University of Sheffield
Department of Computer Science

Automated Runtime Testing of Web Services

Ervin Ramollari

Submitted towards the degree of
Doctor of Philosophy

May 2013

Abstract

Service-oriented computing (SOC) is a relatively new paradigm for developing software applications through the composition of software units called *services*. With services, software is no longer owned but offered remotely, within or across organisational borders. Currently, the dominant technology for implementing services is that of *Web services*. Since service *requestors* do not usually have access to the implementation source code, from their perspective, services are offered as black boxes. However, requestors need to verify first that provided services are trustworthy and implemented correctly before they are integrated into their own business-critical systems. The verification and testing of remote, third-party services involve unique considerations, since testing must be performed in a black-box manner and at runtime.

Addressing the aforementioned concerns, the research work described in this thesis investigates the feasibility of testing Web services for functional correctness, especially at runtime. The aim is to introduce rigour and automation to the testing process, so that service requestors can verify Web services with correctness guarantees and with the aid of tools. Thus, formal methods are utilised to specify the functionality of Web services unambiguously, so that they are amenable to automated and systematic testing. The well-studied stream X-machine (SXM) formalism has been selected as suitable for modelling both the dynamic behaviour and static data of Web services, while a proven testing method associated with SXMs is used to derive test sets that can verify the correctness of the implementations.

This research concentrates on testing *stateful* Web services, in which the presence of state makes their behaviour more complex and more difficult to specify and test. The nature of Web service state, its effect on service behaviour, and implications on service modelling and testing, are investigated. In addition, comprehensive techniques are described for deriving a stream X-machine specification of a Web service, and for subsequently testing its implementation for equivalence to the specification. Then, a collaborative approach that makes possible third-party Web service verification and validation is proposed, in which the service provider is required to supply a SXM specification of the service functionality along with the standard *WSDL* description of its interface. On top of that, techniques are proposed for service providers to include information that ground the abstract SXM specification to the concrete Web service implementation. Having these descriptions available, it is possible to automate at runtime not only test set generation but also test case execution on Web services. A tool has been developed as part of this work, which extends an existing SXM-based testing tool (JSXM). The tool supports the tester activities, consisting of generation of abstract test cases from the SXM specification and their execution on the Web service under test using the supplied grounding information. Practical Web service examples are also used throughout the thesis to demonstrate the proposed techniques.

Acknowledgements

First of all, I wish to express my deepest gratitude to both supervisors of my PhD project: Dr Dimitris Dranidis from City College and Dr Tony Simons from the University of Sheffield. I felt fortunate and privileged to have their insightful ideas and much-needed advice during the whole duration of the PhD research. Also, I would like to thank them for their continuous support and patience whenever I ran into difficulties during this long and tough, yet exciting journey.

Special thanks go to my valued colleagues of the Software Engineering and Service-Oriented Technologies (SoE) research group at SEERC: Dr Iraklis Paraskakis, Dimitris Kourtesis, and Kostas Bratanis. Our discussions and brainstorming meetings generated many of those cool ideas, which contributed directly to the work described in this thesis, as well as to several research papers where we were co-authors.

I could not forget to mention the support from the other colleagues from the CITY College Computer Science Department (CSD), who made me feel part of the department. I would like to single out Professor Petros Kefalas, for his valuable feedback during different stages of my PhD research. His professionalism and pursuit of excellence were always a source of inspiration for me toward improvement.

Above all, I am particularly grateful to my dear family. Undoubtedly, without their indispensable support, patience and encouragement, this PhD project would not have been possible. Thank you for being with me, even when we were far away from each other!

Table of Contents

Chapter 1 – Introduction **1**
1.1 Motivation 1
1.2 Aims and objectives 3
1.2.1 Aims 3
1.2.2 Theoretical objectives 3
1.2.3 Technical objectives 4
1.2.4 Experimental objectives 4
1.3 Contribution of this thesis 4
1.4 Thesis outline 5
PART A – Literature Review **9**
Chapter 2 – Background on Service Oriented Computing and Web Services **11**
2.1 Service Oriented Computing 11
2.2 Web services 12
2.2.1 Web Services Communication - SOAP 13
2.2.2 Web Services Description – WSDL 15
2.2.3 Web Services Discovery - UDDI 17
2.2.4 WS-* Extensions and the WS-I Basic Profile 18
2.2.5 Message Exchange Patterns 19
2.2.6 Message styles: Document- versus RPC-style Web services 20
2.2.7 XPath, XQuery and XSLT 21
2.3 Service descriptions beyond WSDL 22
2.3.1 Describing data maintained by Web services 24
2.3.2 Describing Web service conversation protocols 26
2.4 Semantic Web services 28
2.4.1 Necessity for Semantic Web services 28
2.4.2 SWS frameworks 29
2.4.3 Semantic Web Services Grounding 32
2.5 Service composition 33
2.5.1 Current standards in service composition 33
2.5.2 Service composition beyond current standards 34
2.6 Summary 35
Chapter 3 – Related Work on Web Service Verification and Testing **36**
3.1 Verification, Validation and Testing 36
3.2 Types of testing 37
3.3 Formal Methods and Model-Based Testing 39
3.3.1 Formal methods 39
3.3.2 Model-based testing 39
3.4 Testing SOA and Web services 40
3.5 Formal verification of Web services 41

3.5.1 Formal methods and Web services 41
3.5.2 Formal verification of individual Web services 41
3.5.3 Formal verification of Web service compositions 43
3.6 Testing tools 45
3.6.1 Web service testing tools 45
3.6.2 Tools for model-based testing 45
3.7 Summary 47
Part B – Specifying and Testing Stateful Web Services 49
Chapter 4 – Web Services with State and Testing Implications 51
4.1 Stateless versus stateful Web services 51
4.2 Web service state and behaviour 53
4.2.1 Web service state 53
4.2.2 Web service behaviour 54
4.2.3 Effect of state on Web service behaviour 55
4.3 Characteristics of Web service state 56
4.3.1 State accessibility 56
4.3.2 State duration 56
4.3.3 Views on private state 57
4.3.4 Private state identification 58
4.3.5 Classification of Web service state 64
4.3.6 Classification of Web services based on state 66
4.4 Implementation of stateful Web services 68
4.4.1 Stateful Web services in Apache Axis2 68
4.4.2 Stateful Web services in JAX-WS 70
4.4.3 Stateful Web services in Oracle Weblogic 70
4.4.4 Stateful Web services in IBM WebSphere studio 70
4.5 Prevalence of stateful Web services 71
4.6 Summary 71
Chapter 5 – Modelling Stateful Web Services with Stream X-Machines 73
5.1 Two service examples: Bank Account and Supply Order 74
5.1.1 Bank Account 74
5.1.2 Supply Order 75
5.2 State-based formalisms and Web service modelling 76
5.2.1 Finite State Machines 77
5.2.2 Extended Finite State Machines 80
5.2.3 Stream X-Machines 80
5.3 Background on Stream X-Machines 82
5.3.1 The Stream X-Machine formalism 82
5.3.2 Other properties of stream X-machines 84
5.3.3 Other variants 84
5.4 Correspondence between Web service elements and SXM elements 85
5.4.1 SXM inputs 85
5.4.2 SXM outputs 87

5.4.3 SXM states and memory 87
5.4.4 SXM transitions 88
5.4.5 SXM processing functions 88
5.5 Modelling practices in the Web services domain 89
5.5.1 Modelling individual stateful objects 90
5.5.2 Other abstraction techniques 92
5.5.3 Modelling large data repositories 95
5.5.4 Specifying sample input values 97
5.6 Deriving a stream X-machine model from IOPE specifications 99
5.7 Controllability and completeness of specifications 100
5.7.1 Controllability 100
5.7.2 Completeness 101
5.8 Nondeterminism 103
5.8.1 Nondeterminism of implementations and specifications 103
5.8.2 Shared-state Web services and nondeterminism 106
5.9 Summary 106

Chapter 6 – Notation and Examples 108

6.1 Notation for defining stream X-machine models 108
6.1.1 XMDL 108
6.1.2 FLAME 109
6.1.3 JSXM 110
6.1.4 Adopted notation 112
6.2 Examples 113
6.2.1 The Account example 113
6.2.2 The SupplyOrder example 115
6.3 Summary 120

Chapter 7 – Testing Web Services Modelled as Stream X-Machines 121

7.1 The Stream X-Machine integration testing method (SXMT) 122
7.1.1 Theoretical basis 122
7.1.2 Derivation of sequences of processing functions 123
7.1.3 Derivation of sequences of test inputs 124
7.1.4 Test case generation for non-controllable and partially-specified specifications 125
7.2 Test case execution 127
7.2.1 Overall process 127
7.2.2 Derivation of expected outputs 127
7.3 Examples 128
7.3.1 The Account example 128
7.3.2 The SupplyOrder example 130
7.4 Equivalence versus conformance testing 132
7.5 Error outputs and negative testing 135
7.5.1 Outputs, error responses and faults 135
7.5.2 Negative testing 136

7.6 Testing considerations in the Web services domain 138
7.6.1 Message Exchange Patterns of Web services under test 138
7.6.2 The need for a sandbox (test) interface 139
7.6.3 Services with undesirable side effects 139
7.6.4 Resetting the WSUT to the initial state 140
7.7 Finding faults 141
7.7.1 Evaluation of test cases through manual injection of control flow faults 142
7.7.2 Test cases evaluation through manual injection of individual processing function faults 148
7.7.3 Test cases evaluation through automated mutation testing 149
7.7.4 Effectiveness of test cases when design-for-test conditions are not satisfied 151
7.8 Summary 152

Part C – Approach for Run-Time Testing of Third-Party Web Services 153

Chapter 8 – Distributed Approach for Verification and Validation of Services in a SOA Environment 155

8.1 The big picture 155
8.1.1 The service provider perspective 156
8.1.2 The service broker perspective 156
8.1.3 The service requester perspective 157
8.2 Benefits of including SXM specifications in service descriptons 158
8.3 Testing scenarios 158
8.4 Discussion 158
8.5 Summary 159

Chapter 9 – Technical Approach for Testable Web Services with Stream X-Machines 160

9.1 Bridging the abstraction gap 161
9.1.1 Adaptation versus transformation 162
9.1.2 Adaptation 163
9.1.3 Transformation, lowering and lifting 164
9.1.4 Patterns of mismatch 164
9.2 SAWSDL annotation mechanisms 165
9.2.1 Augmenting WSDL with the JSXM specification 167
9.2.2 Annotations for grounding 167
9.2.3 Schema mapping examples 170
9.3 Runtime mapping mechanisms 172
9.3.1 Correspondence between Web service and JSXM inputs and outputs 173
9.3.2 Extracting the schema mappings 175
9.3.3 Mapping types 176
9.3.4 Dispatching approach 176
9.4 Handling the Constant Field Pattern and Manager Pattern 177
9.4.1 Constant Field Pattern 178

9.4.2 Manager Pattern 179
9.5 Summary 180

Chapter 10 – Toolset for Automated Testing of Web Services Modelled as SXM 181

10.1 Test case execution toolset 181
10.2 Review on available tools and libraries for writing Web service tools.. 182
10.3 Used tools/APIs: 182
10.4 Summary 185

Chapter 11 – Conclusions and Future Work 186

11.1 Summary of findings 186
11.2 In support of the initial aims 187
11.2.1 Formal verification and testing of stateful Web services with stream X-machines 187
11.2.2 Feasibility of testing third-party Web services 188
11.2.3 Degree of test automation 189
11.2.4 Tool support 190
11.3 Future work 190
11.3.1 Testing individual processing functions 190
11.3.2 Nondeterministic Web services and specifications 191
11.3.3 Testing service compositions 191
11.3.4 Editor and graphical modelling tool for JSXM specifications 192
11.3.5 Graphical tool for SAWSDL annotations and mappings 192
11.4 List of Publications by the Author 192

Glossary and Acronyms 195

References 197

Table of Figures

Figure 1 - Elements of the basic Web services framework 12

Figure 2 - Structure of a SOAP Envelope; notice that the business logic information is carried in the Body payload 14

Figure 3 - Conceptual UML-based representation of the contents of a WSDL 1.1 document 16

Figure 4 - Top concepts defined by the WSMO ontology [27]................................ 29

Figure 5 - The three OWL-S sub-ontologies [28] 31

Figure 6 – Different variations of testing (Tretmans 2004 [33])............................. 38

Figure 7 - State duration for different types of state 57

Figure 8 – Structure of internal state typically maintained by a multi-user Web service... 58

Figure 9 - State identification by a client filters the state that is accessible by operation calls. Identification can also be performed in steps: client identification, and then object identification. 59

Figure 10 – Identification information can be supplied in three different layers of service requests.. 61

Figure 11 - Sessions store state in the server machine, while cookies, stored in the client machine, identify that state 62

Figure 12 - Web service state is heterogeneous and varies along several dimensions, projected as axes in a five-dimensional space 65

Figure 13 - The Axis2 context hierarchy [75] 69

Figure 14 – Deterministic FSM model of a simplified Bank Account 78

Figure 15 – Nondeterministic FSM model of the simplified Bank Account 79

Figure 16 - State-transition diagram of the Account SXM 88

Figure 17 – Three different views adopted by the specifications of a SupplyOrder Web service with authentication functionality... 91

Figure 18 – Extract of the tree representation of the XML contents of a complex SOAP request message to the UPS Shipping Web service [58] 94

Figure 19 – Partially specified Account SXM 102

Figure 20 – Completely defined Account SXM... 102

Figure 21 - State-transition diagram of a nondeterministic SXM specification modelling a stateful SupplyOrder Web service with inventory lookup. Notice the extra transitions labelled by function "itemUnavailable"...................................... 105

Figure 22 - Parts of a JSXM definition of a processing function and generated Java code 111

Figure 23 – State-transition diagram of the SupplyOrder SXM 115

Figure 24 - Java implementation of Web service operation "deposit", illustrating the operation predicates. The shaded area is the implementation of processing function “deposit” 137

Figure 25 - State-transition diagram of a SupplyOrder implementation with erroneous next state fault 143

Figure 26 - JUnit execution results on the implementation with erroneous next sate fault 143

Figure 27 - SupplyOrder implementation with erroneous transition label 144

Figure 28 - SupplyOrder implementation with missing transition 145

Figure 29 - SupplyOrder implementation with an extra transition fault 145

Figure 30 - SupplyOrder implementation with a missing state fault 146

Figure 31 - Two examples of faulty Web service implementations with one extra state: a) revealed by test sets for k = 1 and b) not revealed by test sets for k = 1 .. 147

Figure 32 - Running the JUnit test sets for values of *k* between 0 and 2 on an order of maximum capacity of three items. The fault is revealed for k = 2 that derives sequences of four adjacent “addOrderLine” functions. 148

Figure 33 - Verification and validation approach of third-party Web services in a SOA environment 156

Figure 34 - Validation and verification paths 159

Figure 35 - Three approaches to bridging the abstraction gap (adopted from Utting and Legeard [34]) 162

Figure 36 - SAWSDL annotations of the SupplyOrder WSDL file with model references and schema mappings 166

Figure 37 - Schema mapping languages for semantic RDF data versus schema mappings for JSXM inputs and outputs 168

Figure 38 - Contents of a Web service fault message 169

Figure 39 – Conventions for correspondence at the instance level between JSXM inputs and SOAP requests 174

Figure 40 – Convention for correspondence at the instance level between SOAP responses and JSXM outputs 175

Figure 41 - Transformation approach for executing test cases 183

Figure 42 - Extension of the FUSION Semantic Registry with service verification capabilities 185

Table of Tables

Table 1 - Correspondence between stream X-machine and Web service elements . 89

Table 2 - Versions of SupplyOrder Web service specification and implementation 119

Table 3 - Combinations of SXM specification and implementation according to their determinism 135

Table 4 - Comparison of mutation testing tools 149

Table 5 - List of publications by the author and relationship to contributions 192

Chapter 1 – Introduction

1.1 Motivation

Service-oriented computing is an emerging paradigm for distributed computing that is changing the way software applications are architected, realised, delivered, and consumed. The term Service-Oriented Architecture (SOA[1]) refers to a software architecture perspective where nodes on a network make computational resources available to other network nodes in the form of services. Services are self-contained, autonomous, highly reusable software components with programmatic interfaces that can be described, discovered and used independently of their underlying platform, implementation language, or software vendor. The prevailing approach for realising SOA today is through Web services, primarily due to the way in which Web services naturally implement the SOA philosophy of loose coupling and reusability. Web services also promote interoperability by adopting widely accepted standards like WSDL, SOAP, and UDDI.

Web services can be offered within organisational borders in private SOA deployments, as well as across organisational borders by third party providers. Recent years have seen an increasing number of Web services made available over the Web by various providers. As a consequence, the issues of trust and dependability on third-party providers have been receiving increasing importance. Service requestors need to ensure that provided Web services satisfy their requirements in different aspects and that they have also been correctly implemented, before integrating them into their systems.

One problem with the current standard for Web service description (WSDL) is that it lacks support for descriptions beyond the external interface of operation signatures. WSDL descriptions lack the means to specify the functionality of a Web service, so that requestors are aware of the exact behaviour expected from the consumed service. Therefore, different standards or languages are required to describe the additional functional and non-functional Web service aspects, including its behaviour.

Besides the need to know more about Web services functionality, it is also necessary to build confidence that they correctly implement that functionality. In

[1] Used acronyms are also defined in the end of this thesis report for quick reference.

other words, service requestors should be able to verify that a Web service implementation complies with its intended behaviour described in its specification. A common technique for performing verification of systems is through testing. As with all types of software artefacts, testing is an integral component of the Web services development lifecycle. However, owing to distinctive characteristics that Web services possess, such as reusability, composability, and substitutability, but also key challenges like trustworthiness and interoperability, testing is indispensable for post-development lifecycle phases as well. In addition, users of Web services offered by third-party providers do not have access on their implementation, given that services are used rather than owned. Therefore, verifying third-party Web services introduces new problems to overcome, since there is a lack of information regarding their behaviour and the tester has no control on their implementation.

As a result, the research work described in this thesis focuses on the possibility of functional testing of Web services, especially ones offered by third parties for which testing must be performed at runtime and in a black-box manner. This work aims towards automation of Web services testing so that requestors or third-party certification authorities are capable of verifying their correctness with accuracy and reasonable effort. Furthermore, it is imperative for testing to be systematic and proven in its ability to reveal possible Web service faults.

Under these circumstances, this work makes use of formal methods to specify the intended behaviour of Web services. Formal specifications have the important advantage of being precise, consistent and unambiguous, owing to their mathematical basis. As a result, formal specifications are suitable for automated testing since they can be processed by means of automated tools and algorithms with sound theoretical foundations. The well-studied stream X-machine (SXM) formalism has been adopted in this thesis, since it is demonstrated to be intuitive and efficient in modelling both the dynamic behaviour and static data of Web services. Furthermore, a powerful testing method applicable to SXMs is capable of deriving test sets, which can prove the correctness of the implementation.

The task of creating a formal behavioural specification of a Web service under test is especially difficult due to the extra complexity that originates from their internal state. In Web services persisting state between invocations, the outcomes of calling the service depend on state, besides the provided request messages. As this thesis will present, state is prevalent in most nontrivial Web services, thus it has to be taken into account during the tasks of specification and subsequent testing.

Taking advantage of the capability of Web services to be self-described, this work proposes to enhance their WSDL descriptions with the inclusion of formal SXM models that explicate their internal behaviour. In this manner, third-party Web services are able to advertise their functionality, overcoming the limitation of WSDL descriptions to declare functional aspects besides the external service interface. More importantly, the supplied SXM specification can be utilised by any interested requestors to generate test sets that can verify the correctness of the

implementation. Thus they are aware of the internal behaviour of Web service operations, and are assured that they have been correctly implemented in the service that is about to be integrated in their systems.

As it will be explained in this thesis, executing the derived test sets on the third-party Web service under test is in reality more complex. The specification represents an abstraction on the Web service under test in several aspects. Thus, the generated tests have to be grounded to the same level of abstraction as the Web service. The work in this thesis aims to tackle this problem by requiring the service provider also to supply additional information that specifies mappings between abstract and concrete inputs/outputs. Having these descriptions available, Web services also possess the desirable property of being testable, since requestors are capable of automatically executing test sets on third-party Web services.

The following sections present the aims and objectives for the work described in this thesis, a summary of contributions, and a synopsis of the rest of the thesis report.

1.2 Aims and objectives

1.2.1 Aims

I. Examine methods and unique challenges in verification and testing of stateful Web services.

II. Investigate the feasibility of testing third-party Web services, for which implementation is unavailable and testing is black-box.

III. Investigate the degree to which testing of third-party Web services can be systematic and automated.

1.2.2 Theoretical objectives

TH01. Investigate the occurrence of state in Web services and its characteristics.

TH02. Classify Web services according to their state and resulting behaviour.

TH03. Investigate modelling and testing requirements for each category

- a. What kind of formalism and expressive power is required
- b. What testing strategy and coverage is adequate

TH04. Perform a review of different testing methods and related work on testing and verification of Web services.

TH05. Propose methods and best practices for specifying Web services as SXMs.

TH06. Devise a methodology for inferring a SXM specification of a Web service.

TH07. Examine the feasibility of testing Web services specified as steam X-machines by application of SXM-based testing methods.

TH08. Propose technical solutions for accomplishing testable third-party Web services which can be tested automatically by interested parties.

TH09. Provide methods and derive patterns for grounding the specification of a Web service to its implementation in order to execute derived test sets on services.

1.2.3 Technical objectives

TE01. Design an architecture to support the different activities in model-based testing of Web services

TE02. Implement a toolset according to the architecture

TE03. Provide facilities for:

TE04. The modeller to create a formal specification of service behaviour and augment it to the WSDL description

TE05. The certification authority/tester to utilize the formal model and test the service

1.2.4 Experimental objectives

E01. Find a set of motivational Web service examples, mainly mock-up, but also at least one real to demonstrate the described techniques.

E02. Evaluate the SXM-based testing approach on the Web service examples and demonstrate its ability to reveal various kinds of faults.

1.3 Contribution of this thesis

This section briefly lists the contributions of this work to be used as guidance for reading the rest of the chapters. Those contributions are summarised as follows:

C1. The study of stateful Web services and service state, a classification of stateful Web services based on state, as well as the implications to specifying and testing such services;

C2. The analysis of the suitability of state-based formalisms for specifying the behaviour of stateful services, with a focus on stream X-machines;

C3. The investigation of the correspondence between SXMs and Web services, a list of best practices for Web service modelling and abstraction, as well as a method for SXM derivation from IOPE-based descriptions;

C4. The investigation of unique testing challenges for third-party Web services;

C5. Evaluation of produced test sets with faulty implementations and mutation tools.

C6. The approach for collaborative validation and verification of third-party Web services, with a focus on testing;

C7. The technical framework for standards-based specification of testable third-party Web services, towards automated test case generation and automated test case execution;

C8. The proposal for bridging the abstraction gap between the concrete Web service implementation and its abstract specification during test cases execution through the definition of schema mappings and patterns;

C9. The architecture and the toolset that support the above ideas;

C10. The set of demonstrative examples.

1.4 Thesis outline

This thesis is logically divided into three main parts. The first part presents a selected overview of the area of Service Oriented Computing and surveys related work on testing and verification of Web services. The second part focuses on specification and testing of stateful Web services using the stream X-machine formalism. The third part describes the application of SXM-based modelling and testing to third-party Web services, both from a theoretical and from a technical perspective, including a description of the developed testing tool.

The summarised contents of each chapter are as follows:

Chapter 2 is a selected overview of the field of Service Oriented Computing. It describes Web services, the first-generation standards of SOAP, WSDL, and UDDI, as well as more advanced topics, such as WS-* extensions, message exchange patterns, message bindings, and finally XML query and transformation languages. Then, this chapter reviews current proposals for addressing the description of Web services beyond the capabilities offered by WSDL, such as describing stateful resources and conversation protocols. Next, semantic Web services and selected frameworks are reviewed and in the end the topic of Web service composition through orchestration and choreography is briefly described.

Chapter 3 provides a brief theoretical background on the topics of verification, validation, and testing. Then it proceeds with a review of existing work that addresses testing and verification of Web service compositions and individual Web services.

Chapter 4 describes stateful Web services and disambiguates the concepts of service state and behaviour. Next, the characteristics and variation of service state are described, including a treatment of state scope, identification and duration. In addition, Web services are classified with respect to state and behaviour into a few practical categories, along with the implications for specification and testing of each category. This chapter concludes with a presentation of techniques for implementing stateful Web services in some of the prevailing Web service frameworks.

Chapter 5 addresses specification of stateful Web services using SXMs. It introduces two Web service examples to be used for illustration throughout the rest of the thesis. This chapter follows with a theoretical background on stream X-machines. Also, it compares three state-based formalisms with one another: FSMs, EFSMs, and defends the choice of SXMs for specifying Web service behaviour and data. Next, in order to provide modelling insight, parallels are drawn between the SXM elements and their Web service counterparts. This chapter further describes modelling practices in the domain of Web services, tackling various problems and unique service characteristics. Derivation of SXM models from IOPE descriptions of service operations is also presented here. In addition, specific SXM properties, such as controllability and completeness of specification are critically investigated. In addition, the notion of nondeterminism referring to Web services and SXM specifications is discussed.

Chapter 6 continues the discussion from the previous chapter with a presentation of the JSXM notation adopted to describe SXMs. This notation is contrasted with the alternative XMDL notation. Also, the two illustrative examples of Account and ShippingOrder are specified in this chapter using the JSXM notation.

Chapter 7 focuses on the application of SXM testing methods to derive test sets for Web services. It describes the theory for derivation of sequences, test inputs, and expected outputs from a SXM specification. Test set derivation is critically illustrated with the SXM specifications of the two Web service examples, which do not satisfy the design-for-test conditions and are partially-specified. A number of unique testing considerations in the domain of Web services are also explored along with proposed solutions. The chapter concludes with the description of some experiments intended to evaluate the test sets derived earlier to reveal various faults, such as control flow faults and ones introduced by mutation tools.

The techniques described in chapters 5, 6 and 7 can be applied in any context, such as by the provider during development time. On the other hand *Chapter 8* specifically addresses testing of third-party Web services. It uses the idea of SXM-based Web service specification and testing in the context of a collaborative approach that involves the service provider, broker, and requestor. It consists of requestor-based validation of SXM models and registry-based testing of third-party Web services specified by SXMs.

Chapter 9 focuses in depth on the service verification part of the approach from the previous chapter. At first, it introduces the problem of bridging the abstraction gap between the SXM specification and the Web service implementation and presents three alternative approaches to run the abstract test cases on the concrete WSUT. Next, techniques are described for accomplishing the vision of testable third-party Web services. These techniques involve service providers who perform annotations (using the SAWSDL W3C recommendation) of WSDL descriptions with extra information for testing, and certification authorities who utilise the extra information to derive test sets and execute them on the WSUT.

Chapter 10 describes the tool developed as part of this work in support of testing third-party Web services, drawing from the techniques presented in chapter 9. The tool is based on the transformation approach for bridging the abstraction gap. It takes advantage of an existing tool for SXM-based test case generation, and other APIs for the rest of the tasks. In particular, the tool relies on EasySAWSDL for parsing annotations in testable Web services to extract the SXM model and the schema mappings used during test case execution. Furthermore, the tool is incorporated into an open-source service registry, which invokes the tool to test Web services prior to their registration.

Chapter 11 is an important chapter that concludes the work described in this thesis.

PART A – Literature Review

- *Chapter 2 – Background on Service Oriented Computing and Web Services*
- *Chapter 3 – Related Work on Web Service Verification and Testing*

Chapter 2 – Background on Service Oriented Computing and Web Services

2.1 Service Oriented Computing

Service Oriented Computing (SOC) is a new computing paradigm that utilizes services as the key abstraction to support the development of rapid, low-cost and easy composition of distributed applications even in heterogeneous environments [1]. Services are loosely coupled, reusable, and implementation-independent software modules with well-defined interfaces. They can be *described*, *published*, *discovered*, and dynamically *assembled* for developing massively distributed, interoperable, evolvable systems. Services are provided by service providers within or outside the boundaries of an enterprise, and consumed by service requestors.

The subject of SOC is vast and enormously complex, spanning many concepts and technologies that find their origins in diverse disciplines that are woven together in an intricate manner [1]. The material in research spans an immense and diverse spectrum of literature, in origin and in character. As a result research activities at both worldwide as well as at European level are very fragmented.

Central to SOC is the concept of Service Oriented Architecture (SOA), which is a technology-agnostic *architectural style* for organizing distributed applications with services. SOA promotes the concepts of alignment between the problem domain and IT by raising the level of abstraction of the fundamental units (services) to the business level. Multiple patterns that define design, implementation, and deployment of the SOA solutions, complete this architectural style [2]. A number of technology alternatives for realizing SOA are available, of which the most popular is the Web services framework. Two other less widespread service oriented technologies are the Grid services and P2P services.

SOA allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration (EAI).

2.2 Web services

Currently, Web services are the dominant implementation alternative for SOA. The basic Web services framework consists of three areas: communication protocol, service description, and service discovery, all of which are specified by open standards. The standard for communication between requestor and provider is the Simple Object Access Protocol (SOAP) [3], the standard for Web service description is the Web Services Description Language (WSDL) [4], and the standard for Web service discovery in service registries is the Universal Description Discovery and Integration (UDDI) [5], [6]. All of these standards build upon the XML language, also defined by W3C. Interaction between the three main participants that are involved, that is, *service requestors*, *service providers*, and *service brokers*, occurs as follows. Service requestors[2] discover Web services in a UDDI service registry maintained by service brokers. They retrieve WSDL descriptions of Web services offered by service providers, who previously published those WSDL descriptions in the UDDI registry. After the WSDL has been retrieved, the service requestor *binds* to the service providers by invoking the service through SOAP. These activities and the involved standards are illustrated in Figure 1.

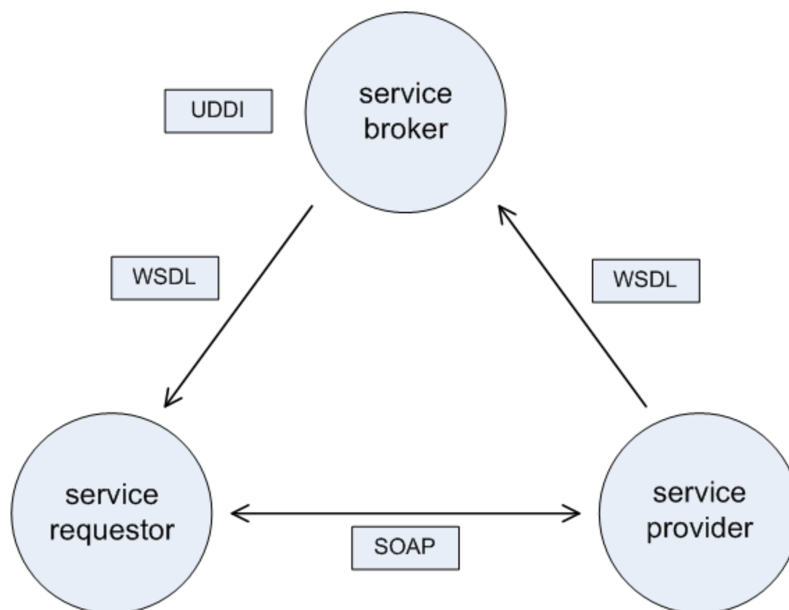


Figure 1 - Elements of the basic Web services framework

Since Web services rely on open and widely-used standards (SOAP, XML, HTTP, URL), they have a high potential to achieve integration of heterogeneous systems, within or across the borders of enterprises. Their long term goal is to provide the infrastructure for plug-and-play and ubiquitous computing [7].

---

[2] The terms “requestor”, “client”, and “consumer” are used interchangeably in the rest of this thesis to refer to the same SOA participant.

Due to the prevalence of the Web services framework in the implementation of services, contemporary SOA has become intrinsically reliant on Web services [8]. Web services concepts and technology used to actualise service-orientation have continuously shaped this paradigm. Therefore, the terms "service" and "Web service" are almost used interchangeably in this thesis: a Web service is a kind of service, and vice versa, by service we usually mean a Web service.

The following three subsections describe the first-generation Web service standards in more detail. The later subsections briefly describe more advanced topics, such as WS-* extensions, message exchange patterns, message bindings, and finally, query and transformation languages operating on XML.

2.2.1 Web Services Communication - SOAP

Web services interact with service requestors and one another by exchanging XML messages over a network. The protocol that governs the exchange and structure of exchanged messages is the Simple Object Access Protocol (SOAP) [3]. The specification of SOAP is currently in version 1.2, which became a W3C Recommendation in 2003 [3]. The SOAP acronym should not be confused with SOA (Service-Oriented Architecture), described previously, which stand for a different concept.

SOAP ensures that the message format and the transport protocol are standard, so that services of heterogeneous implementations can communicate with each other. Exchanged XML messages are structured into *SOAP envelopes*. SOAP envelopes sent as inputs to Web services are known as *request messages*, while SOAP envelopes produced as outputs by Web services are known as *response messages*. The structure of a SOAP envelope in terms of its high-level contents is illustrated in Figure 2.

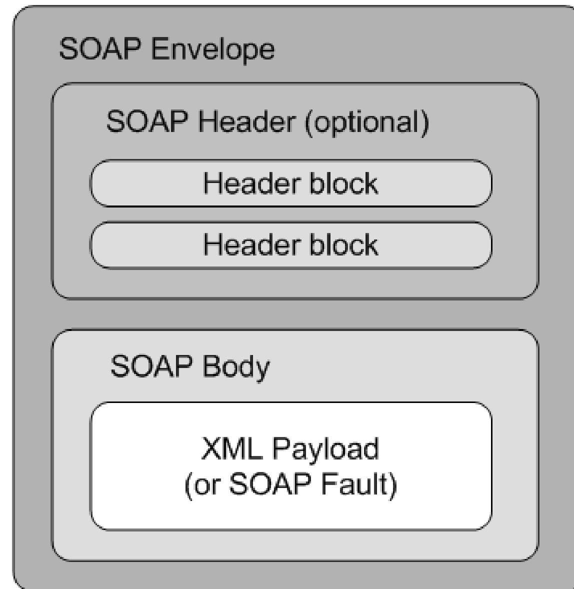


Figure 2 - Structure of a SOAP Envelope; notice that the business logic information is carried in the Body payload

Each SOAP envelope contains two main parts: the header and the body. The *header* is optional, that is, it may or may not be present in the message contents. The header area is dedicated to carrying meta information about the message, which can be structured into several header blocks. Usually, each header block holds information for a corresponding WS-* extensions protocol, such as WS-Security or WS-Addressing (explained later). On the other hand, the SOAP *body* is the actual XML message being conveyed. It represents the message payload and is mandatory in every envelope. Usually this is the only part of the message that is consulted by the business logic implementation in the service and in the requestor.

As a communication protocol, SOAP ignores the semantics of the messages it transports. Thus, SOAP does not prescribe any further structuring of the header blocks and the body payload. Nevertheless, one additional factor, known as the *binding* (see next section on WSDL), is taken into consideration as the final XML message is constructed or interpreted. Overall, the binding tells the SOAP processor whether to follow a document-style or an RPC-style approach. The different possibilities for the binding are explained in further detail in section 2.2.5, Message Styles.

It is important to notice that a SOAP envelope does not represent all the possible information that can be communicated between a requestor and a service. Since SOAP envelopes are transported by a lower level protocol (usually HTTP), they are associated by additional meta information in the headers of that protocol. As an example, HTTP headers can be used to carry identification information as part of cookies in order to manage stateful sessions (section 4.3.4). In addition, the HTTP headers of request messages can contain information necessary for dispatching the message to the correct operation of the Web service, which may not be available

from the carried SOAP envelope. This information is specified in the *SOAPAction* HTTP header, and unless consulted by the recipient, message delivery might fail. An important implication of including additional information in the transport protocol headers is that SOAP envelopes alone are not always the analogues of service inputs and outputs. This fact is taken into consideration in section 9 for the concretisation of abstract inputs to request messages.

2.2.2 Web Services Description – WSDL

An essential characteristic of services, which enables loose coupling, is that they can be described. For this purpose, description documents are required to accompany Web services so that they can be used by prospective requestors. These description documents are written in an XML-based language, called the *Web Services Description Language*, or *WSDL* [4]. The first version of the specification was WSDL 1.0[3], developed in September 2000, and later revised to version 1.1 in March 2001, without any significant changes. The next and current version, WSDL 2.0, introduced major changes to WSDL 1.1 and became a W3C recommendation in June 2007.

The main role of a WSDL document is to describe the *interface* of the Web service. In addition to the interface, WSDL also includes additional details for accessing the service, which are required in the absence of a common middleware platform [9]. As a result, a WSDL service description is organized into two major parts: the abstract part, and the concrete part. The *abstract* part describes the interface characteristics of the Web service, without any reference to protocol binding or hosting details for accessing the service, which constitute the *concrete* part.

Both the abstract and concrete description elements and their relationships are conceptually depicted in Figure 3 in a UML class diagram fashion. The WSDL version is 1.1, while the changes introduced in WSDL 2.0 will be described further below.

---

[3] The *D* in the acronym stood for *Definition*, which was changed to *Description* in WSDL version 2.0.

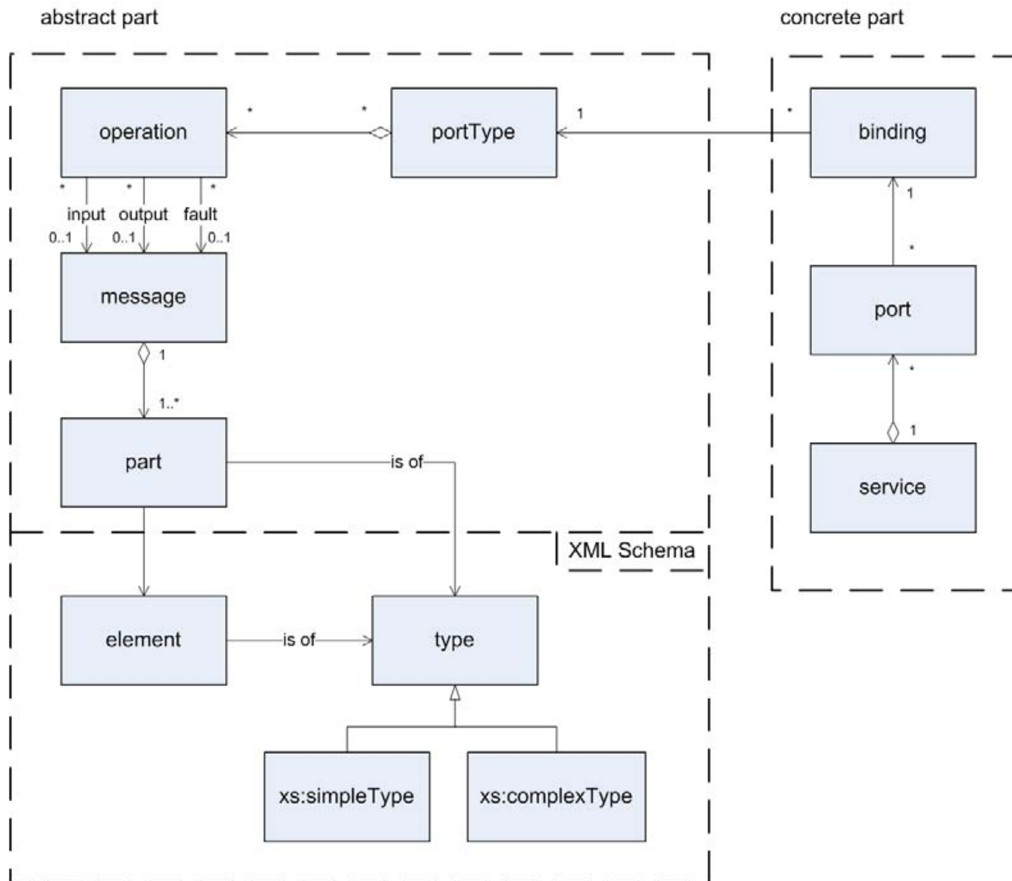


Figure 3 - Conceptual UML-based representation of the contents of a WSDL 1.1 document

The core element of the abstract description is the *port type*, which represents a logical collection of related *operations*. Each operation represents a specific action and is defined as a simple exchange of *messages* (as described further below in Message Exchange Patterns). Messages are the basic unit of communication with a Web service and in WSDL they can be defined as *input* or *output* messages, as well as *fault* messages in cases of operation failures. In turn, every message definition consists of one or more message *parts*. At this point, to further specify the structure of a message part, *XML Schema* (*XSD*) definitions are employed, either inline with the WSDL document or referenced in an external document [10]. The definition of the message part may refer to either an XSD element or an XSD type in the XML Schema. XSD types can be either predefined primitive types (e.g. integers, booleans, etc) or user-defined (either simpleType or complexType).

The concrete part of a WSDL description specifies how to access and invoke a Web service. It consists of three constructs: the binding, the port and the service. A *binding* specifies the message encoding and protocol bindings for all operations and messages defined in a port type. Thus, the binding can be considered as an implementation of an abstract port type, with several different bindings potentially *reusing* the same port type. For example, a binding specifies the messaging style

(document or RPC) and encoding rules for serializing WSDL message definitions to SOAP messages. Also a binding defines the transport protocol (e.g. HTTP or SMTP) to use for carrying SOAP messages.

The *port* construct, also known as an *endpoint*, specifies the physical address (as a URI) over which a binding is made available. Decoupling the port from the binding makes it possible for several ports to reuse the same binding, especially to increase service reliability and to balance load. Finally, a *service* in WSDL is defined as a logical grouping of related ports.

WSDL version 2.0 brings in some substantial changes to the WSDL 1.1 specification described above, in both syntactical and semantical terms [11]. Notably, WSDL 2.0 eliminates the *message* construct (hence, message parts) from the abstract description of the interface. Consequently, operation inputs and outputs refer directly to XML Schema global elements or types, rather than to message definitions. As a result, the abstract description of WSDL is simplified significantly. Moreover, the definition of message types is decoupled from the WSDL language and left entirely to the XML Schema. Another change in WSDL 2.0 is that operations do not support operator overloading. Finally, some syntactical changes are introduced, including the renaming of the root element "definitions" to "description" (hence the change in the meaning of the letter D in WSDL), the renaming of "portType" to "interface", and the renaming of "port" to "endpoint".

One interesting thing to note here is that from the real-world Web services we have investigated so far (such as Google, Amazon, UPS, Paypal, OneAPI, and other services), the WSDL version 1.1 seems to be quite more popular and ubiquitous than WSDL 2.0. This may be due to the fact that, as of the date of this writing (July 2011), the WS-I Basic Profile (see below) does not yet address WSDL version 2.0 [12]. Moreover, a number of the Web service platforms and tools, which aim to be WS-I compliant, do not yet support WSDL 2.0 descriptions. Given these observations, it will be assumed in the rest of this thesis that WSDL documents are described in version 1.1. Nevertheless, this is not a restriction on the version of WSDL expected from service implementations, rather than an assumption for demonstration purposes. The described techniques are expected to be equally applicable to WSDL 2.0, unless stated otherwise.

2.2.3 Web Services Discovery - UDDI

Another important characteristic of services is the ability to advertise and discover them in service *registries*. Registries are especially beneficial when the amount of services increases within and outside organisational boundaries. There are two types of service registries: *private* and *public*. Private registries are implemented within organizational boundaries to keep track of all services maintained by the organization in private SOA deployments. On the other hand, public registries serve to register services provided by any organizations (*third-party services*) as well as the organisations themselves.

In the Web services framework, the specification for service publication and discovery is the *Universal Description Discovery and Integration* (*UDDI*) [5], [6]. UDDI is an OASIS[4] standard that defines data structures and APIs for publishing business entities and service descriptions to the registry and for querying the registry for published descriptions. The defined data structures are businessEntity, businessService, bindingTemplate, and tModel [6]. Eeach *businessEntity* record in the registry contains basic profile information about an organisation acting as a service provider. This record also consists of several *businessServices* each of which describes abstract services offered by the business entity. In a similar fashion to WSDL, UDDI separates binding information from the abstract descriptions. Therefore, the technical information necessary to use a particular Web service is stored separately in *bindingTemplates*. Each businessService can reference one or more bindingTemplates. The information in a bindingTemplate may or may not refer to an actual Web service. If it does, then it references a *tModel* (short for technical model). The tModel finally provides pointers to actual service descriptions, and optionally, additional informal descriptions of what the service does.

In addition to the above data structures, UDDI also specifies APIs for three different types of registry users: service providers that publish services, service requestors that look for services, and other registries that need to share information. Two of the most important UDDI APIs are the *Inquiry API* and the *Publishers API* [5]. Interaction with UDDI APIs takes place through the exchange of SOAP messages, therefore UDDI registries are themselves made available as Web services.

Nevertheless, registries implementing UDDI lack the means for supporting automated service discovery. The main reason is that indexing and retrieval in UDDI is simply based on informal textual descriptions that can be retrieved through keyword-based search. Instead, automated service discovery requires unambiguous and machine-processable representations of Web service capabilities. Considerable research has been performed to overcome this problem with semantically-enhanced UDDI registries. An example of such a registry, which will be considered later on in this thesis, is the open source FUSION Semantic Registry, which offers semantically-enhanced publication and discovery functionalities [13].

2.2.4 WS-* Extensions and the WS-I Basic Profile

The first-generation Web services framework, consisting of SOAP, WSDL, and UDDI, has been extended with further specifications to address new features. Some of these specifications are relatively established including: *WS-Security*, *WS-Addressing*, *WS-Coordination*, *WS-Transaction, WS-Policy*, and many others. WS-Security, for example, defines how to use XML Encryption and XML Signature in SOAP to secure message exchanges, as an alternative or extension to using HTTPS to secure the channel. These extensions are collectively referred to as WS-*, or

[4] http://www.oasis-open.org/

second-generation Web services specifications [8]. Some of the WS-* extensions aim to address the limitations of WSDL in describing certain service characteristics, as will be described in section 2.2.7 – Service descriptions beyond WSDL.

Given the number of available Web service specifications, their complexity, and the different ways in which they can be implemented, it is still a challenging goal to achieve interoperability among Web services from different providers and platforms. Consequently, a well-defined collection of the available standards should be agreed upon to form an interoperable architecture. The Web Services Interoperability Organisation (WS-I)[5] has taken on the task to define a specification for Web services interoperability with their *WS-I Basic Profile* or *WSI-BP* [12]. The latest version of this specification is 2.0, which has been finalized in November 2010 [12]. This version proposes that organisations standardise on the following specifications:

- WSDL 1.1
- SOAP 1.2
- UDDI 2.04 API Specification
- XML 1.0
- XML Schema 1.0
- WS-Addressing 1.0

In addition to recommending the specification versions for interoperability, the basic profile also prescribes how the different features of those specifications should or should not be implemented. For instance, WSI-BP 2.0 requires compliant Web services to use only the document-literal and RPC-literal binding styles, and document-literal messages to contain only one message part (further described in section 2.2.5 - Message Styles).

Most of the prevailing Web service infrastructures, such as Apache Axis, IBM WebSphere, Oracle Weblogic, and Glassfish Metro, aim to be compliant with this profile, since it guarantees a level of industry-wide conformance.

2.2.5 Message Exchange Patterns

WSDL defines ways to organize message exchanges into operations, through what are known as *message exchange patterns*, or *MEPs*. The MEP of an operation is defined in WSDL by the appearance and order of the input and output elements within a WSDL operation definition.

The specification of WSDL 1.1 defines four different message exchange patterns [14]:

- request-response;
- solicit-response;
- one-way;

[5] http://www.ws-i.org/

- notification.

The *request-response* is the most common MEP among Web services and distributed application environments in general. An operation following this MEP accepts a message from the requestor and responds back with a normal or fault message. The *solicit-response* MEP is the reverse of request-response: after submitting a message, the operation expects a normal or fault message. An operation specified with the *one-way* MEP expects a single message and does not have to respond. Finally, an operation specified with the *notification* MEP sends a message and expects no response.

The above MEPs are primitive, single-operation, patterns that do not encompass multiple-operation sequences of more than two messages. Defining more complex and longer sequences of message exchanges requires specification languages or modelling notations beyond WSDL.

2.2.6 Message styles: Document- versus RPC-style Web services

A service requestor must be able to successfully communicate with a Web service based solely on its WSDL description. This means that the requestor must eventually derive SOAP request messages in order to invoke Web service operations, and know the expected structure of SOAP response messages in order to process them. Although the port type in the abstract WSDL description defines Web service operations, messages, and their types, it is still not adequate to derive SOAP messages. Translation of abstract WSDL messages to SOAP messages further depends on the *bindings* defined in the concrete WSDL description. Therefore, continuing the previous discussion on SOAP, this section looks further into how the SOAP body payload is structured and represented, thus clarifying the correspondence between WSDL and SOAP messages.

The WSDL binding definition consists of two attributes, *style* and *use*. The binding style can be either *RPC* (standing for Remote Procedure Call) or *document* style. In addition, a SOAP binding can have either *literal* or *encoded* use. Document style Web services support embedding entire XML documents within the SOAP body. On the other hand, RPC Web services mirror traditional RPC communication and therefore support parameter type data [8]. It is important to notice that the RPC binding style must not be confused with the traditional RPC programming model; it is simply one way to translate a WSDL binding to a SOAP message. The use attribute indicates the type system used in the message. The "literal" use states that XSD data types in WSDL will be directly used to represent the XML content of messages. On the other hand, the "encoded" use dictates that SOAP encoding rules defined as part of the SOAP specification [3] will be applied.

The style and use attributes give four possible combinations that are supported by SOAP:

- RPC + encoded

- RPC + literal
- document + encoded
- document + literal

Of the above binding combinations, only *document-literal* and *RPC-literal* bindings are WSI-BP compliant, thus the other two styles with "encoded" use will not be considered here [12]. The RPC-literal binding uses the method name as the root element of the body payload and inserts all WSDL input message parts as children elements. The resulting request message determines the procedure name and input parameters of an RPC call. The advantage of the RPC style is that the operation name appears in the message body, thus it is easier for the receiver to dispatch it to the correct operation implementation. However, it is difficult to validate such a message with an XML validator, since much of the body contents come from the WSDL rather than XML Schema [15].

In the document-literal binding, WSDL message parts, which reference XSD elements or types, are translated to entire XML documents that are embedded in the SOAP body. The advantage of this approach is that the entire message body is defined in XML Schema independently of WSDL, thus it can be easily validated. Also, unlike RPC style, document-style messages do not assume any convention on the contents and meaning of the payload elements, thus they allow XML documents of arbitrary structure and complexity. However, the fact that document-style messages do not include the operation name, makes dispatching more difficult or even impossible. The WS-I Basic Profile restricts the maximum number of WSDL message parts for document-style bindings to one. Therefore, it can be observed that the basic profile always results in SOAP bodies with at most one root element: RPC-style messages contain the operation name as the root element, while document-style messages are allowed to contain only one document. Also it can be noted that WSDL 2.0 does not allow multiple message parts since WSDL operation inputs and outputs refer to only one XSD element or type directly.

The WSI-BP binding styles are taken into consideration later on in this thesis, where the correspondence between abstract inputs/outputs and SOAP messages is investigated. Nevertheless, most of the binding details are abstracted away and left to tools that generate client implementations in accordance with the specified WSDL bindings.

2.2.7 XPath, XQuery and XSLT

Since almost all Web service languages and specifications revolve around XML, it is appropriate to give at this point a brief overview of the XPath, XQuery, and XSLT languages, which operate on XML documents.

XPath [16] is a language for selecting parts of an XML document. XPath makes use of *path expressions*, which resemble paths in file systems, for hierarchical navigation of XML documents. The XML document is viewed as a tree of nodes (e.g. element, attribute, and text nodes), so that the evaluation of an XPath

expression on the document returns a set of nodes. In support of expressions, XPath also provides basic facilities for manipulation of strings, numbers, and Booleans [16]. XPath can be useful in various contexts in Web service tasks, such as in extracting items of interest from SOAP messages. XPath also serves as a base language upon which more elaborate languages, such as XQuery and XPath, are built.

XQuery [17] is a language designed to build more intelligent queries on XML data than XPath expressions. It adopts a syntax and approach similar to SQL for databases. For example, XQuery uses FLWOR (For, Let, Where, Order by, Return) clauses, which serve as building blocks to build queries of any complexity and nesting level. However, unlike SQL, XQuery supports only querying of XML data and does not handle updates to XML data. To address this issue, an extension to XQuery, called Update Facility, has been defined as a new W3C recommendation [18]. The Update Facility makes it possible to perform complex updates to XML documents, such as node insertions, deletions and modifications. Nevertheless, as a standard, the Update Facility is quite recent, has not been well-established in the industry, and there is little support by libraries and tools.

Finally, XSLT (EXtensible Stylesheet Language Transformations) [19] is a language used to transform an XML document (source tree) into another XML document (result tree). XSLT defines transformations in a declarative style. In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document according to template rules. An XSLT processor takes two input documents, the XML source document and the XSLT stylesheet, and produces an output document.

XSLT scripts can be used in various contexts in Web service environments, such as in mediating between SOAP messages of incompatible schemas. Also, they can transform between SOAP messages and service input/output representations in other XML-based languages, as employed later on in this thesis.

2.3 Service descriptions beyond WSDL

Despite the aim of the Web Services Description Language to describe and document Web services, it fails in specifying additional aspects, both functional and non-functional (or QoS). Regarding functional aspects, WSDL is unable to specify Web service behaviour beyond its external interface, expressed as a collection of operation signatures, input/output messages, and their XSD types. As a result, WSDL omits important aspects of Web services functional behaviour, including:

- *Data accessed and/or modified by the Web service*. Frequently, Web services operate on large and complex data repositories. In many cases these data repositories are structured into collections of data entities. For example, a banking Web service may operate on a collection of bank accounts, each of which is hidden behind an interface of several service operations.

Describing the structure of these data entities and their lifecycle is important in understanding the role and behaviour of the Web service.

- *Correct and accepted sequencing of operations*. Real-world Web services often behave like interactive systems. They operate in accordance with a conversation protocol (also called a choreography), when service operations are invoked in sequences. Only certain sequences are accepted for proper interoperability and for transactions to complete successfully, otherwise they fail. For example, when buying items with a shopping cart Web service, a requestor has first to create an empty shopping cart, and then continue adding or removing cart items, before proceeding to checkout with a non-empty cart. Except for message exchange patterns (MEPs) defined for single operations, the WSDL specification does not support the definition of complex MEPs, which span several operations.
- *Mapping function from inputs to outputs*. For each operation, WSDL specifies input and output message types in XML Schema. This information is intended for the service requestor to properly interact with the service through request and response SOAP messages of valid XML structure. However, beyond message type information there is no further description about how response messages are computed from request messages, that is, what the operation does. Instead, a human individual has to consult other, informal, service documentation or use intuition in order to understand the behaviour of individual operations and the service as a whole.
- *Preconditions and effects on data*. Web services that access and modify data repositories consist of operations that take information from an initial expected state (preconditions), and modify it to a post state (postconditions). It is therefore important to specify how the information is modified, in order to fully describe the role and behaviour of the service.

The interaction model that is directly supported by WSDL is essentially a stateless model of request-response or uncorrelated one-way interactions. This amount of specification is unsatisfactory in describing stateful Web services and Web services operating on complex data structures.

There have been different attempts to address the above shortcomings of WSDL with additional specifications, such as new WS-* extensions. This section gives an overview of such specifications from standards organisations, such as W3C and OASIS, which attempt to describe further functional characteristics of Web services. The focus is on specifications that attempt to describe Web service data and conversation protocols. It should be noted that, in addition to these standards, service description beyond WSDL has also been addressed with what are known as Semantic Web Services (SWS), which are the subject of the next section.

This section is divided into two parts: describing data maintained by Web services, and describing Web service conversation protocols.

2.3.1 Describing data maintained by Web services

It is sometimes important to describe the data structures accessed by Web services in a standardized and consistent manner. It is also useful to specify the relationship between Web service operations and those data structures. This promotes interoperability between service requestors and stateful Web services. An established standard in this direction is the Web Services Resource Framework described below.

The Web Services Resource Framework

The Web Services Resource Framework (WSRF) is a family of specifications from OASIS, which attempt to represent the relationships between a Web service and data objects (*stateful resources*) it acts upon, in an explicit manner [20].

The framework attempts to describe those kinds of services which provide access to or manipulate a set of stateful resources. These services are considered as having stateful interfaces but stateless implementations, in the sense that the implementation delegates responsibility for management of state to another component (such as database or file system) while externally appearing stateful [21].

The specification is founded on the concept of a stateful resource, which contains a specific set of state data, with a well-defined lifecycle, and is acted upon by a Web service. The combination of a Web service and a stateful resource is referred to as a WS-Resource. WSRF defines the type of a WS-Resource in WSDL via the use of the “resourceProperties” attribute of the WSDL portType (interface), which references a Global Element Declaration (GED) in XML Schema. For example, the GED for a simple bank account resource, which consists of two boolean status attributes and an integer balance, would be defined in XML Schema and referenced from the portType as follows:

```
<types>
  <xs:schema>
      <xs:element name="AccountResource">
            <xs:sequence>
                  <xs:element name="isOpened" type="xs:boolean">
                  <xs:element name="isClosed" type="xs:boolean">
                  <xs:element name="balance" type="xs:int">
            </xs:sequence>
      </xs:element>
  </xs:schema>
</types>
...
<portType name="accountPortType"
  wsrp:resourceProperties="tns:AccountProperties">
...
</portType>
```

The Web service operations are associated with the modelled stateful resource through the *implied resource pattern*. This pattern is a set of conventions that allow messages to identify a particular stateful resource through the WS-Addressing

protocol, in which the ResourceID identifier is supplied in the header of every SOAP message.

WS-Resources follow a lifecycle model, incorporating the creation, use, and destruction of the resource. A stateful resource is created through a factory operation, which brings a new instance into existence, assigns an identifier, and returns it with the response message. The identifier is then provided with Web service operation invocations during the use phase, so that the implementation of those operations can use it to identify the stateful resource to be used. To destroy the resource, the requestor sends a destroy request message to the Web service with the identifier, which causes the destruction of the corresponding stateful resource.

Finally, the WS-ResourceProperties specification makes it possible to read, modify, and query the values of resource properties defined in the XML Schema GED (see the above code listing). This is accomplished through standard message exchanges, which should be included as WSDL operations in any portType that uses the wsrp:ResourceProperties attribute to declare a WS-Resource properties document. These messages should identify both the stateful resource with the ResourceID identifier, and the particular resource property. For example to query the “balance” property of an account stateful resource with identifier “ACC0001”, the message exchange complying with WS-ResourceProperties should look as follows:

Request:

```
<soap:Envelope>
    <soap:Header>
          <tns:resourceID>ACC0001</tns:resourceID>
    </soap:Header>
    <soap:Body>
          <wsrp:GetMultipleResourceProperty>
            <wsrp:ResourceProperty>
                tns:balance
            </wsrp:ResourceProperty>
          </wsrp:GetMultipleResourceProperty>
    </soap:Body>
</soap:Envelope>
```

Response:

```
<soap:Envelope>
    <soap:Body>
          <wsrp:GetMultipleResourcePropertyResponse>
                <balance>1500</balance>
          </wsrp:GetMultipleResourcePropertyResponse>
    </soap:Body>
</soap:Envelope>
```

Similarly to the WSRF approach, in this thesis single instances of stateful resources (objects) are modelled whenever possible (called the *per-object view* in section 4.3.3). However, while WSRF models only the *static* XML structure of stateful resources, we also model their *dynamic* behaviour in terms of states, transitions, and computed functions. In addition, no conventions are assumed on the Web service implementation under test as the implied resource pattern does, so that more

generality is allowed. Identifiers of stateful resources can appear anywhere in the SOAP message (header and body), or even in HTTP headers during sessions (as described in section 4). When identifiers are placed in the SOAP body, the modeller is required to specify the locations of the identifiers in arbitrary places within the XML document. Then, during test case execution, the correct stateful resource instance is specified in request messages and driven through the different states.

2.3.2 Describing Web service conversation protocols

The need for specifying and supporting service conversation protocols has led to many different standardization efforts, such as WS-Coordination, WS-Transaction, and WSCL. Besides, other existing standards, such as BPEL and WS-CDL, have been adopted in an ad-hoc manner to specify operation sequencing constraints. Unfortunately, these specifications are not always coordinated and even competing. The next subsections briefly overview some of them.

WS-Coordination

WS-Coordination is a second generation Web Services specification developed by BEA Systems, IBM, and Microsoft in August 2002 [22]. It describes a generic framework for supporting protocols that coordinate the actions of several Web services.

First of all, WS-Coordination does not define a language for describing coordination protocols; instead it is a meta-specification that supports other specialised specifications for coordination (or *coordination types*) [9]. The framework provides means for managing context information in long activities, and supplying that information to multiple participating Web services. For this purpose, the framework involves a central *coordinator*, which lessens the need for the participating services to maintain any context information. Consequently, this specification is not suitable for describing the conversation protocols of the individual Web services.

The two most common coordination types associated with WS-Coordination are WS-Transaction and WS-BusinessActivity. WS-Transaction defines a coordination type to support long-running transactions among participating Web services. In particular it ensures that the ACID (atomicity, consistency, isolation and durability) properties are maintained, and the commit and rollback features are implemented. On the other hand, WS-BusinessActivity defines a coordination type for long-running, complex service activities, involving several participants that are required to follow specific protocols. In contrast to WS-Transaction, business activity protocols do not offer rollback capabilities. Given the potential for business activities to be long-running (hours, days, or even weeks), it would not be realistic to expect ACID properties to be satisfied. Instead business activity protocols provide an optional *compensation* process that can be invoked when exceptions occur [8].

Web Services Conversation Language (WSCL)

WSCL is a W3C draft specification developed by Hewlett-Packard in March 2002, which defines the conversation protocols supported by *individual* Web services [23]. It specifies the XML messages being exchanged and the sequencing of those messages.

The building blocks of the specification are document type descriptions, interactions, transitions, and conversations. The *document type descriptions* specify in XML Schema the types of XML payloads that are exchanged between the requestor and the service. *Interactions* represent Web service operations as document exchanges, which follow any of the primitive message exchange patterns (MEPs) described earlier, with the exception of the initial and final interactions, which are defined as empty interactions with no message exchanges. *Transitions* link two interactions, i.e. they advance the conversation from a source interaction to a destination interaction. Transitions can have constraints on the type of the response message from the source interaction. However, WSCL lacks the means to specify more complex transition constraints, which involve other factors, such as the actual contents of previous messages or Web service internal data. In this sense, the transitions can be nondeterministic. Finally, *conversations*, which are the top-level constructs, are composed of several interactions and transitions linking interactions.

Conversations in WSCL can be depicted as *UML activity diagrams* or *transitions graphs*. The nodes are also called activity states, since they represent activities. Activity diagrams are the reverse of state transition diagrams, since nodes represent activities rather than states, while transitions represent continuation from one activity to the next, rather than actual activities. Given that the WSCL transitions are nondeterministic, WSCL conversations are also *nondeterministic*. This implies that the set of all possible sequences in a correct WSCL conversation specification is a superset of the set of all possible operation sequences accepted by the Web service implementation.

WS-CDL

Although WS-CDL [24] (Web Service Choreography Description Language) is a Web service choreography language (described in more detail further below), it has also been leveraged to describe the dynamic protocol and conversation rules of Web services.

In WS-CDL, the protocol between a user and a service is defined primarily in terms of roleTypes, channels, and choreographies. A channel represents a connection between one client and one service provider. Whenever an operation of the service is invoked, a message, which has an informationType, is sent through the channel.

WS-CDL supports repeating same piece of information in several messages in sequences of invocations through what are known as tokens. Tokens are referenced

in the choreography by tokenLocators, which locate tokens in a message by an XPath expression in the “query” attribute. This technique is often used to locate identifiers within messages, in a similar approach to the one used in this thesis.

e.g

```
<token name="accountID" informationType="xsd:string"/>
...
<tokenLocator tokenName="accountID"
    informationType="depositRequestType"
    query="/depositRequest/AccountId"/>
...
```

Abstract BPEL

Abstract BPEL, which is a BPEL process specification without the concrete bindings, has been used in various works to explicate the conversation protocol of Web services [25]. An Abstract Process may be used to describe observable message exchange behaviour of each of the Web services involved, without revealing their internal implementation. [26] BPEL also supports standard imperative constructs such as if-then-else, case choices, and loops, in order to define complex processes.

Discussion

In this thesis, state-based models are proposed to specify the set of correct conversations between a requestor and a service. In these kinds of models, the states define the possible stages of a conversation. The conversation can be found in only one state at any given time, and only operations associated with transitions from that state are accepted. The invocation of an operation potentially transitions the service to a new state, from which other operations can be invoked, and so forth.

State-based models of Web service protocols are specified in this thesis using the stream X-machine (SXM) formalism, which has the advantage of being mathematically precise, unambiguous, and more significantly, amenable to different verification and validation techniques, including testing. In the subsequent sections we justify the use of state-based models, and in particular, the choice of SXMs among those models, as an intuitive way to specify not only the dynamic conversation protocol of a Web service, but also its internal data (state).

2.4 Semantic Web services

2.4.1 Necessity for Semantic Web services

Current Web service technologies around SOAP, WSDL, and UDDI operate at a syntactic level. As a result, although Web services support interoperability through open standards, they still require human intervention to a large extent, in activities like service discovery, selection, composition, and invocation. These themes are of huge importance to the industry and an active topic of research. One solution to this

problem, which has been widely investigated by researchers, is the addition of semantics to Web service descriptions, with what are known as Semantic Web Services (SWS). Semantics makes Web service artifacts machine-understandable, and introduces the possibility of automation to the activities of service discovery, selection, composition, data mediation, and, as explained in this thesis, service verification. Other approaches involve application of formal methods and mathematical descriptions to achieve unambiguous specification of user goals, Web services, service compositions, and so on. In summary, the recurring problem is that existing standards, such as WSDL, lack semantic and precise description and, in addition, often miss important information.

The following subsections provide an overview of some of the proposals that have emerged in the recent years for adding semantics to Web services. The most prominent of these are WSMO, OWL-S, and SAWSDL (evolved from the older WSDL-S). Collectively they are referred to as SWS frameworks.

2.4.2 SWS frameworks

WSMO

The Web Service Modelling Ontology (WSMO), which is part of the Web Service Modelling Framework [27]) is a formal ontology and language that provides ontological specifications for the core elements of Semantic Web services. It is a W3C member submission that has been developed by the Digital Enterprise Research Institute (DERI) in Galway, and is being promoted by the WSMO initiative. WSMF also includes a Web Services Modelling Language (WSML), a language that provides a formal syntax and semantics for WSMO [27], and the Web Service Modelling Execution Environment (WSMX), an integrated environment for execution.

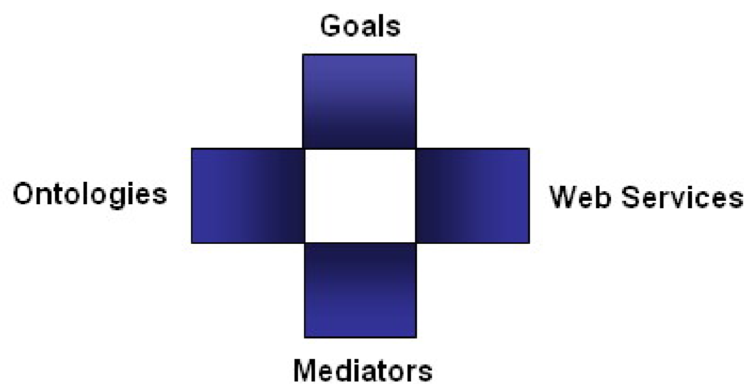


Figure 4 - Top concepts defined by the WSMO ontology [27]

The WSMO ontology consists of four different main elements for describing Semantic Web Services: Ontologies, Web Services, Goals, and Mediators. Ontologies provide the formal semantics to the information used by all other

components. Goals specify objectives that a client might have when consulting a Web service. Web Services represent the functional (and behavioral) aspects which must be semantically described in order to allow semi-automated use. Finally, mediators, used as connectors, provide interoperability facilities among the other elements.

OWL-S

OWL-S [28], formerly DAML-S, is a W3C member submission that defines an OWL-based ontology for Web services. OWL-S consists of a core set of mark-up language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form. OWL-S provides building blocks for rich, formal semantic service descriptions, in a way that builds naturally upon OWL, while the OWL-S ontology provides a vocabulary that can be used together with the other aspects of the OWL to create service descriptions. OWL-S mark-up of Web services aims to facilitate the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition and execution monitoring.

OWL-S is an upper ontology for services, already developed and presented to the Semantic Web Services project of the DAML program, while the OWL-S specification has already been submitted, in November 2004 [28], to become a W3C standard regarding Semantic Web Services. OWL-S classifies the Web Services into two categories as:

- “primitive” in the sense that they invoke only a single Web-accessible computer program, sensor, or device that does not rely upon another Web service, and there is no ongoing interaction between the user and the service, beyond a simple response.
- “complex” that are composed of multiple primitive services, often requiring an interaction or conversation between the user and the services, so that the user can make choices and provide information conditionally.

OWL-S upper service ontology consists of three interrelated sub-ontologies, known as the profile, process model, and grounding, providing three essential types of knowledge about a service, each characterized by the question it answers:

- What does the service provide for prospective clients? The answer to this question is given in the "profile", which is used for service advertising, constructing service requests, and matchmaking,
- How is it used? Or how does it work? The answer to this question is given in the "process model", which enables service invocation, enactment, composition, monitoring and recovery, and
- How does one interact with it? The answer to this question is given in the "grounding". Grounding provides the needed details about transport protocols, mapping the constructs of the process model onto detailed specifications of message formats and protocols.

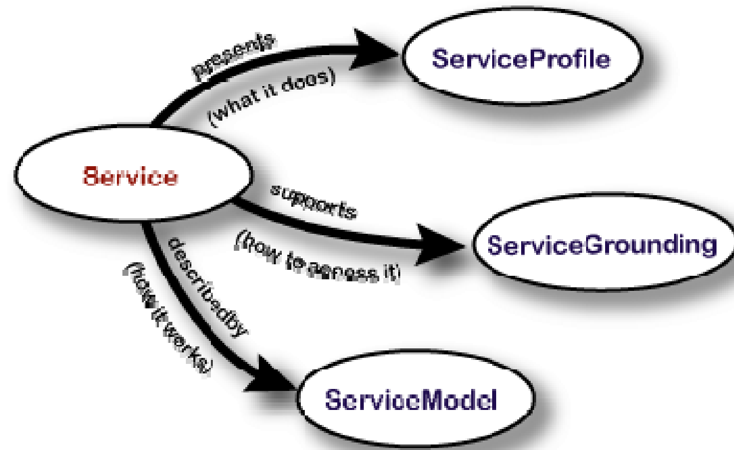


Figure 5 - The three OWL-S sub-ontologies [28]

All OWL-S sub-ontologies (profile, process model and grounding) are linked to the top-level OWL-S concept called Service, which owns the properties "presents", "describedBy", and "supports", serving as an organizational point of reference for declaring Web Services.

SAWSDL

SAWSDL [29], which evolved from the older WSDL-S, is a relatively recent W3C recommendation (August 2007) that defines a set of extensions for WSDL. The extensions specify how to add *semantic annotations* to various parts of a WSDL document, such as input and output message structures, interfaces and operations. The SAWSDL extensions take two forms: *model references* that point to semantic concepts, and *schema mappings* that constitute data grounding for mappings between XML messages and the corresponding semantic model.

A model reference is an extension attribute, *sawsdl:modelReference*, that annotates WSDL and XML Schema constructs in order to point to one or more semantic concepts. The value is a set of URIs, each one identifying some piece of semantics. The unique feature of SAWSDL is that it does not prescribe any particular ontology representation language; a modelReference can point to anything that carries further semantics, such as an OWL instance, a choreography model, the specification of the function of an operation, or even a picture. The annotations only serve as hooks for attaching semantics. In this sense, SAWSDL is considered as a lightweight SWS framework.

SAWSDL provides two attributes for attaching schema mappings: *sawsdl:liftingSchemaMapping* and *sawsdl:loweringSchemaMapping*. Lifting mappings transform XML data from a Web service message into a semantic model (for instance, into RDF data that follows some specific ontology), whereas lowering mappings transform data from a semantic model into an XML message.

Lifting and lowering transformations are useful for communicating with a Web service from a semantic client - for example, the client software will lower some of its semantic data into a request message and send it to the Web service; when the client software receives the response message, it can lift the data contained in the message for semantic processing.

We can also use lifting and lowering annotations for XML data mediation through a shared ontology (see Figure 3b). An automated mediator can lift the data in one XML format to data in the shared ontology and then lower it to another XML format using the lifting annotation from the first format's schema and the lowering one from the second schema.

In XML Schema, we describe an XML element's content by a type definition and add the element's name as an element declaration. SAWSDL model reference and schema mapping annotations can be both on types and on elements; in fact, a type's annotations also apply to the elements of that type.

In particular, a SAWSDL processor merges the type's model references with the element's model references, and all of them apply to the element. Schema mappings, on the other hand, are only propagated from the type if the element doesn't declare any schema mappings of its own. This lets a type provide generic schema mappings and an element specify more concrete mappings appropriate for the type's specific use.

2.4.3 Semantic Web Services Grounding

Any semantic model that describes a Web service needs to be linked with, or *grounded* to, the syntactic WSDL specification, if it is to be used in activities involving service execution. For example, to invoke a discovered semantic Web service, the client needs to know how to construct the request message. *Grounding* is considered as the glue that links the semantic layer with the syntactic WSDL layer of specifications. In the case of testing, a SXM model must be grounded to WSDL, so that the SXM model inputs and outputs can be correlated with Web service requests and responses, during test case execution.

Grounding information can be put in three different places according to Kopecky et al [30]:

- within the semantic model (WSMO, OWL-S);
- embedded in the WSDL document (SAWSDL pointers);
- in an external document.

There are two major types of grounding: *data grounding* and *behaviour grounding*. Data grounding addresses the problem of mapping between Web service SOAP messages and semantic models of those messages. That is, data grounding describes how to transform semantic data to XML messages that will be sent to the Web service, and how XML messages coming back from the service will be interpreted semantically. On the other hand, behaviour grounding addresses the problem of

linking the semantic behavioural model in the SWS specification to the WSDL model of separate operations, each one with a simple exchange pattern (MEP). The behaviour described at the semantic level is also known as the *choreography model* of the service. The choreography model is described either explicitly, in terms of the allowed sequencing of operations, or implicitly, in terms of operation preconditions and effects. Sometimes inputs and outputs of operations are included in the implicit choreography description. The resulting inputs, outputs, preconditions, and effects are also known as IOPE.

In SAWSDL, grounding is accomplished through schema mappings added to WSDL, pointing to XSLT transformation scripts. Grounding for SAWSDL is described in further depth in section 9.2.

2.5 Service composition

Services are normally designed to form part of larger applications by being composed with other services. From a software engineering perspective, applications are no longer developed with traditional design and coding techniques, but through composition of reusable services, which may be provided by third parties. As a result, service composition, and more specifically, Web service composition, has attracted a huge amount of interest from the research community. This interest is also partly due to a number of grand challenges raised in the service composition domain, which require sound and practical solutions, including automated composition, planning, interoperability between composed services, verification of functional and QoS properties in composed services, etc. The next subsection describes current standards in service composition, which tend to be static and manual, while the other subsection briefly gives the state of the art on service composition beyond current standards, including dynamic and automated service composition.

2.5.1 Current standards in service composition

There are two distinct approaches to achieve service composition: service orchestration, and service choreography. In service orchestration services are combined by a central coordinator (the orchestrator) to realise business processes. In contrast, service choreography does not assume a central coordinator, but defines business processes in terms of the conversation that should be undertaken by each participant individually. The resulting process is the summation of the peer-to-peer interactions between the participating services. Several proposals exist for service orchestration, while the proposals for choreography languages are still at a preliminary stage.

Notably, the result from composition of Web services can be published as a new Web service with a new WSDL description, and is known as a composite service. Therefore, it is irrelevant from the perspective of requesters whether a Web service is atomic or composite, since it is only an implementation issue.

The dominating standard for Web service composition through orchestration is the Business Process Execution Language for Web Services (BPEL4WS or simply BPEL), which is also an XML based standard [26]. BPEL originates from two previous languages: WSFL from IBM, and XLANG from Microsoft, mainly to compete with an earlier language, BPML, developed by BPMI.org. Unlike BPEL whose roots were in workflow theory, BPML was inspired by the π-calculus, and hence had a more complete semantics. BPEL models the flow of services through processes, which are net-based concurrent descriptions connecting activities that exchange messages with external WS providers. The BPEL orchestration model combines the activity diagram approach with the activity hierarchy approach [9]. The control structures offered for combining activities are: sequence, switch, pick, while, and flow. In addition to constructs for control flow, variables are used in order to maintain the state of processes and to modify control data. BPEL also supports exception handling with a try-catch-throw approach, as well as transactional properties of processes with compensation handlers. However, BPEL omits certain semantics and process constructs, which make it impossible to model all conceivable business processes. As a result, BPEL is often used in conjunction with other programming languages, such as Java, or extended with proprietary constructs in vendor-specific process execution engines.

On the other hand, the currently prevailing standard for Web services choreography is WS-CDL, an XML-based language that defines the peer-to-peer collaborations of Web service participants [24]. WS-CDL complements BPEL since it defines process behaviour in terms of the common and complementary observable behaviour of the participant, instead of defining it from the point of view of one particular service. The most important element of WS-CDL is the interaction, which describes an information exchange between parties. It consists of three main parts: the participants being involved, the information being exchanged and the channel over which to exchange the information. Messages exchanged between participants are modeled with variables and tokens, whose types can be specified in XML schema or in WSDL. Channels are used to specify how and where message exchanges can take place. Synchronisation among activities is achieved via work units, which define the guard condition that must be fulfilled to continue specific activities.

2.5.2 Service composition beyond current standards

A significant amount of research effort, both in academia and in the industry, is being dedicated to better service composition techniques, than current orchestration and choreography standards can offer. One dimension of improvement in service composition is the replacement of current static and design-time composition strategies, with dynamic and run-time ones. The other dimension, closely related to the first, is automated service composition, as opposed to current manual service composition approaches.

Static composition takes place during design-time, where all participating services are discovered and service interactions are anticipated at design-time. However, the service environment is a highly flexible and dynamic environment. New services become available on a daily basis and the number of service providers is constantly growing. Furthermore, as certain services in a composition fail, or service level agreement (SLA) criteria are not met, it should be possible to discover new similar services (or composite services) from other providers during execution time. Any adaptations to environment changes or user requirements should be able to occur transparently with minimal user intervention. They require dynamic re-composition of services, where a degree of automation is necessary.

Numerous research initiatives are also directed to automated composition of services, where the complex and error-prone task of service composition is moved from the human developer to automated tools. One of the most promising techniques to solve this problem views service composition as a planning problem where individual services are the building blocks that are put together to create end-to-end business processes that satisfy user goals. Some AI planning methods have been proposed, such as situation calculus, PDDL, and rule-based planning [31]. Another approach is the semantic annotation of service descriptions and user goals to make them machine-processable, and hence, to introduce automation in the discovery and composition process. Finally, formal methods, such as automata, Petri nets, process calculi, and Abstract State Machines, have already been proposed as part of automated and semi-automated service composition approaches.

However, in spite of these research initiatives, dynamic and automated service composition is in an early stage of maturity, and is far from being achieved in practice. No effective, easy-to-use, flexible support is provided that can cope with the lifecycle of distributed business processes. Service composition today is largely a static affair, where all service interactions are anticipated in advance and there is a perfect match between output and input signatures and functionality.

2.6 Summary

This aim of this chapter was to familiarise the reader with the general area of service oriented computing and with Web service concepts that are used throughout this thesis. It provided a technical overview of SOAP, WSDL, and UDDI, which are the core standards of the Web services framework. Also, more advanced topics relevant to this thesis were described, such as WS-* extensions, message exchange patterns and bindings, XML query and transformation languages, and frameworks for describing stateful resources and conversation protocols. Next, semantic Web services and selected frameworks were reviewed, which are considered later in this thesis for annotating Web services with behavioural models and their groundings. In the end the important area of Web service composition through orchestration and choreography was briefly described.

Chapter 3 – Related Work on Web Service Verification and Testing

This chapter starts the review of existing work on Web services verification and testing. But, before proceeding to the main part, a brief theoretical background is provided on the topics of verification, validation, and testing. This chapter is split into five sections. The first section aims to disambiguate the commonly-used terms verification and validation, to clarify the role of testing in the context of verification and validation. The second section is a high-level overview of the different flavours of testing, which vary along a number dimensions. The third section introduces formal methods, and their advantages in specifying the behaviour of systems. It also introduces the need for automation of test case generation through formal specifications and model-based testing. The fourth section investigates the application of traditional testing techniques to the service-oriented paradigm, and specifically to Web services. It turns out that, due to the paradigm shift and special characteristics of services, some traditional testing techniques have to be retrofitted into service-oriented computing, while others are not applicable. Finally, the fifth section is the core review of related work on testing and verification of Web services and service compositions, employing formal methods. In the end, the conclusion of this chapter presents the gaps in the existing work, such as achieving automation and bridging of the abstraction gap. The conclusion also motivates the use of the stream X-machine formalism in support of the Web service testing approach described in this thesis.

3.1 Verification, Validation and Testing

According to the IEEE standard computer dictionary [32], v*erification* is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Therefore, verification checks that the system correctly implements the specifications.

On the other hand, *validation* is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified

requirements [32]. That is, validation evaluates the product itself and ensures that it meets user needs. This is difficult to determine and often involves subjective judgements.

It is sometimes said that validation can be expressed by the query “Are we building the right thing?” and verification by “Are we building it right?”. “Building the right thing” refers back to the user's needs, while “building it right” checks that the specifications are correctly implemented by the system.

Since the focus of this thesis is testing of individual third-party Web services, it is relevant to define this activity in the context of verification and validation. Generally, testing as an activity can be used to support *both* validation and verification, depending on what is being tested, and how. Unit, component, and integration testing can be considered as verification activities (being checked against another specification), while system and acceptance testing are generally considered validation activities (checked against user requirements). (Consider the V-model and its different types of testing) Is the black-box testing of individual third-party Web services validation or verification? Is it unit or system testing?

Model-based testing of third-party Web services as described in this thesis is a verification activity, since the Web service implementation is checked against another artefact, which is the SXM specification. Therefore, testing Web services against a formal stream X-machine specification aims to *verify* conformance of the implementation to the behavioural specification. On the other hand, the SXM model itself is normally *validated* against unspecified and ambiguous user requirements. Ultimately, the verification of the implementation against the SXM specification (this service has right implementation) is not useful unless the service requestor is sure the service satisfies the requirements (this is the right service).

3.2 Types of testing

Testing as a concept encompasses a wide variety of testing methods, which differ along a number of dimensions, such as the scale of the system being tested, the properties being tested, and the degree of knowledge about the system implementation (Figure 6).

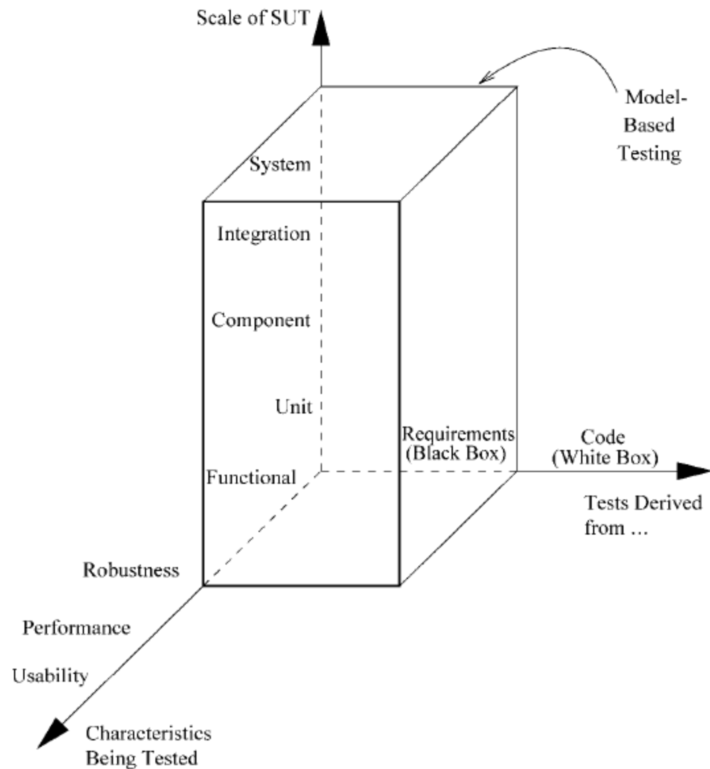


Figure 6 – Different variations of testing (Tretmans 2004 [33])

At first, a high-level distinction can be made between *active testing* and *passive testing (monitoring)*. In active testing the tester drives the system under test with test data and observes the outputs. On the other hand, in passive testing the tester monitors the results of a running system without introducing any special test data.

Black-box (or *functional, behavioural*) *testing* builds a test-set from the system's specification and attempts to prove that the abstract behaviour of the implementation is identical to the specification. *White-box* (or *clear-box*, *structural*) *testing* bases its strategy directly on the implementation code and attempts to show that all parts of the software have been exercised without failure.

Next we make a distinction between *functional* testing and *non-functional* testing. Functional testing tests the SUT against business requirements. Functional testing is done using the functional specifications provided by the client or by using the design specifications like use cases or a formal model provided by the design team. Some common types of functional testing include: unit testing, smoke testing, integration testing, system testing, regression testing, and user acceptance testing. On the other hand, non-functional testing tests the SUT against non-functional requirements. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users. Common types of non-functional testing include load and performance testing, stress testing, security and penetration testing, etc.

Usually, functional testing techniques rely on the availability of a *test oracle*. A test oracle is any entity, which determines the expected, correct output from a system under test. The expected output is used for comparison with the actual output returned by the system, in order to obtain a verdict regarding system correctness. The oracle can be a human tester who knows the expected system behaviour, another system (such as a legacy system), or, as will be discussed further below, a formal specification of the system under test.

3.3 Formal Methods and Model-Based Testing

3.3.1 Formal methods

With *formal methods* systems are specified and modelled by applying techniques from mathematics and logic. Such formal specifications and models have a precise, unambiguous semantics, which enables the analysis of systems and the reasoning about them with mathematical precision and rigour. Moreover, formal languages are more easily amenable to automatic processing by means of tools. Until recently formal methods were a merely academic topic, but now their use in industrial software development is increasing, in particular for safety critical systems and for telecommunication software [33].

A formal specification is a precise, complete, consistent and unambiguous basis for design and code development as well as for testing. This is a first big advantage in contrast with traditional testing processes where such a basis for testing is often lacking. A second advantage of the use of formal specifications for testing is their suitability to automatic processing by means of tools. Algorithms have been developed which derive tests from a formal specification. These algorithms have a sound theoretical foundation. Moreover, they have been implemented in tools leading to automatic, faster and less error-prone test generation. This opens the way towards completely automatic testing where the system under test and its formal specification are the only required prerequisites. Formal methods provide a rigorous and sound basis for algorithmic and automatic generation of tests. Tests can be formally proved to be valid, i.e., they test what should be tested, and only that.

3.3.2 Model-based testing

Model-based testing is defined as the automation of the design of black-box tests [34].

Functional (or black box) testing, should start with a functional specification or description of what the desired system should behave like. The test set is then constructed on this basis and the result of applying the test set is evaluated and compared with the desired result deduced from the specification. It is difficult to construct test sets from informal specifications, it has to be done by hand.

Coverage criteria of testing methods from state-based models [35]:

- State coverage

- Transition coverage
- Full predicate coverage
- Transition pair coverage
- Complete sequence

Effectiveness is another measure, indicating the ratio of the number of detected faults over the overall number of faults in the implementation.

3.4 Testing SOA and Web services

Although traditional verification and testing techniques can be reapplied to service oriented architectures, they need re-inspection due to several specific properties of services. Services are used but not owned, thus they are just interfaces to service requestors. As a result of the lack of access to the source code, it is not possible to perform white-box testing, as well as mutation-testing techniques, which require seeding the code with errors [36].

Another major implication of the lack of service ownership is the need for trustworthiness, security, and reliability, especially when third-party services are being integrated in business-critical or mission-critical applications. Therefore, robust service testing and verification techniques are crucial in order for consumers and integrators to build confidence on third-party services. In addition to testing functional and behavioural conformance, it is also necessary to ensure that a service delivers the expected quality of service (QoS), including indicators such as performance, availability, stress-tolerance, failure handling, etc. Such QoS indicators are often part of established service level agreements (SLAs) that are negotiated between providers and consumers. Testing to guarantee compliance to SLAs has to be performed continuously since the QoS can often vary unpredictably over time.

The separation between the provider and the consumer introduces a number of issues to testing. For example, QoS testing by the provider is not realistic because it doesn't take into account the provider and consumer infrastructure, and the network configuration or load [36]. Since services are consumed remotely, considerable costs in terms of bandwidth and time are involved during testing. Therefore, conducting exhaustive test cases upon services is neither feasible nor practical, and the set of test cases to be applied has to be as selective as possible (Zhang et al., 2005). Moreover, the traditional way of Independent Verification and Validation (IV&V) by the developer is not adequate in the context of services. All parties that are involved, including providers, brokers, and clients, must collaborate in what is referred to as Collaborative Verification and Validation (CV&V) [37].

Integration and regression testing of service oriented systems also raise serious issues [36]. Because of runtime binding to services, it is difficult to exactly predict the services that will be part of a composition and their relationships. Thus, performing integration testing against all possibilities would be costly, and possible endpoints might be unknown at testing time. Regression testing is also challenging,

since it is the provider controlling the implementation of individual services, and the consumer is unaware of any updates that may occur beyond the service interface. Therefore, the consumer does not know when to ensure that changes to individual services have not caused any adverse effects to the whole system.

3.5 Formal verification of Web services

3.5.1 Formal methods and Web services

Formal methods have been attracting significant attention in the domain of service verification, because of their sound mathematical foundation, precise semantics, and automation support. Besides their utilization in the derivation of test cases to verify the correctness of services and service compositions, formal models are also useful in other activities, such as animation and model checking. Different kinds of formal methods have been found valuable in modelling single and composed services. Those formalisms range from process calculi, to algebras, Petri nets, graph transformation rules, and finally, the various kinds of state machines (UML Protocol State Machines, STS, abstract state machines, EFSMs, etc).

The first part of this section, which is more closely related to the work presented in this thesis, describes research works on verification of individual Web services through model-based testing. The second part reviews existing work on formal verification of Web service compositions.

3.5.2 Formal verification of individual Web services

A number of research works have addressed the verification of individual Web services through model-based testing. Different notations have been proposed to specify the desired Web service behaviour in order to verify the compliance of the service implementation to the model. Some of those works make use of formal methods to specify the Web service model.

One of the earliest attempts in this direction is by Tsai et al [46]. The authors propose attaching so-called “test scripts” to WSDL descriptions for use by both service registries and service requestors. The test scripts contain information helpful to testing, including input-output dependency, invocation sequences, hierarchical functional description, and sequence specifications. However, no method is given for generating test cases from this augmented information. In a subsequent work by the same authors [47], they describe a specification-based validation and verification technique, where the specification is written in OWL-S, and the method of Boolean expression analysis is used to extract the full scenario coverage of Boolean expressions. The results are then provided as input to a tool named “Swiss Cheese” in order to generate both positive and negative test cases. The test cases can be used for verifying the correctness of individual service operations but cannot be applied to sequences of operation invocations that constitute the complex behaviour of stateful Web services. As a result, the testing procedure cannot be considered adequate for verifying the functional behaviour of stateful Web services.

Heckel and Mariani [48] propose Graph Transformation (GT) rules as the modelling formalism for specifying the behaviour of stateful services. In essence, the GT rules provide a graphical notation similar to UML class diagrams, which represent the internal state of the service before (preconditions) and after (postconditions) an operation is invoked. A test case derivation method is used to test the actual service implementation against the provided model. This verification is proposed as part of a "high-quality service discovery" approach. The basic idea behind this approach is that both the behaviour of the provided service and the requestor's requirements are specified with GT rules. The service broker utilises the provided specifications to automatically test services before they are admitted in the registry, ensuring that all registered services comply with their formal advertisements. On the other hand, the broker enables matchmaking of request and advertisement models expressed as GT rules during discovery. Thus it returns verified service candidates that satisfy the consumer's behavioural constraints.

Bertolino et al [49] describe a framework where the provider augments the WSDL document with behavioural descriptions in a UML 2.0 Protocol State Machine (PSM) diagram that can be semi-automatically transformed into a Symbolic Transition System (STS) on which existing automated test generation methods can be readily applied. On the other hand, the broker utilises the attached STS model to automatically generate the test cases and run them on the provided Web service for behavioural conformance verification. Upon successful test results the Web service is admitted in the UDDI registry as a certified service. For this reason, the authors call their approach an "audition framework", where the Web service undergoes a monitored trial before being put "to stage".

Keum et al [50] propose Extended Finite State Machines (EFSMs) to model and test stateful Web services. They describe a manual procedure to derive the EFSM model from a WSDL document and additional informal descriptions supplied by a human individual. With proper tool support the EFSM model can be used to automatically generate Web service test cases with increased test coverage that includes both control flow and data flow. The authors provide experimental results showing that their method has the potential to find more faults compared to other methods, but notably without completeness guarantees.

Some existing research work proposes utilising semantic Web service descriptions to infer a formal state-based model of the Web service. In [51] a method is proposed for annotating a WSDL document with concepts from an OWL ontology representing inputs, outputs, preconditions and effects (IOPE), and automatically translating the resulting WSDL-S specification into a semantically-equivalent extended Finite State Machine (EFSM) model. A set of manual or automated techniques for generating test cases based on the EFSM model is also provided. The techniques vary in terms of adequacy criteria, coverage and completeness. However, the derived EFSM model contains only one control state, which is not

sufficient to represent the control flow and state transitions. Therefore the model is not helpful in the process of generating test sequences.

3.5.3 Formal verification of Web service compositions

Numerous formal approaches and languages have been suggested to formally verify certain properties in service compositions, including functional and non-functional ones. Formal verification is also used in conjunction with service composition, especially in dynamic approaches, to ensure the correctness of the resulting compositions. In most verification approaches, service compositions are treated as single processes to be modelled, analysed, and verified, oblivious of the constituent services. On the other hand, in some rare approaches, individual services in service compositions are modelled as communicating entities, to build up a model that can undergo formal verification. Among the most common formalisms used in verification of service compositions are the *process calculi*, *Petri nets*, and *automata*.

A plethora of *process calculi* (SCC, PEPA, SOCK, COWS, SC, etc) is extensively used in EU-Funded Integrated Project SENSORIA [38]. They serve as a basis of mathematical specification of several complementary aspects of service oriented systems, and allow analysis and verification on the models. The Service Centered Calculus (SCC) is a general purpose calculus which enriches traditional process calculi with the concept of sessions. Performance Evaluation Process Algebra (PEPA) is an expressive formal language for modelling distributed systems, which is used for quantitative analysis of services, such as scalability verification. The Service Oriented Computing Kernel (SOCK) is a three-layered calculus, which, among other things, allows reasoning about the whole system composed of all services.

The authors in [39] make use of the Finite State Processes (FSP) calculus to verify Web service composition implementations against specification models through the technique of trace equivalence verification. To perform this type of verification, both the implementation and specifications are expressed in the same language, i.e. FSP. On one hand, the requirements are modelled as Message Sequence Charts (MSCs), a similar notation to UML sequence diagrams, which are in turn compiled to FSPs. On the other hand, the BPEL4WS implementation is translated with tool support to FSPs, and fed to the tool that checks message trace equivalence. Additionally, model-checking is performed on both FSP models to check for certain properties, such as reachability.

Petri nets have also been widely used to model service compositions, since they are a natural way of modelling the various aspects of concurrent systems. A rich theory of concurrent systems based on Petri nets has been developed, and Petri nets have become the model of choice in many applications. For example, the authors in [40] describe an approach where service compositions in BPEL are semantically annotated in DAML-S in terms of a first order logic. With tool support, the DAML-

S descriptions are automatically converted to Petri nets, which are analysed, tested, and verified. Three of the most important properties that are checked by the tool are *reachability*, *liveness*, and *existence of deadlocks*. In [41] a complete and formal Petri-net semantics for BPEL is presented, thus including exception handling and compensations. Furthermore, the authors present their BPEL2PN parser which can automatically translate BPEL processes into Petri nets. As a result, a variety of Petri-net verification tools are applicable to automatically analyze BPEL processes.

Automata, or *labeled transition systems* are a well-known formalism that can model system behaviour in an intuitive way. Several variations of automata exist, such as I/O automata, timed automata, and team automata. Their precise semantics and tool support makes them appropriate to specify, compose, and verify service compositions. One widely used form of automaton is the Promela notation, which can faithfully capture behavioural semantics of processes, and is supported by mature model-checking tools. In [42], the author describes an approach where composition of services is written in WSFL (Web Services Flow Language) and translated into Promela processes, the input language of the SPIN model checker. The application-specific properties to be checked are encoded as formulas of LTL (Linear Temporal Logic), which are also fed into SPIN. General application-independent properties are also checked, which are reachability, and deadlock-freedom. In [43] a case study shows how Web service choreographies written in WS-CDL can be automatically translated to timed automata and subsequently verified by the well-known model checker UPPAAL. In [44] the Orc programming model is used to provide a structured way of orchestrating distributed Web services. It offers intuitive constructors to manage concurrent communication, time-outs, priorities, failure of sites or of communication, etc. Although the precise semantics of Orc makes it suitable for model checking, no tools exist, so the authors define a Timed-Automata semantics for Orc expressions. UPPAAL is then used to model check the Orc models.

The use of state machines and related formalisms to verify service compositions is rarely encountered in the literature. Some approaches that model services as finite state machines, usually aim at automated service composition. For example in [45], a version of Abstract State Machines (ASMs) is used to semi-automatically construct collaborative business processes composed of Web services or simpler individual processes. To the best of the author's knowledge, no approach based on state machines is used to derive test cases for verifying service compositions. A potential formalism to be used for this purpose is the X-machine model, which has a sound theoretical basis on modeling critical systems and concurrent systems (communicating X-machines). Exiting research on X-machines already offers *algorithms for deriving test cases* for complete testing, *model checking techniques*, and an adequate number of *tools* to support different activities related to specification and verification.

3.6 Testing tools

3.6.1 Web service testing tools

A number of Web service testing tools that have emerged support a wide variety of types of testing, in addition to functional testing. These are generic tools and not specialised to a specific form of testing. None are model-based testing tools, but commercial ones, requiring complex executable test scripts. Examples include: SOAPUI, Parasoft SOAtest, PushToTest, SOAPSonar, WSUnit for testing WS consumers, etc.

For example, SOAtest by Parasoft[6] can perform functional testing, load testing, security testing, and interoperability testing. Coyote, described in [90] consists of two parts, test master and test engine. The test master allows testers to specify test scenarios and test cases, and may use WSDL specifications to derive test scenarios, which are sequences of service operations. The test engine interacts with the web services under test, and provides tracing information.

3.6.2 Tools for model-based testing

Spec Explorer

Spec Explorer [91] is a Model-Based Testing tool from Microsoft. It extends the Visual Studio Integrated Development Environment with the ability to define a model describing the expected behavior of a software system. From these models, the tool can generate tests automatically for execution within Visual Studio's own testing framework, or many other unit testing frameworks.

Spec Explorer uses a theory of interface automata to generate tests from Spec# models. Test generation is viewed as a game between the test generation process and the SUT. To enable this game approach to test generation, each method in the model can be annotated as either an Action method (which is under the control of the test generator) or an Observation method (which is under the control of the SUT).

The online testing algorithm (OLT) works as follows (see [91] for a more detailed description). The test generation starts from the initial state of the model and can end whenever execution reaches an accepting state of the model (the modeler can specify which states are accepting states). In each state, Spec Explorer first waits for a state-dependent timeout period to see if an observable event arrives from the SUT. If one does arrive, Spec Explorer checks that the event is allowed by the model, and then takes that transition so that the model follows the SUT behavior. If no observable events arrive before the timeout, Spec Explorer executes one of the controllable methods in the model whose precondition is true, sends the

[6] http://www.parasoft.com/jsp/products/soatest.jsp

corresponding event to the SUT, and checks that this transition is allowed by the SUT.

ModelJUnit

The ModelJUnit library [92] is an open-source extension of JUnit for model-based unit testing of Java classes. ModelJUnit supports both Finite State Machine (FSM) and Extended Finite State Machine (EFSM) models. EFSM models are written in Java language, and because it is an extension of JUnit, the tests are run in the same way as other JUnit tests. With the ModelJUnit library, one can start with an extremely simple FSM model and begin testing immediately, and then progress to slightly more sophisticated EFSM models as desired.

The basic philosophy of ModelJUnit is to take advantage of the expressive power of Java (procedures, parameters, inheritance, annotations, etc.) to make it easier to write EFSM models, and then provide a collection of common traversal algorithms for generating tests from those models. It is typically used for online testing, which means that the tests are executed while they are being generated. The EFSM model usually serves both as the abstract specification of possible states and transitions, as well as the adaptor that bridges the gap between the specification and the SUT (which is usually another Java class).

To be a valid EFSM model, a Java class must have minimally four methods:

- Object getState(): returns the current visible state of the EFSM. So this method defines an *abstraction* function that maps the internal state of the EFSM to the visible states of the EFSM graph. Typically, the result is a string, but it is possible to return any type of object.
- void reset(boolean): This method resets the EFSM to its initial state. When online testing is being used, it should also reset the SUT or create a new instance of the SUT class.
- @Action void $name_i$(): The EFSM must define several of these action methods, each marked with an @Action annotation. These action methods define the transitions of the EFSM. They can change the current state of the EFSM, and when online testing is being used, they also send test inputs to the SUT and check the correctness of its responses.
- boolean $name_i$Guard(): Each action method can optionally have a guard, which is a boolean method with the same name as the action method but with “Guard” added to the end of the name. When the guard returns true, then the action is enabled (so may be called), and when the guard returns false, the action is disabled (so will not be called). Any action method that does not have a corresponding guard method is considered to have an implicit guard that is always true.

Each action method typically defines a short, straight-line sequence of JUnit code that tests one aspect of the SUT by calling one or more SUT methods and checking the correctness of their results. The effect of applying model-based testing to the

EFSM is to make a traversal through the EFSM graph, and this weaves those short sequences of test code into longer sequences of more sophisticated tests that dynamically explore many aspects of the SUT.

Using Java as the notation for writing EFSMs has benefits and limitations. The benefits include the familiarity of Java, having the expressiveness of a full programming language available, and the ability to quickly change the structure of the EFSM graph simply by redefining the getState() abstraction function or by modifying the guards and actions.

Some of the limitations are that the guards and transitions are defined as executable methods rather than as symbolic formulae. So graph exploration and test generation algorithms can execute guards and transitions and inspect their results (true/false from a guard or a new EFSM state after a transition), but they cannot inspect the internal structure of the guards or transitions. To create the EFSM graph, ModelJUnit is limited to exploring it dynamically by executing enabled transitions. This means that it can be difficult to obtain the whole graph if some guards are rarely true. On the other hand, even if the EFSM graph is too large to explore completely, some forms of test generation are still possible, so the EFSM approach is still useful.

Another limitation is that the SUT interactions are handled internally within each transition, so the SUT input and output values are not explicitly represented in the EFSM graph as they are in a Mealy machine FSM model. This places some small limitations on the test generation algorithms and coverage metrics that we can use in ModelJUnit. For example, we can measure action coverage and state coverage but not input coverage or output coverage. *One can use transition-tour test generation algorithms but not some other test generation methods, such as the W-method, that analyze the output part of transitions.* However, in practice this limitation is outweighed by the benefit of being able to generate rich SUT inputs dynamically and perform more sophisticated checking of the SUT outputs than the simple equality check of a Mealy machine FSM.

3.7 Summary

This chapter provided a brief theoretical background on the topics of verification, validation, and testing. More significantly, previous research performed by other authors on Web service testing was critically evaluated in order to position the work described in this thesis relative to the state-of-the-art. Existing research was categorised into work addressing testing and verification of individual Web services, as well as Web service compositions. In the end of the chapter, two software tools used in model-based testing of systems were reviewed: Spec Explorer and ModelJUnit. This tool review is considered as a necessary background to the Web service testing toolset described in chapter 10, which makes use of the JSXM model-based testing tool that utilises the stream X-machine formalism.

Part B – Specifying and Testing Stateful Web Services

- Chapter 4 – *Web Services with State and Testing Implications*
- Chapter 5 – *Modelling Stateful Web Services with Stream X-Machines*
- Chapter 6 – *Notation and Examples*
- Chapter 7 – *Testing Web Services Modelled as Stream X-Machines*

Chapter 4 – Web Services with State and Testing Implications

Frequently, Web services operate on internal state (or data), which affects and is affected by the execution of the service operations. The presence of state and its characteristics, have major impact on the functional behaviour and complexity of a Web service. The extra complexity introduced by state brings about further challenges to specifying Web service behaviour as well as testing the implementation against the specified behaviour. Therefore, it is important to clarify what Web service state is, what are its characteristics, how it is implemented, and how it affects the tasks of modelling and testing. This is the topic of this chapter, which is divided into four main parts.

The first part clarifies the term *stateful*, referring to Web services, as commonly used in the literature, and as adopted in this thesis. The next section aims to define what exactly Web service state is, what is Web service behaviour, and how the former affects the latter. Since state is highly heterogeneous and takes on various forms in service implementations, the third section identifies the characteristics of state and its effects on behaviour. It describes state scope and identification, state duration, and its variation along other dimensions. The fourth section provides a categorisation of Web services with respect to state and behaviour, and investigates testing implications for the different categories. The fifth section gives a practical flavour, where the different techniques for implementing stateful Web services are investigated, in some of the prevailing Web service frameworks. The final section contains some closing remarks regarding the occurrence of stateful Web services in the real world.

4.1 Stateless versus stateful Web services

Service *statelessness* is often regarded as a good service orientation principle, which promotes service reusability, composability and scalability [8], [75], [21]. According to T. Erl [8], services should minimize the amount of state information they manage and the duration for which they hold it. Upon the completion of each operation, the service should not have to remember any local state information for

processing the subsequent requests. This stateless model differs from the one adopted in object-oriented programming, where objects remember their state in the form of attributes and the results of method calls depend on previous calls.

Nevertheless, in certain situations it is useful for a Web service to remember the communication status in conversations that involve multiple steps. For example, after a client is authenticated to a banking Web service, it can proceed with bank transactions in subsequent requests without having to resend the credentials, since the service remembers the client. This is usually achieved by means of sessions (described later on) which are managed automatically, allowing the Web service to simulate a single-user interactive system. Generally, the Web services community uses the term "*stateful*" to refer to this kind of Web services, i.e. services which manage sessions to keep state specific to a current conversation [8][75]. Web services with sessions raise reliability and scalability concerns: resetting the session, restarting the service following a failure, or creating new service copies for load balancing should take into account the previous history of invocations.

The need to maintain sessions can be avoided if the client identifies itself with every request message sent to the service. Nevertheless, the service still has to maintain internal state for that client, which is accessed based on the supplied identification. For example, even though clients of a banking Web service supply their credentials with every request message, the service has to keep track of the bank account information for each client. Therefore, such a Web service can be also considered as stateful, even though it does not manage sessions. Finally, services that delegate the responsibility for the management of state to another component such as a file system or database (stateful resources [21]) are fairly common. Although those services are conventionally regarded as stateless, they still keep state that persists between service invocations. Therefore, the range of Web services storing some form of data, or state, which persists between operation invocations, is much broader.

Consequently, in this thesis, the definition of stateful services is generalised to encompass any services that maintain some form of state, rather merely services that maintain sessions. The state, along with the provided inputs, affects the outcomes of operation calls. *Therefore, while in a stateless service the response of any operation depends solely on the provided input, in a stateful service, the response of an operation depends not only on the input but also on the service state*.

While part of the data used by a service operation may come from state, the rest is supplied by the client with the request message. According to T. Erl [8], as more information is included in a request message (encouraged by document-style messaging), dependence on state information is reduced, and thus statelessness is supported. However, as further explained in section 4.5, the client cannot pass all the state information in the contents of request messages, since certain information must be stored by the service in any case. Consequently, it is not always possible to avoid the need for implementing stateful services.

4.2 Web service state and behaviour

Non-trivial Web services are expected to operate on some form of internal data, or state information. This state eventually affects the responses returned from the invocation of service operations. That is, the final result returned from a service operation depends not only on the input (request message) but also on the state the service is found in.

The relevance of service state in this section is that it significantly affects service behaviour and complexity. Web services exhibiting state are more challenging to test, and need to be modelled beyond their WSDL interfaces. Also the nature of state and the category of the Web service with respect to state (see section 4.3.5) have implications on the suitable modelling approach, as well as the testing strategy employed for ensuring correct behaviour.

4.2.1 Web service state

As will be discussed later in this chapter, Web service state exhibits a high level of heterogeneity over several dimensions. Thus, it is important to seek an umbrella definition that encompasses all forms of state.

Web service state can be persistent in the form of a file and stored indefinitely, or it can be volatile and pertaining to a single multiple-step transaction. State can be in the form of HTTP session variables; variables maintained by service instances serving separate clients; context and configuration information kept by the Web service platform or application server; a file on the server's secondary storage; or even a whole database. Additionally, state can be maintained for and accessed by a single client, or shared among several clients invoking the service.

Amongst all this heterogeneity, it is possible to observe one common characteristic of service state: in all cases it is *information* (or *data*) accessed by the Web service. State information is stored, read, modified and/or deleted by the service operations. If any information is maintained by the service requestor, then it is *not* considered as state, since it plays no role on the service behaviour, unless it is supplied with the inputs. For example, HTTP cookies are files stored on the client machine, thus they are not considered as state, although cookies do serve to *identify* state stored on the service machine (see section 4.3.4 – state identification).

The other common characteristic of service state is that, as data, it can persist between one operation invocation and the next. Even HTTP sessions, whose durations are among the shortest (section 4.3.1), are able span the invocations of several operations in sequences. If, instead, a Web service does not support sessions, then any memory variables maintained by its implementation are lost with the next invocation, since a fresh service instance is spawned by the infrastructure. This is generally due to the stateless nature of Web service platforms, such as

Axis2[7], which by default do not remember the state of conversations for scalability reasons [75]. In all situations, the data that pertains only to the current operation invocation is not regarded as state.

4.2.2 Web service behaviour

Web service behaviour and state are closely related to each other, thus it is appropriate at this point to define what is meant by behaviour in this thesis. The term behaviour is restricted to encompass only the *functional* characteristics of a Web service, excluding any other non-functional (QoS) characteristics, such as performance, availability, robustness, and stress-tolerance.

The functional behaviour manifested by a Web service may be seen from two different perspectives. One perspective is the imposed sequencing of operations for successful interaction with the service, also called the *explicit choreography (protocol)* of the service in [30], and alternatively referred to as the conversation protocol in this thesis. For instance, an order management Web service may require a requestor to authenticate with the login operation prior to creating an order with the createOrder operation, after which items can be added or removed with the addItem and removeItem operations, respectively. Moreover, an order quotation has to be first requested and then confirmed, through the invocation of the corresponding service operations, before the order is finalised.

The other perspective on service behaviour is the computation logic of individual service operations to produce outputs in response to inputs (IO), as well as pre- and postconditions (effects) on the internal state (PE). This collective viewpoint of inputs, outputs, preconditions, and effects (commonly known as IOPEs) is also called the *implicit choreography (protocol)* of the service. For example, consider an operation of the order management Web service, which allows adding a new item to the current order. The operation takes as *inputs* the identifier and quantity of the desired item; with *precondition* that the order is in a manipulation status and the item is available in the inventory for the requested quantity; with *postconditions* that the requested quantity has been added to the current order and subtracted from the items inventory; and finally, producing an *output* indicating success and reminding of the item identifier and added quantity.

The two views on service behaviour are interrelated. The preconditions for successfully invoking an operation, and the effects produced by its execution, determine how the operation can be placed in sequences of invocations. For example, the addItem operation described above requires the order to be in manipulation status, thus it cannot be successfully invoked if the order has been cancelled with the cancelOrder operation, since the effect of the latter sets the order to a cancelled status. Therefore, the explicit service choreography is dependent on implicit choreography, and vice versa. As demonstrated in section 5.6, starting with

[7] Apache Axis2 (http://axis.apache.org/axis2/java/core/) is an open source Web service platform developed by the Apache Software Foundation.

pre- and postconditions of individual Web service operations in form of IOPEs, it is possible to derive the control states and transitions of a stream X-machine, which are a representation of the explicit choreography of the service.

4.2.3 Effect of state on Web service behaviour

In non-trivial Web services, the output (response message) returned by an operation is often not based solely on the provided input (request message). Other factors come into play, such as the state maintained by the Web service, as well as other nondeterministic factors.

In short, the behaviour exhibited by a Web service may depend on:

- The request message (or input) contents;
- The state information (data) maintained by the Web service;
- Nondeterministic factors:
    - Unknown state information,
    - Timing constraints,
    - Back-end applications updating the state information,
    - Human and manual factors,
    - Reliance on third-party Web services, etc.

In this thesis we aim to model the first two factors on service behaviour: *inputs* and *state information*, while specifying the processing logic that from inputs and service state computes outputs and modifies the state. I.e. state may be checked by the precondition predicates of Web service operations, whose execution, in turn, may modify that state information (postconditions). These IOPE characteristics also affect the rules for correct sequencing of service operations.

When additional factors, such as the ones listed above, affect the outputs (and hence behaviour), then the observed behaviour of the WSUT will appear to be nondeterministic. Those factors are not specified in the modelling approach described in this thesis and the returned outputs are determined nondeterministically. Refer to section 5.8 for a more detailed description of nondeterministic services and nondeterministic specifications.

In the light of the above discussion on state and behaviour we can make a note on the suitability of the stream X-machine computational model for specifying stateful Web service behaviour. SXMs combine behavioural modelling with data modelling, and decouple computation blocks, known as processing functions, from the high-level specification of the control flow. Therefore, in this thesis we consider SXMs as highly suitable for formally explicating the different facets of service behaviour and the internal state data. Control flow is modelled by abstract SXM states and transitions between those states. Data, on the other hand, is modelled by the SXM memory construct for state data, and by inputs and outputs declarations for service input and output data. Furthermore, control flow transitions are labelled by processing functions (instead of simple inputs), which allow expressing the

computation logic of operations in terms of preconditions (processing function domain) and effects (processing function memory update). Section 5.2 presents a more comprehensive analysis on the appropriateness of SXMs as models of stateful Web services, especially as compared to other, simpler, state-based formalisms, such as finite state machines, and extended finite state machines.

4.3 Characteristics of Web service state

4.3.1 State accessibility

State information managed by a Web service can be accessible by operation invocations from a single client, from several clients, or from other applications or third-party Web services. Based on these levels of accessibility, state can be categorised into *private* and *shared* (or *global*). As the term suggests, private state is stored for an individual client, and therefore is accessible only by operation calls from that client. It is not accessible by other clients, applications, or Web services. This implies that the value of private state at any instant is determined exclusively by the history of previous interactions between the specific client and the Web service. The contents of the shopping cart in an e-commerce Web service are an example of private state.

In contrast, shared state is shared among all clients, and possibly other applications or Web services. That is, it can be accessed and possibly modified by operation invocations from any client. This implies that the value of shared state may change over time by invocations from other participants, and is not determined exclusively by the previous conversation history of any specific client. As an example, the inventory of available items in the above e-commerce Web service is an example of shared state. It is accessed and potentially modified as other clients purchase items. Also, the inventory is possibly modified by applications that manage its contents, e.g. when new supplies arrive in the warehouse.

4.3.2 State duration

The lifetime, ore duration, of state entities in a Web service implementation affects the way the test sequences are executed on the implementation. It is especially important to know whether state entities live within single sessions, within single server instances, or span multiple server instances. In addition, it is necessary to know how to reset the state to its initial value (see section 7.6.4).

Duration of state is determined by the form it takes (e.g. whether it is session data, configuration information, file etc.) and by the platform-specific details of its implementation in the Web service. For example, in the Apache Axis2 Web service platform, duration of state stored in the context hierarchy is determined by the value of the “scope” attribute in the service descriptor file (section 4.4).

Overall, in terms of duration, we can distinguish state into *volatile* and *persistent*. This distinction is not necessarily determined by whether state variables are stored

in the volatile memory or in the persistent storage. While data in the volatile memory could persist for the whole duration of the server instance, data in the persistent storage could be deleted as soon as a service operation completes execution. Here we define the distinction between volatile and persistent state by whether it can span several sessions (SOAP or HTTP sessions). Volatile state lives within a single session and is erased as soon as the session ends, while persistent state outlives sessions.

Figure 7 compares the durations of different forms of state to one another. *Persistent objects* have the longest durations of all. Some types of persistent objects do not survive a server or system crash or restart. Such persistent data, for example, can be part of the configuration information pertaining to an instance of the application server or the Web service platform. Other persistent data, such as data stored in the secondary storage, is not lost when the server or system is shut down. The lifetime of such a persistent object starts upon its creation and ends with its deletion, by means of invocations of the appropriate CRUD (Create-Read-Update-Delete) service operations. On the other hand, SOAP and HTTP session state information is created by the server upon a session start, and is deleted upon the session end. Both types of sessions can span several operation calls and store context information pertaining to complete operation sequences (transactions). A SOAP session may span more than one HTTP session, but not vice versa. Observe that no type of session can survive a client shut down or restart, thus causing the session data to be lost.

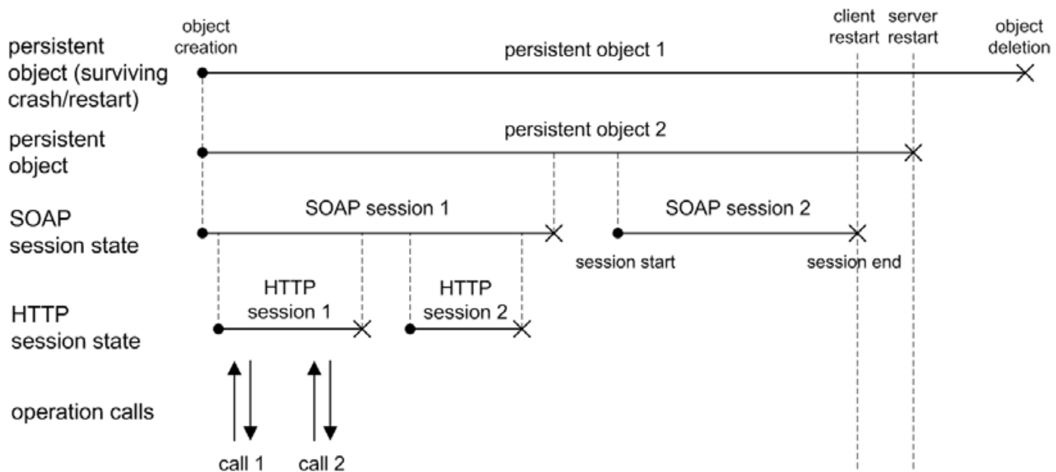


Figure 7 - State duration for different types of state

4.3.3 Views on private state

Web services are usually designed to be accessed by several clients (users) simultaneously. To address the concurrency, stateful Web services maintain state data separately for each client, which in turn may be structured into several stateful objects (instances), as shown in Figure 8. The state allocated to any client is private and is not meant to be accessed by the other clients using the Web service

simultaneously. Therefore, the state *scope* of service operations invoked by a particular client is restricted to that client's private portion of the complete Web service state. From the perspective of the particular client, the behaviour of the Web service is affected only by the private state. The rest of the service state, which is allocated to the other clients, is invisible, thus creating the illusion of isolation. Therefore, we distinguish between two views on Web service state and behaviour, referred to as the *per-client* and *pan-client* views, terms adopted from Atkinson et al [72].

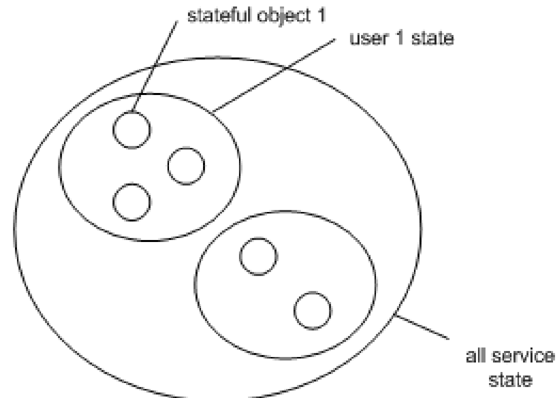


Figure 8 – Structure of internal state typically maintained by a multi-user Web service

As an example, consider an order management Web service where every service client is allowed to create and place one order at a time. In this case an individual order instance being manipulated constitutes the per-client view, while the collection of all orders being manipulated by all clients of the service constitutes the pan-client view. To an individual client the outcomes of the service operations are affected only by the status and the items on the specific order being placed, regardless of the other orders belonging to the rest of the clients.

Next, consider an extended order management service where every client is allowed to manipulate several orders at a time. The per-client view now becomes a collection of order instances. Nevertheless, it is possible to abstract away a conversation encompassing only the operation calls referring to one order object. To such a conversation the data in the other order objects is irrelevant, and its behaviour is affected solely by the specific order object. Hence, in addition to the two views defined earlier, a third, *per-object*, view is introduced in this thesis.

As will be seen later on, adopting the per-client and per-object abstractions vastly simplifies the behavioural specification of the service as well as the testing process.

4.3.4 Private state identification

In the previous section it was stated that operations invoked by a particular client are allowed to access only the private state allocated to that client. Moreover, a group of operation calls referring to a single object are affected by and affect only

the data for that object. However, given the stateless nature of the HTTP (or other) protocol used to transfer messages between the client and the service, a mechanism is necessary to correlate those messages into conversations, as well as to tell the service the target of those messages. Without such a mechanism the Web service cannot determine which stateful entity to access and modify.

The mechanism for informing the Web service of the client or the stateful object being targeted is referred to as *state identification*. The client supplies one or more unique identifiers with every request message, which associate the request with the corresponding client and/or target stateful object. The invoked operation uses the provided identifiers to reduce its scope from the complete service state to a fraction of it. As depicted in Figure 9, client identification enables the per-client view, while object identification enables the per-object view.

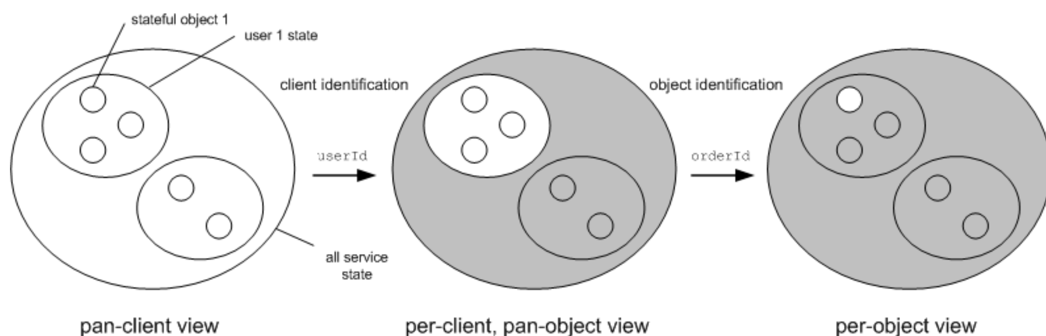


Figure 9 - State identification by a client filters the state that is accessible by operation calls. Identification can also be performed in steps: client identification, and then object identification.

Identifiers are also used by the Web service implementation to correlate messages into conversations (execution contexts). To an external observer with the pan-client perspective, the messages from different clients arriving to the service are interleaved. There seem to be no conversations or sequencing of operations according to any protocol, but random invocations. However, if messages are grouped by their IDs, then the per-client (or per-object) view is adopted and individual conversations emerge.

It is important to examine how state identification is performed in practice, since the identification information must eventually be part of every input sent to the service during test execution. If the service model adopts an abstracted per-client or per-object view, then the state identification is not captured in the model and it does not appear in the generated abstract inputs. The options for inserting identifiers during concretisation of abstract inputs to request messages are discussed in section 0 – Bridging the abstraction gap.

Obtaining identification information

There are two key scenarios the service requestor knows the identification information to supply with every request:

(a) known in advance, and

(b) retrieved from the Web service at run time.

The first case is simplest to address, where it is possible to derive concrete test inputs with identifiers during test case generation and before test case execution (the Constant Field pattern, section 9.4.1). In the second scenario client and/or object identifiers are initially obtained from the Web service during run time. As a result, in this second case it is not possible to fully concretise test inputs during test generation time, since the identifier information is unavailable. The derivation of concrete test inputs is deferred until run time when the tests are executed.

A client that obtains identifiers from the Web service at run time expects to find them in a response message after a specific event. Two such common events are:

(a) the creation of a stateful entity at the beginning of a conversation, and

(b) substitution of previous identifiers with new identifiers.

In the first case, conversation starts with the creation of a new stateful entity by the Web service and the return of the entity's identifier in a response message. After the client retrieves the identifier from the service, it adds the identifier to every subsequent request message to associate the conversation with the newly-created stateful entity. For example, consider the previous Web service, which manages (creates, reads, updates, and deletes) several supply order instances. Invoking the create operation instantiates a new order and returns its unique orderID. That orderID is then repeated in the following request messages to operations manipulating the specific order. In the end, the delete operation supplied with the orderID deletes the order instance and concludes the conversation. Another identification mechanism that works according to this scheme is that of sessions, as described further below.

An example of the second case, where identifiers are substituted by new ones, is the invocation of a user authentication operation. During the execution of that operation the username and password identifiers are substituted by an authentication token, which is returned by the Web service. Alternatively the username and password can be substituted by a session identifier. It is noteworthy that although the identification information changes over time, it is associated with exactly the same stateful entity.

Location of identification information

Technically, state identification information may be inserted in different places in service requests and responses, depending on the identification mechanism being used. In this thesis we pinpoint three different locations for identifiers, as shown in Figure 10:

1. the header of HTTP (or other protocol) messages carrying SOAP envelopes,
2. the header portion of SOAP envelopes, and

3. the body portion of SOAP envelopes.

Recall from section 2.2.1 that SOAP envelopes consist of a SOAP header and a SOAP body, and are carried over a transport protocol, most commonly HTTP.

Figure 10 – Identification information can be supplied in three different layers of service requests

Usually, *HTTP headers* are used in HTTP sessions to carry session identifiers that group operation invocations within the same session. *SOAP headers* are utilised by second-generation WS-* protocols, such as WS-Addressing (see below) to carry protocol-related information, including identifiers. Identifiers in HTTP and SOAP headers are usually hidden from the business logic and are accessed only by the infrastructure that implements the particular WS-* protocol. This implies that HTTP and SOAP header identification is handled automatically and the service client implementation does not have to worry about the details of retrieving and adding identifiers.

The third case, *SOAP body* identification, operates at the business logic level. The identifiers are part of the XML information exchanged in SOAP body payloads. For example, when invoking operations of the order management Web service on a specific order instance, each request message includes the orderID identifier as a child of the root element of the body payload (Figure 10). Note that the identifier element appears in arbitrary nodes and nesting levels in the DOM tree of the XML document, both in request and response messages. This implies that the service client implementation has to know where exactly to locate an identifier in a response message, and where to insert it in the body of a request message. The location of the identifier element does not follow any conventions as in the case of sessions, but is inferred by human individuals from the WSDL document or the informal service documentation. Nevertheless, if some recognisable pattern is followed, it may be possible to automate SOAP body identification as well. Two

such patterns (the Manager and the Constant Field pattern) are introduced in section 9.4, to allow automated execution of test cases with identifiers.

The next two subsections illustrate the discussed concepts on private state storage and identification with the case study of sessions: HTTP sessions and SOAP sessions.

HTTP sessions

The underlying transport protocols, such as TCP, HTTP, and SMTP, specified by the bindings to carry the SOAP messages are *stateless* in nature, i.e. they do not remember any previous communication. The most prevalent of those protocols in Web services is the HTTP (HyperText Tranfer Protocol), which is the standard for transporting documents on the World Wide Web [73]. HTTP supports a request-response model of transferring data between a client and a service and does not correlate requests from the same client.

It is through a popular mechanism combining HTTP *sessions* and *cookies* that HTTP communication can be made stateful. This mechanism addresses the problem of *storing state information* between one request and the next, as well as *the identification problem*, for relating requests from the same client with that state information. While HTTP sessions store bits of data on the server, HTTP cookies store bits of data on the client machine.

As Figure 11 shows, state is maintained by the service in the form of *session variables*, which can contain any information pertaining to a conversation with a client, such as the contents of a shopping cart. The identification of that state on the client part is solved through cookies, which are stored automatically by the client infrastructure whenever instructed by the server in the HTTP header of a response. The data stored in cookies can include *session identifiers*, which are then automatically supplied in the HTTP header of every request.

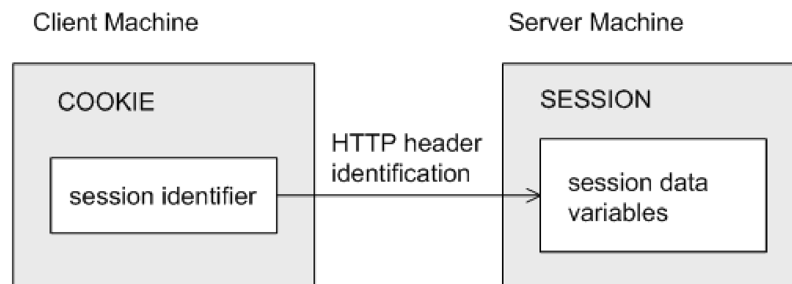


Figure 11 - Sessions store state in the server machine, while cookies, stored in the client machine, identify that state

The following monitoring logs illustrate the stateful conversation between a client and an Axis2 Web service. The contents of the HTTP headers are exposed in addition to the contents of the SOAP envelopes. In the following response message, the service instructs the client to store a session identifier (JSESSIONID) in a cookie, marking the start of the session.

```
==== Response ====
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=2293BF8E54A3B13ED4CFACD8C235177B; Path=/axis2
Content-Type: application/soap+xml; action="urn:openResponse";charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 17 Nov 2010 16:53:14 GMT

107
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Body>
  <ns:openResponse xmlns:ns="http://ws.apache.org/axis2">
       <ns:return>openOut</ns:return>
  </ns:openResponse>
</soapenv:Body>
</soapenv:Envelope>
0
==============
```

The client stores the identifier in a cookie and repeats the contents of that cookie in the HTTP header of every subsequent request for the length of the session:

```
==============
Listen Port: 8888
Target Host: 127.0.0.1
Target Port: 8080
==== Request ====
POST /axis2/services/Account?wsdl HTTP/1.1
Content-Type: application/soap+xml; charset=UTF-8; action="urn:deposit"
Cookie: JSESSIONID=2293BF8E54A3B13ED4CFACD8C235177B; Path=/axis2
User-Agent: Axis2
Host: 127.0.0.1:8888
Transfer-Encoding: chunked

101
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Body>
  <axis:deposit xmlns:axis="http://ws.apache.org/axis2">
       <axis:param0>5</axis:param0>
  </axis:deposit>
</soapenv:Body>
</soapenv:Envelope>
0
```

SOAP sessions

By design, the SOAP protocol is stateless and one-way, to support loosely-coupled applications that interact by exchanging asynchronous messages with each other. As a result, operation message exchange patterns (MEP) and complete stateful conversations have to be implemented by the underlying system. Similarly to HTTP as explained above, it is possible for SOAP envelopes to simulate stateful conversations by carrying identification information in their headers. One common WS-* protocol that is utilised to for this purpose is WS-Addressing [74]. The following snippets show that a similar approach to HTTP sessions is followed, except that this time the information is put in a different location.

The SOAP envelope returned by the first operation invoked on the Web service within a SOAP session instructs the WS-Addressing-enabled client to repeat the contents of ReferenceParameters, which include the session identifier:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">

  <wsa:ReplyTo>
       <wsa:Address>http://www.w3.org/2005/08/addressing/none</wsa:Address>
       <wsa:ReferenceParameters>
              <axis2:ServiceGroupId
                     xmlns:axis2="http://ws.apache.org/namespaces/axis2">
                     urn:uuid:B9AB09FCC14882B1521230369826635
              </axis2:ServiceGroupId>
       </wsa:ReferenceParameters>
  </wsa:ReplyTo>

  <wsa:MessageID>urn:uuid:B9AB09FCC14882B1521230369826637</wsa:MessageID>
  <wsa:Action>urn:getCountResponse</wsa:Action>

  <wsa:RelatesTo>urn:uuid:80CBCC10EFFA51034F1230369826309</wsa:RelatesTo>
</soapenv:Header>

<soapenv:Body>
  <ns:getCountResponse xmlns:ns="http://service.session.sample">
       <ns:return>1</ns:return>
  </ns:getCountResponse>
</soapenv:Body>
</soapenv:Envelope>
```

For subsequent requests, the WS-Addressing-enabled client includes the identifier in the SOAP header.

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">

  <axis2:ServiceGroupId
       xmlns:axis2="http://ws.apache.org/namespaces/axis2"
       wsa:IsReferenceParameter="true">
       urn:uuid:B9AB09FCC14882B1521230369826635
  </axis2:ServiceGroupId>

  <wsa:To>
http://localhost:8088/axis2/services/SampleSessionService.SampleSessionServ
iceHttpSoap12Endpoint/
  </wsa:To>

  <wsa:MessageID>urn:uuid:80CBCC10EFFA51034F1230369826738</wsa:MessageID>
  <wsa:Action>urn:getCount&it;/wsa:Action>

</soapenv:Header>
<soapenv:Body/>
</soapenv:Envelope>
```

4.3.5 Classification of Web service state

As described in the previous sections, Web service state is highly heterogeneous and is characterised in a number of ways. Characteristics of Web service state affect the approach taken to modelling and testing Web services, as will be discussed in other parts of this thesis. In an attempt to categorise Web service state and resultant behaviour, Figure 12 identifies five different dimensions along which state can

vary, on the basis of the discussion from the previous sections. These dimensions are projected as different axes and consist of: state implementation (non-exhaustive), state duration, state accessibility, and state identification method, regarding both identifier location and identifier availability.

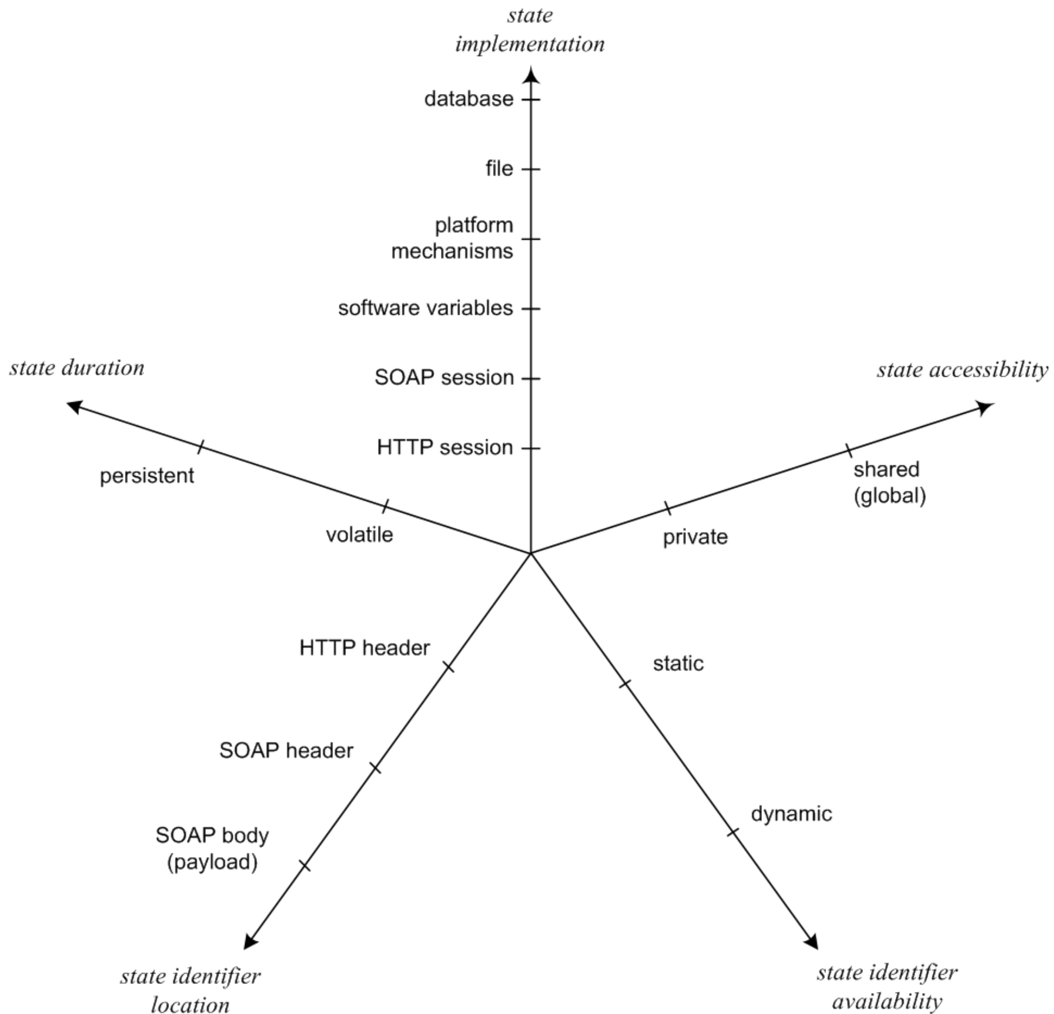


Figure 12 - Web service state is heterogeneous and varies along several dimensions, projected as axes in a five-dimensional space

The above identified dimensions are not always mutually orthogonal since the different characteristics may depend on one another. For example, the implementation form a service takes determines most of the other characteristics. State implemented with HTTP sessions, SOAP sessions, or software variables is generally private and volatile, while identifiers are dynamically obtained from the server and located in the respective headers. In contrast, state implemented in the form of files or in databases can be both private and shared, is usually persistent, and identification takes place in the SOAP body (business logic). Private state can be both volatile and persistent and is identified in different forms. Also, the different perspectives (per-object, per-client, and pan-client) pertain only to private state. On the contrary, shared state is always persistent (there is no concept of

sessions) and is usually not identified. Therefore, although a large number of combinations result from the five dimensions of variation, practically only a few of them are meaningful.

4.3.6 Classification of Web services based on state

First of all, from a high-level perspective, Web services can be categorised into stateless and stateful ones, depending on whether they exhibit any observable state. Recall that in a stateless service the response of any operation depends solely on the provided input; the same result is delivered for the same input every time the operation is invoked (e.g. a simple Web service converting temperatures between Fahrenheit and Celsius). In contrast, in a stateful service, the response of an operation depends not only on the input arguments but also on the internal state of the service. As a result, attempting to specify and test the behaviour of a stateful Web service should take into account its state. Since the work described in this thesis focuses on stateful services, a further classification is attempted for this category.

The types of service state described in the previous section can be taken as a basis for deriving a few meaningful stateful Web service categories. In addition, Web services can also be distinguished according to the *behaviour* that emerges from their state, rather than from state alone. One such important distinction is into non-conversational and conversational:

- In a *non-conversational service* all operations are successfully accepted at all states without producing any errors. From an explicit choreography perspective (see above), this means that the service does not impose a conversation protocol and accepts any sequences of operations. From an implicit choreography perspective (IOPE), the success scenarios of operations do not have any preconditions on Web service state. The testing implication for non-conversational services is that the behaviour of individual operations can be tested in isolation from other operations. An example of a non-conversational stateful service is a currency converter Web service: although its operations access a database of exchange rates, they can be invoked in any sequences.
- In a *conversational service* only specific operation sequences are successfully accepted. Therefore, a conversational service imposes a conversation protocol that consists of the set of all acceptable operation sequences. From an implicit choreography perspective, some operations have preconditions on Web service state for successful completion. The testing implication for conversational services is that it is not sufficient to test individual operations in isolation, but as part of sequences of invocations. A shopping cart Web service is a typical example of a conversational stateful service: items cannot be removed from the cart if they were not added in the cart in a previous step.

Secondly, an important distinction between stateful Web services is whether the state they access is private, shared, or both. Therefore, stateful services can be further categorised depending on state accessibility:

- In a *private-state service* all accessed state is private. Thus, to a client, the behaviour of the service depends only on state which is determined exclusively by the interactions among the client and the service. An example of a private-state service is a shopping-cart Web service whose behaviour is affected solely by the contents of the internal shopping cart.
- In a *shared-state service* some or all of the accessed state is shared. Thus, the state of the service cannot be fully determined by the sequences of previous service invocations. The behaviour of the service depends on some state variables which may be modified by invocations from other clients, Web services, or applications. An example of a shared-state service is a shopping cart Web service whose behaviour also depends on inventory information for stock levels. Although the shopping cart itself is private to a single service client, the presence of a shared inventory classifies the Web service as a shared-state one.

As will be seen later in chapter 0, shared-state services introduce other challenges to service specification and testing. Since part of the state is shared among a number of clients, it can potentially attain huge sizes and may be difficult to model. Furthermore, since shared state can be modified by other, unknown, clients or applications, the resulting behaviour of the service is nondeterministic.

While shared state is persistent, private state can be both volatile and persistent. Thus, it is possible to further split private-state services according to state durability:

- In a *volatile-state service* part of the state information persists only for one session. The next time the client starts a conversation with the service the context information relating to the previous conversation is lost. An example is a shopping cart Web service that stores cart information in session variables. If a new session is initiated or the service is restarted, conversation has to start all over again with an empty shopping cart.
- In a *persistent-state service* all of the state information is persistent and outlives sessions. Therefore, the next time a conversation is started with the service no context information relating to the previous conversation is lost. An example is a shopping cart Web service that stores cart information on a database. If the current conversation is disrupted by a session end or service restart, the client can still continue shopping with the original cart stored in the database.

As will be discussed later on in this thesis, state durability details are not usually captured in a service model. The modelled state is assumed to persist indefinitely. However, state persistence is relevant during testing. The tester has to know whether the next time a sequence of operation invocations is exercised, the state in

the service implementation is automatically reset to its initial values (volatile-state services) or special techniques are required to perform the reset (persistent-state services). More details are given in chapter 0.

4.4 Implementation of stateful Web services

The preceding sections described stateful services, the variations of state, and a classification of services according to state characteristics. This section proceeds with a brief technical overview on implementation of state in Web services. Techniques are described for implementing stateful Web services in four representative Web service platforms: *Apache Axis2*[8], *JAX-WS*[9], *Oracle Weblogic*[10], and *IBM Websphere*[11]. More focus is given to Apache Axis2, which will be used as a basis for development of the testing tool.

4.4.1 Stateful Web services in Apache Axis2

Apache Axis2 provides specific mechanisms to persist state between one operation invocation and the next. State is stored in what is called the *context hierarchy*, illustrated by the diagram in Figure 13 [75]. The Axis2 engine stores contextual information in this hierarchy, starting from Message context, going up to Operation context, Service context, ServiceGroup context, and finally Configuration context. Message context is information that pertains to a single request or response message. Similarly, Operation context refers to one operation, Service context refers to one Web service, ServiceGroup context refers to one service group, and Configuration context refers to the whole Axis2 application, hence to all services.

[8] http://axis.apache.org/axis2/java/core/
[9] http://jax-ws.java.net/
[10] http://www.oracle.com/technetwork/middleware/weblogic/
[11] http://www-01.ibm.com/software/websphere/

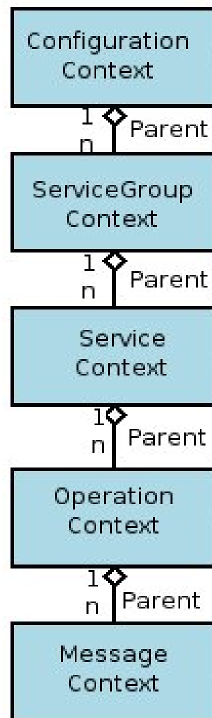


Figure 13 - The Axis2 context hierarchy [75]

Moreover, in Axis2 the duration (or *scope*) of all these context objects is controlled in the service configuration file through the "scope" attribute. The context objects can have *request* scope, *transport* (i.e. HTTP) *session* scope, *SOAP session* scope, and *application* scope. With request scope the context objects live for only one request, thus it defines stateless services. With transport and SOAP session scopes, the context objects live as long as the HTTP and SOAP session, respectively. Hence, this scope defines stateful, conversational Web services. Finally, in application scope, the context objects in the hierarchy persist as long as the Axis2 executable instance, thus they span several sessions and are shared by multiple clients. Therefore, this scope defines shared-state Web services.

By default, the Axis2 platform spawns a new instance of the implementation class (server instance) to serve a request message and, upon completion of the operation, the software object is destroyed. Since no session is maintained, the next time a request message is received, a different instance is created, even if the request is from the same client. On the other hand, if sessions are specified with the transport and SOAP session scopes, the service associates clients with implementation objects. Subsequent requests from the same client are dispatched to the methods of the same object, thus context state persists in the object attributes.

It can be observed that the hierarchical level of the context object and its scope are not orthogonal to each other. The scope attribute only affects the duration of the Service and ServiceGroup context objects. Regardless of the specified scope, Message and Operation context objects pertain to individual SOAP messages and operations respectively, thus they do not have to span more than one operation

invocation. On the other hand, the Configuration context pertains to the whole service platform and persists as long as the Axis2 executable instance, regardless of the specified scope.

4.4.2 Stateful Web services in JAX-WS

JAX-WS is the Sun technology for building Web services, and defines a set of APIs for Java starting from version 1.5.

JAX-WS provides interfaces and methods for implementing stateful Web services that manage HTTP sessions [76]. The service implementation class is required to implement the `ServiceLifecycle` interface. This interface defines two methods: `init` and `destroy`. In the implementation of the init method, upon the start of a session, the infrastructure passes as parameter an instance of `ServletEndpointContext`, where session variables are stored. The rest of the methods that implement corresponding operations have access to this `ServletEndpointContext` instance and retrieve the HTTP session object through invocation of method `getHttpSession()` on that instance. The returned `HTTPSession` object consists of key-value pairs that constitute the session data.

The implementation class is then packaged as a Web Application Archive (war) file representing the service, and deployed in the Glassfish[12] application server.

4.4.3 Stateful Web services in Oracle Weblogic

Like the previous Web service platforms, Oracle Weblogic allows implementation of Web services with sessions [77]. Such services are coded with the aid of special Java annotations such as `@Conversational`, `@Context`, and `@Conversation`. The state duration is defined by a life cycle containing phases START, CONTINUE, and FINISH, through overriding the start, middle, and finish methods in the implementation class.

Alternatively, stateful (conversational) Web services can be implemented without sessions. Instead, conversations can be mimicked by allowing clients to supply unique identifiers in every request message (as described in section 4.3.4 on state identification). Weblogic provides mechanisms to implement state through mechanism such as database connectivity and entity beans, while state identification is performed at the SOAP body layer [78].

4.4.4 Stateful Web services in IBM WebSphere studio

IBM WebSphere allows development of both stateful services with sessions, and stateful services operating on persistent data [79]. For the second type of services, WebSphere provides mechanisms to implement WS-Resource services via what is known as the *Common Information Model* (*CIM*). [80]

---

[12] http://glassfish.java.net/

4.5 Prevalence of stateful Web services

Although it is possible to use stream X-machines to model and test both stateful and stateless Web services, the SXM testing method is especially useful when applied to stateful Web services. Thus, it is relevant to consider how frequent stateful Web services are in the real world, either in private SOA deployments within organisational boundaries or publicly over the Internet. It is reasonable to expect that non-trivial Web services have to operate on internal data of some sort. From our experience, services that support sessions (conventional definition of stateful services) are relatively rare in the real world. This is mainly due to the scalability concerns mentioned earlier. Nevertheless, services operating on persistent data are fairly ubiquitous, including services from Amazon, Google, UPS, Paypal, etc. A considerable number of these Web services store data separately for each client (private-state services) and moreover they assume a conversation protocol (conversational services).

There are cases when the responsibility for maintaining state can be handed over from the Web service to the service requestor. In those cases implementing the Web service as stateful can be avoided. The requestor managing the contents of state data can supply that information to the service with every request message, so that the service can remain stateless. For example, a shopping cart Web service is stateful if it maintains shopping carts for clients, to keep track of the items to be purchased. But alternatively, the shopping cart contents can be managed by the client application as the user adds shopping items to the cart. Then, upon checkout, the shopping cart contents are passed to the stateless Web service.[13]

Nevertheless, delegating state maintenance to the service requestor is not always feasible. The client cannot be entrusted with maintaining sensitive data, which could be accidentally or purposefully corrupted. For example, the client of a banking Web service should not be allowed to keep track of the status of its bank account and the remaining balance, as this is business-critical information. In addition, stateful Web services have the responsibility for ensuring correct behaviour and enforcing a conversation protocol for the successful completion of transactions. For instance, when ordering raw materials from a supplier, a supply order is defined in accordance with a multi-step protocol. Requiring the client to remind the Web service, with every request message, about the current step of the transaction, can potentially result in violation of the protocol.

4.6 Summary

This chapter presented an in-depth investigation of stateful services, i.e. services that maintain state between operation invocations. The different characteristics of service state, such as state scope, state identification and sate duration, were

[13] In many real-world shopping cart Web services, such as the Amazon E-commerce Service [56], shopping cart contents are indeed managed by the (stateful) service rather than the client, for more convenience.

identified along with their variation. Also, a classification framework for Web services with respect to state was presented. This classification framework gives rise to a few practical Web service categories, which have distinct requirements for formal modelling and testing. The results from this chapter, which relate to contribution C1 of this thesis (see section 1.3), will be referred to in the subsequent chapters, where ad hoc modelling and testing techniques are described for Web services with different state characteristics.

The next two chapters present challenges and techniques for modelling Web services using the stream X-machine formalism.

Chapter 5 – Modelling Stateful Web Services with Stream X-Machines

The core activity of the testing approach described in this thesis is the creation of a formal stream X-machine (SXM) specification of the Web service to be tested. Having investigated and classified stateful Web services in the previous chapter, this chapter describes how stateful Web services are modelled by SXMs. Creating a SXM specification requires modelling techniques as well as an expressive language to write the specification, possibly supported by editor tools.

Before starting the main discussion, this chapter introduces two Web service examples, which are used in the rest of the chapter and thesis for illustration purposes. It follows with a comparison of three representative state-based computational models, which are FSMs, EFSMs and SXMs. The strengths and limitations of each of those three formalisms in specifying various kinds of stateful Web services are examined, and the choice of SXMs is justified. Next, SXMs are described in further depth, including their mathematical definition, properties, and variants. Having introduced SXMs, the fourth section clarifies the correspondence between the elements of SXM specifications and their counterparts in stateful Web service implementations, using the Bank Account Web service example introduced in the beginning of this chapter. The next section describes more advanced modelling techniques and best practices. Given that the model should stand at a higher level of abstraction than the implementation, abstraction techniques in terms of data and behaviour are suggested. This section also discusses modelling obstacles for more complex Web services, and suggests possible solutions. Those Web services include services managing several instances of data objects, services maintaining huge data repositories, and services requiring confidential inputs or inputs that are difficult to generate. The next section of this chapter demonstrates that a SXM model, which represents the explicit Web service choreography, can be inferred from an implicit choreography description through IOPE specifications of service operations, which are considered easier to declare. The described algorithm ensures that states and transitions are properly derived, and that the necessary design-for-test properties are satisfied. Since SXM specifications do not always

satisfy certain expected properties, sections 7 and 8 critically investigate completeness, controllability, and determinism and their implication in the context of Web services.

5.1 Two service examples: Bank Account and Supply Order

This section introduces the Bank Account and Supply Order Web service examples, which are used for motivating and better explaining the proposed testing approach and the employed methods. The Bank Account (or simply *Account*) Web service serves as a simple Web service of minimal complexity, to demonstrate basic modelling and testing activities. The Supply Order (or simply *SupplyOrder*) Web service serves as a more sophisticated example and representative of real-world Web services, which will be used for explaining more complex specification and testing techniques. Both Web services are stateful and conversational.

5.1.1 Bank Account

The Bank Account Web service exposes operations for performing elementary transactions on a bank account over the Internet. For simplicity, it is assumed that the service interface consists of five operations: (i) *open*, (ii) *deposit*, (iii) *withdraw*, (iv) *getBalance*, and (v) *close*. When an account is created it is initialised as inactive and therefore needs to be set to active (opened) before any transaction can be performed. The deposit of an amount will result in increasing the balance of the account as appropriate, while the withdrawal of an amount can take place only if the amount does not exceed the balance, and will result in reducing the balance accordingly. A successful deposit or withdrawal will also result in having the updated balance returned to the client as part of the invocation response message. Finally, an account can be closed only if its balance is zero, and once closed cannot be re-activated.

Web service WSDL

The following is an extract of the WSDL document for the Account Web service. It lists only the abstract interface (portType), which summarises the service operations and their inputs and outputs.

```
<wsdl:portType name="AccountPortType">
  <wsdl:operation name="withdraw">
    <wsdl:input message="ns:withdrawRequest"/>
    <wsdl:output message="ns:withdrawResponse"/>
  </wsdl:operation>
  <wsdl:operation name="open">
    <wsdl:input message="ns:openRequest"/>
    <wsdl:output message="ns:openResponse"/>
  </wsdl:operation>
  <wsdl:operation name="deposit">
    <wsdl:input message="ns:depositRequest"/>
    <wsdl:output message="ns:depositResponse"/>
  </wsdl:operation>
  <wsdl:operation name="getBalance">
    <wsdl:input message="ns:getBalanceRequest"/>
```

```
    <wsdl:output message="ns:getBalanceResponse"/>
  </wsdl:operation>
  <wsdl:operation name="close">
    <wsdl:input message="ns:closeRequest"/>
    <wsdl:output message="ns:closeResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

5.1.2 Supply Order

The SupplyOrder Web service is another example of a stateful Web service, which allows the procurement of new raw materials by a manufacturer from a supplier partner [81]. The order processing transaction is performed in a number of steps and in accordance with a conversation protocol. The SupplyOrder Web service consists of the following operations: createOrder, cancelOrder, addItem, removeItem, getQuotation, rejectOrder, and confirmOrder, which can be called in sequences permissible by the protocol. This second Web service example has been selected on purpose, since it exhibits more complex behaviour and operates on complex data repositories, in order to be closer to the kinds of Web services that are expected to be found in the industry.

Normally, in accordance with the CRUD (create-read-update-delete) lifecycle of data objects, the manufacturer should be able to create new orders, and read, update, or delete existing orders. However, for simplicity, in this scenario only the creation of a new empty order with the createOrder operation is modelled. The manufacturer can populate the new supply order by adding items specifying their id and requested quantities, through the repetitive invocation of the addItem operation. Order items can also be removed or the order cancelled altogether, after which the manufacturer has to create a new order. The addItem operation is successfully fulfilled if the items of the requested quantities are available in the inventory. The getQuotation operation returns an order quotation (unless the order is empty), listing the items that are ordered, their availability and their prices. This gives the manufacturer the choice to proceed with the confirmation of the order, even if it is partially fulfilled (because some items are out of stock), or alternatively reject the order. The getQuotation operation temporary locks the ordered items of the requested (or available) quantities in the inventory, so that no other client simultaneously accessing the system can order them until the current order is confirmed or rejected. Upon confirmation of the supply order, the item quantities that are fulfilled are subtracted from the inventory and the transaction ends.

In contrast to the Account Web service, the SupplyOrder service follows the Manager (or Factory) pattern [72], which manages several order instances and allows order creation, modification, and deletion. It also accesses a large database of the available inventory items and respective quantities, which are simultaneously accessed and possibly modified by other clients. That is, the SupplyOrder Web service consists of shared state and introduces new testing challenges.

Different versions of the SupplyOrder service have been implemented for experimentation purposes, starting from a naïve Web service managing a single order, standing at the same level of abstraction as the specification, and without any inventory lookup. Later, as new testing techniques are introduced, implementations of increasing degree of sophistication and complexity are used. These include a multiple-order Web service, an implementation with SOAP and WSDL faults for negative testing, an implementation with inventory lookup, and finally a Web service implementation following the Manager pattern. In addition, a number of faulty implementations are used in the next chapter to demonstrate the ability of generated test cases to reveal various types of faults.

Web service WSDL

An extract of the WSDL document for the SupplyOrder Web service, describing the abstract interface (portType) is the following.

```
<wsdl:portType name="SupplyOrderPortType">
    <wsdl:operation name="getQuotation">
        <wsdl:input message="ns:getQuotationRequest"/>
        <wsdl:output message="ns:getQuotationResponse"/>
    </wsdl:operation>
    <wsdl:operation name="confirmOrder">
        <wsdl:input message="ns:confirmOrderRequest"/>
        <wsdl:output message="ns:confirmOrderResponse"/>
    </wsdl:operation>
    <wsdl:operation name="cancelOrder">
        <wsdl:input message="ns:cancelOrderRequest"/>
        <wsdl:output message="ns:cancelOrderResponse"/>
    </wsdl:operation>
    <wsdl:operation name="addItem">
        <wsdl:input message="ns:addItemRequest"/>
        <wsdl:output message="ns:addItemResponse"/>
    </wsdl:operation>
    <wsdl:operation name="createOrder">
        <wsdl:input message="ns:createOrderRequest"/>
        <wsdl:output message="ns:createOrderResponse"/>
    </wsdl:operation>
    <wsdl:operation name="rejectOrder">
        <wsdl:input message="ns:rejectOrderRequest" />
        <wsdl:output message="ns:rejectOrderResponse"/>
    </wsdl:operation>
    <wsdl:operation name="removeItem">
        <wsdl:input message="ns:removeItemRequest"/>
        <wsdl:output message="ns:removeItemResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

5.2 State-based formalisms and Web service modelling

A stateful Web service keeps track of internal state, which is determined by the previous inputs applied to the service, and in turn, determines the outcome of future inputs. Attempting to model such a service should take into consideration the current state the service is found in. Thus, we believe a stateful service is intuitively modelled as a set of abstract states and transitions between those states. A considerably large variety of computational models that adopt this modelling

approach have been developed up to date, including X-machines, which are collectively referred to as *state-based formalisms*, or *state machines*.

Selecting a proper state-based formalism for the purpose of modelling a Web service is dependent on different factors. These include, the expressive power of the formalism being used, the modelling overhead, the characteristics of the Web service being modelled, as well as the power of the testing method applicable to the formalism. In this section we briefly examine three representative state-based formalisms, which are: finite state machines (FSMs), extended finite state machines (EFSMs), and finally, stream X-machines (SXMs). Advantages and drawbacks of each formalism are considered. Overall, going from FSMs through EFSMs to SXMs, the formalism becomes more expressive and powerful, the complexity of models and modelling overhead increases, and the set of possible Web services that can be modelled becomes broader.

5.2.1 Finite State Machines

Among the simplest variations of state machines are the finite state machines (FSMs). They model computation as transitions between a finite set of states. Although different flavours of finite state machines exist (finite automata, accepters, transducers, etc), here we adopt the definition by [68].

A *nondeterministic* (or stochastic) *finite state machine* is mathematically defined as a 6-tuple $(\Sigma, \Gamma, Q, F, \lambda, I)$, where:

- Σ is a finite set called the input alphabet;
- Γ is a finite set called the output alphabet;
- Q is a finite, non-empty set of states;
- F is the state-transition function, $F: Q \times \Sigma \rightarrow 2^Q$;
- λ is the output function, $\lambda: Q \times \Sigma \rightarrow \Gamma$;
- $I \subseteq Q$ is the set of initial states.

Deterministic finite state machines are defined similarly to nondeterministic finite state machines as above, with two differences:

- The transition function F maps each (state, input) pair into at most one state, i.e. $F: Q \times \Sigma \rightarrow Q$;
- The machine contains only one initial state, i.e. $I = \{q_0\}$.

In other words, a finite state machine specifies the states, the transitions between states, the inputs triggering transitions, and the outputs produced. Thus, FSMs are valuable in describing the dynamic behaviour of systems. Furthermore, since FSMs can be represented graphically by state transition diagrams, they are easy to understand by people with minimal mathematical background.

However, there are serious limitations when plain FSMs are employed to specify Web services. The main limitation is that FSMs cannot model complex data structures that constitute the static aspect of the system. This causes what is known

as the *state explosion* problem, which is made obvious if we attempt to model the Account Web service described above. At first, let us consider a simplified (and rather naïve) version of the Account Web service, in which one is only allowed to make deposit and withdrawal transactions by discrete amounts of 10 (currency is not specified), and the maximum allowable balance is 50. Figure 14 is the state transition diagram of the FSM model for the simplified Account service. Notice that there is a state for every possible value of the bank account balance. When the state "max" is reached, only withdrawals are allowed. Imagine now that deposit and withdrawal transactions of amounts of 5 are allowed. The diagram in Figure 14 would consist of five additional states. If transactions of amounts of 1 are allowed the resulting FSM would have to contain many more states. If unlimited transaction amounts are allowed, or if there is no limit on the balance value, then there would be infinite possible values for balance, which would be impossible to model with a finite state machine.

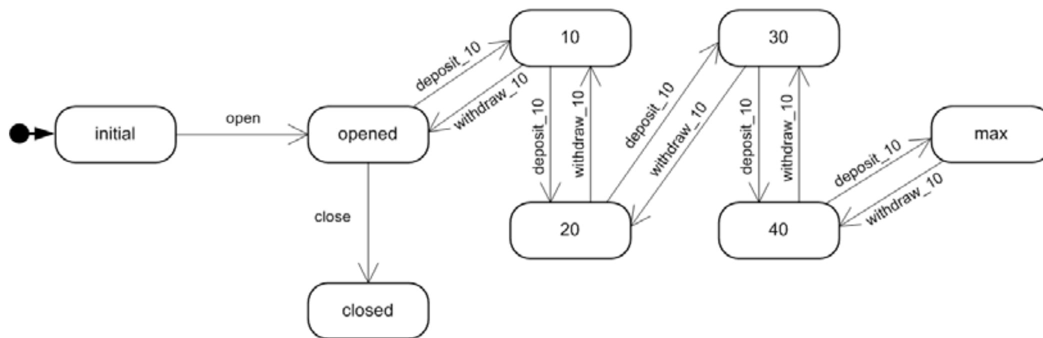


Figure 14 – Deterministic FSM model of a simplified Bank Account

An alternative approach to deal with state explosion is to abstract all states representing the values of balance as a single state called normal (Figure 15). The resultant FSM consists of fewer states, but it does not fully specify the Account service, as it does not capture the balance information that affects outputs produced by the service. As can be noticed, the FSM model is nondeterministic. For example, the input-state pair (deposit_10, normal) is mapped to a set of two possible states, normal and max (and produces two different outputs). This form of nondeterminism, in which there is more than one possible next state for an input-state pair, is also referred to as state nondeterminism in [69].

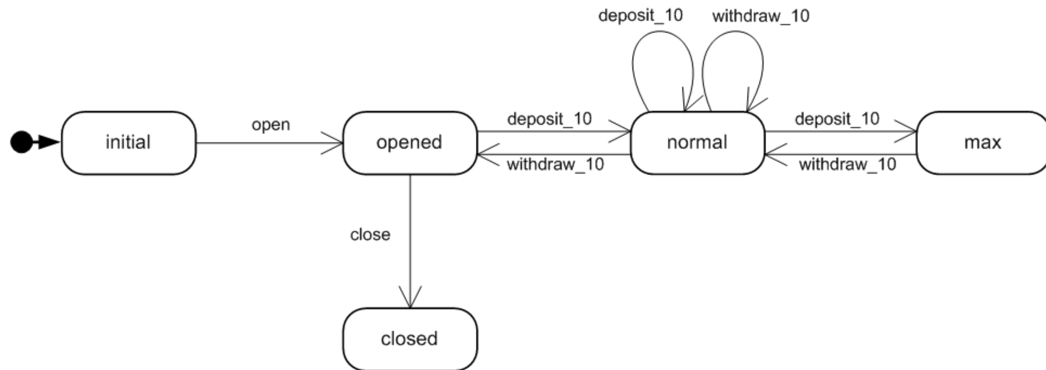


Figure 15 – Nondeterministic FSM model of the simplified Bank Account

It can be concluded that there are two possibilities in employing FSMs to specify stateful Web services:

- create deterministic specifications of very simple, trivial Web services;
- create abstracted, nondeterministic specifications of realistic Web services.

Apparently, most stateful Web services are expected to operate on data variables that can take on infinite values, hence the second alternative is a more likely scenario.

On the positive side, finite state machines are a simple formalism requiring little modelling overhead. FSM models are kept abstract, while being able to capture in the machine language L the control flow of *conversational Web services*, which follow a conversation protocol of allowed operation sequences. Nevertheless, for Web services maintaining non-trivial internal state or services operating on large data repositories, such as databases, finite state machines are not appropriate. Not only can't they model non-trivial data structures, they are also unable to specify the computations that produce outputs from provided inputs, except for the state transitions between the abstract states.

For deterministic FSM specifications of trivial Web services, a strong notion of testing, called *equivalence* testing, is applicable (explained in more detail in section 7.4). On the other hand, when nondeterministic FSMs are used to specify more realistic Web services, a weaker notion of testing, called *conformance* testing, can be applied [69]. Testing the conformance of Web service conversation protocols to the language of the specification machine may be useful, but is not always sufficient, especially for services operating on complex data structures. Finally, a significant disadvantage of nondeterministic FSM specifications is that they cannot serve as precise test oracles in defining the outputs, as the expected output will be any from a set of possible outputs.

5.2.2 Extended Finite State Machines

Extended Finite State Machines (EFSMs) enhance FSMs with the addition of a set of variables to the machine, which formally constitute an *n*-dimensional space, or memory. In this way, EFSMs address the state explosion problem by parameterising the states with variables that can potentially assume an infinite number of values.

As with FSMs, there are several different definitions for EFSMs. A common definition, taken from [70], and [64], is provided below. The symbols have been renamed for better comparativeness with the stream X-machine definition.

An Extended Finite State Machine is defined as the 7-tuple (Q, Σ, Γ, D, F, U, T), where:

- Q is a set of symbolic states;
- Σ is a set of input symbols;
- Γ is a set of output symbols;
- D is an n-dimensional space $D_1 \times \ldots \times D_n$;
- F is a set of enabling functions f_i such that $f_i: D \rightarrow \{0, 1\}$;
- U is a set of update transformations u_i such that $u_i: D \rightarrow D$;
- T is a transition relation such that $T: Q \times F \times \Sigma \rightarrow Q \times U \times \Gamma$.

In another variant of EFSM definition, the set of enabling functions F and the set of update transformations U are merged into the transition relation $T: Q \times D \times \Sigma \rightarrow Q \times D \times \Gamma$, hence the machine is a 5-tuple, (Q, Σ, Γ, D, T) [68]. In this rearrangement, the transition relation does not use predefined predicates from the set F and assignments from the set U, but maps (state, memory, input) triples to (state, memory, output) triples directly.

In addition to the benefit of addressing the state explosion problem with memory variables, EFSMs can model both the conversation protocol and the internal data of a Web service. The service protocol is represented by the state transition diagram of the machine, consisting of the states from Q and the transitions between any two states as permitted by T. Service data is represented in the *n*-dimensional space D of the EFSM. As a result, the range of stateful Web service categories that can be specified with EFSMs is broadened, including services operating on *large or complex data repositories*, and *shared state services*. Furthermore, it is possible to create EFSM specifications which are *deterministic*, even for relatively complex Web services, as it is possible to capture all the factors that determine their behaviour.

5.2.3 Stream X-Machines

Stream X-machines (SXMs), formally defined in the next section, are *a kind of* EFSM. The difference is that the variables in the *n*-dimensional space are replaced by a memory element M; the sets of enabling functions F and update transformations U are merged into a set of processing functions, Φ; and the mapping of inputs to outputs is defined by the individual processing functions rather

than the transition function T. As a result, in SXMs the transition function F: $Q \times \Phi \to Q$ is much simpler and is easily represented by a state-transition diagram.

Being a kind of EFSM, the stream X-machine inherits the advantages of EFSMs relative to FSMs. SXM specifications are capable of modelling both the dynamic control and the static data of systems. This specification power is essential in modelling stateful Web services, which often enforce operation sequencing rules in accordance with a protocol (conversational Web services), and operate on data structures which can get fairly complex and large. Notably, SXMs *subsume* the expressive power of FSMs, since the memory element can be specified as empty and the processing functions can map single input symbols to output symbols. More abstract, nondeterministic versions of SXM specifications can also be utilised to model complex and large-scale Web services, as described later in this section.

An important characteristic of SXMs, as compared to the generic EFSMs, is that the preconditions, memory updates, and output computations, which associate transitions between two states, are decoupled from the transition function and modularised as processing functions. Since SXMs employ a diagrammatic approach to modelling the control with state-transition diagrams, the transitions are labelled with processing functions. Therefore, it is possible to define an associated finite automaton (FA) of a SXM, by treating the processing functions as abstract input symbols. Logically, the associated finite automaton represents the integration of individual system components, which are in turn specified by processing functions.

This separation of the system integration level from the lower level components makes it possible to apply the powerful SXM integration testing (SXMT) method [62], [61], which is described in the next chapter. SXMT relies on the application of Chow's W-method [63] to the associated finite automaton of the machine. Assuming the individual processing functions to be correctly implemented, the method *guarantees* that all faults are revealed in the Web service under test. In contrast, it is not possible to apply the SXMT to the generic EFSMs, as they do not decouple state transitions from processing functions. Instead, EFSM-based testing methods involve flattening the EFSM into an equivalent finite state machine (whose states are the state-memory pairs of the EFSM), which frequently results in state explosion [64].

In addition, it is possible to further model the individual processing functions as simpler SXMs and test their correctness separately in a similar manner. The process may proceed down to lowest level where the components can be safely assumed correct (such as standard library calls, or operating system routines). Further details about this divide-and-conquer specification approach and complete SXM testing method are described in [65].

Finally, X-machines, and stream X-machines in particular, are backed by sound theoretical foundations, languages, and supporting tools (described later). Also, X-machines have been used in a wide variety of practical application areas, as diverse as cell biology, multi-agent systems, and hardware design.

The advantages described above make SXMs preferable over EFSMs for modelling and testing stateful Web services. In this thesis SXMs are employed for creating behavioural specifications of Web services, which are amenable to the SXM integration testing method with correctness guarantees. The disadvantages relative to finite state machines, such as specification overhead, are generally compensated by the fact that specifications are more precise, deterministic, and the applicable equivalence testing method ensures the trustworthiness of the Web services, which is an especially important attribute in critical applications.

5.3 Background on Stream X-Machines

This section provides selected theoretical background on stream X-machines, being the formalism employed to specify Web services for testing.

5.3.1 The Stream X-Machine formalism

Stream X-machine (SXM) is a computational model introduced by Gilbert Laycock in 1993 [66], which extends the *X-machine* model introduced by Samuel Eilenberg in 1974 [67]. In essence an X-machine is like a finite state machine, with the difference that transitions are associated with relations (often functions) that operate on a basic *data set X*. SXMs further enhance this model by including input and output streams as parts of the data set *X*, thus making the important distinction between memory and I/O.

A stream X-machine is mathematically defined as a 9-tuple, $(\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ [61] where:

- Σ and Γ is the input and output finite alphabet respectively;
- Q is the finite set of states;
- M is the (possibly) infinite set called memory;
- Φ, which is called *the type of the machine*, is a finite set of distinct *processing relations* that the machine can use; a processing relation is a non-empty relation of the form $\varphi: \Sigma \times M \leftrightarrow \Gamma \times M$; Φ often is a set of (partial) functions;
- F is the next state partial function that given a state and a function from the type Φ, provides the next state, $F: Q \times \Phi \rightarrow 2^Q$ (F is often described by a state-transition diagram);
- I and T are the sets of initial and terminal states respectively, $I \subseteq Q, T \subseteq Q$;
- m_0 is the initial memory value, $m_0 \in \mathrm{M}$.

It is sometimes helpful to think of a SXM as a *finite automaton* (FA) by treating the relations that label the transitions as abstract input symbols [57]. This automaton, defined as the tuple (Φ, Q, F, I, T) is called *the associated FA* of the X-machine. One helpful practice in SXM modelling is the use of state-transition diagrams to depict graphically the control flow of the associated FA. A human person with minimal mathematical knowledge can understand the dynamic behaviour of the

modelled system by viewing the state diagram. Obviously, to further understand the data structure maintained in the memory of the machine and the computations performed on the memory by the processing relations, the interested human individual has to read the detailed specification.

According to the preceding definition, a SXM can be *nondeterministic*, in the sense that the application of an input $\sigma \in \Sigma$ in a state $q \in Q$ for a memory value $m \in M$ may produce more than one possible output. More specifically, the machine may contain more than one initial state, an input may trigger one of several possible processing relations, the triggered processing relation may produce one of several possible outputs and memory updates, and the next state may be one of several possible next states. That is, the starting state, the triggered processing relation, the output, the memory update, and the next state are uncertain.

In this thesis focus will be given to *deterministic stream X-machines* (*DSXM*s). A SXM Z is defined as deterministic if the following hold:

- The associated FA of the machine is deterministic, i.e.
    - Z has only one initial state, i.e. $I = \{q_0\}$;
    - The next state function of Z maps each pair (state, processing function) onto at most one state, i.e. $F : Q \times \Phi \rightarrow Q$;
- Φ is a set of (partial) functions rather than relations;
- Any two distinct processing functions that label arcs emerging from the same state have disjoint domains, i.e. $\forall \varphi_1, \varphi_2 \in \Phi$, $((\exists q \in Q$ with (q, φ_1), $(q, \varphi_2) \in dom\ (F)) \Rightarrow (\varphi_1 = \varphi_2$ or $dom\ (\varphi_1) \cap dom\ (\varphi_2) = \emptyset))$.

Therefore, a DSXM with all states terminal ($T = Q$), is now defined mathematically as an *8-tuple*, $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$. The associated FA of a DSXM is then a 4-tuple, (Φ, Q, F, q_0).

In a DSXM, starting from the initial state q_0 and initial memory value m_0, an input symbol $\sigma \in \Sigma$ triggers a processing function $\varphi \in \Phi$, which in turn triggers a transition to a new state $q \in Q$ and a new memory value $m \in M$, while producing an output $\gamma \in \Gamma$. The sequence of transitions (path) triggered by the stream of input symbols is called a *computation*. The computation halts when all input symbols are consumed. The result of a computation is the sequence of outputs symbols produced by this path [68]. All possible computations performed by a DSXM comprise the *function computed* by the machine, which maps input sequences to output sequences and is denoted by $f: \Sigma^* \rightarrow \Gamma^*$.

Apart from being formal as well as proven to possess the computational power of Turing machines [61], the SXM computational model has the significant advantage of being associated with a well-studied testing method with completeness guarantees [61], [62]. This method generates test sets for a system specified as a SXM whose application ensures that the system behaviour is equivalent to that of the specification, provided that the system is made of fault-free components and some explicit design-for-test requirements are met (see next chapter).

5.3.2 Other properties of stream X-machines

In the previous section, a particular subset of SXMs, the DSXMs, were defined based on the property of *determinism*. In this section, further properties that characterise SXM models are defined. These properties will be referred to in other parts of the thesis, especially in the testing section, since some of them are prerequisites in the application of the SXM testing method. Further properties of SXMs include the following:

- minimalism
- output-distinguishability (observability)
- input-completeness (controllability)
- completeness of specification
- uniformity (of the machine type Φ)

A deterministic FA, *A,* is called *minimal* if any other FA that accepts the same language as *A* has at least the same number of states as *A*. It follows that a SXM is considered as minimal if its associated FA is minimal.

Φ is called *output-distinguishable* if $\forall \varphi 1, \varphi 2 \in \Phi$, $((\exists m \in M, \ \sigma \in \Sigma$ with $\pi_1(\varphi_1(m, \sigma)) = \pi_1(\varphi_2\ (m, \sigma))) \Rightarrow \varphi_1 = \varphi_2)$. This says that we must be able to distinguish between any two different processing functions by examining outputs. If we cannot then we will not always be able to tell them apart.

Φ is called *input-complete* if $\forall \varphi \in \Phi, m \in M, \exists \sigma \in \Sigma$ such that $(m, \sigma) \in dom\ (\varphi)$. This condition ensures that any processing function can be exercised from any memory value using appropriate input symbols (regardless of the state q).

A *completely defined* SXM is one in which there is at least one possible transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$. That is, $\forall q \in Q, m \in M, \sigma \in \Sigma, \exists \varphi \in \Phi$ such that $((m, \sigma) \in dom(\varphi)$ and $(q, \varphi) \in dom(F))$. Therefore, a SXM which is completely defined specifies functionality that handles every input symbol in any state and for any memory value.

It is easy to see that in a DSXM there is at most one possible transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$. Also note that if a deterministic SXM is completely defined then there is exactly one transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$. More on completeness of Web service specifications will be discussed in section 5.7 of this chapter.

5.3.3 Other variants

Object Machines

Object Machines (OM) are a variation of SXMs, intended to bring the formalism closer to the object orientation paradigm [71]. OMs specify the behaviour of individual software objects, whose behaviour is implemented by methods and state is maintained in attributes. Also, protocol machines are decoupled from method machines. In addition, a transition is fired from a state and the next state is decided

after the execution of the processing function (i.e. method code). Thus the logic for deciding the next state is decoupled from processing code. Model animation corresponds more closely to execution of class methods, and Web service operations as well.

Although OMs are relevant to consider as a variant for specifying Web services, the formalism is not yet supported by tools and is not considered in this thesis.

5.4 Correspondence between Web service elements and SXM elements

Although the role of SXMs is to specify the externally-visible system behaviour, and the associated testing method is black-box, it is still useful to investigate the correspondence between the SXM elements and their counterparts in stateful Web service implementations. This correspondence is especially helpful when attempting to model already existing Web services, rather than specifying the user requirements for desired Web services.

Both a stateful Web service and a SXM accept inputs and produce outputs, while performing computations and transitioning between internal states. Therefore, the correspondence is fairly obvious. SXM input symbols model Web service requests, while SXM output symbols model Web service responses. SXM states and memory represent Web service state (data), which was described in the previous chapter. SXM processing functions (relations, in nondeterministic SXMs) represent the computations performed by invoked service operations.

5.4.1 SXM inputs

The correspondence for inputs and outputs is slightly more complex than just presented. As mentioned in the overview of the SOAP protocol, in section 2.2.1, SOAP envelopes are often associated with essential header information from the underlying transport protocol, such as HTTP. This means that, sometimes, Web service requests can be more than SOAP request messages, and responses can be more than SOAP response messages.

In the case of inputs, Web service requests need to define:

(a) the operation to invoke and

(b) the business payload to provide as argument to the operation.

Requests may also contain further header information, such as session identifiers or security protocol information, but this is not considered as essential and is usually abstracted away. While the business payload is contained in the body of a SOAP request message, the target operation could be specified either inside the SOAP envelope or externally to it (See section 9.3.4 for a description of how message dispatching is performed during testing). Therefore, it is not always accurate to consider abstract SXM input symbols as analogues of SOAP request messages. Since the intent of a service request is to invoke an operation on the Web service,

then it is more helpful to view SXM input symbols as representing *operation invocations* (*calls*) with the message data passed as arguments.

In the approach adopted by the JSXM notation [59], described later in this chapter, input symbols are grouped by the Web service operation they represent. For example, in the specification of the Account Web service, all input symbols that represent invocations of the "open" operation belong to one group; those that represent invocations of the "deposit" operation belong to another group, and so on. Each group representing an operation is defined separately in the specification. Input definitions can be either simple or complex. *Simple inputs* are specified only by the target operation, thus they define single SXM input symbols. On the other hand, *complex inputs*, in addition to the target operation, are specified by one or more *arguments* of designated data types. Hence, every complex input defines a potentially infinite set of SXM input symbols. This set is the Cartesian product of the set consisting of the target operation (input name) and the sets defined by the types of all arguments.

- $\Sigma_i = \{input_name\}$, for simple inputs;
- $\Sigma_i = \{input_name\} \times arg_{i1} \times \ldots \times arg_{in}$, for complex inputs of n arguments.

Arbitrary complexities in input argument types are achieved recursively from simpler types, and eventually from elementary types (finite or infinite sets). However, attempting a mathematical representation of all possible types is out of scope of this thesis.

For example, the "open" simple input modelling operation "open" in the Account Web service is defined as the set:

$$\Sigma_{\text{open}} = \{\text{open}\},$$

while the deposit complex input modelling operation deposit, which contains an integer amount argument, is defined as the following set,

$$\Sigma_{\text{deposit}} = \{\text{deposit}\} \times Z = \{\text{deposit}\} \times (-\infty, +\infty) =$$

$$= \{\ldots, (\text{deposit}, -1), (\text{deposit}, 0), (\text{deposit}, 1), \ldots\}.$$

The resulting input alphabet, Σ, of the machine, is the union of all input sets. For a machine of n input definitions in the specification:

$$\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n.$$

Assuming that the SXM specification of the Account Web service consists of five inputs (modelling five operations): open, close, getBalance, deposit and withdraw, of which the last two are associated by an integer argument, then the input alphabet is the following infinite set:

$$\Sigma = \{\text{open}\} \cup \{\text{close}\} \cup \{\text{getBalance}\} \cup (\{\text{deposit}\} \times Z\}) \cup (\{\text{withdraw}\} \times Z\}), \text{ or,}$$

$$\Sigma = \{\text{open, close, getBalance}, \ldots, (\text{deposit}, -1), (\text{deposit}, 0), (\text{deposit}, 1), \ldots, (\text{withdraw}, -1), (\text{withdraw}, 0), (\text{withdraw}, 1), \ldots\}.$$

5.4.2 SXM outputs

Usually, all essential information in a Web service response is contained within a SOAP envelope. Thus, SXM output symbols represent SOAP response messages, or in cases of failure, fault messages. Despite being semantically different kinds of responses from the Web service, they correspond to outputs in the SXM specification, since the SXM formalism does not differentiate between normal and error outputs.

As with SXM input symbols, SXM output symbols can be grouped by the source operation. *Simple outputs* are defined only by the source operation (output name), thus they specify single SXM output symbols. Complex outputs, are additionally defined by one or more parts (called *results* in JSXM) of designated data types. The same reasoning used to derive the input alphabet, applies to the output alphabet of the machine. For the Account SXM:

$\Gamma = \{\text{openOut}\} \cup \{\text{closeOut}\} \cup (\{\text{getBalanceOut}\} \times Z\}) \cup (\{\text{depositOut}\} \times Z\}) \cup (\{\text{withdrawOut}\} \times Z\});$

$\Gamma = \{\text{openOut, closeOut, …, (getBalanceOut, -1), (getBalanceOut, 0), (getBalanceOut, 1), …, (depositOut, -1), (depositOut, 0), (depositOut, 1), …, (withdrawOut, -1), (withdrawOut, 0), (withdrawOut, 1), …}\}.$

5.4.3 SXM states and memory

As described in the previous chapter, Web service state is data in various forms, which is accessed and/or modified by the Web service and affects its functional behaviour. It is possible to break down Web service state into individual variables, which in this thesis are referred to as state variables. Some or all of those variables may be represented in the SXM specification, depending on whether that specification is going to be deterministic, or nondeterministic.

State variables are modelled as control states and in the memory structure of the SXM. Complex state variables of infinite values are specified in the memory element *M* (SXM testing can deal with infinite memory). Important state variables of discrete values can be specified as SXM control states. Since control states stand at a higher level of abstraction (integration level) of the machine, each control state can also represent an abstraction over a range of values assumed by the SXM memory. For example, a SXM state could stand for all non-zero values of the bank account balance. The significance of SXM control states is that they define discrete ranges of values over Web service state, for which the subsequent behaviour is considerably distinct. Determining which part of Web service state is modelled as control states and which as memory is a matter of decision, depending on the modelling and testing goals.

The Account Web service keeps track of the status of the bank account and the remaining amount in the balance. Having an integer type, the account balance assumes infinite values, thus it is driven to the SXM memory to avoid state

explosion. Thus, $M = Z$. The initial memory value corresponds to the initial amount of zero in the balance: $m_0 = 0$.

We can choose the control states of the machine to represent the discrete statuses of the account (initial, active, closed) and two ranges of values for the balance (zero and non-zero). There are four valid combinations (initial and zero, active and zero, active and non-zero, closed and zero). Therefore, the set of states Q of the Account SXM consists of four elements, which we name as follows:

Q = {initial, opened, normal, closed}.

The initial state is q_0 = initial.

5.4.4 SXM transitions

SXM transitions represent transitions between the abstracted states assumed by the Web service. They are triggered by operation invocations and associated with some computations represented by processing functions.

The state transition function $F: Q \times \Phi \rightarrow Q$ is depicted as a state-transition diagram in Figure 16.

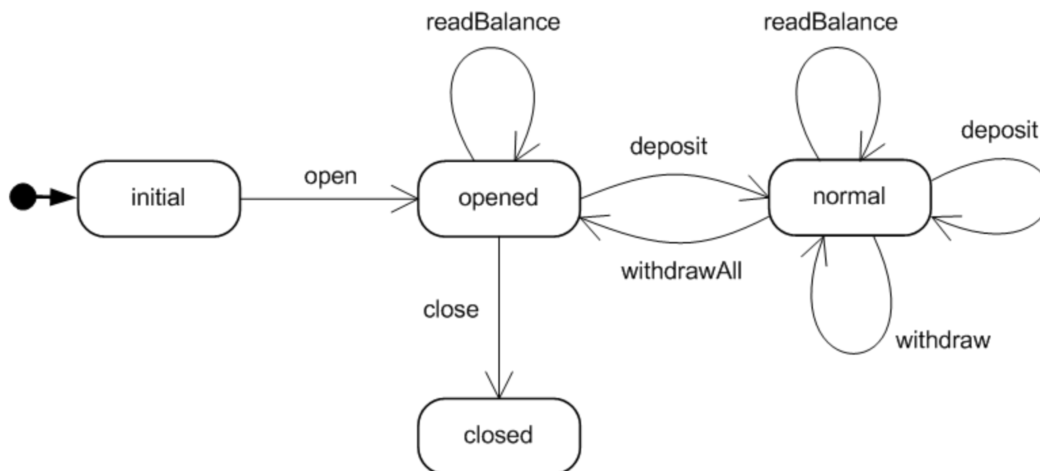


Figure 16 - State-transition diagram of the Account SXM

5.4.5 SXM processing functions

While an input symbol triggers a processing function on a SXM, the invocation of an operation on a Web service triggers a set of actions, which consist of state updates and output computations. Technically, those actions are performed by the execution of the code implementing the invoked operation. However, since a processing function can be triggered only from some of the states and since it defines a domain on inputs and memory, the Web service counterpart is that portion of the code, which is executed upon the satisfaction of specific predicates involving the request message and the service state. Therefore, it can be noticed that processing functions do *not* necessarily correspond one-to-one with Web service

operations and should not be confused with each other. The invoked operation may take different paths under different conditions, which are possibly modelled by different processing functions. As a result a Web service operation implements one or more processing functions.

For example, although in Figure 16 some of the transition labels do coincide with operation names, they do not represent the whole operations, but their *success scenarios*. The invocation of the success scenario of the "open" operation, when the status is "initial", triggers the "open" processing function, but it may also trigger an "openError" function (not modelled, see section 5.7 for completeness of specifications) if the operation is invoked when the Web service has already been opened. Similarly, there are two possible transitions triggered by the invocation of operation withdraw: one leading to the same "normal" state of positive balance, and the other to the "opened" state of zero balance if all money is withdrawn. They are labelled by two different processing functions, in order to keep the associated FA deterministic. Both cases are considered as success scenarios. Another case, which is not modelled to keep the specification uncluttered, is when the amount to be withdrawn is unavailable (failure scenario).

As a result, the set of processing functions for the (partially-specified) Account SXM is specified as the following machine type:

Φ = {open, close, getBalance, deposit, withdraw, withdrawAll}.

Table 1 summarises the correspondence between SXM elements and the Web service counterparts, as discussed in this section.

Table 1 - Correspondence between stream X-machine and Web service elements

SXM	**Web service**
input symbol ($\sigma \in \Sigma$)	operation invocation
output symbol ($\gamma \in \Gamma$)	SOAP response or fault message
state ($q \in Q$)	abstracted state of the Web service, considerably characterizing its subsequent behaviour
memory (M)	other state/data accessed by the Web service
processing function ($\varphi \in \Phi$)	code executed by a Web service operation under certain preconditions on input and state
transition ($f \in F$)	transition between abstract Web service states
initial state (q_0)	the abstract state the Web service is initially found
initial memory (m_0)	initial value(s) of other internal state/data

5.5 Modelling practices in the Web services domain

A number of best practices are suggested when deriving the SXM model of a Web service implementation. These modelling practices account for some characteristics that are common among Web services and present challenges during the modelling

task. Generally, these techniques aim to raise the level of abstraction of the specification and deal with other aspects such as huge service state and confidential inputs required to drive the service.

5.5.1 Modelling individual stateful objects

One common characteristic of real-world Web services is that they are intended to serve several clients and thus maintain data separately for each one of them. Let us return to the example of the SupplyOrder Web service, which manages a collection of supply orders for a number of clients. Attempting to model this Web service as a SXM will normally require capturing all the service state, consisting of all order instances, in the memory structure of the machine. Although individual order instances may be in different stages of their lifecycle, the state of the Web service is defined by the combination of all current order objects in the SXM memory. Thus, one could not abstract a discrete number of useful control states that would specify the high level behaviour of the Web service. Instead there would be one single control state and all the rest of the Web service state represented in the memory. A SXM with one state is inadequate to represent the control flow and is not conducive to test set generation.

Alternatively, it would be more useful to model the state of an individual order instance and the behaviour of operations on that order. As already explained in section 0, there are three different views on stateful Web services with increasing levels of abstraction: pan-client, per-client, and per-object. The per-object view is made possible when properly identified invocations of operations access and modify only one object and are not affected by the rest of the state. Therefore, it is often possible for a SXM specification to adopt any of the above views: the more abstract, the better. Although the Account Web service actually manages a collection of bank accounts for several clients, the SXM specified in the previous section models a single account, corresponding to the per-object view. Similarly, the SXM for the SupplyOrder Web service can represent all service state, the state for a single client, or the state for a single order instance.

The advantages of adopting a per-client or per-object view are summarized as follows:

- The behavioural specification is vastly simplified:
    - Only the portion of the state referring to a single client or object is specified;
    - Only the operations accessing the per-client or per-object state are specified in the model;
- Control states of individual objects are exposed for specification and for testing purposes;
- Client or object identification details are abstracted away from the specification.

Figure 17 aims to demonstrate the relationships between the pan-client, per-client, and per-object SXM models of a SupplyOrder Web service requiring authentication.

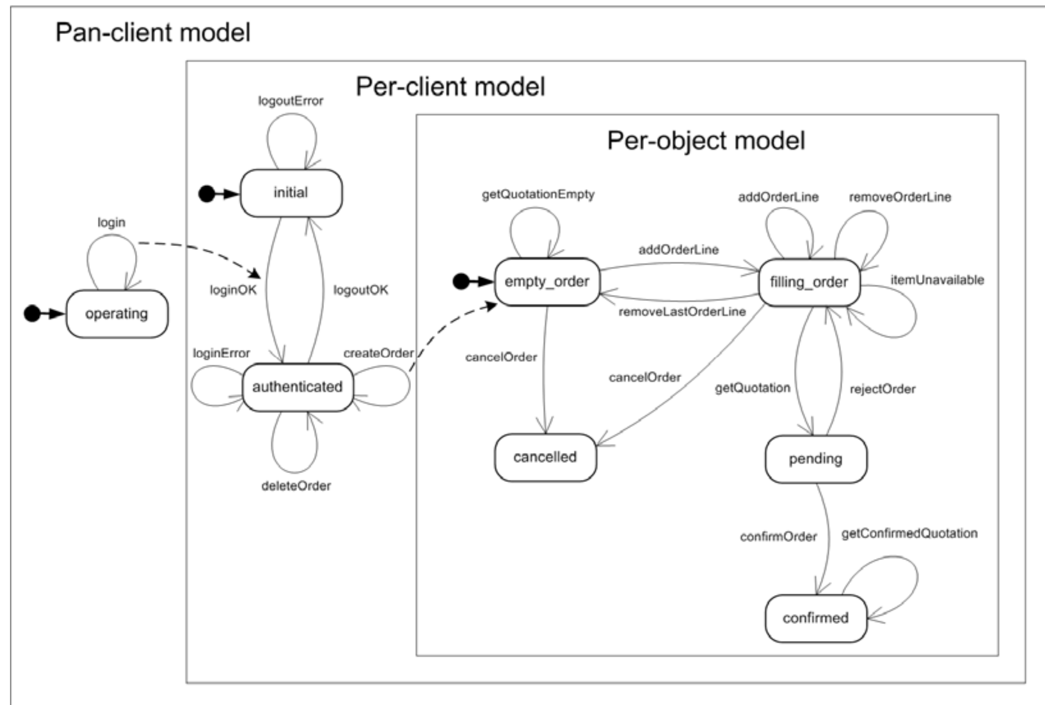


Figure 17 – Three different views adopted by the specifications of a SupplyOrder Web service with authentication functionality

One can imagine the three SXMs as operating concurrently; an input symbol may trigger transitions in all three machines simultaneously. The per-object machine represents the behaviour of one of multiple order instances in the memory structure of the per-client machine, which in turn represents the behaviour of one of multiple client state instances in the memory of the pan-client machine.

Transitions in one machine may relate to transitions or events in the other two machines. For example, a “login” transition in the single-state pan-client machine corresponds to a “loginOK” transition in the per-client machine. The “createOrder” transition in the per-client machine is associated with a creation event, which brings into existence the per-object machine for the newly-created order instance. In addition, the transitions in the order machine also correspond to transitions in the parent machines (not modelled to keep the diagrams simple). For example, input symbol (addItem, *itemId*, *qty*) triggers a transition labelled by “addOrderLine” in the per-object machine. However, in the parent machines the input symbol also has to include identifiers for the target order and/or client, thus it is of the form (addItem, *orderId*, *itemId*, *qty*) in the per-client machine, and (addItem, *authToken*, *orderId*, *itemId*, *qty*) in the pan-client machine. The identifiers filter the *scope* of processing functions to one object in the SXM memory.

Finally, it is necessary to define associations between the per-object SXM specification and the modelled Web service. This is necessary in activities that involve the invocation of the Web service, such as testing. The association can be accomplished through conventions that constitute a pattern. For example, the WS-Resource Framework described in section 0 uses the implied resource pattern, which defines identification mechanisms and Web service operations for creating, reading, updating, and deleting the modelled stateful resource. In this thesis we assume two generic patterns called the Manager pattern and the Constant Field pattern, described in chapter 0. The Manager pattern is outlined by Atkinson et al in [72] and defines a commonly-occurring relationship between a “manager” service interface and “managed” instances of a simple abstract data type (ADT). By “managing” it is meant that the service allows instances of the managed ADT to be created and destroyed and operations of the ADT to be applied to identified instances. In contrast, the Constant Field pattern, introduced in this thesis, does not assume that instances of the ADT are created or destroyed by service operations; thus the identifiers are static and known in advance.

The stateful object modelled by a per-object SXM is a special kind of object among others that are part of Web service state. Every input that triggers a transition in the per-object specification corresponds to an operation in the modelled Web service. Therefore, the per-object machine is *controllable* in practice, since it can be driven through different states and paths by appropriate operation invocations on the Web service under test.

5.5.2 Other abstraction techniques

A formal SXM specification represents a simplified view of the Web service implementation, so that it is more understandable and easier to validate against the user requirements. Unimportant details are left out of the specification in order to capture only the essential service behaviour.

Web service aspects that can be abstracted away include service functionality, inputs, outputs, and state. Abstraction of service functionality may involve leaving out some of the operations, which are not considered important and do not interfere with the modelled behaviour (see section 0). Other functionality is excluded by specifying only the success scenarios of service operations (section 0). In addition, since inputs, outputs, and state can be fairly complex in Web service implementations, it is often necessary to simplify their representation in the SXM specification and exclude certain data elements. Web service inputs and state are taken into account when computing outputs, thus they are factors that determine behaviour. Consequently, exclusion of input and state information from the specification may potentially result in nondeterminism, if the skipped information eventually affects produced outputs. In contrast to inputs and state, outputs may be abstracted in the specification, for comparison at an abstract level, without sacrificing determinism.

Subset of operations

In a number of situations it is possible to simplify the SXM specification of a Web service significantly by omitting functionality for one or more inputs. This means that the implementation of the missing functionality is not important in the context of the Web service under test. The missing functionality may correspond to a set of Web service operations, which are logically separated from the core behaviour, which comprises the operations that the modeller wishes to specify and test.

For example, the Amazon E-Commerce Service [56] consists of more than twenty operations, including operations for searching and browsing items, and five operations for managing shopping carts: CartCreate, CartGet, CartAdd, CartModify, and CartClear. If the modeller is interested in specifying and testing the behaviour relating to shopping carts, then only the functionality of those five operations involving shopping carts needs to be captured in the specification.

Theoretically, the SXM integration testing method assumes that both the implementation and the specification are of the same type Φ, i.e. the implementation *cannot* contain more functionality than the specification if they are to be equivalent. However, the Web service functionality restricted to a subset of the input alphabet (subset of the operations) can be logically isolated from the rest of the functionality. Consequently, the aim of testing becomes to verify the equivalence between the restricted Web service functionality and the abstracted SXM specification. Additional processing functions are allowed to be included in the implementation, as long as they are distinguishable from those in the specification [57].

Specifying success scenarios only

Input symbols representing invocations to the same service operation may trigger more than one processing function, depending on the current state and memory value. Those different possible processing functions are implemented by the same Web service operation and represent distinct scenarios after invoking the operation: some of them success and others failure scenarios.

Recall from section 5.4.5, that the processing functions labelling the transitions in the Account SXM state-transition diagram represent the success scenarios of the named operations. As will be further explained in section 5.7, such a specification is incomplete, since inputs are not handled when they trigger processing functions that represent failure scenarios. If the specification was completely defined, the state-transition diagram would also include numerous failure transitions, which would make it much more cluttered.

Therefore, it is generally considered as a good modelling and abstraction principle to specify only the important scenarios of operations, which are usually the success scenarios. The resulting state-transition diagram of the SXM is made more abstract and easier to understand. Its visual semantics represents the conversation protocol

that should be followed for successful conduction of conversations (protocol diagram), and is more appropriate for communicating it to human individuals who validate the service. This modelling practice is taken into consideration in section 5.6, where the modeller can start by determining the preconditions and effects of the success scenarios of individual service operations, from which a partially-specified SXM is inferred.

Data abstraction

SOAP request and response messages of realistic Web services, such as the UPS[14] Shipping Web service [58] or the Amazon E-Commerce service [56] often get enormously complex, including tens or even hundreds of XML leaf elements. In order to give an idea of the size and complexity that concrete SOAP messages can attain, Figure 18 shows only the first of ten pages containing the tree representation of a complex XML request in the UPS Shipping Web services documentation [58].

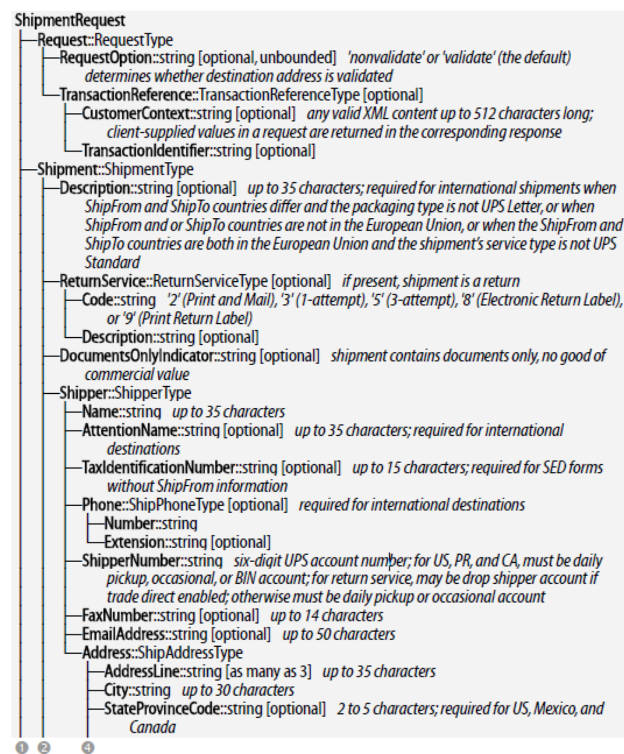


Figure 18 – Extract of the tree representation of the XML contents of a complex SOAP request message to the UPS Shipping Web service [58]

Needless to say that modelling all this complexity in input arguments and output results in the SXM specification is impractical, since it defeats the purpose of having an abstracted and understandable model of the service.

A large portion of the data fields should be abstracted away from the specification of input argument types and only those data fields that affect the behaviour one

[14] United Parcel Service, www.ups.com

wishes to test are captured. XML nodes in the request message, which can be left out include:

- optional nodes (e.g. `nillable=true` or `minoccurs=0` in XML Schema);
- nodes that can be calculated from other data in the message (e.g. a total of the costs of ordered items);
- nodes representing identification/authentication information, which is abstracted away from the specification as described earlier in this chapter;
- nodes that virtually remain constant and can be supplied during test case execution (such as access keys);
- nodes in the request message affecting a portion of the Web service state and functionality that is not captured in the specification.

Finally, it is possible to exclude from the specification input data fields that affect Web service behaviour and the computation of outputs, with the trade-off of making the specification nondeterministic (i.e. same abstract input symbol will potentially produce different output symbols for different values of the omitted data fields).

On the other hand, response message XML nodes should be captured in the specification of outputs only if their values are useful as oracles for the test. This decision depends on the level of granularity at which the tester wishes to make the comparisons between expected and actual outputs.

Apart from excluding data fields in the abstract input and output specifications it is possible to make use of enumerations. Enumerated inputs and outputs are discrete values (enumerated types or booleans) that represent ranges of values or complex XML data. During test case execution, those enumerated values in input symbols are replaced by carefully chosen example values or data instances, one for each enumeration. On the other hand, during outputs comparison, complex XML data instances are mapped to the corresponding abstract enumerations. The technical means for performing these mappings along with examples are further described in chapter 0.

5.5.3 Modelling large data repositories

As discussed in the previous chapter, Web service state often takes the form of complete databases, such as users or inventory databases. These types of Web services, which operate on large data repositories, have become fairly common over the Internet and in private SOA deployments as well. Since the contents of those repositories affect the outcomes of service operation invocations, they should be taken into account when modelling Web services. However, in realistic Web services databases are often exceedingly large and populated with thousands of data instances. Therefore, specifying all of their contents in the memory element of the SXM is not practical.

One possible solution to this problem, as further described in section 5.8, is to exclude large repositories from the SXM specification altogether, with the drawback of introducing nondeterminism in the specification.

The other solution proposed in this section is to capture only a portion of the repository in the initial memory m_0 of the SXM. This portion should consist of a small and manageable subset of all the data instances, which serve as sample values during animation and testing.

As an example, the SupplyOrder Web service introduced in the beginning of this chapter may consult an items inventory for availability, as items are added to the order and order quotations are requested. The items inventory is a shared data repository and includes information on all available items that can be ordered, thus it can attain exceedingly large sizes. Instead of modelling all the inventory items in the initial memory of the SXM, we propose specifying a small number of sample inventory items.

In SXMs the memory element has a generic type and does not prescribe any particular structure. Thus, it is difficult to write and understand the memory structure for complex Web services like SupplyOrder, in which state is structured as collections of objects. For this reason, the more convenient Object X-Machine approach to structuring memory is applied here for specifying the initial memory of the SupplyOrder Web service [54]. In accordance with object-oriented principles, Object X-Machines represent memory as three elements: the set of classes (C), the set of attributes (A), and the set of mappings from attributes to respective types (*type*). For the SupplyOrder example:

$$C = \{INVENTORY_ITEM\}$$

$$A = \{A_{INVENTORY_ITEM}\}\text{, where } A_{INVENTORY_ITEM} = \{itemId, availableQuantity\}$$

$$type_{INVENTORY_ITEM}(itemId) = STRING$$

$$type_{INVENTORY_ITEM}(availableQuantity) = N$$

The initial memory m_0 is then defined by the O_M memory constructor [54] as a set of INVENTORY_ITEM objects and sets of attribute-value pairs. For example, the set of inventory items could be initialised to three sample elements:

$$O_{INVENTORY_ITEM} = \{item_1, item_2, item_3\}$$

$$V_{itemId,INVENTORY_ITEM} = \{(item_1, \text{“I0001”}), (item_2, \text{“I0005”}), (item_3, \text{“I0010”})\}$$

$$V_{availableQuantity,INVENTORY_ITEM} = \{(item_1, 100), (item_2, 50), (item_3, 150)\}$$

As regards the state of the repository in the Web service implementation there are two possible scenarios:

- the service is a sandbox version under controlled test conditions;
- the service is operational and accessing the real items inventory.

If the Web service is deployed under test conditions, it is possible to initialise its inventory with the same sample items as those in the memory of the specification. Therefore, the implementation and the specification are equivalent.

In the second case, if no sandbox version is available for testing, the Web service is accessing the real items inventory. Thus, the initial memory in the specification represents a small portion of the items inventory in the implementation. Theoretically, this means that the specification is not only incomplete, but also incorrect, since the behaviour for adding items that are in the Web service inventory but not modelled in the specification is incompatible between the specification and the implementation. However, if the tester ensures that inputs chosen during testing operate only on the modelled portion of the inventory, then the implementation can be considered to be logically equivalent with the specification. The actual inputs are either produced by the test function (see chapter 0), or supplied by the mappings of abstract inputs to request messages that are sent to the service under test (see chapter 0). As an example, in the case of the addItem operation, the test function or the tester executing the tests selects:

- an *itemId* specified in the SXM memory, in order to exercise the transition for the case when given *itemId* exists;
- an *itemId* not existing in the Web service inventory (hence, SXM memory), in order to exercise the transition for non-existent *itemId*.

The usefulness of the above technique is that the tester is able to drive the different possible paths. In the case of the SupplyOrder Web service, the tester is able to test the addItem operation for both cases: when the provided itemId is available in the inventory, and when it is not available.

5.5.4 Specifying sample input values

It is common for realistic Web services to require input data that is unknown and difficult to generate randomly. Such data items are usually confidential and include: usernames, passwords, authentication tokens, access keys, credit card numbers, and so on. Therefore, in order to be able to test those services, the test inputs should include genuine values for confidential data.

This challenging problem is handled with similar alternatives to the ones described above for modelling large data repositories:

- create a nondeterministic specification;
- supply sample input data in the SXM specification;
- supply sample input data in the mappings that are used during test case execution.

As an example, assume that the SupplyOrder Web service requires user authentication before any other action can be performed. Authentication is achieved through the invocation of operation “login”, which takes two arguments: username and password. A correct username-password combination triggers the success

loginOK processing function, while an incorrect combination triggers the failure loginFailure processing function.

The first possibility is not to specify any sample login data at all. The domains of both loginOK and loginFailure processing functions are defined as the set of inputs with all possible username-password combinations. Thus, having two processing functions with the same (hence, overlapping) domains from the same initial state, the resulting SXM specification is nondeterministic. As it will be explained in section 7.3 on testing of nondeterministic SXMs, it is practically impossible for the test process to find genuine username-password combinations that can trigger the successful loginOK processing function. As a result, only conformance can be ensured, by exercising only the loginFailure processing function, without being able to drive the other paths that traverse the rest of the states and the success scenarios.

Since the previous alternative is not satisfactory for testing, the recommended solution is to supply sample input data in the SXM specification. Those sample input data are used to restrict the domains of the success and failure processing functions, so that the test function is able to derive inputs for exercising both functions. As in the previous section, it is suggested to specify the sample inputs in the initial memory m_0, which is accessed by the guard conditions of the processing functions. For the SupplyOrder example, the memory is initialised with a single user account that consists of a genuine username-password pair:

$O_{USER} = \{user_1\}$

$V_{username,USER} = \{(user_1, \text{“ervin”})\}$

$V_{password,USER} = \{(user_1, \text{“123”})\}$

Once again, the domain of the loginOK processing function is much larger in the Web service implementation than in the specification, since the service is expected to maintain several user accounts. Thus, theoretically, the specification is incorrect, since there are behaviour mismatches for those valid username-password pairs that are not specified in the SXM. It is the responsibility of the test function (or the tester) to avoid generating such inputs that cause incompatible behaviours. Nevertheless, being able to drive the success loginOK function makes it possible for the tester to drive the implementation through the rest of the states and the success scenarios.

The other possibility is to abstract the correct and incorrect sets of credentials as two enumerations in the specification and concretise them during test case execution. As will be described in chapter 0, the modeller specifies the sample input values in the transformation scripts.

It is a matter of decision whether the sample inputs information is included in the SXM specification itself or abstracted away and defined in the transformations. This usually depends on whether those input fields are considered as part of the business logic being modelled and tested. The choice is a trade-off between keeping the SXM specification abstract (leaving details to mappings during test case

execution) and making the specification self-sufficient (facilitating the mapping task).

5.6 Deriving a stream X-machine model from IOPE specifications

The task of modelling a stateful Web service as a stream X-machine is not straightforward, even for the developer of the service. It is possible to derive different SXMs that correctly specify the same Web service implementation, thus the decisions for arriving at the final specification are often a matter of testing priorities. The modelling process requires involves identification of states, transitions and other SXM elements, which has to be performed correctly.

The core step in creating a SXM specification is the identification of the control states, i.e. the set Q. Along with the transitions, the control states define the control flow (or, as referred to in the previous chapter, explicit choreography) of the Web service. The significance of each control state is that it characterises the subsequent behaviour of the modelled Web service, which is defined by the set of sequences of operation scenarios that can be invoked from that state. Therefore, it may be simpler for the modeller to start by defining the distinct scenarios followed by operation invocations, for each Web service operation. Usually, the modeller is interested in specifying only the success scenarios of every operation, which was considered as a good practice in section 5.5.2. It will be shown that having this information available, it is possible to infer the interesting control states of the SXM in a proper way.

Therefore, it is proposed that the modeller starts by declaring the success scenario of every operation in terms of its inputs, outputs, preconditions, and effects (IOPE). As mentioned in the previous chapter, the IOPEs of operations are also referred to as the implicit choreography of the service. Often, the preconditions and effects involve the internal service state, in addition to inputs and outputs: while preconditions check the values of state variables, effects update state variables. Some of those state variables represent control states in the final SXM. Hence, it can be observed that the preconditions for successfully invoking an operation define the pre-states of SXM transitions, while operation effects define the next states. Furthermore, the preconditions for successfully invoking an operation, and the effects produced by its execution, determine how the operation can be placed in sequences of invocations. As a result, some form of equivalence exists between implicit and explicit views on the service choreography (protocol).

A detailed transformation algorithm from IOPEs to SXMs has been described in a co-authored paper [52], which can be utilised by modellers to create SXM specifications. The difference is that the transformation described in the paper assumes that IOPE descriptions already exist as semantic annotations in WSDL. IOPEs are expressed in a rule language called RIF-PRD (Rule Interchange Format –

Production Rule Dialect), while the data model is expressed in terms of OWL ontologies.

The transformation to obtain all the constructs of the SXM specification is outlined by the following steps, while more details are given in the paper.

1. Identifying state variables
2. Partition analysis of state variables
3. Identifying preliminary states
4. Determining inputs and outputs
5. Determining transition pre-states
6. State merging
7. Determining transition next states
8. Determining memory
9. Determining guard conditions for processing functions
10. Determining memory updates for processing functions

There have been some similar attempts to derive formal EFSM specifications from IOPE-based descriptions of individual service operations. For example, Keum et al [50] outline a manual algorithm to derive a multi-state EFSM model from plain WSDL specifications, with extra information supplied by a human individual. However, the described algorithm focuses on the derivation of states, whereas little or no detail is provided on obtaining state transitions and other EFSM elements, such as the functions, memory constructs, inputs, and outputs. In a different approach, Sinha and Paradkar [51] make use of WSDL-S annotations of service operations with SWRL (Semantic Web Rule Language) rules to obtain an EFSM model to test the service. Nevertheless, the resultant model contains only one state, which is not sufficient to express the dynamic behaviour of the service and guide the generation of test sequences. On the other hand, the transformation described in Ramollari et al [52] is more complete, since it starts from IOPE descriptions of the success scenarios of service operations and infers all the elements of the SXM specification.

5.7 Controllability and completeness of specifications

Often, SXM specifications of Web services do not satisfy certain desirable properties, such as controllability (input-completeness), completeness of specification, and determinism. The first two are discussed in this section, while determinism is discussed in the next section.

The JSXM tools, described later, accept both non-controllable and partially specified SXM specifications for animation as well as for test case generation.

5.7.1 Controllability

As mentioned in section 5.3.2, a SXM is called input-complete (controllable) if it is possible to exercise any processing function from any memory value using appropriate input symbols, regardless of the state q.

Those processing functions that do not restrict their domains on memory values are always controllable, since they can always be exercised by any of the input symbols in their domain. On the other hand, controllability may not be satisfied when transitions define guard conditions that involve memory values.

For the Account SXM example presented earlier, processing functions open, getBalance, close and deposit do not define guard conditions on memory. As a result, they can always be exercised by input symbols (operation invocations): open, getBalance, close and (deposit, 5) $\in$ *dom*(*deposit*), respectively, thus they are input-complete. On the other hand, processing functions withdraw and withdrawAll contain guard conditions on the account balance in the memory. For example, withdrawAll requires the balance to be greater than zero, thus there are no input symbols that can exercise this processing function when the balance has a value of zero. As a result, processing function withdrawAll is not input-complete, which makes the whole specification not input-complete.

Controllability is a requirement for SXM integration testing, although test sets can also be derived for non-controllable SXMs, as described in the next chapter.

5.7.2 Completeness

Generally, a Web service following the request-response message exchange pattern responds to all invocations of its operations, either with normal or fault SOAP response messages. This means that the Web service implements functionality that handles all possible request messages defined in its WSDL description and does not ignore any of them. In order to fully capture such functionality in the SXM specification, the SXM should be completely defined as well. As can be recalled from section 5.3.2 on SXM properties, a completely defined SXM specification is one in which there is at least one possible transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$.

However, as explained in section 0, it is sometimes desirable to create partially specified SXMs for simplification purposes. Such specifications usually define transitions only for the success scenarios, while leaving out abnormal functionality that is not considered essential to testing. Input symbols, which consist of an input name and several optional arguments, may not be handled for certain values of the arguments (e.g. in the case of the withdraw input, when the amount is negative), for certain memory values (in the case of the withdraw input, when the available balance in the memory is less than the requested amount), and for certain control states (in the case of the withdraw input, when the state is other than “normal”).

The most usual case of incompleteness is when inputs are not handled in some control states. The intention is to specify in which of the states operations can be successfully invoked, so that the operation sequencing rules (conversation protocol or explicit choreography) are explicated in the state-transition diagram. Complying with the conversation protocol is essential for successful interoperability with the service. The diagram in Figure 19 portrays the partially specified SXM of the

Account Web service where inputs representing each operation are accepted only in some of the states.

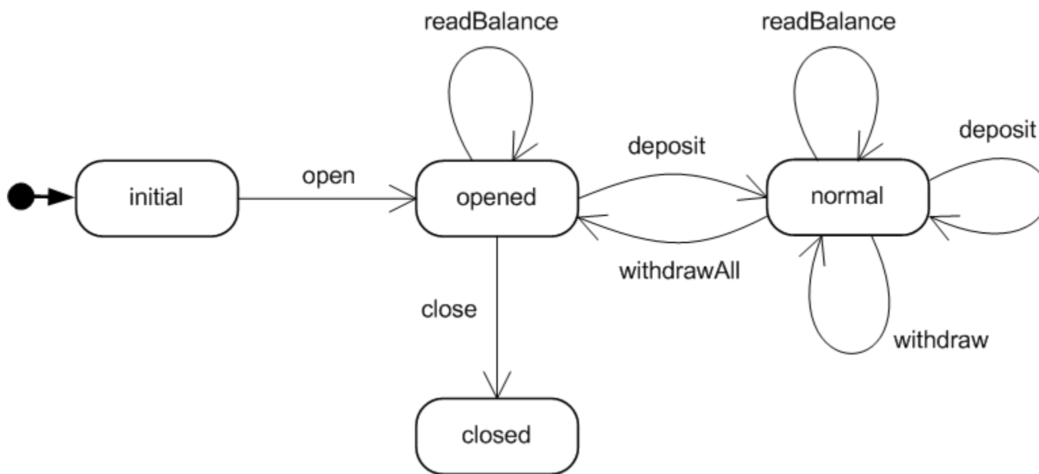


Figure 19 – Partially specified Account SXM

The next figure is the state diagram of a completely defined SXM model of the same Web service. In order to handle all inputs at all states, it defines labelled transitions for the error scenarios as well.

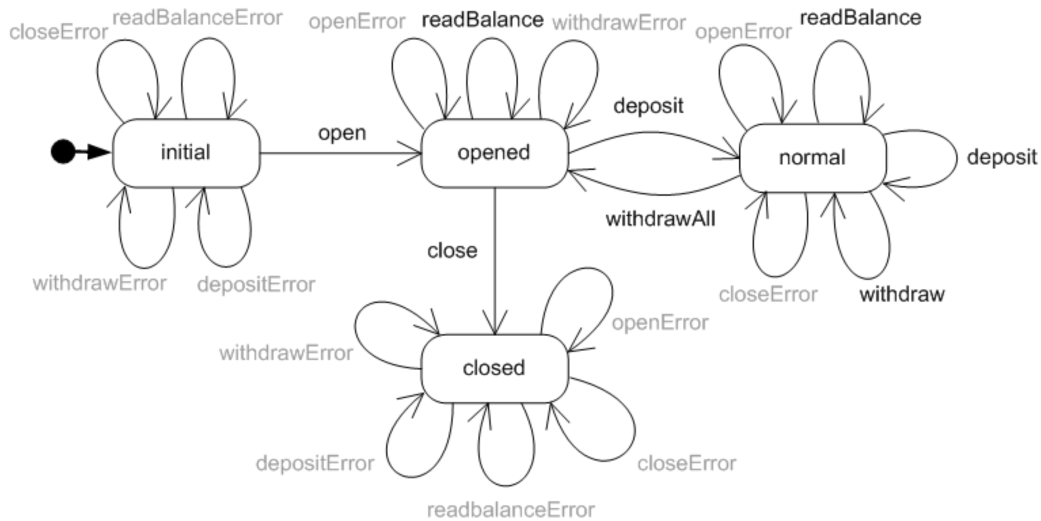


Figure 20 – Completely defined Account SXM

Making a SXM model of a Web service completely defined is not always practical. Such a specification must define in each state *at least* one transition for every operation. For card(Q) states and n operations, a completely defined SXM model will define at least card(Q)*n transitions. If we consider a Web service with 10 operations and 10 states, a completely defined SXM model would consist of at least 100 transitions, which make it quite complex and hard to understand. In addition,

the derived test set X will be much larger in terms of the number and length of sequences, thus increasing the cost of the testing process.

The semantics of an unhandled input, which does not cause any transition in the specification, can be interpreted in different possible ways:

- the input is ignored;
- computation is stopped;
- an error output is produced.

Since in the domain of Web services, normally no request message is ignored and the service continues to accept subsequent requests, the first two cases do not correspond with the actual behaviour of the implementation. Instead, the Web service implementation is expected to produce error SOAP responses or faults, to indicate that it is not being invoked according to the specified rules. For instance, if a requestor of the Account Web service tries to perform a deposit or withdrawal transaction in the initial state, i.e. before it is activated with the open request, the corresponding error or fault message will be returned.

As a result, this modelling approach adopts the convention that unhandled inputs produce error outputs and the SXM remains in the same state with an unchanged value of memory. The JSXM animation and test set generation tools described later in this chapter follow this convention, by generating default error messages as expected outputs for unhandled inputs. In this sense, the SXMs depicted in Figure 19 and Figure 20 are equivalent, with the difference that in the partially specified SXM the self-transitions for unhandled inputs are implicit, while in the completely defined SXM those transitions are modelled explicitly. On the other hand, modelling the failure scenarios explicitly also allows defining custom outputs, and even different next states and memory updates. Thus, the test case derivation method will consider the error transitions as part of the language of the machine and will include them in the produced test sequences. As a result, completely defined specifications are favourable for negative testing, when it is desirable to test the exact behaviour of handling abnormal invocations, especially when they trigger more complex behaviour associated with state transitions or memory updates.

5.8 Nondeterminism

5.8.1 Nondeterminism of implementations and specifications

Nondeterminism is a common phenomenon among real-world Web services. As a property, nondeterminism (and determinism) can refer both to (a) a Web service *implementation* and (b) its SXM *specification*.

Nondeterminism of service implementations is a topic that often causes confusion, since a service considered as deterministic from one’s perspective may be seen as nondeterministic from someone else’s perspective. In fact, determinism is not an intrinsic property of an implementation, but a judgment based on how much is

known about the service and its environment. For example, to a service requestor that knows only the external WSDL interface of a stateful Web service, the same input to an operation can produce different outputs at different times. Since the internal state maintained by the Web service is an unknown factor, the outputs seem to be randomly decided, and the Web service is considered as nondeterministic from the requestor's point of view. If, on the other hand, the requestor also knows the current state and how it affects the produced output, then the service becomes deterministic.

Recall from the previous sections that, in addition to the *input* and *internal state*, there are other factors that take part in determining the final outputs of Web services. Examples of such factors include:

- *shared state* that is accessed and potentially modified by several concurrent clients;
- *timing* constraints;
- *back-end applications* invoking the service operations or directly modifying its internal state;
- *human* and other *manual* factors;
- *other services* invoked by the (composite) service under consideration.

Once again, unless the requestor has *complete* knowledge of any of the above factors involved, and how they *exactly* affect the output, then the service is considered nondeterministic.

In the context of this thesis, only the input and the internal state are considered as deterministic factors, while the rest as nondeterministic. Therefore, we define a deterministic service implementation as one, which, for the same sequence of inputs and initial internal state, produces the same sequence of outputs. If this is not the case, i.e. if the application of a sequence of inputs from a given initial state produces different sequences of outputs at different times, then the service is defined as nondeterministic.

SXM specifications can also be deterministic or nondeterministic. The properties that must hold for a SXM to be deterministic were defined in section 5.3.1. They include determinism of the associated finite automaton, non-intersection of the domains of processing functions, and members of Φ to be functions rather than relations. If any of those properties does not hold, then the SXM specification is nondeterministic. Recall that in a nondeterministic SXM the initial state, the triggered processing relation, the produced output, the memory update, and the next state are uncertain.

Nondeterministic stream X-machines (NSXMs) can be used to specify the behaviour of both deterministic and nondeterministic service implementations. It is often impractical to create deterministic SXM specifications of complex, large-scale deterministic Web services. In these situations, nondeterministic SXM specifications make it possible to capture only the essential features and omit any

complex data structures and computations that determine the final outputs. Thus, nondeterministic specifications are more abstract and easier to understand.

As an example of the utility of NSXMs in modelling complex Web services, consider a more sophisticated version of the SupplyOrder Web service, which accesses an inventory of the available items and their corresponding quantities. The invocation of operation addItem first checks that the item of specified itemId is available in the inventory in the requested quantity. However, due to the huge size of the inventory and its nondeterministic nature (see below), it is impractical to model it in the memory of the SXM specification. Instead, the inventory is left out of the specification, and the resulting SXM, whose state-transition diagram is shown in Figure 21, is nondeterministic. An (addItem, *itemId*, *qty*) input triggers one of two possible transitions: one labelled “addOrderLine” if the item of given *itemId* is available in the requested quantity and the other labelled “itemUnavailable” if it is not available. The domains of these two processing functions are the same (any input symbols of input name addItem and any order contents in the memory), hence they are not disjoint. This form of nondeterminism is also known as domain nondeterminism.

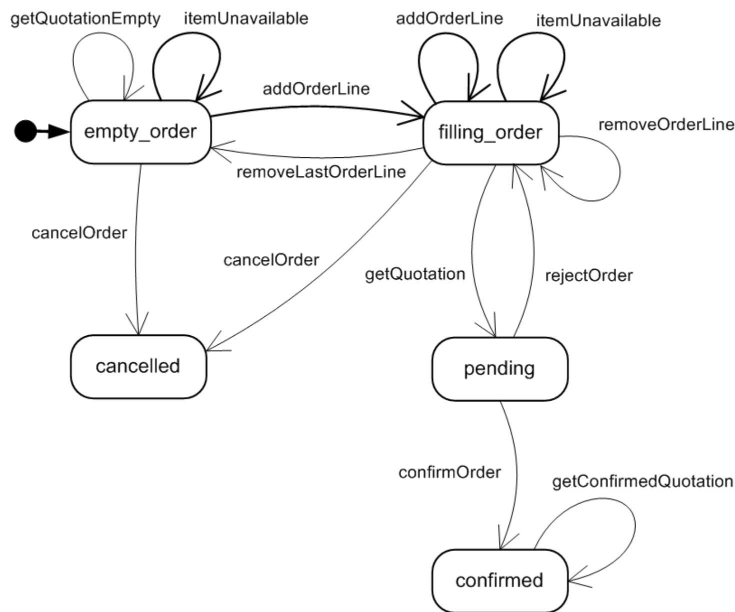


Figure 21 - State-transition diagram of a nondeterministic SXM specification modelling a stateful SupplyOrder Web service with inventory lookup. Notice the extra transitions labelled by function "itemUnavailable".

Nevertheless, the abstraction benefit of nondeterministic stream X-machines comes at the cost of losing modelling precision, uncertain prediction of outputs, and partial testing. Conformance testing, which is more practically applicable to nondeterministic specifications, does not possess the verification power of equivalence testing (section 7).

5.8.2 Shared-state Web services and nondeterminism

It would be possible to create a deterministic SXM specification of the SupplyOrder Web service with inventory lookup, if the inventory is modelled as well in the SXM memory. The inventory is *shared state*, since in a production deployment (interface) of the service it is possibly accessed and modified by other clients and even back-end inventory management applications. As a result, the state of the inventory at any given time is uncertain, the Web service implementation is nondeterministic, and its SXM specification becomes incorrect.

On the other hand, using a sandbox interface of the Web service under controlled testing conditions, it is possible for the tester to be the only client invoking the service. Under such conditions, the state of the inventory is known at any given time and the Web service behaviour appears to be deterministic. As described earlier in this chapter, it is also possible to populate the inventory of the sandbox interface with a few sample items, and model only those items in the SXM memory. Consequently, it becomes feasible to test the sandbox interface of the service for equivalence to its deterministic specification.

However, we pinpoint one problem with the above technique for testing shared-state Web services with deterministic SXMs. In the production interface, additional behaviour may emerge due to the nondeterministic invocations by other clients or applications. Suppose that, in the SupplyOrder Web service with inventory lookup, the getQuotation operation also checks for availability of the ordered items in the requested quantities. There are two possible outcomes: one that the items are still available from the last time they were checked by operation addItem, and the other outcome is that those items have been ordered in the meantime by other concurrent clients and are no longer available. As a result, apart from the successful getQuotation processing function, a new processing function has to be defined, e.g. getQuotationUnavailable. This additional behaviour is not exhibited in the single-client sandbox interface, since availability of items was already checked in the inventory during their addition to the order, so that a subsequent request for quotation is guaranteed to succeed. Therefore, the failure getQuotationUnavailable transition is extra behaviour, which is dependent on the simultaneous access by other clients and cannot be reproduced in the single-client interface.

In spite of the above problem, modelling the item inventory allows precise specification and testing of the behaviour of operation addItem. Therefore, in a number of occasions, the described technique for modelling and testing shared-state Web services with deterministic SXMs is considered useful.

5.9 Summary

This chapter presented some major contributions of this thesis, concerning specification of stateful Web services using SXMs, specifically contributions C2 and C3 of this thesis. In the beginning two Web service examples were introduced, which will be used for illustration throughout the rest of the thesis. Also, three state-

based formalisms were compared with one another: FSMs, EFSMs, and SXMs, while defending the choice of the SXM formalism for specifying Web service behaviour and data. Parallels were drawn between the SXM elements and their Web service counterparts in order to provide modelling insights. This chapter further described modelling practices in the domain of Web services, tackling a number of problems and unique service characteristics. A semi-automated method for derivation of SXM models from IOPE descriptions of service operations were also presented here. In addition, specific SXM properties, such as controllability and completeness of specification were critically investigated. Finally, this chapter provided an in-depth discussion of the notion of nondeterminism referring to Web services as well as SXM specifications.

Chapter 6 – Notation and Examples

This chapter describes the adopted language for serialising SXM specifications, namely JSXM, which is supported by a set of tools for model animation and automated test set generation. Example specifications in JSXM are listed in Section 6.2 for illustration of the JSXM notation.

6.1 Notation for defining stream X-machine models

Tools designated to automate different activities on SXM models, such as model animation and automated test set generation, have their own internal representations of the SXM mathematical model. However, descriptions in standardised languages are necessary for exchanging models between developers and tools and between developers themselves. That is, some form of interlingua is necessary for specifying and serialising SXM models.

Currently there exist two notations for specifying SXMs: XMDL and JSXM, which are described in the following two subsections. These two notations can be considered as complementary works, since they adopt very different styles to describing SXM models.

6.1.1 XMDL

The X-Machine Definition Language (XMDL) [53] is an interchange language for representing the SXM tuple elements in ASCII text files. Therefore XMDL files can be written in any text editor and interpreted by various automation tools. The full syntax of the language is defined in the XMDL User Manual [53].

A SXM written in XMDL consists of definitions corresponding to the elements of the SXM 8-tuple presented in section 5.3.1. Thus, the notation contains declarations for the input and output symbols, the set of states and the initial state, the memory tuple and initial memory, function definitions, and the transition function. The language also provides syntax for built-in types (such as integers, booleans, sets, sequences, bags, etc.), standard operations on these types, and the means for defining new types.

A more challenging task is the specification of processing functions, which can attain high levels of complexity. The XMDL notation makes it possible to define function domains on inputs and memory, computations of outputs, and memory updates. Function definitions in XMDL take two parameters: an input symbol and a memory value, and return two new parameters: an output and a new memory value. A function may be applicable under if-then conditions or unconditionally. Variables are denoted by a preceding “?”. The informative “where” in combination with the operator “<-“ is used to describe operations on memory values.

An extension to the XMDL language for supporting Object X-Machines, introduced earlier in section 5.5.3, is XMDL-O [54]. Thus, XMDL-O allows the specification of the memory element in an object-oriented style in terms of classes, attributes, and objects. As a result, the memory structure is derived by standard-object oriented design techniques (e.g. class diagrams) and the task is significantly simplified for the modeller.

A set of Prolog-based tools, collectively known as X-System, have been developed to support the XMDL notation [55]. X-System tools allow compilation and animation SXM models written in XMDL. Also, a Java-based graphical user interface on top of X-System is available. Nevertheless, no tool for generating test cases from XMDL descriptions has been yet developed.

6.1.2 FLAME

FLAME (*Flexible Large-scale Agent Modelling Environment*) is an agent-based modelling framework which allows defining multi-agent models based on the X-Machine formalism. The framework then generates code in the C programming language. This allows for detailed validation, systematic and formalised simulation and testing of multi-agent systems. The main advantage of using FLAME is that it produces models which are automatically parallelisable, which can thus allow simulations of high concentrations of agents to run on large scale mainframes, without effort required by modellers, and achieve results in finite time [101].

Agents in FLAME are based on communicating X-machines. They comprise of:

- Memory Variables
- Messages for communication
- Functions agents can perform

The FLAME framework uses a model *XMML file* and a functions file as inputs into the parser program, *XParser*. The XParser then converts the inputs into simulation code which with starting values can then simulate the model to produce results. The XMML definition is the X Machine Modelling Language which uses similar XML tags to code the model specifications.

The X Machine Modelling Language (XMML) of the FLAME framework is not considered as a candidate notation in this thesis, since it is specialised for the

specification of multi-agent systems, while in this thesis we aim to specify the behaviour of individual Web services.

6.1.3 JSXM

JSXM (Java Stream X-Machines) is an XML-based notation for defining SXM models [59]. The notation is supported by a set of tools having the same name. The JSXM tools can be used for SXM model animation, automated test set generation, and test transformation to executable tests in JUnit.

JSXM specifications are written in XML with some inline Java code for certain elements. A thorough presentation of the syntax of the notation is provided in the JSXM user manual [59], while specifications for the SupplyOrder and Account examples are given in the next section. Different sections of the specification define the different elements of the SXM 8-tuple: control states (including initial state), state transitions, memory (including initial memory), inputs and outputs, and processing functions.

As described in section 5.4, input and output definitions in JSXM consist of a name and one or more optional typed parts, called arguments in inputs and results in outputs. Thus, input/output symbols are mathematically represented as tuples of one or more elements, where the first element is the input/output name. JSXM makes use of XML Schema (XSD) types to define the types of input arguments and output results. The types can be built-in, as well as user-defined, which are supplied in separate XSD files.

A restriction on JSXM specifications is that every input definition must be associated with only one output definition, although it can also result in possible errors. That is, any two input symbols of the same name that are accepted by any two processing functions of the machine, must always produce output symbols of the same name:

$$\forall \sigma_1, \sigma_2 \in \Sigma \text{ with } \pi_1(\sigma_1) = \pi_1(\sigma_2),\ \forall \varphi_1, \varphi_2 \in \Phi, ((\exists m_1, m_2 \in M: (\sigma_1, m_1) \in \text{dom}(\varphi_1) \textit{ and } (\sigma_2, m_2) \in \text{dom}(\varphi_2)) \Rightarrow \pi_1(\pi_1(\varphi_1(\sigma_1, m_1))) = \pi_1(\pi_1(\varphi_2(\sigma_2, m_2)))).$$

Although this restriction is not part of the SXM formalism, it is convenient in specifying the behaviour of classes in object-oriented languages and Web services. A class method or Web service operation signature, which corresponds to a JSXM input definition, does define a single output type (Web services also define the name of the output root element, in addition to its XSD type). It would be illegal for Web service operations to return response messages of different root element names or XSD types, although they can return different fault responses.

Every processing function definition in JSXM consists of an *initialisation block* for binding variables, a *preconditititions block* of predicates for defining the function domain on Σ and M, and an *effects block* for defining the output computations and memory updates performed by the function. Predicates in preconditions and

computations in the initialisation and effects block are written in embedded Java code. The Java programming language has the advantage of being familiar among developers and possessing virtually unlimited expressive power for complex preconditions and effects. When function inputs (or outputs) consist of arguments (or results) of complex user-defined XSD types, they are in the form of XML documents. Those XML documents are accessed and manipulated in JSXM function preconditions and effects using JAXB bindings into Java objects [60]. If parallels are drawn with XMDL, the initialisation block is similar to the "where" block, the precondition expression is similar to the "if" block, and effect block is similar to the "then" block in XMDL. Unlike XMDL, where the definition of preconditions and effects closely resembles the mathematical notation, JSXM adopts the procedural programming style of Java. The provided inline Java code is directly reused by the JSXM tools to generate Java classes (as shown in Figure 22), which are compiled and exploited to animate the model and to generate test sets.

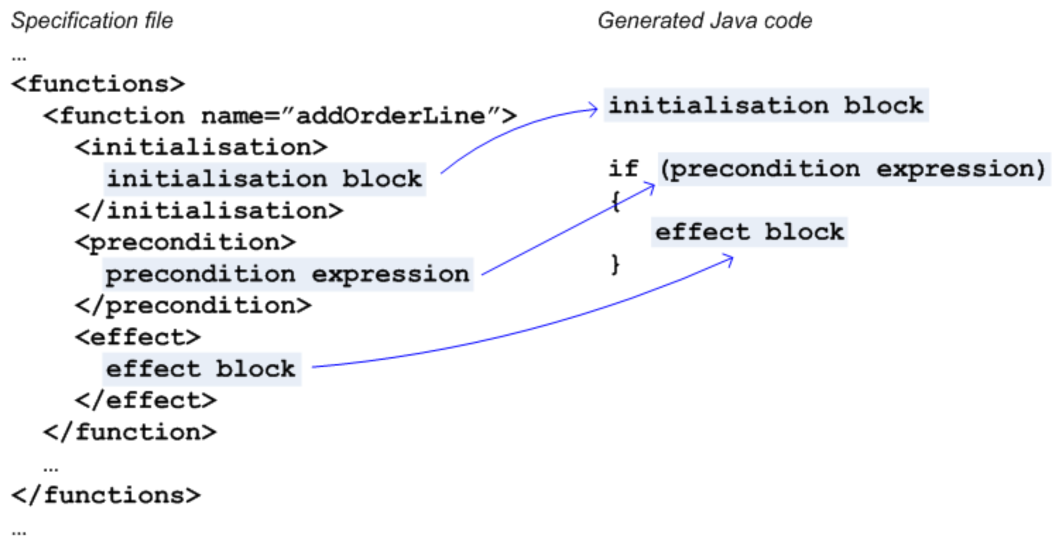


Figure 22 - Parts of a JSXM definition of a processing function and generated Java code

As with processing functions, in JSXM specifications the memory tuple is declared and initialised using inline Java code. Therefore, modellers can specify the structure and type of the memory element by taking advantage of the built-in Java types and the rich library of data structures, such as lists and maps.

Finally, JSXM allows specifying the interaction between two SXMs, by passing whole SXM models as inputs to processing functions. This interaction resembles the object-oriented model of method invocation, where an object may send a message to another object if it knows its identity [59]. The model of operation calls among interacting Web services, in service orchestration and choreography, is also similar. However, modelling interactions among services is out of the scope of this thesis.

There is the possibility for inconsistencies in the JSXM specification regarding the names of processing functions, states, and transitions, as well as definition of inputs, outputs, and memory. E.g. a processing function name mentioned in the definition of a transition may not appear in the processing function definitions section. Or, an input/output name mentioned in the definition of a processing function may not appear in the inputs or outputs definition sections. Also there is a great possibility for typos and errors in the XML specification syntax. Therefore, intelligent editors that validate the specification are needed as well as graphical modelling tools that hide most of the JSXM syntax.

6.1.4 Adopted notation

The earlier sections in this chapter described methods and best practices for modelling stateful Web services as SXMs. The next chapter presents techniques for testing from those SXM models. However, in addition to the SXM formalism, modellers and testers should agree on a common notation for representing SXM models. The two main factors that are taken into consideration are the appropriateness of the notation and the availability of tools.

The two alternative languages for representing SXM models are the ones described above, that is, XMDL and JSXM. XMDL defines a relatively mature and full-fledged syntax for built-in and new types, standard type operations, and for expressing processing function preconditions and effects. As expected for formal specifications, the language is close to mathematical notation and independent of any specific programming language. Moreover, the XMDL-O variant, which supports Object X-Machines, provides convenient syntax for structuring the memory element in an object-oriented manner. In spite of its strong points, XMDL introduces a relatively large amount of new syntax, which may involve a steep learning curve for the average developer. In addition, as of the date of this writing, there is not yet a tool for generating test cases from XMDL specifications.

On the other hand, JSXM defines a smaller amount of new syntax, which is based on XML elements and inline Java code. Since both of these languages are fairly familiar among developers, JSXM is expected to be easier to learn. As mentioned earlier, modellers can take full advantage of the expressive power of the Java programming language to define complex preconditions and effects in function specifications.

Furthermore, as explained earlier in this section, the JSXM specification approach is suitable for Web service modelling in a number of aspects. JSXM inputs represent operation invocations and outputs represent response messages more conveniently than generic tuples. Also, the use of XML Schema types to define arbitrarily complex inputs and outputs is advantageous, since it facilitates modelling and testing of Web services. Being defined by XSD types, the generated test inputs and expected outputs are XML instances, which have the potential to closely represent XML-based SOAP requests and responses of Web services. As regards

automation support, tools are available not only for animation of JSXM models, but also test set generation and transformation. Those tools are capable of handling specifications that are both uncontrollable (non-input-complete) and partially specified.

Therefore, in this research work, JSXM is the preferred notation for describing SXM models of Web services. JSXM is used for demonstration purposes and to describe the examples in the next section. The modelling, WSDL annotation, and testing approach presented in this thesis assumes that SXM specifications are expressed in JSXM. Also, as described later on, the tool developed as part of this research work for testing Web services extends the JSXM toolset.

6.2 Examples

This section presents extracts from the JSXM specifications of the two sample Web services introduced in the beginning of this chapter: Bank Account and Supply Order.

6.2.1 The Account example

First of all, JSXM defines XML elements to represent the states (and initial state) and transitions of a SXM. The following extract is the JSXM representation of the state-transition diagram for the Account SXM depicted in Figure 16.

```
<SXM name="Account">
  <states>
      <state name="initial" />
      <state name="opened" />
      <state name="closed" />
      <state name="normal" />
  </states>

  <initialState state="initial" />

  <transitions>
      <transition from="initial" function="open" to="opened" />
      <transition from="opened" function="close" to="closed" />
      <transition from="opened" function="deposit" to="normal" />
      <transition from="normal" function="deposit" to="normal" />
      <transition from="normal" function="withdraw" to="normal" />
      <transition from="normal" function="withdrawAll" to="opened"
/>
      <transition from="opened" function="getBalance" to="opened"
/>
      <transition from="normal" function="getBalance" to="normal"
/>
  </transitions>
  ...
</SXM>
```

The memory element, which consists of the account balance, is declared and initialized with Java statements, as follows. The `<display>` element is used by the animator to display the memory contents.

```
<memory>
  <declaration>
      int balance
  </declaration>
  <initial>
      balance = 0
  </initial>
  <display>
      balance
  </display>
</memory>
```

As described earlier, in JSXM, inputs and outputs are defined by a name and optional parts for complex inputs/outputs. The Account Web service consists of five operations, thus there are five inputs. As can be noticed, input and output definitions make use of XSD types, which in this case are primitive types.

```
<inputs>
  <input name="open" />
  <input name="close" />
  <input name="getBalance" />
  <input name="deposit">
      <arg name="amount" type="xs:int" />
  </input>
  <input name="withdraw">
      <arg name="amount" type="xs:int" />
  </input>
</inputs>
<outputs>
  <output name="openOut" />
  <output name="closeOut" />
  <output name="depositOut">
      <result name="amount" type="xs:int" />
  </output>
  <output name="withdrawOut">
      <result name="amount" type="xs:int" />
  </output>
  <output name="getBalanceOut">
      <result name="amount" type="xs:int" />
  </output>
</outputs>
```

Processing functions are declared in the `<functions>` section. Function definitions can be simple by specifying the inputs they accept and the outputs they produce, such as function “open” defined below. On the other hand, functions can contain complex preconditions on inputs and memory and effects on outputs and memory, such as function “deposit”. Java predicates are used to express function preconditions on input arguments and memory values, while Java code is used to assign new memory values and output results.

```
<functions>
  <function name="open" input="open" output="openOut"/>
  ...
```

```
  <function name="deposit" input="deposit" output="depositOut">
      <precondition>
            deposit.get_amount() > 0
      </precondition>
      <effect>
            balance = balance + deposit.get_amount();
            depositOut.amount = deposit.get_amount();
      </effect>
  </function>
  ...
</functions>
```

6.2.2 The SupplyOrder example

Figure 23 depicts the state-transition diagram for the SXM specification of a simple version of the SupplyOrder Web service, without inventory lookup.

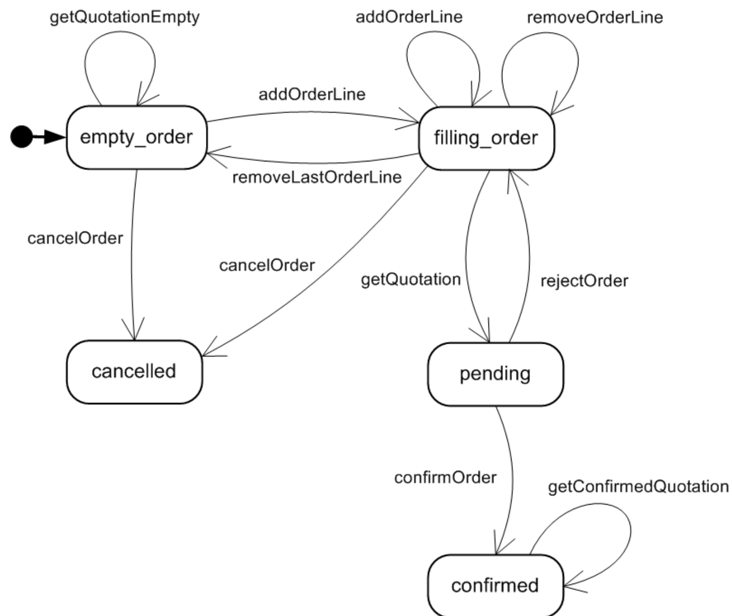


Figure 23 – State-transition diagram of the SupplyOrder SXM

The states and transitions of the SupplyOrder SXM are represented as follows:

```
<SXM name="SupplyOrder">
  <states>
      <state name="empty_order" />
      <state name="filling_order" />
      <state name="pending" />
      <state name="cancelled" />
      <state name="confirmed" />
  </states>

  <initialState state="empty_order" />

  <transitions>
      <transition from="empty_order" function="addOrderLine"
                  to="filling_order" />
      <transition from="empty_order" function="getQuotationEmpty"
                  to="empty_order" />
      <transition from="empty_order" function="cancelOrder"
```

```
                  to="cancelled" />
      <transition from="filling_order"
function="removeLastOrderLine"
                  to="empty_order" />
      <transition from="filling_order" function="addOrderLine"
                  to="filling_order" />
      <transition from="filling_order" function="removeOrderLine"
                  to="filling_order" />
      <transition from="filling_order" function="getQuotation"
                  to="pending" />
      <transition from="filling_order" function="cancelOrder"
                  to="cancelled" />
      <transition from="pending" function="rejectOrder"
                  to="filling_order" />
      <transition from="pending" function="confirmOrder"
                  to="confirmed" />
      <transition from="confirmed" function="getConfirmedQuotation"
                  to="confirmed" />
  </transitions>
  …
</SXM>
```

The memory tuple specifies the contents of the order as a collection of order lines in the form of (*itemId*, *quantity*) pairs, where *itemId* is a character string, while *quantity* is an integer. Therefore, it is possible to take advantage of the `HashMap` data structure provided by standard Java libraries, which defines a list of (key, object) pairs. The order is initialised to an empty map. Also, the JSXM memory declaration includes the `<javaImports>` element to allow defining any necessary import statements.

```
<memory>
  <javaImports>
      import java.util.HashMap;
  </javaImports>
  <declaration>
      HashMap<String, Integer> order;
  </declaration>
  <initial>
      order = new HashMap<String,Integer>();
  </initial>
  <display></display>
</memory>
```

If the modelled Web service also accesses an items inventory, the deterministic SXM specification would need to include another map of items and available quantities in the memory declaration and initialisation:

```
HashMap<String, Integer> inventory;
```

The SupplyOrder Web service consists of seven operations. Since there is a one-to-one correspondence between Web service operations and JSXM inputs, there are seven input declarations in the JSXM specification. The “itemRemoved” output, produced by a “removeItem” input, contains an integer result to indicate the number of order lines remaining in the current order. This additional information in the output is necessary to make the specification *output-distinguishable*, by being able to tell whether function “removeOrderLine”, or “removeLastOrderLine” (when all

items have been removed and the order is empty again), has been exercised. Similarly, output “quotation”, produced by input “getQuotation”, is abstractly defined by a message result to distinguish between function “getQuotation”, when a quotation is requested for a non-empty order, and the failure function “getQuotationEmpty” when a quotation is requested for an empty order. Notably, it would not have been possible to define two outputs of different names, e.g. “normalQuotation”, and “emptyOrderError”. As explained earlier, an input (in this case “getQuotation”) is associated with only one output (in this case “quotation”). In fact, in the modelled Web service, the “getQuotation” operation defines only one XSD type for the output, thus responses of different root element names are not allowed, unless they are fault responses.

```
<inputs>
  <input name="addItem">
      <arg name="itemId" type="xs:string" />
      <arg name="quantity" type="xs:int" />
  </input>
  <input name="removeItem">
      <arg name="itemId" type="xs:string" />
  </input>
  <input name="getQuotation" />
  <input name="getConfirmedQuotation" />
  <input name="rejectOrder" />
  <input name="confirmOrder" />
  <input name="cancelOrder" />
</inputs>
<outputs>
  <output name="itemAdded" />
  <output name="itemRemoved">
      <result name="itemsRemaining" type="xs:int" />
  </output>
  <output name="quotation">
      <result name="message" type="xs:string" />
  </output>
  <output name="confirmedQuotation" />
  <output name="orderRejected" />
  <output name="orderConfirmed" />
  <output name="orderCancelled" />
</outputs>
```

Due to space constraints, only the definition of function “removeLastOrderLine” is provided here. This function is triggered by input “removeItem”, which consists of an integer argument for the *itemId* of the item to remove. The result of exercising this function is that the last (*itemId*, *quantity*) pair is removed from the order, resulting in an empty order. As the extract below shows, the function precondition checks that the *itemId* argument (accessed by the “.” operator) is contained in the memory map and that there is only one item in the order. The effect is that the item of given *itemId* is removed from the order. Also, in the produced “itemRemoved” output, the integer “itemsRemaining” result is assigned the number of items (order lines) remaining in the order.

```
<function name="removeLastOrderLine" input="removeItem"
      output="itemRemoved">
  <precondition>
```

```
      order.containsKey(removeItem.get_itemId()) && order.size() ==
1
  </precondition>
  <effect>
      order.remove(removeItem.get_itemId());
      itemRemoved.itemsRemaining = order.size();
  </effect>
</function>
```

Suppose that the modeller would like to specify the “quotation” output to be more complex than a simple message, e.g. to contain a list of the current items in the order. In this case, the output could be defined with two results, where the second result is of a complex user-defined XSD type, `QuotationType`:

```
<output name="quotation">
  <result name="message" type="xs:string" />
  <result name="items" type="QuotationType" />
</output>
```

`QuotationType` is defined in a separate XML Schema document as follows:

```
<xs:schema>
  <xs:complexType name="QuotationType">
      <xs:sequence>
            <xs:element name="orderLine" type="OrderLineType"
                  minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
  </xs:complexType>

  <xs:complexType name="OrderLineType">
      <xs:attribute name="itemId" type="xs:string" />
      <xs:attribute name="quantity" type="xs:int" />
  </xs:complexType>
</xs:schema>
```

In the definition of function “getQuotation”, the “items” result is an XML document, and is instantiated, accessed and edited as a Java object according to the JAXB conventions [60]:

```
<function name="getQuotation" input="getQuotation"
      output="quotation">
  <effect>
      quotation.message = "successful";

      QuotationType quotationItems = new QuotationType();
      for (String id: order.keySet() ) {
            Integer qty = order.get(id);
            OrderLineType orderLine=new OrderLineType();
            orderLine.setItemId(id);
            orderLine.setQuantity(qty);
            quotationItems.getOrderLine().add(orderLine);
      }

      quotation.items = quotationItems;
  </effect>
</function>
```

Other variations of the SupplyOrder Web service

Several variants of the SupplyOrder Web service have been implemented, with increasing levels of complexity, which require additional specification and testing effort (Table 2). Besides, three different versions of the SXM specification have been created: partially specified, completely specified, and one with inventory lookup.

The simplest SupplyOrder implementation is a single-order Web service, which exposes operations that access and modify that single order instance. This Web service stands at the same level of abstraction as the SXM specification, thus the generated test inputs can be fed directly to the service, and the produced outputs can be compared directly to the ones predicted by the SXM specification. Three other variants of the SupplyOrder Web service maintain multiple order instances, one for each client: one implemented through HTTP sessions, the other one through SOAP body identification, and the third one complying with the Manager pattern regarding identifiers. The last two introduce an abstraction gap between inputs/outputs in the implementation and those in the specification. Consequently, the derivation of concrete inputs to feed to the Web service and the handling of produced concrete outputs require mappings, which are described in section 0. Other versions of the Web service implementation support WSDL/SOAP faults (discussed in the next chapter), and inventory checking for item availability. Finally, the single-order Web service has been seeded with various implementation faults in order to demonstrate the ability of the SXM testing method to reveal meaningful faults, especially in conversational Web services. These faulty versions are further described in the next chapter.

Table 2 - Versions of SupplyOrder Web service specification and implementation

SXM specification			Web service implementation						
	completely defined	inventory lookup		multiple-object with sessions	multiple-object with identifiers	Manager pattern	SOAP/WSDL faults	Inventory lookup	faulty*
1			**1**						
2	✓		**2**	✓					
3		✓	**3**	✓	✓				
			4	✓	✓	✓			
			5	✓			✓		
			6	✓				✓	
			7						✓

* Implementation injected with one of the following control-flow faults: extra/missing transition, erroneous transition label, erroneous next state, missing state, one extra state, two extra states.

6.3 Summary

This chapter presented and defended the use of the JSXM notation for specifying SXM models, which was contrasted with the alternative XMDL notation. The JSXM notation was also used to specify the two example Web services: Account and SupplyOrder, thus relating to contribution C10 listed in section 1.3. Having a SXM model of the WSUT specified in a machine-processable notation, it is now possible to apply a test set generation method that proves the correctness of the implementation (the subject of the following chapter). The JSXM notation described in this chapter is also revisited in chapters 9 and 10, where the technical approach for testable Web services and the toolset implementation are described.

Chapter 7 – Testing Web Services Modelled as Stream X-Machines

One of the main benefits of formally specifying a Web service with a stream X-machine is the ability to *test* the correctness of the Web service implementation against the specification, which is also the focus of this thesis. The formal SXM specification serves a number of important roles during the process of testing a Web service, including:

- Derivation of test sequences (scenarios) of operation invocations from paths in the specification machine.
- Generation of test input data to drive the test sequences.
- Deciding when testing should stop, in accordance with the coverage criteria of the SXM-based test generation method. Testing stops when all test sequences derived from the specification have been exercised.
- Acting as a test oracle for defining the expected outputs to be compared with the actual outputs returned by the Web service implementation, during the execution of the test cases. In the case of deterministic specifications, one expected output is defined, while in the case of nondeterminisitic (stochastic) specifications, a set of expected outputs is defined.

SXMs are associated with test generation techniques that are guaranteed to determine the correctness of the SUT, provided that certain well-defined assumptions hold [61]. The correctness of third-party Web services is considered a crucial attribute when they are integrated in critical applications. Different variants of SXM-based testing (SXMT) methods have been proposed, which basically differ according to the variant of SXM being employed and the characteristics of the specification. Nevertheless, all of those methods are generally founded on the (D)SXM integration testing method, which is supported by extensive research and theoretical underpinnings [62], [61].

The focus of this chapter is on the application of the SXMT method to generate test sets for Web services, and is structured as follows. The first section is an overview of the SXM integration testing method. It describes the process of deriving sequences of processing functions from machine paths and converting them to

sequences of inputs. Given that specifications can often be partially specified and not input-complete, a modified variant of the test generation method is described to handle those cases as well, which is also supported by JSXM. The second section describes derivation of expected outputs, test case execution, and different issues arising during this activity. The third section focuses on the effectiveness of SXMT in accomplishing negative testing, and clarifies the meaning of error outputs for Web services, and their representations in SXM models. The next section illustrates the different steps during the test set derivation process using the previously described Bank Account and Supply Order Web service examples. The SXM testing method is to be applied to Web services, which possess unique characteristics that make them different from traditional software systems. Web service testing is potentially performed at run time, and they often are made available across organisational boundaries. Therefore, section 5 of this chapter explores the issues that need to be considered in the domain of Web services, and suggests solutions to those problems. Finally, the last section of this chapter investigates the ability of SXMT to reveal faults, using the Account and SupplyOrder examples. The implementations are seeded with manual and meaningful business logic faults, both control-flow and individual processing function faults. In addition, the strength of the generated test cases is evaluated for different values of the parameter *k*, utilising the automated mutation testing tools Jumble and Jester.

7.1 The Stream X-Machine integration testing method (SXMT)

7.1.1 Theoretical basis

Stream X-machines are associated with a test set generation method [62], [61], which under certain assumptions, is proven to find all faults in the implementation. Examples of such faults include erroneous transition labels, erroneous next-states, missing states, extra states, etc [82]. The testing method is a generalization of Chow's W-method [63]. It works on the assumption that the system specification and the implementation can be both represented as SXMs of the same type (i.e. both specification and implementation have the same set of processing functions) and satisfy the following *design for test* conditions: input-completeness (controllability) and output-distinguishability (observability). As defined in section 5.3.2 on SXM properties, input-completeness ensures that all processing functions can be exercised from any memory value using appropriate input symbols. Output-distinguishability ensures that any two different processing functions will produce different outputs if applied on the same memory/input pair.

When the above requirements are met, the stream X-machine integration testing method may be employed to obtain a complete test set of input sequences which can be used for the verification of the implementation under test. It is proved that only if the specification and the implementation are behaviourally equivalent, the test set

produces identical results when applied to both of them. Otherwise it is guaranteed that it will expose the faults in the implementation.

7.1.2 Derivation of sequences of processing functions

The first step to constructing the test set of input sequences is based on the application of the W-method to the associated finite automaton of the specification machine, where processing functions are treated as input symbols. It involves the derivation of a characterization set W and a state cover S of the associated finite automaton. A characterization set W is a set of sequences of processing functions for which any two distinct states of the machine are distinguishable and a state cover S is a set of sequences of processing functions such that all states are reachable from the initial state. In addition, an estimate is made on the maximum difference, k, between the number of states in the implementation, and the number of states in the specification. In practice, the value of k is not usually large; for especially sensitive applications one can make very pessimistic assumptions about k at the cost of an exponentially larger test set [57].

Given the above definitions, the test set Y for the associated finite automaton consists of sequences of processing functions and is given by the formula:

$$Y = S(\Phi^{k+1} \cup \Phi^{k} \cup \ldots \cup \Phi \cup \{\epsilon\})W$$

where the concatenation of any two sets of sequences, U and V, is defined by $UV = \{ab \mid a \in U, b \in V\}$. Also, for any set of sequences, U, U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$ for $n \geq 1$. Derivation of the sets S, W, and Y for the Account and SupplyOrder Web service examples is presented in section 7.3.

As can be observed from the above formula, the state cover S is included in all test sets. Thus, Y is minimally characterised by *state coverage* and contains sequences that reach every state $q \in Q$ in the machine. Also, for the minimal value of $k = 0$, the state cover is appended with the set of processing functions Φ (as well as with the sequences from the characterisation set W for distinguishing states). This means that the sequences of the test set reach all the states of the machine, and from each state all possible transitions labelled by elements of Φ are exercised. Therefore, Y is also minimally characterised by *transition coverage*. Similarly, for a value of $k = 1$, from every reached state all possible transition pair combinations (elements of Φ^2) are exercised, resulting in *transition-pair coverage*. Thus, larger values of k, apart from reaching any redundant states in the implementation, also allow traversing paths of several adjacent transitions.

The scope of the testing approach in this thesis covers the DSXM integration testing method, which assumes the individual processing functions $\varphi \in \Phi$ have been correctly implemented. Nevertheless, it is still possible to test the machine at the level of individual processing functions through enhanced SXM specifications and the application of suitable variants of the SXMT method, such as the complete DSXM testing method in [65], and the method in [82]. These two methods are not supported by the JSXM test case generation tool and are not discussed in this thesis.

7.1.3 Derivation of sequences of test inputs

The sequences of processing functions from the set Y derived in the previous section have to be converted to sequences of inputs, which comprise the test set X. This is achieved by a mechanism called *the test function*: t: $\Phi^* \rightarrow \Sigma^*$. For a SXM Z, a test function is defined as follows [61]:

- $t(\epsilon) = \epsilon$;
- for $\varphi_1, \ldots, \varphi_n$, with $n > 0$, $t(\varphi_1 \ldots \varphi_n) = \sigma_1 \ldots \sigma_k$, where $\sigma_1, \ldots, \sigma_k \in \Sigma$ are such that $(m_0, \sigma_1 \ldots \sigma_k) \in \mathrm{dom}(\|\varphi_1 \ldots \varphi_n\|)$ and k is as follows:
    - $\varphi_1 \ldots \varphi_n \in L_Z \Rightarrow k = n$;
    - $\varphi_1 \ldots \varphi_n \notin L_Z \Rightarrow k = i + 1$, where $0 \leq i < n$ is such that ($\varphi_1 \ldots \varphi_i \in L_Z$ and $\varphi_1 \ldots \varphi_{i+1}) \notin L_Z$

where L_Z is the *language* accepted by the associated finite automaton of Z. In other words, for a sequence of processing functions, t finds a sequence of inputs that exercises the longest prefix that is a path in the machine and, if that prefix is shorter than the original sequence, it also exercises the function that follows this prefix. The input-completeness condition ensures that there always exists a sequence of inputs as defined above.

Notably, a test function finds input symbols for the sole purpose of exercising each processing function in a sequence. If there is a finite or infinite set of input symbols that can trigger a function, the input is selected arbitrarily and possibly randomly. There is no heuristic or data coverage criterion to choose good data values for the test input symbols. Other data coverage testing methods could be possibly used for such a purpose, such as equivalence class or boundary-value testing.

In JSXM specifications the test function is defined in the `<testinputgeneration>` section. Input generators are declared for every processing function that receives complex inputs. On the other hand, processing functions that receive simple inputs do not need input generators in the specification, since simple inputs define single input symbols. In theory, the information contained in the SXM specification should be adequate for the tool to generate test inputs, without having to rely on a `<testinputgeneration>` section. However, in practice, the process is time-consuming and hard to automate. The test generators provide shortcuts to generate real test data, either randomly, or according to specified criteria, such that they satisfy the preconditions to exercise the intended processing functions. The shortcuts are in the form of Java code to execute to generate the test data, so that JSXM does not have to check the satisfaction of preconditions on input values and current memory state.

Thus, the derived input sequences constitute the final test set X, which are applied both to the specification and the Web service under test to reveal all implementation faults.

7.1.4 Test case generation for non-controllable and partially-specified specifications

As discussed in section 5.7, SXM specifications do not always satisfy certain properties, such as completeness of specification and controllability. Nevertheless, it may still be possible to apply the SXM integration testing method in such cases.

It is not always feasible for SXM specifications to satisfy the design-for-test conditions, especially the controllability requirement that any processing function can be triggered for any memory value by selecting appropriate inputs. In practice, this condition can be achieved by designing extra functionality that will have to be disabled after testing has been completed. However, this can often be a time consuming and error-prone process. In addition, when testing third-party Web services, the tester does not have control on the implementation of the WSUT and cannot ensure the satisfaction of the design-for-test conditions.

It is not always necessary to be able to trigger processing functions from Φ for *every* possible memory value, if the memory values produced by prefixes of function sequences are restricted within a range. However, the derivation of input sequences is complicated by the fact that whole sequences of functions, rather than individual functions, should be taken into account. If inputs are derived incrementally for each processing function in a sequence, it is possible that at a point no input exists to trigger the next function for the current memory value produced by the prefix. Therefore, a different sequence has to be tried from the beginning with new input symbols. This makes input generation a complex process which is especially difficult to automate.

The input sequence derivation algorithm for non-controllable specifications is simplified significantly if the specification is *input-uniform*. Basically, input-uniformity requires all memory values that are produced by the application of any single sequence of processing functions to any single memory to be processed in a uniform way by any processing function - that is, any function can either process all such memory values or none [83]. In this case, appropriate input symbols can be selected one at a time for each processing function in the sequence without having to know the processing functions that will be applied next. The Account SXM example described in the previous chapter is an example of non-uniform non-controllable specification. The selected amount for exercising function “deposit” matters, since it determines how many times function “withdraw” can be subsequently invoked.

The JSXM test case generation tool supports specifications that are not input-complete. However, during derivation of test inputs, JSXM selects input symbols for each processing function in isolation from the rest of the sequence, based only on the knowledge of the current memory [59]. Thus, JSXM assumes that non-controllable specifications are input-uniform. If the specification is not input-uniform, the tool will fail if the first attempted input sequence cannot be completed for a function sequence, even though other attempts might have succeeded. The

result is that the function sequence is considered as *not applicable* (no input sequence exists to exercise it) and is reduced to applicable by removing the function that is not admitted.

Furthermore, as explained in section 5.7.2, SXMs are quite often partially-defined. Even though in a partially-specified SXM some inputs are not handled, the test function t defined earlier selects inputs such that they can exercise the intended processing functions for the current memory values m (regardless of the current state). As a result, the only case when inputs from the test set are not handled is when the intended processing function is not accepted at the current state q. According to the definition of the test function in the previous section, such unhandled inputs occur as the last elements of sequences that are not in the language L of the machine. There are two possible outcomes for those inputs:

- A different processing function is actually triggered in the machine, if the specification is completely-defined. For example, in the completely-defined Account SXM in Figure 20, function sequence <open, open> is not in the language of the machine. However, since processing function "open" is input-complete (no preconditions on memory), the input "openRequest"[15] is derived by the test function. Thus, input sequence <openRequest, openRequest> is generated by t, which, when animated, exercises processing function sequence <open, openError>. The second "openRequest" is indeed handled, by processing function "openError", because the SXM is completely defined and handles input "openRequest" at any state.
- No processing function is triggered in the machine if the specification is partially-specified, thus the input is not handled. In the partially-specified Account SXM in Figure 19, the second input in the sequence <openRequest, openRequest> is not handled by any processing function. As described in section 5.7.2, the convention adopted for unhandled inputs is that they cause exceptional conditions that result in error or fault messages, thus the animation of the SXM should also produce a *default error*, for instance, "open_Error".

The last input that attempts a transition not in the language of the machine is included in order to check that the attempted transition is not exercised in the implementation. A correct implementation either activates a different processing function, or returns an error. The test set performs this check for all missing transitions (since the W-method reaches all states, and from each state it attempts every member of Φ, regardless of whether it is accepted or not). Therefore, SXMT verifies that the implementation does not take transitions that it is not supposed to take (*negative testing*). No further unhandled inputs are tried after the first one, since the SXM may have transitioned to an unknown state.

[15] Input "open" has been renamed to "openRequest" in this discussion in order not to confuse it with the processing function of the same name.

The JSXM animator and test case generator tools accept partially-specified SXMs and deal with unhandled inputs as described above. That is, whenever the JSXM animator encounters an input that does not trigger any function in the specification, it yields a default error output, “open_Error”. Test set generation for the Account and SupplyOrder SXMs, which are partially-specified and non-controllable, is described in section 7.3 of this chapter.

7.2 Test case execution

7.2.1 Overall process

Having derived the test set of abstract input symbols, as described above, it is necessary to execute them on the Web service under test. According to [33], test execution involves the development of a test environment in which the tests can be executed, the actual execution of the tests, analysis of the execution results, and the assignment of a verdict about the well-functioning of the Web service under test

As will be described later on, test cases can be either executed on a sandbox version of the Web service, which represents the test environment, or on a production deployment of the Web service. The first scenario is applicable when testing is performed during development time or during run time if the provider makes available a sandbox interface. If the first scenario is not applicable, this means that test cases have to be executed at runtime on a production deployment of the third-party Web service, for which the service provider does not make available a sandbox interface.

The derived sequences of abstract input symbols must eventually be translated to corresponding sequences of operation invocations on the WSUT. As explained in section 5.5, the SXM specification represents a simplification of the modelled Web service in a number of aspects (single stateful resources, functional abstraction, data abstraction, etc). Consequently, the derived input symbols stand at a higher level of abstraction than the corresponding request messages. For this reason, additional effort is required to map abstract inputs to concrete request messages to be dispatched to the corresponding Web service operation. This problem is further investigated in chapter 0.

In addition, while test cases are being executed, the tester needs to know the expected outputs for comparison with the actual outputs returned by the WSUT. Derivation of expected output sequences is further examined in the following subsection.

7.2.2 Derivation of expected outputs

SXM-based testing is a *black box* testing method. Thus, all that can be observed are the inputs sent to the SUT and the outputs it produces, without any knowledge of the internal implementation. The input sequences from the test set X are applied to both specification and implementation machines and the produced output sequences

are compared to assign a verdict. If the output sequences are identical, then the two machines are considered to be equivalent (see above the definition of a test set in SXMT), otherwise, the SUT contains implementation faults. Therefore, the specification machine, previously utilised to generate test sequences, also serves as the *test oracle* to determine the correct output sequences. As defined earlier, a test oracle is a mechanism for predicting the outputs that are expected from a correct implementation.

Nevertheless, the application of input symbols to the specification and the implementation machines does not have to be simultaneous. In the testing approach described in this thesis, the input sequences are applied offline to the SXM specification, before the actual test case execution. The derived sequences of expected outputs form part of the abstract test cases, which can be later executed on the WSUT.

Theoretically, it is possible to apply the function (*f*) computed by the machine on the input sequences manually. However, this is a time consuming and error-prone process which requires automation support. The JSXM toolset incorporates a *model animation* tool, which receives sequences of input symbols and produces sequences of output symbols. This tool is also utilised by the test case generation tool to produce extended test sets that contain sequences of input-output pairs.

7.3 Examples

The Account and SupplyOrder SXM examples described in the previous chapter are non-controllable and partially-specified. The following two subsections illustrate the process adopted by JSXM to derive a test set of sequences of input symbols for such specifications.

7.3.1 The Account example

The first step toward the generation of test input sequences is the application of the W-method to the associated finite automaton of the Account SXM, depicted in Figure 16. The state cover S needs to reach all four states and therefore consists of four sequences:

S = {ϵ, <open>, <open, close>, <open, deposit>}

The characterisation set needs to contain sequences that distinguish between four states, thus three sequences are adequate:

W = {<open>, <close>, <deposit>}

The set Φ consists of the six processing functions labelling the transitions in the associated FA:

Φ = {open, close, getBalance, deposit, withdraw, withdrawAll}

If we assume that the implementation does not contain any extra states, i.e. $k = 0$, the set of processing function sequences Y to be traversed is given by:

$Y = S\{\Phi \cup \{\epsilon\}\}W = S\Phi W \cup SW$

= {ϵ, <open>, <open, close>, <open, deposit>}Φ{<open>, <close>, <deposit>} ∪ {ϵ, <open>, <open, close>, <open, deposit>}{<open>, <close>, <deposit>}

= {<open, open>, <open, close>, <open, deposit>, <close, open>, <close, close>, <close, deposit>, <getBalance, open>, <getBalance, close>, <getBalance, deposit>, <deposit, open>, <deposit, close>, <deposit, deposit>, <withdraw, open>, < withdraw, close>, < withdraw, deposit>, <withdrawAll, open>, < withdrawAll, close>, < withdrawAll, deposit> …} ∪ {<open>, <close>, <deposit>, <open, open>, <open, close>, <open, deposit>, <open, close, open>, <open, close, close>, <open, close, deposit>, …}

Therefore, for k=0, the test set is essentially a transition cover of the model, appended with characterisation sequences to distinguish the final states. For k=1 the resulting test set is characterised by transition-pair coverage, again appended with characterisation sequences.

When processing function sequences from Y are not in the language of the machine, the test function t generates input sequences for prefixes $\varphi_1 \ldots \varphi_{i+1}$, where $\varphi_1 \ldots \varphi_i \in L_Z$. Therefore, the JSXM test set generation tool cuts those sequences to prefixes of length $i + 1$, before the test function is applied to Y. In addition, JSXM removes duplicates and sequences that are subsequences of longer sequences. Thus, the set Y has been reduced substantially to 31 sequences as follows:

Y = {<close>, <getBalance>, <deposit>, <withdraw>, <withdrawAll>, <open, open>, <open, withdraw>, <open, withdrawAll>, <open, close, open>, <open, close, close>, <open, close, deposit>, <open, deposit, open>, <open, deposit, close>, <open, getBalance, open>, <open, getBalance, close>, <open, getBalance, deposit>, <open, close, withdraw>, <open, close, withdrawAll>, <open, close, getBalance>, …}

up to sequences of length 4.

The next step is to convert those sequences of processing functions to sequences of inputs. Derivation of input symbols for functions accepting simple inputs is straightforward. For functions that accept complex inputs, the JSXM test case generation tool relies on definitions of test input generators, as described in section 7. The domains of such processing functions include a (potentially infinite) set of input symbols. The input generators are consulted to assign values to the input arguments, since the input names are fixed.

As mentioned earlier, the input is selected for the processing function in isolation from the rest of the sequence, based only on the current memory value. In the Account SXM example, there are three functions with complex inputs: deposit, withdraw, and withdrawAll. The input generators are defined as follows in the JSXM specification:

```
<testinputgeneration>
  <inputgenerator function="deposit">
```

```
      deposit.set_amount(5);
  </inputgenerator>
  <inputgenerator function="withdraw">
      if (balance != 1)
            withdraw.set_amount(1);
      else
            withdraw.set_amount(2);
  </inputgenerator>
  <inputgenerator function="withdrawAll">
      withdraw.set_amount(balance);
  </inputgenerator>
</testinputgeneration>
```

Function deposit accepts any positive value for the amount argument, thus the input generator simply assigns it a fixed value of 5. Functions withdraw and withdrawAll also accept positive amount arguments, but they also have preconditions on memory. Those preconditions are satisfied in the test input generators. Thus, to exercise function withdrawAll, the input generator specifies the value selected for the amount argument to be equal to the balance variable in the SXM memory. The specified selected value for function withdraw is a fixed value of 1, unless it is equal to the balance in the memory, in which case withdrawAll would have been triggered. If this is the case, a different value of 2 is selected.

Some attempted function sequences contain non-input-complete processing functions that are not admitted. For example, function withdraw in sequence <withdraw> is not admitted, since the current memory value when it is attempted has a balance of zero. Given that the precondition of withdraw is:

```
withdraw.get_amount() > 0 && balance > withdraw.get_amount()
```

the withdraw amount should be greater than zero, thus there exists no input symbol that can exercise it. JSXM terminates input sequences from the first encountered function that is not admitted, so that the sequence is reduced to applicable.

Thus the set Y derived above is translated to the final test set X of sequences of input symbols, which contains 25 sequences and is as follows:

> X = {<close>, <getBalance>, <(deposit, 5)>, <open, open>, <open, close, open>, <open, close, close>, <open, close, (deposit, 5)>, <open, (deposit, 5), open>, …}

7.3.2 The SupplyOrder example

The state cover and characterisation sets for the associated FA of the SupplyOrder SXM (Figure 23) are as follows:

> S = {ϵ, <addOrderLine>, <cancelOrder>, <addOrderLine, getQuotation>, <addOrderLine, getQuotation, confirmOrder>}
>
> W = {<getQuotationEmpty>, <getQuotation>, <confirmOrder>, <getConfirmedQuotation>}

The set Φ consists of the nine processing functions labelling the transitions in the associated FA:

Φ = {addOrderLine, removeOrderLine, removeLastOrderLine, cancelOrder, getQuotationEmpty, getQuotation, getConfirmedQuotation, rejectOrder, confirmOrder}

For k=0, the set of processing function sequences Y is as follows:

$$Y = S\{\Phi \cup \{\epsilon\}\}W = S\Phi W \cup SW$$

= {ε, <addOrderLine>, <cancelOrder>, <addOrderLine, getQuotation>, <addOrderLine, getQuotation, confirmOrder>}Φ{<getQuotationEmpty>, <getQuotation>, <confirmOrder>, <getConfirmedQuotation>} ∪ {ε, <addOrderLine>, <cancelOrder>, <addOrderLine, getQuotation>, <addOrderLine, getQuotation, confirmOrder>}{<getQuotationEmpty>, <getQuotation>, <confirmOrder>, <getConfirmedQuotation>}

After cutting the resulting sequences to prefixes of length $i + 1$, and removing duplicates and sequences that are subsequences of longer sequences, the set Y is substantially reduced to 62 sequences as follows:

Y = {<getQuotation>, <confirmOrder>, <getConfirmedQuotation>, <removeOrderLine>, <removeLastOrderLine>, <rejectOrder>, <addOrderLine, getQuotationEmpty>, <addOrderLine, confirmOrder>, <addOrderLine, getConfirmedQuotation>, <cancelOrder, getQuotationEmpty>, <cancelOrder, getQuotation>, <cancelOrder, confirmOrder>, <cancelOrder, getConfirmedQuotation>, <getQuotationEmpty, getQuotationEmpty>, <getQuotationEmpty, getQuotation>, <getQuotationEmpty, confirmOrder>, <getQuotationEmpty, getConfirmedQuotation>, <addOrderLine, rejectOrder>, <cancelOrder, addOrderLine>, <cancelOrder, removeOrderLine>, <cancelOrder, removeLastOrderLine>, <cancelOrder, rejectOrder>, <cancelOrder, cancelOrder>, <addOrderLine, getQuotation, getQuotationEmpty>, <addOrderLine, getQuotation, getQuotation>, <addOrderLine, getQuotation, getConfirmedQuotation>, …}

up to sequences of length 5.

In the SupplyOrder SXM example the test input generators are defined for functions receiving complex inputs, that is, “addOrderLine”, “removeOrderLine”, and “removeLastOrderLine”, as follows:

```
<testinputgeneration>
  <inputgenerator function="addOrderLine">
      String id = UUID.randomUUID().toString().substring(0, 3);
      Integer quantity = (new Random()).nextInt(10);
      addItem.set_itemId(id);
      addItem.set_quantity(quantity);
  </inputgenerator>

  <inputgenerator function="removeOrderLine">
      if (order.size() > 1)
            removeItem.set_itemId((String)
order.keySet().toArray()[
                        (new Random()).nextInt(order.size())]);
  </inputgenerator>

  <inputgenerator function="removeLastOrderLine">
```

```
      if (order.size() == 1)
            removeItem.set_itemId((String)
order.keySet().toArray()[
                         (new Random()).nextInt(order.size())]);
  </inputgenerator>
</testinputgeneration>
```

Since function addOrderLine does not check an items inventory, any item identifier that is a string of characters is accepted. Therefore, the input generator for this function generates a random string for the itemId argument, using Java utilities for random string generation. Similarly, this function accepts any value for the item quantity argument. Thus, the input generator generates a random value between 1 and 9 for the quantity argument.

The input generators for functions removeOrderLine and removeLastOrderLine specify that the value for the itemId to be removed is the identifier of a random item in the current order. The difference is that for function removeLastOrderLine such an argument value is selected if there is only one item remaining, while for the removeOrderLine function there is more than one item remaining in the order.

Functions removeOrderLine and removeLastOrderLine are not input-complete, thus some sequences of processing functions have to be reduced to applicable. Examples of non-applicable sequences are <removeOrderLine> and <addOrderLine, removeOrderLine, getQuotationEmpty>. The initial memory contains an empty order, while function removeOrderLine requires at least one item (order line) in the order, hence there exists no input that can exercise that function. In the second sequence, the prefix <addOrderLine> updates the order in the memory to consist of one item. However, the next function in the sequence, removeOrderLine requires an order of at least two items in the memory. Thus it is not admitted and removed from the sequence.

The final test set X consists of 52 sequences of input symbols as follows:

> X = {<getQuotation>, <confirmOrder>, <getConfirmedQuotation>, <rejectOrder>, <(addItem, a6a, 9), getQuotation>, <(addItem, fc9, 1), confirmOrder>, … , <(addItem, 07a, 9), (removeItem, 07a), confirmOrder>, …}

We remark that the fifth sequence of input symbols, <(addItem, a6a, 9), getQuotation>, was selected to attempt function sequence <addOrderLine, getQuotationEmpty>. Since the second function in the sequence (getQuotationEmpty) is not admitted at the current state (filling_order), the selected input actually triggers a different function, getQuotation. Thus, the expected output is the pair (getQuotation, successful) instead of (getQuotation, emptyOrder).

7.4 Equivalence versus conformance testing

There are two forms of testing with stream X-machines: equivalence testing and conformance testing. *Equivalence testing* aims to establish that the implementation computes the same function (or relation) as the specification, i.e. $f_Z = f_{Z'}$, where Z' is

the implementation machine and Z is the specification machine [57]. The deterministic SXM integration testing method described above is able to determine the *equivalence* of the system under test to its SXM specification.

As explained in section 5.8, SXM specifications can be made nondeterministic in order to raise their level of abstraction and to allow modelling complex or nondeterministic Web services. In certain situations it is considered as sufficient to demonstrate that the behaviour contained in an implementation is a subset of the behaviour contained in its nondeterministic SXM specification [69]. This means that a NSXM allows a range of possible behaviours for the WSUT to implement. Therefore, *conformance testing* aims to establish that the relation computed by the implementation machine Z' is a subset of the relation computed by the specification machine Z, i.e. $f_{Z'} \subseteq f_Z$ [57]. As can be seen, conformance is a weaker notion of correctness than equivalence.

Test generation algorithms for testing an implementation against a nondeterministic SXM for conformance [69] and also for equivalence [84] have been proposed. Both assume that the nondeterministic SXM is completely defined [57]. Like the SXM integration testing method described earlier, these algorithms are based on the application of the W-method to the associated FA, in order to generate sequences of relations. However, test data generation with a test function and the test execution process are complicated by two main cases of nondeterminism:

(a) transitions are labeled by *relations* that can produce different memory values and outputs;

(b) different relations can be triggered due to possibly overlapping domains (domain nondeterminism).

To address the first case of nondeterminism, both equivalence and conformance testing use an adaptive *test process*, where a tester observes the decision made by the implementation in every step of a sequence and adapts testing accordingly. The need to observe decisions taken adds further assumptions on the implementation under test, such as being able to determine memory updates performed by relations by observing the outputs [69]. The test process attempts to exercise a sequence of relations starting from the first relation in the sequence and proceeding toward the last one. In each step the input is chosen based on the new memory value and output produced by the previous step, so that it can possibly trigger the next relation in the sequence.

Equivalence testing mainly differs from conformance testing on the way domain nondeterminism is handled. Testing for equivalence aims to prove that the implementation computes the same relation as the specification, with the assumption that the individual relations have been correctly implemented. Therefore, the tester attempts to exercise all possible paths that may be taken in response to sequences of inputs generated by the test process for an attempted sequence of relations. As a result, the same sequence of relations is attempted up to

a maximum number of times (using possibly different sequences of inputs) and if outputs from an implementation show that it did not execute the expected sequence of relations on any of the attempts, it is assumed that such a sequence of relations is not implemented [84].

Consider the SXM specification of the SupplyOrder Web service example with inventory lookup, depicted in Figure 21 of section 5.8. Recall that the specification is nondeterministic, since it does not model the items inventory in its memory construct, which affect the service behaviour. As a result, the success "itemUnavailable" and the nonsuccess "addOrderLine" processing relations do not check for item availability in the inventory and are triggered nondeterministically (overlapping domains).

In the SXM specification, input symbol (addItem, *itemId*, *qty*) triggers nondeterministically two relations: addOrderLine and itemUnavailable, producing outputs (itemAdded, successful) and (itemAdded, unavailable), respectively. Hence, testing the SupplyOrder Web service for equivalence aims to demonstrate that the implementation follows paths that include both relations addOrderLine and itemUnavailable. In other words, input sequences including input symbol (addItem, itemI*d*, *qty*) should produce output sequences that include both expected outputs in the set {(itemAdded, successful), (itemAdded, unavailable)}. On the other hand, conformance testing tries to show that the language of the implementation machine is included in the language of the specification machine (language inclusion). Therefore, testing the SupplyOrder Web service for conformance aims to demonstrate that the implementation follows paths that include any or both of relations addOrderLine and itemUnavailable, *without having to attempt* them several times. I.e., input sequences including input symbol (addItem, itemI*d*, *qty*) should produce output sequences that include any of the expected outputs in the set {(itemAdded, successful), (itemAdded, unavailable)}, but not other unexpected outputs. As can be deduced, in deterministic SXM specifications only one output is allowed for every input, thus conformance testing coincides with equivalence testing.

In the Web services domain there are limitations on the ability to drive all possible paths by attempting the same sequence of relations multiple times. In the SupplyOrder Web service with inventory lookup described above, it is unrealistic to expect the conditions in the inventory to change so that the same sequence of inputs can trigger both relations on the WSUT: addOrderLine when the item is available in the inventory and itemUnavailable if the item is not available. Even if different input sequences are attempted for the same sequence of relations, it would be difficult for the test function to generate identifiers for items that are available in the inventory, since no inventory items are modelled. The other difficulty in attempting to drive all possible paths in a Web service implementation is that conversations in Web services are often long-running and may involve manual or other factors, which are both time-costly and difficult to reproduce a large number of times.

Therefore, in practice, the SupplyOrder Web service can only be tested for conformance to its nondeterministic SXM specification, which is not satisfactory, since not being able to successfully add an item to the order will exercise only paths that traverse two states: empty_order and cancelled. This problem can be handled by techniques described in sections 5.5.3 and 5.5.4 from the previous chapter, where sample item identifiers are placed in the memory and the SXM is made deterministic to apply equivalence testing.

Because of the above reasons, it is more preferable to have a deterministic SXM specification of a WSUT than a nondeterministic one. The specification and testing approach in this thesis mainly focuses on DSXMs, which are also supported by the JSXM toolset for animation and test case generation. Therefore, this thesis pursues the more challenging objective of raising the level of abstraction of SXM specifications, while keeping them deterministic.

The applicable testing methods for different combinations of SXM specifications and implementations concerning their determinism are summarized in Table 3. Notice that it is not possible to specify a nondeterministic implementation with a correct deterministic specification, since the latter cannot be more concrete than the former. Therefore testing does not make sense in such a scenario.

Table 3 - Combinations of SXM specification and implementation according to their determinism

Specification	Implementation	Testing for
deterministic	deterministic	equivalence [62], [61]
deterministic	nondeterministic	(*undefined*)
nondeterministic	deterministic	conformance [69]
		equivalence [84]
nondeterministic	nondeterministic	conformance [69]
		equivalence [84]

7.5 Error outputs and negative testing

7.5.1 Outputs, error responses and faults

Unsuccessful invocations of Web service operations cause them to return error SOAP response messages or fault messages. Such errors are returned when invalid inputs are provided or when sequences that violate the conversation protocol are invoked. As can be recalled from section 2.2.2, every operation description in WSDL specifies an output message, and optionally one or more fault messages. This model is similar to OOP where every method defines a return value and throws one or more optional exceptions. While a Web service implementation can return faults as defined in WSDL, it is also allowed to produce runtime (or SOAP) faults that are not anticipated in WSDL. Therefore, a WSDL Web service returns one of the following types of outputs:

- Normal SOAP response message
- Error SOAP response message
- Fault message
    - WSDL-defined fault
    - Runtime SOAP fault

It is important not to confuse Web service *fault messages* with *implementation faults*. Web service faults are error outputs that can be part of expected Web service behaviour, while implementation faults are deviations from the expected behaviour. For example, a fault can be returned by the Account Web if the client tries to perform transactions on an inactivated bank account, but the implementation is correct as its behaviour is as expected. On the contrary, if the client is allowed to perform transactions on an inactivated account, without returning faults, then the Web service implementation contains faults.

Moreover, this heterogeneity in the semantics of outputs returned by a WSDL Web service is complicated by the fact that services operate over a network. The introduction of a network into any computing system raises the complexity enormously. As a result, a huge variety of faults is introduced, caused by network and Web service platform failures. It is not feasible to capture such infrastructure faults in SXM specifications. Therefore, only Web service errors at the business logic level are specified and tested in SXMs, excluding any infrastructure errors. If infrastructure errors do occur during testing, they will not match the expected outputs, and will be reported as implementation faults. As a result, an assumption made during Web service testing is that the network, Web service platform, application server, databases, and so on, are correctly set up and work normally.

The described types of Web service error outputs are all modelled as abstract output symbols by SXMs. This means that the formalism represents all types of outputs equally and does not differentiate between normal and error ones[16].

7.5.2 Negative testing

Generally, exercising functionality that produces error outputs constitutes *negative testing*. Negative testing ensures that the WSUT operations handle abnormal invocations gracefully with expected errors, instead of terminating successfully or crashing. In support of negative testing one should be able to compare error Web service responses with SXM output symbols. A human individual can perform the comparison directly; however, if the process is to be automated, one representation should first be mapped into the other before comparison. A more technical treatment of mapping different types of error Web service outputs to abstract output symbols is provided in chapter 0.

[16] An exception is the case of default errors, which are a special type of outputs produced at runtime by the JSXM toolset for unhandled inputs.

The SXM integration testing method does not separate negative test cases from the rest of the test set. Thus, it is appropriate to investigate how well the derived test set is able to cover paths that produce errors. A processing function in the specification that defines the success scenario of a service operation, may fail to be triggered for two possible reasons:

- it is not accepted at a certain control state, or
- the guard condition for triggering the function evaluates to false (input argument(s) and memory value are not in the domain of the function)

As an example, Figure 24 illustrates the internal Java implementation of operation "deposit" in the Account Web service example. To exercise the success scenario of this operation (specified by processing function "deposit" in the Account SXM), both predicates "isOpen" and "amount > 0" must evaluate to true. The first predicate is defined in the control states (state-based conditional) of the SXM, while the second predicate involving the input is defined in the processing function domain (domain-based conditional).

```
public String deposit(int amount)
{
  if (isOpen)
  {
    if (amount > 0)
    {
      balance = balance + amount;
      return "depositOut_" + amount;
    }
    else
      return "deposit_Error";
  }
  else
    return "deposit_Error";
}
```

Figure 24 - Java implementation of Web service operation "deposit", illustrating the operation predicates. The shaded area is the implementation of processing function "deposit"

Proper negative testing should be able to make both conditionals fail in the implementation at different times, and check whether the expected error outputs are produced. That is, test cases should cover both cases: when the function is not accepted in current state, and when the function's guard condition is not satisfied.

As explained in section 7.1.4, the W-method attempts to exercise all processing functions in all states, regardless of whether they are accepted or not. Thus SXMT checks that a function that is not accepted in a state is not exercised in the implementation; instead, either a different (failure) processing function is triggered, or, if the specification is partially-defined, a default error is produced.

On the other hand, SXMT does not cover all failure scenarios which result when processing function guards on input and memory fail. This is the case because the test function *t* always selects input symbols that do satisfy the guard conditions and

trigger the function. In partially-specified SXMs, if no alternative processing functions are defined, the negative scenario is not exercised. However, in completely defined SXMs failure scenarios are also modelled as transitions, and are therefore exercised by the SXMT.

In the above example, SXMT tests what happens when trying to deposit when the account is opened (success expected), as well as when the account is not opened (error output expected). This means negative testing is performed at the integration level. On the other hand, since the specification is partially-defined, SXMT does not test what happens when the input is negative or zero.

In summary, the SXM integration testing method has varying strengths on negative testing. The state transition diagram is fully tested positively and negatively, by attempting all possible transitions, whether they are accepted or not in each state. However, for further negative testing of processing functions labelling transitions, the SXM needs to be completely defined, in order to exercise failure scenarios as well.

7.6 Testing considerations in the Web services domain

Owing to Web service characteristics, the application of the SXM testing method to Web services requires further investigation. Some additional assumptions need to be made on Web services under test, as described in the following subsections.

7.6.1 Message Exchange Patterns of Web services under test

Recall from chapter 0 that Web service operations do not always accept inputs (request messages) and return outputs (response messages), as they are usually portrayed. Besides the common request-response message exchange pattern (MEP), three other possible MEPs characterise operations: solicit-response, one-way, and notification.

- Web service implementations containing *one-way* operations do not produce outputs when those operations are invoked. Therefore, those Web services cannot be output-distinguishable and are not testable by the SXMT method.
- Web service implementations containing *notification* and *solicit-response* operations are not driven by inputs, but initiate those operations internally. Consequently, it is not possible to drive such operations through appropriate inputs, and the Web services containing this kind of operations cannot be controllable. Therefore, these categories of Web services are not testable by the SXMT method.

This thesis considers only Web services, in which all operations follow the *request-response* message exchange pattern. Only this kind of Web services can be controllable and observable – they can be controllable by choosing proper request messages and observable by examining response messages during test execution. This requirement is not considered as too restrictive, since, from our experience,

Web services with operations characterized by MEPs other than request-response are fairly rare on the Internet.

7.6.2 The need for a sandbox (test) interface

The application of active testing (as opposed to passive testing, or monitoring) to the verification of third-party Web services, separates service testing activities from service usage (consumption) activities. Test inputs are generated and actively executed on the service under test without depending on other requestors invoking the service. This gives the opportunity to avoid running the tests on the Web service deployment that is made available to service requestors. Instead, the service provider can make available another deployment of the same Web service implementation, called a sandbox (or test) interface, for the purpose of testing.

The introduction of a sandbox interface available for testing is a powerful technique that provides solutions to a number of problems. Among such solutions are included:

- Augmenting the service to satisfy the design-for-test conditions;
- Implementing the modelled portion of a large data repository (section 5.5.3).
- Avoiding undesirable side effects (described later);
- Implementing the reliable reset (described later);

However certain issues arise when a sandbox interface is tested instead of the production Web service.

- Some test conditions are not satisfied if several tester participants (clients, certification authorities, etc) invoke the sandbox interface simultaneously during testing.
- It can be unclear whether the sandbox interface being tested has the same implementation as the real (production) interface. Nevertheless, the sandbox and real Web services are expected to be different deployments (instances) of the same Web service implementation. It follows that if the sandbox deployment is verified as correct, then the production deployment is also correct.

7.6.3 Services with undesirable side effects

Very often, invoking operations on commercial Web services, apart from returning outputs, will result in undesirable side effects. Such side effects could involve state modifications, invocation of other applications or services, financial transactions, or physical tasks. For example, testing the banking Web service with a real bank account will initiate undesirable financial transactions, which are not practical or feasible for the purpose of testing. Similarly, in the case of a shopping cart Web service, testing the checkout operation involves charging the credit card, which is not desirable.

For that reason, commercial services such as PayPal, UPS, and FedEx make sandbox interfaces available, which simulate the operation of real production Web services, without the side effects. For example, the United Parcel Service (UPS) [58] maintains a special "staging" version of its shipping Web service, which supports testing of applications by simulating transactions with UPS. The staging service responds to Web service requests just like the UPS production service; however, it does not initiate actual business activities. For example, if one sends a shipping request to the UPS production service, a UPS driver will be dispatched to the specified location, expecting to pick up a package (and expecting payment for the service). Sending the shipping request to the staging service will avoid this problem.

Therefore, it is assumed in this thesis that Web service sandbox interfaces do not incur any harmful or undesirable side effects when their operations are invoked.

7.6.4 Resetting the WSUT to the initial state

Every input sequence from the test set is exercised under the assumption that the state of the WSUT has the initial values. However, the execution of previous test sequences may have transitioned the potentially faulty implementation into a state and memory that is unknown. Thus, it is necessary to bring the WSUT back to the initial, hence known, state.

There are two known approaches that address the reset problem. The approach adopted by the SXMT requires the system under test to feature a *reliable reset* function, which is implicitly assumed to be appended at the end of every test sequence. The reset is considered as reliable if it is known to have been implemented correctly. The role of the reset feature is to set the initial state and memory back to their initial values. This technique is not always practical since it requires intervention on the implementation. Concerning Web services, a reset would also require resetting shared state that is captured in the specification. The other technique adopted by some testing methods [50], [85] is to append test sequences with postambles, which are sequences that return the SUT back to the initial state. Although this technique is less restrictive on the SUT, it also produces longer test sequences, and requires that from any state and memory value, the initial state is reachable.

It is important to note that only the *modelled* state and memory need to be reset to the initial values. As described in the previous chapter, SXM specifications model only the per-client state maintained by private-state Web services. As a result, it is sufficient to reset only that portion of the state in the WSUT, instead of the whole pan-client state. For instance, while testing a Web service managing several bank accounts, it is not only unnecessary, but also impractical or impossible to set the bank accounts of other clients back to their initial status and balance, in the end of each test sequence. The reset operation also has to be associated with identification information for the specific client. In a similar fashion, if the abstract specification

captures the per-object state (when every client potentially accesses several objects), it is sufficient to reset only the state variables for the identified object. E.g. if in the SupplyOrder Web service each client places several orders, only the identified order instance is set back to its empty state in the WSUT. Evidently, for any other cases where the abstract specification models only a portion of the service state, then it is adequate for the WSUT to reset those state variables that are captured as control states and memory in the specification. The rest of the service state that is abstracted away is left intact.

Therefore, three different cases are identified based on Web service category and state duration (see section 4.3.6). In each case a distinct approach is proposed for the provider to implement the reliable reset in the WSUT, and for the tester to trigger the reset.

- In *volatile, private-state services* all state lives within one SOAP or HTTP session. No intervention is required on the WSUT, while the tester triggers the reliable reset by terminating the current session. As a result, the application of the next input sequence initiates a new session. All previous context information for the particular requestor is erased and the next conversation starts from the initial state values.
- In *persistent, private-state services* the per-client state spans multiple sessions. Therefore, the WSUT must implement a “reset” operation that accepts the same identifiers as the rest of the service operations. The reset operation resets only that portion of the state variables, which pertain to the client (per-client specification) or object (per-object specification) being identified and tested. On the other hand, the tester triggers the reliable reset by invoking the reset operation in the end of every test sequence. In practice, this involves appending input symbol “reset” at the end of every sequence of input symbols. Identifiers are then supplied during input concretisation as for the rest of the operations.
- In *shared-state services* part of the state is accessed and/or modified by other requestors simultaneously. If part, or all, of the shared state is modelled in the specification, the implementation of the reliable reset is more difficult. As above, the WSUT must include a “reset” operation, which resets not only the per-client state, but also the modelled shared state. This approach is not feasible in production deployments of Web services, since it is accessed concurrently by other requestors and would corrupt the shared state. Thus, a sandbox deployment is assumed, where the tester is the *only* requestor. Similarly, the tester triggers the reliable reset by invoking the reset operation in the end of very test sequence.

7.7 Finding faults

In the examples section the test sets were derived from non-controllable SXM specifications. Since controllability is one of the design-for-test conditions required

by SXMT to find all faults in the implementation, it is possible that the derived test sets do not detect all faults. Also, the SXM integration testing method assumes that the individual components have already been verified as correct. Thus, this section illustrates the faults that can be detected, as well as those that are not detected by the test sets generated for the Account and SupplyOrder Web services.

7.7.1 Evaluation of test cases through manual injection of control flow faults

The SXM integration testing method is able to find all control flow faults in the implementation, such as *missing* or *extra transition*, *erroneous transition labels*, *erroneous next state*, and *missing* or *extra state*. Quite often, such control flow faults in the state machine correspond to meaningful business logic faults and violations of conversation protocols. For example, an extra “cancel” transition from state “pending” to state “cancelled” in the SupplyOrder Web service, violates the business requirement that an order pending confirmation by the client cannot be cancelled, but first rejected, and then cancelled. As the following experiments show, this type of fault is revealed by the toolset for $k = 0$.

In this section the SupplyOrder Web service is tested using the test set generated earlier, in order to reveal various kinds of control-flow faults. This is done for two purposes: to evaluate the ability of SXMT to reveal these kinds of faults for different values of k, and to illustrate some meaningful types of business-logic faults that can be revealed.

Given the assumption that the implementation is a machine of the same type Φ as the specification (i.e. all processing functions have been correctly implemented), the SXMT does not target faults where processing functions are erroneously implemented. On the other hand, it can find all faults where the wrong processing function from the set Φ labels a transition.

Control flow faults are expected to be common in implementations. Normally, in programming code transition pre-states are checked inside the conditions of “if” and other selection control structures, while next states are determined in the processing blocks. It is common to mistake predicate conditions (such as wrong logical and mathematical operators) and variable assignments, which may result in missing/extra states or erroneous next states.

Erroneous next state

An example of a SupplyOrder Web service with erroneous next state fault is represented by the SXM in Figure 25. Due to this fault, rejection of a pending order actually causes confirmation of the order. That is, transition labelled by “rejectOrder” leads to state “confirmed”.

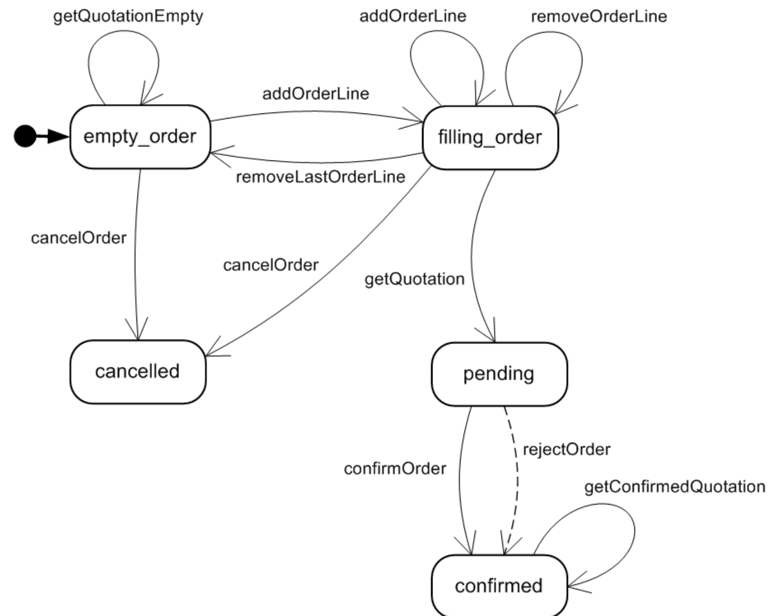


Figure 25 - State-transition diagram of a SupplyOrder implementation with erroneous next state fault

The test set generated for $k = 0$ is able to reveal this fault, as shown in the JUnit output in Figure 26. This is an expected result for for $k = 0$, since the implementation contains no extra states.

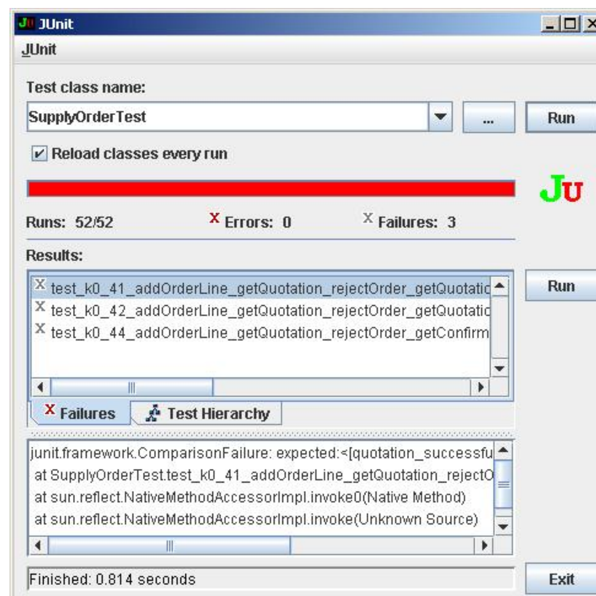


Figure 26 - JUnit execution results on the implementation with erroneous next sate fault

Erroneous transition label

An example of an erroneous transition label fault is depicted by the state-transition diagram in Figure 27. In the faulty implementation, rejecting an order in the normal "filling_order" state actually results in transitioning the order to the "cancelled"

state, instead of a default error. Being in a "cancelled" state, there is no further possibility to continue processing the order. In other words, the cancelOrder transition has been mislabelled with function rejectOrder. Running the test set for k = 0 reveals this kind of fault in the implementation.

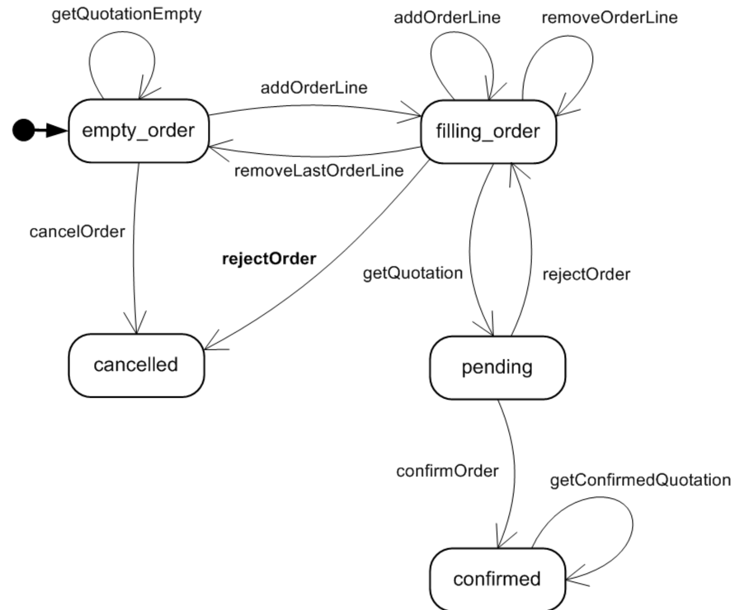


Figure 27 - SupplyOrder implementation with erroneous transition label

Missing transition

In the faulty implementation only an empty order can be cancelled. If the order has items added to it ("filling_order" state) it cannot be cancelled, but first emptied and then cancelled. This violates the business requirement of also being able to cancel non-empty supply orders. The test set for k = 0 reveals this kind of fault in the implementation.

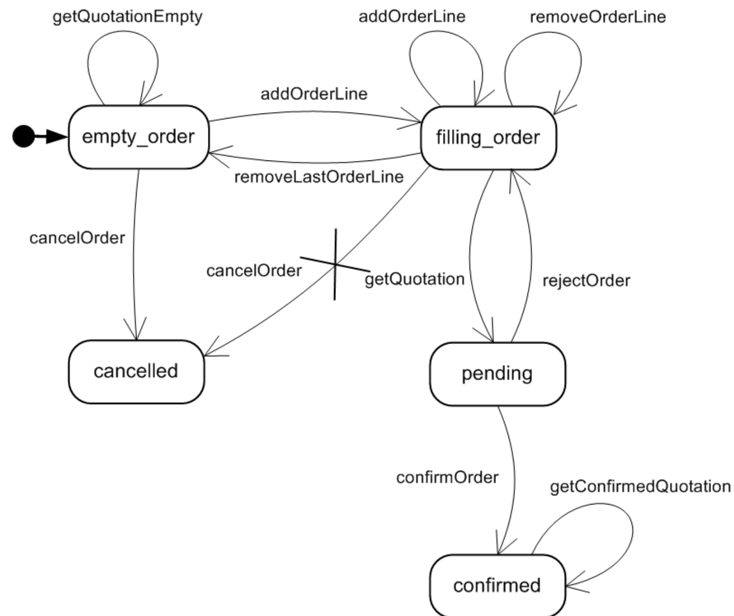


Figure 28 - SupplyOrder implementation with missing transition

Extra transition

In the faulty implementation, the order can be cancelled even after a quotation has been obtained, i.e. while the service is pending for a confirmation or rejection. This means an extra "cancelOrder" transition exists from state "pending" to state "cancelled". Again, as expected, the test set for $k = 0$ reveals this kind of fault in the implementation.

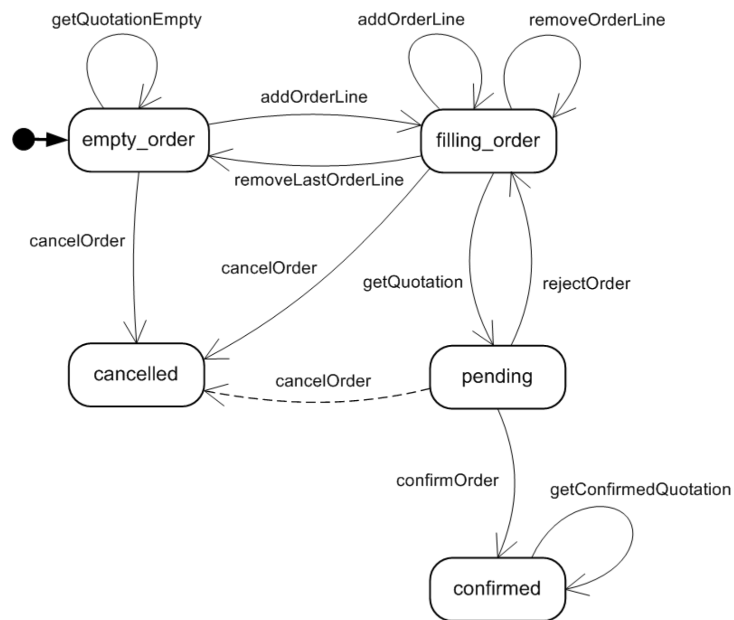


Figure 29 - SupplyOrder implementation with an extra transition fault

Missing state

Missing states may occur, for example, when two modelled states are actually non-distinguishable in the implementation under test. That is, they accept the same set of processing function sequences, essentially merging into one. Usually, mising states are also associated with other control flow faults, such as missing transitions.

As an example, suppose that a request for quotation from the SupplyOrder service directly results in the confirmation of the order. This means that the intermediary “pending” state, where the service waits for a confirmation or a rejection, is omitted. This scenario also involves missing transitions that originate from the missing state. Thus, the state “pending” and transitions “confirmOrder” and “rejectOrder” are missing. The test set generated earlier for $k = 0$ reveals this missing state fault.

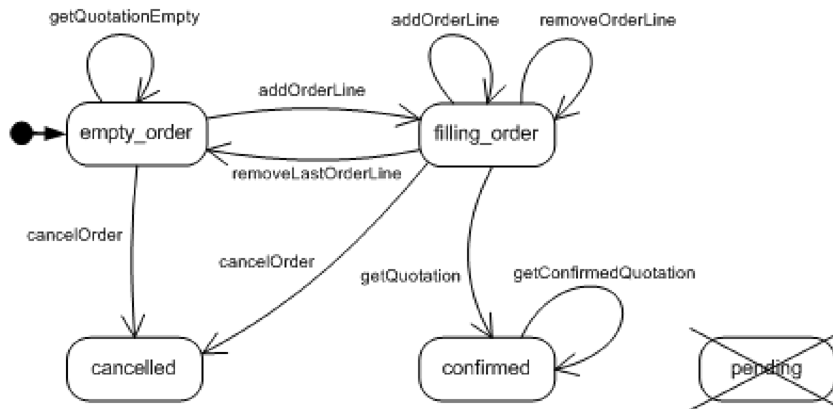


Figure 30 - SupplyOrder implementation with a missing state fault

One extra state

Detection of implementation faults involving extra states is a different case that requires values of k that are greater than zero, depending on the number of extra states in the implementation. Figure 31 shows the state diagrams for two different cases of faulty SupplyOrder implementations with one extra state.

The first example is a SupplyOrder service, in which the removal of all items from the order transitions the service to a state where the order can only be cancelled. In difference from the initial state “empty_order”, from this hidden state the client is not allowed to add any more new items to the order. Running the test set for $k = 0$ on the implementation does not reveal the fault, but as expected, for $k = 1$ the fault is revealed. Notably, the fault might also have been revealed for $k = 0$ if the sequence <addOrderLine> distinguishing between state “empty_order” and the hidden state “emptied_order” was included in the characterisation set W.

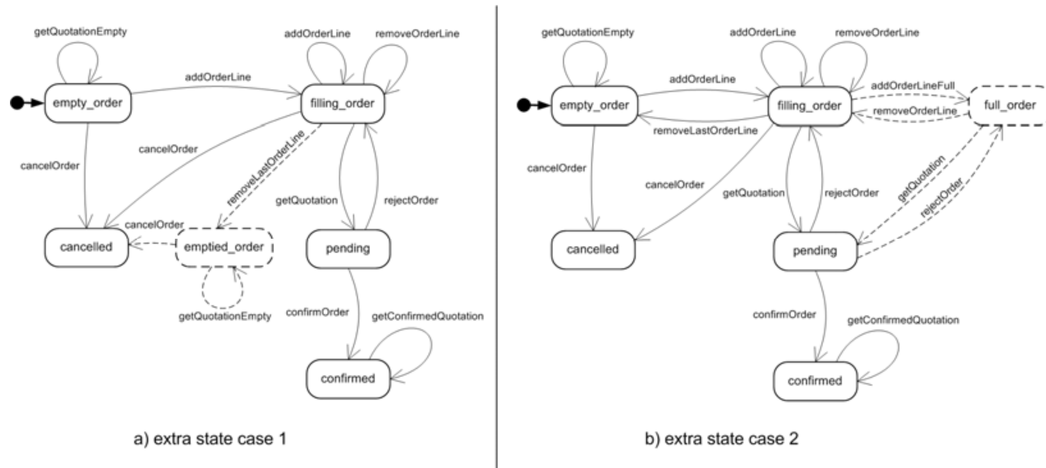


Figure 31 - Two examples of faulty Web service implementations with one extra state: a) revealed by test sets for k = 1 and b) not revealed by test sets for k = 1

The other example of a hidden extra state fault is a SupplyOrder Web service implementation, which does not allow adding more than a maximum number of items (order lines) to the current order. That is, the service transitions to a different state, where it has become full and cannot accept additional requests for adding items. As the diagram illustrates, this extra state is called "full".

Unpredictably, the execution of the test set for $k = 1$ does not reveal this extra state. This situation arises because the "full" extra state is not reached by a sequence of a single processing function, but by repetitive triggers of function "addOrderLine" until the maximum capacity is attained. Large capacities also require large values of k that result in huge test sets. It seems that in this situation, k represents the maximum path length to the extra hidden state, instead of the difference in the number of states (Figure 32). However, closer examination shows that this is a fault in the implementation of processing function "addOrderLine", whose preconditions, in contrast to the specification, also check the number of items in the order. This violates the requirement that the specification and implementation machines are of the same type Φ. However, in practice, it can be difficult for the tester to separate control flow from individual components in the implementation, in order to reach the conclusion that the specification and implementation have identical types.

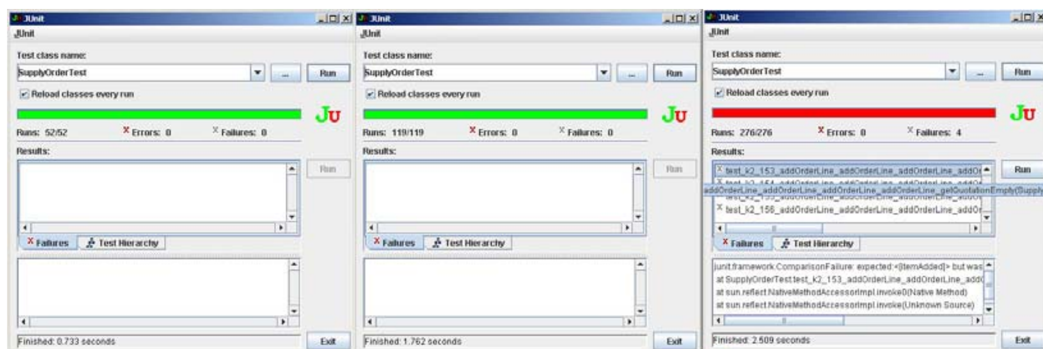


Figure 32 - Running the JUnit test sets for values of *k* between 0 and 2 on an order of maximum capacity of three items. The fault is revealed for k = 2 that derives sequences of four adjacent "addOrderLine" functions.

7.7.2 Test cases evaluation through manual injection of individual processing function faults

The stream X-machine integration testing method is able to reveal all control-flow faults in the system under test, but not necessarily faults in the implementation of individual processing functions. As regards Web services, the SXMT is able to verify the conformance of the Web service under test to the protocol advertised in the SXM model. But this method is not able to verify the correctness of the functions labeling the state transitions. Possibly, the generated test set may detect some of the faults in the implementation of individual processing functions, while trying to reveal control flow faults. However, there are no guarantees regarding coverage and effectiveness.

This section presents a number of experiments with the Account and SupplyOrder examples, in order to demonstrate cases where processing function faults are detected and cases where they are not. In addition, these examples give an idea about how such faults look like and how critical they are. They can be faults in the guard conditions, as well as faults in the memory updates and output computations.

Consider the Account example. One possible fault in the deposit function is an incorrect balance increase, either by a wrong amount or not at all.

In the case of the simple SupplyOrder example, possible faults in the memory update of the addItem processing function include, not updating the order at all, adding the wrong item identifier, or the wrong quantity. The contents of the list of order lines in the supply order are updated by addItem, but they are only checked by getQuotation to determine whether the order is empty or not, and in this case only the number of order items. The item ID's and quantities in the order lines are not checked by any of the processing functions. As a result, while it might be possible that addItem and removeItem faults concerning the number of order lines are reported, no faults concerning item ID's and quantities are detected, as they do not affect the outputs of any processing function. On the other hand, if processing function getQuotation returns not only a simple output message about operation success, but also the current contents of the order, where item identifiers and quantities are listed, it would have been possible to detect those types of faults.

Furthermore, as previously mentioned, the SXM testing method does not include data coverage criteria. Thus, the different cases for inputs to the same processing function are not tested. E.g., the method does not attempt boundary inputs for negative testing, classes of inputs producing different results, etc.

7.7.3 Test cases evaluation through automated mutation testing

In the previous experiments various types of control-flow faults were seeded in the implementation under test, and the effectiveness of the generated JSXM test cases was evaluated regarding their ability to reveal those faults. As the results of test execution demonstrated, all control-flow faults were detected for values of $k = 0$, with the exception of hidden extra states in the faulty implementation where larger values of k were required..

A complementary approach to evaluating the effectiveness of the test set in fault detection is *mutation testing*. Mutation testing involves taking the code under test, seeding faults (called mutants), and executing the test set on the mutated code. That is, while test cases aim to expose faults in the implementation, mutation tests aim to expose weaknesses in the test cases themselves. A number of mutation testing tools exist (Table 4), which introduce various types of mutations automatically, run the test cases and report their effectiveness in detecting ("killing") the mutants.

Table 4 - Comparison of mutation testing tools

	Jumble	**MuJava**	**Jester**	**Judy**
JUnit support	Yes	No	Yes	Yes
Operation on	bytecode	bytecode	source code	source code

In this work a number of simple mutation tests have been performed with two of the above tools: *Jumble* [86] and *Jester* [87]. Both tools are able to seed faults into Java code and evaluate JUnit tests on that code, which were derived by transformation of JSXM abstract tests. For convenience, the test cases were run on the Java classes that implement the actual Web services. The Account and SupplyOrder examples were used as the classes under test, while JUnit test cases for different values of k were evaluated.

In the SupplyOrder example, Jumble tried only 8 *mutation points* (faults) on the class under test, SupplyOrder.java. The test set for $k = 0$ was evaluated with a maximum score of 100%, meaning that all 8 mutations were detected (killed). Obviously, the larger test sets for higher values of k subsume this effectiveness, thus they also received the maximum score of 100%. Nevertheless, the test cases for $k = 0$ cannot be considered as perfect, as the previous experiments with manual control-flow faults demonstrated. The test cases for $k = 0$ often failed to reveal extra-state faults, and test cases for values of k of at least 1 were required to reveal those faults.

In the case of the Account example, the generated JUnit test cases were again evaluated as perfect (100% score). 12 mutation points were tried, and all were detected by the JSXM test cases, as illustrated in the following Jumble output.

Jumble output:

```
Mutating SupplyOrder
Tests: SupplyOrderTest_k0
Mutation points = 8, unit test time limit 2.16s
```

```
........
Score: 100%

Mutating Account
Tests: AccountTest_k0
Mutation points = 12, unit test time limit 2.0s
............
Score: 100%
```

It can be speculated that the kinds of mutation faults injected by Jumble on the class under test are relatively simple and easy to detect by the SXMT test cases. The mutations are not as advanced as introducing new states in the implementation. However, it was expected that Jumble might introduce some faults into individual processing functions, in which case the test cases would probably fail in fault detection, given that SXMT assumes processing functions to be correct. Possibly, the implementation of individual functions might have been modified, but the test cases generated to detect control flow faults were also able to detect faults in individual processing functions as a side-effect. More complex, industrial examples would have to be tried for better evaluation of the strength of JSXM test cases and to derive more accurate mutation scores. In any case, the evaluations on the two examples give some confidence as to the effectiveness of the generated test cases.

The other tool, Jester, also gave the maximum score of 100 to all JUnit tests generated by JSXM for the SupplyOrder class. However, Jester was able to detect problems in the JUnit test cases for the Account class. Out of 20 mutations, 2 of them were not detected (*survived*) by the test sets, apparently regardless of the values of *k* (assigned values from 0 to 2 in the experiment).

Jester output:

SupplyOrder, k=0:

```
0 mutations survived out of 24 changes. Score = 100
took 3 minutes
```

Account, k=0:

```
For File src\Account.java: 2 mutations survived out of 20 changes.
Score = 90
src\Account.java - changed source on line 33 (char index=619) from
if ( to if (true ||
c String withdraw(int x) {
                >>>if (isOpen && balance != 0 && balance >= x) {
src\Account.java - changed source on line 33 (char index=644) from
0 to 1
{
                if (isOpen && balance != >>>0 && balance >= x) {
                        balance -= x;
                        re

2 mutations survived out of 20 changes. Score = 90
took 2 minutes
```

Account, k=1:

```
(same results)
```

Account, k=2:

```
(same results)
```

The first surviving mutation changes the precondition of the success scenario of operation withdraw (isOpen && balance != 0 && balance >= x) to evaluate always to true. Since the test cases do not detect this modification, it means that they never test the operation when the preconditions evaluate to false. The first part of the precondition (isOpen && balance != 0) consists of *state-based* predicates (encoded in the states) and evaluates to false in states "initial", "opened" and "closed". Not being *input-complete*, functions withdraw and withdrawAll cannot be exercised with any input when the balance in the account is equal to zero, which always holds in states "initial", "opened" and "closed". Therefore, SXMT cannot attempt functions withdraw and withdrawAll from those three states. As presented in the examples section, sequences such as <withdraw>, <withdrawAll>, and <open, close, withdraw> were removed from the test set or reduced to applicable.

The other predicate in the preconditions (balance >= x) is a domain-based predicate. As explained earlier, since it represents the guards of processing functions withdraw and withdrawAll, the test function selects inputs such that they make this predicate succeed (i.e. amounts less than or equal to the available balance). Also, the SXM specification is not completely-defined, hence no other processing function handles the case when the predicate evaluates to false (i.e. amounts greater than the available balance).

The second surviving mutation changes the predicate (balance != 0) to (balance != 1). The test cases do not detect this modification since the correct and faulty predicates evaluate to the same truth value for all test cases. The test set would have been able tell the difference if it had tried the case when the correct predicate evaluates to false, i.e. (balance = 0). However, as explained for the first mutant, no possible input can exercise this case, since functions withdraw and withdrawAll are not input-complete with respect to memory.

Therefore, it can be concluded that both surviving mutations are a consequence of the specification not being input-complete and completely-defined. The next subsection briefly discusses the implication of not satisfying the design-for-test conditions on the effectiveness of the test cases.

7.7.4 Effectiveness of test cases when design-for-test conditions are not satisfied

The stream X-machine integration testing method that is able to find all faults requires the specification and implementation machines to satisfy the design-for-test conditions. An algorithm that ensures that these conditions are satisfied is provided in Ipate & Holcombe [88]. This algorithm requires designing extra functionality that will have to be disabled after testing has been completed. However, this process

requires extra effort and may inadvertently introduce new implementation faults. Furthermore, when testing Web services, which are provided across organisational borders by third-parties, the tester has no control on their implementations, thus the satisfaction of the design-for-test conditions depends on the service provider. As a result, it is often expected for Web services to be modelled by SXM specifications that are not input-complete and output-distinguishable.

The previous subsection exposed some faults introduced by mutation-testing tools, which are detected due to the violation of the design-for-test conditions, and specifically input-completeness. However, an empirical evaluation of the above results suggests that the derived test sets are still quite powerful. In the SupplyOrder Web service they were capable of revealing all control-flow faults, as demonstrated in section 7.7.1. Furthermore, the test sets for both Account and SupplyOrder Web service examples were evaluated with very high scores (100%, except the score of 80% given by Jester to the SupplyOrder test set).

An industrial case study described in Vanak [89] has given encouraging coverage results for the SXM integration testing method even when the design-for-test conditions are not met. The method was applied to existing code that was not designed to meet those conditions and could not be augmented with extra functionality. Yet, the results were shown to be significantly better than the inhouse test sets. Statement and branch coverage were all over 94%, while predicate coverage was over 90%.

7.8 Summary

Having already described Web services with state and techniques for modelling those services as stream X-machines, the aim of this chapter was to investigate the application of SXM testing methods to derive test sets for Web services. Test set derivation was demonstrated with the SXM specifications of the two Web service examples, which do not satisfy the design-for-test conditions and are partially-specified. A number of unique testing considerations in the domain of Web services were explored along with proposed solutions (relating to contribution C4). The chapter concluded with the description of some experiments intended to evaluate the test sets derived earlier to reveal various faults, such as control flow faults and ones introduced by mutation tools (relating to contribution C5). Although the models did not satisfy all design-for-test conditions, and although they were partially specified, the experiments demonstrated that the test sets were significantly powerful in detecting various types of meaningful faults in the WSUT.

Part C – Approach for Run-Time Testing of Third-Party Web Services

- Chapter 8 – *Distributed Approach for Verification and Validation of Services in a SOA Environment*
- Chapter 9 – *Technical Approach for Testable Web Services with Stream X-Machines*
- Chapter 10 - *Toolset for Automated Testing of Web Services Modelled as SXM*

Chapter 8 – Distributed Approach for Verification and Validation of Services in a SOA Environment

The preceding chapters described the approach to formal modelling and complete functional testing of Web services behaviour based on SXMs. Complete functional testing of Web services is beneficial in a number of practical scenarios, both in development-time activities (testing and verification of services being developed), as well as in run-time activities (such as publication and discovery). In addition, complete functional testing of Web services can yield benefits for all types of stakeholders in a SOA environment, i.e. the service consumer, the service provider, and the service broker.

This section describes a novel approach based on formal modelling of Web service behaviour with SXMs. The SXM specifications are included as part of service WSDL descriptions and serve as advertisements of their behaviour. The described approach consists of validation of behavioural specifications against consumer needs and in verification of behavioural compliance of service implementations to their SXM specifications. The first part of this chapter provides an overview of the envisioned approach covering both (i) service discovery, and (ii) automated testing of services to be published. The second part focuses in more detail on the second objective, through describing practical challenges and possible solutions in implementing this objective.

8.1 The big picture

The approach proposed for registry-based testing and certification of Web services involves all three main stakeholders in a SOA environment, that is, service providers, service registries, and service requestors (consumers). As depicted in Figure 33, the role of each participant is associated with a number of activities. In brief, we propose that the behaviour of a Web service should be formally modelled at the provider-side, in order to facilitate registry-side verification at the time of service publication and requestor-side validation at the time of service selection.

The following three subsections give an overview of the activities performed by each participant in the scheme.

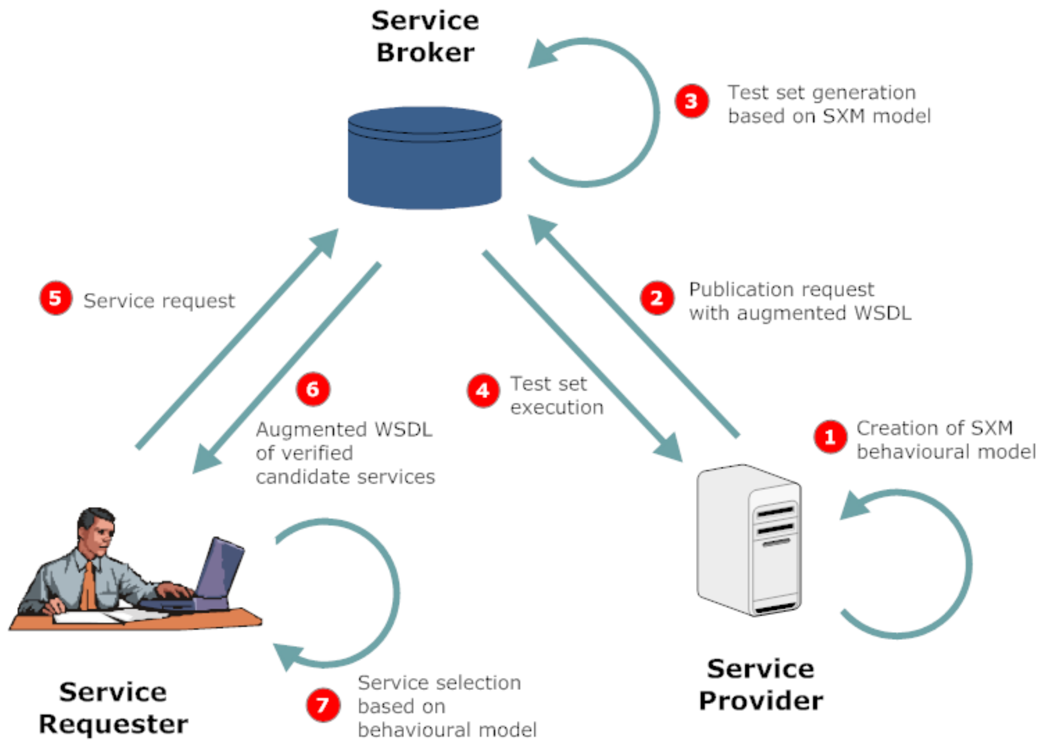


Figure 33 - Verification and validation approach of third-party Web services in a SOA environment

8.1.1 The service provider perspective

The service provider has three main tasks in augmenting the behavioural specifications to the service description: defining the (abstract) SXM model, defining the grounding of the model to the real Web service, and annotating the service description with the extra information.

The next step by the provider is the publication of the Web service to a service registry maintained by a broker. The publication query, which references the semantically annotated WSDL document at the provider site, initiates the publication procedure at the broker site.

8.1.2 The service broker perspective

A key role of the service broker in this approach is to verify the behaviour of the provided Web service implementation through model-based testing, and upon successful test results, to accept it in the service registry. This step is necessary to ensure that the implementation of the provided Web service really conforms to the advertised behavioural specifications. It is possible that this might not be the case, either because of insufficient testing at the provider site, or because of malicious

intent. With the attached SXM specification, the broker is able to derive the test sequences for verification automatically. Established SXM testing methods can be used to derive a complete, finite set of test cases, which is proven to find all faults in the implementation under test.

The input sequences and the expected output sequences produced by the testing algorithm are at the same level of abstraction as the SXM model, so they need to be mapped to concrete data types, which can be understood by the Web service. This is possible if the provider also includes necessary information to link the specification with the Web service implementation. This information is utilised by a test engine to run the test cases by communicating with the Web service under test through SOAP messages. If the test results are successful, i.e. the expected and produced outputs match, then the Web service implementation has been shown to be free of faults with respect to the behavioural specifications. In such a case, an advertisement of the Web service is created and added to the service registry, otherwise the Web service is rejected as faulty. The benefit of performing the verification procedure at the broker site, as opposed to performing it at the consumer site upon discovery, is that it needs to be done only once. Since only successfully tested Web services are accepted by the broker, consumers are ensured that the Web services they discover have been verified with respect to their specifications.

8.1.3 The service requester perspective

As a first step during discovery, the service consumer formulates a service request and submits it to the service registry. In response, the service broker returns a set of annotated service descriptions that match the service request. Notably, this approach is not bound to any particular matchmaking mechanism, so that any existing mechanism may be employed to perform syntactic or semantic matchmaking between the service request and the service advertisements. The service consumer can take advantage of the SXM behavioural model provided with each service candidate, in order to perform service selection. This is a validation process where the consumer ensures that a service model satisfies his or her requirements. An important validation technique is model animation, during which the user feeds the model with sample inputs and observes the current state, transitions, processing functions, memory values, and last but not least, the outputs. For example, X-System is a prolog-based tool supporting the animation of SXM models [55]. In addition, model checking may be employed on the SXM model to check for desirable or undesirable properties, which are specified in a temporal logic formula. Research on X-machines offers a model-checking logic, called XmCTL, which extends Computation Tree Logic (CTL) with memory quantifiers in order to facilitate model-checking of X-machine models [68]. Alternatively, if the consumer has a SXM model of the required service, it can be validated by state and transition refinement against the published SXM of the provided service.

8.2 Benefits of including SXM specifications in service descriptons

The benefits of augmenting WSDL with a formal behavioural specification for the SOA participants include the following:

- Explication of the conversation protocol enforced by the Web service for successful interoperability and binding.
- Explication of Web service behaviour and processing logic of individual operations beyond the WSDL operation signatures.
- The formal behavioural specification serves as a contract between the service provider and service requester in regards to expected service behaviour.
- The SXM model is available to model animation techniques that make it possible for human actors to understand the protocol and behaviour of discovered Web services.
- Being formal, the SXM specification is available to various analysis techniques, such as model checking (XmCTL).
- There is the possibility of eliminating false positives with incorrect behaviour during service discovery, matchmaking, and selection.

8.3 Testing scenarios

There are a number of situations where testing Web services offered by third-party providers is necessary. In these scenarios different stakeholders in a SOA environment need to test the WSUT as follows:

- *Test before you sell/provide (by the developer).* The third-party developer of the Web service performs the necessary functional testing before publishing it in a registry.
- *Test before you register (by the broker).* This is testing performed by the service broker, who acts as a certification authority for the service clients. The advantage of this approach is that services are tested only once and offered as verified services.
- *Test before you buy/consume (by the requestor).* In this scenario it is the requestor who performs all the necessary testing activities before consuming the offered Web service.

8.4 Discussion

The end goal of the described verification and validation approach is for service requestors to discover, select and consume Web services that fulfil their needs. That is, the approach validates the implementation of a Web service offered by a service provider against some informal user requirements.

In essence, the approach breaks down the validation goal into two main activities (Figure 34): (a) verification of the Web service implementation against a formal specification of its behaviour, and (b) validation of the service specification against the requirements of the service requestor, who in this case is a human individual. Service verification is handled by the service broker through functional testing of the service implementation against the advertised specification. Only Web services that are compliant with their specifications are accepted in the registry. On the other hand, the service requestor validates the advertised behavioural specification of a discovered Web service against his/her informal requirements. Since the specification represents the behaviour of a Web service that has already been verified to be compliant, this implies that also the Web service implementation has been validated against the user requirements.

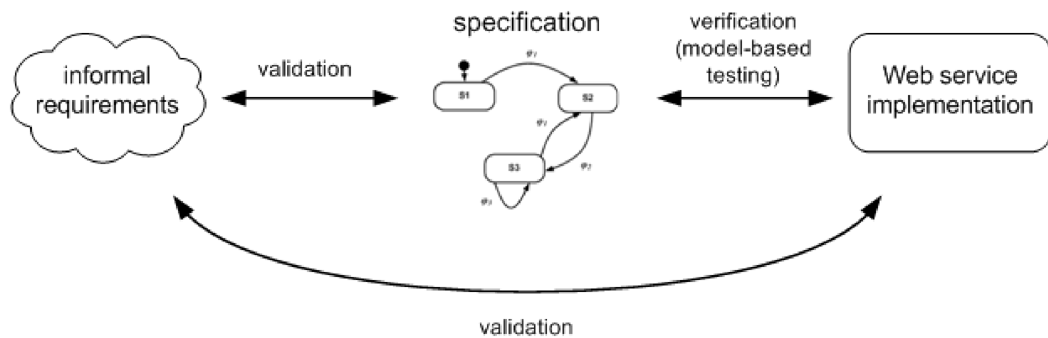


Figure 34 - Validation and verification paths

8.5 Summary

This chapter presented a novel Web service verification and validation approach, which relates to contribution C6 of this thesis. This collaborative approach relies on the previously described SXM-based Web service specification and testing techniques and involves the service provider, service broker, and service requestor. The two main activities are requestor-based validation of SXM models, and registry-based testing of third-party Web services specified by SXMs. The next chapter continues the treatment of this approach by focusing on the registry-based testing of Web services. It describes a technical solution based on open standards, which realises the vision of testable Web services.

Chapter 9 – Technical Approach for Testable Web Services with Stream X-Machines

The previous chapter described a novel approach for functional verification and validation of third-party Web services through a cooperative scheme involving the three main SOA stakeholders: the service provider, the service broker, and the service requestor. It described the advantages of explicating the internal behaviour of third-party Web services in an unambiguous, formal model, for the different involved participants. The role of each participant in this cooperative scheme was also investigated.

This chapter focuses in more depth on the service verification part. A standards-based technical solution is described, which aims to accomplish the vision of testable third-party Web services. It involves service providers who make available Web service descriptions augmented with extra information for testing, and certification authorities who utilise the extra information to derive test sets and execute them on the WSUT. The end goal of the described technical solution is to automate both activities of test case derivation and their execution on the Web service under test.

The problem of automated third-party Web service testing is split into three main parts:

a) annotating the WSDL document with extra information for testing;

b) extracting the information from the annotations; and

c) using that information to test the third-party Web service.

Annotation of WSDL documents with additional information is carried out by the service provider. This information includes the formal SXM specification of the Web service, and any grounding information that links the specification with the Web service implementation. The grounding information consists of transformation scripts, which map abstract inputs to concrete request messages, and concrete response messages to abstract outputs. Hence, these scripts serve to bridge the gap between the abstraction levels of the specification and the implementation.

The additional information supplied with WSDL is utilised by the tester to test the service. This information is in the form of annotations of different WSDL elements, thus the tester has to parse the document and extract the extra information to a convenient representation. The formal SXM specification is used as described in chapter 0 to derive a complete test set consisting of abstract inputs and expected outputs. The transformation scripts that are part of grounding annotations are used during test case execution to concretise abstract inputs to request messages and to transform response messages to abstract outputs for comparison.

This chapter starts with a discussion on the problem of bridging the abstraction gap between the SXM specification and the Web service implementation. It presents three different approaches to run the abstract test cases on the WSUT. Then the chapter focuses on the transformation approach with more technical details. It explains the concepts of lowerings and liftings, identifies a set of common patterns of mismatch between inputs and outputs, and concludes with some examples. The second section describes the mechanisms for annotating WSDL with a SXM model and with grounding information that consists of schema mappings. The mappings address inputs, outputs and service faults. Also the options for dealing with mismatches in operation names are described. The third section describes the approach implemented by the tester infrastructure to extract annotations and use them during test set generation and execution. Emphasis is placed on the mechanism for derivation of concrete request messages and their proper dispatching and for mapping of response messages to abstract outputs. The last section describes a technical solution for handling the two patterns introduced earlier in this thesis: the Manager pattern and the Constant field pattern. As it will be explained, such patterns are a convenient means for grounding the specification to the Web service implementation, since they do not require specification of schema mappings for all inputs and outputs.

9.1 Bridging the abstraction gap

Model-based testing requires the specification to be made more abstract than the system under test. If the specification were not more abstract than the SUT, then the efforts of validating[17] the specification would match the efforts of validating the SUT itself. Abstract models are simpler to understand and convey system behaviour among individuals, thus they are easier to validate.

Section 5.5 described a number modelling practices to accomplish abstraction by deliberately omitting details and losing information that is not considered essential to specification and testing. While the use of abstraction in SXM specifications is indispensable, and for the sake of intellectual mastery, desirable, it comes at a cost. Details of Web service behaviour that are not captured in the SXM specification obviously cannot be tested on the grounds of this specification. In addition, an important aspect of abstraction is data abstraction of requests and responses as input

[17] In the sense that an artifact is compared to often implicit, informal requirements.

and output symbols. This raises the need of bridging the gap between abstract and concrete inputs and outputs as test cases are applied on the WSUT.

The two main motives for having to bridge the gap between the SXM specification and the Web service for inputs and outputs during test case execution are:

- difference in the level of abstraction (as described above)
- difference in their representation.

Differences in the level of abstraction involve loss of information, which will have to be supplied in some way during concretisation. On the other hand, differences in representation between abstract test cases and concrete ones do not involve loss of information. However, abstract test cases are meant to be platform and language independent, while the SUT cat take various forms, such as a Web service, a class in an OOP programming language, a complete system, etc. Furthermore, concrete test cases can be implemented in the scripting language of a specific testing tool, such as JUnit, SOAPUI[18], etc.

9.1.1 Adaptation versus transformation

There are three basic approaches to bridging the abstraction level between the model and the implementation for the purpose of executing the test cases: a) adaptation, b) transformation, and c) hybrid. These different approaches are illustrated in Figure 35.

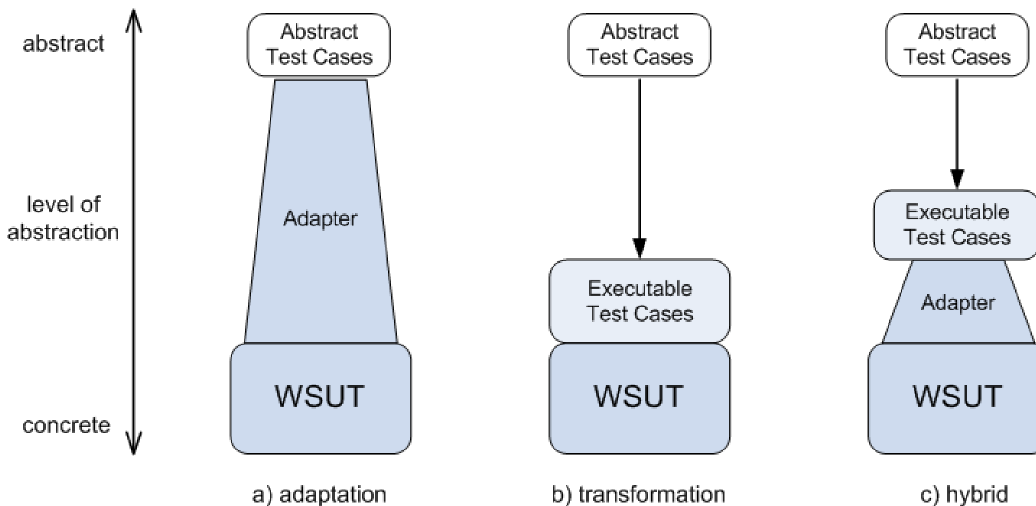


Figure 35 - Three approaches to bridging the abstraction gap (adopted from Utting and Legeard [34])

The adaptation approach involves manually writing adapter code that wraps the WSUT. Then the abstract test cases are run directly on the adapter. The transformation approach involves transformation of abstract test cases to executable test scripts that are understandable by some Web service testing tool, or directly to

[18] http://www.soapui.org/

concrete inputs that can be applied on the WSUT. The hybrid approach is a combination of the previous two approaches. It is sometimes useful to write some adapter code for the Web service under test to raise its abstraction level and make testing easier, and then transform the abstract tests into a more concrete form that matches the adapter interface.

To wrap up, there are the following possibilities regarding the correspondence between the SXM specification and the WSUT:

- No abstraction gap (usually when the WSUT is trivial)
- Abstraction gap (complex, real-world WSUT)
    - Adaptation (manual)
    - Transformation
        - Manual (impractical)
        - Automatic
    - Hybrid

between the specification and the implementation.

9.1.2 Adaptation

Adaptation is a manual approach and involves the service tester, who needs to implement the adapter. We can distinguish between two types of adapters: proper adapters and pseudo-adapters. The role of a proper adapter in the WS testing architecture is to bridge levels of abstraction, while a pseudo-adapter serves to bridge different representations. Thus, proper adapters raise the abstraction of a WSUT to a level where it can be tested without the supply of new information. As a result, they bridge the abstraction gap and cannot be generated automatically. The inputs and outputs accepted by proper adapters do not have to be necessarily the same as the automatically generated abstract inputs/outputs, since the latter can be automatically transformed to other formats, e.g. to JUnit test cases or to XML instances in accordance with some convenstions.

On the other hand, pseudo-adapters can be generated automatically, which bridge the representation gap of inputs/outputs or languages, but not the degree of abstraction. Java stubs are examples of pseudo-adapters that are used in both the adaptation and transformation approaches to hide direct SOAP communication with the Web service behind a local Java object. The stub can make use of generic, untyped object models of SOAP XML, or typed Java beans, in which SOAP XML documents are bound to Java objects (different binding options: xmlbeans, adb, jaxb, jibx, and so on).

Besides stubs, in the adaptation approach, proper Java adapters called wrappers can be employed as well. The Java wrappers wrap the Java proxy to bridge the abstraction gap. That is, proper adapters can wrap pseudo-adapters.

9.1.3 Transformation, lowering and lifting

Input to the SXM, as given by a test case, is concretized before it is sent to the Web service. The output of the latter is abstracted before it is compared to the output of the model as defined by the test case. The granularity of the comparison between the system's and the model's output depends on the desired precision of the test process.

The transformation approach offers the advantage of splitting the complexity of the Web service into an abstract model, and mapping definitions that perform concretizations and abstractions. These mapping definitions are then used during the execution of test cases on the Web service under test. In contrast to the adaptation approach, mappings can be defined by the modeller, thus the bridging of the abstraction gap is a provider-based task. The tester does not have to perform any mappings, since they are part of the Web service descriptions. Consequently, test case execution can be performed automatically by the tester.

There are two types of transformations: lowering and lifting. Lowering, as the term suggests, is the lowering of the abstraction level of a data entity, that is, its concretisation from abstract to concrete. On the other hand, lifting refers to the lifting of the abstraction level of a data entity, i.e. mapping it from concrete to abstract.

Lowering is used when deriving concrete inputs to be applied to the WSUT, while lifting is used to abstract returned outputs for comparison with outputs returned by the model. Generally, lifting is an easier and less challenging task than lowering, as the data is made more abstract, and no new information is introduced. On the other hand, lowering can be difficult, challenging or even impossible, since the concretisation of abstract inputs may require introduction of new information. The new information may be available in advance during concretisation of inputs or at runtime as the test cases are being executed.

9.1.4 Patterns of mismatch

This section lists a number of patters of mismatch between concrete and abstract inputs/outputs. The list is not necessarily exhaustive, but it includes very common cases, and provides an insight into the types of mismatch to consider when executing test cases.

Ranges of values

Often, abstract input symbols in a SXM specification do not represent single values, but ranges of values. For example, a stream X-machine may accept three inputs: r1, r2 and r3, where $r1 = \{0 \ldots 10\}$, $r2 = \{11 \ldots 20\}$, $r3 = \{20 \ldots \infty\}$. During lifting, the mapping determines the range to which a concrete value belongs and results in the abstract input. During lowering, a representative value belonging in each range is selected.

Enumerations

Enumerations in the abstract model offer a simplified view of complex data values such as enumerating a few typical values. For example, shopping cart items: i1, i2, i3, or customers c1, c2, c3 (defined as strings or enums), are mapped to representative XML instances, as specified by the modeller in the grounding.

Enumerations allow comparing outputs at a coarse granularity, without going to details; or when modelling the precise computations is impractical; resulting model may be nondeterministic. Sometimes an enumerated output can be simply a success or error output, which can be considered enough for obtaining a verdict on the correctness of the SUT.

Missing data fields

E.g. customer ID, shopping cart ID, etc provided in SOAP request messages.

Often repeated at the body of every SOAP message (such as the Amazon developer access key).

9.2 SAWSDL annotation mechanisms

In order to annotate the WSDL description file with additional information for testing, the SAWSDL[19] W3C recommendation for semantic annotations has been selected among other SWS alternatives. SAWSDL is both lightweight and non-intrusive, since it only augments WSDL with pointers to external semantic concepts. Furthermore it provides pointers to schema mappings which define the grounding of the abstract model to the WSDL descriptions. Remarkably, SAWSDL does not prescribe any particular representation language for the referenced concepts or any particular mapping language for realising the grounding. The referenced concepts could indeed be entities from an OWL ontology, rules, or even pictures. Therefore, owing to its lightweight nature and its support for schema mappings, SAWSDL is considered as suitable for accomplishing the annotation of WSDL with the JSXM specification, and with mappings between JSXM and XSD input/output representations.

Figure 36 illustrates the SAWSDL annotations of the WSDL file for the SupplyOrder Web service.

[19] Semantic Annotations for WSDL and XML Schema

Figure 36 - SAWSDL annotations of the SupplyOrder WSDL file with model references and schema mappings

Elements that can be annotated with modelReference in WSDL 1.1 are the following [29]:

- portType
- operation
- wsdl:fault
- message part
- XML Schema
    - simpleType
    - complexType
    - Global Element Declaration

Elements that can be annotated with loweringSchemaMapping and liftingSchemaMapping in WDL 1.1 are the following:

- Message part
- XML Schema
    - simpleType
    - complexType

- Global Element Declaration

9.2.1 Augmenting WSDL with the JSXM specification

The SXM model most closely models the WSDL portType, since this is the abstract part of the WSDL interface that abstractly defines which are the operations, inputs, outputs, and their types, leaving out protocol, and access details. That's why we annotate the portType. The portType/interface is annotated with a modelReference pointing to the location (URL) of the JSXM model.

9.2.2 Annotations for grounding

The grounding problem deals with generating concrete test request messages, dispatching them to the proper operation of the WSUT, and mapping the concrete response message to abstract outputs. So there are two parts, mapping inputs and outputs via schema mappings and mapping operations names for correct dispatching. Furthermore, Web services can return fault messages instead of normal SOAP responses. In SAWSDL it is possible to define schema mappings for faults as well.

Schema mapping annotations for inputs and outputs

The SAWSDL specification defines two annotations for grounding abstract inputs and outputs to WSDL messages or XML Schema types: *loweringSchemaMapping*, *liftingSchemaMapping* [29]. As their names suggest, loweringSchemaMapping annotations reference lowering transformations, while liftingSchemaMaping annotations reference lifting transformations.

Referenced transformations convert one XML document to another. In the context of test case execution, XML representations of abstract JSXM inputs would need to be transformed to XML-based SOAP request messages, while SOAP response messages would need to be transformed to XML representations of abstract JSXM outputs. The W3C Recommendation for a language defining transformations of XML documents is the XSL Transformations (XSLT) language [19], described in chapter 0.

XSLT transformations are based on an XML query language, of which the most common is XPath. However, in the SAWSDL working group non-normative example, RDF is used as the base of the semantic model [29]. Consequently, in order to lower RDF triplets to XML, XSLT is used in conjunction with an RDF query language like SPARQL, since with XPath this would have been a challenging task. On the other hand, for lifting, the input is SOAP XML and XPath is more appropriate as a query language. Thus in the working group example XSLT in conjunction with XPath is employed for lifting.

However, in the context of grounding a JSXM model to a WSDL Web service, both the concrete and abstract data are expressed in XML. Therefore, XPath is adopted

as the query language for transformations in both directions: lowering and lifting (Figure 37).

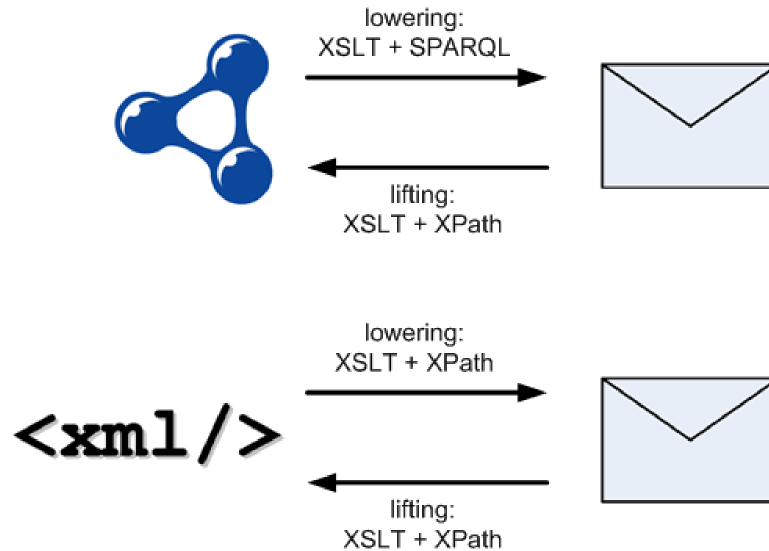


Figure 37 - Schema mapping languages for semantic RDF data versus schema mappings for JSXM inputs and outputs

Of all the schema mappings, the ones referenced by loweringSchemaMapping are most challenging to define, as they usually require the supply of new information. The loweringSchemaMappings cannot be avoided since they are necessary at least for converting abstract inputs to concrete SOAP messages for invoking the Web service under test. In contrast, transformations referenced by liftingSchemaMappings usually involve data abstraction and are easier to define.

Fault annotations

Web service faults are modelled in SXM either as normal output symbols or as default errors produced when inputs are not handled. As mentioned in section 7.5.1 on negative testing, there are two types of Web service faults: WSDL faults and runtime SOAP faults. WSDL faults are declared in the specification of service operations in the abstract part of WSDL descriptions. The document of the XSD type referenced by a WSDL fault becomes part of the WSDL fault detail element (see below). Web service faults do not always have to be declared in the WSDL descriptions, since service operations can return runtime SOAP faults, which are not anticipated in WSDL.

Nevertheless, from the investigation of various real-world Web services (Amazon ECWS, Google, UPS, FedEx, etc) we observed that the use of WSDL faults is rare. Instead, Web service faults are generated at runtime as SOAP faults whenever errors occur.

The components of a Web service fault according to SOAP 1.2 are the following (Figure 38):

- faultcode
- faultstring (corresponds to an exception message string, in Java)
- faultactor
- detail (may include the XML element defined in the XSD of WSDL)

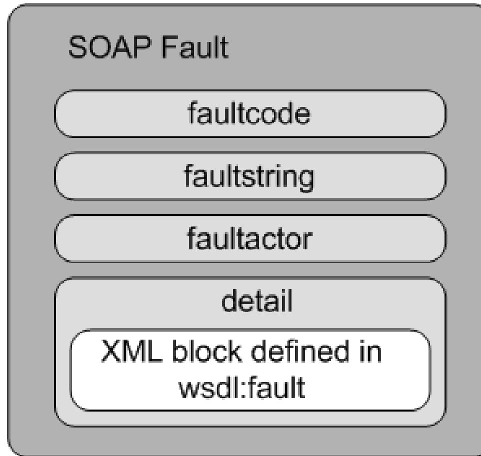


Figure 38 - Contents of a Web service fault message

The faultcode is a concise string identifying the type of fault, while faultstring is a textual fault description in natural language, intended for human individuals. The detail element contains the XML document defined by the XSD type in the WSDL specification of the fault. As a result, the following convention is assumed in specifying faults as SXM abstract outputs, and during the lowering of fault response messages to SXM outputs. The faultstring is specified as the SXM output name, while the XML block within the detail element is specified as SXM output results. Therefore, during mapping, the test engine constructs a SXM output with the same name as the faultcode. If there is any XML block within detail, it comprises the results elements of the SXM output. In case a loweringSchemaMapping annotation is defined in SAWSDL, the referenced transformation is applied to the fault XML block before it is inserted in the SXM output. The derived SXM output, and the output in the test set are then compared.

Operation names

As described earlier in the thesis, inputs model operation calls not just request messages, thus JSXM input names should represent names of operations to be called. Dispatching is based by default on calling the operation with the same name as the output.

There is a problem when *dispatching* (i.e. delivering the request to the appropriate Web service and operation) test messages when the JSXM input name and the operation name disagree. There are three main approaches to mapping JSXM input names to WS operation names, to correctly *dispatch* request messages.

9.2.3 Schema mapping examples

In this section are listed a number of illustrative examples of mismatches between low-level SOAP message elements and abstract SXM inputs or outputs.

Value mismatches

E.g. mismatch in the value of messages returned by the Web service:

SOAP body:

```
<message>Order successfully placed</message>
```

JSXM output instance:

```
<message>success</message>
```

XSLT transformation:

```
<xsl:template match="/">
    <xsl:choose>
        <message>
            <xsl:when test=
                "message == 'Order successfully placed'">
                success
            </xsl:when>

            <xsl:otherwise>
                error
            </xsl:otherwise>
        </message>
    </xsl:choose>
</xsl:template>
```

Element/attribute name mismatches

Mismatches in XML element and attribute names:

SOAP body:

```
<resultDetails>success</resultDetails>
```

JSXM output instance:

```
<message>success</message>
```

XSLT transformation:

```
<xsl:template match="/">
    <message>
        <xsl:value-of select="resultDetails"/>
    </message>
</xsl:template>
```

Structural mismatches

Structural mismatches involve differences in the XML structure, the arrangement of elements, attributes, etc.

E.g. the following extract from a JSXM input instance:

```
<fullName>First Last</fullName>
```

is mapped to:

```
<firstName>First</firstName>
<lastName>Last</lastName>
```

Structural mismatches are often a data mediation problem.

Missing elements

The concrete SOAP body payload and the abstract inputs/outputs can differ with complete XML elements. Since the SOAP body payload normally stands at a lower level of abstraction, mapping a JSXM input to a SOAP body payload requires introducing new data, such as new children elements.

JSXM input instance:

```
<input name="confirmOrder" /> (empty)
```

SOAP body:

```
<confirmOrder>
    <customerId>1234</customerId>
</confirmOrder>
```

Value ranges

Whole ranges of data values may be abstracted as symbols in the SXM model. Lifting is straightforward, while lowering requires selecting a specific value within the range. In the example below, the symbol `positive_value` is replaced by a random positive number by the XSLT script during lowering schema mappings.

JSXM input instance:

```
<input name="updateQuantity">
    <quantity>
          positive_value
    </quantity>
<input>
```

SOAP body:

```
<quantity>
    15.5
</quantity>
```

Enumerated values

As mentioned in chapter 0, enumerations are an important technique in data abstraction. They are discrete values (booleans, enumerated types, strings, etc) that represent complex XML instances. During input concretisation enumerated values are replaced by carefully chosen example XML instances. During outputs comparison, complex XML data instances are mapped to the corresponding abstract enumerations.

JSXM input type:

```
<input name="addItem">
    <arg name="item" type="xs:string" />
</input>
```

JSXM input instance:

```
<input name="addItem">
    <item>
          item1
    </item>
<input>
```

SOAP body:

```
<addItem>
    <item>
          <itemId>1</itemId>
          <ASIN>0131858580</ASIN>
          <type>Book</type>
          <author>Thomas Erl</author>
          <title>Service-Oriented Architecture</title>
    </item>
</addItem>
```

9.3 Runtime mapping mechanisms

The previous section described how guidelines from the SAWSDL W3C recommendation are leveraged to augment WSDL descriptions with the JSXM model and grounding information. Having accomplished these annotations it should be possible to utilise the SXM model and ground it to the real Web service at runtime. However, the SAWSDL specification does not recommend an approach to properly map the schemas while executing the Web service. Instead, this task has been left to the SAWSDL processor that implements the specification, with different processors possibly supporting different approaches.

Therefore, it is important to define in this section the steps of the algorithm used to map data types during runtime and to execute the Web service operations. The implementation of the SAWSDL processor must overcome a number of obstacles:

- Dealing with the unidirectional nature of SAWSDL annotations: lifting schema mappings annotate WSDL output types explicitly, but lowering schema mappings do not annotate inputs in the abstract SXM model.
- Locating a lowering schema mapping for an abstract test input in the SAWSDL descriptions.
- Dispatching the resulting request message to the proper operation.
- Locating the lifting schema mapping for an incoming response message in SAWSDL descriptions.
- Defining the action taken when no schema mapping is defined for a request or response message.

The answer to such questions is not given by the SAWSDL specification, but left to the tool implementers who design the mapping approach. The problem of locating the proper schema mappings in a unidirectional annotations framework like SAWSDL is handled during the extraction of schema mappings described later in this section.

The first subsection investigates the correspondence between inputs/outputs in JSXM specifications and in Web services under test. Therefore, a set of default mapping rules are defined for inputs and outputs, in order to bridge the representation gap. These default mappings are applied:

- When no schema mappings are specified in the SAWSDL annotations for inputs and outputs (i.e. there is no abstraction gap).
- Before the application of schema mappings to abstract inputs or after the application of XSLT schema mappings to outputs, if any (i.e. there is an abstraction gap).

The subsequent subsections describe the approach for extracting schema mappings, runtime transformation of inputs and outputs, and the approach for performing proper dispatching.

9.3.1 Correspondence between Web service and JSXM inputs and outputs

Before discussing mappings let us first examine how Web service inputs and outputs XML types and instances correspond to JSXM inputs and outputs.

Derivation of test SOAP request messages involves the construction of complete SOAP envelopes from the abstract JSXM inputs. In case a SOAP processor tool (such as Apache Axis2) is used, it involves the supply of the information required the tool to construct SOAP envelopes. In this section we describe how abstract inputs correspond to SOAP request messages and operation calls, as well as how SOAP response messages correspond to abstract outputs.

Some pre-processing is performed according to a set of default rules, before inputs are transformed, or after outputs are transformed. As described earlier, in the JSXM notation inputs consist of an input name, and optionally, one or more arguments, each of which defined by its name and XSD type. The correspondence with the domain of Web services is relatively straightforward. The input name corresponds to the name of the service operation to be called, whereas the input arguments correspond to the contents of the SOAP body payload.

We can identify three different cases for JSXM inputs (Figure 39):

- input with a name but no arguments (i.e. simple input),
- input with one argument, and
- input with several arguments.

For simple inputs with no arguments, the operation is called with an empty SOAP body payload. For inputs with one argument, the single document is put in the SOAP body. Inputs with more than one argument are a different case that may cause problems. In the treatment of messaging styles earlier in this thesis, it was explained that the WS-I Basic Profile restricts the maximum number of WSDL message parts to one. For interoperability reasons, this work assumes that Web services under test are WS-I compliant, thus the derived request messages must contain at most one root element (document) in the SOAP body. Therefore, when complex inputs contain multiple arguments, the latter are included as children of a root element with the same name as the input. Finally, transformation may also involve addition of namespaces to XML element names; however these technical details are out of scope of this discussion and are handled by the tool implementation.

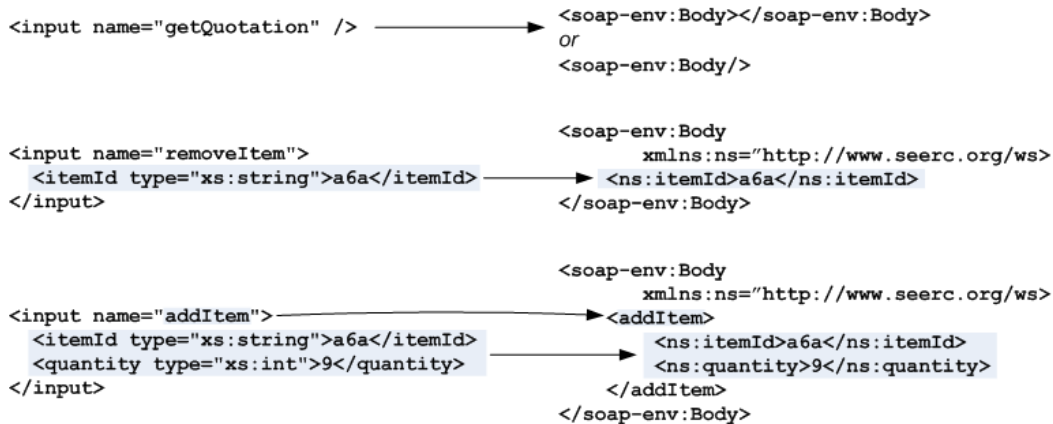


Figure 39 – Conventions for correspondence at the instance level between JSXM inputs and SOAP requests

In mapping outputs, the root element of the body payload becomes the output name. The children elements are treated as results of the JSXM output (Figure 40).

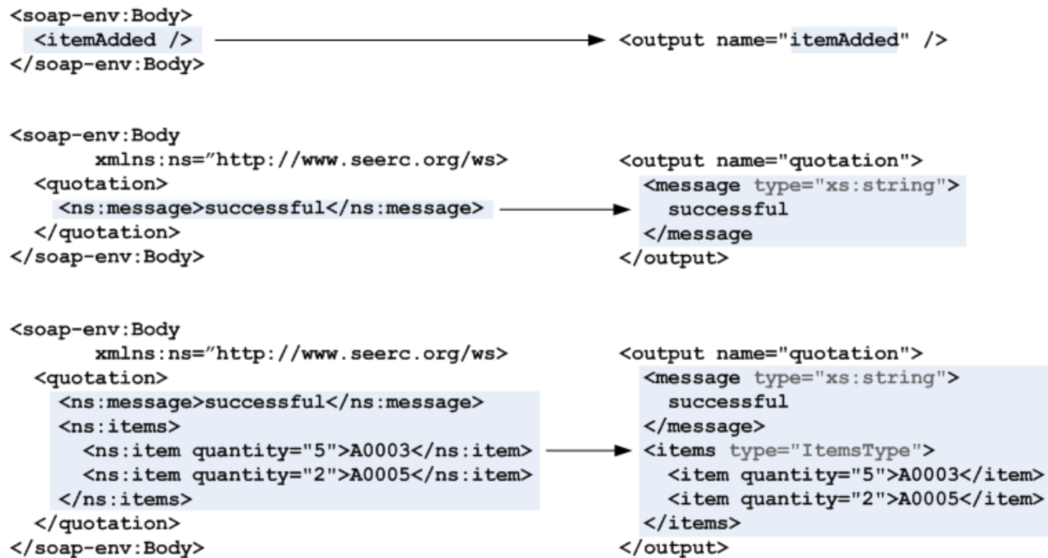


Figure 40 – Convention for correspondence at the instance level between SOAP responses and JSXM outputs

9.3.2 Extracting the schema mappings

The goal of extracting the schema mappings is to locate the lowering XSLT scripts for abstract inputs to generate test request messages, and lifting XSLT scripts for incoming response messages to compare with abstract outputs. The latter is easy, since liftingSchemaMapping annotates the WSDL output (message part/GED/GTD) to be lifted for comparison. However, extracting the lowering schema mappings for JSXM inputs is not straightforward since the loweringSchemaMapping does not annotate the JSXM input to be lowered but the Web service input to be obtained (reverse direction). For this reason an extra modelReference annotation is required for Web service inputs, pointing to the input in the JSXM file (separated by a #).

When annotating a WSDL input/output, it is unclear whether we annotate:

- The message part
- The Global Element Declaration in the XML Schema
- The type declaration in the XML Schema

The SAWSDL parser component of the tool should start from the message part, to the element name and if applicable to the referred XSD type (simple or complex) in that order, to decide whether a mapping exists, and if yes, which one.

A mapping table is constructed in the memory for easy fetching of the schema mappings for every operation: one for lowering of operation inputs and one for lifting of operation outputs. This map is consulted during the execution of test cases as described below.

9.3.3 Mapping types

The mapping table described in the previous subsection consists of (operation name, lowering XSLT script, lifting XSLT script) triplets for all operations.

The steps performed during lowering are:

1. Get next abstract input in test sequence.
2. Get the children elements of the input (arguments) if any.
3. Pre-process the arguments to derive a preliminary XML document according to the default rules defined earlier. This preliminary document would be used as the SOAP payload if there is no XSLT script.
4. Fetch the lowering XSLT from the map, if any.
5. Apply it to the preliminary document derived in step 3.
6. Use the transformation result as the payload of the SOAP request message and dispatch it either directly to the Web service, or as an argument to a Java stub, with the operation/method having the input name.

The steps performed during lifting are:

1. Retrieve the response message body payload either as direct XML message or as a Java binding representation returned by the Java stub's method.
2. Fetch the lifting XSLT script from the map, if any.
3. Apply the transformation and compare the result with the expected output.

Therefore, the inputs sent by the test engine are the results of applying the specified lowering transformations. On the other hand, the results of applying the specified lifting transformations, rather than the actual outputs, are compared with the outputs predicted by the test oracle. A question that naturally arises is whether one should trust the mappings. Checking whether the mappings make sense is a validation task that can be performed by a human individual. It is considered easier for the human individual to validate separately the abstract specification and the individual mappings (which in the absence of automation would have to be performed manually), rather than to validate a large, unabstracted, specification that does not make use of mappings.

9.3.4 Dispatching approach

Since the name of the target operation is not specified in a standard place in service requests (e.g. it can appear in HTTP headers, in SOAP headers via WS-Addressing, in the SOAP body, etc), it is easier to leave the technical details to the Web service platform, such as Apache Axis2. Axis2 can build Java stubs automatically, which provides methods with the same names as Web service operations. By invoking the stub's methods with argument the SOAP body payload, the stub invokes the corresponding operation on the Web service. The XML document passed as

argument to the method comprises the body of the SOAP request message sent to the invoked Web service operation. The return value of the sub's method is the body of SOAP response message from the invoked Web service operation.

Therefore, the stub's method with the same name as the input name is called. The SOAP body that results from the mappings described earlier in this section is passed as an argument to the method.

9.4 Handling the Constant Field Pattern and Manager Pattern

Often, specifying schema mappings for every single input and output is a non-trivial and time-consuming task. Although schema mappings are often necessary for achieving automatically testable Web services, in some cases they can be omitted by the modeller. This is especially the case when adding identifier information to test inputs. As described in section 5.5.1 on modelling individual state objects, identifier information can be excluded from the SXM specification and the generated abstract test cases. They can be supplied later on during test case execution as inputs are concretized to request messages.

One approach is to specify XSLT lowering schema mappings, which insert identifier fields into the abstract inputs to derive the payload of request messages. This has to be done fore every input declaration in JSXM. Nevertheless, as mentioned earlier in this thesis, it is frequently possible for the per-object SXM specification and the modelled Web service to follow a pattern. Patterns are identified in those situations when the mismatch between the specification and the implementation fits the pattern for all inputs or outputs. In the case of identification, identifiers can be the same for all request messages in a conversation and are inserted in the same location in the XML tree of the SOAP payload. In these situations there is the opportunity to specify the identifier information and its location in every request only once, rather than write XSLT transformations for each and every input.

As described in section 4.3.4: Private state identification, there are two main cases the identifier information is known: either in advance or retrieved from the server during run time. In the first case, an identified state entity exists in the Web service before test sequences are executed, thus their identifiers are known in advance. This case is defined by the Constant Field Pattern. The second case is when an object identifier is obtained at runtime from the service after an object is created with the create operation. The obtained identifier is included in every subsequent invocation to the service, until the identified object is finally destroyed with a destroy operation. This second case is defined by the Manager Pattern introduced earlier in this thesis (Section 5.5.1).

This section proposes solutions for realizing both patterns. On the one hand, the modeller who specifies annotations should annotate the Web service portType with a description of the pattern. On the other hand, the test engine implementation

consults this pattern description during execution of the test cases. Identification information is supplied by the test engine after the application of any optional schema mapping that is specified for the input.

9.4.1 Constant Field Pattern

In the *constant field* pattern, there are constant data elements that are repeated in every request message, e.g. an access key or an object identifier. Writing a transformation script for inserting the constant data element in every type of request message is time consuming and error-prone. Therefore, it is desirable to have this task handled by the test engine automatically by declaring the service as one following the constant field pattern, and supplying an XPath expression locating the identifier.

Another possibility would be XQuery [17], however it is too elaborate and is used for complex queries, while selecting children elements can be successfully performed by XPath alone. For inserting the identification information in requests, it would be necessary to use XQuery update facility [18], but again due to the simplicity of the update operation, an XPath expression for the location to insert the ID is sufficient, which translates to XSLT.

Example:

Request message (taken from the Amazon E-Commerce service documentation [56]):

Before XSLT transformation:

```
<input name="ItemSearch">
    <Keywords>Pink Floyd</Keywords>
</input>
```

After XSLT transformation:

```
<ns:ItemSearch xmlns:ns="...">
    <ns:AWSAccessKeyId></ns:AWSAccessKeyId>
    <ns:Request>
          <ns:Keywords>Pink Floyd</ns:Keywords>
          <ns:SearchIndex>Music</ns:SearchIndex>
    </ns:Request>
</ns:ItemSearch>
```

Pattern specification:

```
<pattern name="ConstantField">
    <field location="ns:ItemSearch\ns:AWSAccessKeyId">
          0KRFZH9WHG92C4VK1B02
    </field>
    ...
    <field location="xpath_expression2">field2</field>
    ...
```

```
</pattern>
```

After application of pattern:

```
<ns:ItemSearch xmlns:ns="...">
    <ns:AWSAccessKeyId>0KRFZH9WHG92C4VK1B02</ns:AWSAccessKeyId>
    <ns:Request>
          <ns:Keywords>Pink Floyd</ns:Keywords>
          <ns:SearchIndex>Music</ns:SearchIndex>
    </ns:Request>
</ns:ItemSearch>
```

Complete SOAP request message:

```
<soapenv:Envelope xmlns:soapenv="...">
  <soapenv:Body>
    <ns:ItemSearch xmlns:ns="...">
      <ns:AWSAccessKeyId>0KRFZH9WHG92C4VK1B02</ns:AWSAccessKeyId>
      <ns:Request>
        <ns:Keywords>Pink Floyd</ns:Keywords>
        <ns:SearchIndex>Music</ns:SearchIndex>
      </ns:Request>
    </ns:ItemSearch>
  </soapenv:Body>
</soapenv:Envelope>
```

9.4.2 Manager Pattern

The *manager* pattern [72], also known as the Factory pattern, gives rise to new challenges in generating concrete test cases for Web services, since the object identifier to be supplied with every operation invocation is only known at run time, after it is returned from the response of the object creation operation. Therefore, the generation of concrete test cases is deferred to run time and is accomplished by the test engine.

Since the object identifier (e.g. shopping cart ID) may be nested deep in the XML response message of the object creation operation (e.g. CartCreate) we propose adding to the Manager Pattern annotation the *XPath expression* that selects the identifier information. Additionally, to insert the identifier in the correct place in the subsequent request messages directed to a specific object, another XPath expression is specified, which the test engine uses to dynamically generate an XSLT script.

Example

Pattern Specification:

```
<pattern name="Manager">
    <creation name="create"
identifierLocation="xpath_expression">
    <invocation identifierLocation="xpath_expression">
    <destruction name="destroy">
</pattern>
```

9.5 Summary

The focus of this chapter was the Web service verification portion of the collaborative approach described in the previous chapter (contribution C7). After introducing the problem of bridging the abstraction gap between the SXM specification and the Web service implementation, three approaches were described for running the abstract test cases on the WSUT. The technical solution proposed subsequently adopts the transformation approach through grounding annotations with XSL transformation scripts (contribution C8 of this thesis). While SXM inputs are transformed to SOAP request messages (lowering schema mappings), returned SOAP response messages are transformed to SXM outputs for comparison at an abstract level (lifting schema mappings). Next, this chapter presented a mechanism for annotating Web services with extra information, in order to accomplish the vision of *testable* third-party Web services. According to this mechanism, service providers perform the annotations of WSDL descriptions (using the SAWSDL W3C recommendation), while certification authorities utilise the extra information in the annotations to derive test sets and execute them on the WSUT. The implementation of a toolset, which automates the testing activities in the above mechanism, is described in the next chapter.

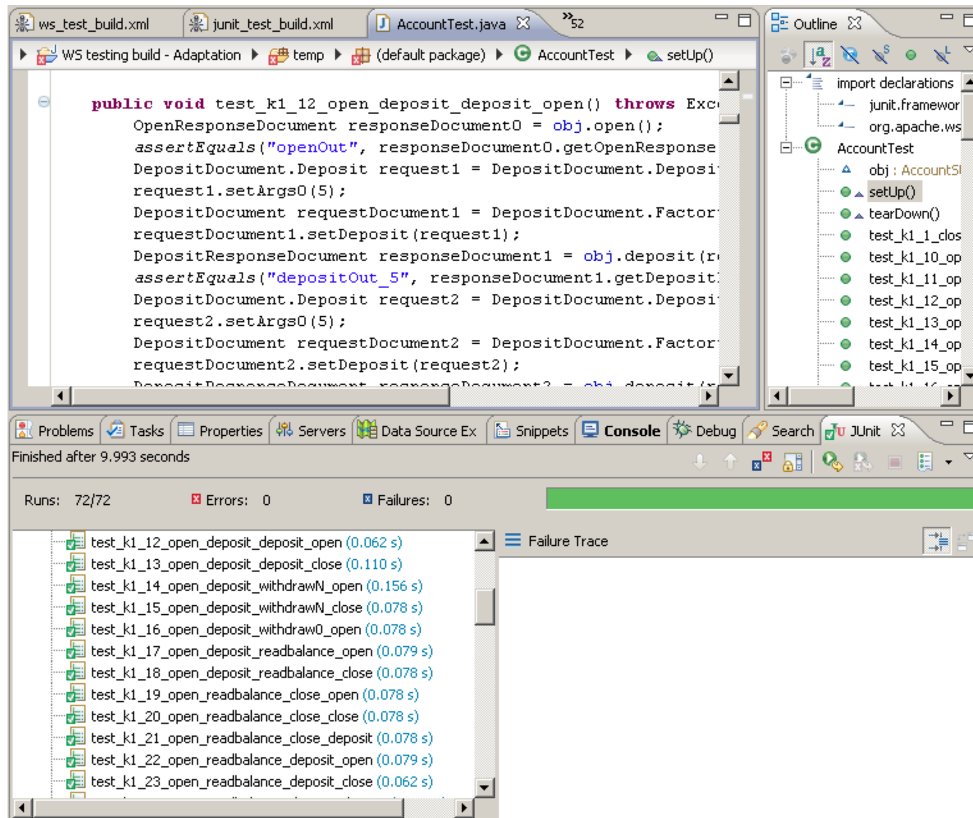
Chapter 10 – Toolset for Automated Testing of Web Services Modelled as SXM

The technical approach described in the previous chapter to automate testing of third-party Web services is supported by a toolset, which implements the activities of the certification authority for extracting semantic annotations, generating abstract test cases, and performing runtime mappings during test case execution.

The tool relies on the JSXM test case generator described earlier.

10.1 Test case execution toolset

- Different architectural alternatives
- Adapter-based versus Transformation-based
- Comparison table



10.2 Review on available tools and libraries for writing Web service tools

- Apache Axis, etc
  - Approach for writing stubs, skeletons, etc.
- Specification, animation, and abstract test case generation tool (JSXM)
- Automatic execution of test cases on WSUT, Java adapters
- UDDI service registry integrating the automated Web services testing tool
  - Extended FUSION semantic registry that uses JUnit programmatically
  - PublicationManager's addService()
  - Returns a AddServiceResponse containing a test results report and registration status (successful/not)

10.3 Used tools/APIs:

- WSDL4J (object model for reading, manipulating, and creating WSDL documents)
- ~~SAWSDL4J~~ EasySAWSDL, Woden4SAWSDL (object model for SAWSDL annotations), v 1.1 vs v 2.0
- Apache HTTPClient (service invocation client)
- XMLUnit (for comparing XML messages), some discussion
- Apache Axis2 (Web service platform)

- Jaxen (XPath Engine)
- Xalan (XSLT engine)
- FUSION Semantic Registry (certification authority)
- JSXM toolset.
- Ant integration

The approach for model-based testing of Web services is based on the architecture depicted in Figure 41. In this particular approach the transformation (rather than adapter) method is used to execute test cases. That is, abstract SXM input/output symbols are mapped to SOAP requests/responses that are used to communicate with the Web service under test. The benefit of this approach is that the concrete representation of inputs/outputs is SOAP XML rather than a vendor-dependent representation that usually is required by adapters (e.g.

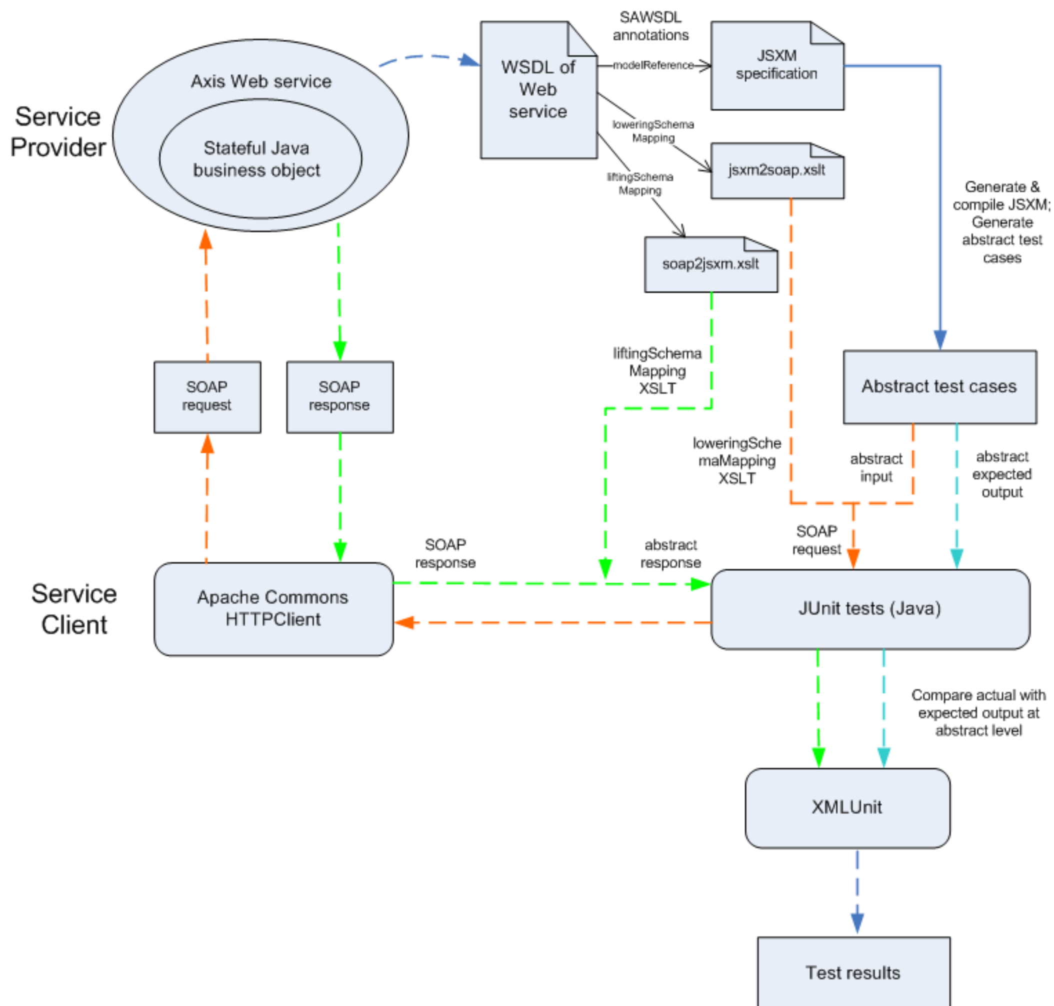


Figure 41 - Transformation approach for executing test cases

All the steps depicted in the above picture are integrated in an ANT script. Given the annotated SAWSDL document, consisting of model reference and schema

mapping annotations as prescribed in the previous chapter, the JSXM specification, and the schema mapping scripts, it is possible to fully automate testing and verification of the Web service relative to the specification.

Apart from parsing SAWSDL annotations, the toolset should also offer the ability to create these annotations, when the modeller wants to link the WSDL with the SXM specifications and the schemaMappings.

Since human intervention is not required, it is possible to integrate the above script for fully automated testing of testable Web services into certification authorities. Usually such authorities are service registries, which may consist of hundreds or thousands of Web services. Testing is invoked upon registration request by service providers, and, if all the test cases pass successfully, the service is accepted and registered. As a result all registered services in the service registry have been verified for behavioural conformance to their advertised JSXM specifications.

A number of registries implementing the UDDI specification are available, some of them open source. However, in order to support the approach proposed in this thesis, the registry should support Web services described by SAWSDL documents. These registries are also known as semantic registries, and offer additional benefits, such as semantic discovery based on inputs, outputs, and category. One open source implementation of a semantic registry supporting SAWSDL is the FUSION Semantic Registry [13].

This registry is implemented as three different Web services, exposing interfaces for registry administration, Web service registration, and Web service discovery. In this thesis we are interested in the registration API. The registration API includes operation “addService”, which is invoked upon registration by service providers. We have modified the implementation of this operation to invoke the Web service testing script.

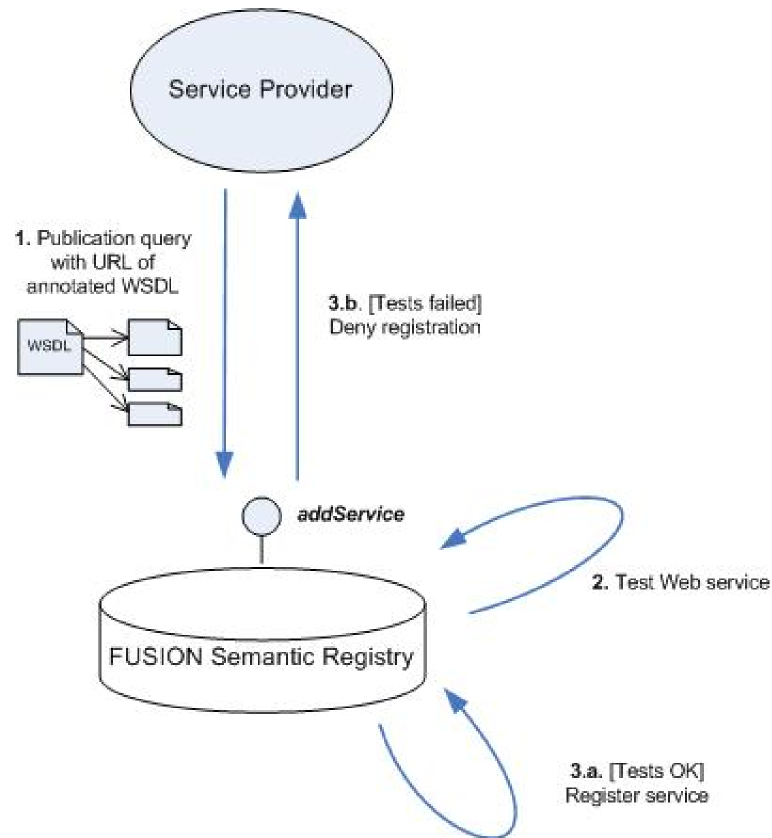


Figure 42 - Extension of the FUSION Semantic Registry with service verification capabilities

10.4 Summary

This chapter concludes the description of the work performed as part of this PhD research. It described the tool implemented to support runtime testing of third-party Web services (contribution C9), drawing from the techniques presented in the previous chapter. The tool takes advantage of another existing tool for SXM-based test case generation, and other APIs for the rest of the tasks. In particular, the tool relies on EasySAWSDL for parsing annotations in testable Web services to extract the SXM model and the schema mappings used during test case execution. Furthermore, the tool is incorporated into an open-source service registry, which tests Web services prior to their registration.

Chapter 11 – Conclusions and Future Work

This final chapter concludes the thesis. Therefore, at this point it presents a summary of the findings and contributions of this research work. Also, in retrospective, this chapter critically examines the fulfillment of the original aims that constitute the thesis, and points out some of the limitations that have been identified. The last section presents ideas for future work that can be inspired from this thesis.

11.1 Summary of findings

The main accomplishments of the research work described in this thesis are briefly summarised as follows:

- A comprehensive study on stateful Web services, service state and its effects on service behaviour. Also, a classification of Web services into a few practical categories, based on their state characteristics. The implications of service state on the tasks of specification and testing are further examined.
- An investigation of the suitability of different state-based formalisms, with a focus on SXMs, for specifying both the behaviour and internal data of stateful Web services.
- Recommendation of guidelines and best practices to create SXM specifications of stateful Web services. Ad-hoc practices are proposed in the domain of Web services, owing to their specific characteristics. Also a method is described for inferring a SXM from IOPE-based declarations of Web service operations.
- Investigation of unique testing challenges in the Web services domain.
- Evaluation of produced test sets with faulty implementations and mutation tools.
- Description of a novel collaborative approach for validation and verification of third-party Web services, with a focus on service verification through testing.
- Realisation of third-party Web service testing in the above approach through technical solutions for:

  - o providers to annotate the services they offer with additional descriptions, which include the formal SXM specification of their behaviour, and the grounding for mapping between abstract and concrete inputs/outputs;
  - o testers to utilise the supplied descriptions to derive test cases and run the tests on the WSUT.
- Solution for bridging the abstraction gap between the specification and the WSUT, both through transformations of individual inputs/outputs and through definition of patterns.
- An architecture and toolset that extends an existing tool (JSXM), for supporting the above techniques for testers verifying Web services.
- A set of examples for demonstrating and validating the described techniques.

11.2 In support of the initial aims

11.2.1 Formal verification and testing of stateful Web services with stream X-machines

One of the original aims of this research work has been to model stateful Web services with formal SXM specifications. Among the different benefits of specifying the behaviour of Web services is testing their implementation. Formal methods were used in the hope of automating the process of testing (see below) and ensuring the effectiveness of the generated test sets.

As chapters 0 and 0 demonstrated, the stream X-machine formalism is quite appropriate for specifying stateful Web services, as it is capable of capturing both dynamic behaviour and internal service state in unambiguous specifications. Moreover, SXMs are seen as fairly intuitive as they have close correspondence with the implementation elements of stateful Web services. A number of techniques were employed to perform abstraction in the specification without making it nondeterministic. Also, as the two case studies demonstrated, it was feasible to derive complete SXM specifications of Web services of varying complexity, which could then be expressed in the JSXM notation for processing by tools. More importantly, those SXM specifications, with automation support, could be used to derive test sets with proven fault detection effectiveness. Although the models did not satisfy all design-for-test conditions, and although they were partially specified, a number of experiments were able to demonstrate that the test sets were significantly powerful in detecting various types of meaningful faults in the WSUT.

Limitations

As mentioned, one of the limitations of the SXM-based specification and testing approach is that it is difficult to satisfy the design-for-test conditions, which are often fairly restrictive. This is especially the case with Web services, since the tester might not have control over the implementation in order to augment it for compliance with the design-for-test conditions.

Another quite problematic assumption is that the individual processing functions corresponding with components in the implementation have already been proven to be correct. Although techniques are suggested by existing work to continue the hierarchical testing process in a similar fashion beyond the integration level, in practice this was considered difficult for a number of reasons. First of all, as demonstrated in the testing chapter, it can be difficult to separate control flow from implementation of the individual components, thus it is difficult to decouple them in the implementation to ensure their correctness. Secondly, from a preliminary examination it was found difficult to express individual processing functions in terms of the function computed by simpler SXMs, since the latter handle sequences of symbols rather than individual input and output symbols. Finally, the JSXM toolset does not yet support test set generation from complete hierarchical specifications. The issue of testing individual members of the machine type is considered as future work.

Finally, although large-scale and commercial Web services are commonly nondeterministic, this thesis was not focused on NSXMs. The test set generation tool is also unable to support test set generation from nondeterministic specifications, which require different versions of the algorithm. Nevertheless, a concise treatment of nondeterminism and conformance testing was provided in this thesis, and is considered as future research. Moreover, abstraction techniques were proposed, which can preserve the determinism of specifications.

11.2.2 Feasibility of testing third-party Web services

Another important aim set out in the beginning of this research was towards testing of third-party Web services, which have recently become pretty common and require methods to ensure their reliability. As was explained, third-party Web services need to be tested at runtime, as opposed to development time. In addition, testers usually do not possess functional service specifications, due to the limitations of WSDL, and have no control on the service implementation.

Testing of third-party Web services has been tackled with a novel approach described starting from chapter 0. This approach requires the cooperation of both providers and certification authorities, such as brokers. Providers append the WSDL descriptions with the formal SXM specification and additional grounding information, so that the services are verifiable by prospective requestors. This vision was addressed with methods and technical solutions based on various standards, which make it achievable in practice.

Limitations

Although the approach for testing third-party Web services is technically feasible with the described techniques, in practice it may be difficult to adopt. First of all, it requires cooperation between different stakeholders (providers, certification authorities, and requestors), thus it requires broad acceptance in the industry.

In addition, the third-party Web service testing approach requires, in a number of circumstances, the availability of sandbox versions from service providers. It is necessary to avoid testing the production version of the service for various reasons, such as shared data repositories, undesirable side effects, etc. This is also a reason for not managing to test a commercial third-party Web service in this work, since providers often do not make such versions available. One question that naturally arises from testing the sandbox interface is whether its verified implementation is indeed the same as the real implementation. However, it can be assumed that providers offer the sandbox service as a different deployment (instance) of the same implementation.

Finally, it is possible for a service provider to offer different versions of a Web service implementation at different times. If a Web service has been verified by a certification authority as correct, it does not necessarily mean that the current version is the same as the verified one. As a result, the tester should be able to know whenever different versions of a Web service are offered by a service provider. Thus, a limitation of the described approach is that it does not yet provide a solution to the versioning issue, which is left as future work.

11.2.3 Degree of test automation

A key research question in this work has been to explore the degree to which Web service testing can be automated. Automation is desirable in order to remove the testing burden from service testers, so as to make service verification affordable for requestors and third-party certification authorities.

As mentioned earlier, the use of formal methods allows derivation of abstract test sets with automation support. However, the major obstacle that had to be overcome was the execution of those abstract test sets on a concrete (and probably less abstract) Web service implementation. Consequently, methods and technical means have been proposed for the provider or modeller to specify grounding information in the WSDL description, apart from augmenting it with the SXM specification. These techniques are based on widely-accepted standards, such as SAWSDL, XSLT, and XPath, which are W3C recommendations.

Limitations

Although automation of both test set derivation and test case execution has been demonstrated to be technically possible, it requires substantial effort from the service provider to specify all the extra information, especially the schema mappings. In cases of simple Web services, no schema mappings might be necessary, but for complex commercial Web services it is highly demanding to specify mappings for all inputs and outputs. This problem is also enforced by the fact that no tool has been developed in this work to support the modeller in performing SAWSDL annotations. Thus it has been left as future work.

Nevertheless, the use of various patterns, described in section 9.4, requires only a minimal pattern descriptor file, which substantially simplifies the task of grounding the SXM model to the WSDL description.

11.2.4 Tool support

It has been a key objective to support some of the techniques proposed in this thesis through tool automation. Development of tools was considered important not only to provide automation, but also to demonstrate the practicability of the described methods. Although, as said earlier, no tool has been developed for the modeller to annotate WSDL and to define schema mappings, a toolset has been developed with the aim of supporting the activities of the tester. It relies on the JSXM test case generation tool which derives abstract test sets as sequences of input/output pairs from JSXM specifications. This tool has been extended to support execution of the test cases on the Web service under test using the approach described in section 0.

Nevertheless, although the architecture that supports the testing approach has been defined, and the tool implementation is under constant improvement, the latter has not yet been completed to support runtime mappings of inputs and outputs, as well as the two described patterns. However, at this point, the tool is able to extract the information from SAWSDL annotations, including the JSXM specification, an is able to run the test cases on Web services under test that are at the same level of abstraction as the specification. The default mapping rules defined in section 9.3.1 are supported by the tool.

11.3 Future work

During the period this research has been performed, additional work has been under consideration for addressing various issues, which had to be left out of the scope of this thesis due to time and priority constraints. Moreover, numerous ideas and opportunities for further investigation have appeared during this research. Therefore, this section provides ideas on possible future research that could be inspired by the work described in this thesis.

11.3.1 Testing individual processing functions

As mentioned in the previous section, it is important to address the restrictive assumption that the SXM and the WSUT contain identical processing functions. There are different alternative solutions that should be further researched. There have been two distinct attempts to test processing functions with SXMs: the work on complete DSXM testing with hierarchical decomposition [65], and the work in Ipate 2007 [82]. These testing algorithms are extensions of the SXM integration testing method and require more elaborate specifications. Besides the option of SXM-based testing, individual processing functions can also be tested with complementary methods, such as equivalence class and boundary testing. These other methods do not require enhanced SXM specifications.

Further work should examine the implication of adopting any of the above alternatives on the presented testing approach. It needs to take into consideration the fact that testing is performed at runtime on Web service implementations, which are not under the control of the tester (as apposed to development-time testing).

11.3.2 Nondeterministic Web services and specifications

Since commercial Web services are often complex and involve nondeterministic factors, it is not always feasible to specify their functionality with deterministic SXM models. Although most of the techniques proposed in this thesis do not require specifications to be deterministic, focus has been given to the latter.

Further work needs to investigate in more depth the feasibility of specifying Web services with nondeterministic SXMs. Thus, it will be possible to cover a much wider range of Web services, including large-scale ones and services with nondeterministic behaviour. Furthermore, to allow automation support for NSXMs, the JSXM tool must be extended to compile, animate, and generate test cases (for conformance as well as for equivalence). The applicability of the algorithms that have been devised for testing NSXMs should be examined for Web services, since besides the test function they also involve an adaptive test process.

11.3.3 Testing service compositions

Since services are often used as part of compositions and orchestrations, an important area that requires further research is verification of service compositions. One of the problems that arise when attempting to test a service composed of other services is that, to an external requestor, composite services appear identical with usual atomic services. On the other hand, SXM-based testing can be employed by the developer who has control of the implementation under test and of the orchestration code that implements business processes. As an example, BPEL processes consist of multiple steps, transitions, activities and data that persists between activities. It might be possible to specify them with SXMs, since the latter are capable of capturing control flow. Nevertheless, a problem that was identified in this work was that the individual steps of BPEL orchestrations are not controllable with inputs, thus it is not feasible to drive the different paths during testing. Instead, the user provides one input to a BPEL orchestration, which upon completion provides an output.

A further direction of research is to specify the internal behaviour of composite Web services by also modelling atomic services. As explained in section 5.8 atomic services invoked by the service under test are not modelled, thus they represent nondetermistic factors. If those atomic services are specified as well, then it is possible to test for equivalence using deterministic SXMs. SXM varieties that may be investigated for this purpose include the JSXM model of interacting SXMs [59], as well as Communicating Stream X-Machines (CSXMs) [93].

11.3.4 Editor and graphical modelling tool for JSXM specifications

The JSXM notation defines a complete syntax and is supported by automation tools for different activities that make use of JSXM specifications. However, currently there is no tool available for editing JSXM files, which often may suffer from inconsistencies and from syntax errors. A graphical editor tool for JSXM would also be convenient for modellers to quickly create SXM specifications. The graphical editor would also be handy for human individuals with minimal mathematical background or knowledge of SXMs, who would like to visualise an available specification of a Web service. For example, visualisation of the state-transition diagram of a SXM would assist the service requestor in validating the behaviour of a provided Web service in the approach described in chapter 0.

11.3.5 Graphical tool for SAWSDL annotations and mappings

As mentioned above, it has been out of the scope of this research work to develop tools that facilitate the job of the modeller. One of the demanding modelling tasks the modeller has to perform is the annotation of WSDL files with model references pointing to the SXM specification and JSXM input/output definitions, as well as with schema mappings that point to XSLT transformations mapping between abstract and concrete data. For the latter, standard XSLT editors can be used to facilitate the task.

As regards the SAWSDL annotation tool, it should be similar to existing tools that facilitate annotation of WSDL files with concepts from OWL ontologies, such as Radiant [94] from the Meteor-S project. In essence, this graphical tool should load the JSXM specification and the WSDL file to be annotated and depict them as trees. Model reference annotations can then be accomplished with drag-and-drop actions, without having to deal directly with SAWSDL XML files.

11.4 List of Publications by the Author

Several papers related to the research presented in this PhD thesis have been published in various journals and conferences by the author (either as first author or co-author). These papers and their relationship to the thesis contributions, as stated in Section 1.3, are listed in Table 5:

Table 5 - List of publications by the author and relationship to contributions

Contribution	Papers
C1	[95]
C2	[98]
C3	[52], [98]

C4	[81], [96], [97]
C6	[81], [96], [97]
C9	[52]

In addition, the author has published two more research papers, which do not correspond directly to any of the above contributions, but nonetheless are relevant to the research work described in this thesis. The first paper [99] presents an overview of the area of service-oriented software engineering and investigates unique issues in the development of service-oriented applications. The second paper [100] is a state-of-the-art survey on the existing service-oriented development methodologies, introducing a novel framework for the evaluation and classification of those methodologies.

Glossary and Acronyms

Glossary

- **Control state** – Member of the set Q of a SXM specification.
- **Memory state** – Values of the memory element M of a SXM at a particular instant.
- **Service broker** – Participant in a SOA that provides a service registry.
- **Service instance** – Separate instance of the software implementing a service, which is spawned by the service infrastructure to serve a particular requestor during a session.
- **Service provider** – Participant in a SOA that makes services available to service requestors.
- **Service registry** – A repository of service descriptions where service providers can publish their service descriptions and service requestors can search for services.
- **Service requestor/client/consumer** – Participant in a SOA that interacts with a service. The terms “requestor”, “client”, and “consumer” are used interchangeably to indicate the same concept.
- **Stateful resource/object; state object; context object** – Logical entity consisting of state data, having a well-defined lifecycle, and accessed by one or more Web services.

Acronyms

- **ASM** – Abstract State Machine
- **BPEL** – Business Process Execution Language
- **EFSM** – Extended Finite State Machine
- **FSM** – Finite State Machine
- **GED** – Global Element Declaration
- **IOPE** – Inputs, Outputs, Preconditions, Effects
- **MEP** – Message Exchange Pattern
- **QoS** – Quality of Service
- **RPC** – Remote Procedure Call
- **SAWSDL** – Semantic Annotations for WSDL
- **SLA** – Service Level Agreements
- **SOA** – Service Oriented Architecture
- **SOAP** – Simple Object Access Protocol
- **SOC** – Service Oriented Computing
- **STS** – Symbolic Transition System
- **SUT** – System Under Test
- **SWS** – Semantic Web Services
- **SXM** – Stream X-Machine
- **UDDI** – Universal Description Discovery and Integration

- **WS-CDL** – Web Service Choreography Description Language
- **WS-I** – Web Services Interoperability
- **WSI-BP** – Web Services Interoperability (WS-I) Basic Profile
- **WSCL** – Web Services Conversation Language
- **WSDL** – Web Services Definition/Description Language
- **WSMO** – Web Service Modeling Ontology
- **WSRF** – Web Services Resource Framework
- **WSUT** – Web Service Under Test
- **XSD** – XML Schema Definition

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Roadmap of Service Oriented Computing. Available: http://infolab.uvt.nl/pub/papazogloump-2006-96.pdf. March 2006.

[2] B. Lublinsky, "Defining SOA as an architectural style," *IBM developerWorks*, 09-Jan-2007. [Online]. Available: http://www.ibm.com/developerworks/architecture/library/ar-soastyle/.

[3] M. Gudgin et al., "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." W3C Recommendation, 27-Apr-2007. Available: http://www.w3.org/TR/soap12-part1/.

[4] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Service Definition Language (WSDL) 1.1." W3C Note, 15-Mar-2001. Available: http://www.w3.org/TR/wsdl.

[5] T. Bellwood et al., "UDDI Version 2.04 API Specification." UDDI Committee Specification, 19-Jul-2002. Available: http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

[6] C. von Riegen et al., "UDDI V2.03 Data Structure Specification." UDDI Committee Specification, 19-Jul-2002. Available: http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm.

[7] M. Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM*, vol. 36, no. 7, pp. 75-84, Jul. 1993.

[8] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[9] G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services: Concepts, Architectures and Applications*, 1st ed. Springer, 2004.

[10] S. Gao, C. M. Sperberg-McQueen and H. S. Thompson, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures." W3C Candidate Recommendation, 21-Jul-2011.

[11] R. Chinnici, J.-J. Moreau, A. Ryman and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language." W3C Recommendation, 26-Jun-2007. Available: http://www.w3.org/TR/wsdl20/.

[12] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham and P. Yendluri, "Basic Profile - Version 1.0 (Final specification)." WS-I - Web Services Interoperability Organization, 16-Apr-2004. Avaliable: http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html.

[13] D. Kourtesis and I. Paraskakis, “Web Service Discovery in the FUSION Semantic Registry,” in *Business Information Systems*, vol. 7, W. Abramowicz and D. Fensel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 285-296.

[14] M. Gudgin, M. Hadley, T. Rogers and Ü. Yalçinalp, “Web Services Addressing 1.0 - WSDL Binding: Section 5.1 - WSDL 1.1 Message Exchange Patterns.” W3C Candidate Recommendation, 29-May-2006. Available: http://www.w3.org/TR/ws-addr-wsdl/#WSDL11MEPS.

[15] R. Butek, “Which style of WSDL should I use?” [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/.

[16] J. Clark and S. DeRose, “XML Path Language (XPath), Version 1.0.” W3C Recommendation, 16-Nov-1999. Available: http://www.w3.org/TR/xpath/.

[17] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie and J. Simeon, “XQuery 1.0: An XML Query Language (Second Edition).” W3C Recommendation, 14-Dec-2010. Available: http://www.w3.org/TR/xquery/.

[18] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton and J. Simeon, “XQuery Update Facility 1.0.” W3C Recommendation, 17-Mar-2011. Available: http://www.w3.org/TR/xquery-update-10/.

[19] J. Clark, “XSL Transformations (XSLT), Version 1.0.” W3C Recommendation, 16-Nov-1999. Available: http://www.w3.org/TR/xslt.

[20] K. Czajkowski et al., “The WS-Resource Framework, Version 1.0.” The Globus Alliance, 03-May-2004. Available: http://www.globus.org/wsrf/specs/ws-wsrf.pdf.

[21] I. Foster et al., “Modeling Stateful Resources with Web Services - Version 1.1.” IBM developerWorks, 03-May-2004.

[22] E. Newcomer and I. Robinson, “Web Services Coordination (WS-Coordination) Version 1.2.” OASIS Standard, 02-Feb-2009.

[23] A. Banerji et al., “Web Services Conversation Language (WSCL) 1.0.” W3C Note, 14-Mar-2002. Available: http://www.w3.org/TR/wscl10/.

[24] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon and C. Barreto, “Web Services Choreography Description Language Version 1.0.” W3C Working Draft, 09-Nov-2005.

[25] M. Pistore, M. Roveri and P. Busetta, “Requirements-Driven Verification of Web Services,” *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 105, pp. 95–108, 2004.

[26] D. Jordan and Evdemon, Eds., “Web Services Business Process Execution Language Version 2.0.” OASIS Standard, 11-Apr-2007. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[27] H. Lausen, A. Polleres and D. Roman, “Web Service Modeling Ontology (WSMO).” W3C Member Submission, 03-Jun-2005. Available: http://www.w3.org/Submission/WSMO/.

[28] D. Martin et al., “OWL-S: Semantic Markup for Web Services.” W3C Member Submission, 22-Nov-2004. Available: http://www.w3.org/Submission/OWL-S/.

[29] J. Farell and H. Lausen, “Semantic Annotations for WSDL and XML Schema.” W3C Recommendation, 28-Aug-2007. Available: http://www.w3.org/TR/sawsdl/.

[30] J. Kopecky, D. Roman, M. Moran and D. Fensel, “Semantic Web Services Grounding,” in *Advanced Int’l Conference on Telecommunications and Int’l Conference on Internet and Web Applications and Services (AICT-ICIW’06)*, Guadelope, French Caribbean, 2006, pp. 127-127.

[31] J. Rao and X. Su, “A Survey of Automated Web Service Composition Methods,” in *Semantic Web Services and Web Process Composition*, vol. 3387, J. Cardoso and A. Sheth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 43-54.

[32] “610-1991: IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries.” IEEE Computer Society, 1991.

[33] J. Tretmans, *Testing Techniques*. The Netherlands: University of Twente, 2002. Available: http://www.cs.aau.dk/~kgl/TOV04/tretmans-notes.pdf.

[34] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Amsterdam; Boston: Morgan Kaufmann Publishers, 2007.

[35] J. Offutt, S. Liu, A. Abdurazik and P. Ammann, “Generating test data from state based specifications,” *Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25-53, Jan. 2003.

[36] G. Canfora and M. Di Penta, “Testing services and service-centric systems: challenges and opportunities,” *IT Professional*, vol. 8, no. 2, pp. 10-17, Mar. 2006.

[37] W. T. Tsai, Y. Chen, R. Paul, N. Liao and H. Huang, “Cooperative and group testing in verification of dynamic composite web services,” in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, Hong Kong, pp. 170-173.

[38] M. Wirsing et al., “Sensoria Process Calculi for Service-Oriented Computing,” in *Trustworthy Global Computing*, vol. 4661, U. Montanari, D. Sannella and R. Bruni, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 30-50.

[39] H. Foster, S. Uchitel, J. Magee and J. Kramer, “Model-based verification of Web service compositions,” in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Montreal, Que., Canada, pp. 152-161.

[40] S. Narayanan and S. A. McIlraith, “Simulation, verification and automated composition of web services,” in *Proceedings of the eleventh international conference on World Wide Web - WWW '02*, Honolulu, Hawaii, USA, 2002, p. 77.

[41] S. Hinz, K. Schmidt and C. Stahl, “Transforming BPEL to Petri Nets,” in *Business Process Management*, vol. 3649, W. M. P. Aalst, B. Benatallah, F. Casati and F. Curbera, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 220-235.

[42] S. Nakajima, “Model-checking verification for reliable Web service,” in *Proceedings of OOPSLA'02 Workshop on Object-Oriented Web Services*, Seattle, USA, 2002.

[43] G. Diaz, J.J. Pardo, M.E. Cambronero, V. Valero and F. Curartero, Automatic Translation of WS-CDL Choreographies to Timed Automata. In *Proceedings of WS-FM'05*, LNCS-3670, Springer, pp 230–242.

[44] J.S. Dong, Y. Liu, J. Sun and X. Zhang, Verification of Computation Orchestration via Timed Automata. In *Proceedings of ICFEM'06*.

[45] J. Lemcke and A. Friesen, “Composing Web-service-like Abstract State Machines (ASMs),” in *2007 IEEE Congress on Services (Services 2007)*, Salt Lake City, UT, USA, 2007, pp. 262-269.

[46] W. T. Tsai, R. Paul, Yamin Wang, Chun Fan and Dong Wang, “Extending WSDL to facilitate Web services testing,” in *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, Tokyo, Japan, pp. 171-172.

[47] W.-T. Tsai, Y. Chen and R. Paul, “Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems,” in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Sedona, AZ, USA, pp. 139-147.

[48] R. Heckel and L. Mariani, “Automatic Conformance Testing of Web Services,” in *Fundamental Approaches to Software Engineering*, vol. 3442, M. Cerioli, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 34-48.

[49] A. Bertolino, L. Frantzen, A. Polini and J. Tretmans, “Audition of Web Services for Testing Conformance to Open Specified Protocols,” in *Architecting Systems with Trustworthy Components*, vol. 3938, R. H. Reussner, J. A. Stafford and C. A. Szyperski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1-25.

[50] C. Keum, S. Kang, I.-Y. Ko, J. Baik and Y.-I. Choi, “Generating Test Cases for Web Services Using Extended Finite State Machine,” in *Testing of Communicating*

Systems, vol. 3964, M. Ü. Uyar, A. Y. Duale and M. A. Fecko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 103-117.

[51] A. Sinha and A. Paradkar, "Model-based functional conformance testing of web services operating on persistent data," in *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications - TAV-WEB '06*, Portland, Maine, 2006, pp. 17-22.

[52] E. Ramollari, D. Kourtesis, D. Dranidis and A. J. H. Simons, *Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing*. Springer Berlin / Heidelberg, 2009.

[53] P. Kefalas, *X-Machine Description Language: User manual, version 1.6*. City College, 2000.

[54] D. Dranidis, G. Eleftherakis and P. Kefalas, "Object-based language for generalized state machines," *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, vol. 1, no. 3, pp. 8-17, 2005.

[55] P. Kapeti and P. Kefalas, "A Design Language and Tool for X-machines Specification," *Advances in Informatics*, pp. 134-145, 2000.

[56] "Product Advertising API, Developer Guide, API Version 2010-10-01." Amazon.com, 2010.

[57] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed and S. Vanak, "Testing methods for X-machines: a review," *Formal Aspects of Computing*, vol. 18, no. 1, pp. 3–30, 2006.

[58] United Parcel Service of America (UPS), "UPS OnLine Tools: Shipping Web services developers guide." 27-Jul-2007.

[59] D. Dranidis, *JSXM: A suite of tools for model-based automated test generation: User manual*. City College, 2009.

[60] E. Ort and B. Mehta, "Java Architecture for XML Binding (JAXB)." Oracle Technology Network, Mar-2003.

[61] Holcombe, M. and Ipate, F. (1998). *Correct Systems: Building Business Process Solutions*. Springer-Verlag, Berlin.

[62] F. Ipate and M. Holcombe, "An Integration Testing Method that is Proved to Find all Faults," *International Journal of Computer Mathematics*, vol. 63, no. 3/4, pp. 159-178, 1997.

[63] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178- 187, May. 1978.

[64] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines-a survey,” *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090-1123, Aug. 1996.

[65] F. Ipate, “Complete deterministic stream X-machine testing,” *Formal Aspects of Computing*, vol. 16, no. 4, pp. 374–386, 2004.

[66] G. Laycock, “The Theory and Practice of Specification-Based Software Testing,” PhD Thesis, Department of Computer Science, University of Sheffield, 1993.

[67] S. Eilenberg, *Automata, languages, and machines, Volume 59A*. New York, NY, USA: Academic Press, 1974.

[68] G. Eleftherakis, “Formal Verification of X-machine Models: Towards Formal Development of Computer-based Systems,” PhD Thesis, University of Sheffield, 2003.

[69] R. M. Hierons and M. Harman, “Testing Conformance to a Quasi-Non-Deterministic Stream X-Machine,” *Formal Aspects of Computing*, vol. 12, no. 6, pp. 423-442, Dec. 2000.

[70] K. T. Cheng and A. S. Krishnakumar, “Automatic functional test generation using the extended finite state machine model,” in *Proceedings of the 30th international on Design automation conference - DAC '93*, Dallas, Texas, United States, 1993, pp. 86-91.

[71] A. J. H. Simons, K. Bogdanov and M. Holcombe, *Complete functional testing using object machines*. Department of Computer Science, University of Sheffield: Technical Report CS-01-18, 2001.

[72] C. Atkinson, D. Stoll, H. Acker, P. Dadam, M. Lauer and M. Reichert, “Separating per-client and pan-client views in service specification,” in *Proceedings of the 2006 international workshop on Service-oriented software engineering - SOSE '06*, Shanghai, China, 2006, p. 47.

[73] R. Fielding, U. Irvine, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee, “RFC 2068: Hypertext Transfer Protocol - HTTP/1.1.” Jan-1997.

[74] D. Box et al., “Web Services Addressing (WS-Addressing).” W3C Member Submission, 10-Aug-2004.

[75] A. Suriarachchi, “Stateful Web Services with Axis2,” *WSO2 Oxygen Tank*, 22-Jul-2009. [Online]. Available: http://wso2.org/library/articles/stateful-web-services-axis2#session_mgt.

[76] P. Wang, “Web services programming tips and tricks: Build stateful sessions in JAX-RPC applications,” *IBM developerWorks*, 02-Sep-2004. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-tip-stateful/index.html.

[77] BEA Systems, “Creating Conversational Web Services,” *WebLogic Web Services: Advanced Programming*. [Online]. Available: http://download.oracle.com/docs/cd/E11035_01/wls100/webserv_adv/conversation.html.

[78] BEA Systems, “Designing WebLogic Web Services,” *WebLogic Web Services: Advanced Programming*. [Online]. Available: http://download.oracle.com/docs/cd/E13222_01/wls/docs81/webserv/design.html#1058330.

[79] S. Hidayatullah and S. Fulkerson, “Implement and access stateful Web services using WebSphere Studio, Part 1,” *IBM developerWorks*, 09-Mar-2004. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-statefulws/index.html.

[80] S. Hidayatullah, A. Jaipaul and R. Subramanian, “Implement and access stateful Web services using WebSphere Studio, Part 4,” *IBM developerWorks*, 27-Jul-2004. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-statefulws4/.

[81] D. Kourtesis, E. Ramollari, D. Dranidis and I. Paraskakis, “Increased Reliability in SOA Environments through Registry-Based Conformance Testing of Web Services,” *Engagement in Collaborative Networks. Special Issue in International Journal of Production Planning & Control: The Management of Operations (JPPC)*, vol. in press, 2009.

[82] F. Ipate and R. Lefticaru, “State-based Testing is Functional Testing,” in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Windsor, UK, 2007, pp. 55-66.

[83] F. Ipate, “Testing against a non-controllable stream X-machine using state counting,” *Theoretical Computer Science*, vol. 353, no. 1, pp. 291–316, 2006.

[84] F. Ipate and M. Holcombe, “Generating Test Sets from Non-Deterministic Stream X-Machines,” *Formal Aspects of Computing*, vol. 12, no. 6, pp. 443-458, Dec. 2000.

[85] C. Bourhfir, R. Dssouli and E. Aboulhamid, “Automatic Executable Test Case Generation for Extended Finite State Machine Protocols,” in *Proceedings of IWTCS’97*, 1997, pp. 75-90.

[86] S. A. Irvine, Tin Pavlinic, L. Trigg, J. G. Cleary, S. Inglis and M. Utting, “Jumble Java Byte Code to Measure the Effectiveness of Unit Tests,” in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Windsor, UK, 2007, pp. 169-175.

[87] M. Ivan, “Jester - a JUnit test tester.,” presented at the eXtreme Programming and Flexible Processes in Software Engineering - XP200, 2000.

[88] F. Ipate and M. Holcombe, “A method for refining and testing generalised machine specifications,” *International Journal of Computer Mathematics*, vol. 68, no. 3, pp. 197-219, 1998.

[89] S. Vanak, “Complete Functional Testing of Hardware Descriptions,” PhD Thesis, University of Sheffield, 2002.

[90] W. T. Tsai, R. Paul, Weiwei Song and Zhibin Cao, “Coyote: an XML-based framework for Web services testing,” in *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, Tokyo, Japan, pp. 173-174.

[91] M. Veanes, C. Campbell, W. Schulte and N. Tillmann, “Online testing with model programs,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 273, Sep. 2005.

[92] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang and P. Douangsavanh, “The ModelJUnit test generation tool,” 15-May-2009. [Online]. Available: http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.

[93] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe and C. Vertan, “Communicating stream X-machines systems are no more than X-machines,” *Journal of Universal Computer Science*, vol. 5, no. 9, pp. 492-507, 1999.

[94] K. Verma, D. Brewer, A. Sheth and J. Miller, “Radiant: A tool for semantic annotation of Web Services Karthik Gomadam,” presented at the 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland, 2005.

[95] D. Dranidis, E. Ramollari and D. Kourtesis, “Run-time Verification of Behavioural Conformance for Conversational Web Services,” in *Proceedings of the 7th IEEE European Conference on Web Services (ECOWS 2009)*, Eindhoven, Netherlands, 2009, pp. 139–147.

[96] D. Kourtesis, E. Ramollari, D. Dranidis and I. Paraskakis, “Discovery and Selection of Certified Web Services Through Registry-Based Testing and Verification,” in *Pervasive Collaborative Networks*, L. M. Camarinha-Matos and W. Picard (Eds.), IFIP International Federation for Information Processing, Springer Boston, pp. 473–482.

[97] E. Ramollari, D. Dranidis and A. J. H. Simons, “Reliable Web Service Discovery based on Formal Behavioural Modelling,” in *Proceedings of the 3rd South East European Doctoral Student Conference (DSC 2008)*, vol. 2, Thessaloniki, Greece: South-East European Research Centre (SEERC), pp. 302–314.

[98] D. Dranidis, D. Kourtesis and E. Ramollari, “Formal Verification of Web Service Behavioural Conformance through Testing,” *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 36–43.

[99] E. Ramollari, D. Dranidis and A. J. H. Simons, “A Survey of Service Oriented Development Methodologies,” in *Proceedings of the 2nd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2007)*, Leicester, UK.

[100] E. Ramollari, D. Dranidis and A. J. H. Simons, “State-of-the-Art and Future of Service Oriented Software Engineering,” in *Proceedings of the 2nd South East European Doctoral Student Conference (DSC 2007)*. Thessaloniki, Greece: South-East European Research Centre (SEERC).

[101] EURACE Project, “FLAME and Agent-Based Modelling”, [Online]. Available: http://www.eurace.org/index.php?TopMenuId=3