particular adequacy criterion to organize the rest of the process. The tester must select a test suite that will satisfy the criterion. A test suite is composed of test cases, each one consisting of an input vector that includes direct inputs to the system, representing functional parameters, as well as inputs to the environment, representing environmental conditions. In the figure, actions 2-4 are repeated until the criterion is satisfied. Once a suite has been selected, the tester uses it to test the implementation by first mapping input vectors into the implementation domain (step 5), and then executing each test case on the implementation (step 6). The system is ready for release once it passes all test cases.

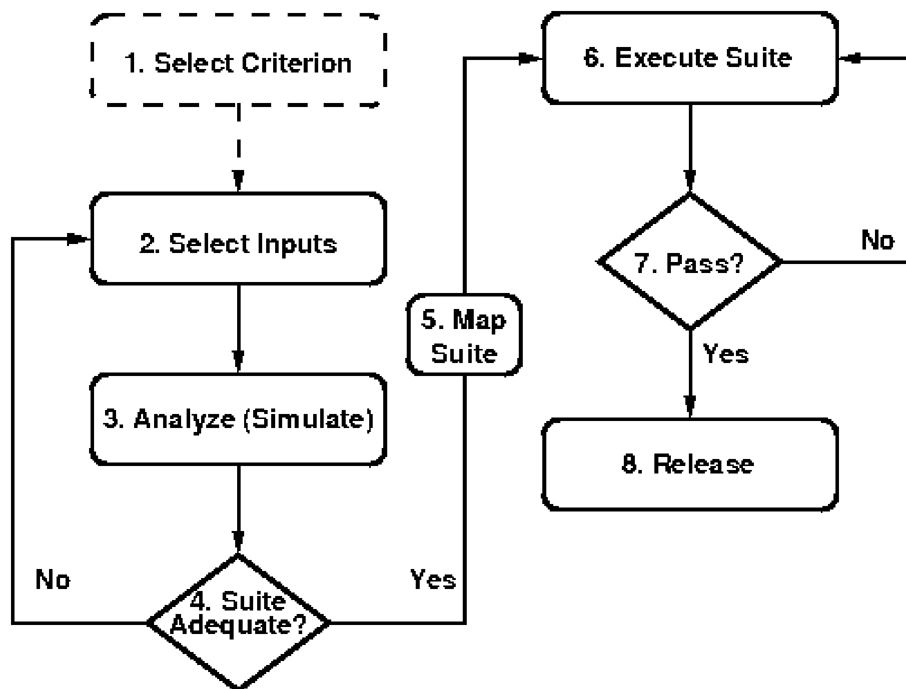


Figure 16: A generic testing process

In traditional testing techniques for simple host-based services, a functional specification of the service is used to select inputs that eventually lead to a “specification adequate” test suite. This is known in the field as *specification-based testing*. In the context of B3G services, whose correctness is driven as much or more so by extra-functional considerations as functional ones, such a traditional specification is too narrow to provide a basis for determining the adequacy of test suites. We thus turn to a richer specification technique that

can account for both kinds of consideration: the discrete event simulation (Section 3.4.1.2), which leads to a new technique that we call *simulation-based testing.*

At a high level, our approach rests on two ideas. The first idea is to use the simulation itself (i.e., the actual program code written to describe the behaviour of processes and events in the simulation) and simulation executions as a basis to formulate general-purpose and/or service-specific test adequacy criteria. Referring to Figure 16, the simulation is used during action 3, analysis. Thus, given a candidate test case, a corresponding simulation is configured and executed to gather data about its coverage *of the simulation*, much as the coverage of the functional specification is used in traditional (specification-based) testing techniques. These data are then aggregated for the test suite to determine adequacy. For example, a general-purpose criterion might call for statement coverage of the simulation code of all non-environmental processes, or a service-specific criterion might require that each event type be dropped at least once during a simulation run. Once a criterion is defined, the developer can evaluate the adequacy of a test suite by running the test cases in a suitably instrumented simulation.

Note that the process by which individual test cases are created or generated is outside the scope of this technique. Similarly, we neither propose nor discuss any specific strategy by which the developer might search the space of test suites to find an adequate one; our concern is with the decision process, not the search process.

Step 1 in Figure 16 requires that the developer chooses a single adequacy criterion. However, the developer may not have enough information to make this decision with confidence. This might be because no good criteria are available or known, due to the lack of experience with a particular service, or because the developer cannot decide which criteria to adopt out of a set of plausible candidates. Even when a criterion has been selected, the developer might be concerned that they have selected an effective test suite.

Therefore, the second idea is to provide the developer with a general ranking mechanism to: (1) fine tune the selection of the most effective suite within the set of adequate suites, given a chosen criterion and (2) guide the selection of the most effective criterion for the service at hand. This ranking mechanism is also based on the simulation code, and in particular it is derived from a fault-based analysis of the simulation code.

As with all specification-based testing techniques, we analyze a specification to select test cases for the implementation. Analysis, in our approach, involves setting up a simulation configuration and executing it. In the following discussion, we distinguish four parts of the analysis process:

(1) modeling: the behavior of a service is coded within a discrete-event simulation (DES) environment;

(2) configuration: parameterized configurations of the simulated service are created;

(3) simulation execution: parameter values (i.e., test cases) are picked, the configuration is executed, and coverage data are collected; and

(4) aggregation: coverage data are post-processed and aggregated to determine the adequacy of a collection of executions (i.e., a test suite).

Each set of parameter values derived from this analysis constitutes the input vector of a test case that is later applied to the implementation during test execution by a suitable test harness.

It is important to understand that the analysis process we outline is a conceptual framework, not a specific tool set. The concepts central to simulation-based testing can be applied within many different environments. In fact, part of the power of this approach is the manner in which it leverages existing tools and techniques. The only constraint we place on the DES environment is that it is programmed in a language that is amenable to coverage analysis

and mutation. If a common general-purpose programming language is used for simulation, then there are a wide variety of coverage analysis tools that can be used off-the-shelf. There is less selection of mutation tools, but the major imperative languages are supported. Furthermore, the process we outline above makes no assumptions and imposes no restrictions on the implementation style of the service.

Having discussed the first part of the analysis process in Section 3.4.1.2, we now discuss the other three parts.

4.3.1 Simulation Environment

In order to develop the simulation-based testing technique, we choose a specific DES environment, specific coverage analyses, and a specific code mutation tool. We describe the technique in this specific context.

To simulate services, we developed a simple simulation environment that encourages a particular programming style and makes certain modeling decisions. Foremost, we represent the network as a single process that has homogeneous drop and delay rates; we do this because our experimental method requires that we can automate the selection of simulation inputs, and more detailed network implementation would make this difficult. We also chose to include the notion of "ports" with communication channels to make the simulation implementation have the flavor of low-level sockets programming. Finally, we adopted a coding style in which message objects (i.e., events) are limited to being structured data types without methods of their own.

We use the Java-based DES library simjava (http://www.dcs.ed.ac.uk/home/hase/simjava/). From this library, we use classes representing processes and events, virtual time management, and the event scheduling framework. We adapt these domain-independent mechanisms to provide a simulation environment appropriate for simulating networked services that exchange messages over an unreliable and latent network. The primary function of our simulation environment is to provide an intuitive interface for representing communication between processes. This is enabled by implementing a pluggable framework for coding network behavior. With this in place, the simulation developer interacts with the network in an intuitive way at appropriate places within the component logic.

4.3.1.1 Process Model

There are two different paradigms for implementing a DES system: event-based and process-based, the difference primarily being how the behavior of a process is implemented. In an event-based simulation, process behavior is implemented within callback functions that are invoked when an event occurs for the process; this accentuates the reactive nature of the processes. Conversely, in a process-based environment, the behavior is implemented within a main routine, and event occurrences must be polled explicitly. We use the process-based approach, primarily because this results in a more natural implementation of each process' behavior.

4.3.1.2 Communication

Many different approaches to representing the network are possible, depending on the focus of the simulation. We chose to represent the entire network as a single process that acts like a switchboard between the components. This makes it quite simple to implement a global behavior for the network. The simulation environment provides implementations of different communication paradigms: datagram based, and stream based. During the modeling stage the engineer need only decide which paradigm to operate within for each interaction, similar to the decision to use either the UDP or TCP transport mechanisms. In this work we consider only services whose communications can be modeled with datagrams. The behavior of the network can be specified when a simulation is configured, and we discuss this below.

Components interact with the network through channels. DatagramChannels, reminiscent of UDP sockets, are bound to a port of the local component either explicitly or implicitly. Typically, port numbers are explicitly provided for channels that will act as server ports; if a port number is not provided, an unused one is selected. Messages arriving at the port are received through a channel object, and the same object may be used to send messages to other channels. Incoming messages are queued and retrieved in the order in which they were received. If no messages are available, the receive operation blocks (with an optional timeout). The send operation does not block.

A component can instantiate as many channels as necessary to implement the behavior correctly and a select mechanism is provided so that a component can wait for incoming messages on multiple channels simultaneously.

4.3.1.3 Messages

Messages are implemented as immutable structures. Aside from application-specific data fields, message objects also contain the source and destination addresses after being sent across the network.

In our experience, it is convenient to represent messages at a high-level using arrays and collections and other advanced data structures as necessary. However, this is a modeling decision that results in message parsing and formatting logic being left out of the simulation and therefore unable to be addressed by simulation-based criteria. The simulation developer has complete control over the level of abstraction of the simulation code, and our environment does not preclude, for example, the implementation of all message content as an unformatted byte array.

4.3.1.4 Components

The core behavior of the system is coded in the components. Components interact with each other by sending message objects through the communication channels discussed earlier. Within the component, virtually any data and control structures can be used.

4.3.2 Configuration

Before a simulation can be run, components must be instantiated and organized into a particular configuration that represents the real-world situation being simulated. In our environment, this is accomplished by extending the Simulation class and adding component instances. In our discussion of configurations, we differentiate between abstract configurations that are parameterized, and therefore unable to be directly executed, and concrete configurations that supply values for all parameters. In testing terminology, abstract configurations are test harnesses, and the values included as part of a concrete configuration comprise the input vector of a test case.

4.3.3 Simulation Execution

The actual means of execution is specific to the DES framework being used, but all simulations must somehow be executed. In ours the name of a concrete configuration class is passed to the simulation engine. The simulator loads the class using a custom class loader that first checks the bytecode to ensure conformance to the implementation limitations mentioned above, and then instruments the classes so that block, branch, and definition-use coverage values can be determined.

The simulation classloader uses the Bytecode Engineering Library (http://jakarta.apache.org/bcel/) to analyze the bytecode. Using this library, instrumentation for block and branch coverage is essentially a matter of inserting tracing method calls before each branch instruction, and before each instruction that is the target of a branch.

Complete data-flow analysis of Java code is quite difficult to accomplish in the general case. In our implementation, we take advantage of some of the constraints imposed on the component coding style by our framework to make this task easier. Therefore, to instrument for definition-use coverage, tracing method calls are inserted before each store instruction and each load instruction. Method calls are conservatively treated as object definitions. To conservatively determine the definition-use pairs, we developed some supporting classes to execute the same data-flow analysis performed during bytecode verification by the Java compiler [72]. The coverage and performance information collected during a simulation execution is stored in a compressed XML format.

4.3.4 Aggregation

Each simulation run generates coverage data for a single test case. To determine adequacy of a test suite, the coverage data must be aggregated properly. To do this with our simulation environment, we created a reporting tool that parses the simulation trace files created during execution and records the coverage of each block, branch, and definition-use pair. By separating the execution and reporting functions in this way, each simulation is executed once even if the adequacy of different combinations of simulations is computed multiple times.

The reporting engine provides output in both HTML and XML. The XML output is used during experimentation when there is no need for graphical examination of the results. During development of the simulations and especially during the development of configurations, the HTML output enabled us to graphically examine the white-box coverages.

4.3.5 Fault-Based Analyses

In fault-based analysis, testing strategies such as adequacy criteria are compared by their ability to detect fault classes. Fault classes are typically manifested as mutation operators that alter a correct formal artifact in well-defined ways to produce a set of incorrect versions. These mutants can be used to compare testing strategies.

For example, an implementation might have a fault that causes a particular state change to be missed, where such state changes are represented as transitions in a finite-state specification. This missing transition fault class is then represented in the specification domain by all specifications that can be obtained from the original specification by removing one of the transitions. Testing strategies that are able to distinguish incorrect from correct specifications are said to cover that particular fault class. The underlying assumption of fault-based analysis is that simple syntactic faults in a specification are representative of a wide range of implementation faults that might arise in practice, so a testing strategy that covers a particular fault class is expected to do well at finding this class of faults in an implementation.

A prerequisite of a fault-based analysis is the existence of a set of mutation operators. Simulations are coded in imperative programming languages and so well suited to the code-mutation operators developed in the context of mutation testing [29]. These operators make simple syntactic changes to code that may result in semantic differences.

In our fault-based analysis, we apply standard code-mutation operators to the component code. Each concrete simulation is then executed against each mutant component in turn. A simulation may (1) terminate normally with reasonable results, (2) terminate normally with unreasonable results, (3) not terminate, or (4) terminate abnormally. For all but the first situation, the test case (configuration) is recorded as having killed the mutant. The mutant score of a test suite is computed as the percentage of mutants killed by at least one test case in the suite.

In most mutation analyses, the exact output from the original version is used as an oracle against which mutant output is compared. This is not always possible in this case because simulations of distributed systems are naturally non-deterministic. In practice, we use

assertions and sanity checks in the simulation code to determine which results are considered "reasonable".

While fault-based analysis is not a new technique, it is typically not used in practice because of the computational cost associated with obtaining mutants and executing test cases against each one. However, in our case, two fundamental features of simulation-based testing enable the use of fault-based analysis. First, when compared to the execution cost of deploying and executing the real (i.e. non-simulation) implementation of a distributed system, simulation execution is efficient. Second, because of the use of abstraction, simulation code is significantly smaller and simpler than the code needed to implement the actual system. In combination, these two features result in a fault-based analysis technique that is actually feasible in practice.

4.3.6 Usage Scenarios

We use simulation-based adequacy criteria and fault-based analysis of the simulation code, individually or in combination, to support the identification of effective test suites. We describe this approach through three different usage scenarios.

4.3.6.1 Conventional

The conventional way to use simulation-based testing is to choose an adequacy criterion defined against the simulation code, and select a single test suite that is adequate with respect to it. In this scenario, the developer has a number of options for which adequacy criteria to use.

The developer might choose a criterion that depends on observable events from the environment. For example, the criterion could require that communication links be used up to their maximum capacity. Or, the developer might choose a classical white-box code coverage adequacy criterion defined over the simulation. However, in both cases, the simulation must be suitably instrumented and then executed for the adequacy to be determined.

In this scenario, the tester is exposed to both intra-criterion and inter-criteria risk. The cost in this scenario is simply the cost of simulating a number of test cases until an adequacy value is achieved.

4.3.6.2 Boosting

In this scenario, the tester has somehow chosen a particular adequacy criterion, as before, but here they want to reduce the risk of picking an ineffective, adequate test suite. Thus, they select multiple adequate test suites, use fault-based analysis to rank the suites by mutant score, and apply the highest-ranked to the implementation.

This usage scenario obviously applies to the code or environmental criteria described in the previous scenario. In this case, the simulation may be used in a first stage to identify a number of adequate suites, and in a second stage to rank the individual suites through fault-based analysis.

But the boosting technique can also be applied to a criterion that depends exclusively on the input vector. In this case, the simulation code is of no use in evaluating adequacy, and the developer must do that through other means. However, fault-based analysis of the simulation can still contribute in this scenario by providing a relative ranking of adequate suites based on mutant score.

This usage is more costly than the conventional usage, since multiple adequate suites must be selected, and each selected test case must undergo a fault-based analysis, but intra-component risk is reduced.

4.3.6.3 Ranking

The developer wants to choose a criterion among many applicable criteria for the particular system under test. Once again, the developer can turn to fault-based analysis of the simulation code to rank the different criteria. Specifically, the developer creates adequate suites for each criterion and then selects the suite with the highest mutant score. At this point, the tester simply applies the boosting usage to the adequate test suites already created for the highest-ranking criterion. This scenario is even more costly than the boosting, because many adequate suites are both selected (using the simulation or not), and then executed against simulation mutants.

In summary, through these scenarios, a DES can be used directly to evaluate the adequacy of a test suite with respect to criteria based on the environment and on the simulation code. This requires running the test suite through an instrumented simulation. In addition, the simulation can be used to improve the effectiveness of any criterion and to guide the programmer in selecting a criterion. This is done by means of a fault-based analysis of the simulation code.

4.4 Model-based testing

A WSDL file describes the static aspects of Web Services. We have visualized these static aspects before (see Section 3.4.1.1) via a UML 2.0 Component Diagram. To do model-based testing we assume to have a Multiport-SSM, which extends the static aspects of the WSDL with the dynamic, functional aspect of data enriched conversation descriptions. Such a conversation consists of messages (including their parameter data) as they are exchanged at WS ports. The goal is to automatically test WS by generating inputs and giving verdicts to observed outputs based on the SSM specification. The general approach is depicted in the following figure.

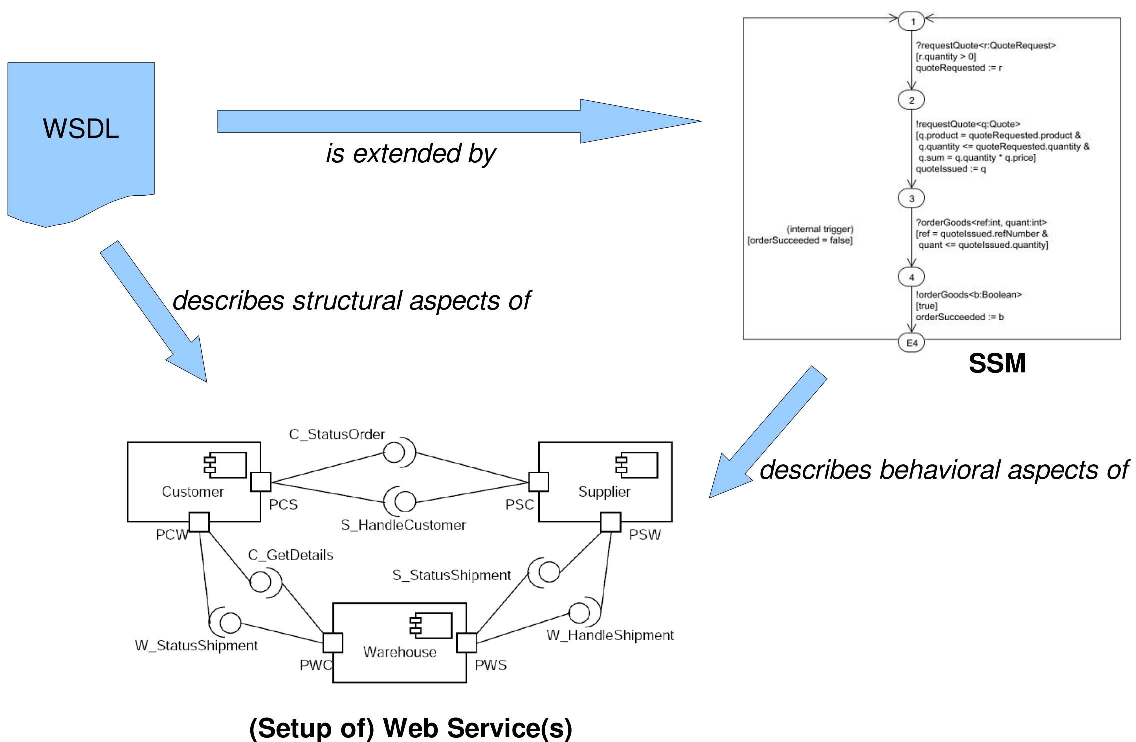


Figure 17: The general MBT approach

In this section the central tools to conduct Model-Based Testing (MBT) within PLASTIC will be described. They will also partially be used by the Audition framework. We focus here on the tools themselves, not on their role within Audition, which will be further explained in Chapter 5.

Under development are the following tools:

1. **AMBITION** (Automatic Model-Based Interface Testing In Open Networks)
   A tool which automatically tests Web Services based on an SSM specification (see 3.4.1.1).
2. **JESSI** (Java Editor and Simulator for Service Interfaces)
   An editor for SSMs (will interface with the PLASTIC editor). Based on a WSDL file dynamic aspects of Web Services can be modeled and simulated.

Before we describe the tools in greater detail, we give a simple taxonomy of WS scenarios. Such a taxonomy allows to characterize the different stages of maturity, and potential application domains, of every theory and tool dealing with WS.

4.4.1 A Taxonomy of Web Service Scenarios

We have motivated in Section 3.4.1.1 that SSM models can comprise a setup of several communicating Web Services (WS), with each of them possibly having several ports. From a testing point of view some principal scenarios can be distinguished which have specific demands on the testing infrastructure and underlying testing theory. We summarize these scenarios next. To do so we refer again to the procurement example as being introduced in 3.4.1.1. We assume that we want to test (aspects of) the Supplier WS.

- Testing a single, passive Web Service:

Figure 18 : Testing a single, passive WS

This is the simplest case, where the Service Under Test (SUT), i.e. the Supplier WS, is tested via a single, passive port. By passive we mean that the Supplier does not actively send messages. This corresponds to a WSDL file which defines only one-way and request-response operations for the PSC port. The tester in this case plays the role of the Customer WS. It does not even have to be a WS itself since every application can invoke operations on WS. Also the underlying theory, which originates from testing reactive systems, can be easily applied to this case.

- Testing a single, active Web Service:

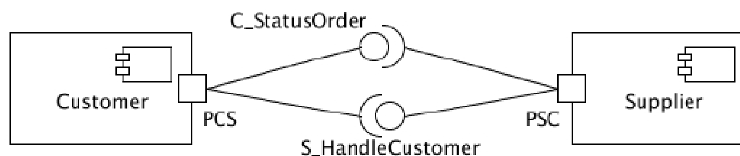


Figure 19 : Testing a single, active WS

This case is more complex since the Supplier WS is allowed to actively send messages to the Customer WS (played by the tester). Hence here we can have also notification and solicit-response operations in the PSC port type. The Customer in this case has to be also a WS since the Supplier needs to contact its PCS port. In a dynamic setting, where several Customer WS can connect to the Supplier WS at runtime, it has to be clarified how this dynamic binding is accomplished. Several proposed solutions exist, but it has to be discovered within PLASTIC if a, and which, solution is aspired. In any case it would be desirable if the dynamic binding is handled by the PLASTIC middleware, not by the services themselves. The underlying testing theory can be straightforwardly adapted to this case since an active WS is very similar to a reactive system. Here we can exploit the full richness of the theory, like quiescence and unstable states (see section 4.4.2.1 for further details).

- Testing a coordinated setup:

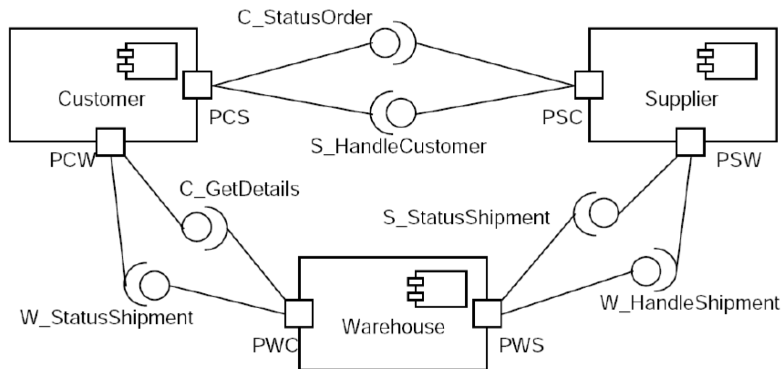


Figure 20 : Testing a coordinated setup

This is the setup already introduced in Section 3.4.1.1. Testing the Supplier WS here is less obvious than in the preceding cases since the Supplier is embedded in a coordinated setup. One can think of two principal scenarios:

Scenario 1 – Let the tester also play the role of the Warehouse:

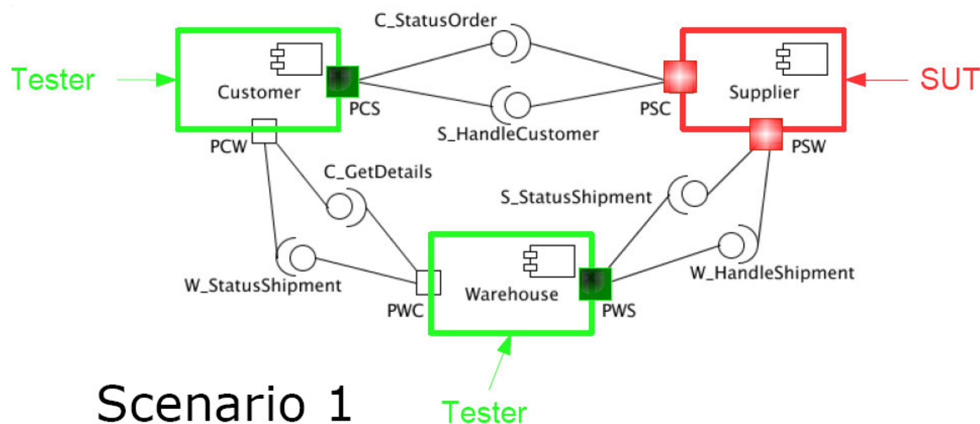


Figure 21: The tester plays the role of the Warehouse

Here we test both ports of the Supplier WS. To do so we need an SSM specification comprising both ports (see e.g. the one given in Section 3.4.1.1). The tester itself can be a centralized or distributed one. In the first case a single tester tests both ports of the supplier.

In the latter case one tester plays the role of the Customer WS and another the role of the Warehouse WS.

Scenario 2 – Test also (one port of) the Warehouse:

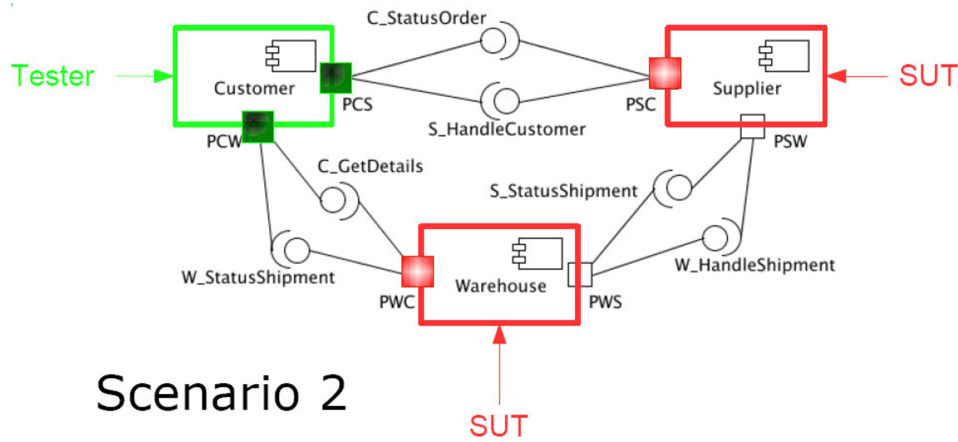


Figure 22 : The environment of the customer is tested

This scenario corresponds more to a testing of the environment of the Customer instead of the Supplier. Here SSM specifications are needed which describe both the PSC and the PWC port. Since connected ports are symmetric this corresponds to having an SSM specification of the Customer WS comprising both the PCS and the PCW port.

Testing within such a coordinated setup requires further extensions and adaptations of the underlying testing theory and framework. We name next just some open issues.

Open Issues

1. How to deal with timeouts and input-enabledness when several ports are involved?
2. Given a full data-enriched specification of a coordination, how to extract the model describing the desired SUT ports?
3. How to avoid testing scenarios where the initiative to start a conversation is not controlled by the tester (like testing the environment of the Supplier)?

See also [42] for further issues and more details. Within PLASTIC a main research activity is focused on solving these issues so that we can develop extended theories and tools which can deal with complex coordinated setups.

4.4.2 The AMBITION tool

The main milestones for implementing the AMBITION tool are these:

1) Support testing a single, passive WS
2) Support testing a single, active WS
3) Support testing scenarios in a coordinated WS setup (given that sufficient theoretical results are available)

Currently we are concerned with the first milestone. Within PLASTIC we aim at reaching at least milestone 2.

General Description

We assume the SUT to have a published WSDL-file which can be used to access it. A Multiport-SSM specification, given in an XML schema format, points to this WSDL, and extends it with the dynamic aspects (states and transitions). The AMBITION tool is invoked with such a Multiport-SSM. The main supported testing-mode is on-the-fly testing, which is described next.

4.4.2.1 On-The-Fly Testing

Here, instead of firstly computing a set of test cases from the specification, and then applying them to the SUT, the tester generates a single input, applies it to the SUT, and continues w.r.t. the observed response of the system. As a consequence the state space explosion when generating test cases is avoided.

4.4.2.2 The On-The-Fly Testing Algorithm

We give here a simplified version of the on-the-fly testing algorithm which will be implemented by AMBITION. The full description can be found in [44]. It tests the SUT based on a Multiport-SSM specification.

The algorithm keeps track of a set ***C*** of instantiated states. An instantiated state is an SSM-state together with a valuation of the global variables. ***C*** may not be a singleton due to nondeterminism. When giving an input, the algorithm computes for each defined input message the disjunction of the guards at the outgoing transitions of states in ***C*** where the message appears. These disjunctions are called the input constraints. Next, an input message together with message parameter values is chosen such that the corresponding input constraint is true. After an input is given or an output observed, the new set of possible instantiated states ***C*** is computed by the ***after*** operation.

A special observation embedded in the underlying theory is the observation of quiescence, meaning the absence of possible output actions. The machine can not produce output, it remains silent, and only input actions are possible. When it comes to implementing, this detection of quiescence is mapped to a timeout. Dealing with quiescence increases the discriminating power of the underlying implementation relation, see [103].

After computing the set of possible initial instantiated states ***C*** the algorithm executes a finite number of applications of the following three non-deterministic choices:

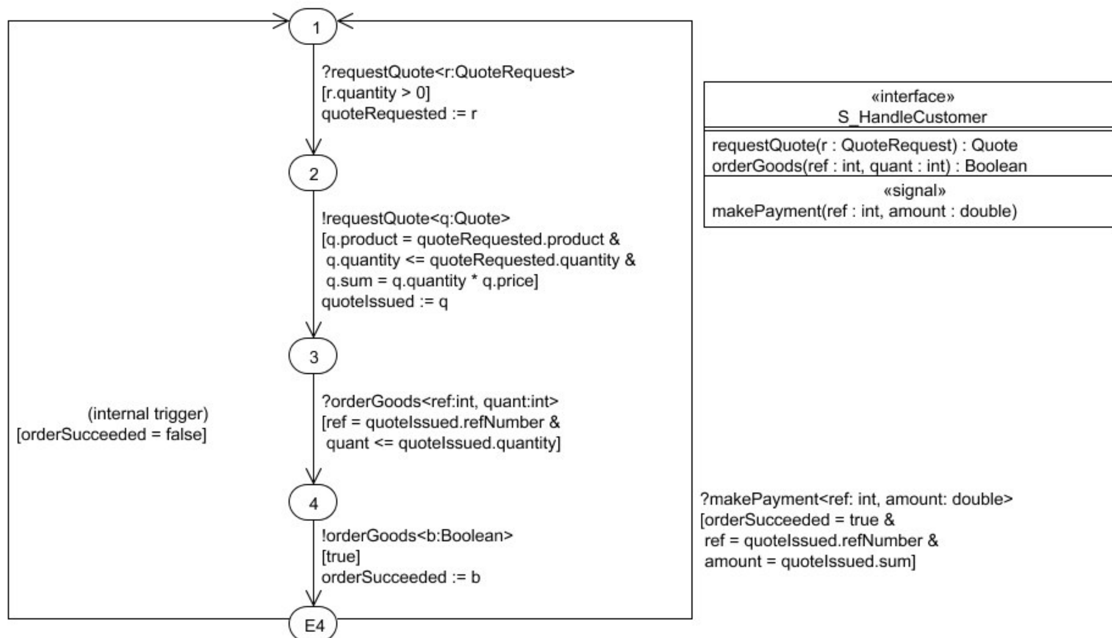
(1) <u>Stop testing</u> and give the verdict **pass**
(2) <u>Give input to the SUT</u>
        Compute the input constraints for ***C***
        Choose an input message ***inp*** which matches its constraints
        Send the input ***inp*** to the SUT
        Compute ***C' = C after inp***
        Repeat the algorithm with ***C'***
(3) <u>Observe output of the SUT</u>
        If quiescence is observed then
                compute ***C' = C after quiescence***
                If (***C'*** not empty) then repeat the algorithm with ***C'***
                else give verdict ***fail***
        else
                Observe output message ***out***
                Compute ***C' = C after out***
                If (***C'*** not empty) then repeat the algorithm with ***C'***
                else give verdict ***fail***

This algorithm is already prepared to test a single, active WS. When dealing just with a passive WS it can be substantially simplified, for instance quiescence cannot occur in this

case. This can be seen in the following main test loop, which is an excerpt of the current AMBITION prototype Java source which supports testing a single, passive WS:

```
void startTestLoop() throws FailureDetected {
   while (!endTesting) {
       // compute next input
       InputEvent pendingInput = computeNextInput();
       // no further input specified
       if (pendingInput == null) {
           addToLog("SUT passed this test run.");
           endTesting = true;
       } else {
           addToLog("generated input: " + pendingInput);
           // apply the computed input to the SSM
           // (may throw an exception)
           giveTestEventToSSM(pendingInput);
           // apply the computed input to the SUT
           if (isSynchronousCall(pendingInput)) {
               OutputEvent pendingOutput = invokeSync(pendingInput);
               addToLog("received return  " + pendingOutput);
               // apply the received output to the SSM (may throw an ex.)
               giveTestEventToSSM(pendingOutput);
           } else
               invokeOneWay(pendingInput);
       }
    }
}|
```

The following is an SSM model of a simplified, passive Supplier WS:



Next we give an exemplary log-extract where AMBITION tests a faulty, passive Supplier WS based on the upper SSM specification:

```
Ambition: started.
Ambition: SSM Manager: 1 instantiated State(s):
```

```
State: 1: [QuoteRequest with quantity 0, Quote with quantity 0, false]
Ambition: generated input: ?requestQuote[quantity 8929]
Ambition: SSM Manager: 1 instantiated State(s):
State: 2: [QuoteRequest with quantity 8929, Quote with quantity 0, false]
Ambition: received return: !requestQuote[quantity 1115]
Ambition: SSM Manager: 1 instantiated State(s):
State: 3: [QuoteRequest with quantity 8929, Quote with quantity 1115,
false]
Ambition: generated input: ?orderGoods[1, 442]
Ambition: SSM Manager: 1 instantiated State(s):
State: 4: [QuoteRequest with quantity 8929, Quote with quantity 1115,
false]
Ambition: received return: !orderGoods[true]
Ambition: SSM Manager: 1 instantiated State(s):
State: E4: [QuoteRequest with quantity 8929, Quote with quantity 1115,
true]
Ambition: generated input: ?makePayment[1, 2.884183224661866E7]
Ambition: SSM Manager: 1 instantiated State(s):
State: 1: [QuoteRequest with quantity 8929, Quote with quantity 1115, true]
Ambition: generated input: ?requestQuote[QuoteRequest with quantity 4720]
Ambition: SSM Manager: 1 instantiated State(s):
State: 2: [QuoteRequest with quantity 4720, Quote with quantity 1115, true]
Ambition: received return: !requestQuote[Quote with quantity 2318]
Ambition: SSM Manager: 1 instantiated State(s):
State: 3: [QuoteRequest with quantity 4720, Quote with quantity 2318, true]
Ambition: generated input: ?orderGoods[2, 432]
Ambition: SSM Manager: 1 instantiated State(s):
State: 4: [QuoteRequest with quantity 4720, Quote with quantity 2318, true]
Ambition: received return: !orderGoods[false]
Ambition: SSM Manager: 2 instantiated State(s):
State: E4: [QuoteRequest with quantity 4720, Quote with quant. 2318, false]
State: 1: [QuoteRequest with quantity 4720, Quote with quant. 2318, false]
Ambition: generated input: ?requestQuote[QuoteRequest with quantity 225]
Ambition: SSM Manager: 1 instantiated State(s):
State: 2: [QuoteRequest with quantity 225, Quote with quantity 2318, false]
Ambition: received return: !requestQuote[Quote with quantity 266]
Ambition: ------ FAILURE DETECTED ------
```

The detected failure corresponds to the transition from state 2 to state 3. Here the guard requires that the quantity of the offered quote is less or equal than the requested quote. This does not hold in the upper example, the offered quote (266) is greater than the requested one (225).

4.4.2.3 Open Issues

Regarding on-the-fly testing there are several issues still to be solved:

1. Inputs must be generated which match the constraints of the guards. This should happen fully automatic, which is currently not the case. Several approaches exist here which will be considered.
2. Since the testing has to be halted at a certain point, a suited halting criterion has to be defined. Several obvious candidates exist here, like transition coverage of the SSM. Also more sophisticated approaches like simulation-based testing will be considered.

One obvious possible extension of AMBITION is to also support testing via predefined test suites. Such test suites should also have a symbolic character (i.e., they contain variables and guards, etc.). The testing theory which underlies AMBITION does not have a concept for such symbolic test cases, yet. Such predefined test suites could for instance be developed for standard WSDL-interfaces (for instance based on a standard SSM-specification for such

an interface). Every service which claims to adhere to such a standard specification could then be tested based on the suite. Such a suite could fulfill specific criteria, tailored to the application domain, like model coverage criteria.

4.4.3 The JESSI tool

Since we aim at testing (and monitoring) PLASTIC services based on SSM specifications, a comfortable editing tool for SSMs is mandatory to allow developers to model and simulate SSM specifications. We adhere to the current trend in modeling by focusing on graphical models (derived from the UML), and we plan to develop a graphical editing capability for SSM as well within the PLASTIC editor.

General Description

Every SSM model is based on a WSDL file. When creating a new model, the WSDL is read, and the defined XML-schema data types, messages and operations are extracted. The given messages can then be used to model SSM transitions, together with guards and variable updates, as shown before. To do so, additional global variables can be defined. The modeled SSM can then be saved in a dedicated XML-schema format (which also serves as the input format for AMBITION). This process is depicted next.

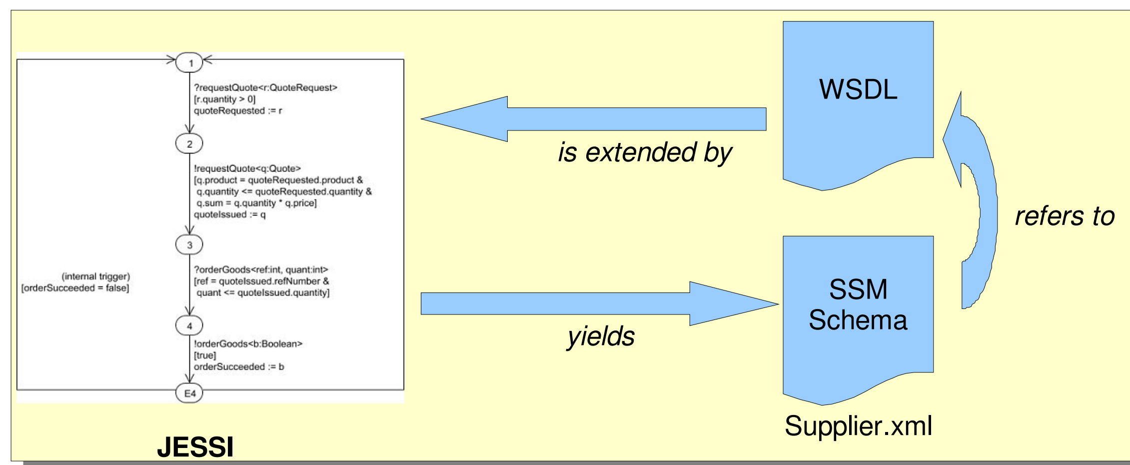


Figure 23: The JESSI approach

Besides editing an SSM, JESSI can also be used to simulate it. Here the service provider can explore and validate the behavior of the modeled SSM by giving inputs or outputs (including quiescence) to the model, and observing the change in state and variable values. To do so, JESSI uses an SSM simulator, which is also the core component of AMBITION.

JESSI and AMBITION

In the off-line testing phase, the AMBITION service will take the generated SSM schema (which points to the corresponding WSDL), and, based on this, automatically test the services modeled. The complete off-line MBT approach via JESSI and AMBITION is summarized in the next figure.

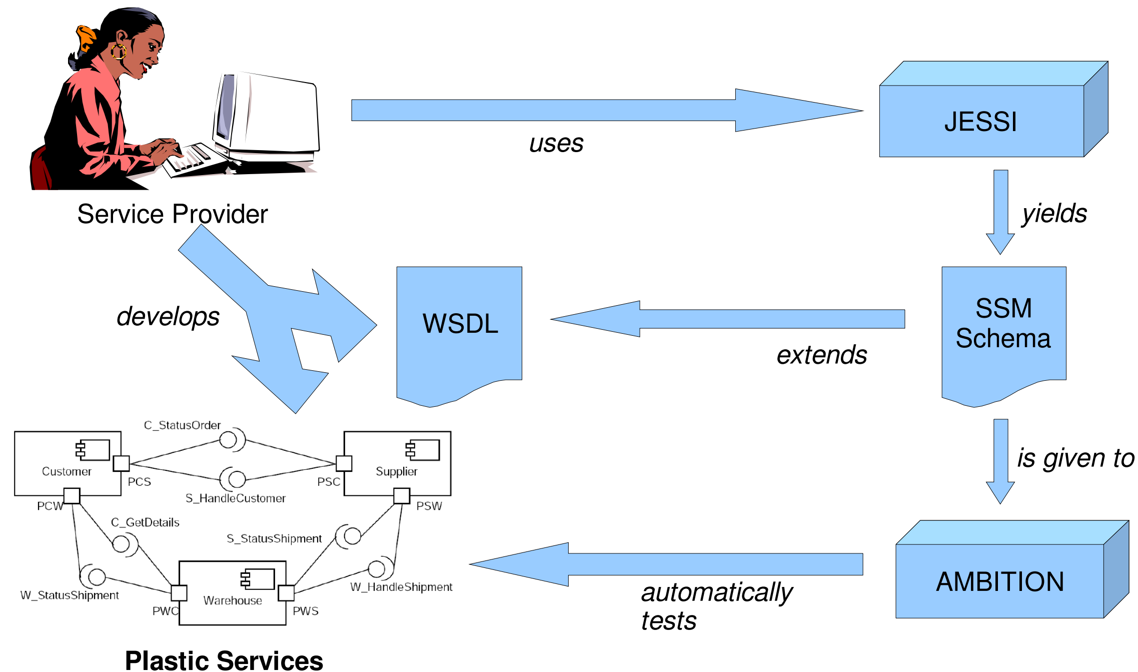


Figure 24 : The off-line MBT approach via JESSI and AMBITION

When being used within AUDITION, the SSM models generated by JESSI will be stored in the PLASTIC registry, and the AMBITION tool will be encapsulated into a WS itself, so that it can be called from AUDITION. This will be further explained in Chapter 5.

4.5 QoS Evaluation

The QoS aspect of the off-line validation stage concerns an approach for the automatic derivation of test harnesses. We call such approach Puppet. The goal is to evaluate different QoS characteristics for a service under development and before its final deployment. In particular, such approach focuses on assessing that a specific service implementation can afford the required level of QoS (e.g., latency, reliability and workload) defined in a corresponding Service Level Specification (SLS) for a composition of services (choreography/orchestration) in which the Service Under Evaluation (SUE) will play one of the roles.

The technologies used in PLASTIC include for each service a specification describing the functional interface exported by the service (e.g. WSDL), a description of the services that compose it (e.g. in terms of WSBPEL [85]), and a machine readable specification of the QoS agreement for the services in the composition. At this point, the goal of the tool for the QoS evaluation in the Off-line validation stage is to automatically generate a test harness to validate the implementation of a service before its deployment in the target environment.

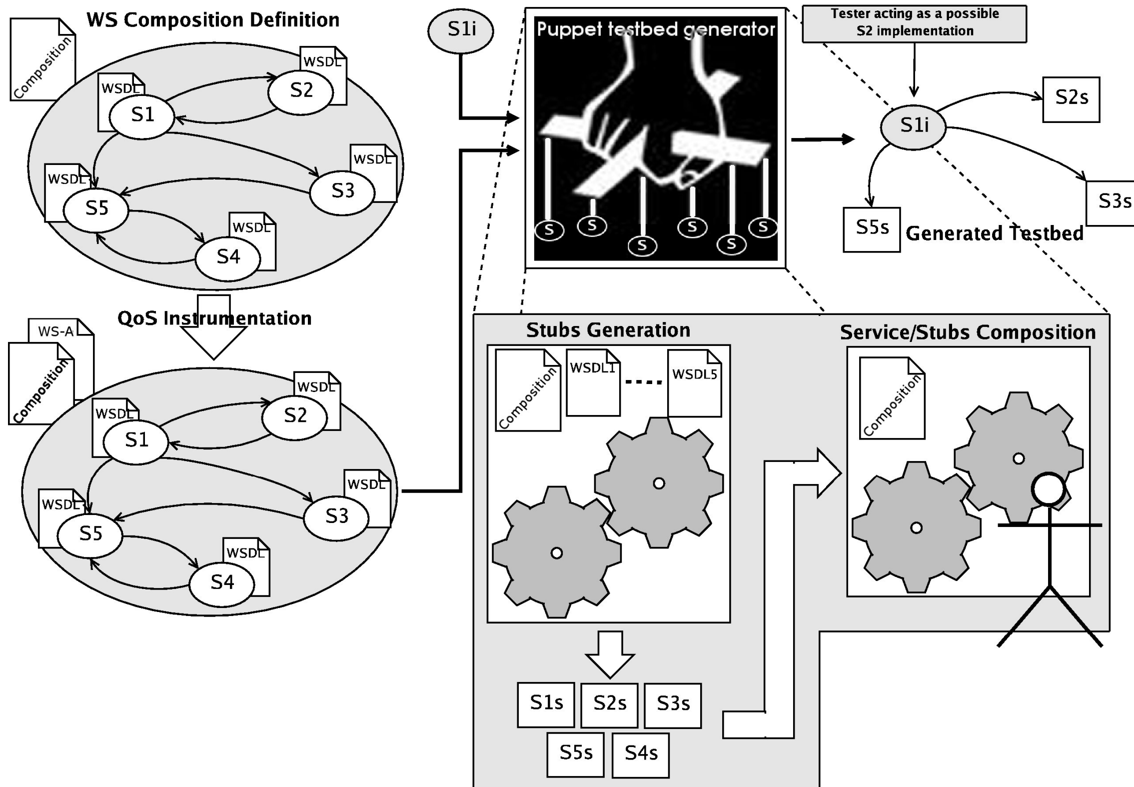


Figure 25: Puppet approach and supporting tool

Concerning the SLS specifications, the proposed approach is quite general and does not focus on a specific technical solution. Even thought SLAng [100] is included as reference language for SLA descriptions in PLASTIC and in line with the minor technical issues discussed in Section 3.4.2, the tool here presented will consider QoS specification based on WS-Agreement. We remark that the two descriptions will consider concepts with the same semantics defined in SLAng by means of the PLASTIC Conceptual Model.

As illustrated in Figure 25, the generation of the test harness proceeds through two different phases. The first one is the generation of the stubs simulating the extra-functional behavior of the services in the composition; the second one, instead, foresees the composition of the implementation of a service, called “S1i” in Figure 25, with the services with which it will interact. In the following, both phases will be described to give a complete overview of the approach. A proof-of-concept implementation of the proposed approach is under development [15].

The generation of the stubs consists in turn of two successive sub-steps (see Figure 26). In the first one the skeletons of the stubs are generated starting from the functional interface description of the service (e.g. WSDL). The generated skeletons contain no behavior. Hence, in the second sub-step the implementation is “filled in” with some behavior that will fulfill the required extra-functional properties for the service corresponding to the stub. This step is carried on retrieving the information from the SLS and applying automatic code transformation according to previously defined patterns matching each SLS with a portion of code that simulates its behavior. At the end of the first phase, a set of stubs providing the services specified in the composition according to the desired properties are available.

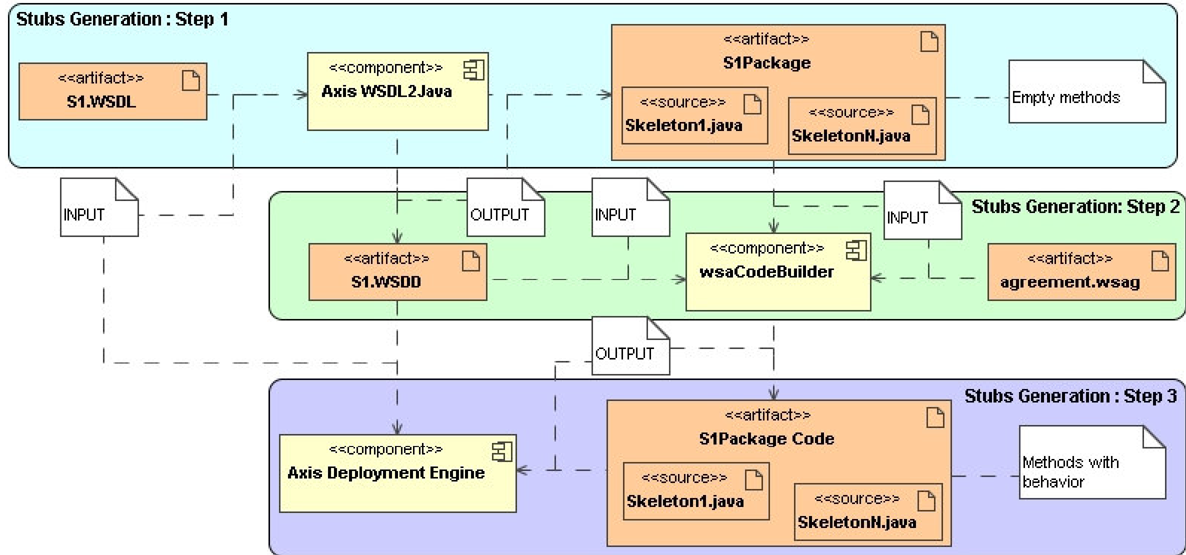


Figure 26: Stubs generation logical process

Back to the proposed approach illustrated by means of Figure 25, the second phase concerns the setting of the test harness. The goal of this step is to derive a complete environment in which to test the service. To this purpose, the SUE, “S1i” in Figure 25, is composed with the required service and according to the composition specified in the choreography or in the orchestration. Even thought this phase could require the assistance of a human agent[5], one of the main goal that we would like to reach is implementing a complete automatic process based on the forthcoming final WSBPEL specification.

As final result, the application of the proposed approach generates an environment for the evaluation of “S1i”. The evaluation, to be carried on, will then require the availability of a tester, as also reported in Figure 25. This tool is beyond this specification; we can refer to the literature on the argument for possible approaches, e.g. [21]. Such a tool will have to verify that the properties specified in the QoS document (e.g. a SLAng document) are fulfilled, in addition to traditional functional testing.

<table>
<tr>
<td><pre>...
<wsag:ServiceLevelObjective>
  <puppet:PuppetRoot>
    <puppet:TagDelay>
      1000
    </puppet:TagDelay>
  </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...</pre></td>
<td><pre>...
try{
Thread.sleep(1000);
}
catch
(InterruptedException e)
{}
...</pre></td>
</tr>
</table>

Table 4: Example of latency mapping

In the following, a detailed description about the technologies and the tools that will be used is given. In particular, to better explain how the offline validation of QoS properties in a SOA is carried out, the description will explicitly focus on the Web Service infrastructure.

The generation process described above, exploits the information about the coordinating scenario (WSBPEL), the service description (WSDL) and the WS-Agreement document for

[5] As illustrated by the stick man in Figure 25

the QoS agreement that the *roles* will abide to. Tools and techniques for the automatic generation of service skeletons, taking as input the WSDL descriptions, are already available and well known in the Web Services communities [6]. Nevertheless such tools only generate an empty implementation of a service and do not add any logic to the service operations.

Concretely, once a parametric mapping between specification of metric value and the executable code that will be used to characterize the services in the test harness is defined, the empty implementation of a service operation are processed adding the lines of code resulting from the transformation of the service agreements specification.

Conditions on latency can be simulated introducing *delay* instructions into the operation bodies of the services skeletons. For each `Guarantee Term` in a WS-Agreement document, information concerning the service latency is defined as a `Service Level Objective` according to a prescribed syntax. The example in Table 4 reports a WS-Agreement example code for latency declaration of 10000*mSec* and the correspondent Java code that will be automatically generated.

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

According to what described in the conceptual model, the SLA can be enforced under optional conditions describing the context. Such additional constraints are usually defined in terms of accomplishments that a service consumer as well as a service provider or the service running environment must meet: for example the latency of a service can depend on the kind of the network on which the service in deployed when the request is delivered. In these cases, the transformation function wraps the simulating behavior code-lines obtained from the `Service Level Objective` part with a conditional statement.

Constraints on services reliability can be declared by means of a percentage index into the `Service Level Objective` of a `Guarantee Term`. Such kind of QoS can be reproduced introducing code that simulates a service container failure.

```
...
<wsag:ServiceLevelObjective>
  <puppet:PuppetRoot>
   <puppet:Reliability>
    <puppet:TagRate>
      98.00
    </puppet:TagRate>
    <puppet:Window>
      2000
    </puppet:Window>
   </puppet:Reliability>
  </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

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

Table 5: Example of mapping for reliability

According to the definition given in the conceptual model [93], QoS specifications concerning reliability constrain the number of failures that can be seen in each of those modes within the duration of a sliding window. Table 5 provides an example of the transformation for reliability constraint description, assuming that the Apache Axis [6] platform is used.

Agreements on workload assessing can be simulated creating client skeletons for the automatic invocation of the SUE. In particular, the transformation will focus on generating client-side code that is able to guarantee that the rate at which requests can be delivered to the service, the width of a sliding time window and the maximum number of responses that

should be delivered across the service interface during this period [93]. Furthermore, as in the case of latency, it is also possible to implement a transformation function considering a specific distribution of the request arrivals.

4.5.1 Open issues

As argued, the proposal does not explicitly address the generation of test suites. Rather, it addresses the automatic generation of the infrastructure that could be used as a test harness for pre-deployment QoS evaluation of a service. However, the relation between the approach described in Section 4.5 and specific test case selection techniques represent an interesting open issue. In particular, the current proposal does not address how to associate an appropriate mechanism in order to instrument the return value of the methods for each generated stub. At the same time, also its dual situation needs to be further investigated. Specifically, concerning the creation of client skeletons for service workload assessing the generation of appropriate input parameters for the operations exported by a SUE should investigated.

4.6 Summary

In this chapter we used model-based specifications to drive the testing and validation of service based systems, like those written for the PLASTIC platform. Model-based validation is the newest trend in the field of testing reactive and distributed systems. It has the advantage of being more cost- and resource effective than traditional techniques, as testing is automated and the cost is not increasing linearly with the scale of the distributed system. It is also less error-prone compared to traditional testing techniques. We used the latest results in the field, e.g the Symbolic Transition System theory for conformance functional testing, adapted and extended to deal with the PLASTIC platform requirements.

We presented some approaches for functional testing (MBT), QoS (Puppet) and simulation-based testing. All of these can be experimented by means of the Weevil system for distributed experimentation. We believe that these techniques will help to achieve a better productivity and coverage of the overall PLASTIC testing process.

5 On-line pre-publication stage: Audition

5.1 Introduction

The Audition framework (whose ideas has been introduced in [19] [17]) has the main objective to support the evaluation of a service, via testing, at the time it asks for registration. The rationale behind this approach is that verification of services in the service domain is particularly difficult and asks for new solutions. Many reasons can be listed to justify such difficulties among them the high dynamism of service based applications, by which services can discover each other and start interoperating at run-time, and the wide openness of the environment, that allows services to be deployed and registered at any time immediately entering possible usage scenarios. As a consequence the final execution environment is not predictable and tests carried on by the developer does not add much confidence on the behavior of the service when deployed. The main idea of Audition is to introduce mechanisms that permit to test a service when it asks for registration and then when it is inserted in the final execution environment.

In the following sections possible usage scenarios for the framework are discussed. The framework is based on some assumptions that are not satisfied in a general service domain. In particular the most important assumption is that the environment is in some way semi-open, meaning that stakeholders that deploy and expose services are in some way reliable and do not try to act in a malicious way. Moreover the framework relies on an augmented service model in which services are described trough specific behavioral models suitable for test case derivation. Such models can be stored and retrieved via the directory and discovery service. In this chapter the term Audition refers independently to the framework and to the corresponding testing phase. The chosen name is certainly in line with the terms used in the WS domain and want to give the idea of a judgment of the service before putting it on the “scenes”.

The remainder of this chapter provides a high level overview of Audtion scenarios, issues and solutions and the architecture of the framework. In particular the next section describes several scenarios in which to use the Audition framework. For the different scenarios possible testing targets and technology issues are discussed. Successively Section 5.3 discusses the architecture of the part of the framework that will be developed within PLASTIC, focusing in particular on the technologies that will be integrated. Finally a summary of the chapter is provided in Section 5.4

5.2 Scenarios for Audition

In Section 2.2 it has been discussed which are the possible stakeholders interested in testing a service or a set of them. To the list proposed by Canfora and Di Penta [22] one element has been added, that is the directory and discovery service. This section intends to illustrate which could be the motivations that can push the provider of a directory and discovery service in augmenting the service with functionality for testing services that ask for registration. Nevertheless before providing motivations on its possible usage it is worth to provide some more detail on the framework itself.

Audition relies on the possibility of testing the service during the registration. Current specifications of directory and discovery services do not include such functionality. In order to add it, first assumption taken by Audition is that operational models of the registered services are available in the repository, or alternatively that a test suite is stored together with the definition of a service (within PLASTIC we assume that the operational models used for registration are those described in the previous section on off-line testing). When a service asks for registration the corresponding model is retrieved from the registry and passed to a

testing engine to start a testing campaign. Alternatively the testing engine could directly receive from the directory service the corresponding registered test suite and start the execution of it. In case neither a model nor a test suite for the service are available in the registry the service as to provide it when asking for registration. However this last opportunity has to be considered as an exception and does not permit to take full advantage of the characteristic of the framework given that the model is provided by the developer itself. The testing engine, that can be considered a supporting service as it is the directory service, receives together with the model or the test suite a reference to the WSDL definition for the service that asked for registration. Received all this information the testing engine service starts to make invocation on the Service Under test and checks if the answers it gets match the one specified by the model or in the test suite. In case some mismatches are discovered the registration fails.

A more complex version of the framework extends the checks carried on by audition to the monitoring of the interactions of the service under registration with required services. As consequence of the invocations made by the testing engine service the SUT could need to interact with other service providers. In order to do this the service under test asks to the directory and discovery service a reference to a service implementing the required service. At this point the discovery and directory service, using mechanisms that should be provided by the platform activates the monitoring of the messages exchanged between the service under test and the service provider. Scope of this monitoring activities is to verify that the service under test actually behaves as expected by the services provider, making invocations that conform, in terms of sequence and data constraints, to the model defined and registered for the service provider. If during the verification phase no errors emerge the registration will be guaranteed.

Audition can be applied to testing of single services, however it provides greater benefits when applied to testing of services playing a role within predefined compositions/collaborations. In such case also the interactions among services can be in fact observed and verified. Nevertheless involving running services in a testing session can be dangerious and must be done with particular care, since without a suitable support, and in presence of services managing stateful resources, it gives origin to the commitment of “fake” transitions with obvious drawbacks.

In order to derive possible application scenarios for the framework we have to answer to some questions that could provide a clearer picture on the peculiarities of testing in the service domain when it is done at registration time.

5.2.1 Audition Testing Targets

The first question is what can be the test target. The following list probably does not give an exhaustive answer but provides an interesting start for further investigations on the applicability of Audition.

[1] Services to be registered on a discovery service:

a. stand alone service developed on the base of internal specifications: these are services such that at the time they ask for registration no one else has already published a model for the service. Clearly this is not the best opportunity for applying the Audition strategy. Given that the specification is defined by the service provider itself, it is reasonable to foresee that it will probably define a underspecified and less precise model to reduce the number and the power (in terms of probability of discovering a fault) of the test cases used during the evaluation phase. If this strategy could be effective during the registration phase at the same time it will probably reduce the number of possible interested service users, at least if the inquiry for a service on the UDDI is based on the registered models, with clear drawbacks for the service provider. Repeating tests defined by a developer could seem a bit useless. Indeed it could be interesting to do it for a matter of trust. Tests

are in fact executed by an external (the registry) actor, so clients will certainly feel more comfortable with the quality of the service. If no specification is provided the service cannot obviously be retested. Nevertheless a conscious developer will know that in this case the lack of documentation and confidence will highly reduce the usage of the service.

b. stand alone service developed on the base of published specifications: this case is certainly relevant for Audition and probably is the one that shows less technical issues. The service, in fact, will not invoke any external service provider avoiding the necessity for the management of dummy transitions derived by the execution of test cases. Being the specification already published, possibly by recognized actors, avoids the risk discussed in the previous case. Nevertheless it is our understanding that the real potential of the service based paradigm is strongly fostered by the emergence of vertical standards i.e. standards describing services for a particular application domain The availability of standards strongly simplify the development of service, making easier the selection and use of defined services.

c. composite orchestrated service non accessing external services: from an Audition point of view this case does not show any difference with respect to case a) and b) when the specification is not registered or it is, respectively.

d. composite orchestrated service accessing external services whose specification is published: as case a) and b) but in this case we can also apply techniques for the monitoring of the interactions with other services as described above. Applying Audition, and using the mechanisms for monitoring possibly provided by the platform, is possible to verify that the service under test correctly interacts with the other services.

e. complex services to play a role on a published choreography: somehow this is the most interesting scenario for the application of the audition framework and the place in which it fits better. In this case the service asks for registration in a registry specifying also the roles in one or more choreographies that it wants to play. The Audition framework in this case asks for the recovering of the model specification of the choreography/ies and should then derive a model view for the particular service. Using this information and knowing the interactions with the other services in the choreography the Audition wants to actually verify that the service correctly plays the role it declared.

Among the five listed scenarios the PLASTIC project will mainly focus on the first four. Up to now the project does not specify any technology for the description of choreographies such as WS-CDL. Nevertheless the applicability of the approach to the case of choreography asks for the deeper investigations, even from the theoretical side, to derive a projection of a the service behavior implied by a choreography specification.

[2] <u>Services accessing other services that will not be registered:</u>

- It is certainly possible and relevant to test services that do not ask for registration on a directory service. Nevertheless in this case a service user will have the opportunity of interacting with such a service only if the developer directly knows a reference to the service or if a reference is returned by another service provider.

[3] <u>Services statically accessing other services:</u>

- According to the service paradigm it is also possible for the developer to statically bind required services within a service implementation. In this case if the service will be registered within an “audition enabled” directory service the interactions with the service server will be invisible to the audition checking mechanisms. In this sense Audition indirectly considers the service as the collection of it and of the statically bound services. In case of a failure only the service itself will be considered “guilty” and the registration will be refused.

The list above is probably non exhaustive and other targets can be identified. Nevertheless the items in the list above it is meant to identify the most interesting scenarios for the applicability on the Audition testing phase.

5.2.2 Kind of Checks Used During the Audition Testing Phase

Without distinction on the kind of service target of an Audition phase, two different possible checks can be performed on a deployed service. The first one focuses on the outcome expected from the execution of an invocation or a sequence of invocations. In the following of this document this case will be referred to as “Implementation Based Conformance (IBC)”.

The second possible kind of check asks for the monitoring of the invocations made on a required service to verify that they are actually correct with respect to the protocol expected by the service provider. This kind of check will be referred here as “Usage Based Conformance (UBC)”.

In general IBC can be verified using the specification defined and registered for the implemented service. In Chapter 4 a possible technique for deriving and applying test cases to check the implementation of a service has been described. The technique is actually the best candidate to be adopted to implement the Audition framework within PLASTIC.

As consequence of the invocations made during the IBC testing, services that require other services to carry on their task will possibly start to interact with such required services. The verification of such interactions is in the scope of the UBC. The objective in this case is trying to verify that the service actually makes correct invocations, in terms of parameters values and order of messages, on the required services. To check this the specification for the invoked service is considered. Such specification should be available within the registry. Figure 27 provides a visual description of the activities carried on during the Audition phase and on the needed information to check for IBC and UBC.

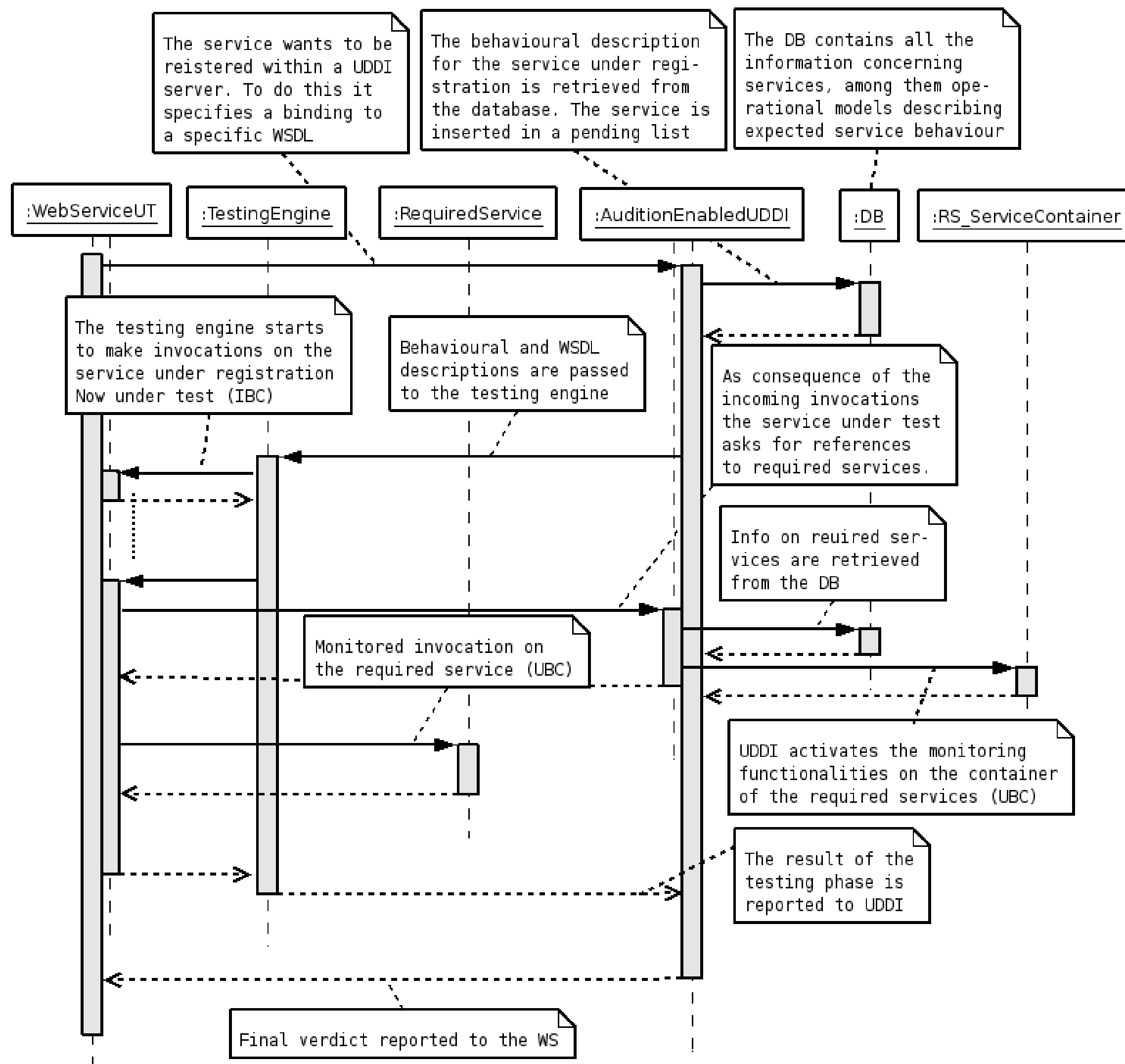


Figure 27: Audition components and interactions

The possibility of putting in place the whole framework described in Figure 27 is really challenging. The development of mechanisms for UBC is appealing, but their implementation is complex and presents many theoretical and technical challenges. Within PLASTIC our scope is to develop mechanisms to put in place checks classified as IBC, and possibly to explore the domain of UBC trying to figure out possible approaches. As consequence within PLASTIC during the Audition phase a service will be only evaluated on the base of the answers it gives to an invocation without exploiting information concerning its interactions with service provider.

5.2.3 Putting in Relation Service Testing Scenarios and Check Typologies

Answering to the question "*What are we interested in testing and where audition can play a role?*" we highlighted a set of possible testing scenarios and the specification that we could use in the different case. Here we provide some more detail on who should provide the specification for a service that asks for registration. For audition purpose we only consider

services that are intended for being registered within a directory service. The items in the next list correspond to those in Subsection 5.2.1 above:

a. In this scenario the developer will directly test the service on the base of an internal specification for the service, possibly comprising interface usage definitions specified using WSCL or similar languages. Successively the developer should store in the registry the specification representing the correct usage of the service. This should be done in order to document the service itself providing a semantically richer specification w.r.t, WSDL. Within PLASTIC we assume that such specifications can successively be translated into the formalisms presented in the chapter on Off-line testing.
In the scenario we are considering the service will not call any other service to carry on its task, so the verification step does not present major issues. At the same time the developer could directly derive a test suite, given its knowledge of the specification and of the implementation (White-Box testing), and pass it to the registry for testing in the field. In some way these recall the approaches to built-in testing developed within the CB community. According to the classification above testing will provide in this context a mean for IBC verification of the service. Nevertheless the availability of a specification describing the accepted protocol for the service under registration will successively enable the checking of the interactions performed by possible service users (see below) for UBC.

b. In this scenario the developer is implementing a service whose specification has been defined by someone else. The specification could have been provided through a WSCL specification, or a XML based representation of an automata, and stored in the registry. This scenario assumes that the specification is available on a repository accessible to the developer. Several organizations could be interested in such a scenario. As an example consider an important portal that wants to provide a selection of specific products to clients. They could define a specification of the behavior they assume on their service providers, i.e. on the behavior of services that they will contact asking information on products and prices. This service will be probably deployed/developed by the smaller company.

c. According to this scenario the developer on the base of the internal specification and having complete control over the environment can internally check the orchestration (implementation of the composition). At the same time it could be interested in publishing, as in case a), a specification of the observable behavior of the service in the registry. From a user perspective this case is not distinguishable from case a) and the same type of conformance is applicable in the same scenarios.

d. The composed service will access external services registered in a registry. Access to services for which conversation protocol have been stored enables checking the verification of UBC for the specified protocol. As in case a) and c) registering the service the developer should store a specification of the accepted conversation, that will permit UBC for possible future users of the service.

e. In this case the developer will take as reference an external specified choreography probably released by some standard body. From this specification the developer derives a view of the role that the service s/he is implementing will play. Such specification will be the reference for the implementation of the service as illustrated also for the scenarios above. The choreography will specify at the same time the correct behavior for the service interface to be implemented and for the usage of the other services foreseen by the choreography. This scenario is certainly the most interesting and in our opinion is the one that could really bring the service oriented computing a step above all the other proposals. Many usage scenarios can be imagined for such an approach. Particularly interesting seems the trend taken within

the European STREP Project TELCERT[6] ended in June 2006, and more in general within the e-learning community. In such a domain it is emerging the idea of organizing complex application such as Learning Management Systems (LMS) using choreographies. LMS are complex softwares that permit to completely manage the needs of an eLearning organization in terms of delivering courses to students, managing pupils administrative data and CV, providing tools for courses authoring, and whatever is necessary in the eLearning world. The definition of choreography for each different task will strongly open the market of the developers of LMS, permitting to a single organization to focus on the implementation of a single service having strong guarantee that it will be able to cooperate with other services in the choreography that are implemented by different organizations.

Audition can be adopted to increase the confidence that a service user has on the registered services. Nevertheless the high dynamism of service oriented computing asks for mechanisms that allow to check the execution also during run-time i.e. monitoring. In this context Audition can be useful to provide suggestions to monitoring mechanisms that in generally cause a lot of overhead on the system execution. Monitoring then can be started only to check interactions of services whose composition has not been "audited". Certainly investigations in this direction are necessary.

5.2.4 Using Audition

This section shows how the framework we introduced can be used in different scenarions. Therefore taking as reference the classification defined in Section 5.2.1 for the case of "services to be registered on a discovery service" we have:

a. In case the spec for the service under registration is provided, the registry could resubmit the service to a testing session deriving the tests from the registered spec. If repeating the test could seem trivial, indeed it could be interesting for a matter of trust.

b. From the point of view of audition we can re-conduct this situation to a) when the spec for the service is provided.

c. This situation does not show, from the point of view of the registry, any difference from case a). The fact that the implementation of the service is actually an internal orchestration is completely transparent to the registry. The orchestration spec will never be registered in a registry. Simply it does not make sense to do it. So the same check of a) are applicable, and nothing more.

d. As case b) but in this case the orchestrated service can be checked for conformance against the usage of other services. In the initial definition of the spec this check should be carried on by "proxy services" but as we discussed many times we should opt for a solution that will encapsulate monitoring and checking mechanisms within the middleware.

e. Having choreography aware registry would be a great opportunity for the take-off of such kind of approach. The audition framework strongly fits in this vision combining the two kind of checks (usage based and interface based) that choreography specifications make available. So in this case a service should ask to be registered declaring also the roles in the choreographies in which it is interested in taking part. The registry using appropriate tool should derive test cases from the different choreography and start the execution of them.

[6] http://www.opengroup.org/telcert/

5.2.5 Requirements Issues

From the technological point of view the real implementation of the Audition framework requires some major improvements/modifications to the standard way of developing/documenting services and also on the services to be enclosed in the middleware as platform services such as for instance the Directory and Discovery service. Nevertheless we do not think that this is a major flaw of our testing approach. Main objective of PLASTIC is to develop a specific platform for service oriented computing that among its characteristics intend to increase the quality and the trust on provided services.

In general terms scope of the Audition framework is the introduction of an audition phase when the service asks for registration. The idea is to submit the service to a testing campaign and guarantee the registration to the service only when no mismatches, with respect to the considered specification, emerge. The evaluation is based on the two different kinds of check (IBC and UBC) depending on the kind of service under test and on the necessity of cooperating with other registered services.

Scope of this section is to detail the requirements/assumptions, in particular on the underlying technologies, in order to make the Audition phase a feasible task. For the sake of simplicity we base the description roughly following the sequence foreseen by the Audition validation phase described in Figure 27 (see [19] for details).

First assumption of the framework is the availability of a specification for the service under registration. How this specification is defined and who defines such specification has been discussed above nevertheless deeper investigations are certainly necessary. Currently we are considering two different options. The first is the availability of an abstract automata describing the accepted protocol both containing data or not (at the moment we are assuming formalisms as powerful as SSM). At the same time we are studying and developing tools for the automatic derivation of test cases from these specifications. Chapter 4 provided detail on the approach we intend to apply within PLASTIC. The availability of specific behavioral models and of algorithms to automatically derive test cases from such model permits to put in place a strategy for IBC validation. Strictly related to the availability of an operational model suitable for test-case derivation is the availability of a service acting as testing engine. Within PLASTIC we are working on the encapsulation of the AMBITION tool within a service that can be directly contacted by the UDDI server when an Audition phase must be started.

A second important requirement on services and platform imposed by the framework concerns the characteristics of the services to be developed and successively deployed within the PLASTIC platform. In particular we assume that PLASTIC services will be aware that they will undergo a testing session when asking for registration (to be confirmed by the registry when no test has positive result). This requirement directly derives from the necessity of removing all the effects caused by the interactions happened during the testing phase on stateful resources. Such invocations have to be considered not real for the service and in case of services handling stateful resources modification to the state must be undone. The same problem must be solved for stateful services providing functionality to services under an Audition phase. Therefore a basic requirement that the application phase poses on services is the ability of removing all the effects caused by invocations related to the audition phase. The solution to this problem is strictly related to how a stateful resource is defined and handled within the service. Currently we suppose that such functionality will be explicitly foresee by the service developer. Nevertheless we imagine that general solutions can be defined given the emergence of standard ways to handle stateful resources such as the WS-Resource proposal. Another important factor to consider in order to set up an environment that permits to remove the effects of the audition phase on the service provides, is the ability of recognizing all the invocations generated as consequence of the audition phase. Different solutions are possible also in this case asking for support by the platform or not. It is worth noting that for the case of the service under registration this problem is not particularly hard

and can be easily solved directly by the developer given that the service asking for the registration is aware that it has to undergo to a testing phase. The end of the testing phase is directly related to the notification of acceptance/rejection within the registry. For a running service involved in an audition phase as provider of a service it is instead not possible to recognize when a message is originated within an Audition session, unless specific support are provided by the platform. Solutions to this problems are currently under investigations nevertheless main focus in this phase has been the development of mechanisms to start the Audition phase.

Finally the last important requirement posed by the framework is the ability to observe the interactions as they happen in the deploying environment. This requirement asks for the availability of mechanisms similar to those required by run-time monitoring. The idea is that (as shown in Figure 27) the UDDI can directly interact with the container of a service to activate and deactivate the monitoring of a contained service. Also this feature is related to the required mechanisms for the verification of UBC properties which will be the topic of future research and for which ready solutions are not available yet.

The Audition framework has been defined and developed having in mind the availability of a centralized directory and discovery service following to which all services directly refer for publishing and discovering. Nevertheless within PLASTIC other kind of discovery services will be targeted such as notification based discovery. The applicability of the Audition phase in such a context are currently under investigation.

Another important issues that still has not be solved and that will require deeper investigations concerns how the framework can be useful to evaluate properties such as adaptability and context-awareness of services. Within the project these concepts are under definition, nevertheless we already started to investigate how the framework can be adapted to consider thse concepts also during testing.

5.3 Audition Architecture

In this section we provide a general description of the framework and of the technologies that will be used to implement it. As stated already above within the project we are currently targeting mainly IBC verification and the development phase has been mainly focused on providing support for such kind of verification. The development of mechanisms supporting UBC verification will not be further discussed and shown in this section since still some deeper investigations on how to implement them are necessary.

Implementing Audition within PLASTIC we intend to reuse as much as possible what already exists. Figure 28 shows which are the technologies that we are planning to integrate in order to have a running implementation of the framework. As can be observed all the technologies that we plan to use are open source and free distributed.

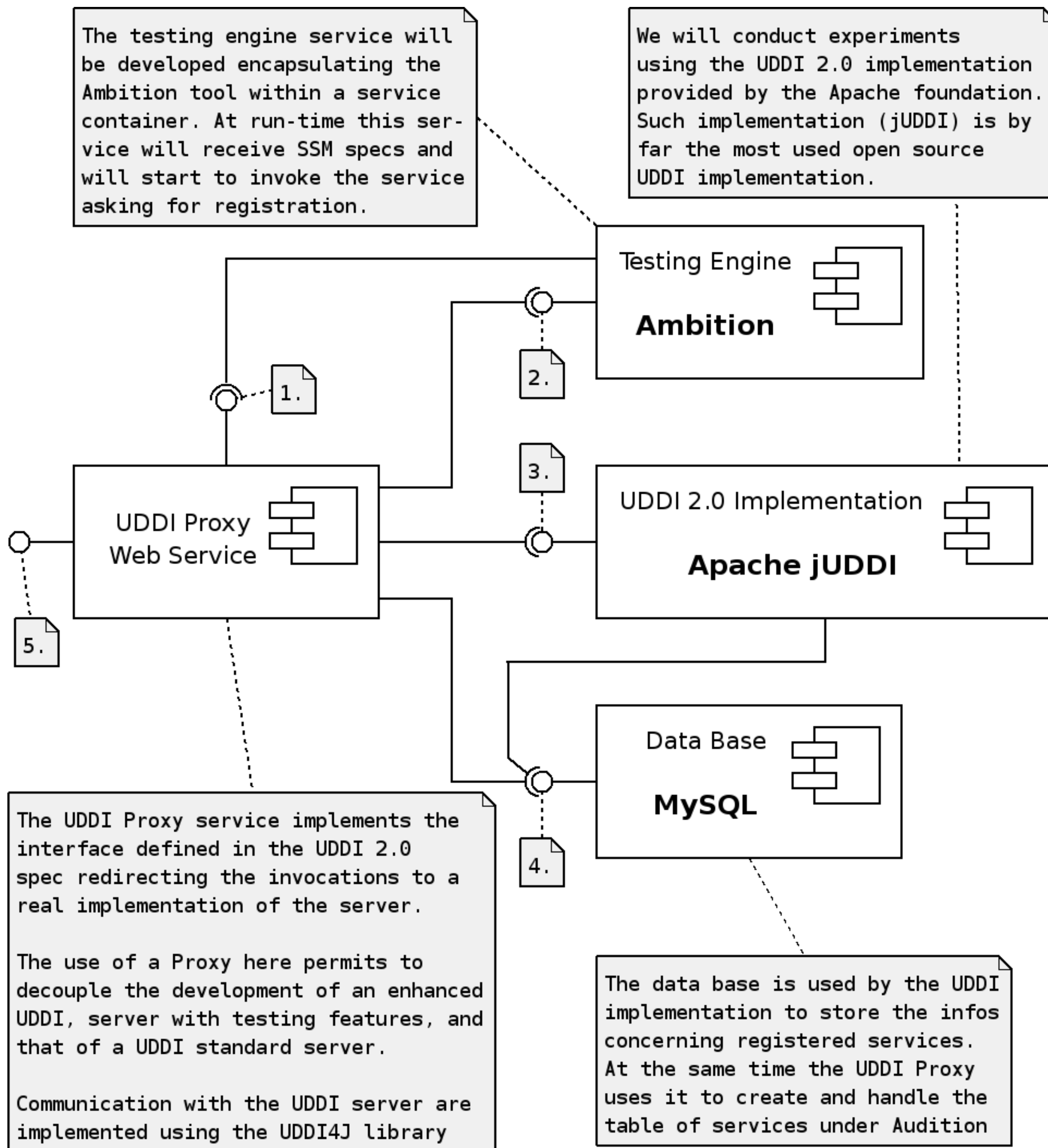


Figure 28: Audition components implementation

Following the labels given in the picture the functionality of each shown interface is:

1. This interface is used by the testing engine to notify the result of a testing campaign. According to this result the registration will be accepted or not;
2. This is the interface that allows the UDDI proxy to pass the information about a service that asked for registration. This information concerns the location of a WSDL file and the SSM model associated to the implemented service. After having received this information the testing engine (based on Ambition) will start to invoke and test the service.

3. This interface is actually the jUDDI interface. We will use it within the proxy of the UDDI4J library. This will make easier the implementation of the interactions with the server from the proxy.
4. This is the interface of the data base (MySQL in our implementation). The DB will be used by the Proxy to store information on the service under registration (pending table).
5. This is the standard UDDI interface used by services to publish and retrieve information on published services.

5.4 Summary

Audition is a framework that intends to introduce a supplemental phase in the evaluation of services. The whole framework bases the evaluation of a service on two different aspects. The first is related to the specification for the same service (called IBC) and the second instead refers to the usage of required services (called UBC). Within PLASTIC we are targeting at implementing mechanisms for the verification of IBC. Support for Usage Based Conformance presents many challenging issues that will be investigated in the future nevertheless real implementation for this part is not in the scope of the current phase of the PLASTIC project. Other interesting issues that we will investigate in the next future is the introduction of concepts such as context-awarness and adaptability in testing during Audition.

The implementation of mechanisms for Implementation Based Conformance will be carried on in the next phase of the project. The development will try to reuse already available technologies, as illustrated in Figure 28.

6 On-line Live-usage Validation Stage

6.1 Introduction

The goal of the PLASTIC project is to enable the development and deployment of cost-effective application services, both in terms of development and usage costs, regarding both financial and resource usage aspects, for B3G networks. Service development platforms for B3G networks can be effective and successful only if the services they deliver are adaptive and *offer quality of service guarantees* to users despite the uncontrolled open wireless environment, which is a key focus of the PLASTIC project. In this chapter we will describe the validation of PLASTIC services and service compositions, performed at run-time by using monitoring techniques.

As introduced in Sec. 2.3, the main goal of monitoring activities is to discover potential critical problems while a system is executing. The aim of monitoring is to collect data about a phenomenon and analyze that data with respect to certain criteria. The data to be collected and the analysis criteria must be defined as part of the monitoring parameters, and may change dynamically. The criteria may optionally state that certain external functions (called *reactions*) must be called depending on analysis results.

Data Collection concerns the storage and the access to the data that is relevant for the objectives of monitoring. Data Analysis refers to those practices that, by querying one or more data collection systems, and by exploiting information contained in some kind of models, checks whether the observed operation of the system being monitored conforms to the expected behavior.

In the context of WP4, the phenomena to be monitored concern either stand-alone services or workflows, which may use external services or a choreographed set of services. The type of data that is collected must support the analysis of functional and extra-functional (QoS) properties of services and service compositions.

The overall architecture of the monitoring system is made up of two main components: one component is meant to perform continuous monitoring of a set of services, by analyzing relevant extra-functional properties; another component, whose behavior is triggered by events fired by services - i.e. external services invocations - is supposed to check the behavior of invoked services against a certain specification.

Both components provide a high degree of configurability, since users can switch the monitoring of each single service on and off, to dynamically tune the monitored parameters and attach priorities to them.

The rest of this chapter is structured as follows: Section 6.2 presents the component that performs monitoring of functional properties; Sections 6.3 and 6.4 propose two complementary approaches for monitoring extra-functional properties; finally, section 6.5 summarizes the contribution of the chapter.

6.2 Dynamic Monitoring of WS Composition

One of the components of the monitoring system will be in charge of monitoring the behavior of web service compositions; although in the following paragraphs we assume that service compositions are described in terms of services orchestration using BPEL, the model at the basis of our monitoring approach can be adapted to deal with compositions described using choreography-based languages.

The idea on which our approach is based is to provide a runtime checking of the behavioral conformance of a service against its specification. A service can be specified in two ways:

- in the case of stateless services, by using a Design by Contract [80] approach, i.e. specifying pre- and post- conditions (hence after also called contracts) for service invocations;
- in the case of conversational services, by using a language, inspired by algebraic specifications [8][20][50], which allows to abstract the state of the service and to implicitly specify it by defining equivalences among sequences of operations.

An example of the former case is the specification of a condition over an input parameter or the return value of the service. An example of the latter is the specification of a service, such as a shopping cart, in terms of its constructors, observers and derived operations, and predicates over operations sequences.

The architecture of the reactive monitoring is based on the dynamic aspectization of the BPEL engine executing monitored service compositions, using AspectJ [63] as Aspect Oriented Programming language [64]. In this way, we can add monitoring facilities to an already existing application, such as the open source ActiveBPEL[7] execution engine, exploiting the benefits of aspect-oriented programming.

By using an AOP-enabled programming language, we can keep business logic and monitoring logic separate, guaranteeing good code modularization. Moreover this architectural choice allows users to execute monitoring in a transparent - i.e. without modifying the structure of the process - and dynamic way, i.e. by inserting and enabling/disabling assertions at runtime on the basis of performance needs of the monitored process.

This represents an alternative to proxy-based solutions, such as the one proposed in [12], which tries to remove the architectural bottleneck related to a central, unique, proxy. Furthermore, it eliminates the need to produce a second BPEL process, obtained by weaving the original BPEL process with the monitoring rules.

In PLASTIC, our solution extends the standard implementation of the ActiveBPEL engine with four additional components, shown in Figure 29:

- Main Interceptor: it intercepts and modifies the execution of a process within the engine, at some *pointcuts*, using aspect-oriented programming;
- Rules registry: it contains the assertions to monitor;
- Monitor: monitoring rules verifier;
- GUI Interceptor: utility to set up a wizard for defining and installing monitoring rules.

[7] http://www.activebpel.org

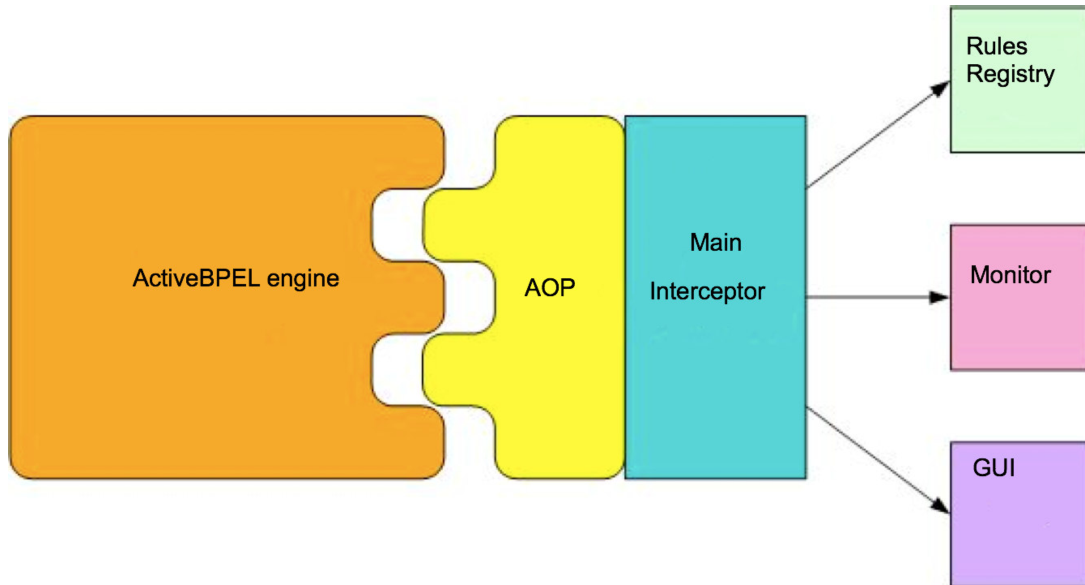


Figure 29: Monitoring embedded within the BPEL engine

Main Interceptor is obviously the component that plays a major role in our solution: it allows defining monitor-related *pointcuts* and *advices* in AspectJ. As stated before, we assume that pointcuts correspond to the activities in the BPEL process that interact with external services; thus we suppose the local workflow of the process to be correct and interactions with the external world to be potential causes of anomalies. With this assumption, `receive`, `invoke` and `pick` are the only activities to which you can attach monitoring rules.

Our approach defines pointcuts for several events of the engine:

- Engine start/stop
- Process construction/destruction
- Activity execution: this is the most important pointcut in order to perform monitoring, since the advices associated to it are in charge of verifying the specification, by invoking the monitor rule verifier (the Monitor).

Rules can be defined using a graphic tool, which allows to annotate the activities of a process with assertions and to attach a specification to an invoked service. Analogously to the Main Interceptor, this component exploits AOP techniques to extend the functionalities offered by the management tool bundled with ActiveBPEL.

Furthermore, users can specify basic QoS parameters (e.g. response time) that the engine can collect while performing monitoring activities, and constraints relative to the context in which the service will be executing, to perform context-aware monitoring.

Open issues that will be addressed by a further development of the approach presented above, concern the specification language for describing the behavior of conversational services, and the extension of the framework to service compositions described as choreographies.

6.3 SLA Monitoring Based on Formal Language

As the complexity of inter-organizational co-operations increases, so does the likelihood of parties not behaving as expected. A monitoring framework for SLAs is thus required if service guarantees need to be provided to the end users of B3G services.

There is a growing industrial interest in formal languages for SLAs and related analysis techniques, following the recent trends in service outsourcing. We propose here the following methodology for on-line monitoring of SLAs:

1. Express SLAs using an extended version of TCTL, a logic to reason about time and time intervals.
2. When an SLA is encoded as a TCTL formula, define its corresponding timed automaton.
3. Reduce the problem of online monitoring SLAs to the problem of checking whether or not the actual execution of the system is a (timed) word accepted by the automaton.

The remaining of this section presents the detail of this methodology.

6.3.1 Preliminaries

6.3.1.1 Timed Automata

A "standard" automaton is $A=(\Sigma,Q,Q^0,\delta,F)$. For instance, see Figure 30 where $\Sigma=\{a,b\}$, $Q=\{0,1\}$, $Q^0=\{0\}$, $F=\{1\}$, δ as depicted.

Automata recognize languages. For instance, L(A) includes the words {a,ab,abb,abaaaaabaab,…}.

A time sequence is a sequence of real numbers $\tau=\tau_1\tau_2\ldots$ s.t. $\tau_i>\tau_{i-1}$ and the sequence is non-Zeno. A timed word is a pair (σ,τ) where σ is a standard word and τ is a time sequence, e.g., {(aab…),(0.1,0.3,1.2,…)}.

Timed automata [24] accept timed words, i.e. they recognize timed languages. For instance, consider the timed automaton TA1 in Figure 31 $L(TA1) = \{ (((abcd)^\omega, \tau) \mid (\tau_{4j+3} < \tau_{4j+1} + 1) \wedge (\tau_{4j+4} > \tau_{4j+2} + 2)\}$.

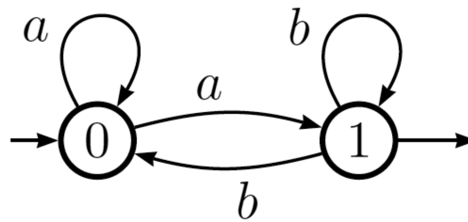


Figure 30: A simple automaton

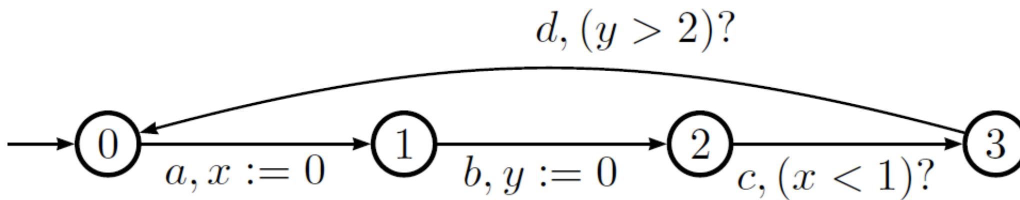


Figure 31: A timed automaton TA1

6.3.1.2 TCTL, a logic for real time

TCTL [3] is an extension of CTL which allows to reason about durations.

(Syntax of TCTL). Let AP be a set of atomic propositions; let I be an interval in $\Re$; let $<$ denote any of the binary relations $<,\leq,>,\geq,=$. TCTL formulae are defined inductively by:

$$\varphi ::= p \mid \neg\varphi \mid \varphi\vee\psi \mid E[\varphi\ U_{<c}\psi] \mid A[\varphi\ U_{<c}\psi]$$

where p is an atomic proposition and c is a real number.

The remaining temporal operators are derived in a standard way: $EF_c\varphi \equiv E[\text{ true } U_{<c}\varphi]$, $AF_{<c}\varphi \equiv A[\text{ true } U_{<c}\varphi]$, $EG_{<c}\varphi \equiv \neg AF_{<c}\neg\varphi$, and $AG_{<c}\varphi \equiv \neg EF_{<c}\neg\varphi$.

The standard semantics of TCTL is given by means of Timed Automata. We refer to [3] for more details.

Efficient algorithms and tools have been developed for model checking TCTL, see for instance [29][30][13][90].

6.3.2 Counting events and real time constraints

The problem: TCTL is not good at counting events. For instance, “there are 10 requests in a given minute”: there is no easy way to express this in TCTL. Moreover, counting events in an interval does not appear as a specification pattern, neither “qualitative” [35] nor “quantitative” [65].

We propose the following extension to the syntax of TCTL: EC(p,n,t), where p is a propositional variable, $n \subseteq N$ is a positive natural number, and $t \subseteq \Re$ is a real number. EC(p,n,t) is read as: there exists a path such that p happens at least n times in the next t time units along the path. The semantics for this operator is easily defined using Timed Automata, similarly to TCTL.

Figure 32 depicts an automaton TA2 which accepts runs for EC:(p,3,5), i.e., runs where p happens 3 times within 5 times unit.

In general, an automaton accepting EC(p, n, t) has n+2 states.

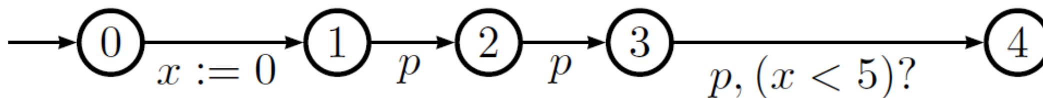


Figure 32: A timed automaton TA2

6.3.2.1 Verification of traces

We do not need the full machinery of model checking a generic temporal formula in a generic model for TCTL, but we can take advantage of model checking techniques in a number of ways. Indeed, the problem of monitoring a requirement in simpler than model checking: there is a single trace (the execution trace), and one or more temporal formulae that have to be verified along that trace. We exploit this fact in two ways:

1. Verification can be performed “on the fly” (i.e., while the trace is being built: this is a standard optimization technique for model checking, for instance see [87]).

2. The most common model checking methodology for TCTL involves the construction of equivalence classes for clock evaluations and their (possibly symbolic) representation. Instead of using this technique, we suggest the use of the model checking technique used in the SPIN model checker [53]: we test that $E \in L(\varphi)$, where E is a string corresponding to the sequence of events, and $L(\varphi)$ is the language recognized by the automaton encoding a temporal formula φ (the observation required to hold); see [23] and references therein for an introduction to automata-based model checking. Notice that this technique has been applied to the logic LTL only, but it can be easily extended to TCTL.

More in detail, we define E and $L(\varphi)$ as follows:

- E: This string is generated by recording the events (each event is an element of the alphabet) and the time of their occurrence. This generates the timed word E.
- $L(\varphi)$: automata for TCTL formulae can be built in a standard way. Automata for formulae of the form EC(p,n,t) are built similarly to Figure 32: n+2 states are introduced, relevant transitions are labeled with p, and the clock constraints are defined using t.

Verification can be performed at this point on-the-fly: every time a new event is generated, i.e., every time E changes, we need to check whether or not E is a word permitted by the automaton $L(\varphi)$ (notice: if more formulae need to be verified, the verification is performed for each formula).

6.3.3 From SLAng to real-time verification

In this section we present an example of reduction of an SLA to timed automata.

Part of an SLA in SLAng is reported in Figure 33 essentially, this is a reliability clause for a service provider, with a required 90% reliability in time windows of 1 min. Additionally, the client is subject to an `inputThroughputClause`: no more than 10 requests per minute can be submitted to the service provider.

```
failureModeDefinition = {
  // A failure if latency > 5s
  [...]
  maximumLatency = ::types::Duration(5, S)
  [...]
}

[...]

inputThroughputClause = {
   FailureModeDefinition[f1]() {
      // Max 10 requests in 1 min
      [...]
      inputWindow = ::types::Duration(1, min)
      inputConcurrency = 10;
      [...]
   }
}

reliabilityClause = {
   // A failure is the thing defined above
   failureModes = { FailureModeDefinition[f1] }
   [...]
   // Reliability requested: 90% over 1 min intervals.
   reliability = ::types::Percentage(0.9)
   window = ::types::Duration(1, min)
   [...]
}
```

Figure 33: HUTN code for SLAng (excerpts)

We start by analyzing `inputThroughputClause`: this requirement imposes that no more than 10 requests can be performed in 1 minute. The requirement corresponds to (the negation of) the following formula: EC(request,10,60). This formula, in turn, can be represented using an automaton similar to the one in Figure 32, against which execution traces can be validated.

`failureModeDefinition` is a standard bounded response pattern [65]: a response should follow a request within a certain time. This is translated into the TCTL formula AG:(request $\Rightarrow AF_{<5s}$response). We assume that when this formula is violated, the client enters a state where a new proposition “fm” holds.

The reliability requirement imposes that, in any given minute, fm cannot occur more than 90% of the times. As a client cannot submit more than 10 requests per minute[8], reliability can be expressed using the automaton in Figure 34. Notice that this automaton accepts all the sequences of events in which two failure modes (fm) occur within 60 seconds: an execution trace satisfying this automaton would be a violation of the reliability requirement.

In this example, it seems natural to install the monitors for failure modes and for reliability on the client, and on the server for `inputThroughputClause`. A possible implementation is presented in the next section.

---

[8] The definition of EC does not allow for the expression of “derived” quantities (such as percentage, fractions of executions, etc.). Thus, we have to establish a bound of the form “number of events per unit of time”. However, this limit seems to make sense in a number of cases, while a simple percentage may cause problems. For instance, 90% reliability in a minute, without an `inputThroughputClause`, permits an unbounded number of requests in the time unit.

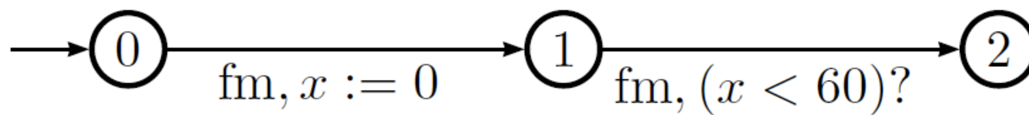


Figure 34: Timed automaton for the reliability clause

6.3.4 A methodology for on-line monitoring

A possible implementation of an automata-based monitor is presented in Figure 35. When an event is detected at an interface, it is processed by a dispatcher. The dispatcher holds the list of SLAs; if an event is relevant to some of the existing SLAs (expressed as automata), an SLA checker is created and the event is passed to the newly created checker. Also, if the event is relevant to an existing SLA checker, it is passed to it. Additionally, the Dispatcher delivers the event to a Log manager (see below).

The SLA checkers (in circles) implement the automata-based verification of traces of events. See http://www.cs.auc.dk/~behrmann/utap/libutap-0.90.tar.gz for a library for automata manipulation. SLA checkers place lock on logs. When the SLA checker terminates (either because no violation was detected in the execution, or because a violation occurred), the lock on the log is released.

The log manager keeps track of the locks and stores the relevant log in a “live” log. When all the locks are released, the manager stores the “unlocked” part of log in a more suitable/compressed form (archived log).

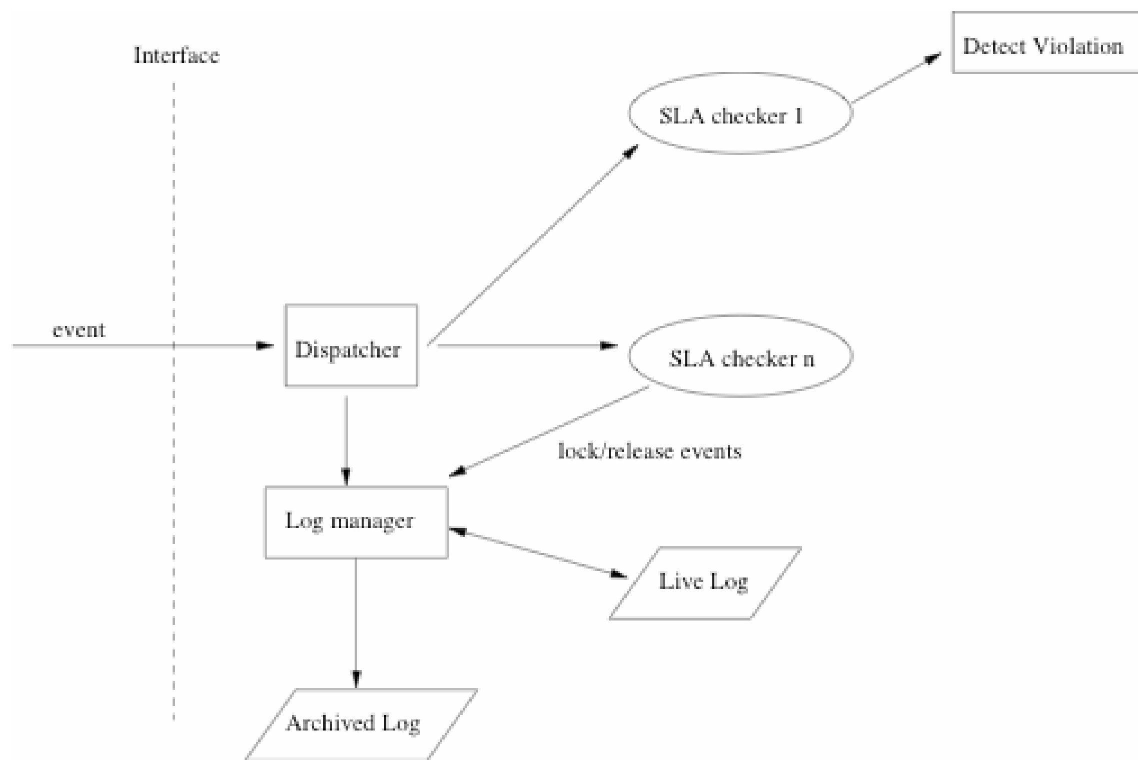


Figure 35: A proposed implementation of an automata-based monitor.

6.4 Qos Monitoring

Being targeted at heterogeneous hardware platforms, PLASTIC could be deployed on nodes with reduced capacity, for which prudent resource management is key. In real-life scenarios, a naive approach to monitoring can introduce considerable QoS degradation, influencing those very parameters that one aims to measure.

Several potential solutions to overcome this issue could be considered. For example, one could transfer the logs to a different dedicated machine, where they would be analyzed offline. While this could apparently alleviate the problem, in fact it would just shift the overhead from the storage to the network resources, potentially producing similar negative effects to those that one wanted to avoid. As an alternative, one could compute a condensed representation of the logs to minimize the storage and transfer requirements, but this would just shift the load towards the CPU resource again. It is then evident that, in order to minimize the negative influence of monitoring on the observed quantities, a different technique must be devised.

The basic principle we propose is that *the amount of system resources assigned to monitoring must be adjusted dynamically, based on the measured load of the underlying hardware platform*. More specifically, when operating system parameters (e.g., CPU, memory, bandwidth) reach certain upper levels, the monitoring of certain services should be reduced or suspended, based on suitable policy.

We now outline our proposal for a monitoring framework designed following this general principle. We identify the main components of the framework and define their high-level responsibilities and operation, pinpointing, where appropriate, interesting research problems that we would like to investigate in the next stages of the project.

6.4.1 The Monitoring Framework

The framework is composed of the following parts: the Platform Observer, the *Data Collection Controller*, the *Dynamic Loggers,* the *SLA Parser* and the *SLA checker*. In the next sections we will describe the operation and high-level responsibilities of each of them.

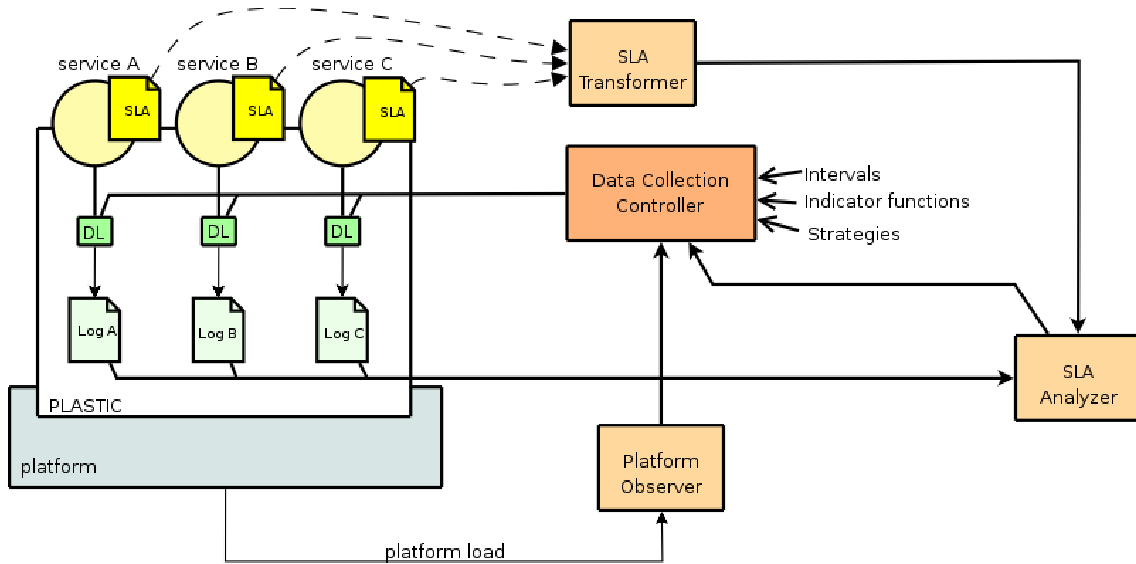


Figure 36: Architecture of the proposed QoS monitoring framework

6.4.2 Dynamic Loggers

The collection of data from service operation needs to be adjustable at run-time. A dedicated logger is assigned to each service for which an SLA must be verified. Loggers expose an interface whereby their behavior can be dynamically configured. More precisely it should be possible to:

1. set the level of detail at which the logging must be done (for instance, to specify the frequency at which events should be recorded, i.e. sampled);
2. indicate what type of events must be observed and recorded in the logs;

Table 6: Typical metrics captured by logs

Event	**information**
receive service invocation	time-stamp, interaction identifier
send reply to service invocation service	time-stamp, interaction identifier
context changes	time-stamp, specification of the new context
exception caught by the middleware	time-stamp, exception type, additional info

The loggers capture information like that reported in Table 6. It can be used to verify the QoS of services that are running on a given node and that act as servers in the considered interactions. In practice, a service S can in turn require other services (Z in Figure 37 below) to carry out its task and thus, it can play, on one side, the role of a server and, on the other side, the role of a client. Observations of the interactions where S is the client and that are made from the node where S is deployed unavoidably include the effects (increased latency, higher failure rate, *burstiness*) of the communication network. These effects need to be taken into proper account collecting additional logging information, such as:

- the time-stamp, the called service and the interaction identifier for each invocation addressed by the service to another one;
- the time-stamp, the called service and the interaction identifier for each reply received from other services.

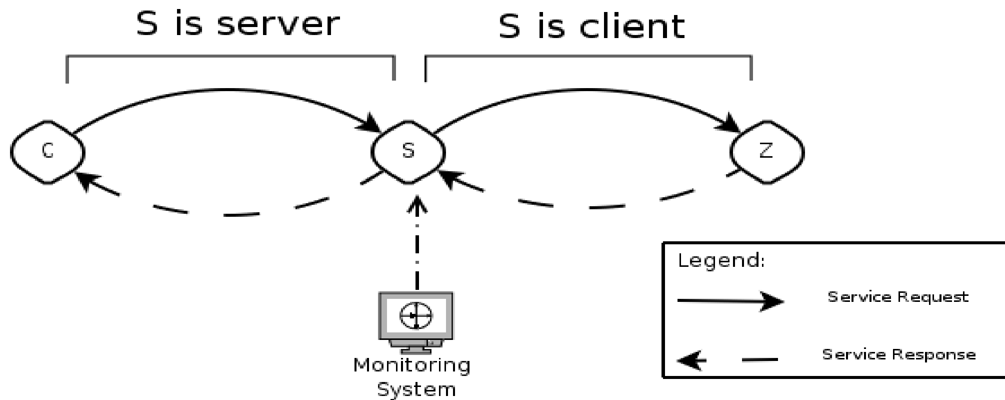


Figure 37: In-bound and out-bound service requests

6.4.3 Platform Observer

The Platform Observer gathers instantaneous measures of platform level indexes, including:

- CPU utilization rate
- memory occupation
- utilization rate of network interfaces
- disk activity

Also, the Observer continuously computes the values of a set of indicator functions, whose parameters are platform indexes like those exemplified above, and provides these values to the Data Collection Controller.

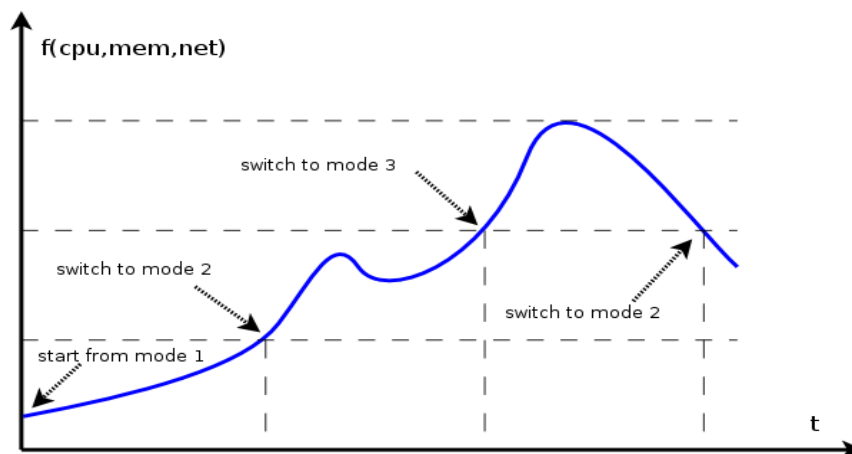


Figure 38: Example of logging mode switching based of an indicator function

6.4.4 Data Collection Controller

The Controller is the core element of the framework. It gathers the instantaneous values of the indicator functions obtained from the Platform Observer and drives the operation of loggers. In order to do so, the co-domain of each indicator function is partitioned into two or more intervals, so that different operational conditions are identified based on the function's value. The *logging mode* of a logger can then be changed when the newly observed value of a suitable indicator function falls into a different interval than the preceding. Basic logging modes, which all loggers are capable to offer, are:

- *full logging* mode, in which all events are observed and as much information as is available about them is logged;
- *disabled* mode, in which nothing is logged. A logging mode is a specification of the type and level of detail of the information that must be logged. It is then the Controller's task to maintain a proper correspondence between observed indicator function values and logging modes.

The simplest control strategy is to partition the co-domain of an indicator function in just two intervals and associate them with the "full logging" and "no logging" modes respectively. Of course more sophisticated policies can be defined, and this is an interesting direction that may be worth exploring in future research. According to the mechanism sketched so far, the decision as to whether the logging mode of one logger should be changed or not is taken based on *one* indicator function and *independently* of what is being done to the other loggers. More sophisticated alternatives could be experimented with, which could take into account multiple indicator functions and which could coordinate mode switching of concurrent loggers.

Also, the definition of modes and of the control algorithm that associates modes to indicator functions is highly dependent both on the type of monitored services and on the clauses contained in the SLAs that are being checked. Conceivably, the knowledge of this information could be leveraged to realize more effective control policies. Furthermore, besides this "static" data, also information about the discovered violations - or lack thereof - could be considered as a useful input for the control algorithm.

6.4.5 SLA Parser

This component is responsible for parsing SLAs to express the information contained therein in a simpler format, which is based on elementary constraints that are readily understandable by the SLA Analyzer. By interposing this component between the SLAs and the Analyzer we are able to cope with the diversity of notations that might be used to describe SLAs, thus treating separately the two distinct concerns of their representation and their analysis. Support for additional SLA languages can thus be easily plugged into the framework, just by implementing suitable new parsers.

6.4.6 SLA Analyzer

The purpose of this component is to compare the information available from the logs with the constraints that are produced by the SLA Parser. Existing approaches and techniques can be exploited for the implementation of the SLA Analyzer. As an example, an approach called "performance assertion checking" is presented in [88], where the Pspec language is presented along with a prototype tool to check execution logs against assertions specified in Pspec. A similar idea, according to which the assertions are embedded in the code and evaluated at run-time, is presented in [109]. Another related work is presented in [69], which introduces an extension of the methodologies mentioned above, addressing the specific needs of performance testing for applications running on mobile devices.

6.5 Summary

This chapter has presented the approaches adopted to accomplish online validation of PLASTIC services. Validation is performed using monitoring techniques tailored for functional and extra-functional properties.

Further research effort will be put to explore the opportunity of integration of the three approaches presented. In particular, the two approaches for extra-functional monitoring described in sections 6.3 and 6.4 could be combined: the first proposes a specific approach to verify that a specified SLA is fulfilled, the second design an adaptive framework to make monitoring efficient; moreover, the framework described in section 6.2 could act as a data collector of QoS parameters for the extra-functional monitoring component.

7 Validation Framework Architecture

7.1 Introduction

In the previous chapters we have presented in detail the various approaches and tools under development within WP4. Earlier in Chapter 3 we have anticipated our vision of how all presented approaches fit within the comprehensive validation process of B3G services which we envisage. As already said, in PLASTIC WP4 our position is that we are not going to build one fixed monolithic environment; instead we will develop a set of technologies which can be used in independent manner or in different combinations, so giving rise to a flexible modular framework which the service developers or integrators can adapt to their exigencies.

In this chapter, we will now sum up the presented technologies and put them in context within an abstract specification of the PLASTIC validation framework architecture. The process we have outlined in Chapter 3 constitutes the means for collating the "architecture". Following the above vision, the architecture is not meant in terms of a design structure, but rather consists of an abstract framework unifying all approaches presented so far, organized along the three identified stages of off-line, on-line pre-publication and on-line live-usage.

The next section describes such a framework, and makes plans for the development of prototype implementations of the proposed solutions in the next future. Section 7.3 then discusses foreseen risks and mitigation actions. Conclusions of the chapter are summed up in Section 7.4.

7.2 Integrated Framework Architecture

As described in Chapter 3, the validation of B3G applications in Plastic is organized into three subsequent stages, which are shown in Figure 39. As already explained, and as also early illustrated in Figure 4 (p. 43) , off-line testing refers to validation activities conducted during development of the service, in a fake environment. On-line validation refers instead to testing a software deployed in its real environment; more in particular in the context of services, we have distinguished between validating the service behaviour before it is made available for general public invocation, or after, in which case the validation is a synonymous with monitoring.

With reference to Table 2 (p. 42), we now discuss the stakeholders who could be interested in each stage. In principle of course any stakeholder could be interested in conducting off-line validation, but the problem as extensively discussed in this document is the availability of the information necessary for building the reference model for testing. Therefore, the most likely case is that off-line testing is carried on by the service developer, and by the service integrator as well as regarding the integration testing of the composite service. In some measure also the service deployer could be entitled to carry on some off-line validation, mainly focused on testing the service integration within the environment in which it is deployed, but to this purpose our vision that the service information is enriched with semantic information of its behaviour, both functional and extra-functional, needs to be realized.

On-line pre-publication in our vision is of interest for the service deployer (in its vest of provider of the service), the directory service and, if the SUE has to conform to standardized specifications, also to the relevant standard body. They are motivated by different objectives: the service deployer wants to verify that the service behaves as expected; the directory service want to guarantee the integrity of published services; the standard body wants to check conformance of the implementation to the agreed standard specifications. But all of them share the concern of checking beforehand that the service provides guaranteed quality levels, by performing a sort of qualification test before granting publication.

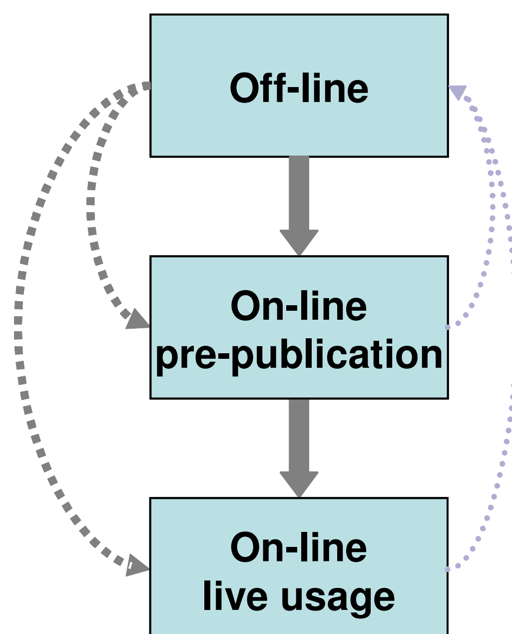


Figure 39: The three validation stages in PLASTIC

Finally on-line live usage is interesting directly to the service providers, since they may want to be able to log information about the usage of the services, and to promptly adopt adequate countermeasures in case anomalies or malfunctions arise; but also indirectly to the service users and to the directory service, who may want to monitor that the contractualized levels of quality are actually provided.

As shown in Figure 39, it is assumed that the three stages proceed in sequential order, since a service is first developed, then deployed, then used. There are clear relationships among the stages. In particular the results of the analyses conducted off-line may be used to guide the on-line validation activities, as shown by the dashed grey arrows on the left of the above figure. Moreover, the results from analysis during the on-line validation might provide a feedback to the service developer or to the service integrator, highlighting necessary or desirable evolutions for the services, and therefore might be used for off-line validation of successive enhanced versions of services, as hinted at by the light dashed arcs on the right.

Having made clear the context for the three introduced stages for validation in PLASTIC, we now describe the integrated framework in which all the approaches and tools introduced in this document fit together. The framework is illustrated in Figure 40. We provide first a general statement about the view of a validation framework for B3G applications, and then, marked by a check symbol, we point in particular at the proposed approaches within WP4. Even more specifically, since the time and the resources available within PLASTIC are limited, for each approach we provide an indication of what we are planning to implement in the six months remaining to the next deliverable which is “Test framework, prototype implementation release 1.0”, foreseen at month 18th. Indeed, as such information is extensively discussed in the dedicated chapters, here we just make an integrated overview and refer to the specific sections where the approaches are described.

Concerning off-line validation, the framework is illustrated in Figure 40. part a). First of all, common to any approach is the need for a realistic testbed within which the service can be

executed. The development of the testbed is crucial for off-line validation success, but it is behind the scope of WP4.

In PLASTIC the integration and evaluation of all proposed technologies is foreseen within Work Package 5, which is specifically devoted to assess the integrated PLASTIC platform through the development of mobile e-services. In particular four scenarios have been earlier identified, namely e-health, e-business, e-learning and e-voting. It is obvious that a specialized testbed for each of the scenarios in which the validation framework will be experimented is necessary. This is a prerequisite to the future evaluation of the proposed test framework.

Above the testbed, it is necessary to provide means for facilitating and automating the experimentation. Indeed, as extensively described in Section 4.2, testing of distributed heterogeneous systems, such as the PLASTIC applications, is much more than simple deployment and execution. It can involve complex activities such as experimental design, workload generation, data collection, data analysis, and overall experiment management. So we need a comprehensive framework to help software engineers manage and automate experiments with highly distributed systems performed on distributed testbeds.

In WP4 we will adapt and use for automating distributed experimentation the distributed exprimentation platform described in Section 4.2. This platform automates a three-phase process: i) a workload is generated, ii) the trial is run; iii) post-processing of the collected data is performed. A running version of Weevil is already available, in the next six months we will work on adapting the platform to the B3G context, and on permitting integration of the platform with the foreseen test approaches.

As said, the validation will include both functional and extra-functional testing. Different approaches could be considered to this purpose, many of which have been surveyed in Chapter 2.

In WP4, for functional testing we have concentrated our efforts toward Model-based testing approaches, since these are widely recognized as a promising solution for rigorous systematic testing. Given the characteristics of B3G applications, we have worked to customize the existing well-known ioco-based testing methodology [103] to deal with the notions of ports and conversations proper of Service-oriented applications. We have assumed that the static aspects of the WSDL are extended with the dynamic, functional aspect of data enriched conversation descriptions. Such a conversation consists of messages (including their parameter data) as they are exchanged at WS ports. The goal is to automatically test WS by generating inputs and giving verdicts to observed outputs based on the SSM specification. The general approach is depicted in Section 4.4, and it lays down a series of difficult research challenges. In the next six months we will release a first release implementation of the AMBITION tool which supports the automated testing of a single WS. We are also working in parallel to develop the JESSI tool that provides graphical editing capabilities for the SSM specification.

Concerning the off-line testing of QoS properties, in general we need to be able to reproduce in the testing laboratory the behaviour of the SUE even for its extra-functional properties.

In WP4 we have defined a possible approach, called Puppet, which addresses the automatic generation of an infrastructure to be used as a test harness for pre-deployment QoS evaluation of a service. The generation is performed in two different phases: first the generation of the stubs simulating the extra-functional behavior of the services in the composition; second the composition of the service implementation

with the services with which it will interact. In WP4 we will build a proof-of-concept tool for Puppet: the choice of the strategy to be used for selecting the extra-functional test cases for invoking Puppet is still an open problem.

Since B3G applications consist generally of large-scale networked service implementations, whose execution cost may be high, the usage of simulation-based execution of test cases could be useful.

In WP4 we will also make available an approach to Simulation-based testing, which can be used in combination with both functional and QoS validation approaches in order to choose the most effective adequate test suite or the most effective adequacy criterion. The adopted approach is described in Section 4.3. It has already undergone some experimentation, and in PLASTIC we will investigate the possibility of its combination with the proposed off-line approaches. In the figure, as an example, we show the case that the MBT approach produces an Abstract Test Suite (ATS), and that this is given in input to the Simulation-based test approach, which can optimize the suite. One basic issue to the applicability of simulation-based testing ideas is that the approach requires the availability of a discrete event simulation in executable form of the service under test.

We now consider that the service application is deployed on the server node. Before it is published we in general assume an admission testing session, aimed at ascertaining that the service provides adequate behaviour, as explained in Chapter 5. This is a sort of quality certification, which assumes that the service world is semi-open: not all services are admitted. The outcome of this stage is a YES or NO verdict which allows or denies service publication. As illustrated in Part b of Figure 40, in general the qualification check may concern both functional and extra–functional properties. Clearly, such stage poses many complex challenges, both technical and administrative, which hinder its full realization within PLASTIC.

In WP4 we are targeting the functional admission testing, which we have called Audition. This poses many challenges, as described in Section 5.2.5, in particular for taking into account both that the service under test adequately fulfils the incoming invocations, as well as that it also in turn makes correct invocations to other services. For the moment only the first case, called IBC, will be realized in WP4. The test cases to be launched during audition could be the same test cases derived in the off-line analysis from the SSM model: in this way an integration is realized between off-line and on-line audition.

Finally, we deem that for validation of B3G services the dynamic monitoring of service runtime behaviour is an indispensable approach to tackle with mobility and context-awareness. In general different, more or less invasive, approaches to monitoring can be put in place. In passive monitoring, validation just consists of observing the behavior, by registering the execution traces and/or appropriate parameter values, while the system executes; this is useful to timely detect possible problems or malfunctions, and for profiling purposes, e.g. for accounting: By reactive monitoring, we mean the possibility to intervene and drive the system behaviour, based on the analysis results of the monitored behaviour. Finally, proactive monitoring is meant as stimulating the system with a set of test cases, as is done in off-line testing. The latter clearly poses problems of side-effect and of controllability.

In WP4 we will target passive monitoring, which is the basic, less demanding approach, although also less effective in terms of guaranteeing the application QoS. We will probably not be able to fully implement reactive monitoring, but at least partially we are investigating the capability to make the monitoring function adaptive, not to negatively affect system performance. We are currently developing functional

monitoring as described in Section 6.2, using Aspect-oriented technology; we will show how QoS can be monitored using a formal description of the service level agreements, by means of temporal logic technologies. Besides, we will work at a framework for making the QoS monitoring more efficient, using an adaptive monitoring framework, as described in Section 6.4.

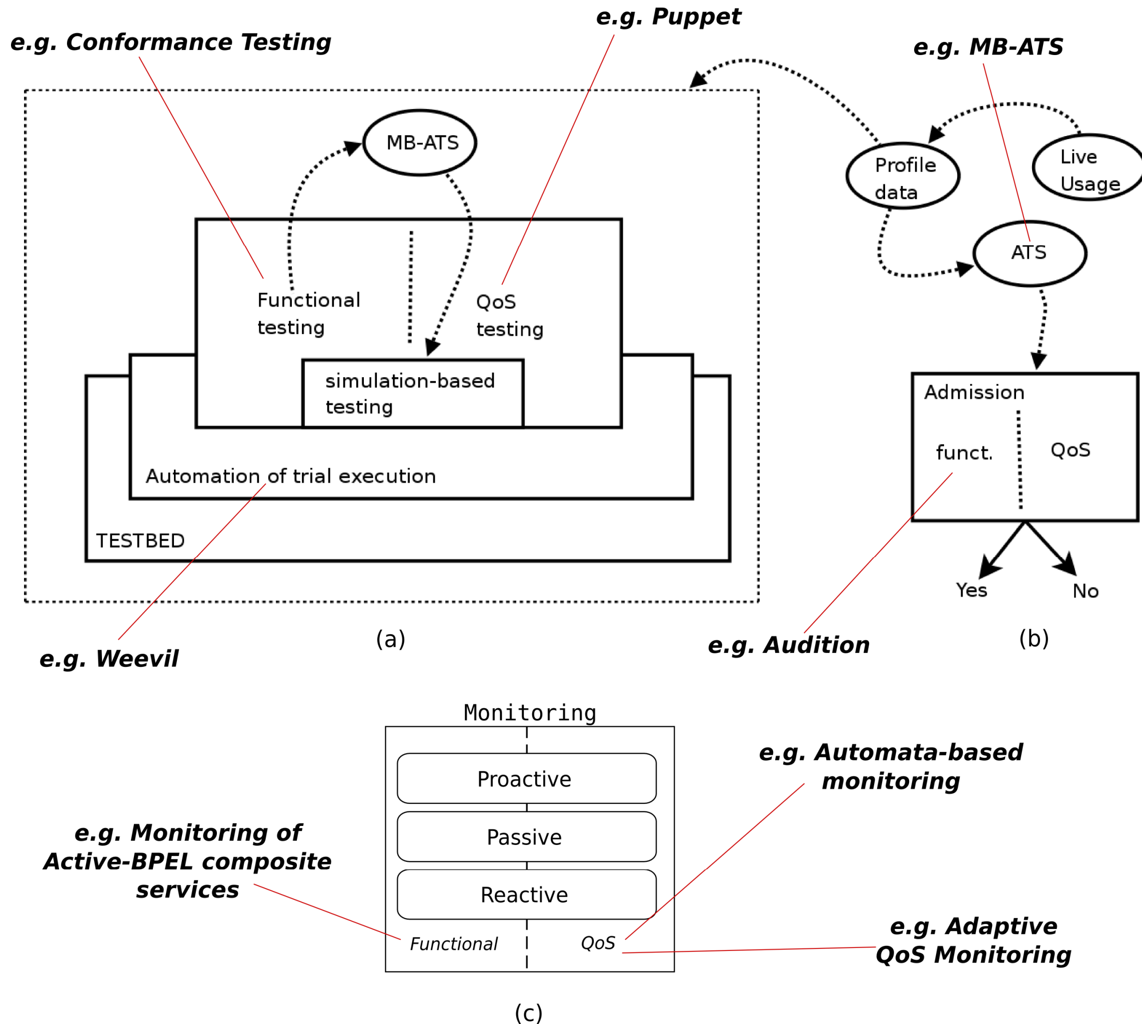


Figure 40: Integrated validation framework

7.3 Risks and Mitigation Actions

As the previous section shows, the foreseen architecture for the WP4 validation methodology includes several different technologies and tools. While it is certainly positive and agreeable that the validation of the PLASTIC targeted B3G applications is pursued by a blend of approaches, the wide spectrum of technologies under development poses the risk that efforts are fragmented and eventually the hoped-for results are not achieved.

Indeed, we believe that an ambitious research project as PLASTIC, in terms of general definition of a methodology for validation should foresee a framework as comprehensive as possible, and this is what we have done. Then what we plan to realize is a proof-of-concept tool for each of the proposed techniques, that can be used for the evaluation stage in the second part of the project. Therefore, for each of the adopted technologies, we have distinguished between: i) defining the technology in the most general and comprehensive way, for the purpose of outlining a generally applicable validation architecture; and ii) implementing a specific instance of the technology to a restricted scenario, so that we can get also a functioning proof-of-concept tool. The first mitigation action has been hence to identify for each proposed approach a minimum instance that can realistically be implemented within the course of the project. The specific assumptions and restrictions taken for each technology have been amply discussed in the sections devoted to the specifications of the technology itself.

Another mitigation action that we have taken is to adopt, wherever possible, technologies and tools which were already existing and adapt them to the exigencies of the project, rather than developing new tools from scratch. In this way, we have already some evidence of the feasibility of the technology.

7.4 Summary

In this chapter we have unified under a common view, also illustrated in Figure 40, the various technologies presented in Chapters 4, 5 and 5. We have also explained here that we distinguish between outlining a generally valid framework for the validation of B3G applications, embracing all the three stages identified in Figure 39, with a variety of techniques, and developing a proof-of-concept set of tools for carrying on an empiric evaluation of the proposed approaches on the PLASTIC testbeds. For the latter, we have identified some instances of the technologies which are produced making adequate limited assumptions.

8 Conclusions

This deliverable has covered broadly the topic of validating PLASTIC B3G applications. We have first laid down the context and goals of WP4, and have provided a wide and uptodate overview of the current state of art in the related topics. Then, we have provided an extensive specification of the technologies that we will adopt for validation of PLASTIC services, and finally we have discussed how all these technologies fit together into the test framework architecture. Actually, as we have discussed in the previous chapter, the architecture is not really conceived as a structure of components, but rather the tools under development are glued together by the vision of a three stage process, in which the different prospected techniques can be combined.

The task of validating B3G applications poses many difficult challenges. Whilst this document has provided an ambitious framework of attractive approaches to put in place, we are aware of the many remaining issues to be solved, some not even considered here. In particular, the most pressing issue is to tackle context awareness and mobility, which we do not deal with in off-line testing, but only approach by on-line monitoring of live-usage behaviour. The reason for this is that B3G applications operate in very heterogeneous and dynamic contexts, whose reproduction in laboratory for the purpose of testing would be too costly.

It is important to emphasize that even though we have always considered feasibility and viability of the proposed techniques, even the most effective validation technique is never obtained for free, but has an associated cost. For each of the proposed approaches there are requirements to their applicability. Model-based testing, specified in Section 4.4, is a black-box approach, but requires that the service developer releases a specification of service behaviour. Simulation-based testing (see Section 4.3) can be used to improve the test suite effectiveness, but can only be applied when a discrete event simulation is available. On-line pre-publication testing, such as the Audition approach in Chapter 5, is an innovative idea to make the deployment environment of a service more trustable, yet it is viable only if the idea of a service quality certification is accepted. On-line live usage monitoring is generally acknowledged as an important component of modern dynamic systems development, accompanying their real world usage, for profiling and validation purposes, but it requires enhanced logging and analysis capabilities incorporated within the middleware.

Therefore, an important point is that the validation of the functional and QoS adequacy of B3G services cannot be conceived as an add-on feature which is plugged in from outside their development process, but rather is a stringent requirement which must be accepted, sought and supported by all service stakeholders. Indeed, in the joint work conducted within the PLASTIC project we have experimented the many strict relations existing among what has been proposed in WP4, and the activities ongoing in parallel within WP2 and WP3.

While this first WP4 deliverable addressed the specification of the adopted techniques, the next one will deal with the implementation of the techniques into prototype tools and successively, in WP5, their experimentation with the PLASTIC adopted testbeds.

9 References

[1] R. Adams. Take command: The m4 macro package. Linux J., 2002(96):6, Apr. 2002.

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

[3] R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real-time. Information and Computation, 104(1):2–34, 1993.

[4] R. Alur and D. Dill. A theory of Timed Automata. Theoretical Computer Science, 126(2):183–235, 1994.

[5] Apache Logging Services. LOG4J Introduction. http://logging.apache.org/log4j/docs/

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

[7] ASG - Adaptive Services Grid. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. http://asg-platform.org/

[8] E. Astesiano, H.-J. Kreowsky and B. Krieg-Brckner, editors. Algebraic Foundations of Systems Specification. IFIP State-of-the-Art reports. Sprinter, 1999.

[9] M. Autili, V. Cortellessa, A. Di Marco, P. Inverardi. A conceptual model for adaptable context-aware services. In *Proc. of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pp. 15-33. 2006.

[10] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, C. Sciafanell "Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step" in Proceedings of Web Services and Formal Methods, LNCS 3670, pp. 257-271.

[11] L. Baresi, C. Ghezzi and S. Guinea. Smart Monitors for Composed Services. In Proceedings of the 2nd International Conference on Service Oriented Computing, 2004.

[12] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. Proceedings of ICSOC05, 3rd International Conference On Service Oriented Computing. 2005.

[13] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of Uppaal. In Proceedings of the International Workshop on Software Tools for Technology Transfer, 1998.

[14] A. Bertolino, A. Bonivento, G. De Angelis, and A. Sangiovanni Vincentelli. Modeling and Early Performance Estimation for Network Processor Applications. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006)*. LNCS 4199, pp. 753-767. Springer–Verlag, 2006.

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

[16] A. Bertolino, W. Emmerich, P. Inverardi, and V. Issarny. Softure: Adaptable, Reliable and Performing Software for the Future***,*** Proc. FRCSSat ETAPS 2006, March 26th - Apr 2nd, Vienna,Austria.

[17] A. Bertolino, L. Frantzen, A. Polini, J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols, in J. Stafford, R. Reussner, C. Szyperski (Eds.), *Architecting Systems with Trustworthy Components*, special issue in Springer - LNCS n.3938.

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

[19] A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In Proceedings of the 31st EUROMICRO International Conference on Software Engineering and Advanced Applications, pages 134142, Porto, Portugal, August 30th - September 3rd 2005.

[20] M. Bidoit and P. Mosses. CASL user manual. Volume 2900 of Lectures Notes in Computer Science. Springer. 2004.

[21] M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, editors. Model-based Testing of Reactive Systems: Advanced Lectures, volume 3472 of Lecture Notes in Computer Science. Springer, 2005.

[22] G.Canfora, M. Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. In IT Professional March-April 2006 pp.10-18

[23] R. H. Carver and K.-C. Tai, Replay and Testing for Concurrent Programs, IEEE Software, Vol.8 (2), Mar. 1991, 66-74.

[24] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. The MIT Press, Cambridge, Massachusetts, 1999.

[25] COMET - Converged Messaging Technology. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. https://www.comet-consortium.org/

[26] C. Courbis and A. Finkelstein. Towards aspect weaving applications, Proceedings of International Conference on Software Engineering, St. Louis, 2005.

[27] F. Curbera, M. J. Duftler, R. Khalaf, W. A. Nagy, N. Mukhi, and S. Weerawaran. Colombo: lightweight middleware for service-oriented computing. IBM Syst. J., 44(4):799-820, 2005.

[28] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-directed requirements acquisition. Science of Computer Programming, vol. 20, pages 3-50, 1993.

[29] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Hybrid Systems III, volume 1066 of LNCS, pages 208–219. Springer-Verlag, 1995.

[30] P. Dembinski, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Wozna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), volume 2619 of LNCS, pages 278–283. Springer-Verlag, 2003.

[31] R. A. De Millo, R.J. Lipton, and F.G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (1978), no. 4, 34–41.

[32] G. Denaro, A. Polini, and W. Emmerich, Early performance testing of distributed software applications, Proceedings of the fourth international workshop on Software and performance, 2004, pp. 94–103.

[33] E. J. Dowling, Testing distributed ada programs, Proceedings of the conference on Tri-Ada '89, 1989, pp. 517–527.

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

[35] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98), pages 7–15, New York, 1998. ACM Press.

[36] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby and Shmuel Ur, Multithreaded Java program test generation, IBM Systems Journal, Vol. 41 (1), 2002, 111-125.

[37] T. Erl, Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services. Prentice Hall PTR, 2004.

[38] O. Ezenwoye and S. Masoud Sadjadi. Enabling robustness in existing BPEL processes. In Proceedings of the 8th International Conference on Enterprise Information Systems, Paphos, May 2006.

[39] N. Fenton and S. Pfleeger. Software Metrics: A Rigorous and Practical Approach. PWS Publishing Company, 1998.

[40] H. Foster, S. Uchitel, J. Magee, J. Kramer. Model-based verification of web services compositions. In Proc. ASE2003, pages 152{161, Oct., 6-10 2003. Montreal, Canada.

[41] L. Frantzen. The PLASTIC Process of Specifying and Testing Services, Version: September 11, 2006. See PLASTIC website (Work Packages -> WP4 Kick Off).

[42] L. Frantzen, J. Tretmans and R. d. Vries. Towards Model-Based Testing of Web Services. In Andrea Polini, editor. Proceedings of International Workshop on Web Services - Modeling and Testing (WS-MaTe2006), pages 67-82, 2006.

[43] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A symbolic framework for model-based testing. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, FATES/RV 2006, number 4262 in LNCS, pages 40–54. Springer-Verlag, 2006.

[44] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, FATES 2004 , number 3395 in LNCS, pages 1–15. Springer-Verlag, 2005.

[45] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In Proc. of WWW2004, May, 17-22 2004. New York, New York, USA, pp. 621-630.

[46] S. Ghosh, N. Bawa, G. Craig, and K. Kalgaonkar, A test management and software visualization framework for heterogeneous distributed applications, HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering Washington, DC, USA), 2001, p. 0106.

[47] S. Ghosh, N. Bawa, S. Goel, and Y. Raghu Reddy, Validating run-time interactions in distributed java applications, ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (Washington, DC, USA), 2002, p. 7.

[48] S. Ghosh and A. Mathur, Issues in testing distributed component-based systems, Proceedings of the First International ICSE Workshop Testing Distributed Component-based Systems, May 1999.

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

[50] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Current Trends in Programming Methodology, volume IV: Data Structuring, pages 80-149. Prentice-Hall. 1978

[51] J. Grundy, Y. Cai, and A. Liu, Softarch/mte: Generating distributed system test-beds from high-level software architecture descriptions, Automated Software Eng. 12 (2005), no. 1, 5–39.

[52] R. Heckel and L. Mariani. Automatic conformance testing of web services. In Proc. FASE, Edinburgh, Scotland, Apr., 2-10 2005.

[53] G. J. Holzmann. The model checker SPIN. IEEE transaction on software engineering, 23(5):279–295, 1997.

[54] H. Hrasna. Glassfish community building an open source JavaEE5 application server, 2006.

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

[56] A. Hubbard, C. M. Woodside, and C. Schramm, DECALS: distributed experiment control and logging system, Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, 1995, p. 32.

[57] D. Hughes, P. Greenwood, and G. Coulson, A framework for testing distributed systems, P2P ’04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P’04) (Washington, DC, USA), 2004, pp. 262–263.

[58] IBM. Tivoli: Composite Application Manager for SOA, 2006.

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

[60] INFRAWEBS. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. https://www.comet-consortium.org/

[61] I. Jacobsen G. Booch, J. Rumbaugh. The Unified Modeling Language User Guide. Addison Wesley, 2nd edition, 2005.

[62] A. Keller and H. Ludwig. Defining and Monitoring Service-Level Agreements for Dynamic e-Business. In Proceedings of the 16th Conference on Systems Administration, 2002.

[63] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 327-353, London. Springer-Verlag. 2004.

[64] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. Proceedings of the European Conference on Object Oriented Programming. 1997.

[65] S. Konrad and B.H.C. Cheng. Real-time specification patterns. In ICSE ’05: Proceedings of the 27th international conference on Software engineering, pages 372–381, 2005.

[66] P. Krishnamurthy and P. A. G. Sivilotti, The specification and testing of quantified progress properties in distributed systems, Proceedings of the 23rd international conference on Software engineering, 2001, pp. 201–210.

[67] D.D. Lamanna, J. Skene, and W. Emmerich. SLAng: A language for defining service level agreements. In FTDCS, page 100. IEEE Computer Society, 2003.

[68] A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. Proceedings of the 2nd International Conference on Service Oriented Computing, pages 94-104. ACM, 2004.

[69] R. Lencevicius, E. Metz, Performance assertions for mobile devices, Proceedings of the 2006 international symposium on Software testing and analysis, Portland, Maine, USA. 2006

[70] Z. Li, Y. Jin and J. Han. A Runtime Monitoring and Validation Framework for Web Service Interactions. In Proceedings of the 2006 Australian Software Engineering Conference, 2006.

[71] Z. Li, W. Sun, Z. B. Jiang, X. Zhang. BPEL4WS Unit Testing: Framework and Implementation. In Proc. of ICWS'05, Orlando, Florida, July 11-15 2005, pp. 103-110.

[72] T. Lindholm and F. Yellin, The java virtual machine specification, second ed., Addison-Wesley Professional, April 1999.

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

[74] B. Long, D. Hoffman and P. Strooper, Tool Support for Testing Concurrent Java Components, IEEE Transactions on Software Engineering, Vol. 29 (6), Jun 2003, 555-566.

[75] B. Long and P. Strooper, A case study in testing distributed systems, DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications (Washington, DC, USA), 2001, p. 20.

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

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

[78] MADAM - Mobility and Adaptation Enabling Middleware. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. http://www.ist-madam.org

[79] K. Mahbub and G. Spanoudakis. A framework for requirements monitoring of service-based systems. Proceedings of the 2nd International Conference on Service Oriented Computing, pages 84-93. ACM, 2004.

[80] A. Martinez, Y. Dimitriadis, and P. de la Fuente. Towards an XML-based model for the representation of collaborative action. In Proceedings of the Conference on Computer Support for Collaborative Learning (CSCL '03), pages 14–18, Bergen, Norway, June 2003.

[81] B. Meyer. Object Oriented Software Construction, 2nd ed. Prentice-Hall. 1997.

[82] MUSIC - Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/music-project-story_en.pdf

[83] T. Mackinnon, S. Freeman, and P. Craig, Endo-testing: Unit testing with mock objects, Proceedings of eXtreme Programming Conference 2000 (XP2000), May 2000.

[84] OASIS. *UDDI Core v2 and v2/v3 Utility Classification Schemes, Taxonomies, Identifier Systems, and Relationships*. http://uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm

[85] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, December 2005

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

[87] D. Peled. Combining partial order reductions with on-the-fly model-checking. In Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94), volume 818 of LNCS, pages 377–390. Springer-Verlag, 1994.

[88] S. E. Perl, W. E. Weihl, Performance assertion checking, Proceedings of the fourteenth ACM symposium on Operating systems principles, p.134-145, Asheville, North Carolina, USA. 1993.

[89] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, A blueprint for introducing disruptive technology into the internet, ACM SIGCOMM Computer Communication Review 33 (2003), no. 1, 59–64.

[90] P. Pettersson and K. G. Larsen. UPPAAL2k. Bulletin of the European Association for Theoretical Computer Science, 70:40–44, February 2000.

[91] M. Pistore, F. Barbon, P. Bertoli, D. Sharapau and P. Traverso. Planning and Monitoring Web Service Composition. Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[92] PLASTIC IST STREP Project. Deliverable D1.1: Scenarios, Requirements and Initial Conceptual Model.

[93] PLASTIC IST STREP Project. Deliverable D2.1: SLA language and analysis techniques for adaptable and resource-aware components.

[94] W. Robinson. Monitoring web service requirements. In Proceeding of the International Conference on Requirements Engineering, 2003.

[95] M. Rutherford, A. Carzaniga, and A.L. Wolf. Simulation-Based Test Adequacy Criteria for Distributed Systems. Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-14), November, 2006.

[96] A. Sahai, V. Machiraju, M. Sayal, A. P. Moorsel and F. Casati. Automated SLA Monitoring for Web Services. In Proceedings of the 13th IFIP/IEEE international Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications, LNCS 2506, 2002.

[97] I. Satoh. Software Testing for Wireless Mobile Computing. *IEEE Wireless Communications*, Oct. 2004, pp. 58-64.

[98] SeCSE - Service Centric System Engineering. European Commission Project. Information Society Technology. Funded under 6th Framework Programme. http://secse.eng.it

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

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

[101] R.N. Taylor, D.L. Levine and C. D. Kelly, Structural Testing of Concurrent Programs, IEEE Transactions on Software Engineering, Vol. 18 (3), Mar 1992, 206-215.

[102] L. Tauscher and S. Greenberg. How people revisit web pages: Empirical findings and implications for the design of history systems. International Journal on Human-Computer Studies, 47(1):97–138, 1997.

[103] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In *Software—Concepts and Tools 17 (1996), pp. 103–120*.

[104] W.T. Tsai et al. Verification of Web Services Using an Enhanced UDDI Server, Proc. of WORDS 2003, Guadalajara, Mexico, Jan., 15-17, 2003, pages131—138.

[105] W. T. Tsai et al., Scenario-Based Web Service Testing with Distributed Agents, IEICE Transaction on Information and System, 2003, Vol. E86-D, No.10, pp. 2130-2144.

[106] W.T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, R. Paul "Testing Web Services Using Progressive Group Testing" in Proceedings of Advanced Workshop on Content Computing (AWCC'04) LNCS3309, pp. 314-322

[107] UDDI Consortium. UDDI Executive White Paper, Nov. 2001. http://uddi.org/pubs/UDDI_Executive_White_Paper.pdf

[108] B. Verheecke, M. A. Cibrán and V. Jonckers. Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection. In Proceedings of European Conference on Web Services 2004, LNCS 3250, September 2004.

[109] Vetter, J. S. and Worley, P. H. 2002. Asserting performance expectations. In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Baltimore, Maryland). Conference on High Performance Networking and Computing. IEEE Computer Society Press, Los Alamitos, CA, 1-13. 2002.

[110] Y. Wang, M. Rutherford, A. Carzaniga, and A.L. Wolf. Automating Experimentation on Distributed Testbeds. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), Long Beach, California, November 2005, pp. 164-173.

[111] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, Massachusetts, December 2002, pp. 255-270.

[112] A. W. Williams and R. L. Probert, A practical strategy for testing pair-wise coverage of network interfaces, Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96), 1996, p. 246.

[113] A. W. Williams and R. L. Probert, A measure for component interaction test coverage, Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, 2001, p. 304.

[114] WS-I. The WS-I organization web page: http://www.ws-i.org/