

---

# An Empirical Investigation of SSDL

Patric Fornasier

School of Computer Science and Engineering, University of New South Wales, Australia  
[patricf@cse.unsw.edu.au](mailto:patricf@cse.unsw.edu.au)

---

*A thesis submitted in partial fulfilment of the requirements for the degree of:*

**Master of Science in  
Computer Science & Engineering**



**UNSW**  
THE UNIVERSITY OF NEW SOUTH WALES  
SYDNEY • AUSTRALIA

Sydney, October 27, 2007



# An Empirical Investigation of SSDL

**Patric Fornasier**

School of Computer Science and Engineering, University of New South Wales, Australia  
[patricf@cse.unsw.edu.au](mailto:patricf@cse.unsw.edu.au)

*A thesis submitted in partial fulfilment of the requirements for the degree of:*

**Master of Science in  
Computer Science & Engineering**



**UNSW**  
THE UNIVERSITY OF NEW SOUTH WALES  
SYDNEY • AUSTRALIA



## Abstract

The SOAP Service Description Language (SSDL) is a SOAP-centric language for describing Web Service contracts. SSDL focuses on message abstraction as the building block for creating service-oriented applications and provides an extensible range of protocol frameworks that can be used to describe and formally model Web Service interactions. SSDL's natural alignment with service-oriented design principles intuitively suggests that it encourages the creation of applications that adhere to this architectural paradigm. Given the lack of tools and empirical data for using SSDL as part of Web Services-based SOAs, we identified the need to investigate its practicability and usefulness through empirical work. To that end we have developed Soya, a programming model and runtime environment for creating and executing SSDL-based Web Services. On the one hand, Soya provides straightforward programming abstractions that foster message-oriented thinking. On the other hand, it leverages contemporary tooling (i.e. Windows Communication Foundation) with SSDL-related runtime functionality and semantics. In this thesis, we describe the design and architecture of Soya and show how it makes it possible to use SSDL as an alternative and powerful metadata language without imposing unrealistic burdens on application developers. In addition, we use Soya and SSDL in a case study which provides a set of initial empirical results with respect to SSDL's strengths and drawbacks. In summary, our work serves as a knowledge framework for better understanding message-oriented Web Service development and demonstrates SSDL's practicability in terms of implementation and usability.



### **Originality Statement**

“I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.”

Signed .....

Date .....





### **Copyright Statement**

“I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International.

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.”

Signed .....

Date .....



### **Authenticity Statement**

“I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.”

Signed .....

Date .....



## **Publications**

Parts of this work have been published in the following conference proceedings:

P. Fornasier, and J. Webber, and I. Gorton, “Soya: a programming model and runtime environment for component composition using ssdl,” in *Component-Based Software Engineering*, pp. 227–241, Springer Berlin / Heidelberg, 2007.



# Acknowledgements

Very rarely is the successful completion of a project the achievement of one person alone. Indeed, this work would not have been possible without the support of faculty members, colleagues, fellow students, family, partner and many dear friends all over the world.

In particular, I would like to thank my supervisors Prof. Ian Gorton and Prof. Ross Jeffrey as well as my co-supervisors Dr. Liming Zhu and Dr. Helen Paik for guiding me through my studies and providing me with valuable feedback and continuous advice.

I am also deeply grateful to Dr. Jim Webber for his commitment and active involvement in my work. His advice on various levels and many lively discussions have been extremely encouraging and have significantly contributed to this work.

In addition, I want to express my gratitude to the following three organisations for offering me their trust and generous support: the National ICT Australia (NICTA), which is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council; the Hasler Foundation, which is a Swiss non-profit organisation that promotes research and training in the field of telecommunications, distributed information systems and related topics; and ABB Switzerland, a multinational company, which provides power and automation technologies and supports young engineers in pursuing further education.

*Thank You All!*





# Contents

<b>Abstract</b>	<b>iii</b>
<b>Originality Statement</b>	<b>v</b>
<b>Copyright Statement</b>	<b>vii</b>
<b>Authenticity Statement</b>	<b>ix</b>
<b>Publications</b>	<b>xi</b>
<b>Acknowledgements</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xx</b>
<b>List of Tables</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Thesis Focus and Contribution . . . . .	2
1.2 Thesis Structure . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 RPC and Distributed Object Technologies . . . . .	5
2.1.1 Shortcomings . . . . .	6
2.2 Messaging . . . . .	8
2.2.1 Asynchronous Calls . . . . .	9
2.2.2 Message-Oriented Middleware (MOM) . . . . .	10
2.2.3 Message Brokers . . . . .	10
2.3 Service-Oriented Architecture (SOA) . . . . .	12
2.3.1 The Four Tenets of Service Orientation . . . . .	13
2.4 REpresentational State Transfer (REST) . . . . .	14
2.4.1 Rationale . . . . .	15
2.4.2 Uniform Interfaces . . . . .	15
2.4.3 Messages and Data Representations . . . . .	16
2.4.4 Resources or Services? . . . . .	16
2.5 MESsage Transfer (MEST) . . . . .	17
2.5.1 One-Way Messages and Interaction Protocols . . . . .	17
2.5.2 Logical Operation . . . . .	17
2.6 Web Services . . . . .	17
2.6.1 Defining Web Services . . . . .	18

2.6.2	Do we need Web Services?	20
2.6.3	Web Services Architecture	21
2.6.4	SOAP	23
2.6.5	Interaction Styles	25
2.6.6	Web Services Description Language (WSDL)	28
2.7	Interaction Protocols	29
2.7.1	Protocol Languages	31
2.8	Service Composition and Workflows	33
2.8.1	Relationship with Interaction Protocols	33
2.8.2	Business Process Execution Language (BPEL)	34
2.9	SOAP Service Description Language (SSDL)	35
2.9.1	Structure	35
2.9.2	Messages	36
2.9.3	Protocols	37
2.9.4	Claimed Benefits	38
2.9.5	Tool Support	39
2.10	Summary	39
<b>3</b>	<b>The Programming Model</b>	<b>41</b>
3.1	Development Life Cycle	41
3.2	Defining SSDL Contracts Using Metadata	42
3.2.1	Defining Messages	43
3.2.2	Defining Messaging Behaviour	45
3.3	Implementing a Service	47
3.4	Client API	47
3.4.1	Initiating a Conversation	48
3.4.2	Replying to a Previous Message	48
3.5	Configuring a Service	49
3.6	Client-Server Symmetry	50
3.6.1	XML Firewalls	52
3.7	Service Deployment	52
<b>4</b>	<b>The Runtime Environment</b>	<b>55</b>
4.1	Introduction	55
4.2	Runtime Components	56
4.2.1	Processing Flow	57
4.3	Message Correlation & State Maintenance	57
4.3.1	Leveraging WS-Addressing	59
4.3.2	Session Context Management	59
4.3.3	Session Context Lifetime and Maintenance	61
4.3.4	Synchronisation of Session Context Access	62
4.4	Structural Validation	63
4.5	Protocol Validation	63
4.5.1	Dynamic State Pattern	65
4.5.2	Usage	66
4.5.3	Implementation	67
4.5.4	Decorating the State Machine	68
4.5.5	Non-Determinism and Ambiguity	68
4.5.6	Persisting State and Cloning	70
4.6	Message Dispatching	71

4.6.1	State-Based Method Selection . . . . .	71
4.6.2	Asynchronous Operation Invocation . . . . .	72
4.7	Metadata Generation and Exposure . . . . .	75
4.8	Building the Runtime . . . . .	75
4.8.1	Building the Internal Service Model . . . . .	77
4.8.2	Building the Protocol State Machine . . . . .	77
<b>5</b>	<b>Case Study</b>	<b>79</b>
5.1	Motivation and Background . . . . .	79
5.1.1	Lending Industry XML Initiative (LIXI) . . . . .	80
5.1.2	Property Valuation Process . . . . .	80
5.2	Approach . . . . .	81
5.3	Case One: Property Valuation . . . . .	82
5.3.1	Focus . . . . .	82
5.3.2	Case Description . . . . .	82
5.3.3	Protocol Translation . . . . .	84
5.3.4	Development Process . . . . .	86
5.3.5	Discussion . . . . .	90
5.4	Case Two: Intermediary . . . . .	93
5.4.1	Focus . . . . .	93
5.4.2	Case Description . . . . .	93
5.4.3	Sequencing Constraints . . . . .	95
5.4.4	Discussion . . . . .	99
5.4.5	Comparison with WSDL & BPEL . . . . .	101
5.4.6	Discussion . . . . .	105
5.5	Conclusion . . . . .	107
<b>6</b>	<b>Discussion</b>	<b>109</b>
6.1	State Machine Expression Power . . . . .	109
6.2	State Maintenance using WS-Addressing . . . . .	110
6.2.1	Uniqueness of MessageID . . . . .	110
6.2.2	Multi-Party Conversations . . . . .	111
6.3	Defining Protocols using Attributes . . . . .	112
6.4	SOAP Action Semantics . . . . .	112
6.5	WSDL Operations . . . . .	112
6.6	Asynchronous Interaction Model . . . . .	113
6.7	Assuming SOAP Only . . . . .	113
6.8	Content-Based Message Dispatching . . . . .	114
6.9	SSDL Faults . . . . .	115
6.9.1	Fault XML Schema . . . . .	116
<b>7</b>	<b>Conclusions</b>	<b>119</b>
7.1	Summary of Work Undertaken . . . . .	120
7.2	Summary of Findings . . . . .	121
7.3	Directions for Future Work . . . . .	122
	<b>Bibliography</b>	<b>123</b>
	<b>Appendices</b>	<b>133</b>



# List of Figures

2.1	Synchronous and Asynchronous Interaction Semantics . . . . .	9
2.2	Messaging System . . . . .	10
2.3	Message Broker . . . . .	11
2.4	Basic SOA . . . . .	13
2.5	MEST Architecture . . . . .	18
2.6	Web Service as XML Processor . . . . .	19
2.7	First Generation Web Services Architecture . . . . .	21
2.8	WS-* Architecture . . . . .	22
2.9	Web Service Engagement . . . . .	24
2.10	Simple SOAP Message . . . . .	24
2.11	Difference: Protocols – Workflows . . . . .	34
2.12	SSDL Structure . . . . .	36
2.13	SSDL Message Definition . . . . .	37
2.14	MEP Protocol . . . . .	38
2.15	SC Protocol . . . . .	38
3.1	Typical SSDL Service Implementation . . . . .	42
3.2	Inferring SSDL Contract from Service Implementation . . . . .	43
3.3	SSDL Contract . . . . .	51
3.4	XML Firewall . . . . .	52
4.1	Soya Middleware Layer . . . . .	56
4.2	Soya System . . . . .	56
4.3	Soya Runtime Architecture . . . . .	58
4.4	Scalability . . . . .	59
4.5	Session Context Creation and Access . . . . .	60
4.6	Session Context Synchronisation . . . . .	62
4.7	MEP State Machines . . . . .	64
4.8	Combined MEP State Machine . . . . .	65
4.9	TCP Connection . . . . .	65
4.10	Dynamic State Pattern . . . . .	66
4.11	IExtensibleObject<T> Pattern . . . . .	68
4.12	State Machine Decoration . . . . .	69
4.13	Deterministic, Ambiguous Service . . . . .	69
4.14	Unambiguous Service . . . . .	70
4.15	Non-Deterministic, Unambiguous Service . . . . .	70
4.16	State-Based Method Selection . . . . .	72
4.17	Asynchronous Invocation . . . . .	74
4.18	Metadata Retrieval . . . . .	75

4.19	Building the Runtime	76
4.20	Protocol (In)Dependent Model Creation	77
4.21	Building the Protocol State Machine	78
5.1	Valuation Lifecycle	81
5.2	Valuation Conversation	83
5.3	Valuation Protocol	84
5.4	MEP Protocol	85
5.5	Protocol Comparison: Business – MEP	85
5.6	Requestor Contract in C#	88
5.7	Interactive Console	90
5.8	Valuation Conversation including Intermediary	94
5.9	Intermediary Protocol	95
5.10	SC Protocol using <i>Multiple Semantics</i>	96
5.11	Iteration in SC Protocol	97
5.12	SC Protocol: Intermediary – Valuation Firm	98
5.13	SC Protocol: Intermediary	98
5.14	Intermediary Contract in C#	102
5.15	BPEL Partner Links	102

# List of Tables

2.1	Formal Models Comparison . . . . .	32
5.1	LIXI Terminology . . . . .	80
5.2	Message Exchange Patterns . . . . .	84
5.3	Number of Lines of Code Comparison . . . . .	107





# Chapter 1

## Introduction

“ I find that the harder I work, the more luck I seem to have. ”

— *Thomas Jefferson*

Complex software systems are often constructed according to simpler and more comprehensible abstract models and guidelines. One architectural style that has garnered recent attention is service-oriented architecture (SOA) [1]. SOA is a named set of coordinated architectural constraints that can guide software developers during the creation of large-scale distributed software systems. Fundamentally, SOA follows the common practice of decomposing a complex problem into smaller, independent and thus more manageable abstractions. In an SOA, these abstractions are autonomous units of logic called services. Services use messages to communicate and exchange structured information among each other while descriptions capture the form and patterns of how these interactions can take place. Together, services, messages, and descriptions form the three main components of a basic SOA [2].

Although service-orientation per se is neither a new nor a technology-dependent paradigm, the advent and emergence of Web Services has reinvigorated interest in the approach [3]. Indeed, Web Services offer a suitable technology platform for realising service-oriented applications. By defining a set of standards and models, they enable the integration of independent, heterogeneous components and make it possible to create a new class of interesting distributed applications. Simply using Web Services, however, does not automatically lead to service-oriented design [4]. As a matter of fact, Web Services technology can likewise be used to create applications that adhere to other architectural principles which are less suitable for building Internet-scale applications. The design of WSDL [5], for example, is procedure call-centric and can constrain Web Services practitioners from adopting a more message-oriented mindset. Its focus on

operations as the primary abstraction for communication has resulted in a large number of tools (e.g. [6, 7]), which generate code that shields network communication in order to make remote service invocations look like local method calls. Web Service applications built in such ways are consequently not architecturally different from RPC systems [8]. In other words, these systems are often tightly coupled, brittle at distribution boundaries and limited in scalability [9].

In contrast, then, the SOAP Service Description Language (SSDL) [10] is an XML-based language that describes Web Services in a message-oriented way. In its crudest form, it can be seen as a direct language replacement for WSDL. Yet unlike the latter, SSDL focuses on one-way SOAP messages [11], not operations, as the building blocks for creating service-oriented applications. Moreover, it provides mechanisms, known as protocol frameworks, which can be used to combine messages into interaction protocols that define their expected relative ordering and thus enable protocol-based reasoning. In general, the message-centric concepts underlying SSDL provide a more natural fit with service-oriented design principles than the operation-centric design of languages like WSDL. Intuitively, this suggests that SSDL inherently fosters the creation of loosely coupled and scalable applications.

## 1.1 Thesis Focus and Contribution

Currently, virtually no data exists on experiences in implementing tool support for SSDL or using the same as part of Web Services-based SOAs. Consequently, this raises a number of questions that this study seeks to answer. At one end of the spectrum, it is unclear if the approach proposed by SSDL is truly practicable. In other words, can sensible programming abstractions and runtime support be provided that aid developers in building service-oriented applications without adding unrealistic development burdens. At the other end, we can only speculate if SSDL indeed has significant benefits compared to the incumbent approaches that exist for describing Web Services (e.g. in terms of fostering service-orientation, reducing complexity, increasing semantic expressiveness, etc.).

Unfortunately, SSDL has not been widely adopted by the community. We speculate that the main inhibitors to SSDL being adopted more widely is the general lack of engagement by leading Web Services technology companies such as IBM, Microsoft, Oracle, Bea etc. and the absence of tool support for creating and executing SSDL-based Web Services. There are neither tools nor programming abstractions available that could assist developers in modelling and implementing SSDL-based Web Services. Furthermore, no SSDL-aware runtime platform exists, which is needed to realise the benefits of SSDL's machine-

processable message and protocol descriptions.

To that end we have developed Soya [12], a programming model and runtime environment for creating and executing SSDL-based Web Services. The development of Soya presented us with a number of non-trivial research and engineering challenges, which we discuss in this thesis. In summary, Soya proposes to address problems in the following areas of SSDL-based Web Service development:

- *programming abstractions*: the efficient creation of SSDL Web Services necessitates straightforward programming abstractions that foster the language's underlying message-oriented practices. Most developers are familiar with synchronous method call semantics only. SSDL's interaction model, however, is asynchronous. The programming model must therefore deal with this dilemma by providing abstractions that are easily adoptable, even by developers who are not accustomed to working with asynchronous interactions. In addition, it needs to provide mechanisms for capturing contractual information (e.g. XML schema, message descriptions, protocols, etc.) without adding unrealistic development effort;
- *runtime support*: an SSDL engine should exhibit the same features that are commonly found in contemporary SOAP-processing middleware (e.g. efficient processing of SOAP messages, support of prevalent Web Services standards such as security or reliability, etc.). In addition, it must obviously provide functionality and semantics related to SSDL. This includes message validation in terms of structure and ordering, state maintenance by means of WS-Addressing [13], protocol-based message dispatching and so forth. Finally, the runtime architecture must take SSDL's extensible protocol framework mechanism into account and ideally be independent of the programming abstraction in use.

The main contributions of this work to the field of software research are the design and implementation of a programming model and runtime environment for creating and executing SSDL-based Web Services, respectively. By providing the community with this knowledge and infrastructure, we hope to promote message-oriented Web Services design and motivate further research in this direction. In order to validate the usability of Soya's programming abstractions and the proper functioning of its runtime, we apply it to a case study in connection with the Australian lending industry. In addition to demonstrating its practicability, this research is important because it provides an initial set of empirical results that differentiate Soya and SSDL from the prevalent approaches.

## 1.2 Thesis Structure

Chapter 2 provides an extensive and critical overview of relevant literature to articulate the problem field to which this work applies. We follow a chronological perspective of different architectural styles and practices for creating distributed software applications. In particular, we emphasise Web Services technology as a platform for building service-oriented applications. We also introduce concepts, models and languages from the domain of protocol and workflow research. We conclude the chapter with a presentation and discussion of SSDL. In Chapters 3 and 4 we give an in-depth presentation of Soya's programming model and runtime environment, respectively. These constitute the original empirical work of the thesis. First, we elaborate on how Soya helps developers to model SSDL-based Web Services. Then, we follow with a detailed explanation of how the various runtime components work. Chapter 5 presents a case study in the creation of a service-oriented system in connection with the Australian lending industry. We show how the system can be realised with the aid of Soya and SSDL. Additionally, we re-model the application using incumbent approaches and compare the results. In Chapter 6 we discuss remaining open issues, which had not been addressed in previous chapters. Chapter 7 concludes the thesis by summarising the findings and contributions of our work and indicating some directions for future research.

## Chapter 2

# Background

“ The task is not so much to see what no one yet has seen,  
but to think what nobody yet has thought,  
about that which everybody sees. ”

— *Arthur Schopenhauer*

Over the years, architectural styles and best practices for creating software have evolved and changed many times. This has required software engineers to continuously adapt both technology and mindset. At the same time, the large number of existing paradigms, principles, tools and so forth can also be confusing or even deceptive. In this chapter, we discuss some relevant architectural styles and technologies and their implications in terms of creating Web Service applications. Although this comprises a review of existing relevant literature, it holds conceptual originality. Specifically, it produces the framework and rationale on which we built Soya.

### 2.1 RPC and Distributed Object Technologies

In 1976, James E. White from the Stanford Research Institute published an RFC<sup>1</sup> containing details about what he called the “procedure call model” [8]. The ideas described in his original RFC 707 became later known as remote procedure calls (RPC). White’s intention was to abstract and hide network complexity in order to provide an environment that looked familiar to software developers who were used to writing non-distributed applications. In the late 1970s, RPC replaced client-server database connection designs. In the 1990s, RPC technologies such as CORBA [14] and DCOM [15] emerged, which provided a complete distributed system architecture. Upon the arrival of the World

---

<sup>1</sup>RFC stands for “Request for Comments”

Wide Web in the mid-to-late 1990s, Internet technology was incorporated into distributed architectures and custom client components were often replaced with browsers. In these architectures, proprietary RPC protocols were consequently replaced by HTTP [16]. Some years after, Java followed with its object equivalent of RPC called Remote Method Invocation (RMI) [17] and later Enterprise Java Beans (EJB) technology [18], which likewise provided significant additions to the original RPC. Internally, these components communicated via proprietary APIs while still using one or the other form of RPC for communicating across program boundaries.

All these technologies promised to make developers' lives easier by hiding tedious network communication code and making remote objects look like local objects, in accordance with White's original ideas. This was achieved by generating intermediary proxy and stub classes that shielded and handled the network communication. During proxy generation, the expected interactions between components were taken into account by statically embedding references among each other into the proxy code. Even though many software projects have successfully applied RPC-based technologies over the last two decades, these technologies have a range of shortcomings. As a matter of fact, it has been known for several years that RPC is not ideal for Internet-wide computing (e.g. [19, 20, 21, 22]).

## 2.1.1 Shortcomings

### 2.1.1.1 Programming Model

Back in 1994, Waldo et al. suggested that “objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space”<sup>2</sup>. According to the authors, the vision of unified objects cannot be successfully applied to large distributed systems because it tries to conceal fundamental dissimilarities between local and remote communication. This includes differences in network latency, memory address space, concurrency and partial failure scenarios. Consequently, Waldo et al. suggested that calls across a network must be clearly distinguished and treated differently from local calls within the same address space [9]. Yet RPC programming abstractions conceal these dissimilarities, and do thus not allow developers to differentiate between them.

---

<sup>2</sup>Waldo et al., “A note on distributed computing”, 1994

### 2.1.1.2 Interface Complexity

RPC-based applications tend to be tightly coupled and thus limited in scalability. Their components often expose a large number of complex, fine-grained and diverse interfaces that mirror implemented procedures or object structures. Also, components are connected in a point-to-point fashion, resulting in many dependencies among each other. As the number of application components increases, the number of possible connections grows exponentially. As a result, keeping track of the semantics of each component's interface becomes increasingly difficult as Vinoski notes in [23]. In practice, making changes to interfaces can be virtually impossible, as it may imply updating a large number of dependent components; all the more, if components are scattered across organisational or trust boundaries. Fowler remarks in [21] that these problems are often not significant within single n-tier applications but become so when different independent applications are integrated.

### 2.1.1.3 Synchronous Communication

Despite its many advantages, synchronous communication has also a number of drawbacks, in particular when used in highly distributed environments. Synchronous communication is not well suited for long-running activities, because clients keep connections open while waiting for results. As a consequence, servers need to maintain large numbers of concurrent connections, which affects both their performance and complexity. Further, synchronous calls make components dependent on the availability of others. In a distributed environment, such as the Internet, where it is a reality that network links fail or applications become unavailable, this can disrupt the entire application.

### 2.1.1.4 Enterprise Application Integration (EAI)

RPC couples applications further in terms of middleware technology that needs to be the same on both ends of communicating components. Still, we mentioned above that many applications have been built successfully with RPC technologies. These achievements, however, were mostly limited to environments characterised by platform homogeneity and predictable latencies, such as the corporate Internet [20]. Yet in order to create truly large-scale applications that span organisational, trust and geographical boundaries, where networks are latent, message loss is common, transmission speed varies and so forth, RPC is not an apt technology. Hohpe and Woolf sum it up accurately in [21] by saying that while RPC could be used to distribute n-tier applications, it is not suitable for integrating independent applications.

## 2.2 Messaging

Messaging is an architectural style in which independent applications communicate with each other remotely by exchanging structured data called *messages*. Based on the message content, the receiving application performs appropriate actions and potentially sends new messages back. Because the applications interact in an asynchronous manner, they are inherently loosely-coupled<sup>3</sup>, more dynamic and more reliable.

In message-oriented systems, the lines between client and server (or sender and receiver, producer and consumer, etc.) are blurred. Both sides can freely send and receive messages in either direction and their roles can thus change during the course of a single conversation. The distinction is purely conceptual and only makes sense in connection with the semantics of the message exchange.

Besides enabling remote communication, messaging systems can overcome platform and language issues by providing universal communication interfaces for different technologies. Additionally, asynchronous communication can improve performance because the sender of a message does not need to wait for the receiver to process it. Instead, it can perform other work and therefore increase its throughput. Finally, messaging interactions are more reliable than standard RPC. If, for example, the network link or the receiver is not working properly, a messaging system tries to resend the message until it succeeds. Hohpe and Woolf give a more extensive list and a description of the advantages provided by messaging in [21].

As a result, messaging has been the favourite approach for integrating autonomous and heterogeneous applications into large-scale enterprise systems for a number of years [3, 24]. In fact, messaging solutions have been successfully applied in a wide range of domains including the financial sector, which is known for its high performance and reliability requirements. The Society for Worldwide Interbank Financial Telecommunications (SWIFT), for example, processed more than 2 billion messages per year during 2003 [25].

Although asynchronous messaging architectures are powerful, they also introduce new challenges. One of them is the significant differences in the design approaches that are associated with asynchronism. Because developers are normally familiar with synchronous method-call semantics, the idioms and peculiarities of asynchronous communication requires working with a more complex event-driven programming model. Further, there are issues relating to message ordering and synchronisation. Again, an extensive description of the challenges introduced by asynchronous messaging is given by Hohpe and Woolf in [21].

---

<sup>3</sup>The fewer assumptions two applications need to make about each other in order to exchange information, the less coupled they are.



### 2.2.1 Asynchronous Calls

Interactions in software systems are often characterised in terms of blocking (synchronous) or non-blocking (asynchronous) behaviour. In a synchronous interaction, a process calling a sub-process blocks and must wait for the response from the sub-process before it can proceed. Synchronous calls also block when the called process executes in an independent, possibly remote environment. In an asynchronous interaction, conversely, the calling process does not need to wait for a response and can proceed while the sub-process executes. This difference in semantics is illustrated in Figure 2.1.

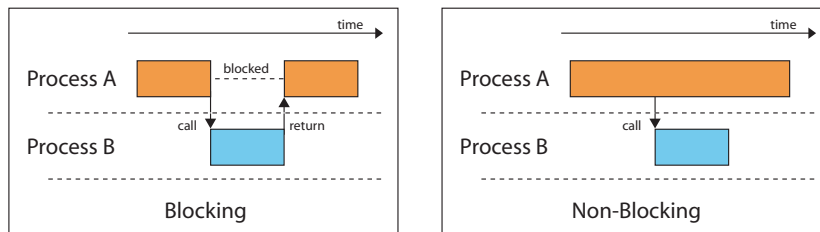


Figure 2.1: Comparison between synchronous and asynchronous interaction semantics. Adapted from [21].

Asynchronous communication decouples processes and can improve performance. However, it is generally more complex because processes run concurrently and their exact execution sequence can no longer be determined. Additionally, results of an asynchronous call must be communicated to the calling process somehow (e.g. via a callback or shared variables) and then be associated correctly with the context in which the call was made.

Undeniably, synchronous behaviour has advantages in terms of simplicity and makes perfect sense in many situations, such as invoking operations on local objects. However, we have reasoned earlier that communication across a network is fundamentally different from interactions among objects in the same address space. Abstracting the former into the simple semantics of local method calls can thus be deceptive. Synchronous communication among distributed applications couples them in a tight manner. As a result, applications integrated in this fashion have more complex dependencies and are harder to maintain. In contrast, applications that communicate by means of asynchronous message exchange can operate more independently of each other. Consequently, the integrated application components will be more loosely coupled.

### 2.2.2 Message-Oriented Middleware (MOM)

The systems that provide messaging capabilities are known as *Message-Oriented Middleware* (MOM) (e.g. [26, 27, 28]). These systems coordinate and manage the sending and receiving of messages between applications, which are connected using virtual pipes called *message channels*.

MOM systems normally connect applications in a *point-to-point* fashion, where a sender places a message into a queue. Normally this is done in a non-blocking fashion. Then, the middleware moves the message from the sender to the receiver, which extracts it from the other end of the queue and starts processing it. If the receiver is not available, the system retries the delivery until it succeeds. The receiver can obtain the message in either a blocking or a non-blocking fashion; by either waiting for a new message or providing a callback function that is invoked when a new message arrives. This process is illustrated in Figure 2.2.

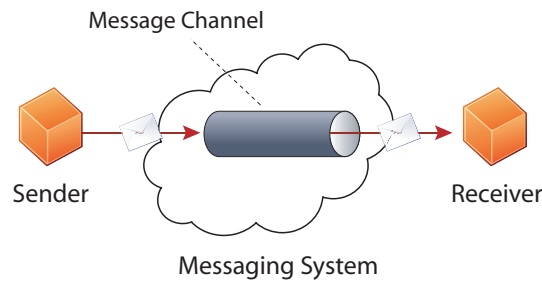


Figure 2.2: A messaging system that connects two components through a message channel. Adapted from [21].

Unfortunately, the point-to-point model requires communicating applications to be directly connected with each other. This design is thus relatively static and inflexible. Further, the number of connections (i.e. queues) grows exponentially with the number of integrated applications. Because of its many connections between applications, this kind of design is often referred to as *spaghetti architecture*.

### 2.2.3 Message Brokers

Message brokers address the limitations of basic MOM systems by extending them with centralised message routing, filtering and processing capabilities. A broker thus basically acts as a *Mediator* [29] between the integrated applications. For example, the *publish-subscribe* messaging model, which is supported by virtually every message broker, enables *many-to-many* communication over one channel. Essentially, the publish-subscribe paradigm has the same underlying

principle that is described by the *Observer* pattern in [29]. Subscriber applications can subscribe to *topics*, which are basically logically named queues. When a new message is published to a topic, the middleware notifies each subscribed application by sending it a copy of the message.

Additionally, message brokers can centralise message content transformations. As a result, heterogeneous applications can send messages in their own format, which are then transformed by the broker into a unified format.

By introducing a message broker, the complexity and number of communication interfaces in the endpoints is reduced to a great extent. Figure 2.3 shows an architecture with a message broker that is often referred to as a *hub and spoke* model, because of its similarity to a bicycle wheel.

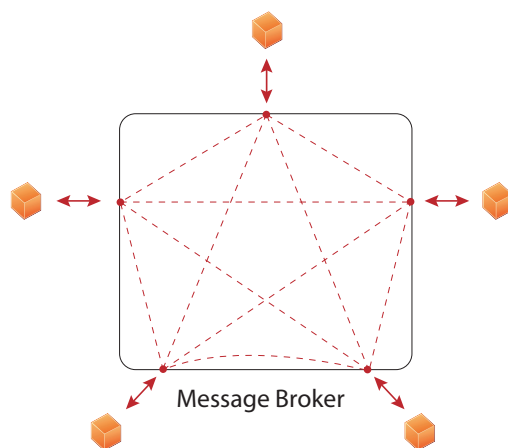


Figure 2.3: Using a message broker reduces complexity between applications at integration points. Adapted from [24].

Although a lot of the complexity is captured in a central place and message broker products normally include powerful tools to describe routing or transformation logic the approach still has a number of drawbacks. First and foremost, the spaghetti architecture still exists inside the broker as visible in Figure 2.3. Second, placing application logic into the broker can make applications more difficult to debug and maintain. Third, brokers are potential performance bottlenecks, because all messages flow through a central point.

Despite their benefits and success in corporate Intranets, MOM systems are not ideal for Internet-wide application integration. On one side, homogeneous middleware platforms are required to connect applications. On the other side, the middleware needs to be placed – as the name suggests – “in the middle” of the integrated applications. When applications exceed organisational boundaries, this often causes practical problems, the solutions to which aren’t only of a technical nature. It is very likely that companies do not have the same kind

of MOM systems and are not able or willing to change them. Further, it raises security issues, as one company needs to access another companies middleware infrastructure.

## 2.3 Service-Oriented Architecture (SOA)

Service-oriented architecture (SOA) [1] is a recent architectural style which guides software architects during the creation of large-scale distributed software systems<sup>4</sup>. Although the fundamentals underlying SOA are not at all new and some even claim that they are as old as trade and the commercial marketplace itself [1], the emergence of Web Services technology has clearly reinvigorated interest in service-oriented principles. Nevertheless, SOA per se is a technology-agnostic paradigm and can potentially also be realised using other implementation strategies. Indeed, Sprott and Wilkes [30] claim that distributed architectures were early attempts to realise SOAs and that the notion of service is an integral part of component thinking.

A lot has been written about SOA but often with a strong inclination towards Web Services technology (e.g. [31, 32, 2, 33]). A more puristic attempt to define a common language and understanding of SOA independent of technology was realised by the Organisation for the Advancement of Structured Standards (OASIS), which defines SOA as follows:

*“Service Oriented Architecture (SOA) is a paradigm for organising and utilising distributed capabilities that may be under the control of different ownership domains.”*<sup>5</sup>

Fundamentally, SOA follows the common practice of decomposing a complex problem into smaller, independent, more manageable abstractions. In an SOA, these independent and autonomous units of logic are called services. Services are higher-level abstractions that provide capabilities which other services can use without knowledge of how they were provided. Services use messages to communicate and exchange structured information among each other while descriptions capture the form and patterns of how these interactions take place. Together, services, messages, and descriptions form the building blocks of a basic SOA [2].

<sup>4</sup>It has been argued that the naming of SOA is unfortunate because discussions relating to it often involve areas such as business design or delivery processes, which exceed the scope of architecture. Some authors have thus suggested to use the term Service Orientation (SO) instead [30]. On account of its widespread use in the field of software architecture and because this thesis focuses on software development, we will nevertheless use the three-letter nomenclature.

<sup>5</sup>OASIS, “Reference model for service oriented architecture v 1.0”, Section 2.1, 2006

For two services to be able to interact, they have to be aware of each other's existence. This requires that a service consumer either knows directly where the service provider is located or that it has some means to locate a suitable service provider that can satisfy its needs [1]. In the latter case, some sort of discovery mechanism (e.g. service registry) is needed to bridge this information gap. Figure 2.4 shows how a service provider publishes its description to a registry that can then be located and used by a service consumer.

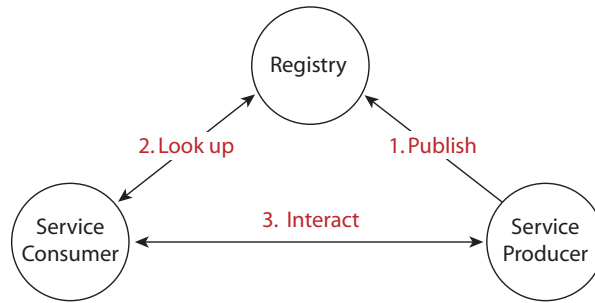


Figure 2.4: A basic SOA. Service producers publish their descriptions in registries, helping service consumers to discover and subsequently interact with them.

### 2.3.1 The Four Tenets of Service Orientation

Although there is no general consensus what an SOA exactly is, many authors (e.g. [34, 35, 36]) agree that a typical SOA reflects the following *four fundamental tenets* proposed by Don Box in [37]:

**Boundaries are explicit.** In an SOA, services communicate through the exchange of messages across service boundaries, which are well-defined and explicit. Services have no knowledge about what is behind a boundary, which keeps service implementations private and decoupled from other services. Because services span separate processes, trust domains or geographical boundaries, each boundary crossing is potentially expensive in terms of processing overhead, performance penalties or complicated failure scenarios. For this reason, inter-service communication must be consciously distinguished from local method invocations. By making boundaries formal and explicit, developers recognise this difference between local and remote communication.

**Services are autonomous.** Services are self-governed and fully control the logic they encapsulate. They are modular building blocks that do not require knowledge of each other's internal workings in order to interact.

As a result, services can evolve independently from each other as long as they do not alter their public contracts. Moreover, as the topology of a service-oriented system is expected to change over time, adding, upgrading or removing services should not disrupt the overall system.

**Services share schemas and contracts, not classes.** Services maintain implementation independence by exposing only schemas and contracts that are expressed in a platform-neutral format. Schemas define the structure of messages a service can receive or send, while contracts determine the mechanics of these interactions. Together, schemas and contracts are shared beyond service boundaries.

In [38], Helland premises that data residing inside services is actually different from data residing outside services in many essential points. He argues that the only way these boundaries can be crossed is by means of messages carrying outside data to the inside or vice versa and that after receiving or before sending messages it lies within a service's responsibility to cope with the necessary transitions. He concludes that different technologies (e.g. SQL and object-oriented languages for inside data and XML [39] for outside data) must be used to allow for the different characteristics that apply to inside and outside data.

**Compatibility is determined based on policy.** Besides using schemas and contracts for agreeing on structural compatibility in terms of messages and exchange sequences, services might have further constraints on the semantics required for communication to take place. Therefore, both requirements and capabilities are expressed in a machine-readable policy description. This separates the description of a service's behaviour from the specification of constraints for accessing it.

Despite ongoing and often nearly religious debates about architectural styles (e.g. REST vs. SOA [40]), it has been widely accepted that SOA provides a flexible and useful approach to manage the complexity that arises when developing large-scale distributed software systems (e.g. [31, 35, 34, 41, 42]). For this reason, we are not investigating SOA further at this point but will resume the topic to some extent later in conjunction with our discussion on Web Services in Section 2.6.

## 2.4 REpresentational State Transfer (REST)

REpresentational State Transfer (REST) is an architectural style for describing distributed hypermedia systems such as the World Wide Web. Indeed, REST

was developed as an abstract model of the Web architecture, or more precisely a model of how the Web should work [43]. The term was coined by Fielding in his doctoral thesis in 2000 [44], just shortly after the first Web Services-related work had started<sup>6</sup>. The fact alone that the largest distributed software system, the World Wide Web, is built in a *RESTful* way, justifies mentioning REST in this thesis. Yet we are particularly interested in REST because many argue that the principles it advocates can and should likewise be applied to Web Service architectures. As a matter of fact, there have been ongoing debates splitting the Web Services community into different camps. One side advocates the creation of Web Services based on REST concepts using mainly HTTP and XML. The other side maintains that Web Services should be implemented using SOA and the WS stack (i.e. SOAP and other WS-\* protocols) [40].

### 2.4.1 Rationale

REST defines a set of architectural constraints that are intended to increase performance and scalability of distributed hypermedia systems. In REST, the world is perceived as a collection of resources which are each named with a unique identifier (e.g. a URI [46]). Interactions occur between clients and servers in a pull-based style and involve exchanging representations of resources at a given state (i.e. *state transfer*). In the case of the World Wide Web, a client could, for example, send a HTTP GET request to a server, which in turn sends back a representation of the resource expressed in HTML. Of course, the resource could also be represented as XML, as a picture, as plain text and so forth. Every resource is accessed through a *uniform interface* that defines a fixed set of verbs (e.g. GET, PUT, UPDATE, DELETE), which allow clients to interact with it. Other REST constraints require that servers are *stateless*, content is *cacheable*, components can be *layered* and so forth.

### 2.4.2 Uniform Interfaces

RESTful applications work with uniform interfaces. In contrast, RPC-oriented systems, including most current Web Service architectures, tend to have specialised verbs for each interface or service, respectively. As we have seen in Section 2.1.1.2, complex interfaces have been a major obstacle for creating scalable distributed applications in the past. In REST this issue does not exist because all interfaces are described with a fixed number of verbs. As a result, RESTful applications must focus more on the exchanged messages. Not surprisingly, REST promotes the notion of self-describing messages [44].

---

<sup>6</sup>Box et al. released SOAP 1.0 in November 1999 [45].

Although most of today's Web Services expose specialised and diverse operations, there are also some Web Service practitioners that advocate a more RESTful approach to Web Service interfaces. The MEST architectural style, described in the next section, for example, constrains Web Services to only one logical operation, which effectively eliminates interface complexity.

### 2.4.3 Messages and Data Representations

Despite the fact that interfaces in RESTful architectures are uniform, the data which is exchanged between them remains variable. Messages can contain data in different standardised formats that are specified within the message itself. In HTTP, for example, the *content-type* header can be used to inform the client in what format the data inside the message is represented. Since many formats are standardised and supported by most web browsers, they can easily be displayed or processed.

### 2.4.4 Resources or Services?

Clearly, Web Services have been successfully designed using either a RESTful or a service-oriented approach and it is, in fact, unclear if one approach is generally better than the other. Vinoski discusses differences between the two schools of thought to some extent in [40], but without coming to a decision that would clearly favour one over the other. Zur Muehlen et al. did a comparison in the context of choreography [47]. Although they used a superseded version of SOAP and chose an RPC interaction model (see Section 2.6.5.1), their paper similarly did not seem to reflect a distinct inclination towards either approach.

*Amazon.com*, for example, provides both REST and SOAP interfaces to access their Web Services [48]. Remarkably, 85% of developers in this example are apparently using the REST interface, whereas only 15% are interacting with the service using SOAP [49]. The high percentage of REST interface users in Amazon's statistics, however, might be due to the fact that browser-based clients are a fairly natural way in which to interact with their Web Services.

The fact that the World Wide Web works is a solid argument that REST principles are sound for creating at least certain types of distributed systems. Yet we are not aware of the existence of RESTful Web Service applications that implement some of the more advanced features Web Services can provide, such as transferring messages via intermediary nodes and different transport protocols. Also, Web Service applications are normally not implemented as hypermedia systems. Time, more research and further practical work will tell if one of the two approaches is indeed more suitable.



## 2.5 MESsage Transfer (MEST)

MESsage Transfer (MEST) is an architectural style that views distributed applications only in terms of messages that are transferred between services. The original authors of MEST claim that it “is for service-orientation and Web Services what REST is for resource-orientation and the Web” [50]. Despite the pun, MEST is actually not very similar to REST. In terms of commonalities, both styles particularly advocate uniform component interfaces and self-describing messages.

### 2.5.1 One-Way Messages and Interaction Protocols

In MEST, interactions between services are purely modelled as one-way messages that are exchanged among services in a loose and asynchronous manner. Using one-way messages as the fundamental building blocks, they can be combined into simple patterns and sophisticated interaction protocols that describe the interaction behaviour of applications (see Section 2.7). MEST perceives messages as independent, self-contained units of information that do not convey details of underlying APIs or transport protocols. Accordingly, it expects applications to reason and act solely based on message’s content.

### 2.5.2 Logical Operation

Recognising that many developers are familiar with the semantics of operation abstractions, MEST provides the concept of a logical asynchronous operation called `ProcessMessage()`. However, this does not mean that services actually implement this method. The notion is purely conceptual and should be perceived as a way to help convey the basic concepts underlying service communication. In fact, the semantics associated with executing this method is equivalent to the transfer of a message to a service combined with the implicit request to process it. Of course, the realisation of the message transfer changes with respect to the underlying transport level (e.g. HTTP POST, FTP PUT, TCP send/recv, SMTP DATA). Yet this does not affect the higher-level abstractions used in MEST. Figure 2.5 illustrates this view graphically.

## 2.6 Web Services

The focus of this thesis is to investigate SSDL – a language for describing Web Services. Consequently, the latter form the conceptual framework to which this research applies. A lot has been written about Web Service concepts and technology in the past few years (e.g. [51, 52, 3, 53, 54, 2]). Rather than giving

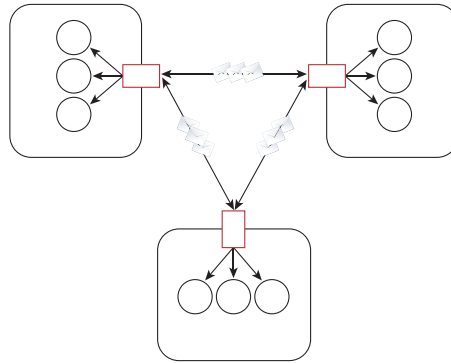


Figure 2.5: Services communicate with each other in a loose and asynchronous manner by exchanging one-way messages. This is abstracted by invoking a single logical operation that is uniformly present on all services (rectangles).

an exhaustive overview of Web Services, we focus only on the aspects most relevant to our work.

### 2.6.1 Defining Web Services

At the end of 2004, the World Wide Web Consortium’s (W3C) Web Services Architecture working group (WSAWG) produced a note on Web Services architecture [4]. The document identifies and abstracts characteristics common to most Web Service applications and explains their relationships among each other in general terms. Moreover, it provides a definition of what a Web Service is and a common vocabulary about Web Service concepts.

Rather than using the W3C’s definition of Web Service<sup>7</sup>, however, we will use our own definition, as it reflects our personal view more aptly:

*“A Web Service is a software system designed to support interoperable machine-to-machine interaction. This is achieved through exchanging messages over an arbitrary number of network and application protocols. Normally, a Web Service exposes a machine-processable contract that captures the mechanics of how other systems can interoperate with it.”*

#### 2.6.1.1 Web Services are XML Processors

In [20], Vogels proposes that Web Services can be viewed as XML processors that send and receive XML documents over a combination of transport and application protocols. This view is illustrated in Figure 2.6. Upon receiving a message, the Web Service processes it, executes some application logic based

<sup>7</sup>W3C, “Web services architecture”, Section 1.4, 2004

on the message content and possibly constructs a new message, which it drops back onto the network.

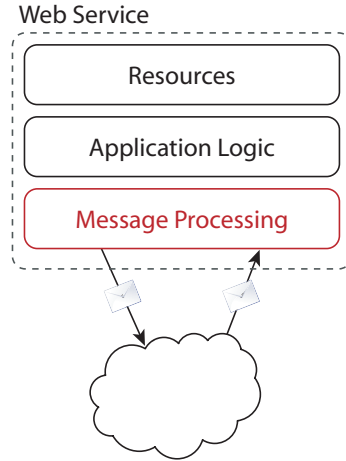


Figure 2.6: Web Service seen as an XML message processors. Adapted from [34].

In fact, this behaviour is quite similar to how businesses or people communicate in the real world. Imagine a company  $C_1$  that wants to order some items from another company  $C_2$ . To do so,  $C_1$  fills out an order form (i.e. message) with the order details (i.e. message content) and posts, faxes or emails (i.e. transport protocol) it to  $C_2$ .  $C_2$  in turn receives and reads the message (i.e. message processing) and forwards the order request to an internal department in order to deal with it (i.e. application logic). Unfortunately, the department realises that it is currently out of stock and thus writes a letter back to  $C_1$ , saying that it is currently unable to deliver the requested items.  $C_1$  receives the letter, reads it and so forth.

### 2.6.1.2 Web Services are Interoperability Standards

Note that in the above example, there are no notions like *operations*, *invocations*, *interfaces*, etc. All the actions are performed solely based on the information contained in the message. Of course, there are some implied semantics that correspond to the ones above and we need to make sure, for example, that the other party understands our request. Writing a letter in Japanese to an Italian company might not lead to the expected result. Likewise, sending an order for semiconductor parts to a clothing warehouse would probably not yield the desired result. In a similar way, Web Services need to ensure that exchanged messages are mutually understood. For this reason, Web Services technology defines open standards that enable different heterogeneous applications to interoperate. During the last few years, Web Services have matured into a com-

moditised platform and it is widely believed that they can significantly advance worldwide interoperable distributed computing [20].

### 2.6.1.3 Web Services enable SOA

Although we said earlier that SOA is an implementation-agnostic paradigm, Web Services have received wide acceptance as a technology for realising SOAs. The Web Services framework provides standards and models that are conceptually aligned with SOA. This includes the abstract notion of service in addition to specifications relating to service description and discovery, messaging, service composition, quality of service (QoS) and so forth.

Yet as reasoned earlier and noted by the W3C in [4], using Web Services technology alone does not imply that any given software architecture will magically become service-oriented. Wrapping and exposing application objects as Web Services, for example, is a practice commonly promoted by many toolkits. In [41], Vinoski discusses some of the problems that are associated with this approach, including semantic mismatches, data type mapping and state management issues, limitations in scalability, performance and so forth.

## 2.6.2 Do we need Web Services?

Many technologies have been applied successfully in creating distributed systems and integrating applications. Yet all of them have always posed restrictions on their environment (e.g. low latency networks, homogeneous platforms, etc.) [20, 3]. This prevented them from achieving the ubiquity that the World Wide Web has. Web Services are another step in the evolution of distributed systems engineering and are hoped and believed to enable interoperability and application integration on a global scale.

Conventional middleware such as MOM or RPC (see Section 2.2 and Section 2.1) limits large-scale integration across organisational, trust or geographical boundaries for several reasons. MOM approaches integrate existing applications through a central point of access. This inevitably raises logistical, political, confidentiality and other issues that hinder the practicability of this approach. RPC-style integration efforts, on the other hand, integrate applications directly in a point-to-point fashion. This implies that connected parties individually agree on a common infrastructure, data format, protocol, etc. In practice, however, different parties might want to use diverse protocols, formats and infrastructure (e.g. corporate policy, existing infrastructure, political motivations, etc.). This results in the creation of hard-to-maintain heterogeneous systems in addition to the issues discussed in Section 2.1 and Section 2.2.

Web Services aim to solve these problems by providing open interoperability

standards, which enable application composition and integration irrespective of underlying technology, implementations and platforms. As a result, applications can be created that transcend organisational or geographical boundaries, network protocols and component heterogeneity.

### 2.6.3 Web Services Architecture

#### 2.6.3.1 First Generation Web Services

The Web Service architecture is based on a set of standards and specifications that provide a framework to build interoperable applications on top of existing network protocols. The standards are notably based on XML in order to achieve platform and language independence<sup>8</sup>. According to several sources (e.g. [51, 53, 32]), the early Web Service architecture consisted of three main specifications. They were SOAP [11], the *Web Services Description Language* (WSDL) [5] and the *Universal Description, Discovery, and Integration* (UDDI) directory. SOAP provided the semantics for communication, WSDL defined a vocabulary to describe the capabilities of a Web Service in a machine-processable format and UDDI specified how to publish and discover information about Web Services. Figure 2.7 shows how the three specifications relate. Note that this architecture is congruent with the basic SOA we defined in Section 2.3.

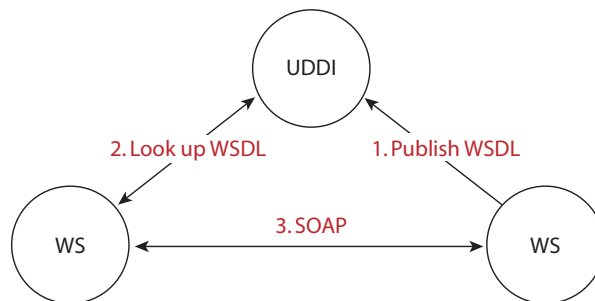


Figure 2.7: First generation Web Services architecture.

#### 2.6.3.2 WS-\*

Today, the relationship between these basic specifications is commonly known as *first generation Web Services architecture* [32]. This architecture had a number of limitations and did not cover more advanced topics such as security, reliable messaging, transactions, service orchestration and so forth [55]. These issues were addressed in subsequent iterations by additional specifications that are

<sup>8</sup>As a matter of fact, many refer to XML as the *lingua franca* of Web Services.

often collectively referred to as *WS-\** [56]. Figure 2.8 depicts the key *WS-\** specifications and their interrelationships in a schematic way.

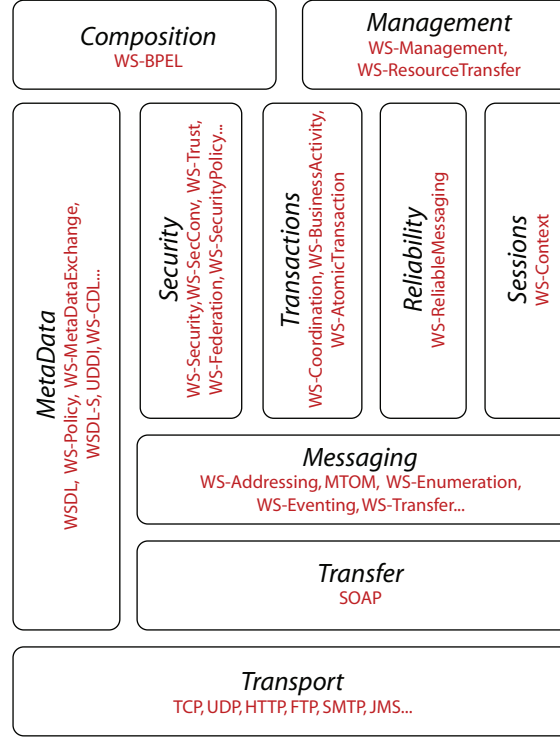


Figure 2.8: Schematic overview of the *WS-\** architecture. The transport layer is not actually part of the *WS-\** specifications, but nevertheless shown for clarity. Adapted from [55, 56]

The number of specifications is large and not all of them have been implemented or have become actual industry standards. Also, because specifications have been written by different authors (i.e. companies), various standards co-exist that address similar problem domains. Vinoski discusses this problem in [57], observing that no specification standardisation has yet occurred and that the vast number of (often overlapping) specifications can be confusing (e.g. [58, 59]). This is aggravated by the fact that no clear common guidelines exist that could help the community in implementing Web Service systems. Although the W3C has produced a note on Web Services architecture [4], it describes only some architectural areas and not how the *WS-\** specifications fit into the overall architecture – mainly because the notes predates most of the *WS-\** specifications.

### 2.6.3.3 Web Service Concepts and Engagement

Still, the W3C note [4] defines characteristics common to most Web Service applications at an abstract level. This includes the identification of important concepts and how they relate to each other. One of them is the general process of how two Web Services engage in a conversation, illustrated in Figure 2.9. We will use this example in the following paragraph to explain some fundamental concepts.

Because Web Services can be both clients and servers in the traditional sense, we normally use these terms more out of convenience than correctness. For that reason, two interacting Web Services are more accurately referred to as *requestors* and *providers*. Interaction is typically facilitated through some agent (e.g. software built on middleware such as Axis [6], XFire [60], WCF [61], etc.). Before interaction occurs, however, the two parties need to agree on the semantics and mechanics of the subsequent interaction. This information is used to configure, build or generate agents that know how to interact with each other. Some of this information is typically captured in machine-processable documents such as WSDL descriptions [5] or WS-Policy files [62]. Other information, however, might not be of machine-processable, explicit or written nature but merely exist in an implicit, oral or human-oriented form. Clearly, it is desirable to have machine-processable Web Services descriptions that are semantically rich enough to capture every aspect of service-level agreement and interoperation in order to enable full automation. Over the last few years, languages and specifications have emerged that aim to describe the mechanics of message exchange more accurately as part of Web Service descriptions (e.g. [63, 64, 10]). Other approaches go even further and try to infuse different types of semantics (e.g. data, functional, non-functional, execution) into Web Services (e.g. [65, 66]). Still, the results of this early research have not yet been widely consolidated into current Web Service development practices. As a matter of fact, most of today's Web Service architectures still rely on information of one sort or the other that cannot be conveyed in machine-processable ways.

### 2.6.4 SOAP

SOAP [11] is the communication protocol that enable Web Services to exchange data in a standardised way<sup>9</sup>. SOAP is transportation agnostic and can thus be used over a number of underlying application or network protocols such as HTTP, TCP, FTP, SMTP, JMS and others.

---

<sup>9</sup>The acronym originally stood for “simple object access protocol”, which was a bit of a misnomer, since it was neither simple nor object-oriented. For this reason, the W3C decided in 2001 to just stick with “SOAP”.

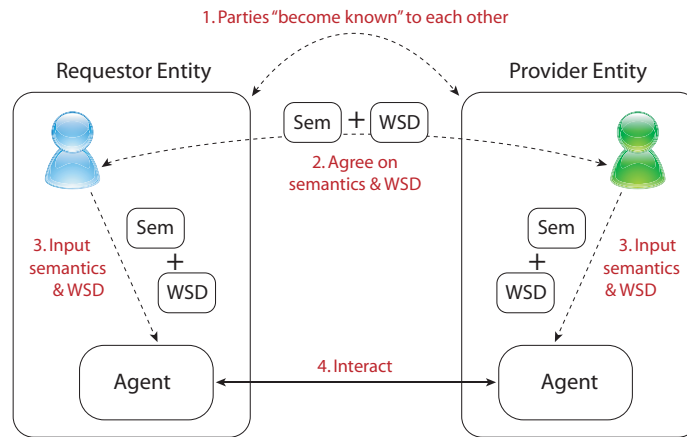


Figure 2.9: The general process of two Web Service engaging in a conversation. Adapted from [4].

The main goals when creating SOAP were to make it interoperable, self-describing, simple and extensible [67]. The first two goals have been accomplished by using XML as SOAP's defining language. Simplicity has been realised by giving SOAP messages a straightforward structure that consists of an envelope containing zero or more header elements followed by one or more body elements. Finally, headers provide the means to extend the SOAP model in a decentralised and modular way. In fact, headers have become an important mechanism that is widely used by WS-\* specifications and SOAP engines to realise functionality such as security, reliable messaging or transactions (e.g. [68, 69, 70]). A simple SOAP message is depicted in Figure 2.10.

---

```

<soap:Envelope xmlns:soap="http://.../soap/envelope/"
  xmlns:a="http://.../ws/2004/08/addressing">
  <soap:Header>
    <a:MessageID>uuid:6B29FC40-CA47-1067-B31D</a:MessageID>
  </soap:Header>
  <soap:Body>
    <EmployeeDetailRequest xmlns="urn:example:schemas">
      <EmployeeNumber>T243261</EmployeeNumber>
      <Detail>basic</Detail>
    </EmployeeDetailRequest>
  </soap:Body>
</soap:Envelope>

```

---

Figure 2.10: A simple SOAP message. The headers define WS-Addressing information while the body contains the application data.



#### 2.6.4.1 WS-Addressing

Until the release of the WS-Addressing standard in 2006 [13], there was no standardised way to embed addressing information in SOAP messages. Essentially, the underlying transport protocol (e.g. HTTP) contained the information as to where the message should be delivered. SOAP messages, however, are inherently transport-independent and can thus potentially travel through many different SOAP intermediaries and an arbitrary number of networks and protocols. WS-Addressing solves this issue by providing mechanisms for embedding addressing information at the message-level.

WS-Addressing also defines some additional standardised SOAP headers that can be used to convey information related to message delivery. In particular, the *MessageID*, *RelatesTo* and *ReplyTo* headers are crucial for delivering and correlating messages that are part of asynchronous interactions. In fact, SSDL layers on top of WS-Addressing for exactly that reason.

#### 2.6.5 Interaction Styles

SOAP was originally designed to unify proprietary RPC communication. Influenced by distributed object technology paradigms, the primary idea was to use SOAP to serialise and deserialise object graphs into an interoperable format (i.e. XML) for transmission over the network.

Since its version 1.1 in 2000, the SOAP protocol has included two distinct *styles* of messages: *RPC-style* and *document style*. The way XML data is represented in XML can be defined by *encoding* rules (i.e. *encoded*) or external schemas (i.e. *literal*). This leads to a total of four style-encoding combinations. Yet in practice, Web Services normally use either *document/literal* or *RPC/encoded*. The latter combination, however, is no longer supported by the WS-I Basic Profile [71]. Unfortunately, these different styles have caused a lot of confusion in the community during the past years [72].

##### 2.6.5.1 RPC-Style

In RPC-style interactions, the body of a request message contains the name of the remote procedure the requesting service wants to invoke, including the parameters that are expected by the procedure. Similarly, a response message encapsulates the output of the remote procedure, formatted in XML. In most cases RPC-style Web Services operate synchronously and use simple *request-response* patterns for communication. As a result, most are implemented using HTTP, because its *request-response* model fits well with RPC.

However, RPC-style Web Services suffer similar drawbacks to those exhib-

ited by RPC-oriented distributed systems, which were discussed in Section 2.1. These kind of applications are normally tightly coupled, limited in scalability and brittle at distribution boundaries [9]. In particular, RPC-style Web Services tend to expose the method signatures of applications objects, which makes it hard for individual service implementations to evolve independently without changing or breaking their public contract. Moreover, the synchronous nature of RPC-style Web Services does not support long running transactions well, keeps connections open during message processing, makes services dependent on each others availability and cannot easily accommodate more complex conversations. Further, interfaces that are based on methods, parameters and return values tend to be rather fine grained. Since a message is sent for each method invocation, this results in “chatty” services with a large number of calls across the network. This level of granularity often does not fit well with business process models, which commonly operate at a higher level of abstraction and use more coarse grained business concepts such as *purchase orders* or *invoices* for communication.

In [67], Loughran and Smith additionally discuss O/X mapping<sup>10</sup> issues that arise when developing Web Services in an RPC fashion. The problem is known as *Object/XML Impedance Match* and roots in the fact that the XML Schema language [73] is richer than the models underlying current object-oriented languages such as Java or C#. In other words, this means that it is not generally possible to serialise method parameters and return values into valid XML or vice versa.

Still, many of the Web Services that exist today are RPC-based and many keep advocating these practices (e.g. [74, 75]). Pasley argues in [76] that the cause of this situation is rooted in current development practices and the widespread use of tools that generate WSDL from local objects and APIs (e.g. [7, 6]). Many developers favour this approach, because they can develop services using familiar programming abstractions (i.e. method semantics) within their preferred environment (i.e. programming language). Pasley notes, however, that these kind of Web Services – in addition to exhibiting the previously discussed undesirable characteristics – tend to reflect the environments in which they were developed (e.g. programming languages, tools, etc.).

#### 2.6.5.2 Document-Style

In document-style interactions, the SOAP message body encapsulates a complete XML document, whose structure is normally defined in an XML schema. Given the primary purpose of Web Services is to enable interoperability be-

<sup>10</sup>This is the process of mapping objects to and from XML, respectively.

tween heterogeneous applications and foster their integration, they often involve business processes and business documents, rather than function-oriented components related to programming languages. Indeed, document-based messages tend to be rather coarse grained and align quite naturally with business models. The documents are typically self-contained and include all the contextual information necessary to process the message. Document-style Web Services therefore tend to consider documents as the main representation and purpose of the interaction. This makes these type of Web Services ideal for asynchronous communication, which allows for creating robust, scalable and loosely coupled distributed applications. Although document-style messages tend to be larger than RPC-style messages, this does not necessarily imply performance penalties, because fewer messages are usually required to accomplish a given task, hence resulting in less network calls. In fact, Maheshwari [77] presented some results that even showed an increase in performance when using Web Services in combination with message queueing.

As a matter of fact, document-style Web Services share many characteristics with messaging systems, which were discussed in Section 2.2. Unlike the latter, however, document-style Web Services do not need a centralised point of access and communicate using open standards. Indeed, centralised middleware and system heterogeneity were two of the main reasons that prevented messaging systems in the past from being more widely used across organisational or geographic boundaries. Banavar et al. identified the need for some “glue technology for loosely integrating distributed systems”<sup>11</sup> back in 1999, before Web Service even existed. They suggested that this technology should extend existing asynchronous messaging (or eventing) paradigms rather than RPC. Carzaniga argued in his doctoral thesis [78] in 1998 and subsequent papers [79] that systems operating in a reactive manner based on asynchronous events do not require integrated components to be tightly coupled. He likewise envisioned that these paradigms could be used for integrating applications on an Internet-scale. Indeed, the underlying principles of these asynchronous messaging and event-based systems are essentially the same as the ones underlying document-based Web Services.

### 2.6.5.3 Document-Style fosters Loose Coupling

Admittedly, it is fair to ask why people still build RPC-style Web Services. The answer to that question is twofold. First, many Web Service applications do not require a high degree of loose coupling, scalability and so forth. Hence, the benefits of using document-style interactions might not outweigh the benefits of

<sup>11</sup>Banavar et al., “A case for message oriented middleware”, 1999

using RPC-style communication. Second, there is clearly less development support available for creating document-centric Web Services in terms of both tools and developer skills. Developers are familiar with synchronous call semantics and traditional programming languages. Also, adopting an approach focusing on documents inevitably requires more familiarity with XML technologies. Yet the proper use of them is what would let developers take full advantage of these standards (e.g. richer descriptions, validation, etc.).

For the reasons given above, it is widely agreed that the asynchronous and document-centric view better coheres with the idea of creating Web Services that function as independent, loosely coupled application components (e.g. [23, 72, 21, 80, 67, 76, 31]). This, however, requires development practices to move away from RPC-style code-generators to tools that foster the creation of document-centric Web Services.

### 2.6.6 Web Services Description Language (WSDL)

Traditionally, Web Services are described in an extensible machine-processable format called the Web Services Description Language (WSDL) [5]. WSDL descriptions can be separated into an abstract and a concrete part. The abstract part is defined in terms of input and output messages that are supported by logical groups of operations. The concrete part binds the abstract definition to a particular network location and implementation style (i.e. binding). The obvious benefit of this separation is that the abstract definitions can be reused and implemented in different manners (e.g. SOAP, REST). Together, the abstract and concrete parts form a public service contract that is exposed to other services.

#### 2.6.6.1 RPC All Over Again

In a similar way to SOAP, it seems that WSDL's design was influenced by RPC-based ideas. Strictly speaking, the fundamental abstraction underlying WSDL is based on message-passing and makes no assumptions about synchrony. Yet it is fair to say that WSDL's focus on operations as the primary abstraction for communication encourages developers to use it as an Interface Description Language (IDL) [81]. This practice is mirrored by the large amount of vendor products that support developers in using WSDL for generating service proxies (e.g. [6, 7]) in order to shield the details involved in accessing remote services. As a result, Web Service applications built in such ways are not architecturally different from RPC systems. Vogels shares this view in [20] and claims that Web Services might solve some interoperability issues, but "provide no magic that can suddenly overcome what excellent protocol architects were unable to

achieve with established RPC systems”<sup>12</sup>.

#### 2.6.6.2 Complexity and Scope

In its current version, the WSDL 2.0 core standard [5] is over 100 pages long and contains many improvements compared to its predecessor. Yet some comments on it have not been favourable, mainly complaining of its unnecessary weight and complexity [82, 83]. Despite its ponderosity, WSDL can only capture a fraction of the mechanics and semantics that are essential for a successful service-level agreement. In fact, WSDL merely addresses connectivity issues such as message format and transport bindings. Yet Meredith and Bjorg [84] argue that much of the substantive complexity of distributed applications does not actually lie in connectivity but interoperation issues, such as order, duration and QoS.

In particular, Meredith and Bjorg elaborate on the necessity of making order constraints explicit in service descriptions (see Section 2.7). WSDL, however, does not provide any support for capturing message ordering constraints, apart from the eight simple message exchange patterns that are defined in the WSDL adjuncts specification [85]. As a result, it is – except for the most trivial cases – not possible to automatically determine the sequence of messages that can be sent and received to and from a service, respectively. Although additional specifications exist that can be layered on top of WSDL (e.g. abstract BPEL [64], WS-CDL [63]), they tend to further increase the complexity of the Web Service description. Indeed, Parastatidis et al. [86] argue that these specifications are more verbose and complex than they would be if more fundamental messaging abstractions had been chosen from the outset. Either way, most agree that in order to increase automation of service-level agreements, the amount of explicit and mechanised service information should be maximised.

## 2.7 Interaction Protocols

Protocols<sup>13</sup> describe how two or more services can interact meaningfully by defining constraints on the relative ordering of exchanged messages. They are public documents that focus solely on interactions among participants and do not reveal details about how a service actually implements them [87]. The ordering conditions and constraints under which messages are exchanged can be observed either from the perspective of an *individual* service or from a *global* viewpoint [88]. In the latter case, this is often referred to as *choreography*.

<sup>12</sup>W. Vogels, “Web services are not distributed objects”, 2003

<sup>13</sup>In the present body of literature, the terms interaction protocol, coordination protocol or abstract process are used interchangeably.

Although the degree of a protocol's complexity can vary significantly, every service implementation inherently holds a protocol that other services must adhere to. In some cases, the protocol might exist only implicitly, defined by the constraints that a service implementation imposes on the structure of valid message exchange sequences. In contrast, it can also exist in an explicit form, which is, of course, ideally congruent with the implicit definition. If the protocol is expressed not by mere implication, its format can range from informal descriptions (e.g. verbal, business documents, etc.) to formal and machine-processable representations (e.g. abstract BPEL [64], WS-CDL [63], SSDL [10], etc.). Because interactions between services are normally constrained and cannot happen independently from each other, explicit protocol descriptions have the evident benefit that they can be used by other services for deriving the correct interaction behaviour. Moreover, machine-processable descriptions can be leveraged in a number of ways by middleware implementations.

Although not yet consolidated in contemporary product development practices, it has been widely accepted in the research community that Web Services are best described in terms of message exchanges (e.g. [89, 87, 90, 84, 63, 50]). Benatallah et al. claim that protocol descriptions can have positive effects on service development, binding and execution and thus considerably simplify the service lifecycle [91]. Although still an active research area, protocols can potentially be used to determine if two services are equal, replaceable or compatible in terms of their protocols [88, 91]. Nevertheless, substituting one service with another is, in general, an inherently hard problem. In fact, ongoing efforts in the context of semantic web research [66] have been addressing this problem for a number of years without reporting major breakthroughs.

As discussed to some extent in [91], protocol-aware middleware can exploit the benefits of protocol descriptions in a number of ways:

**Protocol enactment.** Protocol descriptions enable middleware to establish at runtime whether the sequence of exchanged messages comply with a defined protocol. In case the protocol is violated, the middleware can automatically take appropriate actions such as dropping the message or sending a fault message. This frees the developer from having to implement this kind of exception handling in the application code.

**Conversation-based message dispatching.** Protocols can be used to control the logic for dispatching messages to application code. This accounts for the fact that the semantics of a message type can change during a conversation. For example, while it might be fine to cancel an order after it has been placed, it might not be possible after it has been shipped. Protocol-aware middleware can take advantage of this by dispatching the

same type of message to different local service methods. Again, this removes development burdens from programmers as they do not have to implement this logic in application code.

**Code generation.** A service’s protocol specification can be used to generate client source code that “knows” how to interact with the service.

**Analysis.** A range of analytic activities can be performed by protocol-aware tools. This includes static design-time verification to check, for example, if a client implementation is compatible with a service. Further, it can be used to monitor service evolution and detect potential breaking changes. Finally, it can help to determine service compliance with industry standards such as RosettaNet [92] or Lixi [93].

In order to realise the potential benefits of protocol descriptions, current research has identified a need for models, languages, protocol algebras and tools facilitating protocol-related tasks [91].

### 2.7.1 Protocol Languages

A wide range of languages for describing service protocols currently exists. Unfortunately, there is no unanimous consent or agreement in the broader community as to which one is best fitted for the purpose. At present, the only two standardised initiatives within the Web Service community are abstract BPEL [64] and the Web Services Choreography Description Language (WS-CDL) [63]. As with other Web Service standards, however, concerns have been raised about the verbosity and complexity of these languages [94]. In the research community, a number of other languages and models exist in parallel to these standards. They include approaches based on Petri nets [95, 96], process calculi [97, 64, 63, 98], finite state machines [99], etc.

Van der Aalst et al. argue that abstract BPEL, WS-CDL or Petri nets are not suitable as protocol languages because they are too procedural and imperative. They advocate that given the autonomous nature of services, more declarative approaches are needed and compare the difference between a service and its description to the one between a program and its specification, emphasising that one can specify a program without specifying its implementation [100]. Indeed, van der Aalst and Pesic propose a new, more declarative language called Declarative Service Flow Language (DecSerFlow) [94], which can be used to describe protocols from both an *individual* or a *global* perspective. Other declarative approaches include the SSDL SC [98], CSP [101] or Rules [102] protocol frameworks.

Salaün et al. [89] agree that Web Services and their interactions are best described using protocol languages and suggest that process algebras such as CCS [103],  $\pi$ -calculus [97], CSP [104] and so forth should be used for doing so. Meredith and Bjorg likewise propose the use of process calculi for modelling protocols in [84]. Clearly, such languages provide useful and practical advantages, including formal verification to ensure the absence of deadlocks or race conditions, establishment of compatibility and replaceability between services, etc. Moreover, advanced automated tools (e.g. SPIN [105]) exist that support these kind of tasks. Indeed, many languages claim to be based on one or the other form of a more abstract process algebra (e.g. [64, 63, 98, 101]). In [88], Brogi et al. show how protocols captured in the Web Services Choreography Interface (WSCI)<sup>14</sup> language can be formalised using CCS. In [107], Foster et al. present a model-based approach to verification and a tool that translates protocols written in BPEL or WS-CDL to a process algebra. In [108], Ouyang et al. demonstrate how BPEL can be mapped to Petri net structures. Still, in many cases connections between languages and formal underpinnings have not been rigorously and formally established [109, 110].

Model	Completeness	Compositionality	Parallelism	Resources
<b>Turing M.</b>	✓	✗	✗	✓
<b><math>\lambda</math>-calculus</b>	✓	✓	✗	✗
<b>Petri Nets</b>	✓	✗	✓	✓
<b>CCS</b>	✓	✓	✓	✗
<b><math>\pi</math>-calculus</b>	✓	✓	✓	✓

Table 2.1: Comparison of different formal models according to [111].

Yet other authors including Desai, Chopra and Singh criticise above approaches for either ignoring business semantics altogether or merely specifying them at a low level [112]. Instead they propose formalisms that enrich business protocols semantically and capture their contractual essence and meaning [113] at a higher abstraction level.

As a result of the large number of different approaches and languages, it is not always very clear whether differences among them are of fundamental or merely syntactic nature. Even if a language cannot represent a given structure directly, it is still sometimes possible to transform it into a representable form with equivalent semantics. Although in this case, the expression power of the language is not compromised, transformations are generally not desirable. The resulting protocols usually tend to be dissimilar to the originals and are often-

<sup>14</sup>The WSCI [106] specification was one of the primary inputs for WS-CDL, but was superseded by the latter in 2005.



times harder to read and understand [114]. In fact, analysing the differences in features supported by languages not of the same kind is an active research area. A considerable amount of effort has been put into systematically analysing the expressiveness of workflow languages by comparing them against a catalogue of patterns [115, 116]. The results of this research are to some extent also applicable to protocol languages. Indeed, more research in this direction would be worthwhile in order to converge to a commonly accepted standard.

## 2.8 Service Composition and Workflows

Because of the uniform notations and syntax introduced by Web Services, the functionality and interfaces of application components can be described in a standardised way. As a result, existing applications can be plugged together in unanticipated ways and form new composed applications. Yet composed applications can again be exposed to the network as services in a recursive manner. This allows an arbitrary deep nesting of logic and complexity at increasingly higher levels of abstraction. Because the logic is encapsulated in the service and not publicly visible, consuming services are unaware whether functionality is provided by conventional means or by combining the functionality of aggregated services.

Often, the composition logic is implemented as part of the service code using standard object-oriented languages. This approach, however, has disadvantages because the composition logic is hard coded and thus difficult to change and maintain. As a result, methods and languages have been created that enable separating the composition logic from its implementation and describe it in a more abstract way as a workflow<sup>15</sup>.

### 2.8.1 Relationship with Interaction Protocols

Even though there are strong links between workflows and protocols, they address different goals. The former are private detailed descriptions that define internal implementation aspects of a Web Service while the latter are public documents that omit implementation details and capture only valid message exchange sequences. Protocols specify *what* a service does in terms of the exchanged messages, whereas workflows specify *how* it is done. Consequently, workflows can be executed but protocols cannot. Inevitably, changing a workflow will influence the associated protocol and vice versa. In fact, we established earlier that every service (composed or conventional) implicitly defines a protocol. If the service implementation is specified through a workflow language, tools

---

<sup>15</sup>Often also called process or business process.

can be built to generate an explicit version of the protocol. Analogously, protocol descriptions can be used to drive the design of workflows [3]. Figure 2.11 illustrates the difference between protocols and workflows.

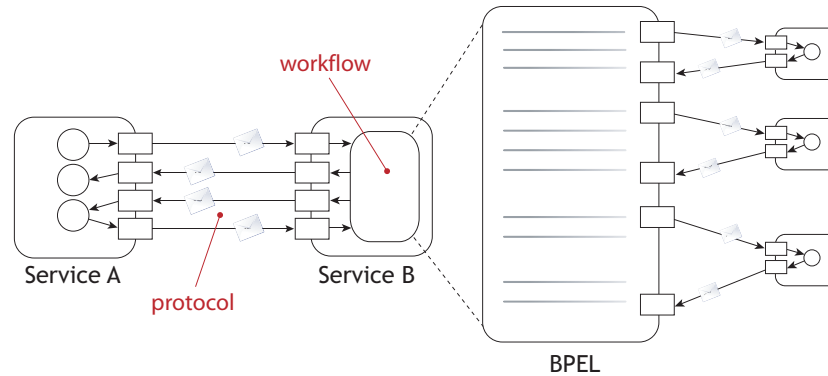


Figure 2.11: Difference between protocol and workflow. The messages exchanged between Service A and Service B are triggered by the respective service implementation and must adhere to a protocol. Service B is internally implemented through a workflow (e.g. BPEL) that specifies how it interacts with the aggregated services on the right. Service A, however, is not aware of these interactions, because they are part of Service B’s implementation.

### 2.8.2 Business Process Execution Language (BPEL)

The Web Services Business Process Execution Language (WSBPEL) [64] has emerged as the de-facto standard for describing and executing workflows. Apart from offering a large number of constructs for capturing various workflow structures, BPEL also supports a model for long-running transactions, message correlation, dynamic partner binding and so forth. BPEL processes can be either *executable* or *abstract*. *Executable* BPEL processes are typically private documents and specify all the information necessary to execute the process on a workflow system. Conversely, *abstract* BPEL processes are – despite their name – used to capture protocols. They are intended to be published and define only the possible service interactions without revealing implementation details. Although the name might not suggest it, BPEL cannot just be used in enterprise computing, but likewise in other process-aware applications. Especially in Grid computing [117, 118], it has been successfully applied for describing and executing scientific workflows [119, 120].

Even though a lot has been written about workflows and workflow management (a comprehensive overview is given in [121]) there is still little consensus about their conceptual and formal foundations [114]. Van der Aalst et al., for example, generally accept BPEL as a powerful language, but argue that it is

too complex, too verbose and too difficult to use. In an attempt to tackle this problem from a different angle, Russel et al. present an extensive catalogue of workflow patterns and use it to systematically compare features and semantics of numerous workflow languages [116]. Based on this work, Wohed et al. established that BPEL is, for example, not capable of representing arbitrary cycles [122]. Indeed, we were confronted with this limitation when modelling the protocol as an abstract BPEL process in our case study, presented in Chapter 5.

Based on the workflow pattern research, van der Aalst and ter Hofstede have created a new language including runtime support called Yet Another Workflow Language (YAWL) [123, 124]. According to [116] YAWL supports about three quarter of the workflow patterns. Still, BPEL has been widely adopted in industry and academia and is unlikely to be replaced by another proposal in the near future. It is now firmly established as a standard and a considerable number of tools providing support at different levels have been created (e.g. [125, 126, 127]).

## 2.9 SOAP Service Description Language (SSDL)

The SOAP Service Description Language (SSDL) [10] is an XML-based language for describing asynchronously communicating Web Services in a message-oriented way. It focuses on one-way messages as the building blocks for creating service-oriented applications and provides mechanisms for describing the structure of SOAP messages. Moreover, it offers an extensible range of pluggable protocol frameworks, which can be used to combine messages into protocols that define their relative ordering (see Section 2.7). Additionally, some protocol frameworks can be used to create protocols that are amenable to formal verification using model checkers to ensure the absence of deadlocks or race conditions.

### 2.9.1 Structure

An SSDL contract can be separated into the following four major sections:

- **Schemas section:** Defines the structure of SOAP message elements used by the service, normally using XML Schema;
- **Messages section:** Declares the SOAP messages that a service supports, including body elements and header elements that cannot be inferred from an associated policy document;
- **Protocols section:** Defines how messages relate to each other and the valid sequences in which they can be exchanged. Different protocol frame-

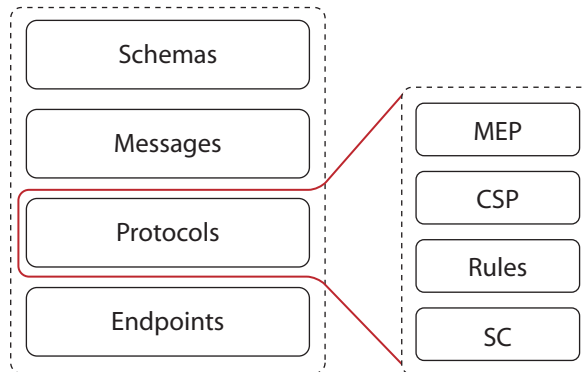


Figure 2.12: The structure of an SSDL contract. Initially four protocol frameworks have been released with SSDL, but new ones can be added, if required. Adapted from [10].

works can be used, depending on the required level of formal verification and the number of parties involved in a protocol;

- **Endpoints section:** Uses WS-Addressing [13] to define endpoints of Web Services that are known to support the given contract.

## 2.9.2 Messages

SSDL assumes SOAP (over arbitrary transport protocols) together with WS-Addressing as the only means of transferring messages between services. In fact, SSDL relies on WS-Addressing headers to correlate and dispatch messages (see Section 4.3.1). Consequently, defining bindings for different transport protocols is unnecessary and messages can be described in a more lightweight way than can be with WSDL, which does not explicitly target SOAP. Likewise, adopting SOAP from the outset gives developers greater control over message structures, because it makes it possible to define SOAP header elements as part of the contract. Figure 2.13 illustrates how a message is defined in an SSDL contract.

Yet this reduction in complexity compromises flexibility to some degree. Admittedly, most of today's Web Services use SOAP for communication, but there are other applications, such as those based on REST principles, that do not. Also, some applications (e.g. grid computing, mobile devices, etc.) might not choose SOAP because its XML syntax can lead to an increase in message size. Instead, they might choose more lightweight message representations or binary XML formats (e.g. [128]) in order to increase performance.

---

```

<ssdl:messages targetNamespace="urn:my:messages" xmlns:s="urn:my:schema">
  <ssdl:message name="MsgA">
    <ssdl:header ref="s:MyHeaderX"
      mustUnderstand="true" />
    <ssdl:header ref="s:MyHeaderY"
      role=".../ultimateReceiver"/>
    <ssdl:body ref="s:MyBody" />
  </ssdl:message>
</ssdl:messages>

```

---

Figure 2.13: A message defined as part of an SSDL contract. The *header* and *body* refer to XML schema elements.

### 2.9.3 Protocols

Messages defined in the contract can be combined and related into protocols. Currently, four protocol frameworks – MEP (Message Exchange Pattern) [129], CSP (Communicating Sequential Processes) [101], Rules [102] and SC (Sequencing Constraints) [98] – have been specified, but additional protocol frameworks can be created and plugged into SSDL, if needed. Although this kind flexibility is generally appreciated it is a mixed blessing and we fear that it might be obstructive to achieving a common standard. One could create a Web Service description language simply by posing no restrictions on its content (e.g. `<xsd:any processContents="skip"/>`). Clearly, almost everything could be represented in this language. However, this of course would not solve the actual interoperability issues, but merely shift them to another area.

#### 2.9.3.1 MEP Protocol Framework

Of the four initial SSDL protocol frameworks, the MEP framework [129] is the simplest and least sophisticated. It does not demonstrate SSDL's full strength and has primarily been designed for capturing the message exchange patterns defined in [85] so it can be used as a simple SOAP-centric language replacement for WSDL. Figure 2.14 shows a protocol captured using the MEP framework. The first *mep* element on line 2 defines an *in-only* pattern in which *MsgA* represents the incoming message. The second *mep* element on line 5 defines an *in-optional-out* pattern with *MsgB* representing the incoming message, *MsgC* being the outgoing message and *FaultX* standing for the optional fault message.

#### 2.9.3.2 SC Protocol Framework

Other protocol frameworks, however, allow specifying much more sophisticated protocols. The semantics of the SC framework, for example, are based on  $\pi$ -calculus and enable the definition of protocols over and above simple *request-*

---

```

1 <ssdl:protocol xmlns:mep="urn:ssdl:mep:v1">
2   <mep:in-only>
3     <ssdl:msgref ref="MsgA" direction="in"/>
4   </mep:in-only>
5   <mep:in-optional-out>
6     <ssdl:msgref ref="MsgB" direction="in"/>
7     <ssdl:msgref ref="MsgC" direction="out"/>
8     <ssdl:msgref ref="FaultX" direction="out"/>
9   </mep:in-optional-out>
10 </ssdl:protocol>

```

---

Figure 2.14: Interaction protocol specified using the MEP SSDL protocol framework.

*response* patterns. Figure 2.15 illustrates a simple SC protocol that could not be expressed using WSDL. The protocol defines that the service is initialised by *MsgA* from another service referred to as *U*. Next, the defined service sends either *MsgB* or *MsgC* to another service referred to as *V*. Then, it expects *MsgD* back from *V* and finally delivers *MsgE* back to *U*. Note that this specification is declarative. It just defines *what* can happen. *How* it will eventually happen, is determined by the application logic that implements the contract.

---

```

1 <ssdl:protocol xmlns:mep="urn:ssdl:sc:v1">
2   <sc:sc>
3     <sc:participant name="U"/>
4     <sc:participant name="V"/>
5     <sc:protocol>
6       <sc:sequence>
7         <ssdl:msgref ref="MsgA" direction="in" sc:participant="U"/>
8         <sc:choice>
9           <ssdl:msgref ref="MsgB" direction="out" sc:participant="V"/>
10          <ssdl:msgref ref="MsgC" direction="out" sc:participant="V"/>
11        </sc:choice>
12        <ssdl:msgref ref="MsgD" direction="in" sc:participant="V"/>
13        <ssdl:msgref ref="MsgE" direction="out" sc:participant="U"/>
14      </sc:sequence>
15    </sc:protocol>
16  </sc:sc>
17 </ssdl:protocol>

```

---

Figure 2.15: Interaction protocol specified using the SC SSDL protocol framework.

## 2.9.4 Claimed Benefits

SSDL claims to take a more lightweight approach at describing Web Services than incumbent standards such as WSDL [86]. By focusing on messages, SSDL expects to encourage the creation of loosely-coupled and service-oriented ap-

plications. Moreover, SSDL supports developers working directly with SOAP messages as their fundamental abstraction and discourages them from thinking about exposing application objects directly as Web Services. Finally, providing mechanisms for capturing a service's messaging behaviour allows exposing this information to other services as part of the service description.

### 2.9.5 Tool Support

Unfortunately, almost no data exists that reports on experiences using SSDL as part of Web Services-based SOAs. There is only one set of published results from a project known to have used SSDL to model its services [130]. The lack of empirical data makes it hard to assess the capabilities and potential of SSDL as a service description language in a general sense. We believe that the main reason why SSDL has not been used more widely is the lack of tool support for creating and executing SSDL-based Web Services. On one side, there are no tools and programming abstractions that aid developers in modelling and implementing SSDL-based Web Services. On the other side, no SSDL-aware middleware exists, which could exploit the benefits of machine-processable protocol descriptions we described in Section 2.7.

## 2.10 Summary

Diverse architectural styles and technologies exist for creating distributed software applications. Choosing a particular style normally has a number of consequences on the application design. RPC-based approaches attempt to shield the network and provide developers with familiar programming abstractions based on operation invocation semantics. Yet RPC applications tend to be tightly coupled, brittle at distribution boundaries and limited in scalability. Messaging, on the other side, is well suited for integrating applications in a loosely coupled and scalable way. Traditional messaging approaches, however, require centralised infrastructure. In practice, this is a major obstacle for integrating components across organisational, trust or geographical boundaries.

Web Services address interoperability and integration issues by providing a broad range of XML-based standards. They can be used for building applications that adhere to SOA principles, where the primary abstraction for communication is one based on message passing. Yet simply using Web Services does not automatically lead to service-oriented systems. Indeed, there are two main views of Web Services, an RPC-centric view and a message-centric (or document-centric) view. In particular, we accepted that the design of WSDL is RPC-centric and can therefore be obstructive for building service-oriented

applications (e.g. focus on operations rather than messages, insufficient control over SOAP messages, high complexity, unable to capture message ordering constraints over and above simple *request-response* patterns). We concluded the chapter with a discussion of an alternative Web Services description language called SSDL, which provides a more message-centric approach for building service-oriented applications.



## Chapter 3

# The Programming Model

“ Things should be made as simple as possible, but no simpler. ”

— *Albert Einstein*

The programming model presented in Soya [12] allows developers to build SSDL services in a straightforward manner. It encourages the creation of service-oriented applications without imposing unrealistic development burdens. Developers define message structures and protocols using metadata. Soya in turn leverages this information to infer SSDL contracts, which are utilised by its runtime and can be exposed to other services.

Soya is built on top of the Windows Communication Foundation (WCF) [61] and its programming abstractions are partially based on the latter’s programming model, which offers a number of ways to implement services. While this generality and flexibility is normally appreciated, it can also lead to confusion. Although WCF can certainly be used to build message-oriented applications, choosing the wrong programming abstractions can easily corral a developer into creating RPC-like applications. Conversely, Soya’s programming model is more restrictive. In fact, it aims at forcing developers to think in terms of one-way messages and therefore encourages the creation of loosely coupled message-oriented systems.

### 3.1 Development Life Cycle

Creating and deploying a service implementation, normally involves the following basic tasks [131]:

1. defining the service contract including messages and protocols;
2. implementing the contracts;

3. configuring the service;
4. deploying the artefacts created in the previous steps.

Typically, the contractual data is defined declaratively using C# attributes. Contracts are implemented by means of C# code and configuration-related aspects are specified in XML. Although different ways exist to creating these artefacts, we favour the one just described. Figure 3.1 shows these artefacts, to which we will collectively refer to as *service implementation*.

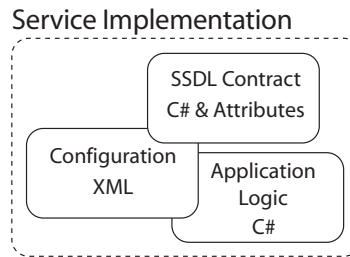


Figure 3.1: The different artefacts of a service implementation. C# code is used to implement application logic, metadata attributes describe the SSDL contract and XML configuration files specify service endpoints, security settings, etc.

## 3.2 Defining SSDL Contracts Using Metadata

C# attributes are a mechanism for declaratively embedding metadata in C# source code. This metadata adds additional information to the code that can be retrieved, processed and interpreted by other programs. In WCF's programming model, service and message contracts are typically defined in this declarative manner [36]. Soya reuses this programming model and provides additional SSDL-specific attributes and functionality. On the one hand, this allows developers to define the structure of messages supported by an SSDL contract. On the other hand, it can be used to describe how these messages relate to each other using different protocol frameworks. This attribute-oriented approach makes it possible to specify contract data with very little code, yet provides extensive control over the contract when warranted.

In Soya, we have adopted the attribute-oriented programming model for the following reasons:

- less code and hence less scope for error introduction;
- more easily maintainable due to single source location;
- seamless integration with WCF's programming model and provision of familiar idioms to existing C# programmers.

Defining contracts using C# attributes is a fundamental concept in both WCF and Soya and different attributes are available for different parts of the contract. Figure 3.2 clarifies the interrelationship between service implementation and inferred SSDL contracts.

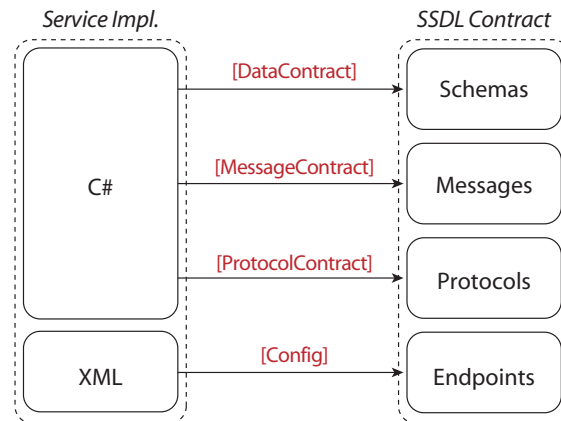


Figure 3.2: SSDL contracts are inferred mostly from C# source code and attributes. Endpoints are usually defined in an XML configuration file.

### 3.2.1 Defining Messages

Where possible, we reused existing WCF attributes to make the transition from WCF to Soya as smooth as possible. For concepts unique to SSDL, however, we introduced additional attributes (e.g. SSDL message names and namespaces, protocols, etc.). The following code snippet shows how message contracts are defined in Soya:

---

```

[SSdlMessageContract] // Soya attribute
public class MsgA {
    [MessageHeader]      public string MyHeader;
    [MessageBodyMember]  public MyData MyBody;
}
  
```

---

The *MessageHeader* attribute maps a class member to a SOAP header. The value of *MyHeader*, for example, is mapped to a primitive XML data type (i.e. *string*). Similarly, *MessageBodyMember* maps to a SOAP body. Above, the value of *MyBody* is mapped to a XML complex type, because it references a custom type. As a result, it is necessary to define how this type is serialised and deserialised to and from XML, respectively. This is done by annotating the class with a *DataContract* and every serialisable class member with a *DataMember* attribute:

---

```
[DataContract(Namespace = "urn:my:schema")] // WCF attribute
public class MyData {
    [DataMember] public int id;
    [DataMember] public string code;
}
```

---

Attributes can take additional property parameters that are used to override default values and give developers more control over the message data. For example, to explicitly specify the qualified name of the SSDL message element in the code above, one simply defines values for the *Name* and *Namespace* properties:

---

```
[Ssd1MessageContract(Name="...", Namespace="...")]
```

---

From the example code above, Soya can infer the following XML Schema and SSDL message element, which are part of the SSDL contract:

---

```
<xs:element name="MyHeader" type="xs:string"/>
<xs:element name="MyBody" type="s:MyData"/>
<xs:complexType name="MyData">
  <xs:sequence>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="code" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
...
<ssdl:message name="MsgA">
  <ssdl:header ref="s:MyHeader"/>
  <ssdl:body ref="s:MyBody"/>
</ssdl:message>
```

---

### 3.2.1.1 Faults

Faults are defined in a similar way to messages. Since SSDL 1.0 does not allow for capturing of the headers of a fault, we only need to define one class that represents the detail element of the SOAP fault message<sup>1</sup>.

---

```
[Ssd1FaultContract]
[DataContract]
public class FaultX {
    [DataMember] public string Code;
    [DataMember] public string Description;
}
```

---

<sup>1</sup>A more extensive discussion of issues related to SSDL faults is provided in Section 6.9.

Again, Soya uses this information to infer XML Schema code as well as SSDL fault elements, which are both part of the SSDL contract:

---

```

<xs:element name="FaultX" type="s:FaultX" />
<xs:complexType name="FaultX">
  <xs:sequence>
    <xs:element name="Code" type="xs:string" />
    <xs:element name="Description" type="xs:string" />
  </xs:sequence>
</xs:complexType>
...
<ssdl:fault name="FaultX">
  <ssdl:detail ref="s:FaultX" />
</ssdl:fault>

```

---

As illustrated in the above examples, Soya leverages the attribute infrastructure provided by WCF (e.g. *MessageHeader*, *MessageBodyMember*). Instead of generating WSDL, however, Soya uses it to generate SSDL contracts.

### 3.2.2 Defining Messaging Behaviour

Apart from defining sets of messages supported by a service, Soya's programming model also allows for the description of how they relate to each other. Instead of correlating them based on operation semantics, however, protocol frameworks are used. The basic building blocks for creating protocols are incoming and outgoing messages and there is no native *request-response* construct. If we want to model such semantics, we need to define them using a protocol framework. Indeed, that is exactly what the MEP framework [129] does. Other protocol frameworks, however, allow the definition of more sophisticated messaging behaviours over and above simple *request-response* and its ilk. This has the positive side-effect of disallowing a one-to-one mapping between message exchange patterns and API method invocations.

Although Soya has been designed to accommodate SSDL's extensible protocol framework model and provides the necessary hooks to plug in new ones (see Section 4.8.1), we have only implemented a programming model for the MEP protocol framework [129]. The reason behind this decision is twofold: First, designing good programming abstractions for supporting protocol frameworks is a non-trivial task and implementing support for additional protocol frameworks was thus beyond the scope of this work; second, the Soya runtime is not aware of which protocol framework or which programming model was used to create it. At runtime, it is thus irrelevant which framework was used to capture the protocol. In the same way, it is insignificant to the runtime in what form the protocol had been defined (i.e. C# attribute, XML files, etc.).

This separation between the protocol framework, the programming model and the runtime is clearly desirable, as it allows Soya to be extended in multiple directions using different approaches. Further, it enables us to evaluate the runtime behaviour in a generic way, without restricting our observations to a particular protocol framework or programming abstraction.

### 3.2.2.1 Modelling MEP-Based Interactions

An SSDL contract is defined by annotating the service interface with a *ServiceContract* attribute. Messaging behaviour is captured using *SsdProtocolContract* and *MEP* attributes. The following lines show how simple MEP-based interactions can be modelled using Soya:

---

```

1  [ServiceContract(Namespace = "urn:my:contract")]
2  [SsdProtocolContract(Namespace = "urn:my:protocol")]
3  public interface IService {
4      [Mep(Style=MepStyle.InOnly)]
5      void Process(MsgA msg);
6
7      [Mep(Style=MepStyle.InOptionalOut, Out=typeof(MsgC),
8          Fault=typeof(FaultX))]
9      void Process(MsgB msg);
10 }

```

---

The attribute on the first method declaration defines an *in-only* MEP in which *MsgA* represents the incoming message. The second method declaration defines an *in-optional-out* MEP with *MsgB* representing the incoming message, *MsgC* being the outgoing message and *FaultX* standing for the optional fault message. From this code, Soya can generate the following SSDL protocol:

---

```

<ssdl:protocol targetNamespace="urn:my:protocol"
               xmlns:mep="urn:ssdl:mep:v1">
  <mep:in-only>
    <ssdl:msgref ref="m:MsgA" direction="in"/>
  </mep:in-only>
  <mep:in-optional-out>
    <ssdl:msgref ref="m:MsgB" direction="in"/>
    <ssdl:msgref ref="m:MsgC" direction="out"/>
    <ssdl:msgref ref="m:FaultX" direction="out"/>
  </mep:in-optional-out>
</ssdl:protocol>

```

---

These examples show how Soya uses type and attribute metadata to infer SSDL contracts. They also give an idea of how little additional code is necessary to create an entire SSDL contract including XML Schema definitions, method and fault declarations and protocol descriptions.

### 3.3 Implementing a Service

A service implementation encapsulates the application logic that we want to expose to other clients through the contract it implements. Although, from an application perspective this can be the most difficult and time-consuming task, it is relatively straightforward from a technical point of view. To do so, a service developer just implements the service contract interface, as the following example illustrates:

---

```
public class MyService : IService {  
    public void Process(MsgA msg) { /* application logic */ }  
    public void Process(MsgB msg) { /* application logic */ }  
}
```

---

Service operations must have exactly one input parameter (i.e. the incoming message). For ease of development, the XML infoset [132] of the incoming SOAP message is mapped to a custom type. If desired, Soya also provides mechanisms to access the XML infoset directly.

In order to adhere to the protocol defined in the service contract above, the service implementer needs to ensure that potential outgoing response messages are sent using Soya's client API at some point during the execution of the service method. Failure to do so might lead to protocol validation exceptions. In the above example, none of the service operations require an outgoing message because we defined two patterns that do not require an outgoing message (i.e. *in-only* and *in-optional-out*). However, if we decide to send an outgoing message during the execution of *Process(MsgB)*, it needs to be either of type *MsgC* or *FaultX*.

### 3.4 Client API

By nature of asynchronous messaging, Soya maps incoming messages to internal API methods without output (i.e. no return value and no output parameters). In order to send response messages, services use a client API provided by Soya. Client functionality is encapsulated in a class called *SoyaClient*. It facilitates sending outgoing messages and is used in the following two cases:

- initiating a new conversation with a remote service (i.e. sending the first message of a conversation);
- sending a response message (i.e. a message relating to a previous one) back to a client.

As a consequence of the architectural thinking behind SSDL and its message-oriented nature, response messages are not distinguishable from other one-way messages. Correlation is done only by means of protocols and not *request-response* patterns. This implies that every (client) service expecting to receive messages (initial or response), must expose a public endpoint that accepts incoming messages.

### 3.4.1 Initiating a Conversation

In Soya, it is therefore always necessary to create a service instance prior to sending a message, so that potential response messages can be handled by the locally running service. Consequently, clients cannot be created explicitly by a service developer, but must be obtained from a running service. This is illustrated in the code snippet below.

First, a new service host is created, from which a pre-configured client can be obtained. The endpoint of the service to which the client will connect is normally declared in a configuration file but can also be specified programmatically via method parameters of the `GetClient()` method. Subsequently, the client can be used to send messages. Moreover, the underlying framework automatically handles runtime issues such as session management, validation, correlation and so forth:

---

```
SoyaServiceHost host = ...
SoyaClient client = host.GetClient();
Message m = ...
client.Send(m);
```

---

Not allowing the explicit creation of clients has another reason: At the time of sending out the first message, no session context yet exists. When a client is obtained, Soya thus creates and registers a new session context for that (client) service so that it can correlate future messages accordingly. The mechanisms used to achieve this are presented in Section 4.3.2 and apply to both clients and servers alike.

### 3.4.2 Replying to a Previous Message

Sending messages that relate to previous ones involves writing even less code. Although a *SoyaClient* is implicitly created and used, the service developer does not have to create or obtain one directly. Instead, the Soya API defines a static method that can be used, since the runtime has all the information it needs to send the message (i.e. WS-Addressing *ReplyTo* and *RelatesTo*)<sup>2</sup>. The

---

<sup>2</sup>As a result of decoupling addressing information from the transport layer (WS-Addressing) and message correlation from operation semantics (one-way messaging), we could also use a



following code snippet shows a service sending a response that will automatically be correlated with the one currently processed:

---

```
public void Process(MyMessage m) {  
    /* some application logic */  
    Message response = ...  
    SoyaServiceHost.Reply(response);  
}
```

---

## 3.5 Configuring a Service

The final step in creating a deployable service implementation consists of writing the service configuration. Although the configuration can be specified imperatively in code, the more flexible and common way is to define the settings in XML.

The configuration file captures different aspects of the service, such as the endpoint address, where it will be listening for incoming requests, the transport protocol and message encoding used for sending and receiving messages, whether it should use security, reliability and so forth. The following lines, define which class implements the contract and where and how it will be available.

---

```
<system.serviceModel>  
  <services>  
    <service name="MyService">  
      <host>  
        <baseAddresses>  
          <add baseAddress="http://my.service.com/" />  
        </baseAddresses>  
      </host>  
      <endpoint address="basic"  
        binding="BasicHttpBinding"  
        contract="IService" />  
    </service>  
  </services>  
</system.serviceModel>
```

---

Given the configuration data, Soya infers the last part that was missing in the SSDL contract, namely the SSDL endpoint:

---

```
<ssdl:endpoints>  
  <ssdl:endpoint xmlns:wsa="http://www.w3.org/2004/12/addressing">  
    <wsa:Address>http://my.service.com/basic</wsa:Address>  
  </ssdl:endpoint>  
</ssdl:endpoints>
```

---

different transport protocol for the response (i.e. HTTP for the initial message and SMTP for the response.)

The complete SSDL contract that is inferred from the example code in this section is shown in Figure 3.3.

### 3.6 Client-Server Symmetry

In a truly service-oriented environment, services communicate freely with each other by actively sending or receiving messages at any given point in a conversation. The terms *client* and *server* can thus be deceptive to some extent and their roles might even change in a dynamic fashion during a single conversation. We thus apply these terms more out of convenience than correctness. Consequently, we use *client* to refer to a service that initiate a conversation and *server* for the other service.

From a message processing point of view, it is not necessary to distinguish between the two roles. Indeed, it is irrelevant whether an incoming message is a response from a server or a request from a client. In the same way, it is irrelevant whether an outgoing message is a response sent back to a client or a request sent to a server.

This makes it possible to use Soya on both *client* and *server* side without making any changes to the runtime or the programming model. What changes, however, is the definition of the SSDL contract. To better illustrate this, consider the following example: *ServiceA* is a server in the traditional sense, because it waits for incoming messages, processes them and sends response messages back to clients. A possible contract definition that captures this messaging behaviour using Soya's programming model could look like this:

---

```
public interface ServiceA {
    [Mep(Style=MepStyle.InOut, Out=typeof(MsgB))]
    void Process(MsgA msg);
}
```

---

Conversely, *ServiceB* is a client in the traditional sense, because it first sends a message to a server and then expects a response message to come back. A contract definition for a compatible client's messaging behaviour could therefore be:

---

```
public interface ServiceB {
    [Mep(Style=MepStyle.OutIn, Out=typeof(MsgA))]
    void Process(MsgB msg);
}
```

---

Note that we only changed the messaging behaviour (i.e. *out-in* instead of *in-out*). This illustrates that the programming model can be used for any kind

---

```

<ssdl:contract xmlns:ssdl="urn:ssdl:v1" targetNamespace="urn:my:contract">
  <ssdl:schemas>
    <xs:schema targetNamespace="urn:my:schema"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="urn:my:schema">
      <xs:element name="MyHeader" type="xs:string" />
      <xs:element name="MyBody" type="tns:MyData" />
      <xs:element name="MyData" type="tns:MyData" />
      <xs:element name="FaultX" type="tns:FaultX" />
      <xs:complexType name="MyData">
        <xs:sequence>
          <xs:element minOccurs="0" name="code" type="xs:string" />
          <xs:element minOccurs="0" name="id" type="xs:int" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="FaultX">
        <xs:sequence>
          <xs:element minOccurs="0" name="Code" type="xs:string" />
          <xs:element minOccurs="0" name="Description" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </ssdl:schemas>
  <ssdl:messages targetNamespace="urn:my:message" xmlns:ns1="urn:my:schema">
    <ssdl:message name="MsgA">
      <ssdl:header ref="ns1:MyHeader" mustUnderstand="false" relay="false" />
      <ssdl:body ref="ns1:MyBody" />
    </ssdl:message>
    <ssdl:message name="MsgB">
      <ssdl:header ref="ns1:MyHeader" mustUnderstand="false" relay="false" />
      <ssdl:body ref="ns1:MyBody" />
    </ssdl:message>
    <ssdl:message name="MsgC">
      <ssdl:header ref="ns1:MyHeader" mustUnderstand="false" relay="false" />
      <ssdl:body ref="ns1:MyBody" />
    </ssdl:message>
    <ssdl:fault name="FaultX">
      <ssdl:detail ref="ns1:FaultX" />
    </ssdl:fault>
  </ssdl:messages>
  <ssdl:protocols>
    <ssdl:protocol targetNamespace="urn:my:protocol"
      xmlns:mep="urn:ssdl:mep:v1">
      <mep:in-only xmlns:ns2="urn:my:message">
        <ssdl:msgref ref="ns2:MsgA" direction="in" />
      </mep:in-only>
      <mep:in-optional-out xmlns:ns2="urn:my:message">
        <ssdl:msgref ref="ns2:MsgB" direction="in" />
        <ssdl:msgref ref="ns2:MsgC" direction="out" />
        <ssdl:msgref ref="ns2:FaultX" direction="out" />
      </mep:in-optional-out>
    </ssdl:protocol>
  </ssdl:protocols>
  <ssdl:endpoints>
    <ssdl:endpoint xmlns:wsa="http://www.w3.org/2004/12/addressing">
      <wsa:Address>http://my.service.com/basic</wsa:Address>
    </ssdl:endpoint>
  </ssdl:endpoints>
</ssdl:contract>

```

---

Figure 3.3: The complete SSDL contract inferred from C# attribute meta data.

of service, independent of their respective role in a conversation. Of course, the same is true for Soya's runtime.

### 3.6.1 XML Firewalls

Clients that define their interactions in this way can locally isolate validation logic, essentially building a sort of *XML firewall*. This protects them from services whose actual messaging behaviour does not adhere to the contract they expose. This could happen because a malicious service tries to exploit potential vulnerabilities or because a service's messaging behaviour has changed but not been propagated to the client. Either way, clients using Soya are protected from these kind of erroneous messages as illustrated in Figure 3.4.

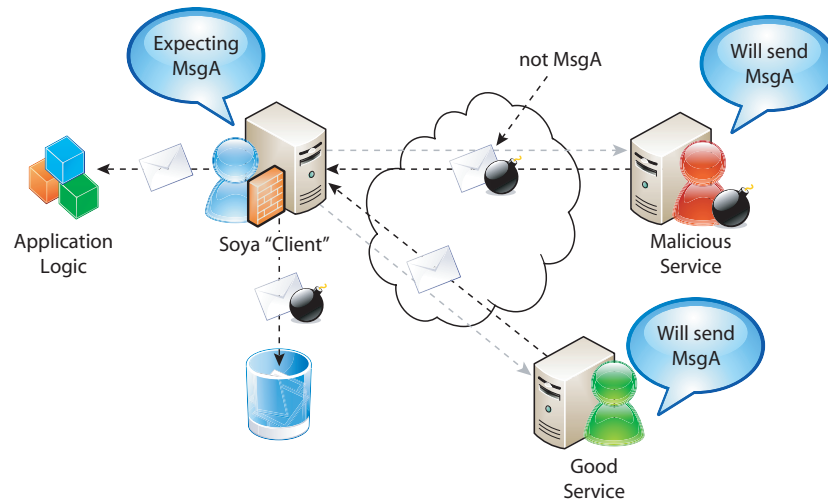


Figure 3.4: Soya acts as an *XML firewall* ensuring that the only messages that reach the application logic are those that it is prepared to receive.

## 3.7 Service Deployment

After defining contractual data, implementing application logic and specifying configuration aspects, the service needs to be deployed and turned into an executable so that other services can interact with it. This is done by creating a *SoyaServiceHost* instance, which is used instead of WCF's default host implementation. The custom Soya host gives us the possibility to modify application-wide behaviour at the service level and encapsulates SSDL-related functionality in a clean abstraction that can easily be reused across many different service applications.

The host is created using a reference to the class that implements the service contract. Upon opening the host, Soya reflects over the service and message types, processes configuration files and builds the runtime from this information (see Section 4.8). The following two lines show the API usage for deploying a service implementation (in this case *MyService*):

---

```
host = new SoyaServiceHost(typeof(MyService));  
host.Open();
```

---



## Chapter 4

# The Runtime Environment

“ Perfection in engineering is achieved not when there is nothing left to add,  
but when there is nothing left to take away. ”

— *Antoine de Saint-Exupéry*

Apart from presenting developers with programming abstractions for creating SSDL-based Web Services, Soya [12] also provides a runtime environment for executing them. The runtime infrastructure processes incoming and outgoing SOAP messages and ensures contract conformance in terms of their structure and ordering. Further, it features facilities for correlating and dispatching both incoming and outgoing messages.

### 4.1 Introduction

The Soya runtime can be seen as a kind of middleware software that adds SSDL-specific functionality and semantics to an existing SOAP engine. It is built on top of the Windows Communication Foundation (WCF) [61], which is a communication platform that provides support for processing SOAP messages and a number of other Web Services specifications. Choosing an existing platform, allowed us to concentrate on implementing SSDL protocol support and delegate issues such as efficient processing of SOAP messages, failure recovery, implementing support for different bindings and so forth to the underlying framework. Figure 4.1 schematically illustrates how the Soya runtime is transparently placed between WCF and the application logic.

Soya leverages WCF in a number of ways. It alters its default runtime behaviour by means of configuration. Moreover, it replaces parts of the default WCF runtime logic with custom components (e.g. custom message correlation, state management or operation invocation implementations). This is done either

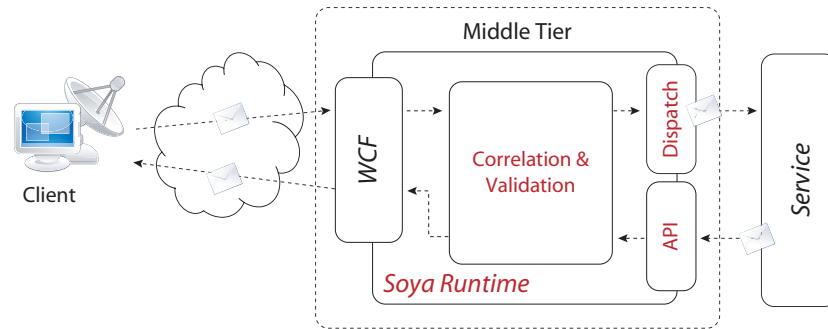


Figure 4.1: The Soya runtime can be thought of as an additional layer between Web Service middleware and application logic.

by directly plugging them into the WCF runtime or by injecting behaviours which in turn add the custom logic. All of these custom components are invoked by WCF at various stages of service execution. Furthermore, Soya includes a number of WCF-independent components that likewise contribute functionality (e.g. message and protocol validation logic) to the overall system. They are, however, not directly invoked by WCF. Figure 4.2 schematically illustrates these interrelationships.

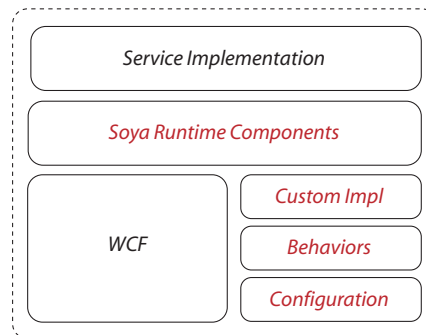


Figure 4.2: Configuration files, injected behaviours and custom implementations modify WCF's default runtime behaviour. Above them is another layer of SSDL functionality that is independent of WCF.

## 4.2 Runtime Components

The components that constitute the runtime each provide specific functionality that are important for the overall runtime behaviour. Consequently, most of them are involved in processing every incoming and outgoing message. In particular, the following seven components realise Soya's core functionality. Apart from the last component, the given order reflects the sequence in which they



are invoked at runtime, when processing an incoming message:

1. message correlation and state maintenance;
2. structural validation;
3. protocol validation;
4. O/X mapping;
5. operation dispatching;
6. metadata generation and exposure.

In the following sections, we will discuss each component individually and provide some more insights into how they work. We will follow the execution order suggested above. However, we will not elaborate on the O/X mapping component, because we mainly used standard WCF mechanisms to realise its functionality.

#### 4.2.1 Processing Flow

When an incoming message is received from the network, it is first of all pushed through WCF's channel stack, which consists of different elements that deserialise, decode, decrypt and so forth the incoming bits into a *Message* object. Immediately after the message exits the channel stack, it is correlated to a potential existing conversation and its corresponding state. Next, it is intercepted by a custom message inspector and handed over to the Soya validation runtime, which performs a number of checks in terms of message structure and ordering. Then, it is passed back to the WCF runtime where it is mapped to a custom type and finally dispatched to a service operation. This is illustrated in Figure 4.3.

Similarly, but in reverse order, outgoing messages sent using Soya's client API are correlated, routed through the validation runtime and finally sent down the WCF channel stack to the network.

### 4.3 Message Correlation & State Maintenance

Inherently, Web Services have no notion of state and do not offer any of the stateful interaction facilities that most distributed object technologies provide as a basic functionality [20]. Generally, ensuring statelessness in the implementation of Web Services is viewed as good engineering practice, as it increases the reliability and scalability of services [133]. A stateless Web Service can easily be interchanged with a different logical or physical manifestation. This might

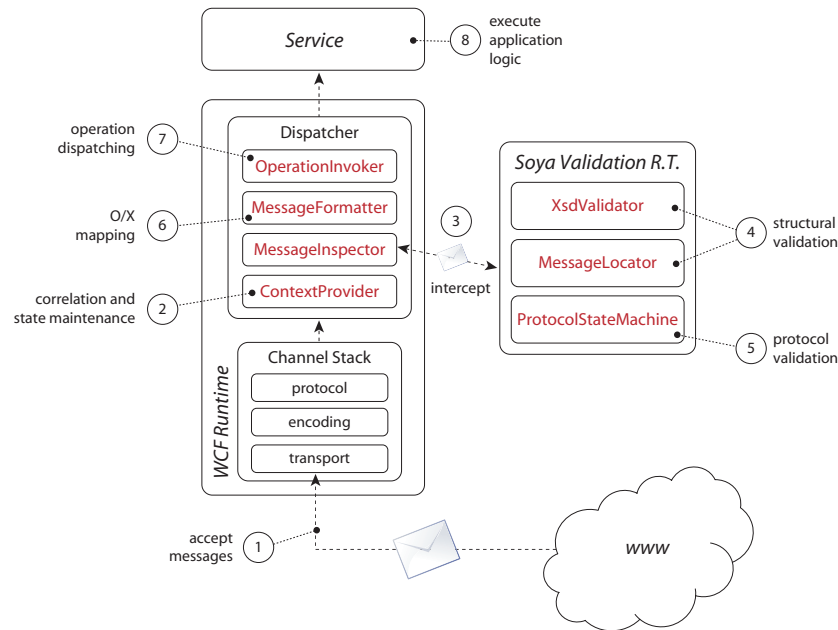


Figure 4.3: Message processing flow. Messages pass through a number of components that perform various tasks and activities before they are dispatched to a service operation.

be necessary, as a result of an increase in server load or failure of a particular service instance.

Not being able to relate consecutive messages into ongoing conversations, however, allows for the creation of only very limited distributed systems [20]. Often, applications need to logically relate a sequence of messages, such that the results of one message exchange can be used as a basis for others. In order to achieve this, exchanged messages need to contain contextual information, which enables services to identify ongoing conversations [134].

A service can use the context information contained in the incoming message to relate it to its internal state. To maintain the benefits of statelessness, a service's internal state needs to be kept in other system components (e.g. an enterprise-grade database), so that other stateless service instances can access it. This allows a service's behaviour to appear stateful, while its implementation in reality remains stateless. Figure 4.4 illustrates these concepts graphically.

Indeed, all of Soya's runtime components are implemented in a stateless fashion and only a minimum of stateful objects are maintained in session context. In fact, the only stateful object that is maintained across message exchanges is a copy of protocol state machine instances (see Section 4.5.6.

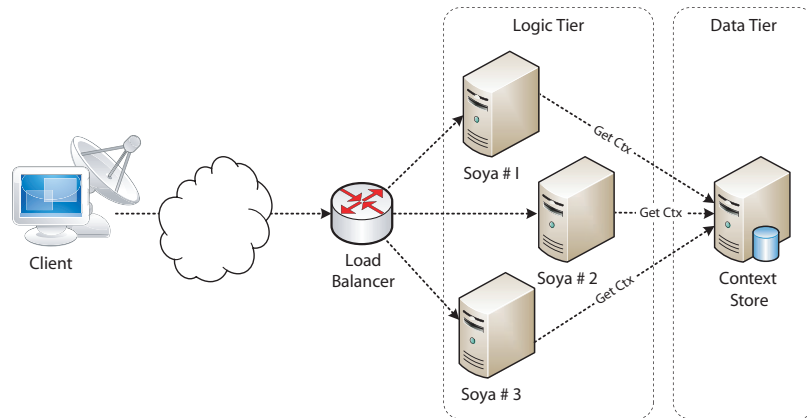


Figure 4.4: Externalising the session context allows service instances to be physically interchanged between message exchanges without losing the session context. Adapted from [135].

#### 4.3.1 Leveraging WS-Addressing

SSDL is predicated on SOAP and WS-Addressing [13]. In Soya, relating incoming messages to conversations and internal state is consequently done by means of *MessageID* and *RelatesTo* headers. Simply put, the *RelatesTo* header of every new incoming or outgoing message is linked with the *MessageID* header of a previous message and thus forms an ordered sequence of header pairs (i.e. messages). For example:

$$(1), (2, 1), (3, 2), (4, 3), \dots$$

Although stateful interactions between services could also be realised using endpoint references and reference parameters<sup>1</sup>, we did not choose this approach because it implies the use of custom – and thus potentially non-standardised – SOAP headers. In contrast, our approach ensures a higher degree of interoperability, as only standardised WS-Addressing headers are used.

#### 4.3.2 Session Context Management

In accordance with above discussions, session context (i.e. state) is stored separately and referenced by contextual information in the form of WS-Addressing headers. In fact, session context stored in Soya is always associated with the last processed message. This means that if a new message arrives, it can be located based on the message's *RelatesTo* header. If no session exists yet (i.e. first message of a conversation), one is created. At the same time, Soya associates

<sup>1</sup>W3C, “Web services addressing”, 2004, Section 2

the context with the new *MessageID* and updates the session context store (e.g. a database) to reflect the fact that both future incoming or outgoing messages must relate to the one just received.

This behaviour is best illustrated with an example. Figure 4.5 shows a simple conversation that spans multiple logically connected messages. A client sends an initial message with id 001. Because it does not relate to a previous one, the service creates a new context and stores it (e.g. in a database). After the service has finished processing the message, the in-memory context is destroyed in order to free resources. Some time later, the client sends a new message with id 002. By setting the *RelatesTo* header to the previous message, it indicates that it is part of the same conversation. Upon receiving the message, the service uses the *RelatesTo* information to retrieve the previously stored session context and load it back into memory. At the same time, it updates the context store to reflect the fact that the message with id 002 has been consumed (by the current operation) and that future messages of the same conversation will need to relate to the new id (i.e. 002). After processing message 002, the service sends a response back to the client. Soya's client API automatically correlates the message by adding appropriate header information (i.e. setting *MessageID* to 003 and *RelatesTo* to 002) (see Section 3.4).

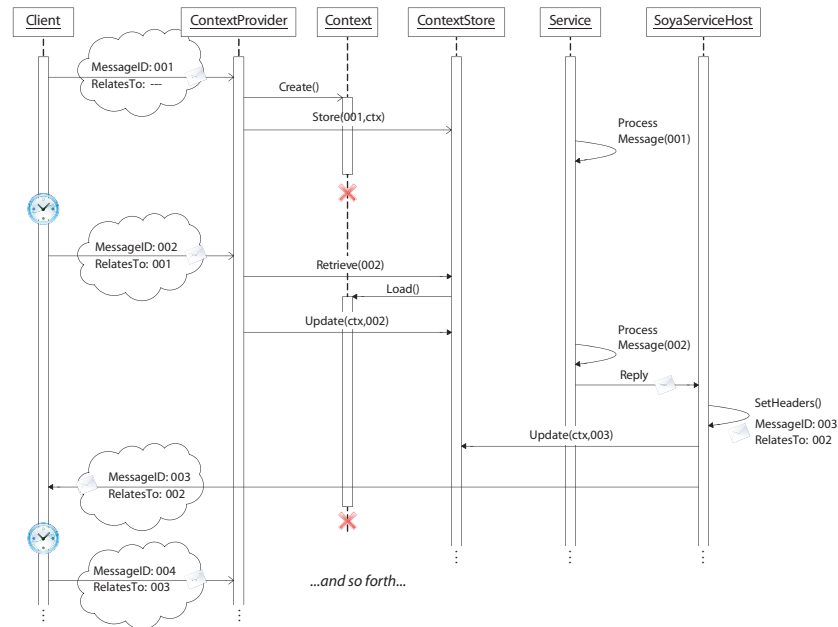


Figure 4.5: A typical conversation that involves creating, retrieving and updating the session context.

### 4.3.3 Session Context Lifetime and Maintenance

In Soya, the lifetime of a session context object is treated differently depending on whether it was originally created by a client (i.e. *SoyaClient*) or by a service (i.e. upon reception of an initial message).

#### 4.3.3.1 Created by Clients

If a program initiates a conversation using a *SoyaClient* (see Section 3.4), the session context it creates is implicitly bound to its lifetime. This behaviour is based on the assumption that a user who disposes of the client, is no longer interested in the conversation with the service. Additionally, this strategy avoids memory leaks caused by session context zombies. Consequently, the client instance needs to be kept in memory for the entire duration of the conversation in order to maintain the session context. Indeed, this behaviour has proven to be sufficiently adequate for the services created in our case study (see Chapter 5).

#### 4.3.3.2 Created by Services

In contrast, the session context created on the server side has no preliminarily confined lifetime<sup>2</sup>. The session context is not destroyed automatically, because it cannot be generally determined in advance whether the client will continue the conversation at some point. On the one hand, SSDL protocol descriptions can have optional constructs (e.g. *out-optional-in*). On the other hand, they do not constrain the temporal extension between consecutive interactions (e.g. if a service has not sent a message after  $t$  time, it does not mean that it will not send it after  $t + \Delta$ ). Of course, this can pose performance problems in production environments, since only a finite number of session context objects can be indefinitely kept in memory. This issue can be addressed using diverse strategies that persist session context after a certain amount of idle time (e.g. into a database). At the same time, this has the positive side-effect of increasing application scalability and reliability as we discussed in Section 4.3. Determining and adjusting the parameters which govern when session context objects should be persisted or destroyed (e.g. client idle time, client activity, server load, etc.) can vary significantly between applications and is thus not within the scope of this work.

---

<sup>2</sup>This is only partially true, because it is actually destroyed if the service host is being disposed of. Yet a persistent session context storage implementation (e.g. database, XML file, etc.) could be used to maintain session state that transcends server restarts. In the current Soya release this has, however, not been implemented.

#### 4.3.4 Synchronisation of Session Context Access

Inherently, an SSDL engine such as Soya operates in an environment where a lot of processing has to be done concurrently because many clients may access a service at the same time. As a consequence, Soya's runtime has been deliberately implemented in a stateless fashion and does not cause synchronisation problems. Yet access to stateful objects, such as the session context, needs to be synchronised between concurrently running threads. Still, communicating with any number of clients simultaneously does not cause any session context synchronisation problems, because the server maintains a separate context for each conversation individually.

As soon as clients send many messages simultaneously within the same conversation, however, many concurrently running threads will be created on the server that can access the session context simultaneously. Therefore, access to the session context needs to be synchronised. To guarantee a consistent, in-sequence processing of incoming and outgoing messages according to a service's protocol, we demarcate the message processing flow into a critical section. Inside this section, no other thread can access the session context while one is already processing a message for the same context. The critical section begins when the session context is assigned to a thread and ends when the message is dispatched to the application (i.e. when the protocol state machine reflects the new state of the conversation). In order to enter the critical section, competing threads must acquire a processing context. This mechanism is shown schematically in Figure 4.6.

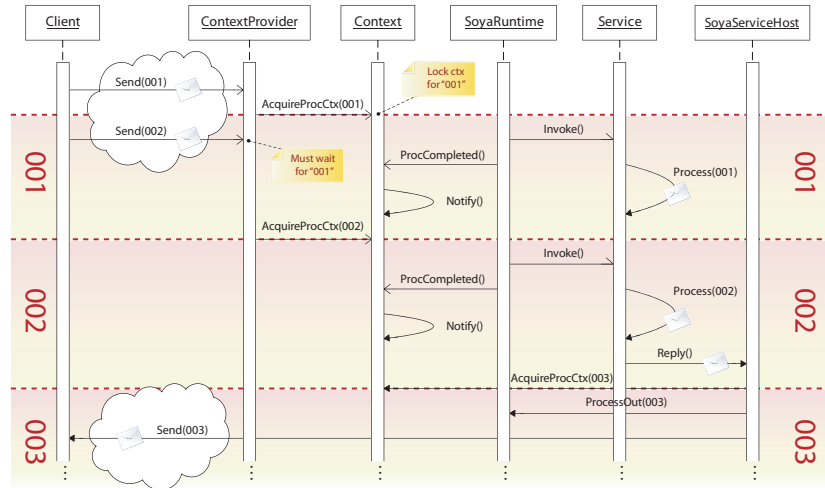


Figure 4.6: Multiple threads that are using the same session context are synchronised at the message-level in order to guarantee a consistent, in-sequence processing.

Not only do we need to synchronise access to the session context for incoming messages but also for outgoing messages. This is shown in the bottom part of Figure 4.6, where the service sends a response message via Soya's client API. Analogous to the procedure for handling incoming messages, Soya ensures that no other messages are currently being processed in the same conversation. It thus likewise needs to first acquire the processing context before it can send an outgoing message.

## 4.4 Structural Validation

The structure of incoming and outgoing messages is validated against both schema and message definitions specified in the service's SSDL contract. If one of the two validation steps fails, processing stops and the message is rejected.

Both message header and body elements are taken into consideration for validation. The former, however, cannot be validated as strictly as the latter, because this would interfere with the SOAP processing model's extensibility mechanism [11]. SOAP nodes can add infrastructure information (e.g. addressing, security, etc.) that is likely not defined in the SSDL contract per se. Yet this does not mean that the structure of these messages is invalid. For this reason, only headers which have been defined in the contract are actually validated, while others are silently ignored. In contrast, if the message body contains elements that are not present in the contract, validation fails.

First, the message is validated against the schema defined in the contract (e.g. XML Schema validation). Next, Soya tries to match it with a message description of the SSDL contract. This ensures that the particular combination of header and body elements is actually a contractually defined message. For example, assume that header  $h$  and body  $b$  are both described in the contract's schema section but not grouped together as a message. If a SOAP message with the given elements (i.e.  $h, b$ ) is processed, schema validation succeeds, yet message validation will fail.

## 4.5 Protocol Validation

One of the core components of the Soya runtime is the protocol state machine, which validates the correct sequence of exchanged messages and decides to which service method incoming messages are to be dispatched. It is built at deployment-time from the protocol information captured in the SSDL contract. Although the classes responsible for building it from protocols are inevitably protocol framework specific, the resulting state machine is generic and thus

protocol framework agnostic. It conveys neither information about the protocol framework that was used to build it, nor whether the protocol was derived from C# attributes, XML files or some other abstraction. This gives protocol framework designers and implementers a lot of flexibility since the use of the programming model for defining protocol-related information is decoupled completely from information used to enact the protocol at runtime.

The incoming and outgoing messages represent the transitions of the state machine while the states do not have explicit meaning or names. Figure 4.7 shows the state machines that can be inferred from the message exchange patterns (MEPs) defined in the MEP protocol framework [129]. Of course, the transition symbol variables are replaced with concrete values when the state machine is constructed.

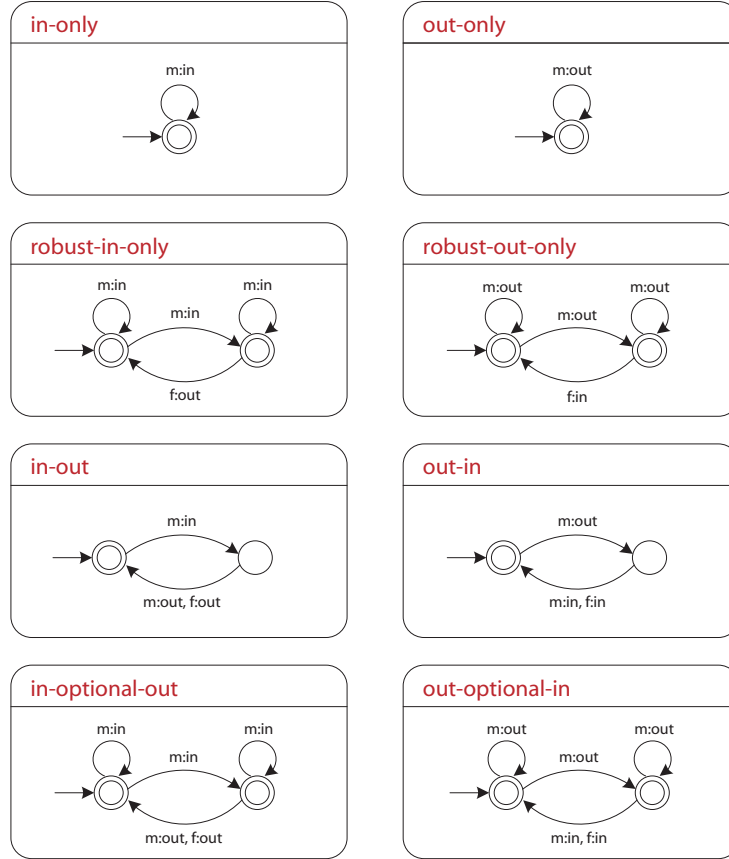


Figure 4.7: State machines inferred from message exchange patterns defined in the MEP protocol framework.

By choosing a common initial state among the different patterns we can create state machines that represent any combination of the eight simple MEPs



or multitudes thereof<sup>3</sup>. Figure 4.8 shows a state machine that combines an *in-only* and an *out-in* MEP.

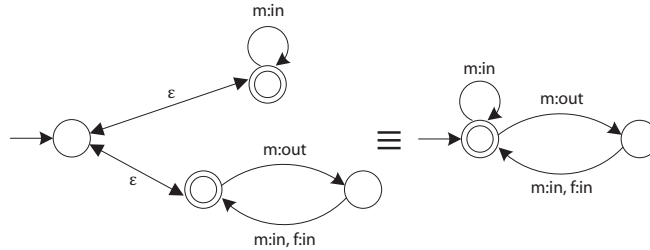


Figure 4.8: A combination of an *in-only* and an *out-in* MEP

### 4.5.1 Dynamic State Pattern

The *GoF State Pattern* [29, 136] is a well-known design pattern that helps to control an object's behaviour by changing its internal state. Normally, one creates a *State* interface that defines a number of state transitions as methods, where each method returns a *State* instance that represents the new state after the transition has occurred. The behaviour of each state is encapsulated in concrete *State* interface implementations, which allows them to be easily interchanged.

In a TCP connection implementation, for example, methods such as `Open()` or `Close()` behave differently depending on the state of the connection (i.e. the concrete *State* implementation that the state machine is currently using). Figure 4.9 depicts the class diagram of a TCP connection example. As visible in the class diagram, each state encapsulates its specific behaviour in a separate class.

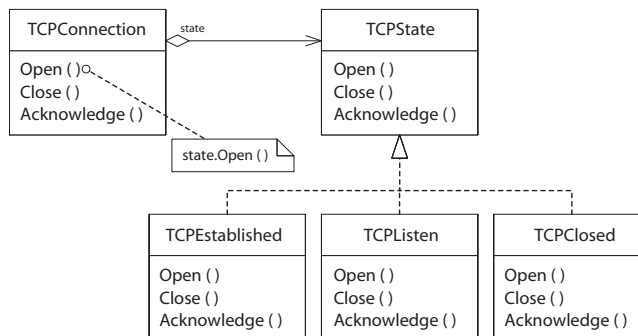


Figure 4.9: TCP connection class diagram from Gamma et al. [29]

<sup>3</sup>We can, of course, also combine multiple MEPs of the same type

In the case of Soya, however, the states are not as clearly defined as in the TCP connection example. Further, we do not know the states or the transitions of the state machine at the time of writing Soya, because they will be created at runtime by a user deploying an SSDL contract.

Still, by slightly twisting the original idea of the *State Pattern* it can be used to dynamically build a generic state machine. Instead of implementing the state behaviour in different classes, however, we encapsulate it in instances of a generic *State* class and configure each instance's behaviour by adding transitions to it, which are valid for the particular state. Transitions are also modelled in a generic way and represented by a two-tuple consisting of a transition symbol and a state where the transition will lead to. In Soya, the transition symbol is composed by the qualified name of a message and its direction (i.e. incoming or outgoing). Because of its dynamic nature, we will refer to this pattern as the *Dynamic State Pattern*. Figure 4.10 illustrates the relationships of the classes that participate in it.

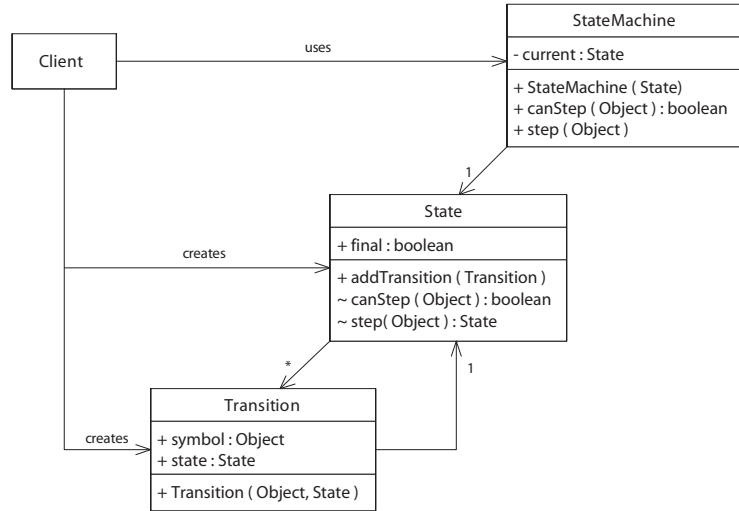


Figure 4.10: Classes participating in the *Dynamic State Pattern*

### 4.5.2 Usage

Soya provides a generic public API for creating state machines that can be used by implementers of new protocol frameworks or programming models. For ease of use, it can be constructed in a non-deterministic way, as this often allows constructing a given state space in a more concise manner.

First, a user creates a number of states, thereby possibly declaring some of them *final*. Then, it creates some transitions and adds them to selected states.

Finally, it creates a state machine by passing it a reference to the initial state. Subsequently, the created state machine can be used to step through the state space and validate input words. If its `Step()` method is called, it internally delegates the call to its current state, which checks if it contains a transition with the required transition symbol. Depending on this evaluation, the method will return the new state defined by the transition or throw an exception. If the method returns successfully, the state machine updates its current state accordingly and waits for the next transition to occur.

### 4.5.3 Implementation

Because the Soya runtime invokes the protocol state machine for every incoming and outgoing message, it is crucial that its implementation can execute the requested transitions efficiently in order to avoid bottlenecks in the processing logic.

Indeed, the *Dynamic State Pattern* just described allows creating a fast-stepping (i.e. constant time) state machine implementation. To achieve this runtime behaviour, we use a hash table<sup>4</sup> for looking up the transitions of a particular state given a transition symbol. In addition, we compile the state space upon creating the state machine, in order to further reduce the execution time of the step operation. This includes performing the following operations:

- eliminating potential non-determinism by creating a deterministic state machine that is equivalent to the non-deterministic one, such that the former and the latter recognise the same language, i.e.  $L(FSA_1) = L(FSA_2)$ ;
- minimising the state space by removing:
  - every useless state  $q$ , where  $q$  is useless iff there is no word  $\omega$  such that there is a transition function  $\delta(q_0, \omega) = q$  or  $\delta(q, \omega) \in F$ ;
  - every state  $q$  that is equivalent to some other state  $q'$  according to the following definition:

$$\begin{aligned} &\text{either } \delta(q, \omega) \in F \quad \text{and} \quad \delta(q', \omega) \in F \\ &\quad \text{or } \delta(q, \omega) \notin F \quad \text{and} \quad \delta(q', \omega) \notin F \end{aligned}$$

- detecting the empty language  $\emptyset$ .

---

<sup>4</sup>Retrieving or setting a value for a given key implemented as an  $O(1)$  operation.

#### 4.5.4 Decorating the State Machine

After the Soya runtime has successfully validated an incoming message, it needs to decide to which internal service method it should be dispatched. This decision is exclusively based on the messaging behaviour defined in the service's contract and the state of the current conversation. It hence stands to reason that we include this dispatching information in the state machine in order to enable other components to look it up at later processing stages.

##### 4.5.4.1 *IExtensibleObject*<T> Pattern

To facilitate this, we implemented a mechanism called the *IExtensibleObject*<T> Pattern [137]. This pattern allows transitions to be dynamically extended by attaching other objects that add new state or functionality. Figure 4.11 depicts the class diagram that shows how *extensions* can be dynamically aggregated by *extensible* objects.

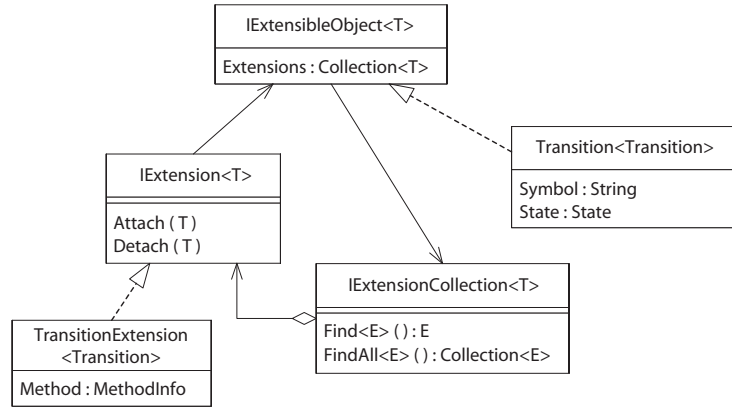


Figure 4.11: Classes participating in the *IExtensibleObject*<T> Pattern

In Soya, we use this pattern to attach extension objects containing service method metadata to transitions that are triggered by incoming messages (we ignore transitions triggered by outgoing messages, because they are not dispatched to service methods). As a state transition occurs, event handlers inside the Soya runtime are notified and extract the method metadata contained in the extensible transition. Figure 4.12 illustrates how dispatching information is attached to transitions of the state machine.

#### 4.5.5 Non-Determinism and Ambiguity

In the context of state machines, the introduction of non-determinism does not provide any extra expression power, i.e. everything a non-deterministic state

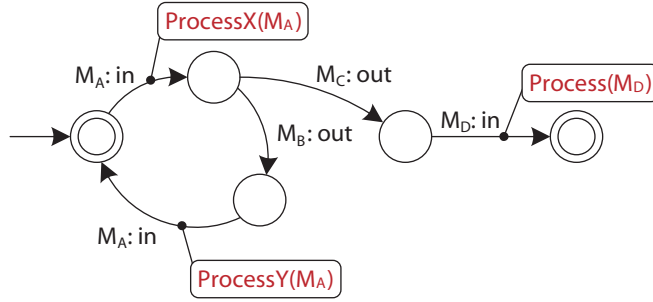


Figure 4.12: Transitions that are triggered by incoming messages are annotated with metadata that is used to decide to which service method the message should be dispatched. In the above diagram, message  $M_A$  is dispatched either to `ProcessX()` or `ProcessY()`, depending on the state of the conversation.

machine can do can be done by a deterministic one and vice versa [138]. However, the use of non-determinism can make a state machine smaller and easier to understand [138].

In order to maintain this flexibility, we allow a state machine to be defined in a non-deterministic way. Upon creating it and compiling the state space, however, we eliminate potential non-determinism by building an equivalent deterministic state machine. This is possible because of the aforementioned equivalence between the two formalisms. At runtime, this allows us to step through the state space without backtracking, since every transition is specified in a fully deterministic way.

#### 4.5.5.1 Ambiguous Service Contracts

Translating the non-deterministic into a deterministic model, however, does not mean that the state transitions are unambiguous in terms of dispatching information. An example of an ambiguously defined service contract is illustrated in Figure 4.13. Because two identical transitions depart from the same state but define distinct dispatching methods, we cannot determine to which service method the incoming message should be dispatched.

```
[Mep(In-Only)]
void ProcessX(MsgA a);

[Mep(Out-In,Out=MsgB)]
void ProcessY(MsgA a);
```

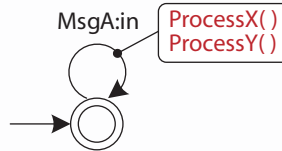


Figure 4.13: A deterministically but ambiguously defined service contract

In contrast, Figure 4.14 presents a service contract that has been defined

unambiguously. Although there are two identical transitions (i.e. *MsgA:in*), they do not depart from the same state and to thus not conflict. It is relatively easy to conceive cases like this where one defines more sophisticated (i.e. non-MEP) protocol descriptions and the same type of message is dispatched to distinct service methods, according to a given state of conversation.

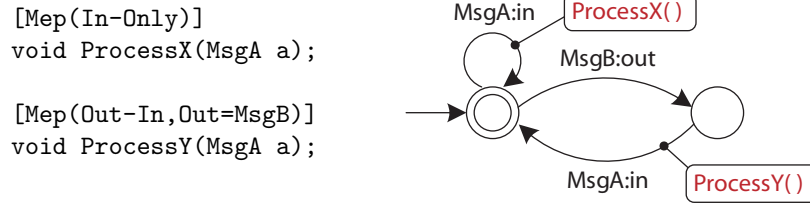


Figure 4.14: An unambiguously defined service contract

However, as we have mentioned before, determinism does not exclude ambiguity in the same way that no ambiguity does not require determinism. This is illustrated – as a last example – in Figure 4.15. By translating the non-deterministic state machine into a deterministic model, it becomes apparent that it had been defined unambiguously.

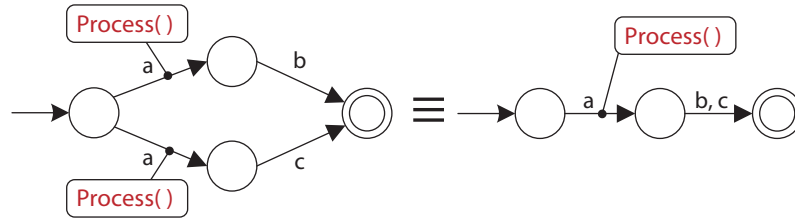


Figure 4.15: A non-deterministically but unambiguously defined state machine

After minimising the state machine and removing potential non-determinism, checking for dispatching ambiguity thus becomes a straightforward task. All we have to do is ensure that only metadata for at most one service method has been attached to any given transition in the state machine.

#### 4.5.6 Persisting State and Cloning

For every deployed SSDL service, Soya creates exactly one protocol state machine. Essentially this can be seen as a blueprint for validating the order of exchanged messages for any conversation. Of course, one service potentially interacts with a number of clients at the same time and therefore needs to keep track of where the conversation currently is for each client individually.

For this reason, a separate cloned instance of the protocol state machine is maintained for each client. As opposed to Soya's runtime components, which

are stateless, the state of the protocol state machine needs to be maintained across multiple message exchanges. For that reason, each conversation has a separate state machine associated with its session context. However, rather than duplicating the entire state space for each client, only the reference to the current state position needs to be copied.

This allows for a convenient programming abstraction, because every client appears to have its own state machine. At the same time, it does not increase the memory footprint unnecessarily, consequently allowing the runtime to handle many clients concurrently.

Similarly, if we want to persist the state of conversations into a relational database, for example, we only need to persist the blueprint (i.e. the original state machine instance) and one reference per client to the current state<sup>5</sup>.

## 4.6 Message Dispatching

### 4.6.1 State-Based Method Selection

In SSDL the concept of *operations* or *service invocations* does not exist and it is expected that applications reason about message content and ordering in order to derive appropriate actions.

Of course, since Soya-based services are built using an object-oriented programming language, a local API method is ultimately invoked. Soya processes incoming messages and decides to which internal method they should be dispatched. This decision is exclusively based on the service's protocol and the current state of the conversation (i.e. the protocol state machine). Method names, however, play no role in this decision-making process. In consequence, this implies that multiple arrivals of the same kind of message can actually be dispatched to different methods.

Although not being able to capture more than two consecutive messages makes the MEP SSDL protocol framework a poor candidate for demonstrating state-based method selection, it can still be used to illustrate the concept. Figure 4.16 shows two methods for processing the same kind of incoming messages. `ProcessX()` is invoked when an unsolicited message of the given type is received. In contrast, `ProcessY()` is only invoked when the service has previously sent an outgoing message of type *MsgB*. The right part of the figure, shows the state machine that is inferred from the protocol definition. Each method on the left may contain application logic that does something different based on the conversation state.

---

<sup>5</sup>The current Soya release 0.1 provides only an in-memory storage facility.

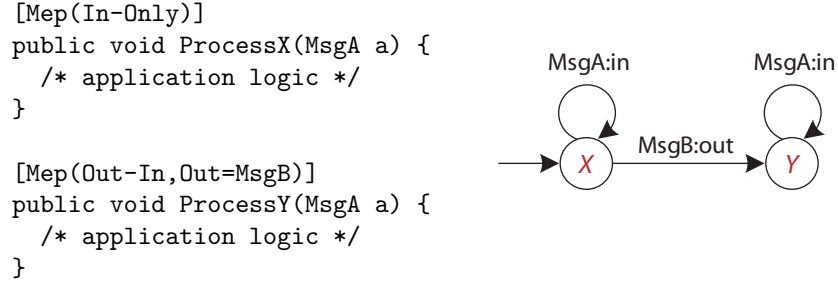


Figure 4.16: State machine inferred from protocol metadata. *X* and *Y* stand for the methods `ProcessX()` and `ProcessY()`, respectively, to which *MsgA* will be dispatched.

Without protocol metadata, the two methods in Figure 4.16 would be ambiguous. In fact, a service developer would need to write application code to distinguish between the two cases, which we understand might impose a significant burden on the developer. Therefore, Soya takes advantage of the protocol metadata to decide to which methods messages need to be dispatched. Presenting the developer with this abstraction removes confusion as what needs to be implemented.

#### 4.6.1.1 Using the Protocol State Machine Decoration

In Section 4.5.4 we have explained how dispatching information is added to the protocol state machine. Every time a state transition is performed, the protocol state machine additionally fires an event that contains transition-related data, including potential dispatching information. As a result, other runtime classes can subscribe to these events and obtain dispatching data. Later, they use this data to invoke application logic and dispatch the message.

### 4.6.2 Asynchronous Operation Invocation

SSDL models interactions between services purely as a collection of exchanged one-way messages. Indeed, messages are related only through defined messaging behaviour and correlated by content and conversation state, not operation semantics. This is facilitated by WS-Addressing header blocks inside the messages and middleware that keeps track of the exchanged messages.

Every actual service invocation thus happens asynchronously, allowing clients to regain control over the connection immediately after the message has been validated and dispatched to the application logic (i.e. the service). Essentially, this decouples the messaging behaviour from operation semantics.

If a client still wants a synchronous interaction behaviour with the service, it can of course block and wait for a correlated message to be returned. This



is a purely local programming decision and does not affect the messages on the wire [139].

#### 4.6.2.1 Decoupling Transport Bindings

SOAP transport bindings, however, may impose temporal constraints on the message exchange. The HTTP binding, for example, requires that *in-out* message exchange patterns are mapped to HTTP *request-response* operations. However, we cannot generally determine when or if a service sends back a response. Consequently, we do not wait for the application logic to finish processing, but instead immediately return an empty HTTP 200 - OK response back to the client, acknowledging that the message has been successfully received and dispatched to the service. Some undefined time later (i.e. seconds or days), the service might send a message back. To do so, a new HTTP request that includes the response message data is created.

If we used SMTP instead, the behaviour at the transport level would be different, because there would be no requirement to send a transport-level response to the initial request. At the service-level, however, this change of transport protocols is completely transparent. This behaviour is illustrated in Figure 4.17.

#### 4.6.2.2 Decoupling Temporal Constraints

By internally calling service methods asynchronously, we also effectively decouple the application logic from temporal constraints imposed by transport bindings, because the semantics of the transport infrastructure are no longer coupled to service operations. Indeed, this practice allows us to reuse a widely deployed transport infrastructure (i.e. HTTP) without imposing the semantics of that infrastructure onto message-oriented services. Further, it has the positive side-effect that Soya inherently supports long-running interactions.

#### 4.6.2.3 Dealing with Infrastructure and Application Errors

Nevertheless, the fact that the client service does not care about the result of a one-way invocation does not mean that it does not care if the invocation took place at all. In fact, if an application requires reliability, additional mechanisms such as, for example, WS-ReliableMessaging [140] should be used, even for one-way messaging. To understand this better, we found it useful to distinguish between errors that can occur in infrastructure and errors that occur in application logic.

Errors occurring in the infrastructure can range from connection problems to protocol validation errors. These kind of errors need to be communicated to the clients (or their middleware), so they will be aware that a given message has

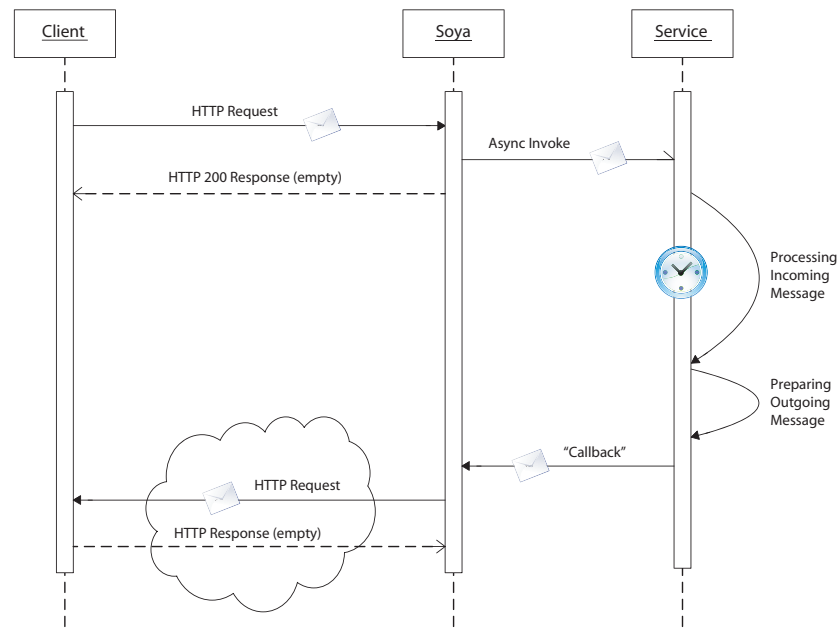


Figure 4.17: The application logic is invoked asynchronously and decoupled from the used transport protocol.

never reached the service. Using the HTTP example again, we could for example return a **HTTP 5xx – Server Error** response, indicating that the message has not been processed.

Errors that occur in the application logic, on the other hand, need to be handled by the application. For example, by sending appropriate error messages to the client that indicate what went wrong. These kind of error messages, however, are part of the normal application flow. In fact, they are part of the service contract and hence even subject to be validated by the runtime.

The following example helps illustrating these concepts: A client sends a *CancelOrderMsg* to the service and does not care about the *result* of the invocation, assuming that no further actions need to be taken. The first time, the client receives a **HTTP 301 – Moved Permanently** response from the service, saying that the URI of the service has changed and the message has not been delivered. The client thus updates the remote endpoint address and re-sends the message to the new URI. This time the message is dispatched to the service successfully, which acknowledges this with a **HTTP 200** response and closes the connection. In the meantime, however, the service application processes the message and finds that the order cannot be cancelled. It therefore sends a new HTTP request to the client (address is taken from WS-Addressing *ReplyTo* header) containing an application error message along the lines of *OrderAlready-*

*ShippedMsg.*

This example illustrates that the latter kind of errors are part of the normal application flow and thus should not be handled by middleware infrastructure, whereas the former should.

## 4.7 Metadata Generation and Exposure

It is essential that a service exposes its contract, so that other services can derive how to interoperate with it or reason about it. Upon creating the runtime, Soya builds an internal model that is subsequently used by its runtime components (see Section 4.8). This model represents different aspects of the service and its contract. Although it cannot be exposed directly, it contains all the information needed to generate SSDL contracts. Consequently, Soya used it to generate XML infoset [132], which in turn can be serialised into XML and published as a SSDL contract.

The way this metadata (i.e. SSDL contract) can be retrieved is via HTTP and the `?ssdl` convention. A client basically sends an HTTP GET request to the base address where the service is located and appends the `ssdl` parameter to the request URL. In turn, Soya sends back a HTTP response that contains XML data representing the SSDL contract. This process is illustrated in Figure 4.18.

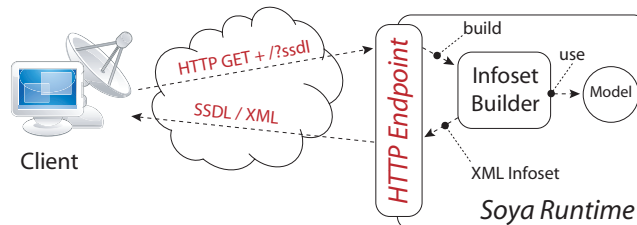


Figure 4.18: Metadata is retrieved using HTTP and the `?ssdl` convention.

## 4.8 Building the Runtime

The creation and configuration of both WCF and Soya runtimes is triggered by deploying a service implementation to a *SoyaServiceHost* (see Section 3.7). On the one hand, the custom Soya host coordinates the creation of an internal service model for subsequent use by runtime components. On the other hand, it configures WCF runtime parameters and replaces default implementations with custom runtime logic (which we have discussed throughout this chapter).

Building the runtime involves the following activities, which are also illustrated graphically in Figure 4.19:

1. reflect over service types and process class, interface and attribute information;
2. process application configuration files (e.g. service endpoints, custom behaviours, etc.);
3. build an internal service model and a protocol state machine blueprint based on the artefacts obtained in the previous steps;
4. create and configure the WCF and Soya runtimes.

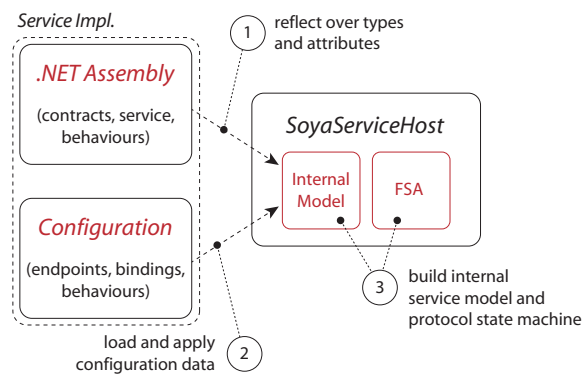


Figure 4.19: The internal service model and protocol state machine are built from service type and attribute information as well as configuration data.

Soya takes SSDL's extensible architecture into account and does not work directly with protocol framework specific classes. In fact, users can add new protocol framework implementations without having to recompile or rebuild Soya. In a sense, the internal service model and the protocol state machine can be seen as an intermediary language between the data captured using the programming model and the SSDL XML contract. They do not convey protocol framework related information and can thus be used in a protocol-independent way. In fact, it does not matter from what source of information these artefacts were originally derived, both in terms of which protocol framework and which programming model were used. This effectively decouples the programming model from the runtime and consequently makes it possible to extend Soya in multiple ways. On the one hand, support for new protocol frameworks can be implemented. On the other hand, different programming models can be conceived, implemented and used (e.g. XML files instead of C# attributes).

Building the internal service model and the protocol state machine is, of course, dependent on the chosen programming model and protocol framework. Therefore, implementers of new protocol frameworks and programming models

must provide the logic for building an internal service model and protocol state machine.

#### 4.8.1 Building the Internal Service Model

As suggested above, the creation of the internal model is dependent on the chosen protocol framework or programming model. As a result, the logic to build the model is delegated and encapsulated in specific classes. These classes must implement interfaces, which represent common abstractions. Decoupling specific implementations from runtime creation or execution logic is achieved using the *Abstract Factory Pattern* [29] as illustrated in Figure 4.20. Deciding which *IProtocolFactory* to use is done automatically at runtime based on the chosen programming model and configuration parameters.

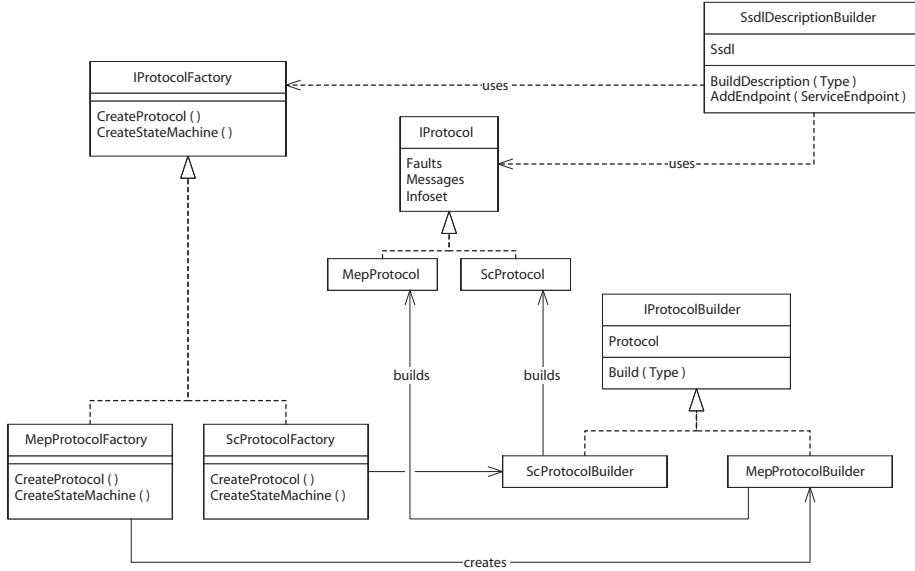


Figure 4.20: Protocol (in)dependent model creation

#### 4.8.2 Building the Protocol State Machine

The protocol state machine is built in a similar manner to the way the internal service model is created. Programming model and configuration data determine which *IProtocolFactory* implementation should be used, which in turn is used to create specific *IStateMachineBuilder* instances. These classes build state machines based on protocol information defined in *IProtocol* instances. Of course, the implementation of protocol and state machine builders must be provided in a consistent way as a family of classes. The *MepStateMachineBuilder*, for ex-

ample, checks if a given *IProtocol* is indeed of type *MepProtocol* before it tries to start building the state machine. Figure 4.21 shows the class diagram of the classes involved in creating state machines.

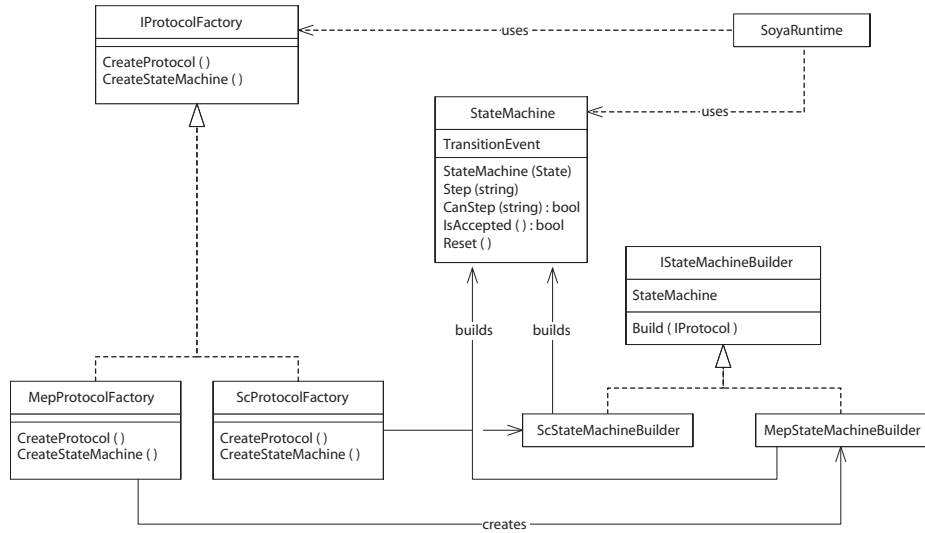


Figure 4.21: Building the Protocol State Machine

# Chapter 5

## Case Study

“ Placet experiri. ”

— *Thomas Mann*

This chapter presents a case study in the creation of a service-oriented system in connection with the Australian lending industry. The services composing the application use an open standard for exchanging data and thus facilitate transactions between lending institutions and valuation firms. We specifically studied two cases of the application and showed how they can be realised with the aid of Soya and SSDL. On the one hand, this enabled us to validate the usability of Soya’s programming model and the proper functioning of its runtime environment. On the other hand, it deepened our understanding of to what extent SSDL fosters the creation, description and execution of Web Services. Finally, by implementing a realistic reference case we did not only verify the applicability of the practices under consideration but also discovered new needs that would possibly have otherwise been missed.

### 5.1 Motivation and Background

In the Australian lending industry, processes such as property valuation are still tedious tasks. A lot of the communication is carried out manually using mail, fax or telephone. A simple task like changing an incorrectly submitted property address can take up to three days and thus unnecessarily slow down the entire valuation process. Because of the low degree of automation, a large number of staff is needed to process tasks which are of a predominantly mechanical nature. Other issues include bad communication between lenders and valuation firms, such as lenders not being informed by valuers of delays in processing a request.

In summary, we can say that the manual system currently used for communication between different parties is costly, time-consuming and unreliable.

### 5.1.1 Lending Industry XML Initiative (LIXI)

The Lending Industry XML Initiative (LIXI) is an independent non-profit organisation that has been established to remove data exchange barriers within the Australian lending industry [93]. Through the work of LIXI, member organisations are able to provide services to their customers more efficiently and at lower cost. This is achieved by establishing an open XML standard for the format and exchange of lending-related data that replaces numerous incompatible and proprietary approaches.

Members of LIXI come from a broad range of companies from across the lending industry. They include major banks, mortgage originators and brokers, mortgage insurers, valuers, settlement agents, trustees and information technology providers. Together, they participate in working groups that shape the standards and direction of LIXI.

LIXI has been used in a case study before [141]. This work focused on WSDL and BPEL and identified issues relating to high complexity and ambiguous semantics using these approaches.

### 5.1.2 Property Valuation Process

Term	Definition
Requestor	Legal entity acting for a lending institution or mortgagor who issues the valuation request.
Valuation Firm	The firm that processes the valuation request.
Valuer	A person acting for a valuation firm who is responsible for the valuation response details.
Intermediary	An entity acting on behalf of multiple valuation firms.
Valuation Request	Contains sufficient information for carrying out the requested valuation.
Valuation Response	The data set returned in response to a valuation request.

Table 5.1: LIXI property valuation terminology.

The property valuation process is a simple workflow that starts with an initial valuation request. The request progresses through a number of processing stages until the valuation is completed and a response is sent back to the requestor. It is in the nature of the business that the period between the valuation request and the subsequent response may span up to several days. During that



time requestors, intermediaries and valuation firms can still communicate with each other by sending messages in either direction. This includes querying and sending updates of the valuation request’s status, cancelling the request, re-negotiating the fees and so forth. Further, the actual processing of the messages often still depends on human activities, such as a manager accepting or rejecting a fee change, a valuer entering valuation data after inspecting a property, etc.

Throughout its lifecycle, a valuation request progresses through several different processing stages. Each stage is defined by LIXI and has a status code that can be used to indicate at which stage a valuation request is within this processing cycle. This is illustrated in Figure 5.1. The corresponding workflow, however, is implemented by a service internally and does therefore not affect the protocol directly.

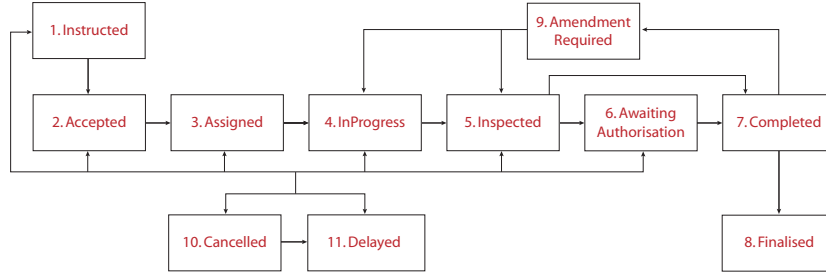


Figure 5.1: The lifecycle of a valuation request.

## 5.2 Approach

The two cases that we studied involved a number of Web Services that communicated with each other by exchanging some predefined messages according to a business protocol. Although the two cases had many common characteristics, they differed from each other in terms of the business protocol’s complexity and the number of involved parties. Notwithstanding, our focus and method differed for each case. During the first case, we concentrated on Soya’s programming model and its runtime behaviour. To do so, we created an executable implementation of the services. Although we captured the protocol using the MEP protocol framework, we were still able to investigate the runtime in a generic way, as a result of its protocol framework independence (see Section 4.8). The focus of the second case was to establish if SSDL’s other protocol frameworks, specifically Sequencing Constraints (SC) [98], were powerful enough to express the given business protocol. Because the development process was the same as in the previous case, we concentrated solely on the protocol description. Since Soya’s programming model does currently only support the MEP protocol framework,

we modelled the protocol description directly in XML. Further, we re-modelled the service description including protocol information using WSDL and BPEL and compared the results.

In general, we aimed at establishing if Soya and SSDL could be used to create the given application and if their use implied significant benefits or drawbacks compared to incumbent approaches. Further, we wanted to find out if Soya and SSDL were sufficiently expressive to capture the relevant aspects of our services; how much effort was required in creating them; if the developed technical solution aligned well with the business case; if employing Soya and SSDL naturally lead to service-oriented application design.

## 5.3 Case One: Property Valuation

### 5.3.1 Focus

The target of the first case in our study was primarily Soya's programming model and runtime environment. Not only did we want to establish if the offered programming abstractions could be used to create the given application, but also if they facilitated the development process and fostered service-oriented design. Further, we wanted to affirm that Soya's runtime behaviour including message processing, contract enactment, dispatching and so forth worked as expected.

### 5.3.2 Case Description

Our first case involved a requestor that communicated directly with a valuation firm. An example of a typical communication between the two services is shown as a sequence diagram in Figure 5.2. First, the requestor sends a valuation request to the valuation firm. Then, after automatically performing some validity checks on the incoming request, the valuation firm assigns it to one of its employees. After the employee has been notified, she decides that the requested fee is not adequate. She hence delays the valuation request and sends a fee change request back to the requestor. The requestor, however, does not accept the fee change request. Therefore, the valuation firm sends yet another fee change request, which is accepted by the requestor this time and thus subsequently also by the valuation firm. After the successful fee-negotiation, the valuation firm informs somebody to have a look at the property and internally updates the status of the request to *in-progress*. Some days later, after the requestor still has not heard anything from the valuation firm, it sends a status enquiry regarding the request. The valuation firm in turn replies that the valuation is *inProgress*.

Finally, the valuer that inspected the property enters the valuation data into the valuation firm's system. Internally, this updates the status of the request to *inspected*. After all the valuation data has been entered, the system updates the status to *completed* and sends the valuation response to the requestor. Once the response has been sent, the status of the request is updated one more time to *finalised*. On the requestor side, the service receives the valuation data, which concludes the conversation between the two services.

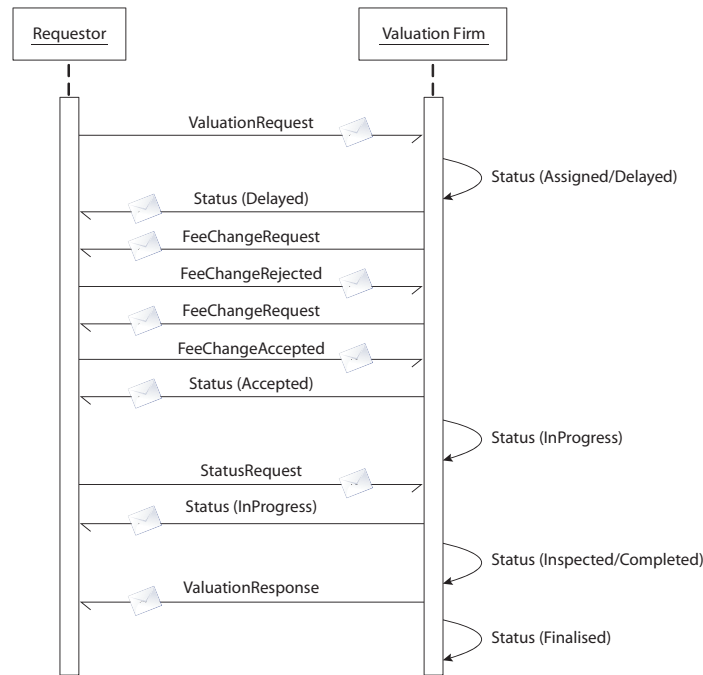


Figure 5.2: A typical conversation between a requestor and a valuation firm.

Of course, the described conversation is one of many such possible conversations. Figure 5.3 represents the actual business protocol between requestor and valuation firm in a more generic way as a state machine. Although the shown diagram represents the view of the incoming and outgoing messages from the requestor's point of view, the valuation firm's perspective can be derived by simply inverting the direction attributes on the state transitions.

The protocol defines that the requestor sends an initial valuation request, which might or might not be acknowledged with a *Status* message. At this point, a number of possible interactions can happen. The requestor can, for example, receive a *FeeChangeRequest*, which must be either accepted or rejected. Further, the requestor might enquire about the request's status, in which case the valuation firm must reply. Moreover, the valuation firm can send any number of *Status* updates at arbitrary intervals. In addition, these interactions are all

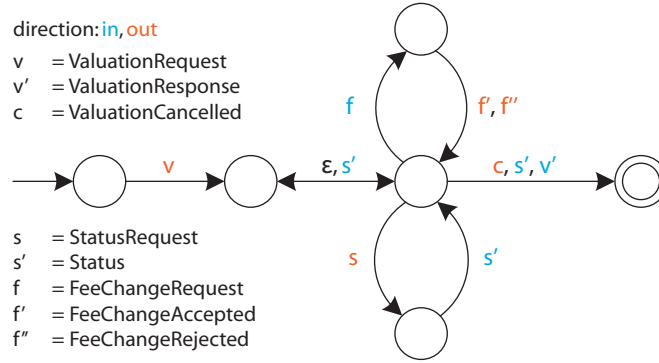


Figure 5.3: The valuation business protocol from requestor's perspective.

optional and may be repeated and intermixed in an arbitrary order. Finally, the conversation is terminated at some point, if either the requestor cancels the request, the valuation firm declines the request (this corresponds to the  $s'$  transition to the final state, because the firm actually sends a *Status* message) or if the inspection was performed successfully and a *ValuationResponse* is sent back to the requestor.

### 5.3.3 Protocol Translation

Before we could implement the business protocol, we needed to translate it into a form that could be captured with the MEP protocol framework. Since the valuation protocol was not available in a formal or machine-processable form, we had to do the transformations by hand. In fact, even the state machine shown in Figure 5.3 was derived manually from the LIXI specification. First, we identified the messages and their direction (i.e. the state transitions). Next, we correlated them into the MEPs shown in Table 5.2. Finally, we did the same for the valuation firm, by simply inverting the direction attributes on the patterns.

Pattern	Messages
out-optional-in	out: ValuationRequestMsg, in: StatusMsg
in-only	in: StatusMsg
out-in	out: StatusRequestMsg, in: StatusMsg
in-out	in: FeeChangeRequestMsg, out: FeeChangeRejectedMsg
in-out	in: FeeChangeRequestMsg, out: FeeChangeAcceptedMsg
in-only	in: ValuationResponseMsg
out-only	out: CancelValuationMsg

Table 5.2: Message exchange patterns derived from valuation process protocol.

By combining the described MEPs into a state machine according to their definition in Figure 4.7, we obtained the state machine depicted in Figure 5.4. Inevitably, some protocol information was lost in the translation process as a result of the insufficient expression power of the MEPs. Consequently, messages could have been dispatched in the course of a conversation that our application code did not expect, because they were not specified. Similarly, our implementation could have sent outgoing messages that did not adhere to the business protocol.

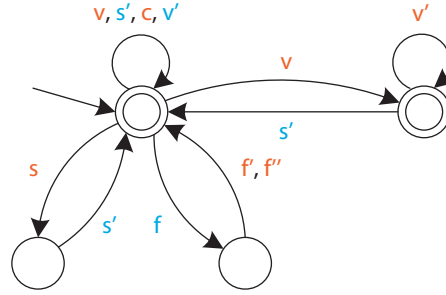


Figure 5.4: The actual protocol defined by the MEP patterns in Table 5.2.

The defined MEP protocol is hence just an approximation of the actual business protocol (see Figure 5.5). As a matter of fact, it can easily be shown that the implemented protocol is less restrictive than the original because it does not succeed in modelling the exclusion of all *disallowed* interactions. For example, it is possible that the requestor sends a *StatusRequestMsg* even though it has not sent a *ValuationRequestMsg* previously.

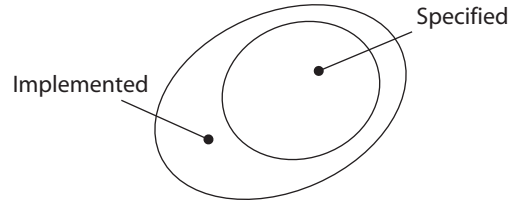


Figure 5.5: Allowed interactions of business protocol compared to implemented MEP protocol.

The cause for this mismatch is the fact that the eight simple patterns defined in the MEP protocol framework were not sufficient to express the required business protocol in an accurate manner. In fact, these patterns do not allow capturing message exchanges that span more than two messages (one incoming and one outgoing). Therefore, we could not model the fact that a valuation request, for example, might be followed by some status and fee negotiation messages and then by a valuation response, because it involved more than two

messages.

This is not a limitation of Soya's programming model or SSDL per se, but a restriction of the MEP protocol framework. As a matter of fact, we will see in the second case of our study that other SSDL protocol frameworks can be used to represent business protocols more accurately.

### 5.3.4 Development Process

#### 5.3.4.1 Message Definition

All the messages that can be exchanged by the requestor and the valuation firm were defined in an XML schema provided by LIXI. Instead of creating the data contracts according to the schema manually, we used it to generate the data contract classes using a configurable .NET [142] code generator. Although the code generator produced acceptable results, we still had to make manual adjustments to the generated code. For example, it was not possible to specify that a data member of a data contract should be serialised as an XML attribute, because the generator did not support this kind of XML projection. Thus, schemas that required the use of XML attributes were not supported per se. The following simple LIXI XML schema element, for example, could not be represented as a data contract, because it defined an XML attribute:

---

```
<xs:complexType name="PropertyType">
  <xs:attribute name="Name" use="required" type="xs:string"/>
</xs:complexType>
```

---

Nevertheless, the generator could be configured to output types that use a custom formatting implementation for serialisation and deserialisation. Consequently, these types could store anything that could be represented in XML. Although this solved the XML attribute problem, the generated code was not particularly useful for our application because it was too generic. In fact, by inferring XML schema from the generated type, we obtained a schema that represented the XML schema type *anyType*.

For this reason, we introduced a strongly typed member that represented the XML attribute and provided a custom implementation for reading and writing to and from XML respectively. Further, we implemented a method that loaded the XML schema for the implemented type according to its definition in the LIXI schema. By using the described approach we achieved the required mapping. Indeed, this method is very powerful in general and can be used to represent even the most complex XML data structures.

After generating the data contracts, we created the message contracts. Unlike generating the former, there was no support for creating the latter from

an XML schema. Therefore, we created the message contracts manually. Since LIXI did not define its messages with respect to SOAP (i.e. headers and bodies), we included the defined LIXI messages without modification in the body of the message contracts. Furthermore, we included the valuation request id in the header of most message contracts, so that it would be transmitted in the SOAP header in addition to the main payload in the SOAP body. On the application-level this allowed us to correlate incoming messages with the respective valuation request. Correlating messages into one continuous conversation, however, was not done using this application header but automatically handled by Soya through means of WS-Addressing headers. Nevertheless, we found it useful to have direct access to the valuation request id and by putting it in the SOAP header, we did not have to change the LIXI XML schema. Since we only had to create eight message classes, the associated effort was insignificant. Yet given the simple but tedious nature of the task, we hold the opinion that it would be useful to write a small code generator if more classes needed to be generated. At the same time, this would also help keeping the classes synchronised with the XML schema.

#### 5.3.4.2 Protocol Definition

Because we had translated the protocol into MEPs earlier, we only needed to apply them as C# attributes to the service contract. This process was relatively straightforward as we just defined service operations for processing incoming messages and decorated them with the protocol metadata. While creating the service interface and the attributes, we were able to focus solely on the service contract. As a result, we did not have to be concerned about implementation details such as, for example, operation names. Figure 5.6 shows the requestor's service contract.

The top-most attribute declares that a *CancelValuationMsg* can be sent at any time. It is attached directly to the interface because it is an *OutOnly* attribute and hence does not require an associated method for processing an incoming message. The metadata attached to the first method defines that a *StatusMsg* can be received in an unsolicited manner (*InOnly*), compulsorily in response to a *StatusRequestMsg* (*OutIn*) or optionally in response to a *ValuationRequestMsg* (*OutOptionalIn*). The attributes on the second method state that a *FeeChangeRequestMsg* can be received at any time but must be followed by an outgoing *FeeChangeAcceptedMsg* or *FeeChangeRejectedMsg*. Finally, the last method defines that a *ValuationResponseMsg* can be received at any time.

---

```

[Mep(Style=MepStyle.OutOnly, Out=typeof(CancelValuationMsg))]
public interface IRequestorService {
    [Mep(Style=MepStyle.InOnly)]
    [Mep(Style=MepStyle.OutIn, Out=typeof(StatusRequestMsg))]
    [Mep(Style=MepStyle.OutOptionalIn, Out=typeof(ValuationRequestMsg))]
    void Process(StatusMsg msg);

    [Mep(Style=MepStyle.InOut, Out=typeof(FeeChangeAcceptedMsg))]
    [Mep(Style=MepStyle.InOut, Out=typeof(FeeChangeRejectedMsg))]
    void Process(FeeChangeRequestMsg msg);

    [Mep(Style=MepStyle.InOnly)]
    void Process(ValuationResponseMsg msg);
}

```

---

Figure 5.6: The service contract specified using Soya’s programming model and the MEP protocol framework.

#### 5.3.4.3 Contract Implementation

Subsequently, we implemented the defined service interface. In our case study, the requestor and valuation firm both used humans to process the requests and only a small part of the entire process was actually handled automatically. Apart from storing and retrieving valuation requests to and from the data tier, our service implementation therefore mainly delegated the requests by sending notifications to employees. The employees in turn interacted with the system through interactive user interfaces.

Both sides used Soya’s client API to send messages. In case the request was processed automatically (e.g. a status query), the service implementation handled the request automatically and potentially sent a response immediately from within the service method. This is illustrated in the following example which shows how a status request that is handled automatically by the valuation firm:

---

```

public void Process(StatusRequestMsg msg) {
    string id = msg.StatusRequest.Id;
    ValuationRequest req = ValuationRequestDAO.Instance.Load(id);
    SoyaServiceHost.Reply(Commons.ConvertToMsg(new StatusMsg(req.Status)));
}

```

---

If, however, the service operation required human interaction, we normally persisted the client for future reference and in order to free resources. Later, it was loaded back into memory again and used to continue existing conversations and send outgoing messages. This is illustrated in the following code snippet, which sends the valuation response to the requestor, after all the data had been entered by a valuer:



---

```
ValuationResponse res = ... // created from data entered by valuer
SoyaClient client = SoyaClientDAO.Instance.Load(requestId);
client.Send(Commons.ConvertToMessage(new ValuationResponseMsg(res)));
```

---

#### 5.3.4.4 Configuration and Deployment

The final step in setting up our case consisted in specifying the (HTTP) endpoints for both services in their respective configuration files. Because both requestor and valuation firm could send messages freely in either direction, they were required to have public endpoint addresses. In the requestor's configuration we additionally had to define the endpoint address of the valuation firm, because the requestor needed this information in order to send the first message. Conversely, the valuation firm did not need to know the endpoint address of the requestor in advance, because it retrieved this information from the *ReplyTo* header contained inside the messages it received. The corresponding inferred SSDL contract can be found in Appendix 7.3.

#### 5.3.4.5 Execution

Finally, we executed the two services. The code snippet below shows how the requestor creates and opens its service host on line 1 and 2, respectively. It then acquires a client from that host on line 3. This client is linked to the host in the sense that it adds the host's endpoint address to the *ReplyTo* header of the messages it sends. Further, it represents a reference to the conversation between the requestor and valuation firm and can thus later be used to continue the existing conversation. The endpoint address of the valuation firm is automatically retrieved from a configuration file, but could also have been passed into the *GetClient()* method. The valuation request is created on line 4 by using input data from a valuer. Finally, on line 5, the message is sent to the valuation firm.

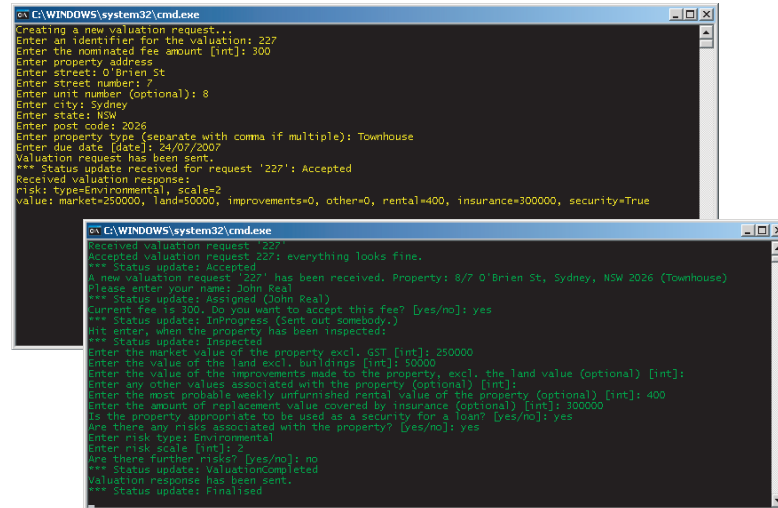
---

```
1 SoyaServiceHost host = new SoyaServiceHost(typeof(RequestorServiceImpl));
2 host.Open();
3 SoyaClient client = host.GetClient();
4 Message valuationRequest = ...
5 client.Send(valuationRequest);
```

---

As we have already mentioned, most of the processing of the requests on either side involved human interaction. As a result, we created a simple user interface that allowed requestors and valuation firms to interact with the system. This is shown in Figure 5.7. The human input either directly triggered new message exchanges (e.g. accepting a request immediately sent a *Status* up-

date message) or was used as input in future message exchanges (e.g. filling in valuation response data).



```

C:\WINDOWS\system32\cmd.exe
Creating a new valuation request...
Enter an identifier for the valuation: 227
Enter the nominated fee amount [int]: 300
Enter property address
Enter street: O'Brien St
Enter street number: 7
Enter unit number (optional): 8
Enter city: Sydney
Enter state: NSW
Enter post code: 2026
Enter property type (separate with comma if multiple): Townhouse
Enter due date [date]: 24/07/2007
Valuation request has been sent.
*** Status update received for request '227': Accepted
Received valuation response:
risk: type=Environmental, scale=2
value: market=250000, land=50000, improvements=0, other=0, rental=400, insurance=300000, security=True

C:\WINDOWS\system32\cmd.exe
Received valuation request '227':
Accepted valuation request '227': everything looks fine.
*** Status update: Accepted
A new valuation request '227' has been received. Property: 8/7 O'Brien St, Sydney, NSW 2026 (Townhouse)
Please enter your name: John Real
*** Status update: Assigned (John Real)
Current fee is 300. Do you want to accept this fee? [yes/no]: yes
*** Status update: InProgress (Sent out somebody.)
At enter: when the property has been inspected:
*** Status update: Inspected
Enter the market value of the property excl. GST [int]: 250000
Enter the value of the land excl. buildings [int]: 50000
Enter the value of the improvements made to the property, excl. the land value (optional) [int]:
Enter any other values associated with the property (optional) [int]:
Enter the most probable weekly unfurnished rental value of the property (optional) [int]: 400
Enter the amount of replacement value covered by insurance (optional) [int]: 300000
Is the property appropriate to be used as a security for a loan? [yes/no]: yes
Are there any risks associated with the property? [yes/no]: yes
Enter risk type: Environmental
Enter risk scale [int]: 2
Are there further risks? [yes/no]: no
*** Status update: ValuationCompleted
Valuation response has been sent.
*** Status update: Finalised

```

Figure 5.7: Interactive console session between requestor (back) and valuation firm (front).

## 5.3.5 Discussion

### 5.3.5.1 Simplicity & Development Effort

Because we were given an XML schema, we were able to generate the data contracts without significant effort. The only time-consuming task at this stage was providing a custom serialisation and deserialisation implementation for types using XML attributes, because the data contract generator was not able to handle this situation. Unfortunately, the required effort for accomplishing this relatively simple and common task was disproportional. Considering that XML attributes are an essential part of most XML documents, we think that appropriate mechanisms for capturing them in data contracts or better tool support for generating source code should be provided in future. Still, the task was mechanical to a considerable extent and we indeed believe that by leveraging the generator, it could easily be automated. The same is true for creating the message contract classes. It is a mechanical task and thus a good candidate for being executed by a code generator.

Applying the extracted patterns to the service contract using Soya's programming model was again an almost mechanical task. We just had to look at the extracted patterns and identify the set of incoming messages types. Then we wrote one service method signature for each distinct type and applied the

MEP to it. In case two or more patterns shared the same input message type, we just added multiple MEP attributes to the service method in any order. Again, because of the mechanical and straightforward nature of this task, we think that it could also be automated by a tool.

### 5.3.5.2 Service-Orientation

All through the development process, the main abstraction we were working with was the message. The data structures defined in LIXI were used as payload in messages. The interaction protocol was defined using MEPs which in turn were defined using messages. The service interface was defined using messages. And finally the data structures that our service implementation received and sent were messages. Therefore, we never had to give up this message-oriented perspective.

To contrast this with an operation-centric approach, we will consider two examples from our case study and compare how we would have implemented them in an operation-centric solution. According to our protocol a *StatusRequestMsg* must be followed by a *StatusMsg*. This corresponds to an *in-out* MEP. If we would correlate the message using operations, we could have defined an operation like, for example, *GetStatus(StatusRequestMsg msg)* returning a *StatusMsg*.

In the above case, this approach makes sense. Notwithstanding, it is limited to these kind of simple scenarios. By looking at another part of the protocol, we see that a *FeeChangeRequestMsg* must be responded to with either a *FeeChangeAcceptedMsg* or a *FeeChangeRejectedMsg*. Because this part of the protocol can have two possible outcomes, we can no longer represent it using the operation-abstraction. Although there are workarounds to this problem, they have disadvantages.

We could, for example, use a one-way operation and provide a callback interface through which we can send either of the two messages. Yet by doing this, we are introducing a new API (the callback API) and mixing it with the operation API. More importantly, we lose the information of what can actually be sent back after a *FeeChangeRequestMsg* has been received. This fact is no longer visible from the service interface or description and hence requires both service developers and service consumers to know interaction semantics defined outside the scope of the machine-processable format.

Another solution is to abstract the two return types into a common type (e.g. *FeeChangeResponseMsg*) and introduce an attribute that facilitates distinguishing between the two former types (e.g. *type=accepted/rejected*). As a result, we would have only one response message type and could therefore map it to an

operation. Unfortunately, this approach necessitates changing the LIXI XML schema or creating an intermediary schema. Further, it requires additional code inside the application logic that parses the message content in order to find out whether it needs to execute the logic that is responsible for handling either an accepted or a rejected fee change request.

In both solutions of the operation-centric approach, the changes made to the implementation are propagated to the outside world and have an effect on the web service's contract (i.e. service description including XML schema), violating service encapsulation and service-oriented design principles.

Conversely, Soya and SSDL forced us to think only in terms of message exchanges throughout the development process, which naturally led to a service-oriented design.

#### 5.3.5.3 Soya Programming Model & Runtime

Soya's programming model allowed us to express the complete MEP protocol with 7 lines of code. Because the *MEP* attributes in Soya directly reflect all the existing MEPs, we had no difficulties representing them in code. While implementing the case for our study, however, we wanted the service to use the LIXI XML schema instead of using the one inferred from the data contracts. Initially, this was not possible in Soya. As a result, we implemented this additional feature and added it to Soya's codebase.

The runtime environment worked as expected. Upon deployment, the inferred SSDL contract reflected the MEPs that we had specified using the C# attributes and could be accessed via HTTP. The services communicated successfully with each other via the specified endpoints. The incoming messages were properly validated against the schema and message definitions in the contract and dispatched to the correct local service methods.

#### 5.3.5.4 MEP Protocol Framework

We mentioned that we formalised the business protocol into a state machine representation. Although the MEP description could then be derived in a fairly direct manner, it still was a manual *trial and error* process that did not follow any formal schemes. It required us to mentally partition and re-think the protocol into MEPs and therefore to translate a business protocol into a technical representation of it. The resulting description did therefore not only miss some aspects of the protocol, but was also dissimilar to the original.

## 5.4 Case Two: Intermediary

### 5.4.1 Focus

In the previous section we have shown that Soya's programming model simplified the development process and that its runtime environment could be used to execute SSDL-based services. Yet we also concluded that the MEP protocol framework was not expressive enough to model the given business protocol. In the case presented in this section we therefore wanted to establish if SSDL's more powerful Sequencing Constraints (SC) protocol framework [98] could represent the protocol more accurately.

Since the second case differed from the first only in terms of its business protocol, we focused solely on this aspect. Because Soya currently does not offer mechanisms to capture SC protocols, we created the protocol description directly in XML. Although this procedure did not directly involve Soya, it was an important step in determining the future direction of both Soya and SSDL. By showing that SC can be used to model even complex protocols, we would have strong reasons justifying the effort to implement direct programming support for this protocol framework in Soya.

Further, we re-implemented the given case using WSDL and BPEL in order to compare it against SSDL and SC in terms of expressiveness, complexity, ease of use and service-orientation.

### 5.4.2 Case Description

Although the XML schema, the transferred messages, the business goal and so forth of the second case in our study was the same as before, the business protocol was more complex and involved an additional service party. The requestor did not communicate with a valuation firm directly, but instead with an intermediary. The intermediary in turn communicated and negotiated with an arbitrary number of valuation firms. Based on the incoming request and some business criteria, the intermediary selected the most appropriate valuation firm from a list of potential candidates. Because, ultimately, only one firm performed the valuation, the intermediary had to ensure that it cancelled pending valuation requests accordingly.

Figure 5.8 illustrates this scenario using an example. The requestor sends a valuation request to the intermediary. Based on the incoming request (e.g. postcode, due date, etc.) the intermediary retrieves a list of potential valuation firms from an internal source and forwards the request to each one. In the given example, the second valuation firm accepts the request immediately and thus gets the order. Nevertheless, at this point the valuation firms could have

also sent requests back to the intermediary to re-negotiate the fees. Essentially, the possible message exchanges between intermediary and valuation firms are the same as the ones in the previous section between requestor and valuation firm. As soon as the intermediary receives the acceptance message, it cancels the other two pending requests with valuation firm *one* and *three*. Apart from the no longer present fee negotiation, the behaviour between the requestor and the intermediary is also the same as in the previous section between requestor and valuation firm. Consequently, the requestor can, for example, send status enquiries to the intermediary at any time. As soon as the intermediary receives the valuation response from the chosen valuation firm, it forwards it to the requestor, which concludes the conversation.

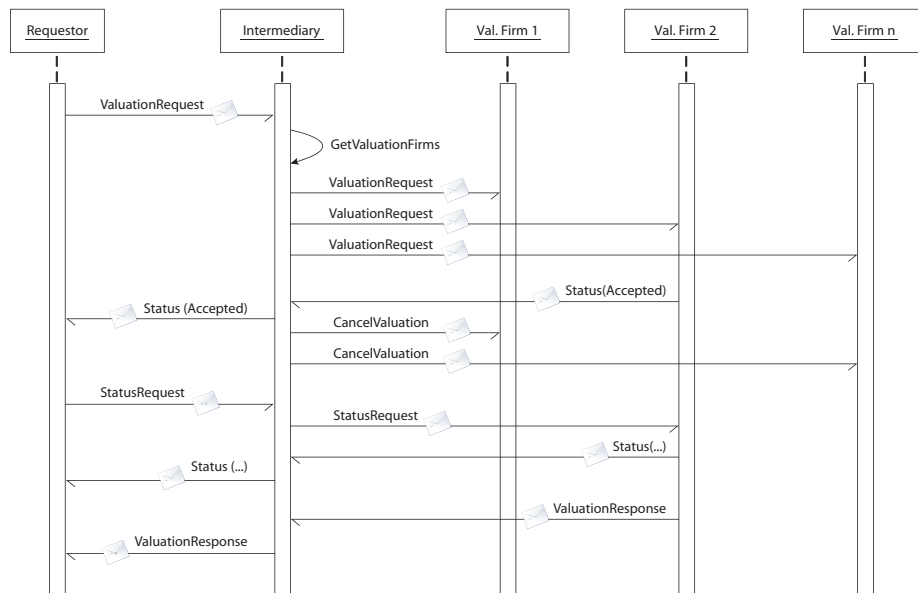


Figure 5.8: A typical conversation between a requestor, intermediary and a number of valuation firms.

Modelling the illustrated scenario posed a number of challenges. First, the intermediary had to communicate with two different parties and hence distinguish between conversations. Second, the interactions with the requestor and the valuation firms were required to occur concurrently, but still demanded for some level of synchronisation. Third, the intermediary needed to communicate with multiple instances of valuation firms concurrently. Fourth, the number and physical locations of the valuation firms had to be determined at runtime in a dynamic fashion.

As a result, we could no longer clearly represent the interactions using a state machine the way we did in the previous sections. Nevertheless, we abstracted

and summarised the intermediary's protocol in Figure 5.9. Essentially, the main diagram represents the communication with the requestor. The red square box in the middle stands for the interactions with the valuer. As we have just described, these interactions run in parallel to the ones with the requestor. Additionally, an arbitrary number of these interactions can execute concurrently.

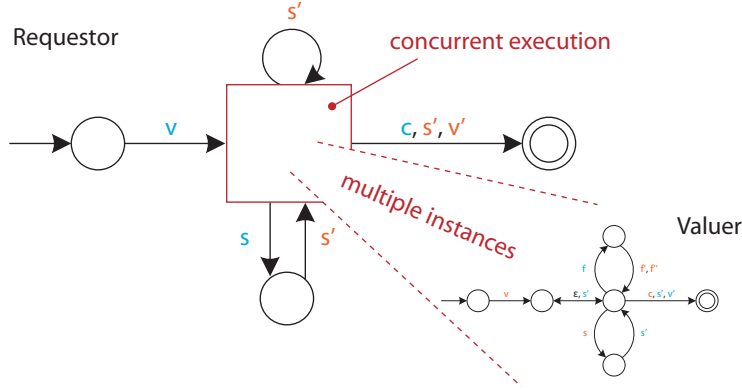


Figure 5.9: The intermediary's protocol. The red box can fork an arbitrary number of state machines that run concurrently to the main one and each other.

### 5.4.3 Sequencing Constraints

As we can see, the intermediary's interaction behaviour was rather complex. Surprisingly, the SC protocol framework allowed describing this behaviour in a fairly concise way. Unfortunately, however, it does not provide native support for representing iterative structures, which required us to translate these structures into recursive representations.

#### 5.4.3.1 Representing Cycles

In our first trial to model the cycles in the communication between the intermediary and a valuation firm, we used a *multiple* element. The semantics of this element are defined analogously to the *Replication* (written  $!$ ) construct in  $\pi$ -calculus [97]. Unfortunately, this implies that its children can occur a multiple number of times in parallel [98], which leads to the state machine shown in Figure 5.10. As we can see, this is a different, and in fact less restrictive state machine than the one defined by the business protocol. As it turned out, we could not use the *multiple* element for modelling the iterative structures because several instances of a structure could have occurred concurrently and thus potentially interleaved with each other in a capricious manner.

In our second attempt, we expressed the iterations using recursive references

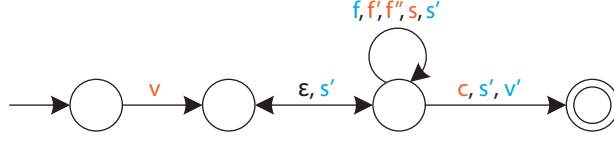


Figure 5.10: Graphical representation of SC protocol using *multiple* element.

to sub-protocols. To illustrate this better, we will, for now, consider only the part of the business protocol defining that a *StatusRequestMsg* must be followed by a *StatusMsg* and that this pattern can occur zero or more times. By ignoring the direction attribute of the transition symbols, we can rewrite this part in a compact way using the following regular expression:

$$(ss')^*$$

Because the rules of regular languages are more restrictive than the productions of context-free languages [138] we can also substitute the above language using a context-free grammar:

$$X \rightarrow ss'X \quad | \quad \epsilon$$

Based on the observation that not only is XML a context-free language [143] but that it can be used also for describing its own structure [73], we represented the above grammar in XML. Of course, an associated XML schema might disallow such a definition. In the case of SSDL's SC protocol framework, however, this was possible. The code in Figure 5.11 shows how we modelled the above grammar. The key is the *sc:protocolref* element that allows for creating arbitrary cycles by referencing itself. Also note the *sc:nothing* element that corresponds to  $\epsilon$  above and guarantees that the protocol can exit the cycle. Consequently, we partitioned the entire protocol into sub-protocols<sup>1</sup> as required by the iterative structures and then referenced them in this recursive fashion.

By using this approach, we rewrote the complete business protocol between intermediary and valuation firm with the following regular expression:

$$v(s'|(f(f'|f''))|(ss'))^*(s'|c|v')$$

Similarly to above, we then represented it using the context-free grammar below:

<sup>1</sup>Since the SC specification does not clearly define the semantics of multiple *sc:protocol* elements, we assumed that the first one is considered as the only entry point to the protocol.



---

```

<sc:protocol name="recurse">
  <sc:choice>
    <sc:sequence>
      <ssdl:msgref ref="StatusRequestMsg" direction="out"/>
      <ssdl:msgref ref="StatusMsg" direction="in"/>
    </sc:sequence>
    <sc:protocolref ref="recurse"/>
    <sc:nothing/>
  </sc:choice>
</sc:protocol>

```

---

Figure 5.11: A loop specified in the SC protocol framework.

$$\begin{aligned}
X &\rightarrow vAB \\
A &\rightarrow A'A \mid \epsilon \\
A' &\rightarrow s' \mid fA'' \mid ss' \\
A'' &\rightarrow f' \mid f'' \\
B &\rightarrow s' \mid c \mid v'
\end{aligned}$$

As we can see,  $A$  is defined recursively. We therefore defined a separate protocol for that part. The protocol description that we obtained by translating the above grammar is depicted in Figure 5.12. Note that there are two separate protocols (line 1 and 12). Further note the recursive calls on line 4 and 26.

#### 5.4.3.2 Protocol Definition

After identifying and translating the iterative structures, we described the remainder of the protocol. Although the complete protocol is not listed in Figure 5.13 due to its verbosity and similarity with previously presented parts, it can be found in its entirety in Appendix 7.3. The keys in modelling the desired protocol were the *sc:participant* elements (abbr. *sc:p*) on line 1 and 2, the *sc:parallel* element on line 5 and the *sc:multiple* element on line 7.

The *sc:participant* element allowed us to distinguish between interactions with the requestor and those with the valuation firms. The *sc:parallel* element was used to model that the intermediary and the requestor could communicate concurrently to the conversations between the intermediary and the valuation firms. This corresponded to the interactions that go to and from the red square box in Figure 5.9. Finally, the *sc:multiple* element made it possible to capture the fact that the intermediary might have conversations with more than one valuation firm at the same time.

---

```

1 <sc:protocol name="intermediary-valuer">
2   <sc:sequence>
3     <ssdl:msgref ref="m:ValuationRequestMsg" direction="out"/>
4     <sc:protocolref ref="recurse"/>
5     <sc:choice>
6       <ssdl:msgref ref="StatusMsg" direction="in" />
7       <ssdl:msgref ref="CancelValuationMsg" direction="out" />
8       <ssdl:msgref ref="ValuationResponseMsg" direction="in" />
9     </sc:choice>
10  </sc:sequence>
11 </sc:protocol>
12 <sc:protocol name="recurse">
13   <sc:choice>
14     <ssdl:msgref ref="m:StatusMsg" direction="in" />
15     <sc:sequence>
16       <ssdl:msgref ref="m:FeeChangeRequestMsg" direction="out" />
17       <sc:choice>
18         <ssdl:msgref ref="m:FeeChangeAcceptedMsg" direction="in" />
19         <ssdl:msgref ref="m:FeeChangeRejectedMsg" direction="in" />
20       </sc:choice>
21     </sc:sequence>
22     <sc:sequence>
23       <ssdl:msgref ref="StatusRequestMsg" direction="out" />
24       <ssdl:msgref ref="StatusMsg" direction="in" />
25     </sc:sequence>
26     <sc:protocolref ref="recurse"/>
27     <sc:nothing/>
28   </sc:choice>
29 </sc:protocol>

```

---

Figure 5.12: The protocol between intermediary and valuation firm described using SC.

---

```

1 <sc:participant name="Req"/>
2 <sc:participant name="Val"/>
3 <sc:protocol name="Intermediary">
4   <ssdl:msgref ref="m:ValuationRequestMsg" direction="in" sc:p="Req"/>
5   <sc:parallel>
6     <!-- interactions with requestor -->
7     <sc:multiple>
8       <!-- interactions with valuation firm(s) -->
9     </sc:multiple>
10  </sc:parallel>
11  <sc:choice>
12    <ssdl:msgref ref="StatusMsg" direction="out" sc:p="Req"/>
13    <ssdl:msgref ref="CancelValuationMsg" direction="in" sc:p="Req"/>
14    <ssdl:msgref ref="ValuationResponseMsg" direction="out" sc:p="Req"/>
15  </sc:choice>
16 </sc:protocol>

```

---

Figure 5.13: The intermediary's protocol captured using SC.

While we modelled the above case, we observed two interesting aspects or potential problems which are related to the communication with multiple valuation firms. The first one was the positive realisation that we could describe a protocol that involved a number of valuation firms which were not known at runtime. This is possible because at design-time, the contract is not bound to a particular valuation firm instance, i.e. a particular endpoint. Based on our experience with Soya and the features of current workflow engines [144] we can safely assume that setting or modifying endpoint addresses at runtime can be facilitated by SSDL runtimes. Of course, the valuation firms must all adhere to the same contract in terms of message types and their relative order of exchange, but their physical location is of no relevance to the contract per se. In fact, the SSDL specification does not define how physical endpoints should be mapped to participants. We therefore assumed that an SSDL implementation could map one participant to multiple physical endpoints. In our case, the participant *Val* would have been mapped to the endpoint addresses of valuation firm *one*, *two* and *three*.

Unfortunately, however, the semantics of the *sc:multiple* element allowed its children to interleave with each other quite arbitrarily as we have already established earlier. Of course, on the one hand, this was exactly what we wanted (i.e. having conversations with multiple valuation firms at the same time) and assumed that middleware would take care of this, by distinguishing among conversations with different valuation firms. On the other hand, the mentioned element also allowed multiple concurrent invocations within a single conversation with one valuation firm. Consequently, some interleaving could have occurred. Essentially, this was the same problem that we encountered in the previous section and solved using the recursive *sc:protocolref* call. Yet in this case this was not possible, because in addition to modelling the loop, its iterations actually had to take place at the same time in parallel.

#### 5.4.4 Discussion

Despite the SC protocol framework's lightweight and small choice of constructs, we succeeded in modelling all the aspects of the given business protocol to a satisfactory degree. The procedure to obtain the final protocol description, however, involved some effort. First, translating the iterative structures into their respective recursive representations did not only require re-thinking and additional development effort, but also resulted in a description that was dissimilar and less readable than the original. We believe that the protocol description could have been more similar to the actual business protocol, if we could have expressed the iterations using, for example, a *while* loop. Although it would

not increase the expression power of the language per se, we hold the opinion that adding some new carefully chosen elements could produce more natural, understandable and concise protocol descriptions. Second, in order to describe the protocol between intermediary and valuation firms, which we represented on less than half a line using a regular expression, we needed about 30 lines of XML code using SC. Still, we are aware that this comparison needs to be put into perspective to some extent for the general case, considering that the SC protocol framework can express behaviours that regular expressions cannot (e.g. concurrency, multi-party conversation, message flow direction, etc.). As a matter of fact, we will see later that other workflow languages such as BPEL produce even more verbose protocol descriptions.

We have mentioned that our final protocol description allowed for some arbitrary interleaving. From a practical perspective, however, we did not consider this to be a problem of major significance in our case. First, because the initial message of the potentially interleaving sequence was outgoing, meaning that our own service implementation would have had to incorrectly send two valuation requests to the same valuation firm in order to trigger this faulty behaviour. Second, because we believe that this problem can be solved on the middleware level. For example, an SSDL engine could be instructed to not create duplicate conversations for the same logical connection. If we consider our previous mapping example again, where the participant *Val* was mapped to valuation firm *one*, *two* and *three*, the middleware could ensure that at most three conversations (i.e. one for each endpoint) are forked off the main process.

Still, we do not think that it is advantageous if the protocol description attempts to model every detail of the business protocol. Indeed, the word *to model* means “*to devise a simplified description of a system*” [145]. For example, our protocol model did not express the fact that only one valuation firm gets the order while the others are cancelled. Hence, a more precise model could have possibly captured this feature and perhaps even expressed that exactly  $n-1$  cancel requests are sent to  $n$  valuation firms after the valuation has been accepted by one firm. Nevertheless, we do not think that this degree of accuracy in a service description would be of great benefit to other services. On the contrary, it would increase the complexity of the description and thus make it harder for service consumers to see the essential features of the protocol. Of course, we realise that deciding what should be included in the service description and what should be omitted can vary from case to case and is therefore very difficult to determine in a generic way. In our case, however, the expression power provided by the SC protocol framework was sufficiently apt and adequate.

### 5.4.5 Comparison with WSDL & BPEL

To get a better understanding of the strengths and weaknesses of SSDL compared to the incumbent approaches, we re-modelled the case presented in this section using WSDL and BPEL. We used WSDL to define the service interfaces and BPEL to model the intermediary's protocol.

On the one hand we selected BPEL because it has emerged as the de facto standard for describing workflows. On the other hand we chose it because in addition to modelling *executable* processes, it can also be used to describe interaction protocols as so called *abstract* processes. The idea of abstract processes is to describe the message exchange behaviour of a Web Service without revealing details about its internal behaviour. As a consequence, abstract processes contain less information than their executable counterparts and can for that reason not be executed. Since our aim was not to implement the intermediary service using BPEL, but to capture its messaging, we used BPEL's abstract notation for modelling it.

#### 5.4.5.1 WSDL

First, we created the WSDL files for the requestor, the intermediary and the valuation firms as all three descriptions were needed for writing the BPEL process. Instead of writing the WSDL files manually, we defined the service interfaces in C# and used WCF to generate the desired artefacts. Since the XML schema was the same as in the previous experiments, we re-used the data and message contracts we had created earlier. The intermediary service's C# interface is shown in Figure 5.14. Instead of using Soya's *Mep* attributes, however we used the *OperationContract* attributes provided by WCF. Because the intermediary's protocol is based mostly on interactions that span more than two messages, we were unable to capture them using WSDL's operation-centric model. As a result, we modelled all interactions except one as one-way operations. By doing so, we inevitably lost most of the information about the protocol. In a similar way, we specified the requestor and the valuation firms interfaces. Then, we generated the WSDL files.

Next, we added the relationships between the intermediary, requestor and valuer to the intermediary's WSDL file as shown in Figure 5.15. Although this created dependencies between the intermediary's service description and the other parties, we had to add this information as we referred to it later on from the BPEL description.

---

```

public interface IIntermediaryService {
    [OperationContract(Name="ProcessValuationRequest", IsOneWay=true)]
    void Process(ValuationRequestMsg msg);

    [OperationContract(Name="ProcessStatusRequest")]
    StatusMsg Process(StatusRequestMsg msg);

    [OperationContract(Name="ProcessCancelValuation", IsOneWay=true)]
    void Process(CancelValuationMsg msg);

    [OperationContract(Name="ProcessStatus", IsOneWay=true)]
    void Process(StatusMsg msg);

    [OperationContract(Name="ProcessFeeChangeRequest", IsOneWay=true)]
    void Process(FeeChangeRequestMsg msg);

    [OperationContract(Name="ProcessValuationResponse", IsOneWay=true)]
    void Process(ValuationResponseMsg msg);
}

```

---

Figure 5.14: The intermediary's service interface defined using WCF's programming model and C# attributes.

---

```

<wsdl:definitions
  targetNamespace="http://www.lixi.org/Valuation/wsdl/Intermediary"
  xmlns:r="http://www.lixi.org/Valuation/wsdl/Requestor"
  xmlns:v="http://www.lixi.org/Valuation/wsdl/Valuer">
  ...
  <plnk:partnerLinkType name="RequestorPartnerLink">
    <plnk:role name="Requestor" portType="r:RequestorService"/>
    <plnk:role name="Intermediary" portType="IntermediaryService"/>
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="ValuerPartnerLink">
    <plnk:role name="Valuer" portType="v:ValuerService"/>
    <plnk:role name="Intermediary" portType="IntermediaryService"/>
  </plnk:partnerLinkType>
</wsdl:definitions>

```

---

Figure 5.15: The partner link definitions added to the intermediary's WSDL file defines the relationship to the requestor and valuation firm.

### 5.4.5.2 BPEL

The challenges in expressing the protocol using BPEL were the same as before: Communication with multiple parties; concurrent and synchronised communication with requestor and valuation firms; concurrent interactions with multiple valuation firms; dynamic determination of number and location of valuation firms.

Identifying the party with which the intermediary communicated with was addressed directly by BPEL's *partnerLink* attributes. The following code snippet, for example, defines that the initial valuation request comes from the requestor<sup>2</sup>:

---

```
<receive partnerLink="Requestor" portType="i:IntermediaryService"
  operation="ProcessValuationRequest" variable="ValuationRequest"/>
```

---

In order to express the fact that a requestor can communicate with the intermediary concurrently with the latter's interactions with valuation firms, we used a *flow* activity, which is used to specify that one or more activities are performed concurrently:

---

```
<flow>
  communication with requestor...
  communication with valuation firms...
</flow>
```

---

Modelling multiple interactions with multiple valuation firms was facilitated using a *forEach* activity. The loop iterates through a list of endpoints that can be retrieved dynamically from an external source [144]. Enabling the activities' *parallel* attribute expresses that its contained activities are executed concurrently:

---

```
<forEach counterName="ValuationFirms" parallel="yes">
  <invoke partnerLink="Valuer" inputVariable="ValuationRequest".../>
  ...
</forEach>
```

---

Although we were able to solve the problems considered difficult in the first place in a relatively straightforward manner, we were confronted with a number of other problems that arose whilst trying to describe the rest of the protocol.

We had difficulties in representing the iterations that the business protocol prescribed. It turned out that BPEL cannot represent processes where parts

---

<sup>2</sup>The complete BPEL process can be found in Appendix 7.3

of it need to be executed repeatedly without any restrictions in regard to the number, location and nesting of these points [122]. Although BPEL offers repeatable constructs such as *while*, *repeatUntil*, *forEach* or *eventHandlers*, they can only capture structured cycles (i.e. loops with one entry and one exit point). Representing arbitrary cycles using control links is likewise not possible because the BPEL specification does not allow such links to cross the boundaries of repeatable constructs<sup>3</sup>. Similarly, the same restriction prevented us from synchronising between concurrently running loops, such as, for example, terminating the running interactions with valuation firms as soon as the requestor cancelled a valuation. The only option BPEL offered to “synchronise” among concurrently running loops in that case was the *exit* activity, which can be called from anywhere within the process and which terminates all activities immediately. Because, according to our business protocol, cancelling a valuation implied terminating the conversation altogether, we were able to use this unstructured workaround. If, however, the protocol had specified further activities after the cancellation, we would not have been able to describe this part of the protocol at all.

In the end, we modelled the iterations using an infinite *while* loop. Inside the loop we nested a *pick* activity that expressed a choice among several possible events. In every iteration, the process effectively stops when it reaches the *pick* activity and waits for one of the possible event to occur (i.e. one of the incoming messages to arrive). Upon arrival of a *StatusRequestMsg*, for example, some activities are executed and the process loops back to the *pick* activity where it waits again for the next message. After receiving a *CancelValuationMsg*, on the other side, the process does not loop back to the beginning of the *while* loop, but is terminated by an *exit* activity instead:

---

```

<while>
  <condition>true</condition>
  <pick>
    <onMessage inputVariable="StatusRequest" ...>
      ...
    </onMessage>
    <onMessage inputVariable="CancelValuation" ...>
      ...
    <exit/>
    </onMessage>
  </pick>
</while>

```

---

In a similar way, we modelled the iterations between the intermediary and the valuation firms. In that case, however, the process exited either after all

<sup>3</sup>OASIS, “Web Services Business Process Execution Language Version 2.0”, Static analysis requirement [SA00070]



valuation firms had declined the request or as soon as a valuation response had been received. Again, we were able to employ the *exit* activity only because no further activities were recorded in the business protocol.

A further difficulty was the fact that the notations provided by BPEL are very execution focused. It would have been more helpful if we could have described *what* kind of interactions could occur rather than *how* they should happen. For example, expressing the fact that a *FeeChangeRequestMsg* can be followed by a *FeeChangeAcceptedMsg* or a *FeeChangeRejectedMsg* required us to describe *how* this is done using an *if-else* construct. Although abstract BPEL makes it possible to masquerade conditional expressions using placeholders, we thought that the result looked a bit awkward if used in a service description:

---

```
<b:onMessage variable="FeeChangeRequest" ...>
  <b:if>
    <b:condition>some business condition</b:condition>
    <b:invoke inputVariable="FeeChangeAccepted" .../>
  <b:else>
    <b:invoke inputVariable="FeeChangeRejected" .../>
  </b:else>
</b:if>
</b:onMessage>
```

---

Unfortunately, we did not find a way to model interactions that were triggered outside the scope of the workflow by, for example, humans or other elements that could not be captured in the workflow directly. In fact, it has been known for some time that BPEL does not allow the definition of human-based activities [146]. Hence, it was not possible to describe that at any point after sending a valuation request to a valuation firm, the intermediary could send an enquiry about the status of the valuation. Similarly, it was not possible to model that the intermediary can send status updates to the requestor at quite random intervals. Again, it essentially came down to what we have described in the previous paragraph: BPEL expected us to describe *how* something happened. Unfortunately, in this case we only knew that at some point the intermediary could, for example, send a status request, but we had no means of determining how or why or when this would happen.

#### 5.4.6 Discussion

The operation abstraction is the only means that WSDL alone provides for describing how messages relate to each other. For that reason, WSDL cannot be used to capture workflows or parts thereof that span more than two messages. As a result, we had to model most interactions as one-way operations. Apart from the service's incoming messages, this unfortunately resulted in the loss

of most protocol-related information. Thanks to WCF's programming model and tool support, the procedure and effort for creating a WSDL description was similar to generating an SSDL contract. Nevertheless, the resulting WSDL file was disproportionately verbose, considering that it contained only the most basic information about the service. As a matter of fact, Table 5.3 shows that the intermediary's WSDL file was more than 3 times longer compared to the same information expressed in the SSDL contract.

As a result, other languages such as BPEL have to be used in addition to WSDL in order to capture a service's messaging behaviour. Compared to other process languages, BPEL is very expressive and powerful when it comes to describing executable workflows [116]. Yet its many constructs and different ways to implement things also make it a very complicated language that is difficult to use. Although BPEL has been widely used and undeniably emerged as the de facto standard for specifying executable processes, some researchers claim that it has failed as a language for modelling abstract processes [100].

Through our experiments, we have come to a similar conclusion, namely that the requirements for a language to capture an executable process and an abstract process are different. Addressing them using different languages might consequently be beneficial. In our experiments, BPEL's execution-centric focus was obstructive for describing the intermediary's protocol. The result was a complicated, verbose protocol description that did not have a lot of resemblance with the original business protocol. Even though certain activities or conditions can be masqueraded in abstract BPEL, we do not believe that it is useful to have conditional constructs and the like in Web Service descriptions, as this unnecessarily increases their complexity and makes it harder for service consumers to reason about it.

Despite prior experience with BPEL, it took us a considerable amount of time to conceive the protocol description. On the one hand, the language's many construct offerings made it difficult to choose the most appropriate one. On the other hand, the BPEL specification imposes a lot of restrictions that became only apparent to us by thoroughly reading it several times. For example, in our first attempt to model the protocol, we used *eventHandlers* to process incoming messages and control links to synchronise between them. Only after we had implemented the protocol, we realised that BPEL does not allow control links to cross boundaries of *eventHandlers*, thus making our first solution invalid. Still, despite all the complexity and expression power BPEL offers, we did not succeed in capturing the complete business protocol.

The fact that BPEL is dependent on WSDL additionally adds weight to its complexity. In order to describe the protocol in BPEL, we had to modify the intermediary's WSDL description and add references to the requestor's and

Artefact	WSDL & BPEL	SSDL & SC
valuer.wsdl	112	
intermediary.wsdl	130	
requestor.wsdl	73	
intermediary.bpel	128	
intermediary.ssd		49
intermediary.sc		56
<b>TOTAL LOC</b>	<b>443</b>	<b>105</b>

Table 5.3: Comparison of lines of code required to describe intermediary's protocol using WSDL & BPEL or SSDL using the SC protocol framework.

valuation firms' WSDL files. As a result, this did not only couple the intermediary's service interface to the other parties, but also required their WSDL files to be present, in order for specifying the intermediary's process. In contrast, the only abstractions necessary for describing the protocol using the SC protocol framework were the messages that the intermediary sent and received. However, there was no need to reference the valuation firms' or requestor's service descriptions in order to capture the message exchanges with them. This complexity is reflected in Table 5.3 which summarises the lines of code that were necessary to capture the service description using either approaches<sup>4</sup>. As we can see there was almost 4.5 times more code involved for achieving the same result using WSDL and BPEL compared to SSDL and SC. This is even more remarkable as we were not able to represent the protocol as precisely with BPEL as we were with SSDL.

## 5.5 Conclusion

Our experiments have shown that there are situations where SSDL can have significant benefits in describing Web Services compared to incumbent approaches such as WSDL in conjunction with BPEL. We have found that SSDL can be a more natural choice in terms of business-to-technology mapping as well as in terms of adhering to service-oriented design principles. Further, it is a more lightweight language requiring less code and its declarative approach has evident benefits in describing business protocols. Still, we do not question BPEL's position as a language to model executable processes. We think quite contrariwise that BPEL and SSDL can complement each other. BPEL can be used internally to model processes while SSDL captures the external interactions other services have with the process.

<sup>4</sup>Excluding comments or documentation; lines were broken at 100 characters per line.

We therefore postulate that more experiments and research should be undertaken in this area in order to confirm our findings in more general terms. We think that SSDL's protocol frameworks in particular can still be improved significantly. Although we have mainly investigated the MEP, Rules and SC protocol frameworks, we believe that of all four initial releases, the SC protocol framework deserves most attention. In order to establish a standard that enjoys broad acceptance and support, we further advance the view that eventually the protocol frameworks should converge into a single standard. A lot of research is currently being undertaken in the area of workflow languages (e.g. [96, 115, 90, 116, 112]) among which some also investigate declarative approaches for describing protocols [94]. Applying the results of this research to SSDL's protocol frameworks could be beneficial by making them more expressive, powerful and hence, ultimately, more widely applicable.

## Chapter 6

# Discussion

The development of Soya has provided us with valuable insights into message-oriented service development and the SSDL specification. Furthermore, we have gained solid experience of extending a contemporary Web Services platform. In this section we therefore discuss some of the findings and experiences we have made in this respect.

### 6.1 State Machine Expression Power

The fact that Soya internally uses a state machine for modelling protocols has implications in terms of what class of protocol languages it can represent. As we have seen earlier, SSDL's protocol frameworks define protocol languages using XML. The fact that XML is not only a context-free (i.e. non-regular) language [143], but can also be used for describing its own structure [73], makes it possible to create languages that exceed the expression power of state machines. The following is a classical example of a context-free language that cannot be represented with a state machine. It defines that a number of  $x$  symbols must be followed by the same number of  $y$  symbols (this could, for example, be used to match opening and closing parentheses):

$$A \rightarrow xAy \mid \epsilon$$

Indeed, the Sequencing Constraints SSDL protocol framework [98] makes it possible to define such structures as illustrated in the following example:

---

```
<sc:protocol name="A">
  <sc:choice>
    <sc:sequence>
      <ssdl:msgref ref="m:x" direction="in" />
```

---

```

    <sc:protocolref ref="A"/>
    <ssdl:msgref ref="m:y" direction="out" />
  </sc:sequence>
  <sc:nothing />
</sc:choice>
</sc:protocol>

```

---

Given its state machine abstraction for capturing protocols, Soya can therefore not process these kinds of protocols. At the current stage it is unclear whether this implementation limitation has significant drawbacks in practice, or if the definition of non-regular protocols is rather unusual and can thus be momentarily ignored.

## 6.2 State Maintenance using WS-Addressing

The WS-Addressing specification [13] suggests that *endpoint references* may be used to identify specific instances of a stateful service. As discussed in Section 4.3.1, we did not implement this approach, because it implies the use of custom and thus non-standardised headers, as shown below:

---

```

<wsa:EndpointReference xmlns:wsa="..." xmlns:c="urn:custom:schema">
  <wsa:Address>http://...</wsa:Address>
  <wsa:ReferenceParameters>
    <c:MyId>6B29FC40-CA47</c:MyId> // non-standard element
  </wsa:ReferenceParameters>
</wsa:EndpointReference>

```

---

Instead, we chose to correlate messages solely based on *MessageID* and *RelatesTo* headers. While this approach uses standardised headers, it likewise has a number of implications. Message sequences, for example, which are logically connected in this way are harder to keep track of, since the session identifier does not remain constant during the lifespan of a conversation. However, there is a simple solution to this problem, because nothing prevents service implementers from generating invariable session ids internally. These unchanging ids can then be used for logging, auditing and other purposes that require some degree of consistency.

### 6.2.1 Uniqueness of MessageID

A more problematic issue is related to ensuring uniqueness of message ids. As discussed in Section 4.3.2, we associate internal state with the *MessageID* value of the last processed message. Consequently, we need to ensure that no two

concurrent conversations send or receive messages with equal *MessageID* values. While it is possible to ensure uniqueness for locally generated message ids, it is not for those generated externally by other services. In fact, every incoming message that starts a conversation contains a *MessageID* value that was generated by a source that cannot be trusted. As a result, messages might be correlated incorrectly and internal state exposed to a wrong client. One could argue that this problem likewise exists when using locally generated constant session ids. Indeed, a service could try to hijack other sessions, by guessing their session ids. Yet this could only happen in the case of an attack and not by mere accident, because the session ids are always generated locally and their uniqueness can thus be guaranteed. Then again, session ids based on *MessageID* tend to be exposed to potential attacks for a shorter period of time, because they change every exchanged message. The point that we want to make here is that we believe this topic deserves some attention and further research, if the mechanism is to be used in production environments.

### 6.2.2 Multi-Party Conversations

Another issue related to the one above arises when a conversation spans more than two services. Obviously, in this case the likelihood that two services generate the same *MessageID* increases with the number of participants. In addition, we can also no longer create single logical conversations by sequentially connecting messages using ids only, because services not aware of message exchanges between other services might experience gaps in the sequence. The following example shows a conversation that involves three services *A*, *B* and *C*.  $S(x, y)$  means that service *S* receives an incoming message with *MessageID* *x*, which relates to a previous message *y*. We assume that the first incoming message received by service *B* was sent by *A*.

$$B(1), A(2, 1), B(3, 2), C(4, 3), B(5, 4), A(6, 5) \dots$$

The problematic elements are  $C(4, 3)$  and  $A(6, 5)$ . In the first case, service *C* receives a message from *B* that relates to a previous message unknown to *C*, because the two services had not exchanged messages yet and the related-to message was actually used in the conversation between *A* and *B*, not *B* and *C*. In the latter case, *A* similarly receives a message that relates to a previous message that it does not know. Although in this case, *A* had communicated with *B* previously, the last message that it remembers actually had a *MessageID* of value 3, not 5.

Consequently, this issue needs to be addressed if Soya's component for state maintenance and message correlation is to be used for multi-party interactions.

One possible solution could be to have separate conversations between exactly two parties only and then aggregate them into a single logical conversation. However, we have not fully investigated the implications of this proposal.

### 6.3 Defining Protocols using Attributes

Despite the benefits of *C#* attributes, we see two main drawbacks to their use. First, defining complex interaction protocols can result in a large number of attributes that can interfere with the readability of the actual program code. Second, because the service's messaging behaviour is no longer separated from the source code, changing the former implies recompiling the latter. Although this was not an issue during our investigation, we understand that it can be one for other projects. Especially in large enterprise projects, recompiling and deploying application code often triggers a large number of additional activities (e.g. automated or human-driven tests, sophisticated deployment mechanisms, management approval, etc.) which might incur a considerable amount of time and cost.

### 6.4 SOAP Action Semantics

The SOAP action attribute is neither properly defined in the SOAP specification nor consistently used across different SOAP engines. As a result, this can cause interoperability problems between services running on different SOAP middleware. Moreover, the common practice of using internal service operation names as SOAP action attribute values leaks implementation details and is ambiguous when overloading service methods. In our opinion, this attribute is thus superfluous. In fact, we have shown that messages can be dispatched correctly and unambiguously without using the SOAP action attribute.

### 6.5 WSDL Operations

We believe that WSDL's focus on operation is obstructive for creating loosely coupled service-oriented applications. Although WSDL certainly can be used and, in fact, is used to build such applications, the operation abstraction encourages the construction of RPC-like systems. Indeed the operation-abstraction led many tool vendors to create software which spurs developers to generate WSDL from existing application objects and interfaces, which subsequently leads to tightly coupled and brittle systems. In contrast, we have shown that Soya and SSDL provide operation-agnostic programming abstractions that encourage and



support developers thinking in terms of message exchanges between services and thus automatically averts the creation of RPC-like applications.

## 6.6 Asynchronous Interaction Model

Besides the impact that asynchrony has on the programming model (see Section 2.2) it also affects the way services connect to each other. Because response messages from services to clients are not sent back over the same connection (see Section 4.6.2), clients must be exposable via public endpoint addresses in order to receive responses. In a large number of settings this poses a problem, because clients are behind firewalls in private networks and thus not directly accessible from outside. Until IPv6 [147] will provide an address space large enough to directly address every device connected to the Internet, we do not see an ideal solution for solving this problem. Although workarounds exist for certain situations, they do not work universally. In order to support long-running transactions over HTTP, for example, current practice in (synchronous) Web Service development suggests that services repeatedly send HTTP 200 - Accepted messages back to clients, so that a connection can remain open over an extended period of time, until a response is sent back. This solution, however, does not scale well, because a large number of connections must be kept open at the same time. In fact, the number of connections that need to be maintained grows with the number of clients that use the service. The asynchronous communication behaviour of Soya and SSDL, on the other hand, requires opening an additional connection for sending back the response, which incurs some overhead compared to using one connection only. Nevertheless, this overhead is constant and consequently not affected by the number of connecting clients. As a result, this solution actually scales better, because a constant amount of additional hardware would neutralise the performance penalty caused by the overhead of opening the additional connection.

## 6.7 Assuming SOAP Only

SSDL praises itself for being simpler and more concise than WSDL as a result of assuming SOAP as the only means for transferring messages. Indeed, it cuts the number of lines of code necessary to describe a service by a factor of approximately two, because it does not require concrete binding code. Yet we believe that this increase in simplicity unnecessarily constrains SSDL's flexibility for two reasons. First, there is a growing number of Web Service applications that do not employ SOAP as the standard mechanism for communication. At one

end of the spectrum, applications might be based on REST principles and use plain XML to convey data among each other. At the other end, performance critical systems such as mobile devices or sensor networks might favour a more lightweight or binary message format in order to reduce message size and processing overhead. By supporting SOAP only, however, SSDL shuts itself off from being used by this class of applications. Second, although the reduction of complexity appears quite significant in terms of lines of code, it is not so much in terms of development effort. This is because specifying the binding data, which becomes unnecessary when assuming SOAP only, is a straightforward and mechanical task, which can easily be automated. Moreover, we think that the benefit in terms of readability is not as significant as claimed by SSDL, because the binding and logical service information could be separated from each other. For these reasons, we think that it would make sense if SSDL assumed SOAP as the *default* means of transferring data and optionally allowed specifying other message formats. An additional binding attribute at the contract, endpoint, protocol or message level would ensure the flexibility of using alternative message formats without adding significant new complexity to the language. The following lines show how we could, for example, define two endpoints that use SOAP and a binary message format, respectively:

---

```
<ssdl:contract xmlns:ssdl="urn:ssdl:v1" targetNamespace="urn:my:contract">
  <ssdl:schemas...
  <ssdl:messages...
  <ssdl:protocols...
  <ssdl:endpoints>
    <ssdl:endpoint xmlns:wsa="...">
      <wsa:Address>http://my.service.com/default</wsa:Address>
    </ssdl:endpoint>
    <ssdl:endpoint binding="urn:my:binary" xmlns:wsa="...">
      <wsa:Address>http://my.service.com/binary</wsa:Address>
    </ssdl:endpoint>
  </ssdl:endpoints>
</ssdl:contract>
```

---

## 6.8 Content-Based Message Dispatching

We believe that message dispatching based on message type and conversation state can be a very useful tool (see Section 4.6). Taking this idea further and applying it at the message content level therefore seems like a natural extension of the mechanism and a likewise useful proposition. Indeed, such a mechanism would have been helpful during our case study (see Chapter 5). The protocol state machine in Figure 5.3 defines a transition symbol  $s'$  (i.e. an incoming *Status* response) that makes the state machine transit to its final state. Yet

by scrutinizing the actual business protocol it becomes apparent that only a *declined* status response can actually trigger this transition. Indeed, capturing this information in the protocol is not possible. As a result, the validation and dispatching logic for this particular part had to be implemented inside the application code. If Soya or SSDL had offered a mechanism to validate and route messages based on content, this would not have been necessary. The following code is a suggestion for how content-based message dispatching and validation could be facilitated in Soya's MEP programming model. An additional attribute could take an XPath [148] expression, which then could be evaluated at runtime:

---

```
[Mep(Style=MepStyle.InOnly, Content="/Status/Name = 'Declined'")]
void Process(StatusMsg msg);
```

---

Whether this additional information should be propagated to the exposed contract, however, is debatable. We have discussed this to some extent in earlier chapters and think that it would add too much complexity to the exposed protocol description. Moreover, we doubt that this additional information could actually be leveraged in a sensible or unambiguous manner by protocol-aware software. Internally, however, we believe the additional information could prove useful, as we have just described.

## 6.9 SSDL Faults

While implementing Soya, we realised that the SSDL specification expects that runtime values such as fault codes, fault reasons and so forth are defined inside the *ssdl:fault* element as part of the service contract. We argue that it is generally not possible to know all possible fault values in advance (e.g. codes, subcodes, reasons, etc.). Even if so, we do not consider it good practice to enumerate all possible infrastructure faults in the contract – possibly even region and language dependent.

We believe that there is an important distinction between *infrastructure faults* and *application exceptions*; and that the latter should not be transmitted as SOAP faults. SOAP faults should be used to signal a problem that has occurred in the underlying infrastructure. An example for an *infrastructure fault* could be a SOAP node that was not able to process a SOAP header block. An *application exception*, on the other side, is a regular SOAP message containing application data. It is up to the application or message processing logic to interpret the message as an *application exception*. An *OutOfStock* message, for example, is clearly an *application exception*, not an *infrastructure fault*, and should consequently not be transferred as a SOAP fault.

Unfortunately, there does not seem to be general consensus on the topic, neither in the Web Services nor in the SSDL space. In both, the SOAP [11] and SSDL [149] specifications we can find examples that transfer application-level information as *infrastructure faults* (i.e. SOAP faults).

Given our experience developing enterprise Web Services, we conclude that it is bad practice to transmit application data within *infrastructure faults*. Still, given the fact that SSDL ultimately exists to describe SOAP messages, we understand that it needs to support the description of those faults. We thus propose a change in the structure of the *ssdl:fault* element to the following, where *detail* refers to an element defined in the SSDL schemas section:

---

```
<ssdl:fault name="xs:string">
  <detail="xs:QName"/>
</ssdl:fault>
```

---

As a matter of fact, this is how we have implemented it in Soya. Moreover, we have provided feedback to the SSDL specification maintainers as well as the wider SSDL community on this matter.

### 6.9.1 Fault XML Schema

In addition to the changes above, we propose two further modifications that relate to the *ssdl:fault* element. First, the way this element is currently defined does not allow for the specification of header elements for faults. In order to maintain consistency between the definition of *ssdl:message* and *ssdl:fault* as well as giving developers full control over defining the structure of expected SOAP faults, we suggest that this should be considered in a future version of SSDL. Second, we think that referencing faults using a separate element would not only increase readability of protocols but give protocol framework designers better control in specifying protocol languages. For example, the *robust-in-only* MEP is currently defined as follows:

---

```
<mep:robust-in-only>
  <ssdl:msgref direction="in" />
  <ssdl:msgref direction="out" /> +
</mep:robust-in-only>
```

---

In this definition, it is neither visible nor can it be checked using simple XML Schema validation that the second element actually must refer to a fault, not a message. As shown below, using an *ssdl:faultref* element instead makes it possible to use XML Schema validation, improve readability and remove confusion:

---

```
<mep:robust-in-only>
  <ssdl:msgref direction="in" />
  <ssdl:faultref direction="out" /> +
</mep:robust-in-only>
```

---



## Chapter 7

# Conclusions

“ The fact that man knows right from wrong proves his intellectual superiority to other creatures; but the fact that he can do wrong proves his moral inferiority to any creature that cannot. ”

— *Mark Twain*

Over the years, paradigms and best practices for building distributed software systems have changed and evolved many times, requiring software architects to continuously adapt both technology and mindset. Yet despite the variety in distributed software technologies, products and paradigms that have been created over time, most of them are based on interaction models that have their roots in either RPC or asynchronous messaging communication. Intuitively, choosing a particular interaction model has a number of consequences on the general application design. RPC-based approaches attempt to shield the network and provide developers with familiar programming abstractions based on operation invocation semantics. Unfortunately, RPC applications tend to be tightly coupled, brittle at distribution boundaries and limited in scalability. Messaging, on the other side, is well suited for integrating applications in a loosely coupled and scalable way. Traditional messaging approaches, however, require centralised infrastructure, which has been a major practical obstacle for integrating components across organisational, trust or geographical boundaries.

In recent years, service-oriented architecture (SOA) has attracted considerable interest as an architectural style for building Internet-scale applications. SOA decomposes applications into autonomous units of logic called services. In accordance with the above, services use message passing as the fundamental abstraction for communicating and exchanging structured information among each other. Web Services are a suitable technology platform for realising service-oriented applications. Moreover, they address interoperability and integration

issues, which have often failed traditional messaging approaches. Yet we accepted in this thesis that simply using Web Services technology will not automatically lead to service-oriented design. In particular, we argued that WSDL's operation-centric design encourages Web Service developers to construct applications that are architecturally indifferent from RPC systems. We furthermore reasoned that WSDL is too complex, produces overly verbose service descriptions, provides insufficient control over SOAP messages and is unable to capture message ordering constraints over and above simple *request-response* patterns.

In contrast, the target for our empirical work used an alternative Web Service description language called SSDL. Because of its focus on one-way messages as the primary means for communication, we suggested that SSDL naturally fosters loosely coupled, scalable and service-oriented design.

## 7.1 Summary of Work Undertaken

Given the lack of tools and empirical data for using SSDL as part of Web Services-based SOAs, we identified the need to investigate SSDL's practicability and usefulness through empirical work. To that end we developed Soya, a programming model and runtime environment for creating and executing SSDL-based Web Services, respectively. We presented programming abstractions that not only allow developers to build SSDL services in a straightforward way but also encourage the creation of truly service-oriented applications without imposing unrealistic development burdens. Furthermore, we provided a detailed explanation of how we leveraged a contemporary SOAP engine by adding functionality and semantics related to SSDL, thus providing an advanced runtime environment for executing SSDL-based Web Services to the community. In order to validate the usability of Soya's programming model and the proper functioning of its runtime environment, we mobilised Soya to create a service-oriented system in the context of the Australian lending industry.

In summary, the following contributions to the field of software research have been made as a result of this thesis:

1. a design and implementation of a programming model and runtime environment for creating and executing SSDL-based Web Services, respectively. This serves as a knowledge framework for better understanding message- and service-oriented Web Service development (i.e. MEST, SOA);
2. a demonstration of SSDL's practicability in terms of implementation and usability;



3. a case study and subsequent comparison with incumbent approaches, providing an initial set of empirical results assessing SSDL's characteristics.

## 7.2 Summary of Findings

The development of Soya and its subsequent use in a case study has provided a valuable insight into message- and service-oriented Web Service development. In addition, this has provided an initial set of empirical results with respect to some of SSDL's characteristics. First, by creating straightforward programming abstractions and advanced runtime support, we have demonstrated that the ideas underlying SSDL are realisable in practice. We have conceived and implemented straightforward programming abstractions based on C# attributes that make it possible to create SSDL-based Web Services without imposing significant additional development burdens. By leveraging a contemporary SOAP engine with SSDL-related functionality and semantics we have created unprecedented runtime support for executing SSDL-based Web Services. This infrastructure processes incoming and outgoing SOAP messages and ensures contract conformance in terms of their structure and ordering. Furthermore, it leverages protocol information and features advanced facilities for correlating and dispatching both incoming and outgoing messages.

Second, our experience has confirmed that Soya and SSDL's insistence on message passing as the primary abstraction for communication naturally leads to loosely coupled, scalable and service-oriented design. In addition, we realised that SSDL's asynchronous communication model is highly suitable for business processes that require human interaction. Indeed, the valuation firms in our case study illustrated that companies sometimes might not have the necessary transaction volume, time, budget and so forth to fully automate a given process. Instead, they might choose humans to process certain tasks. Soya and SSDL make it easy to create Web Services that delegate incoming messages to humans for processing. From outside, this is neither visible nor relevant and thus makes it possible that such companies can still participate in automated processes that exceed their organisational boundaries. Third, we have not only maintained that the operation abstraction is obstructive for achieving service-oriented designs, but showed that it is not needed in service descriptions. Fourth, our initial results suggest that SSDL service descriptions are significantly less complex in comparison with the results obtained from using prevalent approaches. In our case study, for example, the lines of code were cut by a factor of almost 4.5.

### 7.3 Directions for Future Work

Although our case study has provided an initial set of results in regards to various aspects of SSDL, we would like to see Soya being used as a research vehicle in further, more systematic empirical investigations and case studies. This is necessary to confirm our initial results using stronger empirical evidence. It needs to be applied to a larger sample and in different fields to discern explicitly which features are idiosyncratic to the current field and which are germane across the discipline of software engineering, contextually rich though the case studies were.

Also, the area of protocol research is still an active and hotly debated one. Many languages and models for describing protocols currently exist. Yet it is often unclear whether differences among them are fundamental or merely syntactic in nature. As a result, little can be said about the usefulness of SSDL's initial protocol framework choices. We believe that the strongest advantage of the Sequencing Constraints SSDL protocol framework is its claimed link to a formal model (i.e.  $\pi$ -calculus) that has been thoroughly researched for more than a decade. In terms of extending Soya and creating further tool support, we consequently argue that conceiving sensible programming abstractions for this protocol framework ought to be a priority.

Realistically, we do not believe that SSDL will replace WSDL in the near future for two reasons: First, WSDL has been firmly established as a standard and a vast amount of money has been spent in creating related tooling. Second, despite its flaws, WSDL can still be used to create applications that adhere to service-oriented design principles – if developers make the right decisions. Yet even if SSDL will not replace incumbent Web Service description or protocol languages, we hope that our work will help in leading future Web Service development practices in a more message-oriented direction, ultimately resulting in better distributed applications.

# Bibliography

- [1] OASIS, “Reference model for service oriented architecture v 1.0.” <http://www.oasis-open.org/committees/soa-rm/>, 2006.
- [2] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice-Hall, 2005.
- [3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2003.
- [4] W3C, “Web services architecture.” <http://www.w3.org/TR/ws-arch/>, 2004.
- [5] W3C, “Web services description language (WSDL) version 2.0 part 1: Core language.” <http://www.w3.org/TR/wsdl20/>, 2007.
- [6] The Apache Software Foundation, “Axis2.” <http://ws.apache.org/axis2/>.
- [7] Microsoft Corporation, “Web services description language tool (wsdl.exe).” <http://msdn2.microsoft.com/en-us/library/7h3ystb6.aspx>.
- [8] The Internet Engineering Task Force (IETF), “A high-level framework for network-based resource sharing.” Internet RFC 707, 1976.
- [9] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, “A note on distributed computing,” tech. rep., Sun Microsystems Laboratories, Mountain View, CA, 1994.
- [10] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield, “An introduction to the SOAP service description language,” Tech. Rep. CS-TR-898, School of Computing Science, University of Newcastle upon Tyne, 2005.
- [11] W3C, “SOAP version 1.2 part 1: Messaging framework.” <http://www.w3.org/TR/soap12-part1/>, 2003.
- [12] P. Fornasier, J. Webber, and I. Gorton, “Soya: a programming model and runtime environment for component composition using SSDL,” in *Component-Based Software Engineering*, pp. 227–241, Springer, 2007.
- [13] W3C, “Web services addressing (WS-Addressing).” <http://www.w3.org/Submission/ws-addressing/>, 2004.

- [14] Object Management Group, “CORBA/IIOP specification.” [http://www.omg.org/technology/documents/formal/corba\\_iiop.htm](http://www.omg.org/technology/documents/formal/corba_iiop.htm).
- [15] Microsoft Corporation, “Distributed component object model (DCOM).” <http://www.microsoft.com/com/>.
- [16] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol – http/1.1.” Internet RFC 2616, 1999.
- [17] Sun Microsystems, Inc., “Java remote method invocation (Java RMI).” <http://java.sun.com/products/jdk/rmi/>.
- [18] V. Matena, S. Krishnan, L. DeMichiel, and B. Stearns, *Applying Enterprise JavaBeans 2.1: Component-Based Development for the J2EE Platform (2nd Edition)*. Java series, Addison-Wesley Professional, 2003.
- [19] G. Banavar, T. Chandra, R. Strom, and D. Sturman, “A case for message oriented middleware,” in *13th International Symposium on Distributed Computing (DISC)*, pp. 846–863, Springer, 1999.
- [20] W. Vogels, “Web services are not distributed objects,” *IEEE Internet Computing*, vol. 7, no. 6, pp. 59–66, 2003.
- [21] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.
- [22] S. Vinoski, “RPC under fire,” *IEEE Internet Computing*, vol. 9, no. 5, pp. 93–95, 2005.
- [23] S. Vinoski, “Putting the “web” into web services. web services interaction models, part 2,” *IEEE Internet Computing*, vol. 6, no. 4, pp. 90–92, 2002.
- [24] I. Gorton, *Essential Software Architecture*. Springer, 2006.
- [25] The Society for Worldwide Interbank Financial Telecommunications (SWIFT), “SWIFT reaches 2 billion message mark.” [http://www.swift.com/index.cfm?item\\_id=41623](http://www.swift.com/index.cfm?item_id=41623), 2003.
- [26] J. Hart, “WebSphere MQ: Connecting your applications without complex programming,” tech. rep., IBM WebSphere Software White Papers, 2003.
- [27] Microsoft Corporation, “Microsoft message queuing (MSMQ).” <http://www.microsoft.com/windowsserver2003/technologies/msmq/>.
- [28] TIBCO Software Inc., “TIBCO rendezvous.” <http://www.tibco.com/software/messaging/>.
- [29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [30] D. Sprott and L. Wilkes, “Understanding service-oriented architecture,” *Microsoft Architects Journal*, vol. 1, no. 1, pp. 10–17, 2004.

- [31] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005.
- [32] T. Erl, *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice-Hall, 2004.
- [33] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Kroghdahl, M. Luo, and T. Newling, *Patterns: Service Oriented Architecture And Web Services*. IBM Redbooks, IBM, 2004.
- [34] J. Webber and S. Parastatidis, “Realising service oriented architectures using web services,” in *Service Oriented Computing*, MIT Press, (In press).
- [35] P. Greenfield, “Service-oriented architectures and technologies,” in *Essential Software Architecture*, pp. 217–237, Springer, 2006.
- [36] D. Pallmann, *Programming INDIGO*. Microsoft Press, 2005.
- [37] D. Box, “A guide to developing and running connected systems with indigo,” *MSDN Magazine*, vol. 19, no. 1, 2004.
- [38] P. Helland, “Data on the outside versus data on the inside,” in *CIDR*, pp. 144–153, 2005.
- [39] W3C, “Extensible markup language (XML).” <http://www.w3.org/XML/>.
- [40] S. Vinoski, “REST eye for the SOA guy,” *IEEE Internet Computing*, vol. 11, no. 1, pp. 82–84, 2007.
- [41] S. Vinoski, “Web services interaction models. current practice,” *IEEE Internet Computing*, vol. 6, no. 3, pp. 89–91, 2002.
- [42] BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, and Sybase, “Service component architecture - building systems using a service oriented architecture,” November 2005.
- [43] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” *ACM Trans. Inter. Tech.*, vol. 2, no. 2, pp. 115–150, 2002.
- [44] R. T. Fielding, *Architectural Styles and the Design of Network-Software Architectures*. PhD thesis, University of California, 2000.
- [45] D. Box, G. Kavivaya, A. Layman, S. Thatte, and D. Winer, “SOAP simple object access protocol,” 1999.
- [46] T. Berners-Lee, R. T. Fielding, and L. Masinter, “Uniform resource identifiers (uri): Generic syntax.” Internet RFC 2396, 1998.
- [47] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, “Developing web services choreography standards: the case of rest vs. soap,” *Decis. Support Syst.*, vol. 40, no. 1, pp. 9–29, 2005.
- [48] “Amazon web services.” <http://www.amazon.com/webservices>.
- [49] T. O’Reilly, “REST vs. SOAP at amazon.” <http://www.oreillynet.com/pub/wlg/3005>, 2003.

- [50] S. Parastatidis, "The MEST architectural style." <http://savas.parastatidis.name/2004/11/09/92ede84c-ca1f-41ab-8feb-8ba50d07e86f.aspx>, 2004.
- [51] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.
- [52] P. Fremantle, S. Weerawarana, and R. Khalaf, "Enterprise services," *Communications of the ACM*, vol. 45, no. 10, pp. 77–82, 2002.
- [53] S. Chatterjee and J. Webber, *Developing Enterprise Web Services: An Architect's Guide*. Prentice-Hall, 2003.
- [54] L. F. Cabrera, C. Kurt, and D. Box, "An introduction to the web services architecture and its specifications," *Web Services Technical Articles*, 2004.
- [55] Microsoft Corporation, "Web services and the microsoft platform." <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnwebsrv/html/wsplatform.asp>.
- [56] IBM developerWorks, "Standards and web services." <http://www-128.ibm.com/developerworks/webservices/standards/>, 2006.
- [57] S. Vinoski, "WS-nonexistent standards," *IEEE Internet Computing*, vol. 8, no. 6, pp. 94–96, 2004.
- [58] Sun Microsystems, Inc., "Web services essentials." <http://java.sun.com/webservices/>, 2006.
- [59] Microsoft Corporation, "Web services specifications." <http://msdn.microsoft.com/webservices/webservices/understanding/specs/>, 2006.
- [60] The Codehaus, "XFire." <http://xfire.codehaus.org/>.
- [61] Microsoft Corporation, "Windows communication foundation (WCF)." <http://wcf.netfx3.com/>.
- [62] W3C, "Web services policy 1.2 - framework (WS-Policy)." <http://www.w3.org/Submission/WS-Policy/>, 2004.
- [63] W3C, "Web services choreography description language version 1.0, candidate recommendation." <http://www.w3.org/TR/ws-cdl-10/>, 2005.
- [64] OASIS, "Web services business process execution language version 2.0." <http://docs.oasis-open.org/wsbpel/2.0/>, 2006.
- [65] K. Verma and A. Sheth, "Semantically annotating a web service," *IEEE Internet Computing*, vol. 11, no. 2, pp. 83–85, 2007.
- [66] W3C, "OWL-S: Semantic markup for web services." <http://www.w3.org/Submission/OWL-S/>, 2004.

- [67] S. Loughran and E. Smith, “Rethinking the Java SOAP stack,” Tech. Rep. HPL-2005-83, Hewlett-Packard Bristol Laboratories, May 2005.
- [68] OASIS, “Web services reliable messaging.” <http://docs.oasis-open.org/ws-rx/wsrn/200702>, 2007.
- [69] OASIS, “Web services security: Soap message security 1.1.” <http://docs.oasis-open.org/wss/v1.1/>, 2006.
- [70] OASIS, “Web services coordination.” <http://docs.oasis-open.org/ws-tx/wscoor/2006/06>, 2007.
- [71] The Web Services-Interoperability Organization (WS-I), “Basic profile version 1.1.” <http://www.ws-i.org/Profiles/BasicProfile-1.1.html/>, 2006.
- [72] D. Kaye, *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.
- [73] W3C, “XML schema.” <http://www.w3.org/XML/Schema>, 2004.
- [74] G. Pavlou, P. Flegkas, S. Gouveris, and A. Liotta, “On management technologies and the potential of web services,” *Communications Magazine, IEEE*, vol. 42, no. 7, pp. 58–66, 2004.
- [75] K. J. Ma, “Web services: What’s real and what’s not?,” *IEEE IT Professional*, vol. 07, no. 2, pp. 14–21, 2005.
- [76] J. Pasley, “How bpel and soa are changing web services development,” *Internet Computing, IEEE*, vol. 9, no. 3, pp. 60–67, 2005.
- [77] P. Maheshwari, H. Tang, and R. Liang, “Enhancing web services with message-oriented middleware,” in *IEEE International Conference on Web Services*, pp. 524–531, IEEE Computer Society, 2004.
- [78] A. Carzaniga, *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, 1998.
- [79] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, “Achieving scalability and expressiveness in an internet-scale event notification service,” in *19th annual ACM symposium on Principles of distributed computing (PODC)*, pp. 219–227, ACM Press, 2000.
- [80] U. Zdun, M. Voelter, and M. Kircher, “Pattern-based design of an asynchronous invocation framework for web services,” *International Journal of Web Services Research*, vol. 1, no. 3, pp. 42–62, 2004.
- [81] Object Management Group, “IDL syntax and semantics chapter.” <http://www.omg.org/cgi-bin/doc?formal/02-06-39>.
- [82] R. Salz, “Wsd1 2: Just say no.” <http://webservices.xml.com/pub/a/ws/2004/11/17/salz.html>, 2004.
- [83] D. Hinchcliffe, “Web service description languages: When there is nothing left to take away.” <http://hinchcliffe.org/archive/2005/05/10/215.aspx>, 2005.

- [84] L. G. Meredith and S. Bjorg, “Contracts and types,” *Commun. ACM*, vol. 46, no. 10, pp. 41–47, 2003.
- [85] W3C, “Web services description language (WSDL) version 2.0 part 2: Adjuncts.” <http://www.w3.org/TR/wsd120-adjuncts/>, 2006.
- [86] S. Parastatidis, S. Woodman, J. Webber, D. Kuo, and P. Greenfield, “Asynchronous messaging between web services using SSDL,” *Internet Computing, IEEE*, vol. 10, no. 1, pp. 26–39, 2006.
- [87] N. Desai, A. K. Mallya, A. K. Chopra, and M. P. Singh, “Interaction protocols as design abstractions for business processes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1015–1027, 2005.
- [88] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, “Formalizing web service choreographies,” in *Workshop on Web Services and Formal Methods (WS-FM)*, Elsevier Science Publishers B. V., 2004.
- [89] G. Salaün, L. Bordeaux, and M. Schaerf, “Describing and reasoning on web services using process algebra,” in *IEEE International Conference on Web Services*, pp. 43–50, IEEE Computer Society, 2004.
- [90] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani, “Developing adapters for web services integration,” in *Advanced Information Systems Engineering*, pp. 415–429, 2005.
- [91] B. Benatallah, F. Casati, and F. Toumani, “Representing, analysing and managing web service protocols,” *Data & Knowledge Engineering*, vol. 58, no. 3, pp. 327–357, 2006.
- [92] RosettaNet, “RosettaNet: Lingua franca for e-business.” <http://www.rosettanet.org>.
- [93] “Lending Industry XML Initiative (LIXI).” <http://www.lixixi.org.au>.
- [94] W. M. P. van der Aalst and M. Pesic, “Specifying, discovering, and monitoring service flows: Making web services process-aware,” Tech. Rep. BPM-06-09, BPMcenter.org, 2006.
- [95] W. Reisig and G. Rozenberg, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets (Lecture Notes in Computer Science)*. Springer, 1998.
- [96] W. M. P. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.
- [97] R. Milner, “The polyadic pi-calculus: a tutorial,” in *Logic and Algebra of Specification*, pp. 203–246, Springer, 1993.
- [98] S. Woodman, S. Parastatidis, and J. Webber, “Sequencing constraints SSDL protocol framework,” Tech. Rep. CS-TR-903, School of Computing Science, University of Newcastle upon Tyne, 2005.
- [99] X. Fu, T. Bultan, and J. Su, “Conversation protocols: A formalism for specification and verification of reactive electronic services,” in *Implementation and Application of Automata*, pp. 189–212, Springer, 2003.



- [100] W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, N. Russell, H. M. W. Verbeek, and P. Wohed, "Life after BPEL?," in *Formal Techniques for Computer Systems and Business Processes*, pp. 35–50, 2005.
- [101] S. Parastatidis and J. Webber, "CSP SSDL protocol framework," Tech. Rep. CS-TR-901, School of Computing Science, University of Newcastle upon Tyne, 2005.
- [102] D. Kuo, S. Parastatidis, and J. Webber, "Rules SSDL protocol framework," Tech. Rep. CS-TR-902, School of Computing Science, University of Newcastle upon Tyne, 2005.
- [103] R. Milner, *Communication and concurrency*. Prentice-Hall, 1989.
- [104] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [105] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
- [106] W3C, "Web service choreography interface (WSCI) 1.0, W3C note." <http://www.w3.org/TR/wsci/>, 2002.
- [107] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based analysis of obligations in web service choreography," *aict-iciw*, vol. 0, p. 149, 2006.
- [108] C. Ouyang, W. M. P. van der Aalst, S. Breutel, M. Dumas, A. H. M. ter Hofstede, and H. M. W. Verbeek, "Formal semantics and analysis of control flow in ws-bpel," Tech. Rep. BPM-05-15, BPMcenter.org, 2005.
- [109] X. Zhao, H. Yang, and Q. Zongyan, "Towards the formal model and verification of web service choreography description language," tech. rep., LMAM and Department of Informatics, School of Math., Peking University, 2006.
- [110] A. Barros, M. Dumas, and P. Oaks, "A critical overview of the web services choreography description language (WS-CDL)," tech. rep., BP-Trends, 2005.
- [111] S. Ross-Talbot, "Web services choreography and process algebra." SWSL Committee: Working Materials, 2004.
- [112] N. Desai, A. K. Chopra, and M. P. Singh, "Representing and reasoning about commitments in business processes," in *22nd Conference on Artificial Intelligence (AAAI)*, AAAI Press, 2007.
- [113] A. K. Chopra and M. P. Singh, "Contextualizing commitment protocols," in *5th international joint conference on Autonomous agents and multiagent systems (AAMAS)*, pp. 1345–1352, ACM Press, 2006.
- [114] B. Kiepuszewski, A. H. M. ter Hofstede, and C. J. Bussler, "On structured workflow modelling," in *12th International Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, pp. 431–445, Springer, 2000.

- [115] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
- [116] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar, "Workflow control-flow patterns: A revised view," Tech. Rep. BPM-06-02, BPMcenter.org, 2006.
- [117] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organization," *The International Journal of High Performance Computing Applications*, vol. 15, pp. 200–222, 2001.
- [118] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, pp. 217–249, John Wiley & Sons, 2003.
- [119] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price, "Grid service orchestration using the business process execution language (BPEL)," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 283–304, 2005.
- [120] A. Slominski, "Adapting bpm to scientific workflows," in *Workflow for e-Sciences*, pp. 212–230, Springer, 2006.
- [121] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice-Hall, 1999.
- [122] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Analysis of web services composition languages: The case of BPEL4WS," in *Conceptual Modeling - ER 2003*, pp. 200–215, 2003.
- [123] W. M. P. van der Aalst and A. H. M. ter Hofstede, "Yawl: Yet another workflow language," Tech. Rep. BPM-05-01, BPMcenter.org, 2005.
- [124] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede, "Design and implementation of the yawl system," Tech. Rep. BPM-04-02, BPMcenter.org, 2004.
- [125] Oracle, "Oracle bpm process manager." <http://www.oracle.com/technology/bpel/>.
- [126] Active Endpoints, "Activebpm." <http://www.active-endpoints.com/>.
- [127] Eclipse, "Bpm project." <http://www.eclipse.org/bpel/>.
- [128] K. Chiu, T. Devadithya, W. Lu, and A. Slominski, "A binary xml for scientific applications," in *First International Conference on e-Science and Grid Computing (e-Science)*, pp. 336–343, IEEE Computer Society, 2005.
- [129] S. Parastatidis and J. Webber, "MEP SSDL protocol framework," Tech. Rep. CS-TR-900, School of Computing Science, University of Newcastle upon Tyne, 2005.

- [130] E. R. de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, and J. Webber, "Secure and provable service support for human-intensive real-estate processes," tech. rep., School of Computing Science, University of Newcastle upon Tyne, 2006.
- [131] Microsoft Corporation, "Windows software development kit (sdk) for windows vista." <http://winfx.msdn.microsoft.com/>.
- [132] W3C, "XML information set (second edition)." <http://www.w3.org/TR/xml-infoset/>, 2004.
- [133] I. Foster *et al.*, "Modeling stateful resources with web services," tech. rep., Computer Associates International, Fujitsu, Hewlett-Packard, IBM and The University of Chicago, 2004.
- [134] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, "A grid application framework based on web services specifications and practices," tech. rep., North East Regional e-Science Centre, University of Newcastle upon Tyne, 2003.
- [135] J. Webber, "Message oriented web services (using WCF)." <http://jim.webber.name/presentations.html>, November 2006. ACS Web Services SIG.
- [136] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*. O'Reilly Media, 2004.
- [137] Microsoft Corporation, "The IExtensibleObject<T> pattern." <http://msdn2.microsoft.com/en-us/library/ms733816.aspx>.
- [138] M. Sipser, *Introduction to the Theory of Computation, Second Edition*. Course Technology, 2005.
- [139] D. Lea, S. Vinoski, and W. Vogels, "Guest editors introduction: Asynchronous middleware and services," *IEEE Internet Computing*, vol. 10, no. 1, pp. 14–17, 2006.
- [140] OASIS, "Web services reliable messaging (WS-ReliableMessaging)." <http://docs.oasis-open.org/ws-rx/wsrn/200608>, 2006.
- [141] H. Wirawan and W. Ekaslim, "Mapping of business standard to service oriented architecture (SOA)," tech. rep., School of Computer Science and Engineering, University of New South Wales, 2006.
- [142] Microsoft Corporation, ".NET framework 3.0." <http://www.netfx3.com/>.
- [143] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Trans. Inter. Tech.*, vol. 5, no. 4, pp. 660–704, 2005.
- [144] J. Bolie, M. Cardella, S. Blanvalet, M. Juric, S. Carey, P. Chandran, Y. Coene, K. Geminiuc, M. Zirn, and H. Gaur, *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development*. Packt Publishing, 2006.

- [145] *Oxford English Dictionary*. Oxford University Press, 2007.
- [146] M. Kloppmann, D. König, F. Leymann, G. Pfau, and D. Roller, “Business process choreography in websphere: Combining the power of BPEL and J2EE,” *IBM Systems Journal*, vol. 43, no. 2, 2004.
- [147] The Internet Engineering Task Force (IETF), “IPv6.” <http://www.ipv6.org/>.
- [148] W3C, “Xml path language (XPath) 2.0.” <http://www.w3.org/TR/xpath20/>.
- [149] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield, “SSDL - the SOAP service description language.” <http://www.ssd1.org/>.

# Appendix

## SSDL Contract with MEP Protocol

```
1 <ssdl:contract targetNamespace="http://www.lixix.org/Valuation/Contract"
2     xmlns:ssdl="urn:ssdl:v1"
3     xmlns:xi="http://www.w3.org/2001/XInclude">
4     <ssdl:schemas>
5         <xi:include href="http://www.lixix.org/Valuation/" />
6     </ssdl:schemas>
7     <ssdl:messages targetNamespace="http://www.lixix.org/Valuation/Message"
8         xmlns:ns1="http://www.lixix.org/Valuation">
9         <ssdl:message name="CancelValuationMsg">
10             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
11             <ssdl:body ref="ns1:CancelValuation" />
12         </ssdl:message>
13         <ssdl:message name="StatusMsg">
14             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
15             <ssdl:body ref="ns1:Status" />
16         </ssdl:message>
17         <ssdl:message name="StatusRequestMsg">
18             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
19             <ssdl:body ref="ns1:StatusRequest" />
20         </ssdl:message>
21         <ssdl:message name="ValuationRequestMsg">
22             <ssdl:body ref="ns1:ValuationRequest" />
23         </ssdl:message>
24         <ssdl:message name="FeeChangeRequestMsg">
25             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
26             <ssdl:body ref="ns1:FeeChangeRequest" />
27         </ssdl:message>
28         <ssdl:message name="FeeChangeRejectedMsg">
29             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
30             <ssdl:body ref="ns1:FeeChangeRejected" />
31         </ssdl:message>
32         <ssdl:message name="FeeChangeAcceptedMsg">
33             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
34             <ssdl:body ref="ns1:FeeChangeAccepted" />
35         </ssdl:message>
36         <ssdl:message name="ValuationResponseMsg">
37             <ssdl:header ref="ns1:Id" mustUnderstand="false" relay="false" />
38             <ssdl:body ref="ns1:ValuationResponse" />
39         </ssdl:message>
40     </ssdl:messages>
41     <ssdl:protocols>
42         <ssdl:protocol targetNamespace="http://www.lixix.org/Valuation/Protocol"
43             xmlns:mep="urn:ssdl:mep:v1">
44             <mep:out-only xmlns:ns2="http://www.lixix.org/Valuation/Message">
45                 <ssdl:msgref ref="ns2:CancelValuationMsg" direction="out" />
46             </mep:out-only>
47             <mep:in-only xmlns:ns2="http://www.lixix.org/Valuation/Message">
48                 <ssdl:msgref ref="ns2:StatusMsg" direction="in" />
49             </mep:in-only>
50             <mep:out-in xmlns:ns2="http://www.lixix.org/Valuation/Message">
51                 <ssdl:msgref ref="ns2:StatusRequestMsg" direction="out" />
52                 <ssdl:msgref ref="ns2:StatusMsg" direction="in" />
53             </mep:out-in>
54             <mep:out-optional-in xmlns:ns2="http://www.lixix.org/Valuation/Message">
55                 <ssdl:msgref ref="ns2:ValuationRequestMsg" direction="out" />
```

```

56         <ssdl:msgref ref="ns2:StatusMsg" direction="in" />
57     </mep:out-optional-in>
58     <mep:in-out xmlns:ns2="http://www.lixixi.org/Valuation/Message">
59         <ssdl:msgref ref="ns2:FeeChangeRejectedMsg" direction="out" />
60         <ssdl:msgref ref="ns2:FeeChangeRequestMsg" direction="in" />
61     </mep:in-out>
62     <mep:in-out xmlns:ns2="http://www.lixixi.org/Valuation/Message">
63         <ssdl:msgref ref="ns2:FeeChangeAcceptedMsg" direction="out" />
64         <ssdl:msgref ref="ns2:FeeChangeRequestMsg" direction="in" />
65     </mep:in-out>
66     <mep:in-only xmlns:ns2="http://www.lixixi.org/Valuation/Message">
67         <ssdl:msgref ref="ns2:ValuationResponseMsg" direction="in" />
68     </mep:in-only>
69 </ssdl:protocol>
70 </ssdl:protocols>
71 <ssdl:endpoints>
72     <ssdl:endpoint xmlns:wsa="http://www.w3.org/2004/12/addressing">
73         <wsa:Address>http://localhost:8081/caseOne/requestor/ws</wsa:Address>
74     </ssdl:endpoint>
75 </ssdl:endpoints>
76 </ssdl:contract>

```

---

## Sequencing Constraints Protocol

```
1 <ssdl:protocol targetNamespace="http://www.lixi.org/Valuation/Protocol"
2   xmlns:mep="urn:ssdl:sc:v1">
3   <sc:participant name="Req"/>
4   <sc:participant name="Val"/>
5   <sc:protocol name="Intermediary">
6     <ssdl:msgref ref="m:ValuationRequestMsg" direction="in" sc:participant="Req"/>
7     <sc:parallel>
8       <sc:protocolref ref="recurseReq"/>
9       <sc:multiple>
10        <sc:sequence>
11          <ssdl:msgref ref="m:ValuationRequestMsg" direction="out" sc:participant="Val"/>
12          <sc:protocolref ref="recurseVal"/>
13          <sc:choice>
14            <ssdl:msgref ref="StatusMsg" direction="in" sc:participant="Val"/>
15            <ssdl:msgref ref="CancelValuationMsg" direction="out" sc:participant="Val"/>
16            <ssdl:msgref ref="ValuationResponseMsg" direction="in" sc:participant="Val"/>
17          </sc:choice>
18        </sc:sequence>
19      </sc:multiple>
20    </sc:parallel>
21    <sc:choice>
22      <ssdl:msgref ref="StatusMsg" direction="out" sc:participant="Req"/>
23      <ssdl:msgref ref="CancelValuationMsg" direction="in" sc:participant="Req"/>
24      <ssdl:msgref ref="ValuationResponseMsg" direction="out" sc:participant="Req"/>
25    </sc:choice>
26  </sc:protocol>
27  <sc:protocol name="recurseReq">
28    <sc:choice>
29      <ssdl:msgref ref="m:StatusMsg" direction="out" sc:participant="Req"/>
30    </sc:choice>
31    <sc:sequence>
32      <ssdl:msgref ref="StatusRequestMsg" direction="in" sc:participant="Req"/>
33      <ssdl:msgref ref="StatusMsg" direction="out" sc:participant="Req"/>
34    </sc:sequence>
35    <sc:protocolref ref="recurseReq"/>
36    <sc:nothing/>
37  </sc:protocol>
38  <sc:protocol name="recurseVal">
39    <sc:choice>
40      <ssdl:msgref ref="m:StatusMsg" direction="in" sc:participant="Val"/>
41    </sc:choice>
42    <sc:sequence>
43      <ssdl:msgref ref="m:FeeChangeRequestMsg" direction="out" sc:participant="Val"/>
44      <sc:choice>
45        <ssdl:msgref ref="m:FeeChangeAcceptedMsg" direction="in" sc:participant="Val"/>
46        <ssdl:msgref ref="m:FeeChangeRejectedMsg" direction="in" sc:participant="Val"/>
47      </sc:choice>
48    </sc:sequence>
49    <sc:sequence>
50      <ssdl:msgref ref="StatusRequestMsg" direction="out" sc:participant="Val"/>
51      <ssdl:msgref ref="StatusMsg" direction="in" sc:participant="Val"/>
52    </sc:sequence>
53    <sc:protocolref ref="recurseVal"/>
54    <sc:nothing/>
55  </sc:protocol>
56 </ssdl:protocol>
```

## BPEL Process Description

```
1 <b:process name="intermediary"
2     abstractProcessProfile=""
3     targetNamespace="http://www.lixi.org/Valuation/bpel/Intermediary"
4     xmlns:tns="http://www.lixi.org/Valuation/bpel/Intermediary"
5     xmlns:b="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
6     xmlns:r="http://www.lixi.org/Valuation/wsd/Requestor"
7     xmlns:i="http://www.lixi.org/Valuation/wsd/Intermediary"
8     xmlns:v="http://www.lixi.org/Valuation/wsd/Valuer"
9     xmlns:lixi="http://www.lixi.org/Valuation">
10 <b:partnerLinks>
11 <b:partnerLink name="Requestor" partnerLinkType="i:RequestorPartnerLink"
12     myRole="Intermediary" partnerRole="Requestor"/>
13 <b:partnerLink name="Valuer" partnerLinkType="i:ValuerPartnerLink"
14     myRole="Intermediary" partnerRole="Valuer"/>
15 </b:partnerLinks>
16 <b:variables>
17 <b:variable name="ValuationRequest" messageType="lixi:ValuationRequestMsg"/>
18 <b:variable name="FeeChangeRequest" messageType="lixi:FeeChangeRequestMsg"/>
19 <b:variable name="FeeChangeAccepted" messageType="lixi:FeeChangeAcceptedMsg"/>
20 <b:variable name="FeeChangeRejected" messageType="lixi:FeeChangeRejectedMsg"/>
21 <b:variable name="StatusRequest" messageType="lixi:StatusRequestMsg"/>
22 <b:variable name="Status" messageType="lixi:StatusMsg"/>
23 <b:variable name="CancelValuation" messageType="lixi:StatusMsg"/>
24 <b:variable name="ValuationResponse" messageType="lixi:ValuationRequestMsg"/>
25 <b:variable name="ValuationFirms"/>
26 </b:variables>
27 <b:sequence>
28 <b:receive partnerLink="Requestor" portType="i:IntermediaryService"
29     operation="ProcessValuationRequest" variable="ValuationRequest"/>
30 <b:opaqueActivity>
31 <b:documentation>Get valuation firms and store in ValuationFirms.</b:documentation>
32 </b:opaqueActivity>
33 <b:flow>
34 <b:documentation>
35 Communicate with requestor and valuation firms independently and
36 concurrently. We break out of the loops when one of the following occurs:
37 - the requestor cancels the valuation
38 - all the valuation firms declined our request
39 - we receive the valuation response from a valuation firm
40 </b:documentation>
41 <b:while>
42 <b:condition opaque="yes">we haven't declined or sent response</b:condition>
43 <b:pick>
44 <!-- status request from requestor -->
45 <b:onMessage partnerLink="Requestor" portType="i:IntermediaryService"
46     operation="ProcessStatusRequest" variable="StatusRequest">
47 <b:reply partnerLink="Requestor" operation="ProcessStatusRequest" variable="Status"/>
48 </b:onMessage>
49 <!-- cancel valuation request from requestor -->
50 <b:onMessage partnerLink="Requestor" portType="i:IntermediaryService"
51     operation="ProcessCancelValuation" variable="CancelValuation">
52 <b:sequence>
53 <b:forEach counterName="ValuationFirms" parallel="yes">
54 <b:scope>
55 <b:invoke partnerLink="Valuer" portType="v:ValuerService"
56     operation="ProcessCancelValuation" inputVariable="CancelValuation"/>
57 </b:scope>
58 </b:forEach>
59 <b:exit/>
60 </b:sequence>
61 </b:onMessage>
62 </b:pick>
63 </b:while>
64 <b:sequence>
65 <!-- for each valuation firm -->
66 <b:forEach counterName="ValuationFirms" parallel="yes">
67 <b:scope>
68 <b:sequence>
69 <b:invoke partnerLink="Valuer" portType="v:ValuerService"
70     operation="ProcessValuationRequest" inputVariable="ValuationRequest"/>
71 <b:while>
```



```

72     <b:condition opaque="yes">
73         we haven't received status declined or valuation response from valuation firm.
74     </b:condition>
75     <b:pick>
76         <!-- status update from valuer -->
77         <b:onMessage partnerLink="Valuer" portType="i:IntermediaryService"
78             operation="ProcessStatus" variable="Status">
79             <b:if>
80                 <b:condition opaque="yes">
81                     if status is 'accepted' and we haven't cancelled yet.
82                 </b:condition>
83                 <!-- cancel all other valuation firms-->
84                 <b:forEach counterName="ValuationFirms" parallel="yes">
85                     <b:scope>
86                         <b:if>
87                             <b:condition opaque="yes">
88                                 if valuation firm is not the one that has just accepted.
89                             </b:condition>
90                             <b:invoke partnerLink="Valuer" portType="v:ValuerService"
91                                 operation="ProcessCancelValuation" inputVariable="CancelValuation"/>
92                         </b:if>
93                     </b:scope>
94                 </b:forEach>
95                 <b:elseif>
96                     <b:condition opaque="yes">if status is 'declined'</b:condition>
97                     <b:sequence>
98                         <b:opaqueActivity>
99                             <b:documentation>keep track of valuers that declined</b:documentation>
100                        </b:opaqueActivity>
101                    <b:if>
102                        <b:condition opaque="yes">if all valuers declined</b:condition>
103                        <b:sequence>
104                            <b:invoke partnerLink="Requestor" portType="r:RequestorService"
105                                operation="ProcessStatus" inputVariable="Status"/>
106                        </b:sequence>
107                    </b:if>
108                </b:sequence>
109            </b:elseif>
110        </b:if>
111    </b:onMessage>
112    <!-- fee change request from valuer -->
113    <b:onMessage partnerLink="Valuer" portType="i:IntermediaryService"
114        operation="ProcessFeeChangeRequest" variable="FeeChangeRequest">
115        <b:if>
116            <b:condition opaque="yes">some business condition</b:condition>
117            <b:invoke partnerLink="Valuer" portType="i:IntermediaryService"
118                operation="ProcessFeeChangeAccepted" inputVariable="FeeChangeAccepted"/>
119        <b:else>
120            <b:invoke partnerLink="Valuer" portType="v:ValuerService"
121                operation="ProcessFeeChangeRejected" inputVariable="FeeChangeRejected"/>
122        </b:else>
123    </b:if>
124    </b:onMessage>
125    <!-- valuation response from valuer -->
126    <b:onMessage partnerLink="Valuer" portType="i:IntermediaryService"
127        operation="ProcessValuationResponse" variable="ValuationResponse">
128        <b:sequence>
129            <b:invoke partnerLink="Requestor" portType="r:RequestorService"
130                operation="ProcessValuationResponse" inputVariable="ValuationResponse"/>
131        </b:sequence>
132    </b:sequence>
133    </b:sequence>
134    </b:pick>
135    </b:while>
136    </b:sequence>
137    </b:scope>
138    </b:forEach>
139    </b:sequence>
140    </b:flow>
141    </b:sequence>
142    </b:process>
143

```

---