



UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

School of Electronics and Computer Science

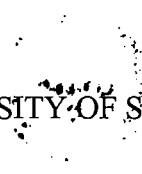
**Document Flow Model – A Formal Notation for Modelling Asynchronous Web
Services**

By

Jingtao Yang

Thesis for the degree of Doctor of Philosophy

May 2006



UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

School of Electronics and Computer Science

Doctor of Philosophy

Document Flow Model – A Formal Notation for Modelling Asynchronous Web Services

by Jingtao Yang

Web service architecture offers advantages over traditional architectures in business-level interoperability across organisations. It has been commonly believed to be the one technology to implement future enterprise systems. Thus the effort to develop and standardise web services is overwhelming. More than 100 web service (WS-*) specifications have been introduced in the past few years. The aim of this work is to bridge the industrial specifications and real implementations using formal models. A formal model is important here to provide a high level abstraction of the system behaviours that could be applied to various web service specifications. It also helps to understand a system and further analyse it by converting the model into code used by traditional model checking tools.

We introduce a new formal notation, Document Flow Model (DFM), to model asynchronous web service composition. Unlike other web service composition languages, our DFM not only captures the common service-oriented system behaviours, such as asynchronous communication, but also addresses the design issues of dynamic configurations, and long running business interactions in particular. In addition, we develop a formal operational semantics for the DFM specification describing the possible behaviours of a system composed of inter-related web services which helps to further analyse and simulate a system that is composed of asynchronous web services.

TABLE OF CONTENTS

ABSTRACT	i
TABLE OF CONTENTS	ii
LIST OF FIGURES	vi
LIST OF TABLES	viii
DECLARATION AUTHORSHIP	ix
ACKNOWLEDGEMENT	x
Chapter 1 Introduction	1
1.1 Research Motivation.....	2
1.2 Research Contributions	4
1.3 Research Methodology	5
1.4 Research Evaluation	7
1.5 Thesis Structure	8
Chapter 2 Background	10
2.1.1 Client-Server Architecture	11
2.1.2 P2P Architecture	12
2.1.3 Messaging System.....	12
2.2 Service-Oriented Architecture.....	13
2.2.1 What is a Web Service?	13
2.2.2 What is a SOA?.....	13
2.2.3 Supporting Standards and Technologies	14
2.2.4 WSDL	16
2.2.5 Web Service Composition.....	17
2.3 Workflow Technologies	21
2.3.1 Activity-Based Business Process Modelling.....	23
2.3.2 Message-Based Business Process Modelling.....	25

2.4	Formal Models and Related Works.....	26
2.4.1	UML Models.....	26
2.4.2	Process Algebra Approach.....	27
2.4.3	Petri Net Model.....	28
2.4.4	SPIN.....	29
2.4.5	Summary.....	29
2.5	Web Service Dynamism.....	30
2.5.1	Web Service Dynamic Behaviours.....	30
2.5.2	Related Dynamic Web Service Works.....	31
2.6	Our Services Composition Approach.....	32
2.6.1	Web Service Composition Requirement.....	32
2.6.2	Our Service Composition Approach.....	36
Chapter 3	Document Flow Model.....	39
3.1	What is DFM?.....	39
3.1.1	Supporting XML Data Structure.....	39
3.1.2	Using Context Coordination Mechanism.....	40
3.1.3	Modelling Asynchronous Communication.....	41
3.2	Formal Syntax.....	44
3.2.1	The Basic Structure.....	45
3.2.2	The Message Definition Body.....	45
3.2.3	Actions.....	46
3.2.4	Conditions.....	47
3.2.5	Control Flow.....	47
3.2.6	XML Data Structure.....	48
3.2.7	ContextStore.....	50
3.2.8	Keywords.....	50
3.3	An Example.....	51
3.3.1	A Travel Agent System.....	52
3.3.2	Summary.....	56
Chapter 4	Illustration.....	58
4.1	An Example of Dynamic Replacing a Service.....	58
4.1.1	Service Coordination.....	58
4.1.2	A Job Submission System.....	59

4.1.3	A Re-Configured Job Submission System.....	63
4.1.4	Demonstration.....	66
4.2	Discussion.....	69
4.2.1	DFM Message Flow Patterns.....	70
4.2.2	Modelling Workflow Patterns.....	72
Chapter 5	Comparison.....	77
5.1	A BPEL Example.....	77
5.2	The BPEL4WS Model.....	79
5.3	The DFM Model.....	82
5.3.1	A Purchase Process Specification.....	83
5.3.2	A Shipping Service Specification.....	85
5.3.3	An Invoicing Service Specification.....	85
5.3.4	A Scheduling Process Specification.....	86
5.4	BPEL4WS and DFM.....	87
5.4.1	Formal Model vs. Industrial Standard.....	87
5.4.2	Distributed Workflow vs. Centralised Workflow.....	88
5.4.3	Support for Long-running Interactions.....	90
5.4.4	Summary.....	90
5.5	WSCI and DFM.....	91
5.5.1	A Travel Agent Example in WSCI.....	92
Chapter 6	Formal Semantics.....	93
6.1	Operational Semantics.....	93
6.1.1	Specification Functions.....	94
6.1.2	Semantic Functions.....	97
6.1.3	System Configurations.....	101
6.1.4	A Transition.....	101
6.2	Discussion.....	103
6.2.1	Dynamic Configuration Scenario.....	103
6.2.2	Summary.....	108
Chapter 7	Conclusion and Future Work.....	109
7.1	Research Contribution.....	109
7.2	Research Evaluation.....	114
7.3	Future Work.....	117

7.3.1	Potential Improvement on the Operational Semantics	117
7.3.2	A Simulation Tool.....	118
7.3.3	BPEL4WS Formal Verification	118
Appendices.....		120
Appendix A An Investigation of Asynchronous Invocations Using Servlet and JSP		120
Appendix B A Travel Agent Implementation Using XSLT and SOAP		120
Appendix C A JavaScript DFM Model		121
Appendix D Formal Verification of DFM using ARC		121
Appendix E A BPEL4WS Implementation using IBM BPWS4J		122
Appendix A An Investigation of Asynchronous Invocations Using Servlet and JSP		123
Servlet and JSP		123
A Synchronous Solution.....		124
An Asynchronous Solution.....		125
Conclusion.....		127
Appendix B A Travel Agent Implementation Using XSLT and SOAP		129
Functional Programming Language and XSLT.....		129
Java SOAP APIs.....		138
Appendix C A JavaScript DFM Model.....		140
Why JavaScript?		140
A JavaScript DFM Tool		141
DFM in JavaScript.....		143
Appendix D Formal Verification of DFM using ARC.....		151
What is ARC?.....		151
DFM ARC Model.....		152
DFM ARC Model Verification.....		157
Appendix E A BEPL4WS Implementation using IBM BPWS4J		159
BPWS4J.....		159
A Job Submission Business Process.....		161
Bibliography.....		168

LIST OF FIGURES

Figure 2-1 Web Service Architecture I.....	13
Figure 2-2 Web Service Architecture II.....	14
Figure 2-3 Web Service Standard Stack	15
Figure 2-4 A Orchestrated BPEL Service Composition	18
Figure 2-5 A WSCI Choreography Service Composition	20
Figure 2-6 An Abstract Workflow Management Model.....	22
Figure 2-7 Activity – based Workflow	24
Figure 2-8 Interaction State in Messages.....	34
Figure 2-9 Interaction State in Database.....	36
Figure 3-1 A Synchronous Communication	42
Figure 3-2 An Asynchronous Communication	43
Figure 3-3 A DFM Control Flow Example.....	48
Figure 3-4 A Simple Travel Agent Sequence Diagram	52
Figure 3-5 A Travel Agent Specification – I	53
Figure 3-6 A Shop Specification.....	54
Figure 3-7 A Travel Agent Specification – II.....	55
Figure 3-8 A Document Flow Chart	55
Figure 4-1 A Simple Job Submission Sequence Diagram	60
Figure 4-2 A Job Submission Example	60
Figure 4-3 A FlowService Specification.....	61
Figure 4-4 A JobService Specification	62
Figure 4-5 A Coordination Service Specification.....	62
Figure 4-6 A Re-configured Job Submission Example	64
Figure 4-7 Updated FlowService Specification.....	65
Figure 4-8 The <i>In</i> Programming Pattern.....	70
Figure 4-9 The <i>In-Outs</i> Programming Pattern	71
Figure 4-10 The <i>Ins-Outs</i> Programming Pattern.....	72
Figure 4-11 Sequence Workflow Pattern.....	73
Figure 4-12 Parallel Split Workflow Pattern	73
Figure 4-13 Synchronisation Workflow Pattern	74

Figure 4-14 Exclusive Choice Workflow Pattern	74
Figure 4-15 Exclusive Choice DFM Model.....	75
Figure 4-16 Simple Merge Workflow Pattern	75
Figure 4-17 Simple Merge DFM Model.....	75
Figure 5-1 A BPEL4WS Example.....	77
Figure 5-2 A Purchase Process in BEPL4WS	80
Figure 5-3 The Sequence Diagram of BPEL4WS Example	82
Figure 5-4 A Purchase Process Specification	84
Figure 5-5 A Shipping Service Specification	85
Figure 5-6 An Invoicing Service Specification	86
Figure 5-7 A Schedule Service Specification	86
Figure 5-8 A TravelAgent Example in WSCI	91
Figure 6-1 A Message Pool	97
Figure 6-2 A Transition Rule.....	101
Figure 6-3 A Travel Agent Transition Diagrams.....	102
Figure 6-4 A Service Flow Example	105
Figure 6-5 A System Flow Example.....	105
Figure 6-6 Replace a Service Locally.....	106
Figure 6-7 Replace a Service Global	107
Figure A-1 Synchronous Travel Agent Communication	124
Figure A-2 Asynchronous Travel Agent Communication.....	126
Figure A-3 Travel Agent System Configuration and Trace.....	127
Figure C-1 A JavaScript DFM Checker.....	142
Figure D-1 ARC GUI	151
Figure D-2 ARC Verification Result	158
Figure E-1 A BPWS4J GUI.....	159
Figure E-2 Deployed Business Processes	160
Figure E-3 Our Job Submission Process.....	161

LIST OF TABLES

Table 3-1 DFM Meta-Symbols.....	44
Table 3-2 DFM Keywords.....	51
Table 4-1 Message Examples of a Re-configured Job Submission.....	68
Table 4-2 Re-configured Job Submission Interaction State.....	69
Table 5-1 BPEL Example - Supporting Services	78
Table 5-2 A Centralised Workflow Model	88
Table 5-3 A Distributed Workflow Model	89
Table 6-1 A Configuration Table of a Service, Si	98

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisors, Prof. Peter Henderson and Dr. Corina Cirstea, for their invaluable guidance, encouragement and patience through the completion of the work presented in this thesis. I am very thankful to them for teaching me how to conduct good research and for sharing me their knowledge and research ideas generously.

I would also like to acknowledge the help, support and friendship which I have received from the people in the Declarative System and Software Engineering research group, University of Southampton.

At last, I would like to thank my parents who give their whole life for the family. No words would be enough to express my guilty for could not be there with you in the critical time. I cannot reach any of the achievements without your love and supports. This thesis is dedicated to you, my beloved parents.

Chapter 1

Introduction

Enterprise systems are the outcome of the fast developed Internet technology. Such technology enables computer systems go across the boundary of an organisation freely. An enterprise as a business unit may perform in different business roles and offer diverse services on the demand of their business partners and customers. Various new enterprise business patterns have been introduced by computer scientists in the past few years. As the enterprise systems continue to become larger and more complex, issues have to be solved to compliant the rapid business requirement change.

First, an enterprise system is about business to business interactions. Modern Internet environments are heterogeneous [1]. Computer systems are developed by different programming languages and running on various platforms. The business level interoperability is essential to design and implement enterprise systems.

Web service technology is developed to achieve universal interoperability between applications by using web standards [2]. In a service-oriented architecture, a web service is a specific piece of functionality that can be accessed by other services or clients through a contractually specified interface [3]. Unlike object-oriented architecture, in which components of a system are accessed via object-model-specific protocols such as the Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP), web services are accessed via ubiquitous web protocols and data formats, such as Hypertext Transfer Protocol (HTTP), Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP) [4]. Using platform independent and standard XML documents, a service consumer can invoke a service following the shared understanding but does not care how the service is implemented which fits into the modern heterogeneous web environment.

One of the challenges in current enterprise systems is how to build business-to-business interaction, adapting to the dynamic changed business environment. The distributed web services in an enterprise system are required to be changed or updated without recompiling and replacing the whole system. Meanwhile, those web services should be able to plug-in and plug-out to the living systems without affecting any normal system behaviour and operations [5].

Correctness and integrity are of course vital for the enterprise systems. Service-oriented architecture offers great advantages in business level interoperability, but it also adds difficulties to the system implementation and validation. Business-to-business interactions are executed on top of advanced middlewares. The completion and correctness of a business interaction rely on both implementations of individual services and low level networking services.

Modelling is a formal method that is used to design software or systems before coding. “Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make” [6].

Enterprise systems are complicated [Section 2.1]. Extending current modelling capabilities to specify business interactions in a service-oriented enterprise system is not only an applicable way to validate systems at design level but also a potential way to build up tools so that non-developer can specify systems in script and generate systems automatically.

1.1 Research Motivation

The motivation of this work is to investigate modern service-oriented enterprise system environments, and use formal modelling to capture a service-oriented business interaction as a workflow specification, in order to facilitate service-oriented system automation, and validation.

Service-oriented systems enable independent distributed web services to interact through asynchronous messages. These services “...whether on the same host, the same network, or

loosely connected through the Internet, use messaging to pass data and to coordinate their respective functions” [7]. This kind of loosely-coupled architecture brings complexity to the implementation and verification.

While traditional activity-based modelling approaches have been successfully used in research and commercial systems for decades, it is not expressive enough to model modern service-oriented system in some aspects. An activity-based modelling approach relies on the organisation structures, roles and relationships. Whereas, a modern enterprise application manages business processes cross organisations; and structures of an organisation are dynamically changed. Moreover activity-based modelling approaches assume components in a system are tightly-integrated, in contrast web services are loosely-coupled by the asynchronous messages in a service-oriented system.

The Business Process Execution Language for Web Services (BPEL4WS) specification has been positioned to be the key standard to specify modern service-oriented business processes. It conjoins web service messaging features with the activity-based modelling approach. Currently, most workflow works [8, 9] leverage the concepts from BPEL4WS, and focus on workflow patterns. However the BPEL4WS separates a workflow from web services involved in its execution, by using a pre-defined workflow instance with full interaction states communicating with stateless web services [10]. The life cycle management concerning long-running interactions and dynamic service configurations has not yet been considered. In respect that modern enterprise applications are much more complex: interactions run for long periods (days, weeks); a component is required to be dynamically replaced by another component or created into the system while some interactions are still active [11].

Experiments [5, 30, Appendix E] show that business process states are necessarily to be maintained consistently, persistently and separately in certain circumstance if dynamic business interactions run over long periods of time. Thus a coordination framework is required to manage the interactions between stateless business components and state components in order to model and specify such dynamic long-running business interactions.

Context is a mechanism used in collaborative distributed applications. A context allows a component to share information such as message correlation and security token and so on. Extension to the basic context operations, a context could be used as a contract shared by

stateless distributed components (web services) and state maintaining components participated in the same business interactions.

Our modelling approach extends traditional activity-based approach with the capability to model the behaviour of asynchronous web service interactions. A context coordination framework is introduced to provide a mean to support long-running business interactions and dynamic behaviours.

1.2 Research Contributions

The work presented in this thesis contributes the state of the art in the followings:

1. Designing and implementing web service oriented system is challenging. We review the common service-oriented system behaviours and address the design issues of dynamic behaviours, and long running business interactions in particular. We introduce a design notation Document Flow Model to reason those behaviours and business interactions.
2. The Document Flow Model uses a novel message-based approach to model service-oriented business interactions. It exhibits the following properties:
 - a. An XML-convertible notation. The document record data structure is invented which describes a tree data structure in a concise notation. It could be converted into XML data structure which could be further fitted into other web service standards.
 - b. Modelling asynchronous communications. Two kinds of communication patterns are supported, one-way, which amounts to a service receiving a message, and notification which amounts to a service sending a message. Any complex and structured asynchronous communication could be decomposed to simple communication using the two patterns.
 - c. Supporting long-running interactions and dynamic configurations. A coordination framework is used in DFM including: a context to identify a business interaction state; a decentralised context propagation mechanism to

structure interaction related data; a persistent component, a ContextStore, to maintain the interaction state.

3. A novel formalisation (a formal operational semantics) is also developed for the DFM specification: It describes the possible behaviours of a system of inter-related web services, in terms of the messages that can be exchanged during the execution of one or more business interactions, and the effect each message execution has on the business interaction states.

The work presented in this thesis has resulted in the following publications:

- “Reusable Web Services”, Peter Henderson and Jingtao Yang, in proceeding of 8th International Conference on Software Reuse, ICSR 2004, LNCS 3107, Madrid, Spain, 2004. This paper supports the above contribution points (1 and 2).
- “Document Flow Model: A Formal Notation for Modelling Asynchronous Web Services Composition”, Jingtao Yang, Corina Cirstea and Peter Henderson, in proceeding of the OnTheMove Workshops 2005, LNCS 3762, Agia Napa, Cyprus, 2005. This paper supports the above contribution point (2).
- “An operational semantics for DFM: a formal notation for modelling asynchronous web service coordinations”, Jingtao Yang, Corina Cirstea and Peter Henderson, in proceeding of the 5th International Conference on Quality Software, QSIC2005, Melbourne, Australia, 2005. This paper supports the above contribution point (2 and 3).

1.3 Research Methodology

Developing large scale distributed systems is expensive. To reduce design cost, it is necessary to make sure that system design is consistent with the system requirements before the implementation. One way to do this is to build an abstract model of the system. Reasoning about the abstract model helps designers understand the consequence of the system and check whether the design satisfies the requirements. Moreover, the variations of the abstract model shall be used in experiments to verify that it really achieve all system objectives. We create a new abstract model to design service-oriented enterprise systems. The model is expressive enough to capture service-oriented system behaviours and also simple enough for non-IT

experts to design enterprise systems. Meanwhile, it provides the capabilities of formal reasoning and validation.

A language is needed to define the abstract model, either an existing language or a new language specifically created for the abstract model. Creating a new language is more challenging than using an existing one. However, a new language can be more suitable for the abstract model and the targeting applications, by incorporating capabilities to describe specific system behaviours.

In this work, we aim to specify the behaviours in the new service-oriented architecture. We examine other related formal models and indicate why they are not expressive enough to model the new service-oriented architecture [Section 2.4] [Section 2.5] [Section 2.3.1]. Therefore, we invent a new language to capture the service-oriented system behaviours, such as loosely-coupled connectivity, stateless web service nature, long-running business interactions, dynamic configurations and so on [Chapter 3]. Moreover, the language is simple but descriptive to model the content of the messages.

To facilitate the use of the abstract model, it is desirable to give a meaning to the language and demonstrate its utility. The popular way to give the language meaning is by defining mathematical semantics, but it is difficult for a system designer to understand the highly abstract mathematic formulas. It is also not straightforward to implement tools or systems using those abstract formulas.

In the abstract of this thesis, it has been emphasised that this work aims to bridge the industry specifications and mathematic models. Hence a traditional descriptive semantics is developed for our language [Chapter 6]. On one hand, it captures the system characteristics that can not be captured by the mathematic semantics, for example our operational semantics captures the loosely-coupled system behaviours by separating the concerns of message deliveries and message executions; on the other hand, it is closer to the real system implementations and well serves the motivation of this work.

There are two ways to demonstrate the utility of a language. One way is to apply the language to a number of distinct, non-trivial applications. A good language should be applicable to a variety of applications.

In this thesis, we demonstrate that our language can specify typical service-oriented systems [Chapter 3], hierarchical grid applications [Chapter 4] and comprehensive BPEL4WS [Chapter 5] applications. Specifically, the travel agent example in Chapter 3 shows that our language can specify a long-running distributed system that is composed of autonomous web services. The re-configurable job submission application in Chapter 4 provides a complete novel solution to solve the plug-and-play design issue in grid applications. It allows a system to be easily expanded and reduced depending on the availabilities of network resources. In the warehouse purchasing example in Chapter 5, a single business interaction includes ten message exchanges, and is carried out by five independent web services (or business processes).

Another way to demonstrate the utility of a language is by comparison. It is difficult to say that one language is better than the other without considering applications.

In this work, we address new design issues [Chapter 2] in web service oriented applications. In Chapter 5, we compare our DFM language with the BPEL4WS specification using the warehouse purchasing example. We examine the two models from different aspects, for example the support for long-running business interactions, the support for dynamic behaviours, and advantages of two different architectures. We also compare our language with the Web Service Choreography Interface (WSCI) specification using the travel agent example. By comparing our language with related works, the benefits of our work in the design issues we have raised are much clearer [Chapter 5].

1.4 Research Evaluation

This section summarises the evaluation methods we have used in this work.

In this work, we developed a new modelling language to design dynamic web service compositions in loosely-coupled web environments. The modelling language has been validated in different aspects.

Firstly, we demonstrated the utility of our language by applying it to a number of non-trivial applications [Chapter 3] [Chapter 4] [Chapter 5]. The diversity of applications validates the usage of the language. Similar evaluation approach which is “through example compositions published by IBM” has been done in other research work [43].

Secondly, we demonstrated our modelling language by comparing it to a number of previous works. The comparison [Chapter 5] clearly shows the advantages of our work against the previous ones.

Thirdly, we validated our modelling language by providing its operational meanings [Chapter 6]. The operational semantics [40] proves that our language solves the architecture issues declared when designing our language, for example, asynchronous communication, loosely-coupled environment, and dynamic behaviours.

Finally, we validated our work by implementations. By compiling our model into ARC model [Appendix D], we demonstrated that our language could be used by existing formal model checking tools. A partial JavaScript message tool simulation [Appendix C] shows that our operational semantics could be developed to validate loosely-coupled applications.

This work aims to solve new architecture issues that have not been covered by traditional languages. We have validated language through comparison, utility, formalisation, and partial implementation. The extension and improvement have been presented in the future work [Chapter 7].

1.5 Thesis Structure

The remainder of the thesis is structured in the following manner:

Chapter 2 presents a literature review of workflow and web service technologies. This chapter gives the motivation of our research by analysing the limitation of traditional process modelling approach and reviewing the different approach of current industrial web services standards.

Chapter 3 presents the formal syntax and informal semantics of our Document Flow Model (DFM) notation. It describes the concept and features of the notation and uses a simple example to explain the use of the notation.

Chapter 4 continues to illustrate the use of the notation. This chapter first uses a typical grid example to demonstrate the DFM capability to support long-running interactions and dynamic configurations, and then summarises the DFM programming patterns and explains how to use those patterns to model typical workflow patterns.

Chapter 5 compares our DFM with current industrial web service composition standards, BPEL4WS, using an example borrowing from the BPEL4WS specification, and briefly discusses the difference between WSCI and DFM.

Chapter 6 presents a formal operational semantics of the Document Flow Model. This chapter describes the abstract machine composed by asynchronous web services, and gives the rule governing the execution of the system. It also discusses the operational rules of dynamic configuration behaviours.

Chapter 7 draws some conclusions from our research and discusses the future work.

Chapter 2

Background

This work contributed in formal models of web service oriented enterprise system. Thus in this Chapter, we introduce the background regarding enterprise system architecture and provide an overview of the previous research reported in the literature. Firstly we review the architecture support for modern enterprise systems and introduce the new web service-oriented architecture. Then we study the traditional workflow technology, formal models and analyse why they are not suitable for the new service-oriented architecture. Finally, we review other web service composition works, summarise the dynamic service composition requirements, and give the basis of our service composition approach. We will not discuss software process modelling technology in this work because we have reviewed the related conferences and workshops like ICSP, ISPW, EWSPT and believe they lack support of formal models and web service technologies.

Enterprise System Review

IT Technology not only assists real world business but also creates new business patterns. Using SQL and client-server platform, a distributed system helps an organisation integrate its separate data sources and business logics together. What is next? Can we create a new business pattern by using the functionalities provided by different organisations? Enterprise system is used to describe this kind of business pattern.

An enterprise system is distributed, large-scale and cross-organisational. Efforts are needed from both industry and academia to build such systems, which include creating new architectures to design the system, constructing platforms to run the system, using new formal models to facilitate the system engineering and so on.

An enterprise system is a composition of components from various organisations. These components are used to serve their own organisational systems. We can not always compromise a legacy system to create a new system. Therefore, the new architecture has to maintain the legacy system so that the components could be used by both the legacy system and the new cross-organisational system.

An enterprise system has to be open and independent of any platforms [44]. Applications are built on heterogeneous web environment. Organisations develop their traditional systems using different software and programming languages. The new platform will support the components interoperability by providing interaction protocols, and common components interfaces.

An enterprise system has to be easily integrated. A tightly integrated system normally tremendously reduces the reusability and dynamism of the system and its components. This is exactly what enterprise systems try to avoid. Furthermore, the dynamism is mandatory in an enterprise system which allows the components to be recomposable and reconfigurable. Complex mechanism in the new architecture has to be avoided so that the system could be easily re-integrated to adapt the constantly changing business environment.

To build a comprehensive and robust enterprise system, software industry and academia have to facilitate each other and provide support on architecture, platform and system engineering. Reviewing the works that have been done all the levels to achieve this goal, we can see that there is no suitable notation for the business analyst to design the system [45]. The problem happened because, on one side, industry provides comprehensive programming and specification languages to implement enterprise system; while on the other side, academia uses highly abstract mathematic models to reason and verify the system. We can not expect a business analyst to design a system using specification languages which is normally done by the IT specialist. Some academic researchers also found it is not straightforward to verify most of the industry specifications [46]. Intermediation is urgently required to fill this gap.

2.1.1 Client-Server Architecture

The client-server architecture is used to describe a system that user interfaces (clients) and business logics (servers) are located at different network places. The development of the SQL technology allows an application to separate its business data from business logics, so that client-server architecture can be presented as 3 tiers: a presentation tier (user interfaces), a logic tier (functional logics) and a data tier (a database or a file system).

The client-server architecture has been widely used in large-scale organisational applications, for example large-scale e-commerce system. The contributions of this architecture are distribution and modular design. The disadvantage is that client-server applications largely

reply on the business logics on the server side. A centralised system architecture sometimes encounters server problems such as scalability, availability and security [47].

2.1.2 P2P Architecture

P2P, Peer-to-Peer, architecture refers to a system composed of a number of equivalent distributed components. This differs from client-server architecture where a component behaves as a server and a client simultaneously. P2P architecture has been used in network resource sharing. Napster is a famous P2P online music sharing system. A user accesses music resources stored in file systems of distributed computers. Grid computing is another example of P2P. Instead of sharing file systems, grid computing shares CPUs. A computer within a grid can join a computation process when its CPU is free. It is extremely useful in scientific project, where the computation tasks are huge.

P2P architecture distributes system responsibility to a group of peer computers, significantly releases the heavy workload on a single computer. The disadvantage of this structure is that collaboration is needed among all peer computers.

2.1.3 Messaging System

A messaging system is basically a P2P system where components interact with peer components by messages. It introduces the loosely-coupled design pattern into modern distributed systems.

Traditional distributed systems are tightly-integrated, a component of a system control other components; the operation of one component may rely on the results of operations taking place in other components. In contrast, components in a loosely-coupled system are much more independent. Components basically process on their own demand. The system is more reliable than traditional tightly-integrated systems, because the failure of one component will not result in the failure of the whole system.

With message-oriented middleware supports, a distributed system can largely increase the system scalabilities, and allow thousands of components interacting with each other simultaneously. Messaging systems separate the concerns of the components functionality from message delivery, thus such systems are more flexible to change their components.

2.2 Service-Oriented Architecture

Service-Oriented architecture is a new architecture for modern enterprise systems. It uses standardised interfaces and ubiquitous web protocols, so that applications running on heterogeneous web environment can interact with each other. In this section, we introduce the service-oriented architecture, fundamental standards and review some related industrial specifications.

2.2.1 What is a Web Service?

A web service is an autonomous entity that provides an interface to describe a collection of operations that are network-accessible through standardised XML messaging. In the W3C Web Services Architecture specification, web service connections are stateless, that is all the data for a given request must be in the request. Therefore, a web service is essentially a well-defined, self-contained function, and does not depend on the context or state of other services [16].

2.2.2 What is a SOA?

A basic service-oriented architecture comprises a service consumer and a service provider. The service consumer sends a service request message to the service provider. The service provider returns a response message to the service consumer. The request and subsequent response connections are defined in some way that is understandable to both the service consumer and service provider.

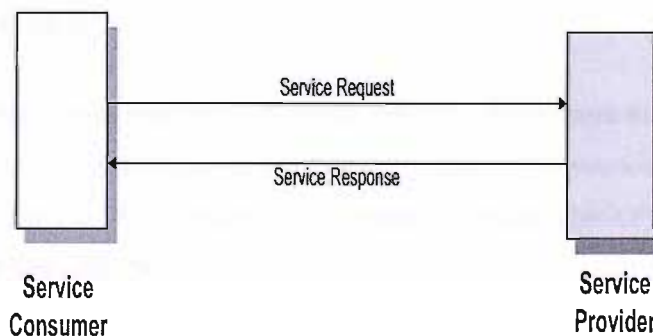


Figure 2-1 Web Service Architecture I

A complex service-oriented architecture consists of a collection of consumers and providers. A service provider can sometimes behave like a service consumer, and a service consumer also could also be a service provider. Each service stands somewhere in the Internet. To assemble services, a registry service is required in the SOA.

A service provider publishes its service descriptions with a service registry; a service consumer then queries and finds published services at registry; subsequently the service consumer can directly bind to the service and send a service request.

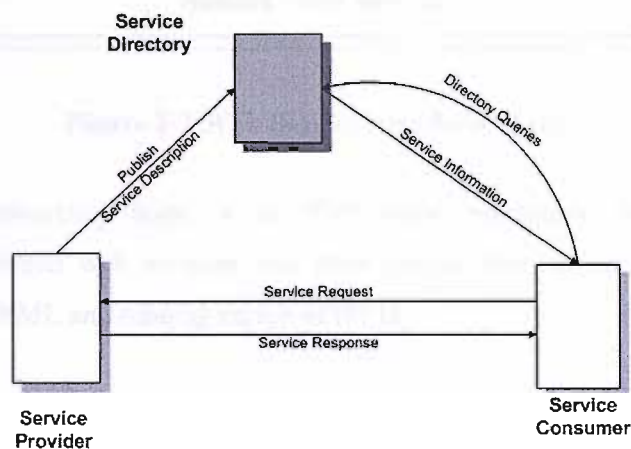


Figure 2-2 Web Service Architecture II

2.2.3 Supporting Standards and Technologies

We have seen the abstract concept of a web service and service oriented architecture. Now we give a brief introduction about supporting standards and technologies from the implementation point of view.

All the required web services must be network accessible. The network is the foundation layer for the web services. The network is normally based on an HTTP protocol, but other kinds of network protocols, such as the Simple Mail Transfer Protocol (SMTP), Internet Inter-Orb Protocol (IIOP), are also used [4].

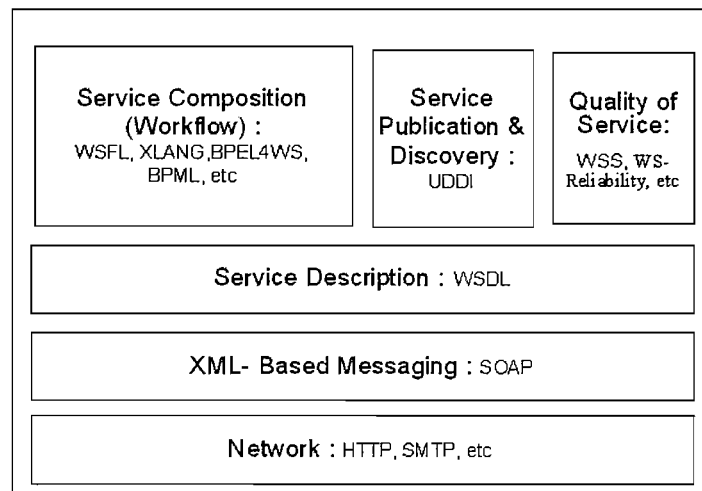


Figure 2-3 Web Service Standard Stack

On top of the networking layer is an XML-based messaging layer that facilitates communications between web services and their clients. Simple Object Access Protocol (SOAP) is based on XML and running on top of HTTP.

WSDL (Web Service Description Language) is a specification that describes web services. Again this is an XML-based service description of how to communicate using the web service; namely the protocol bindings and message formats required to interact with the web services listed in its directory. The supported operations and messages are described abstractly, and then bound to a concrete network protocol and message format [42].

These three layers are the fundamental of SOA. Additional technologies can be placed on top of these layers in order to meet different business requirement.

UDDI, (Universal Description, Discovery, and Integration) is a platform-independent, XML-based registry for web services [17]. It supports web service publication and discovery. A service provider sends the WSDL to registry, so that a service consumer can query the registry and using WSDL.

To provide Quality of Service, various specifications are proposed to satisfy security, Web Service Security (WSS), and reliability (WS-Reliability) requirements.

A high-level web service is a composition of web services. This composition is captured as a workflow specification. Key standards on this level are XLANG, WSFL, and BPEL4WS and so on. We will review them a little later.

2.2.4 WSDL

As the basis of all web service specifications, a WSDL definition is an XML document used to describe a Web Services interface. It also defines how web services are bound to specific network addresses. In WSDL, a web service is a network endpoint, a *port*. There are five parts in a WSDL definition.

- **Types:**

`<type>` defines data types used in the message declarations. Data types are machine-, language-independent and are based upon some agreed upon XML vocabulary.

- **Messages:**

`<message>` defines the data elements of operations. Each message consists of one or more parts. The parts can be compared to the parameters of a function call in a traditional programming language.

- **PortTypes:**

`<portType>` is the most important WSDL element. A *port type* describes service supported *operations* that are internal actions when a service operation is invoked. An operation can be compared to a function in a traditional programming language where the input and output messages correspond to function input and output parameters. There are four types of operations [3]:

1. One-way: The operation receives a message but will not return a response
2. Request-response: The operation receives a message and will return a response message
3. Solicit-response: The operation sends a request message and will wait for a response message

4. Notification: The operation sends a message but will not wait for a response message.

- **Binding:**

<binding> defines how message are transmitted, and the location of the service. A service binding connects port types to a port. A port is defined by associating a network address with a port type. A collection of ports defines a service. This binding is commonly created using SOAP, but other forms may be used.

- **Service:**

<service> is used to group related endpoint services together.

2.2.5 Web Service Composition

In early sections we reviewed the workflow technology, described why traditional activity-based business process modelling is no longer expressive enough for the new web services oriented environment, and gave a list of requirements for modern loosely coupled applications using messages. In this section, we review current established service-oriented business process modelling approaches.

Web service technology aims to achieve universal interoperability between applications by using web standards [2]. We have reviewed web service architectures and supporting standards. The network standards, XML-based messaging standards and service description languages provide the underlying platform for service to service interactions. However, a service-oriented application is composed by various independent web services which are provided by multiple organisations. The composition protocols (standards) are required to enable the integration of these services.

Currently, there are basically two approaches to achieve web service composition: Orchestration and Choreography. Orchestration describes web service interactions at the message level: the business logic and the execution of an interaction are controlled by one of the business parties, the process [18]. The well-known standard using orchestration approach is BPEL4WS [2]. In contrast, choreography describes the messages exchanged between web services. There is no central control of any interaction. Each party involved in the interaction

has to specify its only part of the control. WSCI [20] is an example of choreography web service composition.

WSAH (Web Services Acronym Hell) is a new web service acronym widely spread recently. Here we only review the BPEL4WS and WSCI which are the most commonly acknowledged standards and most related to our work.

2.2.5.1 Business Process Execution Language for Web Services (BPEL4WS)

BPEL4WS is a converging standard of XLANG from Microsoft and WSFL from IBM. XLANG is a XML-based specification language used to describe internal executable business processes to support service public collaborative processes. It focuses on the creation of business processes and the message exchange behaviours among web services. While WSFL, Web Service Flow Language, describes a public (global model) flow that defines data exchange and execution sequence of functions, and a private (flow model) flow that defines how composed web services interact with each other.

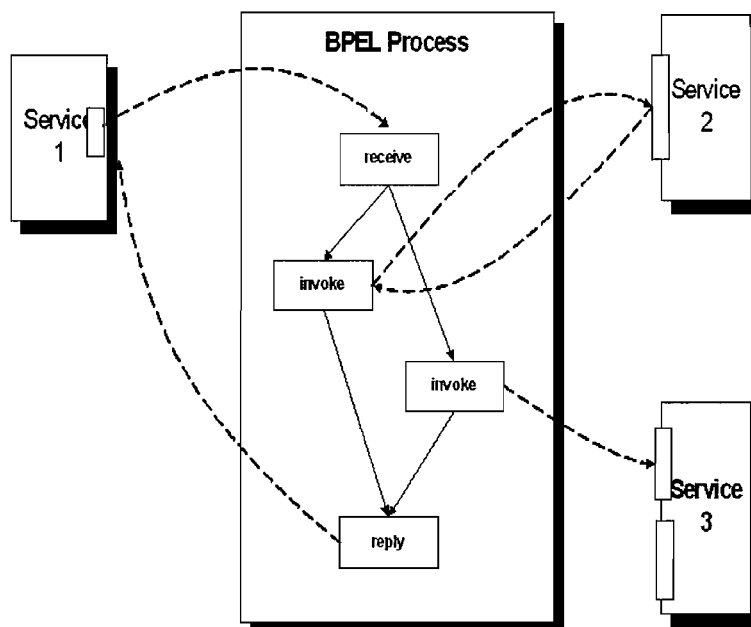


Figure 2-4 A Orchestrated BPEL Service Composition

Lately, two companies combined them and released BPEL4WS (Business Process Execution Language for Web Services) specification together with Siebel Systems, BEA and SAP.

BPEL4WS supports both private and public flows. The private flow models the behaviour of business partners in a specific business interaction. The public flow is a business protocol describing message exchanges between parties.

BPEL4WS introduces the concept of process-oriented form of service composition, whereby each BPEL4WS composition is a business process that interacts with a set of web services to achieve a certain goal [19]. Essentially, a process is a web service that supports WSDL interfaces, interacts with partners by invoking operations they support and exchanges messages through public interfaces. We can see that a BPEL service composition uses a stateful interaction model that allows web services to exchange sequences of messages between different business partners.

BPEL4WS defines two kinds of activities, basic activities and structured activities [2].

Basic activities are actions when a process interacts with external services or processes, including:

- *invoke*: invoking an operation of a web service.
- *receive*: waiting for messages from a web service.
- *reply*: sending a response to a request previously accepted through a *receive* activity.
- *assign*: assigning a value to a variable.

Structured activities indicate the order in which a collection of basic activities take place. These activities support control patterns, data flow, handling of faults and external events and so on. All structured activities can be recursively combined. Several structured activities are defined including:

- *sequence*: defines the sequential of execution of activities.
- *flow*: defines the parallel execution of activities.
- *pick*: selects an execution path based on a set of conditions.
- *switch*: supports multi-choice patterns.

- while: defines an iterative activity.

2.2.5.2 Web Service Choreography Interface (WSCI)

The WSCI is a specification from Sun, SAP, BEA, and Intalio that defines an XML-based language for web services collaboration. WSCI describes the observable behaviour of a web service. It does not address the definition and the implementation of the internal process that actually drive the message exchange [20]. The specification defines a service composition by describing each party's involvement in each interaction. And a WSCI interface describes a party's participation in a message exchange.

The WSCI builds on top of SOAP and WSDL standards. It extends the WSDL in a way that correlating the operations of a web service. It describes the flow of messages exchanged by a web service participating in choreographed interactions with other services [21].

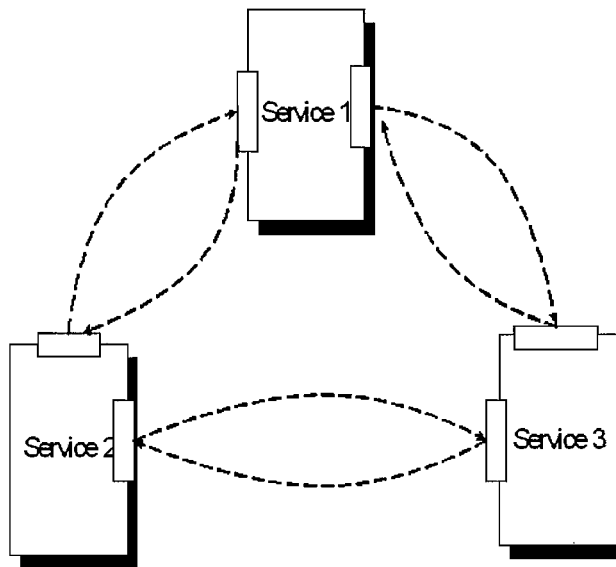


Figure 2-5 A WSCI Choreography Service Composition

WSCI also supports both basic and structured activities. An `<action>` is used to define a basic request or response activity. Each activity specifies the WSDL operation involved and the role being played by this participant. Structured activities like sequential and parallel processing are also given in WSCI. In addition, an `<all>` structured activity is used to indicate the specific actions have to be performed, but not in any particular order, and `<foreach>` structured

activity is used to specify repeatedly execution of inside activities based on the evaluation of a condition or an expression.

2.3 Workflow Technologies

In the Oxford English Dictionary, Work Flow is defined as “in an office or industrial organisation, the sequence of processes through which a piece of work passes from initiation to completion”. In computer science, people develop the workflow technology to improve the way an organisation works by making it better, faster and cheaper [12].

About 20 years ago, computer scientists started to develop software tools to not only do the work, but also assign, deliver work, and even track the progress of the work. A workflow is described in computer terms as “The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant* to another for action, according to a set of procedural rules. *participant = resource (human or machine)” [13].

With the fast development of the Internet technology, the requirement for automation is not limited inside an organisation. A modern enterprise application normally involves several organisations. Participants are situated at distributed network places. The environments are heterogeneous and dynamic.

A workflow system allows users to define and manage a sequence of work activities; invoke human and IT resources; and execute the activities following specified logic and steps.

An abstract workflow management model is summarised by three levels [13].

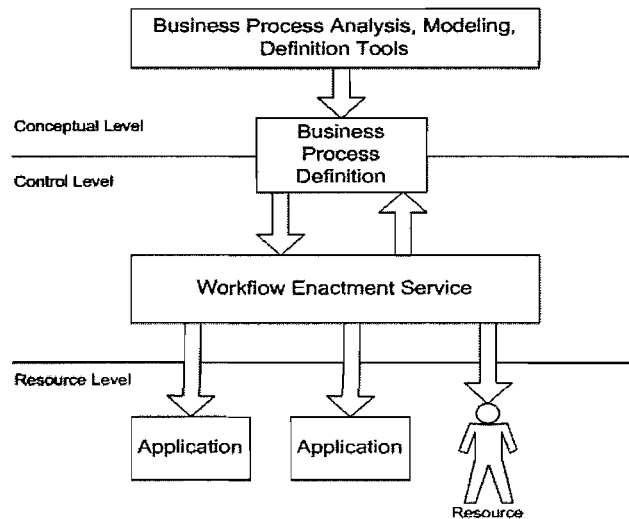


Figure 2-6 An Abstract Workflow Management Model

The conceptual level of the abstract model concerns notations. A real world business is captured by a business process which is translated into formulas using analysis, modelling and definition tools. A business process may contain several sub-processes. A traditional business process comprises activities, components, operation rules and progressive steps. Formal notations have been widely used not just by researchers but also by industry to achieve various goals.

- **System Analysis and Design**

A system normally includes hundreds, thousands or more lines of code. Before we start coding, we have to make sure our design is correct. A formal notation, a specification language, can help us. A system could be described by a model using a formal notation in less than a hundred lines. Reasoning in the model is much easier than implementing the actual system. It helps us understand the consequences of the system and examine whether our design captures all the requirements. Moreover, some unexpected inconsistencies could be uncovered early, for example conflicting or infeasible requirements, confusion over scope or domains.

- **Verification**

Software development and maintenance are expensive. Getting rid of bugs at design phase instead of testing phase will tremendously reduce the cost. Model checking is a

technique used to algorithmically check whether a program satisfies its requirement. A model checking tool normally takes a model (of the program) and a formal specification (of a correctness property) and returns a positive result if the model satisfies the specification, or a negative result together with a counter example, if the model does not satisfy the specification.

- **Code Generation**

Code generation is to produce programs in some automatic manner, reducing need for human programmers to write code manually. Model-Driven Architecture introduces the idea that “the system functionality is first defined as a platform-independent model using an appropriate specification language and then translated to one or more platform-specific models for the actual implementation.” [14].

The control level of the abstract model is the centre of the workflow system. The enactment service (see Figure 2-6) is responsible for process creation and execution. It also handles the interactions with client applications and resources. Real enactment services may also provide interface for the process management or monitoring applications.

The resource level of the abstract model concerns client applications and human operations. Applications are invoked by the workflow enactment service to perform automated activities. Client applications are unconnected. Controls are transferred between applications by the enactment service.

2.3.1 Activity-Based Business Process Modelling

Business Process Modelling is an essential part of the workflow technology. By capturing the business process in a workflow specification, then we can later use the specification to analyse, verify and even generate the system.

Traditional process modelling places emphasis on the organisation structure [15]. A business process is structured around roles and relationships. Most business processes are described as collections of structural activities that have been related to roles.

An activity-based business process consists of many activities which are logically related in terms of their contribution to the overall realisation of the business process [15].

An activity is a piece of work that forms one logical step within a process. It is defined by the following components [13].

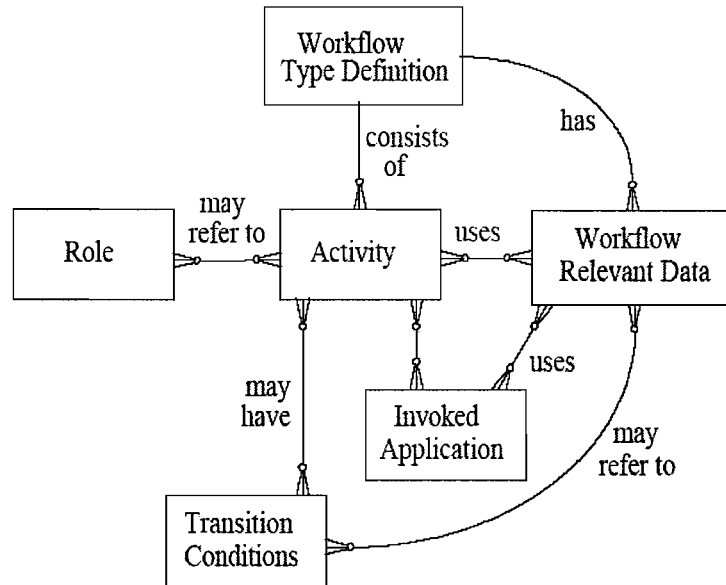


Figure 2-7 Activity – based Workflow

Basic Process Definition Meta-model

- Workflow Type defines the identity of the process; process start and termination conditions;
- Activity defines activity type; pre- and post activity conditions; other constraints;
- Transition Conditions define flow or execution conditions;
- Workflow relevant data defines data name, path and types;
- Role defines name and organisational entity;
- Invoked Application defines generic type or name; Execution parameters; location or access path;

2.3.2 Message-Based Business Process Modelling

Activity-based business process modelling has already been commonly accepted, however, a new appeal for a message-based process modelling is receiving increasingly more attention for the following reasons.

Message-based systems require new modelling approach

Message-based systems enable independent distributed applications or application components to interact through messages. These components “...whether on the same host, the same network, or loosely connected through the Internet, use messaging to pass data and to coordinate their respective functions” [7].

In a message-based system, components are loosely connected. Components deliver asynchronously. A message sent by a component should not depend upon the readiness of the receiving components. And the message will eventually be received by the components when it is ready. Moreover, components are able to exchange messages simultaneously. An asynchronous messaging pattern adds great flexibility to the interplay of components. It adds robustness because of the failure of one component does not translate into the failure of the whole system [7]. But it also gives up some of the system control. One of the most significant message-based systems is intending to use the web service technology. We will review the details of the web service architecture in the next section.

Given this, structured activities can hardly capture the behaviour of asynchronous communications.

Organisations and roles become less important in a business process

Enterprise applications are loosely coupled. Organisations implement functional components using incompatible technologies. Interactions between functional components rely on network protocols.

Unlike a traditional production line application, a business process is defined by examining the formal structures, goals, and activities of an organisation; a modern enterprise application manages business processes cross organisations. Organisations are multi-functional and provide a large amount of functionalities in order to serve different business requirements. As

business goals changed dynamically, the organisation structure becomes obscure. Roles of activities are overlapped, sometimes chaotic.

A containing model is required

A business process consists of set of tasks and assembly of supporting content. Activity-based business process has been focused on the flow of work and not on the definition and content of the work item itself [12].

When the work item is fairly simple and the order of tasks is fixed, a work item could be carried from one activity to another by simply giving definition of pre- and post- conditions of the activity. When the order of the tasks is not fixed, and especially when the order relies on the content of the work item, defining the pre- and post- conditions of an activity is clearly not enough to specify a process. In this case, a containing model is required to represent the content of the work item and maintain content consistency.

Thus we propose a new message-based business process modelling approach which extends the traditional process modelling with the following features:

- The capability to capture the message-based system feature, asynchronous messaging.
- A means to support dynamic changes in business processes.
- A way to present rich content model and maintain data consistency.

2.4 Formal Models and Related Works

In this section, we review some traditional and established formal models, and examine recent web service related works using these models.

2.4.1 UML Models

The Unified Modelling Language uses visualised diagrams to represent a system, including system structure and system design. Three types of diagrams are used in UML system models:

case diagrams exploring the system functionalities, class diagrams defining the system object structure, sequence and activity diagrams describing system internal behaviours.

UML has been widely accepted by both industry and academia. The advantage of UML models over other formal models is that they could be easily used by non IT experts to design systems. With the support of a large number of available tools, a system can be generated from case, class, sequence and activity diagrams. However, UML models do not provide the capability for formal analysis. Designers can not reason about a system design directly using those diagrams.

A number of works using UML models to design web services composition [48] [49] have been reported. Specifically, [48] describes a UML profile which uses existing UML tools to model BPEL4WS business processes. The work creates a mapping from Object-Oriented UML models to BPEL4WS processes and web services. It maps an object in a class diagram to a business process or a web service, and maps object attributes to business process states which are also known as variables. A message is represented as an arrow which is also known as a control link from a sending activity of a service to a receiving activity of another service. The work illustrates its technique with a one-way implementation which generates an executable system from UML diagrams. However, UML models are based on object oriented concepts, therefore the activity models cannot clearly distinguish the synchronous and asynchronous communication patterns in a BPEL4WS process. Therefore, some service oriented behaviours will be lost during the system reasoning based on those UML models.

2.4.2 Process Algebra Approach

Process algebra uses a mathematical approach to represent concurrent systems of interacting processes [50]. It provides algebraic languages to specify processes, and calculi to verify the processes. There are some popular process algebra languages, including CCS (Calculus of Communicating System), CSP (Communicating Sequential Processes), and LOTOS (Language of Temporal Ordering Specifications). A process algebra model could be model checked by temporal logics, and analysed by bisimulation.

The process algebra approach is used in a number of previous works to model web service oriented compositions. [51] introduces a formalisation of WSCI specification using CCS. It describes a web service interaction in an algebra term composed by channel (WSDL

messages), input/output actions (send/receive activities), and operators (parallel composition, choice constructs). The work complements the WSCI specification with reasoning mechanisms to check the compatibility between two web services. Two web services are regarded as compatible if the messages exchanged between them are interpreted properly. CCS basically uses a synchronous and symmetric communication model, in which two partners perform complementary actions together. To reason about interactions over a large amount of web services, an extension to the model is required.

LOTOS is another process algebra specification language for concurrent distributed systems. It is a combination of two specification models: a static algebraic model, ACT ONE, and a dynamic process model (similar to CSP and CCS). [52] describes a two way mapping between LOTOS and BPEL4WS specifications, so that a BPEL4WS model could be formally verified and model checked. However, the work only introduces a static services composition model where web service interactions are established before the conversation starts. It does not include the correlation set which is an important feature in BPEL4WS.

2.4.3 Petri Net Model

The Petri Net model uses bipartite graph to represent discrete distributed systems. A Petri Net is composed of places, transitions and arcs that connect places with transitions. An execution of a transition is simulated by removing a token from an input place and adding a token to the corresponding output place. Because a Petri Net allows a place to hold more than one token, it can model concurrent systems quite well.

Many works have been done using Petri Net based algebra to model control flows in web service compositions. [53] models a service composition by assigning web service operations to Petri Net transitions, and web services state (Not Instantiated, Ready, Running, Suspended, or Completed), to places. A web service represented in a Petri-Net has one input place and one output place. Algebra operators are used to model the web service composition, such as sequence or parallel execution. Applying a web service composition in a Petri Net, one can reason about system behaviours such as absence of deadlock and livelock. [54] uses a similar approach as [53], but introduces a Service/Resource Net (SRN) in order to model complex service compositions. The model extends the Petri Net model with three new elements, i.e., resource, taxonomy, time and conditions. Different from [53] and [54], [55] uses a combination of Petri Net and O-O concepts, G-Nets. G-Nets enrich Petri Nets with the

capability of modelling different communication patterns including synchronous and asynchronous, with well-defined interfaces.

Petri Net based modelling approach has advantages in modelling and reasoning about control flow in a service composition. However all the above works do not support automatic service compositions and the composition scalability using Petri Net is low [56]. Moreover, Petri Net based models do not model the content of the messages.

2.4.4 SPIN

SPIN is an automata-based formal model checking tool. The input language of SPIN is Promela. Promela is a process meta language. It models asynchronous distributed systems as non-deterministic automata. SPIN has been widely used for more than 15 years. Efforts have been made recently using SPIN formally to verify web service compositions, for example, WSFL [57] and BPEL4WS [46] [58] [59].

[57] formulates a set of translation templates that translate WSFL primitives into Promela. It uses a simple example to demonstrate that faulty behaviours can be detected with the SPIN model checker. A more comprehensive work [46] [58] [59] introduces a framework that translates the BPEL4WS specification into an intermediate language, followed by the translation of the intermediate language to Promela.

The SPIN based approach models web service conversation behaviours by mapping messages to actions, and web service internal states (variables) to the states that messages transit between. However, SPIN is a finite state verification tool, it can only partially verify the system by fixing the size of the input queues in the translation.

2.4.5 Summary

This section gave an overview of web service composition related works. Models like BPEL4WS, WSCI and UML provide rich descriptions to specify the service composition behaviours, but fail to provide the capabilities of formal reasoning and model checking. On the other hand, formal models using Process Algebra, Petri Net and Spin complement the industry specifications with formal reasoning and model checking, but are not expressive

enough to describe all the service composition behaviours, such as contents of messages or asynchronous communications.

In this thesis work, we create a new formal model which is concise and, at the same time, expressive enough to describe the service composition behaviours. A novel operational semantics will also be presented to enrich the model with the capability of formal reasoning.

2.5 Web Service Dynamism

To reduce software development cost, computer scientists decompose a large software or system into small components because they are easier to maintain. Meanwhile, these components can be reused to deliver new software functions. Component-based design makes software extensible and easily contracted [60]. It enables a system to be dynamically created from components based on the constantly changing requirements and system environment.

A web service is a more sophisticated form of component which provides a standardised declarative function description and a universally discovery directory, and is accessed by a ubiquitous protocol and data format. These features make a service-oriented system more likely to be dynamic. A web service is a component but autonomous [16]. Although a web service publishes its operations as universally accessible, it will only process an operation request, a service invocation, on its own demand. These loosely-coupled web service features have not been fully considered in traditional component based dynamic systems.

2.5.1 Web Service Dynamic Behaviours

A service-oriented system is composed of independent web services using either orchestrated or choreographic mechanisms [33]. Before we discuss the dynamic requirements and review related works on dynamic web services, we need to understand the dynamic behaviours in a service-oriented system.

In a dynamic service composition, a service user can select service providers from a number of service providers based on functional, non-functional rules, or high level business goals [61]. This describes the behaviour of service assembling, which is characterised as automatic service composition [62].

In contrast, [39] gives a dynamic example where a service is dynamically replaced by another service to improve the system performance during the execution of a user query. This is described as re-configuration [62].

Although the above two dynamic service composition descriptions are given based on different system perspective, they all happen at the system runtime. To specify these dynamic behaviours in formal model, both dynamic description and dynamic execution have to be considered.

2.5.2 Related Dynamic Web Service Works

A web service is essentially a component. The basic design requirements for a dynamic component-based system [63] are still applicable. New requirements are also needed to accommodate the autonomous and loosely-coupled web service features. A web service processes on its own, it does not have to be aware of the system change. Web services must have a common understanding of interaction context, so that an interaction could be continued after a system change.

A number of works have been reported to formalise the web service dynamism. In [64] [61], formal models and fuzzy algorithms are introduced to dynamically compose web services from a Quality-of-Service point of view. By giving QoS parameters, for example performance parameters, the dynamic selection of services from a large number of candidate services can be performed by the fuzzy algorithms. Similarly in [65], web services are dynamically composed in response to a user query. A dynamic reconfigurable architecture is presented in [62], which indicates that four aspects of a formal model, i.e., syntax, semantic, QoS and contextual, need to be considered to achieve the service composition automation.

In this thesis, we will propose a method to support web service dynamism from the business interaction point of view. There is a problem not addressed by most previous works: how the user's query is executed consistently when those web services are dynamically re-configured or re-composed at runtime. An architecture is introduced in Chapter 3 to coordinate business interaction over dynamically re-configurable web services. An operational semantics is also given in Chapter 6 to enable the runtime re-configuration.

2.6 Our Services Composition Approach

Web services have been posited as the key technology to implement future enterprise applications. In this section, we analyse the requirements to model service-oriented applications. These requirements address both the language for describing the composition and the supporting infrastructure to run it.

2.6.1 Web Service Composition Requirement

2.6.1.1 XML-based Interfaces

The goal of web service technology is to achieve universal interoperability between applications by using standards. A web service defines and publishes the interfaces in XML format document, WSDL, and is invoked through a XML-based messaging protocol, SOAP. XML is a platform- and language-independent language. The advantage of using XML-based interface and communication is obvious. A service consumer can invoke a service following the shared understanding but does not care how the service is implemented. This character fits into the current heterogeneous web environments.

Capturing the XML tree data structure is the basic requirement for modelling service composition. A XML convertible design notation can be easily compiled into a real implementation. Moreover, an application can be comprehensively verified using an XML convertible model checking tool.

2.6.1.2 Asynchronous Communication Pattern

In WSDL, a web service supports four types of operations, one-way, request-response, solicit-response and notification. These four types of operations allow a service to communicate with other services asynchronously. Asynchronous communication provides greater flexibility and loose coupling between web services. A service is not blocked waiting for a return of the message and can continue doing work immediately after the call. If a return result is required the caller component can be called back later.

A system that uses asynchronous communication can be more easily distributed over a network than if synchronous communication were used. The asynchronous communication

pattern is believed to be the vital to today's enterprise applications environment. To support an asynchronous web service communication pattern, a mechanism is required to correlate and coordinate exchanged messages between services.

2.6.1.3 Dynamic Configuration

Dynamic configuration is a very important factor to evaluate a system composed by various components. Dynamic configuration refers to the capability of plug-and-play and hot-swapping. When a new component is added to the system or a current component is required to be updated, a system should not have to be stopped or restarted. Components therefore need to be hot-swapped, without disrupting the interactions in which the retiring component is involved and allowing the new component to continue with that sequence while, presumably, providing some improved behaviour [5]. Web service composition must be dynamic and flexible to meet the dynamically changed business needs.

2.6.1.4 Interaction Integrity

In a service oriented architecture, an application is a service composition that combines services following a certain composition pattern to achieve a business goal [22]. This architecture provides great re-usability in the sense that a web service could be used by multiple applications. At the same time, it adds the considerable complexity to the definition and maintenance of an interaction.

To maintain the interaction integrity, the architecture is required to ensure the reliability of the messages and to check the availability of the resources (web services). But we are more interested in the abstract level interaction integrity and assume that all the messages are reliable and secure.

We have stated that web services are essentially stateless in section 2.2.1. Therefore, a mechanism to maintain and coordinate the interaction state over stateless web services is required to design and implement service-oriented systems. We have reviewed that there are different approaches to model the service composition. Various mechanisms are applied to maintain interaction state in those approaches. Different mechanisms fit into different business requirements.

We are going to give three basic mechanisms to handle interaction state over service-oriented applications, analyse the advantages and disadvantages of each solution, and explain the motivation of our approach.

Interaction State in Messages

The first solution is to put the state into the interactions, messages. The idea of this mechanism is putting all the information in the messages, including the required services, the workflow, and the state of the interaction. When a service receives a message, it abstracts the request and processes it. After the process is finished, the service puts the result into the original message, and sends it to other services based on the workflow specified in the message.

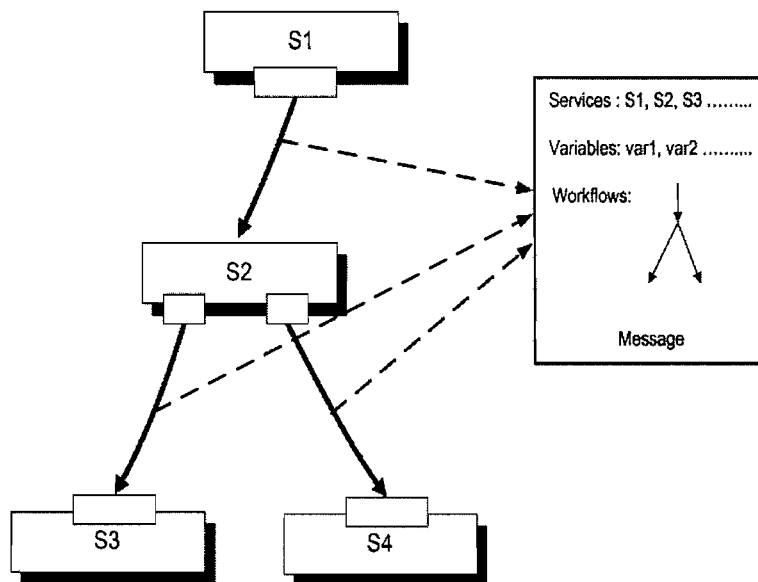


Figure 2-8 Interaction State in Messages

This solution is simple and easy to implement: no additional work is required to handle the interaction state. Using sessions and cookies, a rich data structure could be passed around by messages. It is sufficient enough to implement applications composed by sequences of activities. But because all the interaction state is stored in the message, services are unaware of any state, and therefore it is very difficult to execute parallel activities, for example synchronising two parallel threads of execution. Moreover, carrying interaction state and

workflow may raise some security issues. A security protocol is required for some applications to protect data and control authorisation.

Interaction State in an Object

The second solution is to use a stateful component, a business process, handling the workflow and the interaction state. A business process is essentially a web service but with state. A business process is deployed in a web container. For each business process request, the web container will create a new object. And any message related to the request, the container will pass it to the object to process. This architecture is similar to Figure 2-4

This is a popular solution to implement service oriented applications. Lots of specifications, such as BPEL4WS, use the same idea. Because a business process is a live object at the web container, it ideally could handle sequential, parallel and more complicated composition patterns. To achieve all these, a powerful web container handling concurrent business process, correlating service requests, supporting different composition patterns, is required and expected.

Modern enterprise applications are more complex: interactions run for long periods (weeks, months); a business process is required to be dynamically configured while some interactions are still active. One crucial problem of this idea is that the lifetime of a thread is limited, depending on the web container. Although some middlewares provide much stronger technique such as session beans that survive longer than a thread, they still cannot recover from a server crash. Therefore this solution hardly supports long-running persistent interactions and barely supports dynamic behaviours.

Interaction State in a Persistent Component

The third solution is to separate the state into an independent component, such as a database, and use stateless web services to handle the composition pattern. Interaction states are stored in the persistent component, a database. A context is given to the messages, so that when a web service receives a request, it can access the database and update the interaction state accordingly.

By storing interaction state into independent persistent components, a long-running interaction can be executed consistently. Because all the services are stateless, a component could be

replaced even while an interaction is still active. This solution is flexible and stable enough to support long-running and complex interactions. It releases the strong dependency on the web container. However it requires an elegant framework to coordinate the interaction state, propagate the context, and control database accesses.

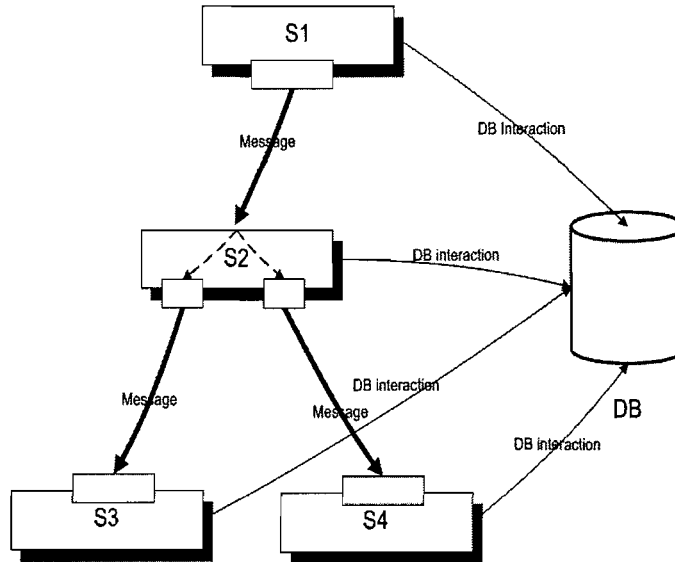


Figure 2-9 Interaction State in Database

As we have discussed, different solutions fit into different business requirements. Sometimes, a combination of two or three is used to provide stronger support for service compositions.

2.6.2 Our Service Composition Approach

A service-oriented system is a composition of independent web services to achieve certain business goals. The goals are fulfilled by service-to-service interactions. We aim to capture the behaviour of service-to-service interactions and provide a specification language to describe the service composition. Our language is general and can be applied to various business environments, in order that we can use it to support workflow automation and validation.

First of all, a business-to-business interaction is not just a transaction. Web service transactions are a subset of web service interactions [23]. A transaction is composed of a group of logical operations that must all succeed or fail as a group. Currently most of service composition standards and specifications such as BEPL4WS and WSCI only provide

mechanisms to support long-running transactions by providing two-phase commit protocols and compensation activities [25]. However, an interaction may include multiple transactions and takes much longer than a transaction. For example, in an online banking application, customers pay an estimated amount of money for their mobile in advance each month. When the bills arrive a month later (maybe longer), the difference has to be paid to the customer's account or mobile company's bank account. This interaction is completed by two payment transactions. As long-running interactions are the basis of modern enterprise applications, they should also be considered as part of the service composition.

We believe that web service compositions should be more dynamic. Storing interaction states in a pre-defined instance at the web containers decided that as part of an interaction the web service cannot be replaced in the middle of the interaction. Real businesses are challenging. The application is required to be recomposable in order to meet the dynamically changed business goals.

Context management is a more fundamental requirement than transactions in some business environments [26]. A context allows web services to share information such as message correlation and security token and so on. We use the context mechanism to model service-to-service interactions by giving each interaction a context. Our model allows the interactions and contexts to be structured hierarchically. Thus an interaction could be managed not only by a certain coordination service, but also by distributed services.

We model the service composition by describing the message exchanged between web services. Each service specifies its contributions to the interaction by updating the interaction state or coordinating the context. The model is executable as it specifies the complete state of an interaction and the sequence of execution based on the flow of the state.

Our solution of using context and coordination to model the service composition is flexible and extensible. It could be used not only to capture the service-to-service interactions, but also to model business transactions in specific domains, or to reason the security sensitive applications. The framework we are going to introduce can be easily either implemented as a new service composition platform or integrated into available service composition platforms as an additional feature.

In the next chapter, we will introduce our formal notation to specify service composition using context propagation and coordination mechanisms, in order to support long-running interactions and dynamic service configurations.

Chapter 3

Document Flow Model

In the previous chapter, we investigated service-oriented system environments, and analysed the asynchronous communications and dynamic configuration requirements in service-oriented systems. In this chapter we introduce a formal design notation to model such behaviours in order to facilitate system automation, and validation. This chapter is structured as follows: Section 3.1 introduces the concept of our Document Flow Model. Section 3.2 gives the formal syntax and informal semantics of the DFM notation. Section 3.3 uses a travel agent as an initial example to illustrate the use of the DFM.

3.1 What is DFM?

Our Document Flow Model describes a system as a set of messages that can be sent from a service, and the consequences for other services of receiving them. Because messages are basically XML documents, we call the modelling notation Document Flow Model (DFM). Our DFM has the following features.

3.1.1 Supporting XML Data Structure

A service-oriented system is composed of independent web services interacting with each other using XML-based web standards and protocols. DFM is a XML convertible notation. The document record data structure in DFM is created based on our experience with XML and its associated technologies [27, 28]. A document record describes a hierarchical (tree) data structure in a concise notation. Therefore we use it to model systems which are eventually realised using XML encoded documents.

Here is a document record with 3 elements.

[to: airline, query: [from:user,query:itinerary,context: userID], function: bookAFlight]

This document record describes a message sent from a Service Consumer, `user`, to a Service Provider, `airline`, to request a flight booking. The service consumer puts the information of its identity, `user`, the travel itinerary, `itinerary`, and a request number, `userId` into a nested document record, a `query`. We can see that, this document record structure could be modelled as a `<document/>` XML node in the following:

```
<document>
  <to> airline </to>
  <query>
    <from> user </from>
    <query> itinerary </query>
    <context> userId </context>
  </query>
  <function> bookAFlight </function>
</document>
```

3.1.2 Using Context Coordination Mechanism

DFM also provides support for long-running interactions and dynamic configuration behaviours. One aspect of our ability to simply unplug something in the middle of an interaction and plug in a substitute is whether or not the component has state. Replacing a stateful component with another is always going to be more difficult than replacing a stateless component with another [5]. This is one of the reasons that one of the principal design criteria for web services is that they should be stateless [29]. Our notation models a business interaction via stateful messages passed around stateless web services. A coordination framework, including a persistent component, a decentralised context generation and propagation mechanism, is given to maintain the integrity of the business interactions.

As a business interaction normally runs for a long period, an airline service necessarily gives a “booking id” so that a customer can process the payment or cancel booking consistently in the future. In DFM, a `generate id` action is provided to create a global unique symbol to identify a business interaction.

generate new bookingId

A persistent component, a `ContextStore`, is used to maintain the execution state. An application may have more than one `ContextStores`, meanwhile a `ContextStore` could be shared and accessed by one or more than one web services, depending on the business requirement. In a travel example, an airline service updates a flight booking process by storing certain data entry into the `ContextStore`:

store bookingId -> entry in ContextStore

The airline service arranges the user and flight information by the new generated identity `bookingId`, at the `ContextStore`, so that for example a **`ContextStore [bookingId]`** may contain the following entries.

[from:user,query:itinerary,context: userId],

[from:user, query: itinerary, result:flight]

Moreover, a context is given to each message to identify the business interaction. The airline service puts the new generated `bookingId` into flight confirmation, for instance.

**send [to:user, query: [from:airline, query:itinerary,result:flights, context:bookingId],
function:flightConfirmation]**

The user gets the flight booking information together with the original query and a booking number issued by the airline service which may be used to query or cancel the booking in the future.

Using this mechanism, a stateless web service simply reacts on the incoming messages and updates the state within an independent `ContextStore` when necessary. By coordinating all state updates with the persistent component, a business interaction can carry on even if the system configuration has changed.

3.1.3 Modelling Asynchronous Communication

A web service invokes another web service by messages. A synchronous service invocation can be illustrated as a telephone conversation which is the service caller is waiting for the reply after the request is sent off. An asynchronous service invocation can be illustrated as an

email conversation which is after a request message is sent off, the reply may not be available immediately, and the service requester could pick up the reply later.

The DFM notation intends to model systems composed by sets of independent web services and orchestrated by asynchronous messages. Since we are not interested in the detailed functionality and performance for each service, we model a web service invocation as a collection of outgoing messages sent in response to an incoming message. A reply message to a service requester whether synchronous or asynchronous is modelled as a service callback, a new service invocation, which is a sending message action.

The behaviour of a web service in response to a request is defined after an **OnMessage** followed by the request message pattern. When certain incoming message matches a pattern in an **OnMessage** of a service, corresponding actions are taken. For example,

```
OnMessage[ to:airline, query:[from:user,query:itinerary,context:userId], function:bookAFlight ]
.....
send [to:user,query:[from:airline, query:flights, context:bookingId], function:flightConfirmation]
```

when an airline service receives a request that matches the pattern of a bookAFlight message, it processes the request and then packs the flight booking confirmation into a message and sends it to the service requester.

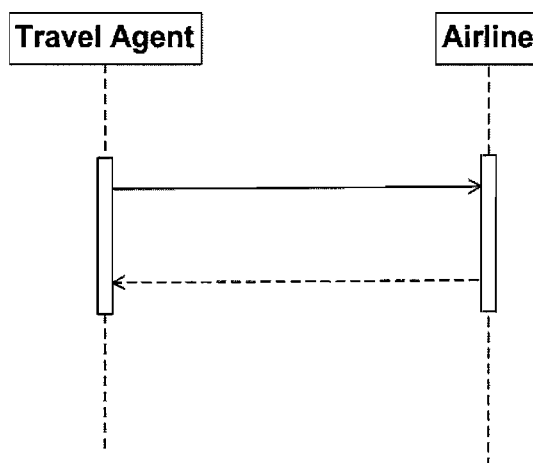


Figure 3-1 A Synchronous Communication

DFM defines two basic kinds of communication patterns in supporting asynchronous messaging, one-way communication, which amounts to a service receiving a message, and notification, which amounts to a service sending a message. The request-response and solicit-

response [3] synchronous conversations are modelled as a one-way communication plus a notification communication.

A travel agent service helps a user booking flights from various airlines. A synchronous communication is showed in Figure 3-1. The travel agent sends a flight booking request to an airline and waits for the response message, a solicit-response service call. The airline service on the other side receives a flight booking request from a travel agent and returns the message later to the travel agent, which is the same thread of the original travel agent that starts the request, a request-response service call.

In DFM, the synchronous communication is modelled as two asynchronous service calls, as the Travel Agent in Figure 3-2.

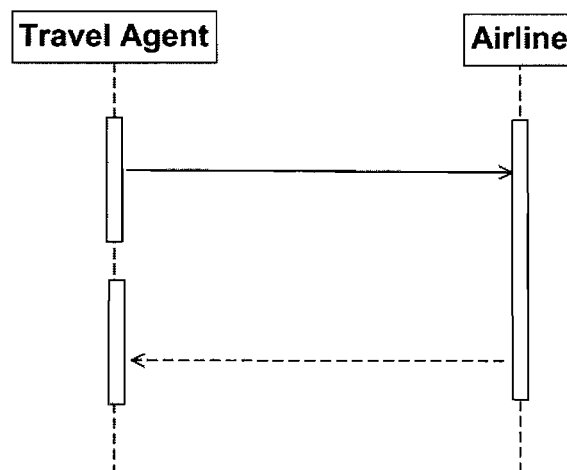


Figure 3-2 An Asynchronous Communication

The travel agent sends a flight booking request message, a notification service call, to an airline service after gets a request from a user, a one-way service call. The travel agent (thread) may close this thread or carry on other businesses without waiting for the reply message from the airline.

```

OnMessage[ to:agent, query: [from:user,query:itinerary,context: userId], function: bookAFlight ]
.....
send [to:airline,query:[from:agent, query:itinerary,context:agentId], function: bookAFlight]
  
```

The airline service replies a flightConfirmation message to the travel agent by creating a new service call on the travel agent, a new notification service call when bookAFlight request has been processed.

```

OnMessage[to:airline,query:[from:agent, query: itinerary,context:agentId], function: bookAFlight]
....
send [to:agent, query:[from:airline, query:itinerary, result:flights, context:airlineId],
                                             function:flightConfirmation]

```

When the travel agent service receives the flightConfirmation message, a new one-way service call, from an airline service, it arranges a new thread to handle the request as in Figure 3-2.

```

OnMessage [to:agent, query: [from: airline, query:itinerary, result:flights, context:airlineId],
                                             function:flightConfirmation]
.....
send [to:user,query:[from:agent, query:itinerary, result:flights,, context:agentId],
                                             function:flightConfirmation]

```

Each **OnMessage** definition describes actions a service takes after receiving a message. No nested incoming message is allowed inside the **OnMessage** body. Thus in DFM, a business interaction is modelled by several **OnMessage** definitions.

3.2 Formal Syntax

Before we introduce the notation, let's first give all meta-symbols a definition.

Symbols	Description
$a ::= b$	the element a is defined as b
$b\ c$	element b is followed by element c
$_$	empty element
$a\ \ b\ \ c$	element a or element b or element c
a,b	a list of elements separated by ','
$[]$	a document record
$x:y$	a pair element comprised of element x and element y
$\{\}$	a block of actions

Table 3-1 DFM Meta-Symbols

3.2.1 The Basic Structure

A DFM specification is built from message definitions, or `messagedefs`. A web service is described by a collection of `messagedefs` specifying the messages which the web service receives and operates on.

```
messagedefs ::= messagedef
              | messagedef messagedefs
```

In DFM, each `messagedef` defines the web service response to an incoming message: when the incoming message matches the message pattern in `messagedef`, the corresponding actions in `msgdefbody` are triggered.

```
messagedef ::= OnMessage message
              msgdefbody
```

3.2.2 The Message Definition Body

A message definition body, `msgdefbody`, defines the set of actions to be carried out when an incoming message matches a certain pattern. Possible actions to be specified in a message definition body include creating a series of identities, storing a document into a document store, and sending a message.

```
msgdefbody ::= idaction storebody sendbody
```

Thus, a `msgdefbody` may contain three pieces of information, `idaction`, `storebody` and `sendbody`, in this particular order; any of these pieces of information could be absent. The `idaction` describes some new identities used to identify interactions started as a result of a message being acted upon.

```
storebody ::= _ | storeaction storebody
sendbody ::= _ | sendaction sendbody
              | csendaction sendbody
```

The storebody describes the set of store actions to be carried out (in parallel), before the (possibly conditional) message sending actions described in sendbody are carried out (also in parallel).

3.2.3 Actions

A message definition may contain essentially four kinds of actions: idaction, storeaction, sendaction and csendaction, as mentioned earlier.

Id Action:

```
idaction ::= _ | generate new ids
ids      ::= id | id, ids
id       ::= string
```

When a service starts a new business interaction or a sub-interaction, it usually creates a new identity to identify that interaction. An idaction specifies the identities generated in this way. The newly generated identities are universally unique, that is, identities generated by the same / different services are different; this can, for instance, be ensured by embedding information such as service identity, message date, time and message content in each newly generated identity.

Store Action:

As we stated earlier, the ContextStore is a component used to maintain the state of a business interaction. The DFM only allows insert operations on the ContextStore. Other operations, such as update or delete, are captured into data entries which will be inserted into the ContextStore by the storeaction.

```
storeaction ::= store id -> entry in ContextStore
```

A storeaction describes the action of storing a piece of information, an entry, into the ContextStore, under a particular identity id.

Send Actions:

The DFM models a service invocation and a service call-back, a reply to a service invocation, using send actions.

```
sendaction ::= send message
```

A sendaction describes the action to simply send out a message.

```
csendaction ::= if condition then { sendactions }
```

```
sendactions ::= sendaction | sendaction sendactions
```

A csendaction specifies one or more sendactions to be performed only when a certain condition (involving the current state of the ContextStore) holds. When the condition evaluates to true, the corresponding sendactions, a list of sendaction, are taken.

3.2.4 Conditions

A condition is a ContextStore evaluation expression, possibly containing logical operators.

```
condition ::= ContextStore [id] contains entries  
            | condition and condition  
            | condition or condition  
            | not condition
```

A Condition allows a web service response to a message request by the current state of a business interaction as stored in the ContextStore. A simple condition is evaluated to *true* when the specified entries are found in the ContextStore under the identity, id, otherwise the condition is evaluated to *false*. Conditions containing logical operators are evaluated in the standard way.

3.2.5 Control Flow

A simple control flow, a collection of non nested *if... then...* statements, is available in the DFM notation. A condition is a Boolean expression and could be evaluated to *true* or *false*. A complex control flow, such as a nested *if... then.... else...* can be translated into this simple

format, as in Figure 3-3, where we assume that $c1$ and $c2$ are conditions while $a1$, $a2$ and $a3$ are actions.

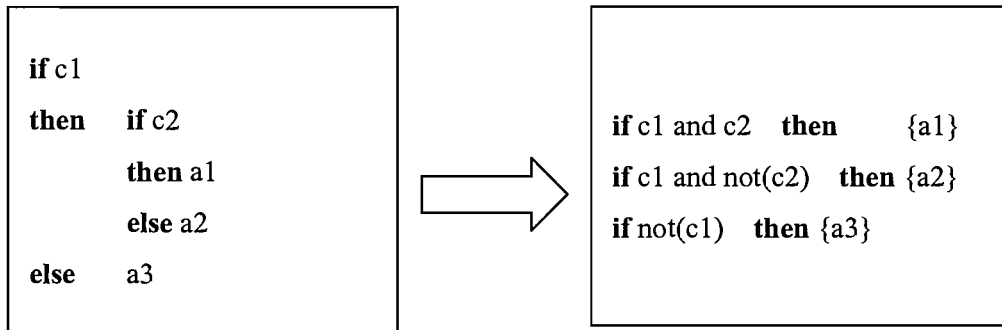


Figure 3-3 A DFM Control Flow Example

3.2.6 XML Data Structure

Web Services interact with each other by messages. The messages are essentially XML documents. To model the XML tree data structure, we introduce a new data structure, a document record.

A document record data structure allows us to specify an object with properties. A document record literal consists of a comma-separated list of colon-separated property name / value pairs, all enclosed within square brackets. In the document record, a property name is simply a string identifier, while a property value is an atom or a document record. A simpler form of the document record contains no property names, only property values, a comma-separated list of values.

In relation to XML, a document record is a XML element. We ignore XML attributes, important though they are in practise, because at the modelling level it is unnecessary to distinguish between nested attributes and nested elements. In a document record, a XML attribute is modelled by a property of that element.

A Message:

Document records with particular property names as **to:**, **query:**, and **function:**, are used to model the messages being passed between web services, as described in the following. The

property values `to` and `function` are simple strings which describe the message receiver and the requested operation.

```
message ::= [to:to, query:query, function:function]
```

```
to, function ::= string
```

Queries:

The property value `query` is a document record that refers to the message data, or message parameters.

```
query ::= element
```

```
    | [from:from,query:query,context:uid]
```

```
    | [from:from,query:query,result:query,context:uid]
```

```
element ::= string | [elements]
```

```
elements ::= element | element, elements
```

Similarly the property values `from` and `uid` are strings. A **from:** indicates who starts the request which is used when a reply is required to be sent back the initiator. A **context:** points out which interaction this query belongs to.

```
uid, from ::= string
```

Three types of queries are defined in DFM. The first one, `element`, is either a simple string or a simple document record with no property names, and the property value given by either a string or a list of `elements`. These strings are normally used to denote simple tasks or task executing results.

The second is a document record with **from:**, **query:** and **context:** properties. It includes the query initiator, query content and query identity. It is used, for example, when a web service wants to start a business interaction by passing a query to other web services.

The third type of query is a document record with **from:**, **query:**, **result:** and **context:** properties. For example, when a query has been completed, the results are put into a message together with the original query. Similar to a message definition, the **query:** property value is a further

document record, and so is the **result:** property value. When a business interaction is composed by several sub-interactions, this query is required to identify which sub-query has been completed.

3.2.7 ContextStore

The ContextStore is a critical component of the system. Different from a web service, a ContextStore is a persistent component storing the interaction state. The systems modelled using the DFM notation are concurrent: multiple process sessions are carried on at the same time. To maintain the system state, a unique identity is created and assigned to each business interaction. The interaction state is structured into document records, **entrys**, and stored under the interaction identity in the ContextStore.

$$\text{entrys} ::= \text{entry} \mid \text{entry}, \text{entrys}$$
$$\text{entry} ::= [\text{from:from}, \text{query:query}] \\ \mid [\text{from:from}, \text{query:query}, \text{result:result}]$$

A business interaction is represented at the ContextStore by a collection of **entrys**. A business interaction may have sub interactions. The **entrys** are stored into when an interaction or a sub interaction has been started, and when the interaction or sub interaction has been submitted or completed.

3.2.8 Keywords

Here is a summary of all keywords in DFM.

Keywords	Description
OnMessage	An event which matches an incoming message to the message pattern specified in the messagedef.
to	The receiver of a message
from	The sender of a message
query	A property name indicates the content of a query
result	A property name indicates the result of a corresponding query
context	A property name indicates the unique identity of an business interaction
function	A property name indicates what function the message are calling
generate new	An action to create a new unique identities
ContextStore[..]	Data entries in ContextStore under index specified in []
store .. -> ... in ContextStore	An action to store an entry into the ContextStore
send	An action to send out a message
contains	A condition to verify whether the ContextStore includes particular entries.
if .. then ..	A simple control flow

Table 3-2 DFM Keywords

3.3 An Example

We provide here a simple example to illustrate the basic structures and some fundamental concepts of DFM. Web services are essentially stateless [16]. In the previous chapter, we analysed the three principal mechanisms for making web services stateless, namely putting the state into the interaction, using a stateful service to handling state and separating the state into

an independent persistent component. We use the last mechanism which is more powerful and flexible than the other two.

3.3.1 A Travel Agent System

The travel agent system offers trip reservation services to its customers via the Internet. A business interaction is very simple as showed in Figure 3-4: a user gives its travel plan to the travel agent and asks for a reservation for a flight and a hotel. The travel agent then books a flight ticket from an airline, and reserves a hotel room from a hotel service or a hotel agent based on the user's travel plan. It then sends a draft itinerary to the user. The travel agent system is concurrent that multiple user queries are handled simultaneously. The following diagram only illustrates the interaction workflow using a single user interaction.

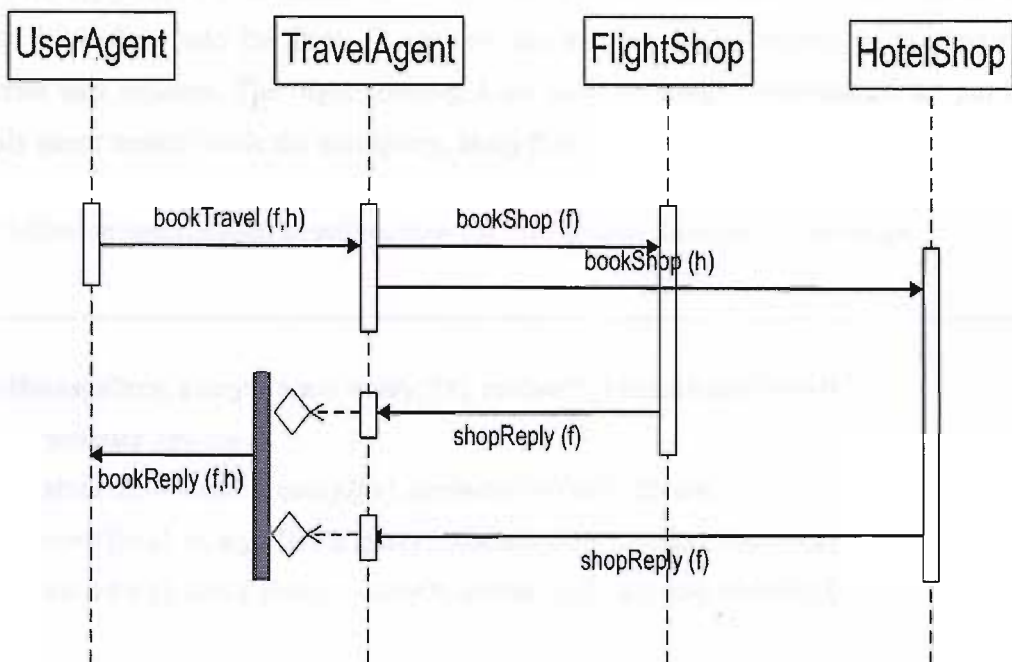


Figure 3-4 A Simple Travel Agent Sequence Diagram

We create 4 types of services in this application:

- A UserAgent, *u*, handles user forms and initiates the booking request. The UserAgent works through a TravelAgent to get the travel reservations.

- A TravelAgent a, passes the booking request to the individual shop services. It extracts the initial request into two new requests and send them to two shops, a FlightShop and a HotelShop, simultaneously.
- Two shop services, a FlightShop, s1, and a HotelShop, s2, process their booking request and send the results back to the TravelAgent.

The UserAgent, u, initiates a message as following, and sends it the TravelAgent, a,

[to:a, query:[from:u, query:[f,h], context:c],function:bookTravel]

Three parts compose the query in this message. A TravelAgent offers the booking service for various services, so if the service expects a reply, it has to let the TravelAgent know where he sends the reply to. For this example, the UserAgent wants to get back the reply himself, so it packs the address into the **from:.** It also creates a new unique **context:.** c, to identify the different user requests. The flight booking, f, the hotel booking, h, information are put into a simply query nested inside the user query, **query:[f,h].**

The following specification describes how the TravelAgent handles this message.

```

OnMessage[to:a, query:[from:u, query:[f,h], context:c],function:bookTravel]
  generate new uid
  store uid -> [from:u, query:[f,h], context:c] in ContextStore
  send [to:s1, query:[from:a, query:f, context:uid], function: bookShop]
  send [to:s2, query:[from:a, query:h, context:uid], function: bookShop]

```

Figure 3-5 A Travel Agent Specification – I

The TravelAgent creates a unique identity, uid, after receives the user request. It will pack this identity into the messages as a context of the future queries. Then it stores the user's query into a persistent component, ContextStore. It did not use the original context, c, as the identity of this query, because the TravelAgent splits the query into two parts and starts a new flow of interactions. Therefore, it gives the new flow of interactions a new identity, and tells the

message receivers that the query in the message is part of the interaction[uid] instead of interaction[c].

We assume here, the TravelAgent does not want the shops and users know each, for business sake. It extracts the user query, and creates two new queries with the new identity for FlightShop, s1, and HotelShop, s2. The two messages in this messagedef are parallel messages, could be sent out in any order.

Since the FlightShop and HotelShop have similar behaviours, we only give one specification in Figure 3-6. When a shop service receives a booking message, it computes the request, and puts the result into the reply messages. We see that there are 4 elements in the query of the reply message this time.

```
OnMessage [to:s, query:[from:a, query:f, context:uid], function:bookShop]  
  send [to:a, query:[from:s, query:f, result:r, context:uid]], function:shopReply]
```

Figure 3-6 A Shop Specification

The shop tells the Travel Agent that it is part of this interaction by putting its address in. And it forwards the original nested query and context, together with computation result to it.

To simplify the example, the shop services use the original context as their interaction identity in this example. In some business environments, the shop services may create new contexts, a booking confirmation number for example, and pack it into the result, it may also store entries in their own ContextStores. We will see it later in other examples.

In the Figure 3 4, the TravelAgent receives 3 messages for each interaction. In the Figure 3 7, we give the specification for the agent receiving a shopReply from the HotelShop or the FlightShop.

When the TravelAgent receives a shopReply or a bookTravel message, it reads the context of the query and saves the other parts of the query into the ContextStore under that context. Because the system we specified is asynchronous, the 2 reply messages could arrive in any order. In the sequence diagram Figure 3-4, the 2 messages are handled by 2 different threads.

Each thread will check the logic of the interaction state flow, if the interaction has been fulfilled a reply message will be sent, otherwise no action will be taken.

```

OnMessage [to:a, query:[from:s, query:f, result:r, context:uid], function:shopReply]
  store uid -> [from:s, query:f, result:r] in ContextStore
  if ContextStore[uid] contains
    [from:a, query:[from:u,query:[f,h], context:c]],
    [from:s1, query:f, result:r1],
    [from:s2, query:h, result:r2]
  then { send [ to:u,
    query:[from:a,query:[from:u,query:[f,h],context:c],result:[r1,r2],context:uid],
    function:bookReply] }
  
```

Figure 3-7 A Travel Agent Specification – II

Thus the 3 incoming messages to the TravelAgent correspondingly associated to one travel booking interaction.

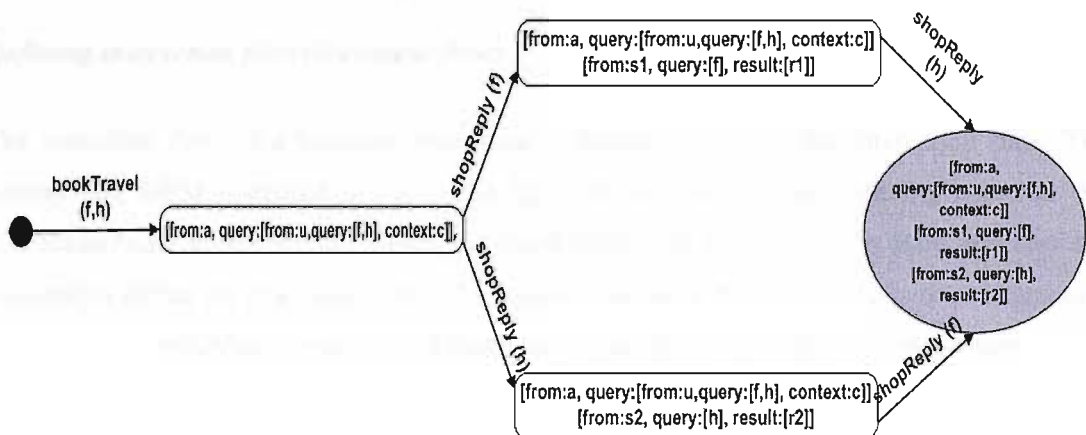


Figure 3-8 A Document Flow Chart

The service interaction state is represented by the entries at the ContextStore. A state flow of a particular interaction starts from an empty record. The travel agent updates the interaction state upon the message contents by storing in certain entries into the store and checks the state of the interaction each time a message arrives.

3.3.2 Summary

The simple travel agent example demonstrates how to use the DFM notation to specify a service-oriented system. The basic method could be summarised as follows:

Using context

Context is an essential mechanism in the DFM. The purpose of using contexts is collaboration. A context is a data shared by web services participating in the same business interaction. A context identifies a web service as part of a composite application and allows each web service to update or observe the execution of the overall business interactions. Extensions to basic context operations provide the ability to support long-running units of work such as business process automations and workflows [23].

In DFM, a web service gives a context to each query in the message as a symbol of participation in a business interaction. A new unique context is required when a new business interaction is started. Some business interactions are maintained by one or more distributed web services. In such circumstances, the web service stores the interaction state into the persistent component, ContextStore, and indexes them by the context so that concurrent business interactions can be maintained.

Defining interaction flow (document flow)

The execution flow of a business interaction is decided based on the interaction state. The system that DFM specified is composed by stateless web services. Having captured the interaction state at the ContextStore, the conditions over the interaction state (entries) are required to define the execution flow. The simple controls and logic operators over interaction entries allow specifying complex workflows as will be illustrated in the next two chapters.

Structuring query

Structuring the query correctly is essential in the DFM specification. A query composed by four parts.

A **from:** in a query indicates the request initiator. By storing this information in a persistent component or keeping it in a message, a service does not have to maintain an active

connection or a session between the service requester and itself. The connection can be re-established after a long period of time. Thus the services are much freer to be replaced without losing any interactions and much more capable to perform long-running interactions.

A **context**: as we explained earlier is put in the query to identify which business interaction the current query belongs to.

A **result**: is an answer from a service regarding a query inside the query request.

Carrying the original query request in a reply is necessary when an interaction is composed by several sub-interactions (sub-queries). The request query is required to identify which sub query has been completed.

A nested query structure allows not only to take query contents, but also to carry parent queries. It is used to coordinate the current and its parent interactions.

We will give more examples in the next two chapters to illustrate the using of DFM to model nested business interactions and to specify services coordination.

Chapter 4

Illustration

In Chapter 3, we gave the formal syntax and informal semantics of our DFM notation, and provided some simple examples to demonstrate how to use the notation. In this chapter, we use a typical grid application, a job submission system, to illustrate the dynamic configuration capabilities, and to discuss the use of notation to model hierarchical service interactions.

4.1 An Example of Dynamic Replacing a Service

When an application involves a large number of tasks, instead of buying a supercomputer, a more effective way is to deliver the subtasks to different computers, subsequently combine their results. Web service is one of the technologies to implement such systems.

4.1.1 Service Coordination

We have described earlier that our notation allows an interaction to be coordinated by one service or by distributed services. In the following example, we introduce a Coordination Service to maintain the state of an interaction over stateless web services.

We have also discussed that any web service may access the state-maintaining component, ContextStores. In a previous example, we specified that all services have access to the ContextStore. In the following example, we only let the Coordination Service access the ContextStore for the following reasons.

- Restricting the access to the ContextStore, largely releases the concurrent control of workloads on the persistent components, especially in the applications that involving huge computing tasks.

- By maintaining the state solely through the Coordination Service, the service can monitor the overall interaction state, so that any system failure can be captured and recovered timely.
- Replacing a component with access to the state component is much more complicated than replacing a component with no access to the state. Thus the use of the Coordination Service makes our system more amenable to dynamic reconfiguration.

4.1.2 A Job Submission System

A simple job submission system is similar to the simple travel agent. We create 4 types of services in this application:

- A User initiates the job submission. The User works through a FlowService to get the job executed.
- A FlowService, an orchestrating web service, makes use of JobServices, specifying the workflow and distributing jobs. A FlowService passes the job tasks to the Coordination Service, and extracts the initial job tasks into two new requests and send them to two JobServices simultaneously.
- Two JobServices, JS1 and JS2, executes jobs and sends the results to the Coordination Service.
- A Coordination Service updates job execution progress with the ContextStore and coordinates with all other services. If the whole job has been completed, the Coordination Service will send the job execution result to the User.

Here is the sequence diagram:

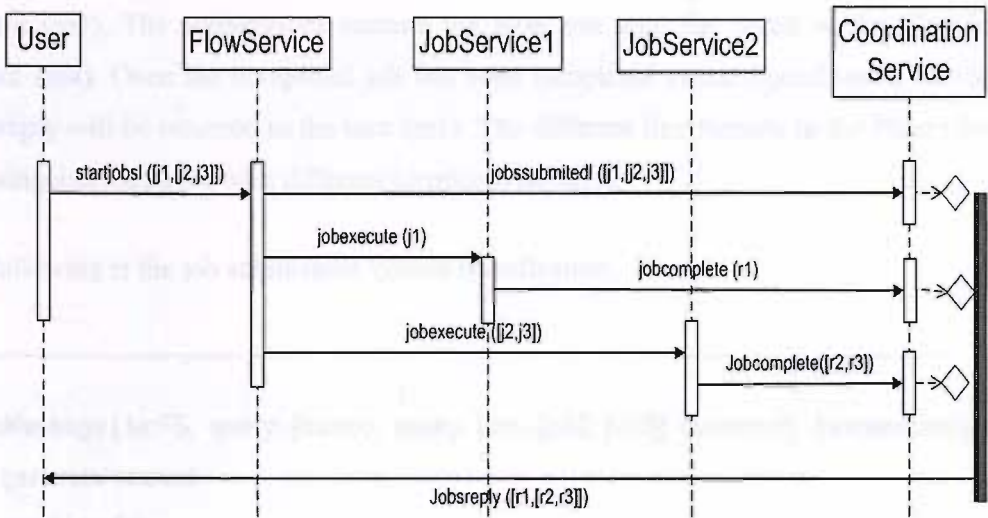


Figure 4-1 A Simple Job Submission Sequence Diagram

Unlike the previous example, that FlightShop and HotelShop process different types of queries, hotel query and flight query, the JobServices are peers that process any job queries. This decides that the system should be more dynamic.

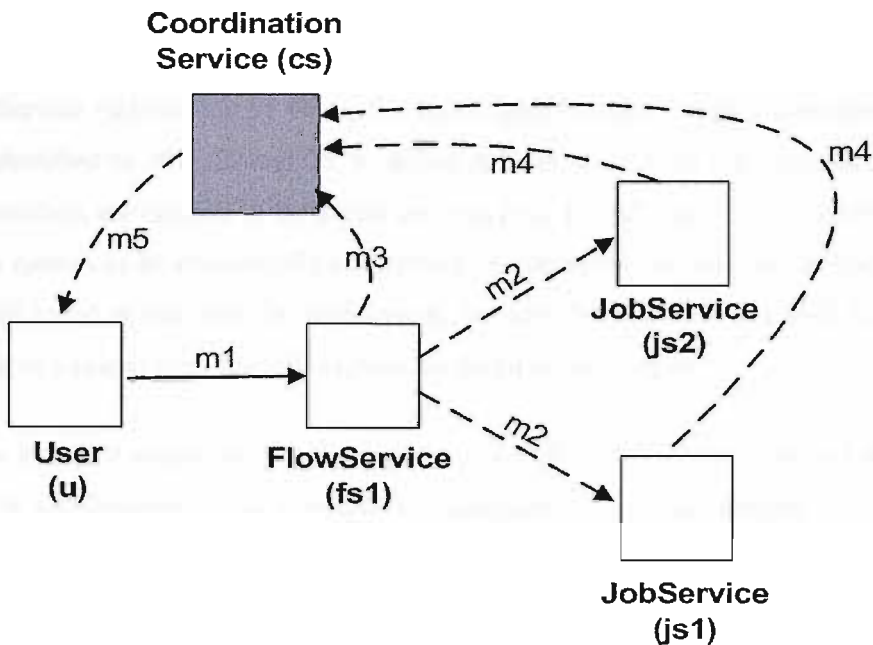


Figure 4-2 A Job Submission Example

In our example (Figure 4-2), when a FlowService, fs1, receives a message (m1) with a composed job, it passes them to JobServices (m2). The FlowService also sends the job submission state to the Coordination Service along with the job identity and the original

initiator (m3). The JobServices execute the jobs and send the result to the Coordination Service (m4). Once the composed job has been completed at the Coordination Service, the final reply will be returned to the user (m5). The different line formats in the Figure are used to distinguish messages with different identities, contexts.

The following is the job submission system specification.

```

OnMessage [ to:FS, query: [from:u, query: [job1,[job2, job3]], context:c], function:startjobs]
  generate new uid
  send [ to:CS,
        query: [from:FS, query:[from:u,query: [job1,[job2, job3]], context:c], context:uid],
        function:jobssubmitted]
  send [ to:JS1, query: [ from:FS, query:job1, context:uid], function:jobexecute]
  send [ to:JS2, query: [ from:FS, query:[job2,job3] context:uid], function:jobexecute]

```

Figure 4-3 A FlowService Specification

The FlowService (referred to by FS in the DFM specification) requires a number of other services, identified by JS1, JS2 and CS, to define its workflow. In the semantics of DFM, such service identifiers are mapped to the actual services (e.g. js1, js2 and cs from Figure 4-2), thus allowing a system to be dynamically-configured. In particular, the services corresponding to JS1 and JS2 could be not only the JobServices, but also the FlowServices with the extended capabilities of a JobService. We will explain the detail in the Chapter 6.

The user's query is composed by three parts: a simple query contains set of jobs; **from:** indicates the user initiates it; and a **context:** to distinguish different job queries.

When, the FlowService, fs1, gets the user's query, it creates a unique identity, uid, to identify a new interaction. In this case, the job results are handed out by the Coordination Service. Thus fs1 tells the cs, that a new interaction is started by forwarding the user's query. The fs1 only extracts the user's jobs and attaches them to two JobServices, js1 and js2, as fs1 believes that the JobServices will not interact with users. The FlowService sends out the messages concurrently and gets on with its business of servicing other interleaved queries and replies.

```

OnMessage [ to:JS, query: [ from:fs, query:job, context:uid], function:jobexecute]
  send [to:CS, query:[from:JS, query:job, result:result, context:uid], function:jobcomplete]

```

Figure 4-4 A JobService Specification

The JobServices execute jobs and forwards the results with the original query and context to the cs. The original query is required for the Coordination Service to indicate which part of the interaction is completed. The js2 gets a job query with 2 jobs as [job2, job3], it executes them and forwards the result, [result2, result3], to cs.

```

OnMessage[to:CS,query:[from:fs,query:[from:u,query:[j1,j2], context:c], context:uid],
  function:jobssubmitted]
  store uid->[ from:fs, query:[ from:u, query:[j1,j2], context:c]] in ContextStore
  if ContextStore[uid] contains [ from:fs, query:[ from:u, query:[j1,j2], context:c]]
    [ from:js1, query:j1,result:r1]
    [ from:js2, query:j2,result:r2]
  then { send [to:u,
    query:[from:CS, query:[ from:u, query:[j1,j2], context:c],result:[r1,r2],context:uid],
    function:jobsreply] }

```

```

OnMessage[to:CS, query:[from:js, query:job, result:result, context:uid], function:jobcomplete]
  store uid -> [ from:js, query:job, query:result, context:uid] in ContextStore
  if ContextStore[uid] contains [ from:fs, query:[ from:u, query:[j1,j2], context:c]]
    [ from:js1, query:j1,result:r1]
    [ from:js2, query:j2,result:r2]
  then { send[ to:u,
    query:[from:CS, query:[ from:u, query:[j1,j2], context:c],result:[r1,r2],context:uid],
    function:jobsreply] }

```

Figure 4-5 A Coordination Service Specification

Similar to the travel agent example, each time the Coordination Service receives a message, whether a `jobsubmitted` query or a `jobcomplete` query, it stores certain entries into `ContextStore` and then checks whether it has sufficient information to complete the job.

When the following information has been gathered at `ContextStore[uid]`,

```
[ from:fs, query:[from:u, query:[job1,[job2,job3]], context:c]]
```

```
[ from:js1, query:job1, result:result1]
```

```
[ from:js2, query:[job2,job3], result:[result2,result3]]
```

the Coordination Service replies to the user a message with the query of the following

```
[from:cs,  
 query:[from:u, query:[job1,[job2,job3]], context:c],  
 result:[result1,[result2,result3]]  
 context:uid ]
```

The message tells the user that the message is sent from the Coordination Service, `cs`; the original jobs sent from user with the context `c`; the nested job execution result; and this job execution record at the Coordination Service is `uid`.

4.1.3 A Re-Configured Job Submission System

The above example shows how asynchronous interactions are described in DFM. Our notation also intends to support long-running interactions and dynamic configurations.

A huge computing job may run for long periods (days, weeks). Storing the job execution state in a short-lived instance as BPEL4WS does is clearly not feasible. Moreover, components need to be hot-swapped, without disrupting the interactions in which the retiring component is involved and allowing the new component to continue with that sequence while, presumably, providing some improved behaviours [30].

Dynamic configuration is a very complicated issue [section 2.5], especially in a system composed of distributed web services. In our work, we are concerned about high level business interactions. We assume that all messages will be delivered reliably, correctly and completely. And we will discuss the operational semantic rules to explain how the system

handles the dynamic behaviours. In the following specifications we focus on the business interactions, analysing the integrity of the interaction and the workflows.

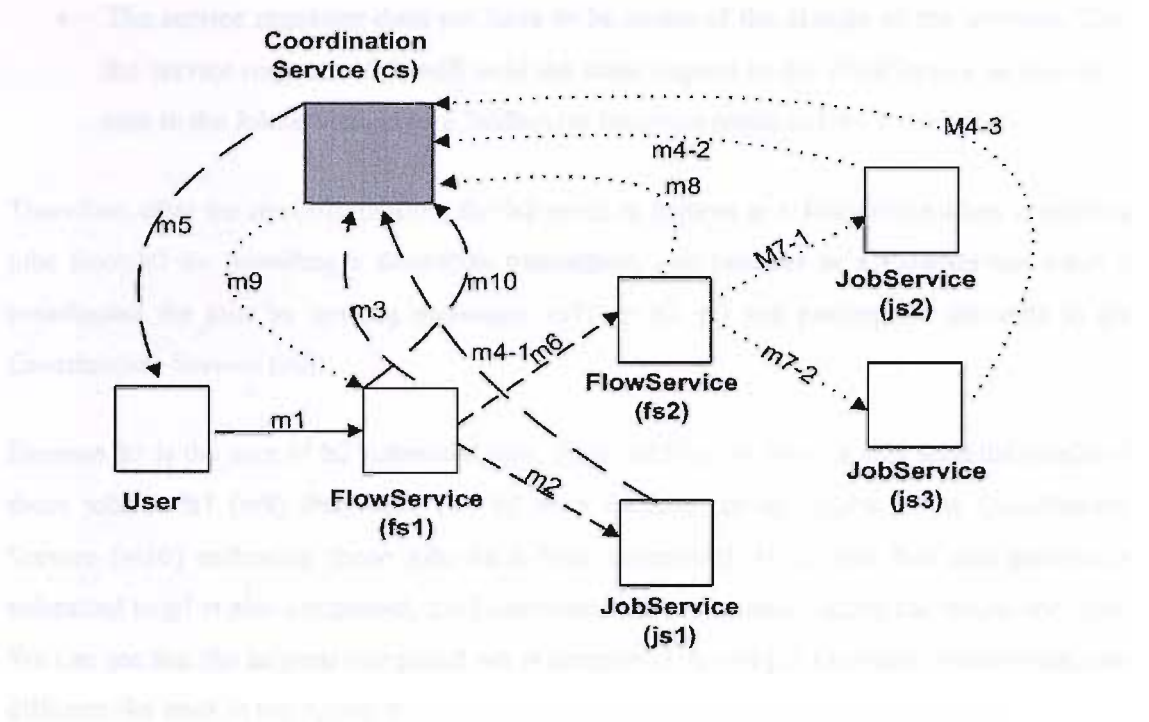


Figure 4-6 A Re-configured Job Submission Example

In our job submission example, to improve the performance, the system introduces some new services. Figure 4-6, a JobService, js2, is replaced by a FlowService, fs2 that has the access to the JobService js2 and js3. The FlowService, fs1, behaves the same as the previous example, except that it will now send a job execute request (m6) to fs2 instead of a JobService. (The actual specification of the FlowService remains unchanged as far as receiving messages from users is concerned. The only change is in how the service identifier JS2 known to fs1 is mapped to an actual service which will be described in Chapter 6 Formal Semantics.) Upon receiving a request from fs1, the fs2 passes two jobs to JobServices, js2 and js3. This way, the jobs received by fs1 can be executed simultaneously by three JobServices.

To dynamically replace a service, the system has to meet the following requirements:

- All interactions started before and after the re-configuration can be executed consistently and without interruption. For example, a JobService is replaced by a FlowService. All requests sent to the original JobService before the replacement can

continue and eventually sent back to the original user. All requests sent to the FlowService after could be executed automatically.

- The service requester does not have to be aware of the change of the services. Thus the service requester, fs1, will send the same request to the FlowService as the one it sent to the JobService, after a JobService has been replaced by a FlowService.

Therefore, after the re-configuration, the fs2 needs to behave as a JobService when it receives jobs from fs1 by providing a jobexecute messagedef, and behaves as a FlowService when it coordinates the jobs by sending messages (m7), to js2, js3 and passing the job state to the Coordination Service (m8).

Because fs1 is the user of fs2 submitted jobs, the Coordination Service will send the results of those jobs to fs1 (m9) this time. The fs1 then forwards those results to the Coordination Service (m10) indicating those jobs have been completed. If the job that was previously submitted to js1 is also completed, the Coordination Service finally sends the results the User. We can see that the original composed job is completed by two job execution interactions, two different dot lines in the Figure 4-6.

OnMessage [to:FS, query: [from:u, query: [job1,job2], context:c], function:jobexecute]

generate new uid

send [to:CS,

query: [from:FS, query:[from:u,query: [job1,job2], context:c], context:uid],

function:jobssubmitted]

send [to:JS1, query: [from:FS, query:job1, context:uid], function:jobexecute]

send [to:JS2, query: [from:FS, query:job2 , context:uid], function:jobexecute]

OnMessage[to:FS, query:[from:cs, query:[from:u, query:job,context:c],result:result,context:uid],

function:jobsreply]

send [to:CS, query:[from:FS, query:job, result:result, context:c], function:jobcomplete]

Figure 4-7 Updated FlowService Specification

We add two `messagedef`s to the `FlowService`: one is how a `FlowService` handles a `jobexecute` request, and another is how it handles a `jobsreply` message.

The first `messagedef` is similar to the `startjobs` `messagedef` in Figure 4-3, but a different function name. And this time, the user, `U`, of `fs2` is a `FlowService`, `fs1`. When `fs2` receives a `jobexecute` message, it will start a new business interaction by providing a new unique identity. As a result, the Coordination Service will create two process records in the `ContextStore` for `fs1`-submitted jobs and `fs2`-submitted jobs. From the Coordination Service point of view, `fs1` is the user of `fs2` submitted jobs; thus, the Coordination Service will send replies to `fs1` corresponding to those jobs.

The second `messagedef` says if the `FlowService`, `fs1`, gets a `jobsreply` from the Coordination Service, it extracts the query and informs the Coordination Service that those jobs that were previously submitted to `fs2` have been completed. The Coordination Service processes a message when the incoming message function matches the one in the **OnMessage** `m`, and updates the state based on the query and context. Therefore, a `jobcomplete` message from a `FlowService` produces the same result as the one from a `JobService`.

Although in Figure 4-6, we only give one coordination service, the nested job submission could be actually coordinated by more than one coordination services and the specifications are similar and straightforward.

4.1.4 Demonstration

Having given the system specification, we will consider a specific job to see how it works. In chapter 6, we will give the operational semantic rules to explain how the system maps the specification parameters to the real values. Here we assume that we have already had those values. We use the value to give the whole job submission interaction flows.

We assume the system has the following services:

- `FlowService1: fs1`
- `FlowService2: fs2`
- `JobService1: js1`
- `JobService2: js2`
- `JobService3: js3`

- User: u
- Coordination Service: cs

We assume the user job is $[[1, [2, j3]]$ and the result will be $[r1, [r2, r3]]$. Also the FlowService1 will generate uid1 as the interaction identity, the FlowService2 will generate uid2 as the interaction identity, and the user will give c as the initial context.

Here are the real messages as marked in Figure 4-6:

No	To (to)	Query (query)	Type (function)	Interaction ID
m1	fs1	from:u, query:[1,[2,j3]], context:c	startjobs	c
m3	cs	from:fs1, query: [from:u, query:[1,[2,j3]], context:c], context:uid1	jobssubmitted	uid1
m2	js1	from:fs1, query:j1, context:uid1	jobexecute	uid1
m6	fs2	from:fs1, query:[2,j3], context:uid1	jobexecute	uid1
m4-1	cs	from: js1, query:j1, result:r1, context:uid1	jobcomplete	uid1
m8	cs	from:fs2, query:[from:fs1, query:[2,j3], context:uid1], context:uid2	jobssubmitted	uid2
m7-1	js2	from:fs2, query:j2, context:uid2	jobexecute	uid2

m7-2	js3	from:fs3, query:j3, context:uid2	jobexecute	uid2
m4-2	cs	from: js2, query:j2, result:r2, context:uid2	jobcomplete	uid2
m4-3	cs	from: js3, query:j3, result:r3, context:uid2	jobcomplete	uid2
m9	fs1	from:cs, query:[from:fs1, query:[j2,j3], context:uid1], result:[r2,r3], context:uid2	jobsreply	uid2
m10	cs	from:fs1, query:[j2,j3], result:[r2,r3], context:uid1	jobcomplete	uid1
m5	u	from:cs, query:[from:u, query:[j1,[j2,j3]], context:c], result:[r1,[r2,r3]], context:uid1	jobsreply	uid1

Table 4-1 Message Examples of a Re-configured Job Submission

Interaction c submits a nested job. The nested job task is completed by 2 parallel interactions, which have been indexed at the ContextStore as uid1 and uid2.

ContextStore[uid1]	ContextStore[uid2]
<pre>[from: fs1, query: [from: u, query: [j1, [j2, j3]], context: c]], [from: js1, query: j1, result: r1], [from: fs1, query: [j2, j3], result: [r2,r3]]</pre>	<pre>[from: fs2, query: [from: fs1, query: [j2, j3], context: uid1]], [from: js2, query: j2, result: r2], [from: fs3, query: j3, result: r3]</pre>

Table 4-2 Re-configured Job Submission Interaction State

This example demonstrates the capability of using DFM to support long-running interactions and dynamic configurations. By sharing interfaces a service behaves multi-functionally. Assuming that the FlowService fs1 is able to handle a jobsreply message, this service does not have to be stopped and rewritten in order to allow a JobService, js2, to be replaced by a FlowService, fs2 (that is able to handle a jobexecute message). A job is distributed and structured using nested context mechanism. The state of the job is captured in the ContextStore and monitored by the Coordination Service. Therefore, a long-running job is executed consistently and a job can be carried on in any state even if a service has been replaced.

4.2 Discussion

We summarised the use of DFM features to model business interactions in the previous chapter. We also saw how using the context mechanism and the query data structure, the DFM can specify a business interaction using a special coordination service in this chapter. Moreover, the way that DFM captures the business interaction state enables that a web service not only to be replaced by a peer web service but also to be replaced by a web service with more workflows.

In this section we summarise the basic DFM message flow patterns (communication patterns) and give the examples of how to use the condition and these basic patterns to specify complex workflows.

4.2.1 DFM Message Flow Patterns

A workflow describes the sequence of information and tasks being executed or passed around. In a service-oriented system, the information and tasks are passed by asynchronous messages. We believe that web services should be stateless which means that each service invocation will be handled by an independent thread. Therefore in DFM, each incoming message will be specified by one and only one `messagedef`. A `messagedef` includes only one incoming message but could have several outgoing messages. Because a control decision may require by several incoming messages, a message flow pattern sometimes is specified by more than one `messagedefs`.

DFM defines two kinds of communication patterns in supporting asynchronous messaging, one-way communication, which amounts to a service receiving a message, and notification, which amounts to a service sending a message. In this section, we use *In* to represent the one-way and *Out* to represent the notification.

A DFM message flow pattern is a combination of the two communication patterns. Three basic message flow patterns are described as follows.

In

In refers to a web service receives a message, but will not send out any messages. This is specified as the following `messagedef`. Any `idactions` or `storeactions` could be given if required.

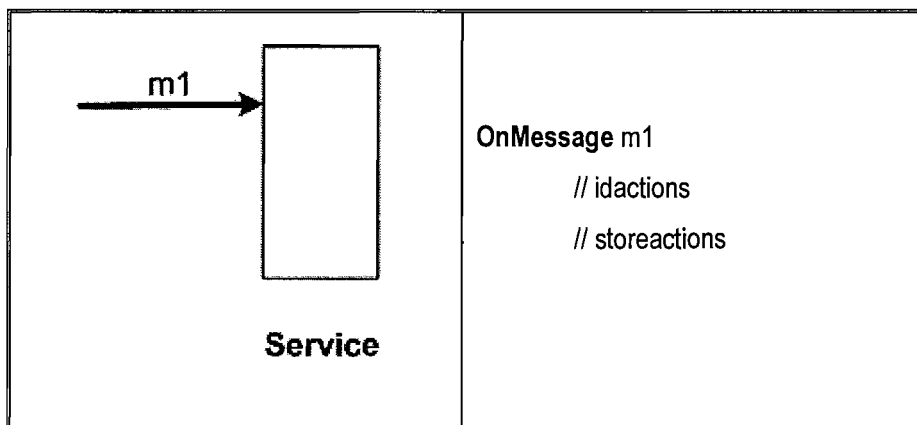


Figure 4-8 The *In* Programming Pattern

In-Outs

In-Outs refers to the situation that a web service may send out a message (or several messages) after receiving an incoming message.

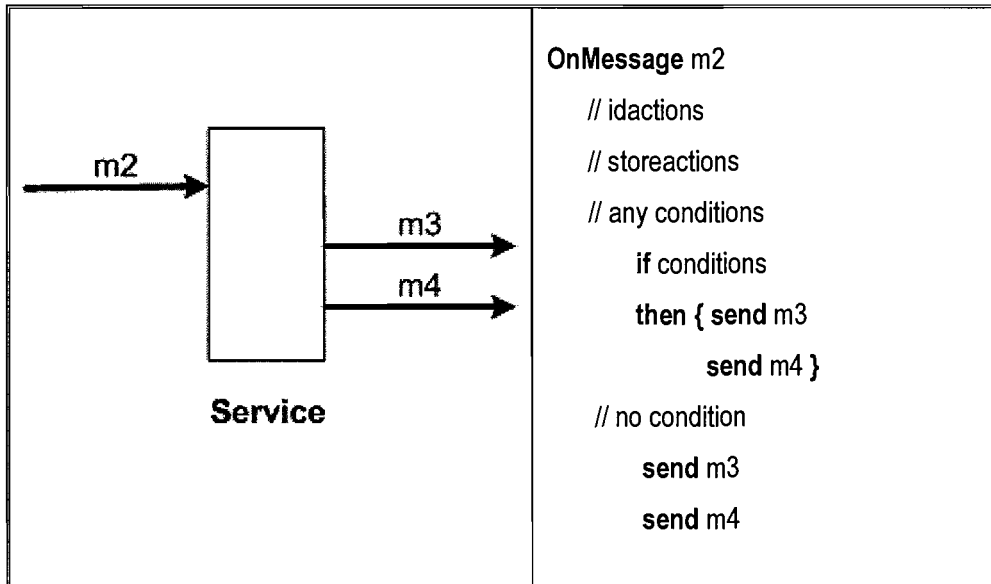


Figure 4-9 The *In-Outs* Programming Pattern

In this pattern, the outgoing messages will not rely on any other incoming messages. As in the above `messagedef`, the service sends out the messages if conditions are evaluated to true or there is no condition, or does not send out the messages if conditions are evaluated to false. The conditions check entries which are only taken from this incoming message (m2 as in the above example); or entries which are internal business data.

Ins-Outs

Ins-Outs refers to the situation that a web service may send out a message (or several messages) after receiving more than one certain incoming messages.

In the Figure 4-10, two `messagedefs` are given to the two incoming messages, m5 and m6. In each `messagedef`, certain entries from the incoming message will be stored into the `ContextStore`, and then a conditional send action checks whether other required messages have been received. Thus, the messages are sent out only if all required messages have been received and any other conditions are true.

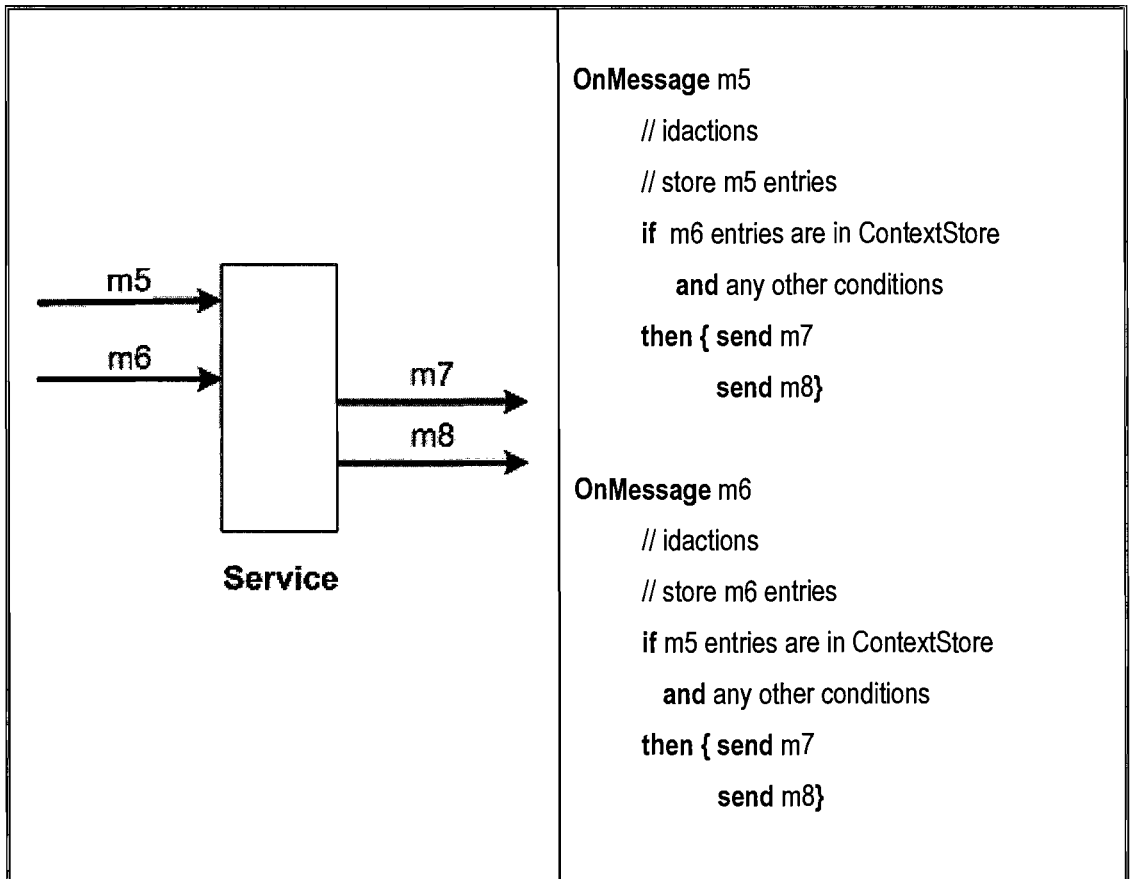


Figure 4-10 The *Ins-Outs* Programming Pattern

In the *Ins-Outs* pattern, the outgoing messages rely on more than one incoming messages. The rely relationship includes logical conjunction, *and*, logical disjunction, *or*, negation, *not*, and exclusive disjunction, *xor*, and so on. The conjunction of the logic conditions and the above basic patterns allow us to specify more flow patterns.

4.2.2 Modelling Workflow Patterns

In this section we discuss how to use the DFM message flow patterns to specify the basic workflow patterns. We will refer the concepts and definitions of workflow patterns from [31] which have been widely cited in other workflow works and publications.

Sequence: “An activity in a workflow process is enabled after the completion of another activity in the same process.” As the travel booking example in Section 3.3, the BookAFlight activity is executed after the BookATravel activity of at the travel agent.

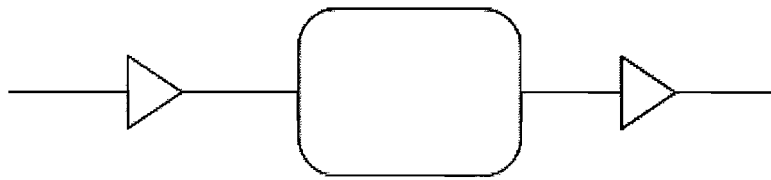


Figure 4-11 Sequence Workflow Pattern

The Sequence workflow pattern could be easily modelled by the DFM *In-Outs* message flow which is similar with Figure 4-9, but only one out message.

Parallel Split: “A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.” As the example in Section 3.3, the BookAFlight activity and BookAHotel activity are executed in parallel.

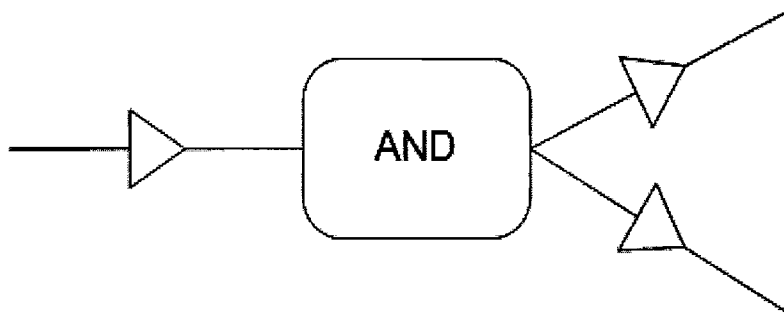


Figure 4-12 Parallel Split Workflow Pattern

The Parallel Split workflow pattern could also be easily modelled by the DFM *In-Outs* message flow which is similar with Figure 4-9. As we have stated early, in DFM, each service invocation will be handled by one and only one thread, the two out messages could be carried out in parallel or in any order whether they are sent to the same or different services.

Synchronisation: “A point in the workflow process where multiple parallel subprocesses / activities converge into one single thread of control, thus synchronising multiple threads.” As the example in Section 3.3, the FlightBookingReply and the HotelBookingReply are synchronised at the travel agent.

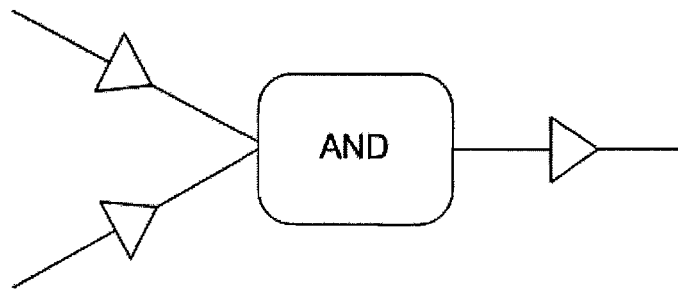


Figure 4-13 Synchronisation Workflow Pattern

The Synchronisation workflow pattern is modelled by the DFM *Ins-Outs* message flow which is similar with Figure 4-10, but no other flow related conditions.

Exclusive Choice: "A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen." As an alternative travel agent scenario in Section 3.3, a user can only book a flight or a hotel instead of book a travel package. The travel agent will choose to execute the BookAFlight activity or the BookAHotel activity based on the user order.

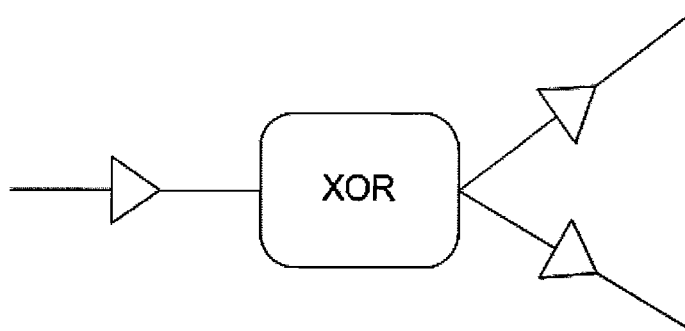


Figure 4-14 Exclusive Choice Workflow Pattern

The Exclusive Choice workflow pattern is modelled by the DFM *In-Outs* message flow which is similar with Figure 4-9, but an exclusive condition.

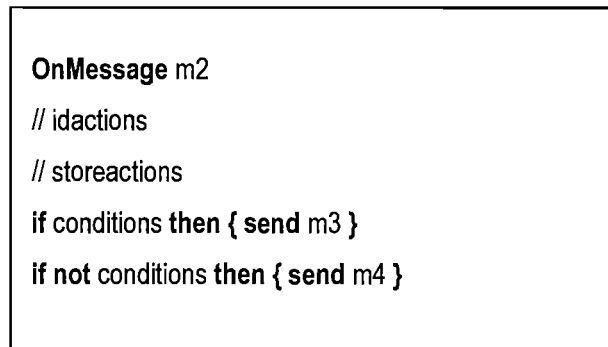


Figure 4-15 Exclusive Choice DFM Model

Simple Merge: “A point in the workflow process where two or more alternative branches come together without synchronisation.” As an updated travel agent scenario in Section 3.3, the travel agent will reply the user the FlightBooking and HotelBook separately, instead of pack them together.

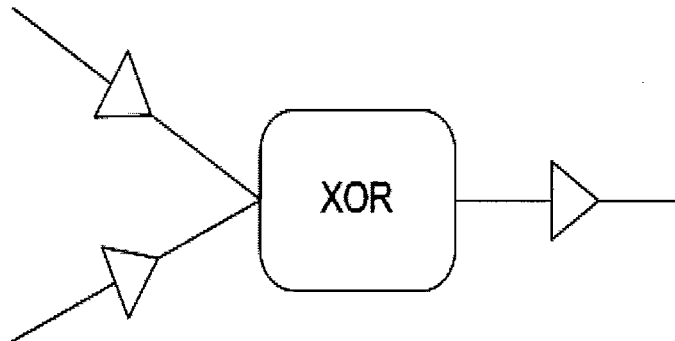


Figure 4-16 Simple Marge Workflow Pattern

The Simple Merge workflow pattern is modelled by two DFM *In-Outs* message flows which are similar with Figure 4-9.

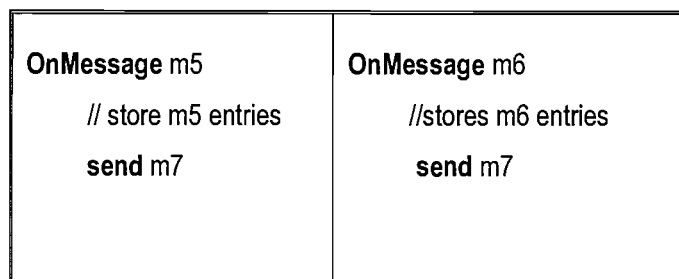


Figure 4-17 Simple Merge DFM Model

In DFM, a `messagedef` is an atomic activity which may only contain simple workflows. A structured workflow will be decomposed into several `messagedefs`. We are not going to model all the structured and advanced workflow patterns introduced in [31]. We believe that most of them could be modelled using the above simple patterns. In the next chapter, we will use those patterns to specify a BEPL4WS application which has complex workflow patterns.

Chapter 5

Comparison

In Chapter 3 and Chapter 4, we gave the formal syntax of our DFM notation, and used some examples to illustrate its use. Also in Chapter 2, we reviewed currently the commonly acknowledged business process modelling languages, BPEL4WS and WSCI. In this chapter, we model a warehouse purchasing example taken from the BPEL4WS specification and compare our DFM with BEPL4WS and WSCI.

5.1 A BPEL Example

In the BPEL4WS specification v1.1, a purchase business process is used through the whole document to exemplify the language. We will model the same example but using our DFM notation.

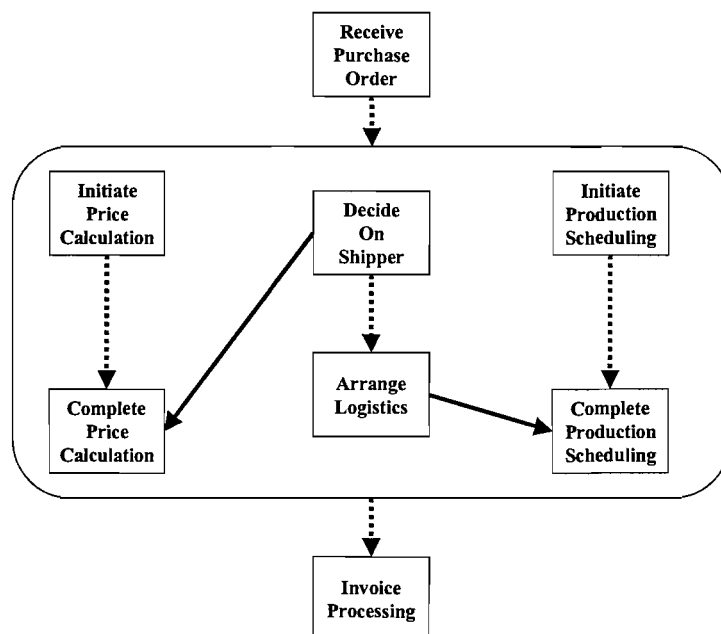


Figure 5-1 A BPEL4WS Example

The purchase business process has been described in section 6.1 of the BPEL4WS specification v1.1 as follows: “The operation of the process is very simple, and is represented in the Figure 5-1. Dotted lines represent sequencing. Free grouping of sequences represents concurrent sequences. Solid arrows represent control links used for synchronisation across concurrent activities.”

“On receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalise the price calculation, and the shipping date is required for the complete fulfilment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.”

Purchase Order Process	
PortType	Operation
purchaseOrderPT invoiceCallbackPT shippingCallbackPT	sendPurchaseOrder sendInvoice sendSchedule
Invoice Service	
PortType	Operation
computePricePT	initiatePriceCalculation sendShippingPrice
Shipping Service:	
PortType	Operation
shippingPT	requestShipping
Schedule Process	
PortType	Operation
schedulingPT	requestProductionScheduling sendShippingSchedule

Table 5-1 BPEL Example - Supporting Services

5.2 The BPEL4WS Model

In order to understand the BPEL4WS, we implemented a simple job submission business process using IBM BPWS4J (see detail in Appendix E A BEPL4WS Implementation using IBM BPWS4J).

We have learned that a BPEL business process is essentially a web service. To deploy a process, a process WSDL is required to provide information like message definitions, supporting operations and partners. We extract the business process description (WSDL) to Table 5-1.

And the workflow of the services (processes) interactions is described in Figure 5-2.

There are four parties involved in this composition, a Purchase Order Process and a Schedule Process, an Invoice Service and a Shipping Service. The Purchase Order Process interacts with user and 3 partners to complete a user purchase.

From the experiment we also know that a BPEL business process is essentially a web service. It publishes its WSDL to the client and is invoked through SOAP RPC calls. As we can see in the BPEL4WS specification, a business process always starts by receiving incoming messages (service invocations). (The BPEL4WS workflow pattern also allows a business process to start from a conjunction of multiple messages.)

In the example, a Purchase Order process is started from a purchase order sent from a client (a user). The *<flow>* element inside the *<sequence>* indicates that the activities inside the *<flow>* are carried out in parallel as in Figure 5-2. Therefore, when a Purchase Order Process receives a *sendPurchaseOrder* from a user, it starts three parallel sub-processes (or threads) correspondingly to interact with the Shipping Service, the Invoicing Service and the Scheduling Process.

```

<process name="purchaseOrderProcess" .....>
<sequence>
  <receive>//purchaseOrderPT(sendPurchaseOrder)</receive>

  <flow>
    <sequence>
      <invoke>//shippingPT(requestShipping)
        //link source "ship-to-invoice"
      </invoke>
      <receive>//shippingCallbackPT(sendSchedule)
        //link source "ship-to-scheduling"
      </receive>
    </sequence>

    <sequence>
      <invoke>//computePricePT(initiatePriceCalculation)
      </invoke>
      <invoke>//computePricePT(sendShippingPrice)
        //link target "ship-to-invoice"
      </invoke>
      <receive>//invoiceCallbackPT(sendInvoice)
      </receive>
    </sequence>

    <sequence>
      <invoke>
        //schedulingPT(requestProductionScheduling)
      </invoke>
      <invoke>//schedulingPT(sendShippingSchedule)
        //target link "ship-to-scheduling"
      </invoke>
    </sequence>
  </flow>

  <reply>//purchaseOrderPT(sendPurchaseOrder)</reply>
</sequence>

</process>

```

Figure 5-2 A Purchase Process in BEPL4WS

However, each of the 3 parallel sub-processes may rely on another sub-process. For example, a `sendShippingPrice` message to the Invoicing Service has to be sent out after the Purchase Process gets the `requestShipping` reply from the Shipping Service. And a `sendShippingScheduling` message to the Scheduling Process relies on the `sendScheduling` message from the Shipping Service. A BEPL4WS specification uses a `<link>` element to describe this kind of cross boundary relationship, which involves two activities that are not in the same syntactic construct.

BPEL4WS is a very complex language, which provides mechanisms include type definition, transaction support, data handling, and workflow patterns. We are not going to explore all of them. We only use the example to review the basic workflow and message patterns so that we can compare them with our DFM notation.

5.3 The DFM Model

We use a customised sequence diagram Figure 5-3 to illustrate the business interactions (messages) between individual parties of the warehouse purchasing example.

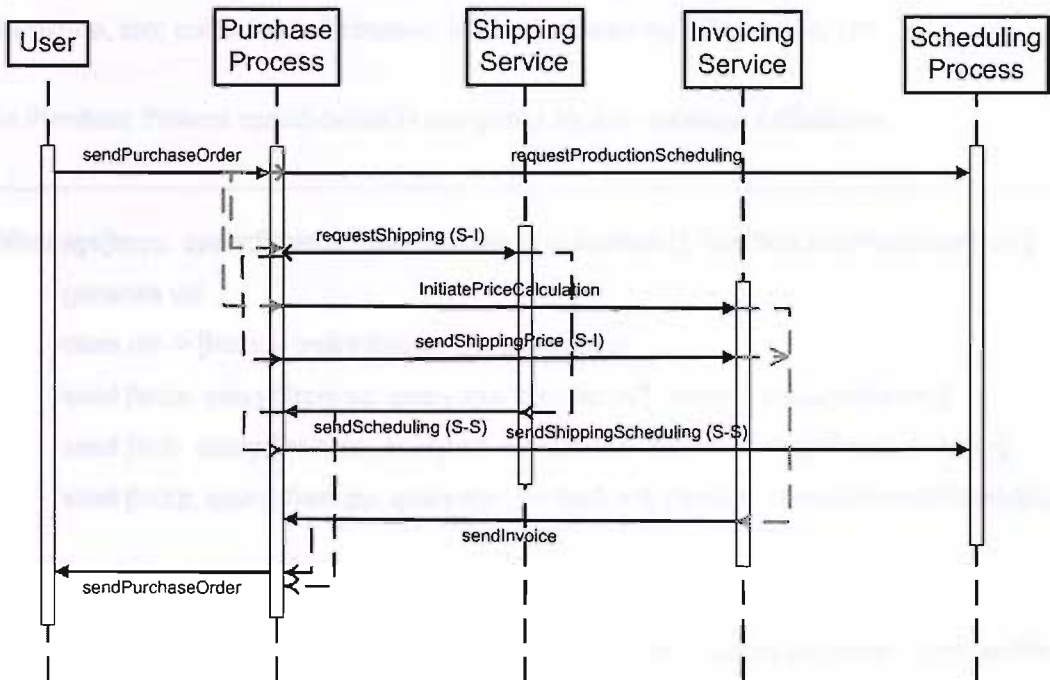


Figure 5-3 The Sequence Diagram of BPEL4WS Example

A one-side arrow line (\rightarrow) indicates a message sent from one service to another. A two-side arrow line (\leftrightarrow) is a request-reply message, where a service will wait for a reply to the request it send out. A dot arrow line (\dashrightarrow) indicates a message correlation, where an outgoing message (arrow end) depends on an incoming message (line begin). We use different colours to distinguish different message correlations.

As we stated earlier, DFM models the service composition by describing the messages exchanged between web services. Each service in the composition specifies its contributions to the business interaction. Therefore, to complete the business interaction, we will give DFM specifications for 4 services (process).

5.3.1 A Purchase Process Specification

In the following example, we assume that business interaction is executed by a User, *u*, and four services or processes: Purchase Order Process, *pp*; Shipping Service, *ss*; Invoicing Service, *is*, and Scheduling Process, *sp*. And a user order is composed by three parts: shipping information, *sso*; invoicing information, *iso*; and scheduling information, *spo*.

The Purchase Process specification is composed by four message definitions.

```

OnMessage[to:pp, query:[from:u, query:[sso,iso,spo],context:c], function:sendPurchaseOrder]
  generate uid
  store uid -> [from:u, query:[sso,iso,spo],context:c]
  send [to:ss, query:[from:pp, query:sso, context:uid], function:requestShipping]
  send [to:is, query:[from:pp, query:iso, context:uid], function:initiatePriceCalculation]
  send [to:sp, query:[from:pp, query:spo, context:uid], function:requestProductScheduling]

OnMessage[to:pp,query:[from:ss,query:sso,result:requestreply, context:uid],function:requestReply]
  store uid -> [from:ss, query:sso, result:requestreply]
  send [to:is, query:[from:pp,query:requestreply,context:uid],function:sendShippingPrice]

OnMessage[to:pp,query:[from:ss,query:sso,result:schedules,context:uid],function:sendScheduling]
  store uid -> [from:ss, query:sso, result:schedules]
  send [to:sp, query:[from:pp, query:ss, context:uid], function:sendShippingSchedule]
  if ContextStore[uid] contains [from:u, query:[sso,iso,spo],context:c]
    [from:ss, query:sso, result:schedules]
    [from:is,query:invoice]
  then{send [to:u, query:[from:pp,
    query[from:u,query:[sso,iso,spo],context:c],
    result:[schedules, invoice],
    context:uid],
    function:purchaseReply ]}

```

```

OnMessage [to:pp,query:[from:is,query:invoice,context:uid],function:sendInvoice]
  store uid -> [from:is,query:invoice]
  if ContextStore[uid] contains [from:u, query:[sso,iso,spo],context:c]
    [from:ss, query:sso, result:schedules]
    [from:is,query:invoice]
  then {send [to:u, query:[from:pp,
    query[from:u,query:[sso,iso,spo],context:c],
    result:[schedules, invoice],
    context:uid],
    function:purchaseReply ]}

```

Figure 5-4 A Purchase Process Specification

First, when pp gets a sendPurchaseOrder request, it creates a new unique id as the new interaction identity. After the user query has been stored into the ContextStore, the pp extracts the user query and creates three messages with attached new interaction id. The three messages: a requestShipping message to Shipping Service; an initiatePriceCalculation message to Invoicing Service; a requestProductScheduling message to Scheduling Process, are as usual sent out in no particular order.

The requestShipping is a request-reply operation in BPEL. The Shipping Service will reply to the Purchase Process the shipping price when it gets the request. Thus, the second messagedef gives the behaviour of receiving the shipping price reply. The pp stores the reply into the ContextStore and then sends the shipping price to the Invoicing Service (the green correlation in Figure 5-3).

As we have stated each incoming message (including reply messages) will have a messagedef in the service specification. The Purchase Process receives four types of messages in total. We have described 2 of them, sendPurchaseOrder and requestReply. The other two, sendInvoice and sendScheduling, are specified in the last two messagedefs. Because the user reply has to include both the information from the last two messages, the two messages are correlated to the purchaseReply (the red correlation in Figure 5-3). Since these two messages are received by the Purchase Process in no particular order, the specifications specify a conditional send

action where the condition is checking whether both two messages of the same interaction have been processed.

5.3.2 A Shipping Service Specification

The Shipping Service basically has two tasks in each business interaction, calculating the shipping price and providing the shipping schedule. Both tasks are triggered by a requestShipping message. In the specification, the Shipping Service packs the shipping price and the schedule into the query and sends them to the Purchase Process.

```
OnMessage [to:ss, query:[from:pp, query:sso, context:uid], function:requestShipping]
    store uid -> [from:pp, query:sso]
    send [to:pp, query:[from:ss, query:sso, result:requestreply, context:uid],
        function:requestReply]
    send [to:pp, query:[from:ss, query:sso, result:schedules, context:uid],
        function:sendScheduling]
```

Figure 5-5 A Shipping Service Specification

We can see that, the requestShipping is a request-reply service invocation in BPEL4WS. In DFM, we model it as an **OnMessage** plus a send action. The outgoing message is similar to all other messages, passing around web services in an asynchronous manner. Therefore, the requesting web service does not have to wait for reply.

5.3.3 An Invoicing Service Specification

The Invoicing Service has two kinds of incoming messages. The first message is received from a Purchase Process asking to calculate product price. The second message is received from the Purchase Process with the shipping price information.

```

OnMessage[to:is, query:[from:pp, query:iso, context:uid], function:initiatePriceCalculation]
  store uid -> [from:pp, query:iso]
  if ContextStore[uid] contains [from:pp, query:iso] [from:pp, query:sprice]
  then { send [to:pp,query:[from:is,query:invoice,context:uid],function:sendInvoice]}

```

```

OnMessage[to:is, query:[from:pp,query:sprice,context:uid],function:sendShippingPrice]
  store uid -> [from:pp, query:sprice]
  if ContextStore[uid] contains [from:pp, query:iso] [from:pp, query:sprice]
  then {send [to:pp,query:[from:is,query:invoice,context:uid],function:sendInvoice]}

```

Figure 5-6 An Invoicing Service Specification

The Invoicing Service will combine the product price and shipping price and create the final invoice (the brown correlation in Figure 5-3). Since the two messages are received in no particular order, the service will check with the ContextStore to see whether all the price information of an order has been collected.

5.3.4 A Scheduling Process Specification

The production and shipping are handled by a Scheduling Process in this example.

```

OnMessage [to:sp, query:[from:pp, query:spo, context:uid], function:requestProductScheduling]
  store uid -> [from:pp, query:spo]
  .....

```

```

OnMessage [to:sp, query:[from:pp, query:ss, context:uid], function:sendShippingSchedule]
  store uid -> [from:pp,query:ss]
  .....

```

Figure 5-7 A Schedule Service Specification

In this particular business interaction, the Scheduling Process will not interact with all other services. Two messages of each business interaction will be sent to the Scheduling Process, one for production and one for shipping schedule. The process stores the interaction into the ContextStore and continues carrying on its business.

5.4 BPEL4WS and DFM

BPEL4WS and DFM are formal notations to specify service-oriented business processes and interactions, aiming to facilitate system automation, especially in business-to-business domains. Given the formal models of a warehouse purchasing application using both BPEL4WS and DFM, we review the differences between them as follows.

5.4.1 Formal Model vs. Industrial Standard

As developed by the two IT giants, IBM and Microsoft, the BPEL4WS specification is becoming an industrial standard. To turn the BPEL4WS into real implementations, especially in commercial systems, the specification must be reliable, secure and comprehensive to provide all sorts of capabilities in order to meet business requirements such as transaction support, compatibilities with other web service standards and so on. The specification itself becomes more and more complex, which makes the system much harder to implement and verify.

The DFM is a research outcome. The purpose of the DFM is to use it in between formal mathematical models and real implementations. Unlike BPEL4WS, the DFM focuses on business interactions over distributed web services, and does not provide strong support on the transactions. The DFM is a concise notation in comparison with BPEL4WS, but has the capability to be converted into XML data structure which could be further fitted into other web service standards. The advantage of DFM over BPEL4WS is that the concise abstract model could be easily verified (see Appendix C and D) or compiled into other formal model checking tools.

5.4.2 Distributed Workflow vs. Centralised Workflow

Although the BPEL4WS and the DFM work on the same business domain, they use different approaches to facilitate the workflow automation. BPEL4WS provides a centralised workflow model, which uses a stateful business process orchestrated with stateless web services or other business processes.

Process State	<pre> <variables> <variable name="PO" messageType="Ins:POMessage"/> </variables> </pre>
Interaction and Partnership	<pre> <partnerLinks> <partnerLink name="purchasing" partnerLinkType="Ins:purchasingLT" myRole="purchaseService"/> </partnerLinks> </pre>
Behaviour and Workflows	<pre> <sequence> <assign> <copy> <from variable="PO" part="customerInfo"/> <to variable="shippingRequest" part="customerInfo"/> </copy> </assign> ... </sequence> </pre>

Table 5-2 A Centralised Workflow Model

A business process is defined by an abstract business protocol and an executable model. A business protocol gives the interaction by describing the partner relationships between a process and web services. An executable model describes the logic and state nature and sequence of web service interactions. The business process states are stored as variables at the BEPL4WS container.

A centralised workflow releases the state maintenance requirement for the web services, but it adds huge complexities to the implementation of the BPEL4WS container. Successfully

developing and deploying a BPEL4WS process using any available engines is really an achievement for anyone.

In contrast, DFM does not have a centralised workflow, but a distributed model. The workflow is specified by all the services involved in a business interaction.

update state	OnMessage [to:a, query:[from:s, query:f, result:r, context:uid], function:shopReply]
enquire state and initiate suitable actions (send messages)	store uid -> [from:s, query:f, result:r] in ContextStore if ContextStore[uid] contains [from:a, query:[from:u,query:[f,h], context:c]], [from:s1, query:f, result:r1], [from:s2, query:h, result:r2] then { send [to:u, query:[from:a, query:[from:u,query:[f,h],context:c], result:[r1,r2], context:uid], function: bookReply] }

Table 5-3 A Distributed Workflow Model

The interaction state in DFM is stored in an independent persistent component, ContextStore. A web service accesses its own or shared ContextStore to enquire or update the state of an interaction.

DFM provides a very simple business protocol between web services and business component, like the ContextStore. It could be easily extended and implemented into different web service environments. It is very flexible, could be used to model various applications. We have demonstrated in the last chapter that DFM is particularly suited to applications which are required to be coordinated.

A centralised workflow design pattern limits the dynamic system behaviours. Storing the process state in temporary variables at the web container as BPEL4WS does, makes it infeasible to dynamically replace a web service or a business process in the middle of a business interaction.

The DFM separates the interaction state from the web services to enable the dynamic behaviours. In the last chapter, we have given an example of using DFM to capture the business interaction when a web service is replaced by another web service with more work flow behaviours.

5.4.3 Support for Long-running Interactions

As we have stated earlier, the DFM is intended to model long-running business level interactions. A business-to-business interaction is not just a transaction. Web service transactions are a subset of web service interactions [23]. The BPEL4WS specification only provides mechanisms to support long-running transactions by providing two-phase commit protocols and compensation activities [25]. However, a business level interaction may include multiple transactions and takes much longer than a transaction.

As long-running business interactions are the basis of modern enterprise application, we believe it should also be considered as part of the service composition. DFM uses a persistent component to store the business interaction state and a context mechanism to coordinate concurrent business interactions. Therefore, business interactions running for days or weeks could be executed consistently.

5.4.4 Summary

BPEL4WS is a comprehensive industrial standard which provides formal specifications for service-oriented applications. In contrast, we developed a concise formal modelling notation, DFM, to specify and verify such applications. DFM and BPEL4WS focus on different scope of the service oriented applications where DFM is intended to support long-running business interactions and BPEL4WS is aimed at long-running business transactions. The approaches and mechanisms to specify the service composition in DFM and BPEL4WS are also different. A BPEL4WS specification is a composition of a fat stateful process and thin stateless web services, whereas a DFM specification is a composition of peer stateless web services.

However, the two specification languages do work together. The DFM can be used as a complement of the BPEL4WS, for example to model context aware applications.

5.5 WSCI and DFM

WSCI (Web Service Choreography Interface) is another web service composition language. It describes the flow of messages exchanged by a web service participating in choreographed interactions with other services. Similar to the idea of DFM, WSCI specifies behaviours of individual services which are part of the business interaction.

```
// WSCI interface agent
.....
<process name="agentbooking" instantiation="message" >
  <sequence>
    <action name="bookTravel" role="Agent" operation="bookTravel" />
    <action name="bookReply" role="Agent" operation="bookReply" >
      <correlate correlation="bookCorrelation" />
      <call process="agentbooking" />
    </action>
  </sequence>
</process>
.....
// wsdl port type definitions
// Correlations and selector definition
.....
<selector property="bookingNo" type="itineraryID" xpath="/text()" />
.....
<correlation name="bookCorrelation" property="bookingNo" >
  <documentation> correlation based on a booking number. </documentation>
</correlation>
```

Figure 5-8 A TravelAgent Example in WSCI

5.5.1 A Travel Agent Example in WSCI

We specify our previous travel agent example in WSCI in Figure 5-8. We can see that WSCI is actually an extension of the WSDL. The extension gives the control flow to operations supported by an individual service. However it does not give an executable model of how distributed services coordinate with each other to achieve the business interaction.

DFM also uses the choreography approach to model service composition where each service has to specify their contribution to the interaction. And the relationships among operations supported by a service is not just based on the control flow but also based on the business interaction state. Thus DFM gives an executable service composition model. In the next chapter, we will give the detail operational semantics of DFM describing the system execution rules.

Chapter 6

Formal Semantics

Web service technologies enable modern enterprise applications to be implemented over a heterogeneous web environment. A web service publishes its interfaces in an XML-based document, WSDL, and is invoked through a XML-based messaging protocol, SOAP [29, 32]. Using platform independent and standard XML documents, a service consumer can invoke a web service following the shared understanding but does not care how the service is implemented.

A service-oriented application is composed of dynamic web services orchestrated using asynchronous messages. The web services are owned and managed by many business partners. This architecture provides benefits over traditional applications on interoperability, flexibility, dynamic configurations. However, it also adds considerable complexity to the implementation and verification [33].

To adapt to the web service architecture, a transition rule is given to describe the possible behaviours of a system of inter-related web services, in terms of the messages that can be exchanged during the execution of one or more business interactions, and the effect each message execution has on the business interaction state. The operational semantics of a DFM specification is defined using such rules.

6.1 Operational Semantics

Our DFM notation is a message based workflow notation. The systems which can be specified in DFM are composed of a set of independent web services, coordinated using asynchronous messages. The execution of a message is handled by the web service which the message was sent to. Since communication is asynchronous, the sequence of the message execution is undetermined. The operational semantics describes all the possible behaviours of a message execution.

The operational environment within which the web service communications are taking place is modelled using a virtual daemon. The daemon uses a message pool to manage the sending and receiving of messages by the services, and at the same time maintains service configurations. When a message is added to the pool, it is organised in a message-set of a web service which the message was sent to. When the message is executed, it will be removed from the message-set and the pool.

The execution of a message is carried out in the following key steps: the message is matched to a message pattern in the specification; the actions which need to be carried out are determined, based on information generated from the system specification and on the current interaction; and finally, the resulting message patterns are evaluated, the results are added to the message pool, and the original message is removed from the pool.

The DFM notation intends to deal with the dynamic configuration: a web service does not have to be stopped and rewritten when the service composition is changed. To support this, we allow the service names used in system specifications to be mapped to actual services at runtime. This is achieved using the configuration tables dynamically provided by the environment.

Formally, the operational semantics associates, to each DFM specification, a labelled transition system whose states correspond to possible states of the system (defined by the messages awaiting execution and by the current state of the ContextStore), and whose labels correspond to message executions. The transition rule is based on the system specification that has been defined using DFM notation and all the variables and constant values that have been provided by the system daemon. Defining this transition system requires several auxiliary functions, which we now describe.

6.1.1 Specification Functions

Specification functions provide the information extracted from a given DFM specification. For example, ids generated by the idactions of a system is used when storing information into the ContextStore or when sending out messages; service names defined in the message patterns are used when matching a message pattern to a real message.

ServiceId

A system is specified using a collection of `messagedefs`. These can be organised based on the type of the service which handles each message. The set `ServiceId` contains all service identifiers present in the specification:

$$\text{ServiceId} = \{s \mid \text{to:s appears inside an } \mathbf{onMessage} \ m \}$$

Message

A `messagedef` describes the actions taken by a web service in response to an incoming message. It contains all message patterns present in the specification:

$$\text{Message} = \{m \mid \mathbf{onMessage} \ m \text{ appears inside a } \text{messagedef}\}$$

ids function

The DFM uses a context to identify a business interaction. A unique identity, `id`, created by an idaction of a `messagedef`, is used to associate an interaction to each message or entry in the `ContextStore`. As part of executing a message, new interaction identities can be created in order to identify a specific sub-interaction. The `ids` function gives, for each message `m`, the interaction identities generated upon the execution of `m`:

$$\text{ids}: \text{Message} \rightarrow \mathbb{P}(\text{String})$$

$$\text{ids}(m) = \{ \text{id} \mid \text{id appears in the idaction of } \mathbf{onMessage} \ m \}$$

context and vars function

Similarly, the `context` and `vars` function give, for each message `m`, the interaction identities passed as parameters to `m` using the `context:` property, and the names of all the other variables used as parameters in definition of `m`, respectively:

$$\text{context, vars} : \text{Message} \rightarrow \mathbb{P}(\text{String})$$

$$\text{context}(m) = \{ c \mid \mathbf{context:c} \text{ appears inside } m \}$$

$$\text{vars}(m) = \{ \text{var} \mid \text{var appears as an element inside } m \}$$

Store action function

The function `storeactions` maps each message `m` to the set of `storeactions` which need to be carried out as a result of executing `m`. The effect of these actions is that certain interaction states are updated, by storing related data into the persistent component, `ContextStore`.

$$\text{StoreAction} = \{\text{storeA} \mid \text{storeA is generated by storeaction}\}$$

$$\text{storeactions: Message} \rightarrow \mathbb{P}(\text{StoreAction})$$

$$\text{storeactions}(m) = \{\text{storeA} \in \text{StoreAction} \mid \text{storeA appears inside } \mathbf{onMessage\ m}\}$$

Send action functions

The `sendactions` function gives, for each message, `m`, the set of `sendactions` which will be carried out unconditionally upon the execution of a message matching `m`.

$$\text{SendAction} = \{\text{sendA} \mid \text{sendA is generated by sendaction}\}$$

$$\text{sendactions: Message} \rightarrow \mathbb{P}(\text{SendAction})$$

$$\text{sendactions}(m) = \{\text{sendA} \in \text{SendAction} \mid \text{sendA appears inside } \mathbf{onMessage\ m}\}$$

For each message `m`, the set `conds(m)` gives the conditions which must be evaluated as part of the execution of `m`:

$$\text{Condition} = \{c \mid c \text{ is generated by condition}\}$$

$$\text{conds}(m) = \{c \in \text{Condition} \mid c \text{ appears inside } \mathbf{onMessage\ m}\}$$

while the function `csendactions(m)` gives the actions associated to each such condition:

$$\text{csendactions}(m) : \text{conds}(m) \rightarrow \mathbb{P}(\text{SendAction})$$

$$\text{csendactions}(m)(c) = \{\text{sendA} \in \text{SendAction} \mid \text{sendA appears} \\ \text{inside } \mathbf{if\ c\ then\ \{...\}} \text{ of } \mathbf{onMessage\ m}\}$$

The set `Service` contains the names of all services relevant to a particular specification. In the following, we assume:

$$\text{Service} = \{S_1, S_2, \dots, S_n\}.$$

For each service, S_i , a message-set, MS_i , gives the pending messages of S_i , as found in the message pool (see also in Figure 6-1).

Service Id	Service
f	S_i
t	S_j
.....

Table 6-1 A Configuration Table of a Service, S_i

In addition, for each service S_i , a configuration table is given that links the service identifiers used in the definition of S_i to actual services (elements of `Service`). This is captured by the function:

$$\text{Config}_i : \text{ServiceId} \rightarrow \text{Service}$$

To give a system configuration, the `Service` set and the configuration table for each service are required and fixed.

MessageVal

The message pool is the container for messages awaiting execution. The set `MessageVal` contains all possible such messages.

$$\text{MessageVal} = \{[\text{to:t}, \text{query:q}, \text{function:fun}] \mid t \in \text{string}, q \in \text{string}, \text{fun} \in \text{string}\}$$

Matching function

For each service S_i , the `matchesi` function gives the message pattern m that corresponds to a message M in the message pool.

$\text{Matches}_i: \text{MessageVal} \rightarrow \text{Message}$

$\text{Matches}_i([\text{to}:t, \text{query}:q, \text{function}:f])$

$= [\text{to}:t', \text{query}:q', \text{function}:f] \in \text{Message}$ if $\text{config}_i(t') = t$

Evaluation of a document

The evaluation function $\text{eval}_{M,i}$ defines how to evaluate document expressions appearing inside a message definition m , based on the values provided by a corresponding actual message M waiting to be executed by S_i , and on the values generated from the environment upon the execution of M . The function $\text{eval}_{M,i}$ is defined inductively on the structure of document expressions. The base cases correspond to service names, message parameters/contexts and interaction identities:

$\text{eval}_{M,i}(t) = \text{config}_i(t)$ if $m = \text{matches}_i(M)$ and **to**: t or **from**: t inside m

$\text{eval}_{M,i}(\text{var}) = \text{value}$ obtained from $m = \text{matches}_i(M)$, if $\text{var} \in \text{vars}(m) \cup \text{context}(m)$

$\text{eval}_{M,i}(\text{id}) = \text{new value}$ generated from the environment,
if $m = \text{matches}_i(M)$ and $\text{id} \in \text{ids}(m)$

while the induction cases correspond to messages, queries and entries:

$\text{eval}_{M,i}([\text{to}:t, \text{query}:q, \text{function}:f]) = [\text{to}:\text{eval}_{M,i}(t), \text{query}:\text{eval}_{M,i}(q), \text{function}:f]$

$\text{eval}_{M,i}([\text{var}_1, \dots, \text{var}_n]) = [\text{eval}_{M,i}(\text{var}_1), \dots, \text{eval}_{M,i}(\text{var}_n)]$

$\text{eval}_{M,i}([\text{from}:f, \text{query}:q, \text{context}:c])$
 $= [\text{from}:\text{eval}_{M,i}(f), \text{query}:\text{eval}_{M,i}(q), \text{context}:\text{eval}_{M,i}(c)]$

$\text{eval}_{M,i}([\text{from}:f, \text{query}:q, \text{result}:r, \text{context}:c])$
 $= [\text{from}:\text{eval}_{M,i}(f), \text{query}:\text{eval}_{M,i}(q), \text{result}:\text{eval}_{M,i}(r), \text{context}:\text{eval}_{M,i}(c)]$

$\text{eval}_{M,i}([\text{from}:f, \text{query}:q]) = [\text{from}:\text{eval}_{M,i}(f), \text{query}:\text{eval}_{M,i}(q)]$

$\text{eval}_{M,i}([\text{from}:f, \text{query}:q, \text{result}:r]) = [\text{from}:\text{eval}_{M,i}(f), \text{query}:\text{eval}_{M,i}(q), \text{result}:\text{eval}_{M,i}(r)]$

Evaluation of a condition

An additional function needs to be defined for evaluating the conditions appearing inside message definitions, given an actual message M to be executed by service S_i , and a particular state of the ContextStore, CS :

$$\text{eval}_{M,i} : \text{conds}(m) \times \text{ContextStore} \rightarrow \{\text{true}, \text{false}\}$$

$$\begin{aligned} \text{eval}_{M,i}(\text{ContextStore}[id] \text{ contains } e_1, \dots, e_n, CS) \\ &= \text{true if each } \text{eval}_{M,i}(e_j) \in CS[\text{eval}_{M,i}(id)] \\ &= \text{false otherwise} \end{aligned}$$

The boolean operators are evaluated in the usual way.

$$\text{eval}_{M,CS,i}(c_1 \text{ and } c_2, CS) = \text{eval}_{M,CS,i}(c_1, CS) \wedge \text{eval}_{M,CS,i}(c_2, CS)$$

$$\text{eval}_{M,CS,i}(c_1 \text{ or } c_2, CS) = \text{eval}_{M,CS,i}(c_1, CS) \vee \text{eval}_{M,CS,i}(c_2, CS)$$

$$\text{eval}_{M,CS,i}(\text{not } c_1, CS) = \neg \text{eval}_{M,CS,i}(c_1, CS)$$

Send function

For each message M waiting to be executed by some service S_i , and each state CS of the ContextStore, the function $\text{send_fun}_{M,CS,i}$ gives, for each service, S_j , the messages that are going to be sent to S_j as a result of executing M .

$$\text{send_fun}_{M,CS,i} : \text{Service} \rightarrow \mathbb{P}(\text{MessageVal})$$

$$\text{send_fun}_{M,CS,i}(S_j) = \{M' \in \text{MessageVal} \mid m = \text{matches}_i(M)$$

$$\text{and send } m' \in \text{sendactions}(m) \cup (\text{csendactions}(m)(c)$$

$$\text{where } \text{eval}_{M,CS,i}(c, CS) = \text{true})$$

$$\text{and } \text{to:to} \text{ in } m' \text{ and } \text{eval}_{M,i}(\text{to}) = S_j$$

$$\text{and } M' = \text{eval}_{M,i}(m')\}$$

6.1.3 System Configurations

The operational semantics of a DFM specification is defined in terms of transitions between system configurations, where a configuration describes the messages waiting to be executed by each service, together with the current state of the ContextStore. Formally, a configuration is tuple:

$$(MS_1, \dots, MS_n, CS) \in \mathbb{P}(\text{MessageVal}) \times \dots \times \mathbb{P}(\text{MessageVal}) \times \text{ContextStore}$$

and transitions between configurations have the form:

$$(MS_1, \dots, MS_n, CS) \xrightarrow{\text{execute}(M)} (MS_1', \dots, MS_n', CS')$$

where the later configuration is completely determined by executing M in the initial configuration.

6.1.4 A Transition

$$\begin{array}{l}
 \text{execute } (M) \\
 (MS_1, \dots, MS_n, CS) \xrightarrow{\hspace{2cm}} (MS_1', \dots, MS_n', CS') \\
 \text{if } (M \in MS_i) \\
 \text{where: matches}_i(M) = m \\
 \text{for every } id \in \text{ids}(m) \cup \text{context}(m), \\
 CS'[\text{eval}_{M,i}(id)] = CS[\text{eval}_{M,i}(id)] \\
 \quad \cup \{ \text{eval}_{M,i}(e) \mid \text{store } id \rightarrow e \in \text{storeactions}(m) \} \\
 \text{for every } j \neq i, MS_j' = MS_j \cup \text{send_fun}_{M,CS',i}(N_j) \\
 MS_i' = MS_i \setminus \{M\} \cup \text{send_fun}_{M,CS',i}(N_i)
 \end{array}$$

Figure 6-2 A Transition Rule

A single operational rule, given in Figure 6-2 describes when the transition is possible, and what the outcome of the transition is. Specifically, the message M being executed must belong to some message-set MS_i , and its execution (by service S_i) results in M being taken out from the message-set, in some interaction states being updated, and in new messages being added to some of the message-sets. The $match_{e_i}$ function (of service S_i) is used to determine the message pattern m that matches the message M , and subsequently the $eval_{M,i}$ and $send_fun_{M,CS,i}$ functions are used to compute the required state updates (as specified in the storeactions of m), and the new messages to be added to the message-sets (as specified in the sendactions of m).

This operational rule can be used to generate a transition system, whose states are configurations, and whose transitions are all possible instances of the given rule. Any configuration containing at least one message in the message-sets can, in principle, be chosen as an initial state. Unfolding the transition system starting from this state yields all the behaviours which can be exhibited by the services S_1, \dots, S_n , while they cooperate towards the execution of the messages in the initial configuration.

For the TravelAgent specification in Section 3, an abstraction of the resulting transition system is given in Figure 6-3 (where the system states are represented using only the messages waiting to be executed).

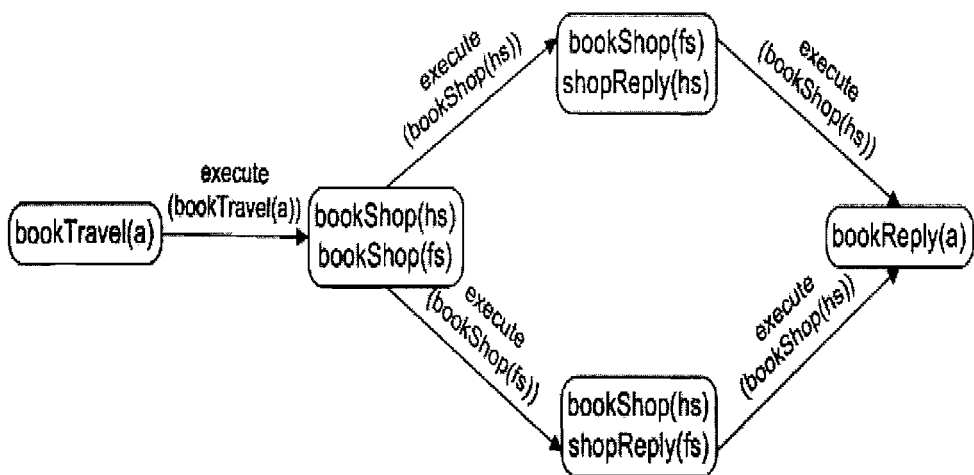


Figure 6-3 A Travel Agent Transition Diagrams

6.2 Discussion

Web services are independent entities in a service-oriented system. As a result, the sending of a message should not rely on the availability of other web services. A service should be able to pick up a message after it has recovered from a failure, or immediately after its addition to a service-oriented system. The operational semantics presented here conforms to this asynchronous communication behaviour which also supports dynamic behaviours such as adding and replacing a web service.

We have stated that DFM intends to deal with the dynamic system configurations. An example of updating a system configuration by replacing a simple service by a work flow service has been presented in chapter 4. We also described earlier how to map a system prototype to an actual configuration using configuration tables. Transition rules for replacing a service by another service could be based on the functions we described in last section and some new functions on the configuration tables.

6.2.1 Dynamic Configuration Scenario

The system we have just discussed is composed of a set of independent web services. Dynamically configuring a system refers to the behaviours like adding a web service, removing a web service, or replacing a web service by another service.

Dynamic configuration is a complex problem. The dynamic behaviours are various in different systems. As our work is not mainly about dynamic systems, in this section, we use some sample dynamic behaviours and discuss the possible solutions of how the system could handle the dynamic behaviours. The discussion aims to demonstrate that our operational semantics has the capability to handle dynamic configuration behaviours. To give the full operational semantic rules we will have to enrich our current DFM notation, [see detail in 7.3 future works].

The dynamic configuration concept here is different from the one we used in Chapter 4 where we demonstrated that the context mechanisms and document record data structure of DFM syntax allows a simple job service being replaced by another web service with more complex workflow patterns. In this section, we are concerned with the operational rules. The operational rules assume that a system specification has been given and fixed.

Dynamic behaviours are various in different systems. Some system configurations are specified and controlled by a human being, a system administrator for instance; some system configurations are updated by the system itself automatically. We are not going to distinguish them because our discussion is concerned with how the business interactions are executed during a system reconfiguration. Thus, we let the daemon to handle the re-configuration which could be initiated by either a human being or by the individual service supported by the system engine.

Adding a Web Service

The message pool and the configuration tables allow that the independent services do not have to be aware of the system change. However, from the operational point of view, when a new service is added into the system, as the job submission example in Chapter 4, a new job service, job service 3, is added to the system to enhance the performance, the system has to provide the capabilities to introduce it to other services, to give a message container for the service and so on. In our current operational environment, when a web service, S_{n+1} , is introduced into the system, the system daemon needs to create a new message-set, MS_{n+1} , and a new configuration table for it.

Adding a web service into the system will not affect any existing business interactions. Since the transition rule describes the behaviour of a message execution, the newly added service will only join a business interaction execution when a message is added into the message-set at the message pool. This happens either when the daemon initialises a message to it, or the service takes the place of another service which we will discuss shortly in replacing a web service scenario.

Removing a Web Service

We have described that the message pool and web services are independent components in our system. Messages are pending at the message pool waiting to be processed. A web service can pick up a message at any point. When a service is going to be removed, the system has to ensure that all the unfinished business interactions that the service took part in will be executed consistently. This could be done either by only removing the service after all the unfinished interactions are completed or by picking up another service that takes the place of the removed service. When a service is removed the daemon has to update the `ServiceId`, `Service` and `Config` to make sure that data is consistent with the system configuration.

Replacing a Web Service

Before discussing the behaviour of replacing a web service, let's refine the workflow concepts in our model.

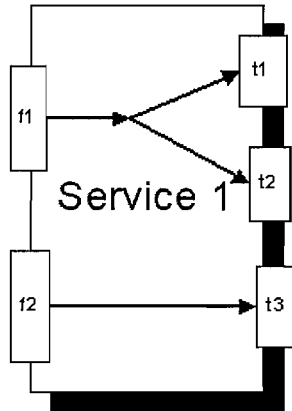


Figure 6-4 A Service Flow Example

A web service provides pieces of functions which could be invoked by various services. A *Service Flow* is given by a specification using DFM, describing the outputs in relation to an input as the opaque arrow in Figure 6-4 and Figure 6-5. The service itself does not have to be aware of the real destination of any outgoing messages.

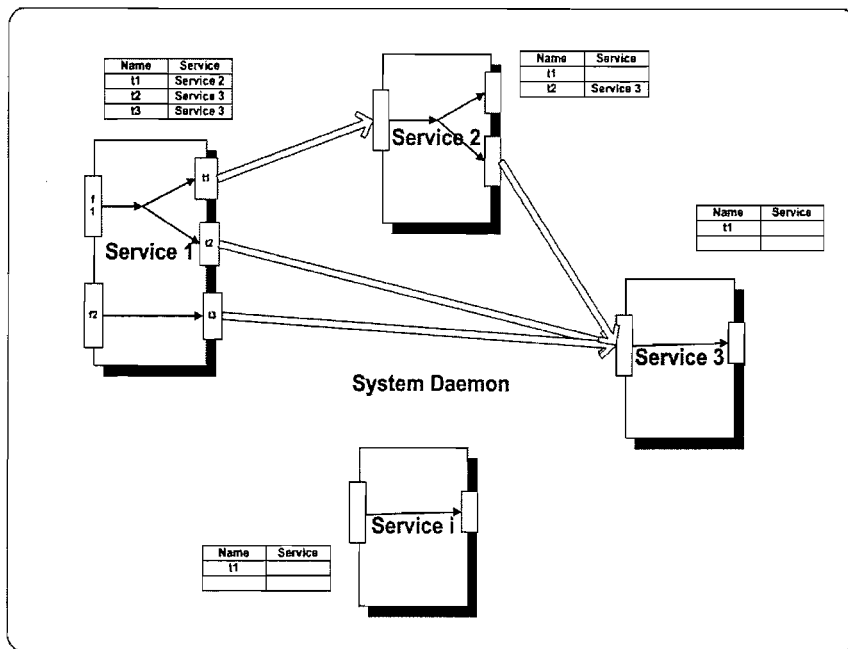


Figure 6-5 A System Flow Example

The *System Flow* describes a service composition that delivers a business function as the transparent arrow in Figure 6-5. A system daemon gives each service a configuration table which maps a *Service Flow* into a *System Flow*. Our operational semantics is based on a given specification. Thus replacing a service here is about updating the *System Flow*.

In a service-oriented system, a web service may provide multiple functionalities and is used by various services or clients. Replacing a service could happen when a service is broken down which is described in replacing a service globally section, or when a client requires a change which is described in replacing a service locally section. Again, in the discussion we will let the daemon handle it whether it is initiated by a system administrator or by an individual service.

Replacing a Web Service locally

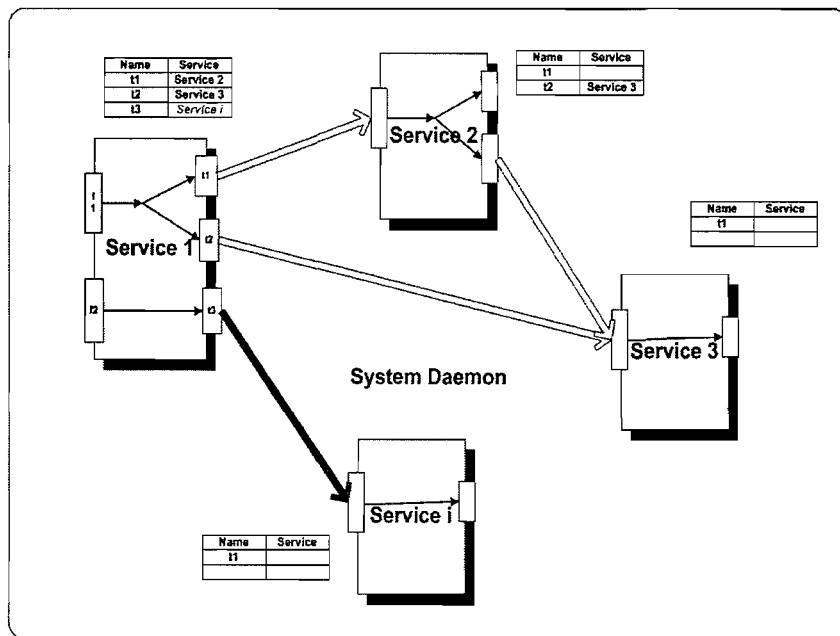


Figure 6-6 Replace a Service Locally

Local replacement refers to the situation that the system daemon tells a service, Service 1, to send a certain type of messages (in red) to a new service, Service i, as in Figure 6-6 instead of previously the Service 3 in Figure 6-5. As in early example in Chapter 4, a FlowService, fs1, is asked to send jobs to a new FlowService, fs2, instead of the JobService 2, js2. We assume that the replaced service is still alive. As in Figure 6-5, the Service 3 executes its messages as before.

From the transition point of view, the messages sent before the update will be processed by the old service, those after the update need to be put into the message-set of the new service. Thus the daemon needs to update the configuration table of Service 1, updating from $t_3 \rightarrow$ Service 3 to $t_3 \rightarrow$ Service i .

Replacing a Web Service globally

Global replacement refers to the situation that a service is going to take the whole position and functionality of another service as in Figure 6-7. For example, the Service 3 has no longer been available, for example a service is out of service. The system then decides to let Service i represent Service 3.

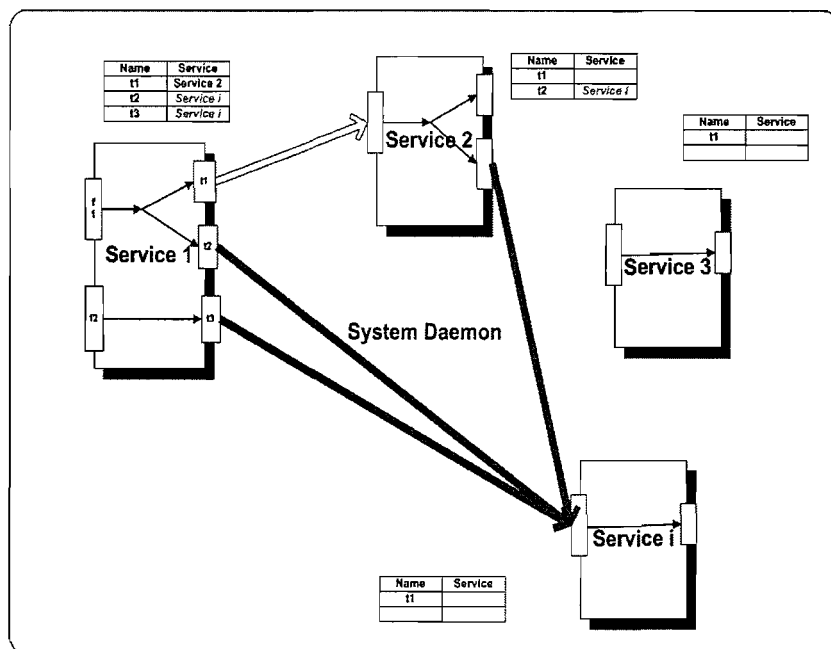


Figure 6-7 Replace a Service Global

To support this kind of replacement, the system has to consider two conditions. First, the system is required to ensure all the services will be notified of the change so that all future messages will be sent to the new service. To do this, the daemon may look through all service configuration tables and update the related rows.

Second, since a service is no longer in function, the system has to make sure that all pending messages of the service will be processed. To do this, the system daemon needs to move all the messages pending at the message-set of the original service to the message-set of the new

service. For those interactions that the original service took part in and haven't been finished, the daemon may need extra engine to ensure the future replies will be sent to the new service.

6.2.2 Summary

In this chapter, we described an abstract machine to coordinate asynchronous web service interactions. We defined the operational semantics of a system specified using our Document Flow Model notation. A transition rule to execute an asynchronous message was given based on the specification functions, generating data from the system definition, and the semantic functions, evaluating data generated from the specification. We also discussed the possible operational solutions for the dynamic behaviours, including adding a service, removing a service and replacing a service.

The formal operational semantics given in this chapter helps us to understand the behaviours of a service-oriented system. It could be used to develop tools to simulate applications composed by asynchronous web services, and also could be extended into real service oriented system implementations.

Chapter 7

Conclusion and Future Work

In the early part of this thesis, we declared our research motivation and introduced a new formal notation: Document Flow Model to model the asynchronous web service composition. We illustrated the use of the notation and compared it with current established industrial specifications. We also gave a formal operational semantics for the DFM notation. In this chapter, we review the contribution of this research. We also evaluate our work and summarise the advantages of our work over other related works. The potential improvements and future plans are also briefly discussed in the end.

7.1 Research Contribution

This work addressed the issue of architecture for modern service-oriented enterprise systems. Previously reported architecture specifications have their limitations. Some specifications, such as BPEL4WS [2], use a centralised orchestration framework to compose distributed web services. But a centralised system architecture sometimes encounters server problems such as scalability, availability [47], and dynamism. Some specifications, such as WSCI [20] and WSCAF [23], employ a peer-to-peer distributed service coordination framework. But they fail to provide an executable model.

In this thesis, we introduced a peer-to-peer distributed service composition framework: an enterprise system is composed of distributed peer web services; these web services are stateless and are unaware of the business interaction states; business interaction states are maintained at separate persistent components; web services communicate with each other by asynchronous messages; a context mechanism coordinates the business interaction execution. The peer-to-peer architecture in our framework largely improves the system availability because it has no centralised components controlling the execution of business interactions. In comparison with the stateful BPEL4WS business process, the stateless feature in our model makes web services much easier to be dynamically replaced. The context coordination

mechanism also complements the WSCI and WSCAF with an executable business interaction model.

This work investigated the challenging task of service-oriented enterprise system design. To achieve this objective, relevant works at all levels were reviewed and examined, which showed a clear gap between the required and available design tools. On one side, the industry developed comprehensive programming and specification languages for the implementation of service-oriented enterprise systems; on the other side, the academics used highly abstract mathematic models to reason and verify the system design. However, there is no suitable language for the business analyst to design a service-oriented enterprise system [45]. It is very difficult for a business analyst to design a system using specification languages or mathematical models which can only be accomplished by IT specialists. To fill this gap, we introduced a Document Flow Model (Chapter 3), which is powerful enough to capture service oriented system behaviour, whilst straightforward for business analyst to understand and design complex systems.

This work also addressed the issue of formal models. A formal model could be used to design, implement and reason about a service-oriented enterprise system. To define such a system in a formal model, different aspects of specific web service behaviours need to be considered. Firstly, message contents are important in a service-oriented formal model. The messages are not only responsible for passing parameters and results but also responsible for workflows. For example, a message correlation describes the execution sequence of two or more messages. Secondly, the loosely-coupled asynchronous communication pattern is essential to design a service-oriented system. Thirdly, long-running business interactions are basic and common scenario in real business environments, and have to be captured in formal models. Moreover, an enterprise system is subject to constant change in order to meet the dynamic changing business requirements and network environments. A formal model needs to support this kind of dynamic behaviour and allow reconfiguration.

We examined related service-oriented formal models in Section 2.4 and Section 2.5. Some models, such as BPEL4WS, WSCI and UML [48] [49], provide rich descriptions to specify the service composition behaviours, but fail to provide the capabilities of formal reasoning and model checking. The formal model using CCS [51] introduces a handshaking approach to reason the compatibility of interacting web services, but it only gives a synchronous communication model. Formal models using Petri Net [53] [54] [55] reason asynchronous

messages among composite web services, but can not describe all service composition behaviour such as contents of the messages. Some formal models [57] [46] [58] [59] [52] [43] aim to use tools to check and validate a service composition, but they are all based on a static service composition. Dynamic web service models of [61] [62] [64] [65] use QoS view to select web services at composition time, but do not provide a means to maintain the business interaction after a system reconfiguration.

In this thesis we proposed a novel formal model which is concise, and at the same time, expressive enough to describe the various service composition behaviours, such as message contents, asynchronous and long-running business interactions, dynamic configurations and so on. We also developed a simple operational semantics to enrich the proposed model with the capability of formal reasoning of the loosely-coupled asynchronous communications. Our formal model improved the above formal models with mechanisms to describe a broad range of service-oriented system behaviours and the capability of formal validation. It is evaluated in detail at Section 7.2.

As we have stated in the abstract, our work aims to bridge industrial specifications and abstract mathematic models. Industrial specifications like BPEL4WS and WSCI are comprehensive to specify service-oriented enterprise systems, but they do not provide the capability of formal reasoning. Mathematic models such as Petri Nets and model checkers such as SPIN provide strong capabilities of formal reasoning and model checking, but they are not expressive enough to describe common service composition behaviours. Moreover, both current industrial specifications and abstract mathematical models are too difficult for non IT experts to design an enterprise system.

In this thesis, we introduced a concise formal notation (DFM). It is descriptive enough to capture various service composition behaviours. Meanwhile it could be easily understood and used by non IT experts to design large scale enterprise systems. The formal operational semantics described in Chapter 6 completed the DFM model with the power of formal validation.

Designing and implementing a service-oriented system is challenging. The motivation of this work is to capture the behaviours of service-oriented system in formal models in order to facilitate service-oriented system implementation and verification.

In early part of this work we did various experiments to investigate the requirements for formally modelling service-oriented systems.

First of all, we developed an asynchronous travel agent system using Servlet and JSP [Appendix A] in order to investigate the asynchronous behaviours. We understood that a mechanism is required to maintain the interaction state consistently if components are loosely connected by the asynchronous interactions. Thus we implemented a service-oriented travel agent using SOAP messaging, simulated asynchronous communications over concurrent interactions [Appendix B].

To further understand the requirements for web service composition, a distributed BPEL4WS job submission system was developed using IBM BPWS4J [Appendix E]. We compared the BPEL4WS implementation with the previous two implementations, and examined the dynamic behaviours and interaction state handling capabilities in all three implementations in particular.

We have drawn the following conclusions based on these experiments and background studies summarised in Chapter 2.

- Web service interaction is a very important part of the web service composition. A service-to-service interaction is not just a transaction. Web service transactions are a subset of web service interactions [23]. Currently most service composition standards and specifications such as BEPL4WS only provide mechanisms to support long-running transactions by providing two-phase commit protocols and compensation activities [25]. However, an interaction may include multiple transactions, and can last much longer than a transaction. As long-running interactions are the basis of modern enterprise applications, they should also be considered when specifying service composition.
- We also believe that web service composition should be more dynamic. Storing the interaction state in a short-lived instance at the web container, as BPEL4WS does, means that web services can not be replaced in the middle of an interaction. However, in order to meet dynamically-changing business goals, web service applications are often required to be recomposable.

After the requirement analysis, we studied the formal programming languages XSLT [Appendix B], JavaScript [Appendix C], Haskell and so on. We extracted some features from these languages and developed our Document Flow Model with the following features.

- An XML-convertible notation which can be easily compiled to other XML-based standards and specifications. The document record data structure is created based on our experience with XML and its associated technologies, XSLT and JavaScript. A document record describes a tree data structure in a concise notation.
- Modelling asynchronous communications. Two kinds of communication patterns are supported, one-way communication, which amounts to a service receiving a message, and notification which amounts to a service sending a message. Request-response and solicit-response conversations are modelled as a one-way plus a notification communication. Any complex and structured asynchronous communication could be decomposed to simple communications using the two patterns [Section 4.2].
- Supporting long-running interactions and dynamic configurations. A coordination framework is used in DFM including: a context to identify an interaction state; a decentralised context propagation mechanism to structure interaction related data; a persistent component, a ContextStore, to maintain the interaction state.

In this thesis, we introduced a concise formal notation Document Flow Model (DFM) modelling the asynchronous web service composition. In Chapter 3 and Chapter 4, we presented the formal syntax and informal semantics of DFM in detail. A travel agent example, a nested job submission application and a warehouse purchase system illustrated the use of the notation. The formal models given by the DFM capture the general behaviours of web service-oriented system with the extensions on the dynamic configurations and the long-running business interactions.

A formal operational semantics has also been developed which describes the possible behaviours of a system of inter-related web services, in terms of the messages that can be exchanged during the execution of one or more business interactions, and the effect each message execution has on the business interaction states [Chapter 6].

Moreover, a formal verification of the DFM models using a formal model checking tool ARC [Appendix D] demonstrates the use of the DFM in formal method. A comparison of DFM with BPEL4WS [Chapter 5] shows the potential to apply DFM to industrial specifications. Thus the DFM successfully plays a role in connecting industrial specifications and real implementations.

7.2 Research Evaluation

This work aims to propose a new architecture and use this architecture to design new service-oriented enterprise systems. For this purpose, we developed a new notation, Document Flow Model, which was validated through comparison, utility, formalisation, and partial implementations. Some possible extension and improvement of this work are presented in the future work [Section 7.3].

To evaluate the proposed DFM model in terms of describing service composition behaviours, we applied the DFM model to a number of non-trivial applications [Chapter 3] [Chapter 4] [Chapter 5]. The examples demonstrated the following observations.

- A business interaction is not just a business transaction. It normally includes multiple transactions and runs for days or weeks. The travel agent example in Chapter 3 defines a travel booking interaction which is composed of three transactions, a user query transaction, a flight shop reply transaction and a hotel shop reply transaction. The coordination framework allows this business interaction to be executed consistently over a long period.
- Web service communications are asynchronous. Basic workflow patterns [31] and different composition of synchronous and asynchronous communication patterns described in the examples have been successfully modelled with our simple yet effective one-way communication pattern and notification communication pattern in Section 4.2.
- The document record data structure that we invented in our DFM notation models the XML based messages in a concise way. The notation complies with the XML based web service standards, but is much easier to learn and use it to design a service-oriented distributed system. For example, a warehouse purchasing system

specification is given within two pages using the document record data structure [Chapter 5], but takes more than ten pages using BPEL4WS [2].

- A recomposable job submission example showed that our DFM model allows web services to be dynamically replaced while some business interactions are still running. It effectively solved plug-and-play design issue in the grid applications. Thus the size a system could be modified flexibly based on the availabilities of network resources.

We compared our DFM models with other works [Chapter 5] and examined other formal models [Section 2.4] [Section 2.5]. The comparison and examination showed the advantages of our work against the others [Section 7.1].

- BPEL4WS and WSCI are rich service composition specifications. They require complex middlewares and platforms to run the system. Our DFM provides a lightweight framework to coordinate distributed business interactions. It could be easily integrated into various distributed systems. Meanwhile, the peer-to-peer composition architecture largely releases the dependency on the availability and performance of the control component (a business process), hence improves the system scalability and concurrency.
- BPEL4WS engine [10] stores a business process state as an instance at the web containers. The web containers keep live connections between web services and the business process state. Thus a web service can not be replaced when the business process is still running. In our DFM framework, business interactions are maintained at separate persistent components - ContextStores. A web service can recover the business interaction state from the persistent components and continue to execute the interaction that has not been finished by the replaced web service. Thus a system can replace a web service without worrying about the completion of all business interactions. Our DFM model is more dynamic than a BPEL4WS model.
- Formal models that have been discussed in Chapter 2 provide means of formal reasoning and model checking. But due to some limitations of different languages, they can not describe some common service composition behaviours that have been captured in our model.

- Specifically, Petri Net models [53] [54] [55] ignore the contents in all the messages and assume a message is a constant signature moving from one node (web service) to another. These models may be good in reasoning the workflow patterns, but they can not reason about the completion and state of business interactions. In our DFM model, we introduced a nested document record data structure to include the message contents, and a context coordination mechanism to facilitate business interaction execution over asynchronous communication patterns.
- The formal models of [61] [62] [64] [65] intend to model dynamic web service compositions. These models introduce QoS parameters and use them to select web services at the composition time. They do not consider the fact that service-oriented system is subject to constant change, a service could be changed in the middle of a business interaction (or a business process). Our model coordinates business interactions and allows a new service to continue executing the business interactions that are owned by retired web services.

We also validated our modelling language by providing its operational meanings [Chapter 6]. The novel operational semantics [40] described a service-oriented system environment where web services are loosely connected by asynchronous messages. It also supports dynamic service configuration behaviour.

- The formal operational semantics provides a run time environment for service-oriented enterprise systems. It captures the loosely coupled connectivity by separating the concerns of message executions and message deliveries. Our novel operational semantics improves other service composition models mentioned in this thesis by providing a way to reason about the loosely coupled web service compositions.

Finally, we validated our work by implementations. All implementations [Appendices] proved that our DFM model can play an important role in connecting formal models with industrial specifications.

- By compiling our DFM model into the formal model checking tool, ARC [Appendix D], a service-oriented enterprise system could be formally validated by complete (finite) state searches.
- The JavaScript message pool [Appendix C] is a partial implementation of our formal operational semantics. It is used to validate distributed systems composed of loosely-coupled asynchronous messages. The tool simulates the message flows as well as message contents and interaction states. The experiment demonstrated that our DFM is capable of not only specifying a service-oriented enterprise system, but also providing the capability of formal validation.

7.3 Future Work

As we just indicated in the last section, in order to reach the above research results we did quite a lot of implementations, most of which have been documented in the appendix. Although the experiments successfully served the purpose of the research, there are a number of ways they could be improved and expanded further. Moreover, during the course of the work, several relevant challenging research ideas have been raised, and many supporting implementations and tools have been developed. Given these, we outline the main directions of the future works.

7.3.1 Potential Improvement on the Operational Semantics

The operational semantics of DFM describes the possible behaviours of a system of inter-related web services, in terms of the messages executions. The operational semantics has the potential to improve on the following aspects.

1. The current operational semantics is expressive enough to describe the asynchronous web service invocations. To help understand the message execution rule, we have described a virtual message pool as a message daemon; configuration tables for each web service to map the service configuration to the system configuration; and the relationships between the interactive web services. The formal representations of those descriptions could be developed to enrich the current operational semantics.

2. We have briefly discussed the dynamic configuration scenario in the Section 6.2 of Chapter 6. With the potential improvement on 1, we should be able to develop new operational semantic rules to specify the dynamic behaviours such as *add a service*, *remove a service*, *replace a service* and so on.

7.3.2 A Simulation Tool

A service oriented system is composed by independent services loosely connected by the Internet. This architecture decided that the system behaviour sometimes depends on the Internet and the supporting platforms of each web service. Those may bring unpredictable behaviours to the systems that are difficult to identify using formal model checking tools.

Our operational semantics is based on the idea that web services interact with each other using a virtual message pool, which messages can be exchanged during the execution of one or more service interactions. The system state transit is represented by messages instead of activities which well conforms to the Internet environment.

A simulation tool based on our operational semantics would help to understand the real-world practical scenario. It could be used to identify the unpredictable system behaviours among loosely connected web services. The improvement on 7.3.1 could also be integrated into the simulation tool to reason the dynamic behaviours in a service-oriented system.

7.3.3 BPEL4WS Formal Verification

BPEL4WS has been commonly accepted as a super complicated specification. Implementing a BPEL4WS application requires tremendous efforts and patience, so does formally verifying the implementation. The BPEL4WS is still in its development. But we can imagine that when the BPEL4WS is widely used to implement commercial systems, there is no doubt there will be huge requirements for formal verification.

We have compared our DFM and BPEL4WS in Chapter 5. The DFM is able to specify a complete business process which has been given by the BPEL4WS specification. As the DFM is a very concise notation, it would be much easier to develop a verification tool. We have also demonstrated that DFM could be compiled into other already existing formal modelling checking tool [Appendix D]. However, the DFM is a short period research outcome and only

captures parts of the BPEL4WS features. To be fully compatible with BPEL4WS, several features need to be enriched, such as transaction support, exception handling, and message definition and so on.

Appendices

The appendices in this thesis are evidence of experiments and implementations that we did in contribution to this work. They are also the resources for future researchers. The summary of each appendix is as follows.

Appendix A An Investigation of Asynchronous Invocations Using Servlet and JSP

Service-oriented enterprise systems must be asynchronous. We started the work by investigating the asynchronous behaviours in web applications. Then we developed a travel agent system using Servlet/JSP and simulated the asynchronous interactions [Appendix A]. This experiment helps us understand the underlying mechanism of Java web component interfaces, Java web application servers. It also helps us examine the feasibilities of using them to implement asynchronous application protocols. The evidence [Appendix A] shows that our asynchronous interactions using Servlet/JSP APIs are synchronised by Java web containers. Therefore, a mechanism is required to maintain the interaction state consistently if components are loosely connected by the asynchronous interactions.

Appendix B A Travel Agent Implementation Using XSLT and SOAP

We improved our Servlet/JSP travel agent implementation using SOAP and XSLT [Appendix B]. SOAP is a standard protocol to exchange XML messages over distributed web services. XSLT is mainly used to separate information content (XML) from presentation on the Web (HTML). But it is also recognised as a high-level declarative programming language. In this experiment, we transformed an XML message and an XML DB into an XML output and a new XML DB using a simple XSLT function. Studies also showed that although functional programming languages provide a simple and mathematically tractable computational model, they do not provide a good engine to handle concurrent operations.

This experiment contributes to our work in two aspects. Firstly, it helps us summarise the requirements in modelling service-oriented enterprise systems [Section 2.6.1] by experimenting with the standard web service interaction protocol, SOAP. Secondly, it incorporates the functional programming language into web applications.

Appendix C A JavaScript DFM Model

JavaScript is a lightweight programming language initially used as a script embedded in web browsers. Syntactically, the core JavaScript language resembles C, C++, and Java, with programming constructs. Although JavaScript gives an object definition, it is not an object-oriented programming language. It accepts un-typed parameters and variables which is an important feature in modelling language.

In this experiment, we are interested in the object literal, functions syntax and event model in the JavaScript. The experiment investigated the capability of modelling XML data structure using JavaScript object model, and created our Document Record data structure [Section 3.2.6]. The tool that we built using JavaScript simulated the loosely-coupled operational environment described in Chapter 6.

Appendix D Formal Verification of DFM using ARC

ARC is a formal modelling tool. It provides means to model service-oriented distributed architectures and autonomous web service behaviours. It also facilitates the function to search complete (finite) state space in order to validate a distributed system.

Our DFM travel agent model has been formally validated using ARC [Appendix D]. The experiment demonstrated that our DFM model could be further compiled into formal model checking tools. Thus, it successfully served a role in connecting real implementations and formal models.

Appendix E A BPEL4WS Implementation using IBM BPWS4J

To further understand the requirements of web service composition, a distributed BPEL4WS job submission system was developed using IBM BPWS4J [Appendix E]. We compared the BPEL4WS implementation with the previous two implementations [Appendix A] [Appendix B], and examined the dynamic behaviours and interaction state handling capabilities in all three implementations in particular.

This implementation demonstrated our understanding on competitive specifications. It also helped us summarise the web service composition requirement [Section 2.6.1] and led to our service composition approach [Section 2.6.2].

Appendix A

An Investigation of Asynchronous Invocations Using Servlet and JSP

Having stated the motivations of our research at the beginning of the thesis, we started the work by investigating the asynchronous behaviours in web applications. We built a real application intending to understand underlying mechanism of Java web component interfaces, Java web application servers and to study the feasibilities of using them to implement asynchronous application protocols.

We implemented two Travel Agent solutions, a synchronous solution and an asynchronous solution, using pure JSP and http transport. The two solutions produced the same result, but after we debugged (traced) the communication in the asynchronous solution, we found that the Java web application server synchronised our asynchronous method invocations. The detail of this experiment can be found in my 3-month research report. Here we only attached part of report, including: APIs we used, two different solutions and analysis results.

Servlet and JSP

Servlet and JSP are technologies defined by Sun Microsystems to create dynamic web content. They provide a component-based, platform-independent method for building web-based applications and have access to entire Java APIs families that include a library of HTTP-specific calls. JSPs are html documents interleaving with Java. They are extensions of Servlet, when a request is mapped to a JSP page, the JSP engine translates the JSP page into a Servlet.

There are different ways to invoke a remote object or a web service, Socket / RMI and http. In this experiment, we use *URLConnection* object to setup http transport [34, 35].

- `URLConnection con = RequestURL.openConnection()`
- `BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()))`

The abstract class *URLConnection* is the superclass of all classes that represent a communications link between the application and a URL. Instances of this class can be used both to read from and to write to the resource referenced by the URL. Request parameters are defined in *RequestURL* object. Response is read from buffer reader. Contents of response are got by parsing input stream.

A Synchronous Solution

The synchronous version of shop and agent uses the normal way of JSP design pattern that service method gets parameters from a request object and sends data back to client through *PrintWriter* by the corresponding response object. The service requester initialises an *URLConnection* of the service provider and read response back by get *InputStream* of that *URLConnection*. The synchronous Travel Agent system sequence diagram is as below. Diagram shows that JSP instances complete their lifecycle after they send responses back to service requesters.

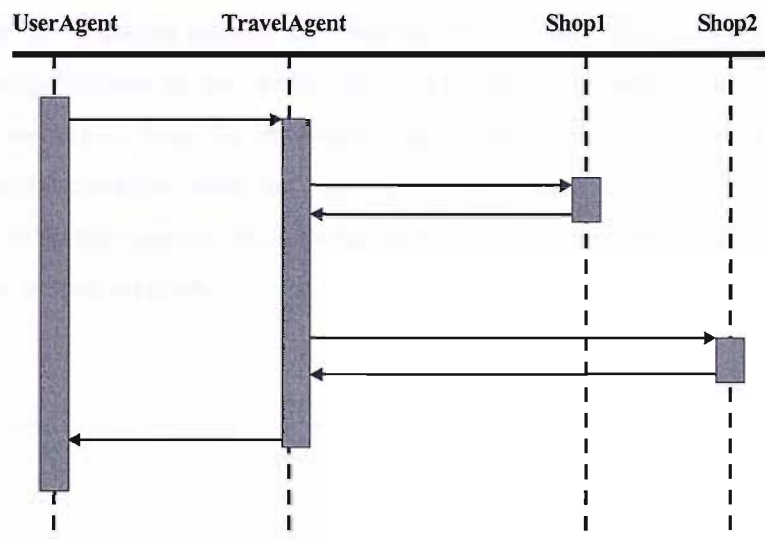


Figure A-1 Synchronous Travel Agent Communication

An Asynchronous Solution

Instead of sending reply back by the response object, our components create a new *URLConnection* with the reply information to the service requester. A random time delay can be set for each component to emulate asynchronous behaviours. In this way, system setup asynchronous communications between components by *URLConnections*.

Experiments are done by sending two orders to two JSPs at different web servers simultaneously and tracing the communication sequence on the web server monitor window. Outputs show that orders were processed in the right sequence that we defined in the prototype and replies were delivered back to correct web server without any confusion. Two processes are handled overlapped by the web server. The second order could start to be processed before the first order complete.

This asynchronous behaviour is supported by the JSP engine that provided by web servers. When a web server receives a request for a Servlet, it forwards it to an instance of that Servlet. This forwarding will create a request object and a response object and pass them as parameters to the service method of that instance. We described in previous part that each Servlet instance has a unique request and response objects pair. The response object has the access to the output stream of the client. The HTML that comprises replies is written to the output stream associated with the response object. After the service method has finished running, the Servlet container sends the contents of the response object back to the web server, which in turn sends the response back to the web browser who submitted the request in the first place. The service methods of each Servlet are called by the servlet container on a per-request basis.

In our asynchronous solution, a service provider sends reply back to the service requester by initialising a new Servlet request. This means that Servlet container creates a new service method call to that Servlet instance. For example, when web server gets a Travel Agent Servlet request from User Agent, it creates a new service method call of the Travel Agent, call-1. This method then invokes shop1. Shop1 creates a reply to Travel Agent by sending a Travel Agent Servlet request. Now web server create another service method call of Travel Agent, call-2, to handle request from shop1. And so forth, there will be three service method calls of the Travel Agent and two of the User Agent in a whole business process. A business process starts from the User Agent call-1 whose response object has the access to the web

browser, and ends with User Agent call-2. How can the second User Agent call access the web browser which is kept by the first User Agent call becomes the key to evaluate our asynchronous design pattern.

Debug shows the following system sequence diagram.

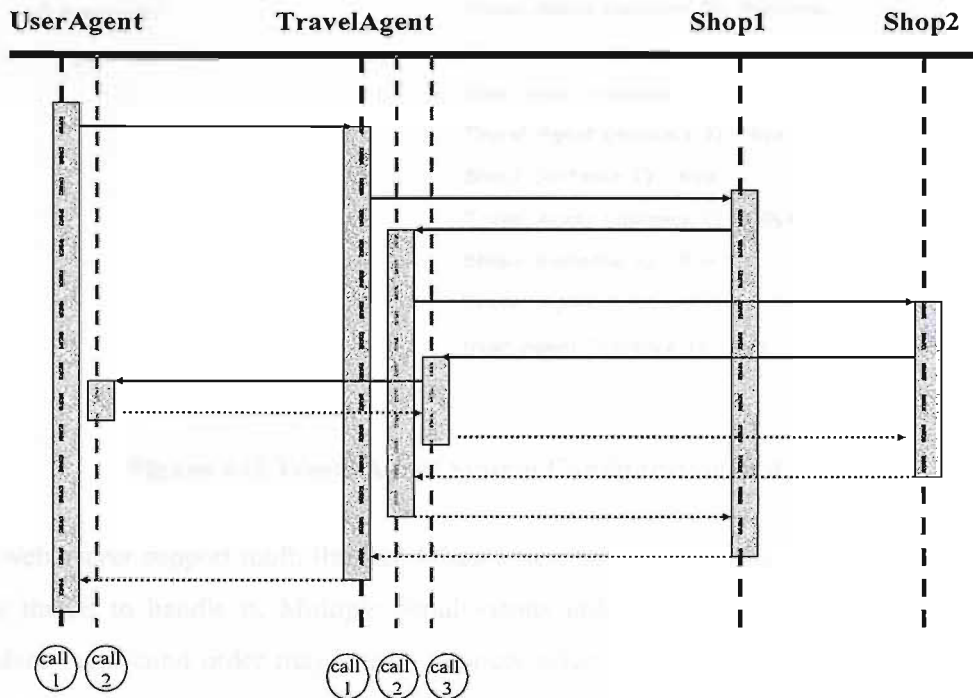


Figure A-2 Asynchronous Travel Agent Communication

Different with the synchronous solution, when each component sends reply by initialising a new Servlet request the service method call does not complete until it gets the response back of that request and passes it back to the request component who previously start this process.

Here are some traces of our Travel Agent system.


```
• Configuration
Shop s1, s2;
TravelAgent a1;
UserAgent c0, c1;
a1.shop1=s1;
a1.shop2=s2;
c0.agent=a1;
c1.agent=a1;

• Log
User Agent (instance 1) -- Welcome
Travel Agent (instance 1) - Welcome
Shop1 (instance 1) -Welcome
Travel Agent (instance 2) - Welcome
Shop2 (instance 1) - Welcome
Travel Agent (instance 3) -Welcome
User Agent (instance 2) - Welcome
User Agent (instance 2) - Bye
Travel Agent (instance 3) - Bye
Shop2 (instance 1) - Bye
Travel Agent (instance 2) -- Bye
Shop1 (instance 1) - Bye
Travel Agent (instance 1) - Bye
User Agent (instance 1) - Bye
```

Figure A-3 Travel Agent System Configuration and Trace

Most web server support multi-threads. When a new Servlet request comes, web server starts a new thread to handle it. Multiple simultaneous orders are handled by separate threads. Therefore the second order may start to process before first order complete. However, how many orders can be processed concurrently will depend on the capabilities of different web servers.

The asynchronous business process is synchronised by the web server in some ways, because the response to JSP has to be on the same thread of the request. Therefore integrating business logic inside JSP may not be the best way to model asynchronous business process.

Conclusion

From the experiment, we learned that Servlet and JSP are the technologies designed for HTTP access. HTTP is basically a kind of synchronous transport protocol. The Java web server synchronised the method invocation on the Servlets. Therefore, to implement the asynchronous method invocation, the application has to separate the business logic from the web component, Servlet or JSP.

Although the method invocations are synchronised by the web server in some way, the business logic interactions are still in an asynchronous way (because of the random delay we implemented at the business logics). It helped us understand the asynchronous business interaction behaviours.

Appendix B

A Travel Agent Implementation Using XSLT and SOAP

We implemented a travel agent application using XSLT and SOAP messaging. In the early phase of this research, we have attempted to use functional programming language in web application to ensure the correctness and eliminate the side effect. We document some of our research result in this section.

Functional Programming Language and XSLT

Functional programming is based on the idea that program and data are unified by represent both as functions. This unification provides for a simple and mathematically tractable computational model.

XSLT, which stands for eXtensible Stylesheet Language: Transformations, is a language defined by the World Wide Web Consortium (W3C). XSLT has its origins in the aspiration to separate information content from presentation on the Web. Now it is being recognised as a high-level declarative programming language. In practises, more and more XSLT examples demonstrate the capabilities of using XSLT as a functional programming language. XSLT is used to manipulate the data structure in XML that is in the same way of the declarative language SQL does in a relational database. Currently most e-commerce systems use XSLT transmitting data between applications. The transmission extracts and combines data from one set of XML document to generate another set of XML document that application used to compute [27].

Functional programming languages contribute greatly on data structure operation. However they do not provide a good engine to handle concurrent operations. Lots of works are working on concurrency handling in functional programming. Most of them are still limited on synchronous communication or single thread systems.

In our experiment, each service has a XSLT file which we called as a function. The function takes a combination of the whole XML database and the query as input and produces a whole new database and the reply message. The following shows the travel agent database and agent function.

Agent DB XML

```

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

= <agentdb>

= <transaction>
  <num>1</num>
</transaction>

= <transaction>
  <num>2</num>

=   <order>
      <num />
      <shop>FlightService</shop>
=     <item>
          <name>BA038</name>
          <quantity>1</quantity>
=     </item>
      <item>
          <name>BA039</name>
          <quantity>1</quantity>
      </item>
    </order>
=   <order>
      <num />
      <shop>HotelService</shop>
=     <item>
          <name>double</name>
          <quantity>1</quantity>
=     </item>
      <item>
          <name>standard</name>
          <quantity>1</quantity>
      </item>
    </order>

=   <orderreply>
      <num />
      <shop>HotelService</shop>
=     <item>
          <name>double</name>
          <status>ok</status>
=     </item>
      <item>
          <name>standard</name>
          <status>ok</status>

```

```

        </item>
    </orderreply>
=   <orderreply>
        <num />
        <shop>FlightService</shop>
=   <item>
            <name>BA038</name>
            <status>ok</status>
        </item>
=   <item>
            <name>BA039</name>
            <status>ok</status>
        </item>
    </orderreply>

</transaction>

= <agentdeployment>
=   <service>
        <name>HotelService</name>
        <url>http://localhost:8080/soap/servlet/rpcrouter</url>
        <shop>HotelService</shop>
    </service>
=   <service>
        <name>Agency1</name>
        <url>http://localhost:8080/soap/servlet/rpcrouter</url>
        <shop>FlightService</shop>
        <shop>CarService</shop>
    </service>

</agentdeployment>

</agentdb>

```

Agent Function XSLT

```

<xsl:transform version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="shopname" match="userorder/order" use="shop" />
<xsl:template match="/" >
    <xsl:variable name="allchildren"><xsl:apply-templates mode="childnodes" /></xsl:variable>
    <xsl:if test="contains($allchildren,'userorder')">
        <xsl:apply-templates mode="orderdb" />
    </xsl:if>
    <xsl:if test="contains($allchildren,'shopreply')">
        <xsl:apply-templates mode="replydb" />
    </xsl:if>

```

```

    <xsl:if test="contains($allchildren,'messages')">
      <xsl:apply-templates mode="agentreplydb" />
    </xsl:if>

  </xsl:template>

  <xsl:template match="root" mode="replydb" >

    <xsl:variable name="repnum" select="shopreply/num" />
    <xsl:variable name="repshop" select="shopreply/shop" />
    <xsl:variable name="repitems"> <xsl:copy-of select="shopreply/item" /> </xsl:variable>

    <xsl:variable name="prenum">

      <xsl:for-each select="agentdb/transaction">
        <xsl:variable name="trann" select="num" />
        <xsl:if test="$repnum=$trann" >
          <xsl:value-of select="order/num" />
        </xsl:if>
      </xsl:for-each>

    </xsl:variable>

    <xsl:variable name="replies">

      <xsl:for-each select="agentdb/transaction">
        <xsl:variable name="trann" select="num" />
        <xsl:if test="($repnum=$trann)" >
          <xsl:value-of select="count(orderreply)" />
        </xsl:if>
      </xsl:for-each>

    </xsl:variable>

    <xsl:variable name="orders">

      <xsl:for-each select="agentdb/transaction">
        <xsl:variable name="trann" select="num" />
        <xsl:if test="($repnum=$trann)" >
          <xsl:value-of select="count(order)" />
        </xsl:if>
      </xsl:for-each>

    </xsl:variable>

  </root>

  <agentdb>

    <xsl:for-each select="agentdb/transaction">

      <xsl:variable name="tranno" select="num" />

      <xsl:if test="($repnum=$tranno)" >

        <transaction><num><xsl:value-of select="num" /></num>

```

```

        <xsl:copy-of select="order" />
        <xsl:copy-of select="orderreply" />

        <orderreply>
            <num><xsl:value-of select="$prenum" /></num>
            <shop><xsl:value-of select="$repshop" /></shop>
            <xsl:copy-of select="$repitems" />
        </orderreply>

    </transaction>

</xsl:if>

<xsl:if test="not($repnum=$tranno)" >

    <transaction>
        <num><xsl:value-of select="num" /></num>
        <xsl:copy-of select="order" />
        <xsl:copy-of select="orderreply" />
    </transaction>
</xsl:if>

</xsl:for-each>

<xsl:copy-of select="agentdb/agentdeployment" />

</agentdb>

<messages>

    <xsl:if test="((($orders)-($replies)=1)" >

        <xsl:for-each select="agentdb/transaction">
            <xsl:variable name="trannum" select="num" />
            <xsl:if test="($repnum=$trannum)" >

                <agentreply>
                    <num><xsl:value-of select="$prenum" /></num>
                    <xsl:copy-of select="orderreply" />
                    <orderreply>
                        <num><xsl:copy-of select="$prenum" /></num>
                        <shop><xsl:value-of select="$repshop" /></shop>
                        <xsl:copy-of select="$repitems" />
                    </orderreply>
                </agentreply>

            </xsl:if>
        </xsl:for-each>

    </xsl:if>

</messages>

</root>

</xsl:template>

```

```

<xsl:template match="root" mode="agentreplydb" >

  <xsl:variable name="repnum" select="messages/agentreply/num" />

  <xsl:variable name="prenum">

    <xsl:for-each select="agentdb/transaction">
      <xsl:variable name="trann" select="num" />
      <xsl:if test="$repnum=$trann" >
        <xsl:value-of select="order/num" />
      </xsl:if>
    </xsl:for-each>

  </xsl:variable>

  <xsl:variable name="replies">

    <xsl:for-each select="agentdb/transaction">

      <xsl:variable name="trann" select="num" />
      <xsl:if test="($repnum=$trann)" >
        <xsl:value-of select="count(orderreply)" />
      </xsl:if>

    </xsl:for-each>

  </xsl:variable>

  <xsl:variable name="orders">

    <xsl:for-each select="agentdb/transaction">

      <xsl:variable name="trann" select="num" />
      <xsl:if test="($repnum=$trann)" >
        <xsl:value-of select="count(order)" />
      </xsl:if>

    </xsl:for-each>

  </xsl:variable>

  <xsl:variable name="agentreplies">
    <xsl:value-of select="count(messages/agentreply/orderreply)" />
  </xsl:variable>

  <xsl:variable name="repliesinagent">

    <xsl:for-each select="messages/agentreply/orderreply" >

      <orderreply>
        <num><xsl:value-of select="$prenum" /></num>
        <xsl:copy-of select="shop" />
        <xsl:copy-of select="item" />
      </orderreply>

    </xsl:for-each>

```



```

</xsl:variable>

<root>

  <agentdb>

    <xsl:for-each select="agentdb/transaction">

      <xsl:variable name="tranno" select="num" />
      <xsl:if test="($repnum=$tranno)" >

        <transaction>
          <num><xsl:value-of select="num" /></num>
          <xsl:copy-of select="order" /><xsl:copy-of select="orderreply" />
          <xsl:copy-of select="$replysinagent" />
        </transaction>

      </xsl:if>

      <xsl:if test="not($repnum=$tranno)" >

        <transaction>
          <num><xsl:value-of select="num" /></num>
          <xsl:copy-of select="order" /> <xsl:copy-of select="orderreply" />
        </transaction>

      </xsl:if>

    </xsl:for-each>
    <xsl:copy-of select="agentdb/agentdeployment" />

  </agentdb>

  <messages>

    <xsl:if test="(($orders)-($replies)=($agentreplies))" >

      <xsl:for-each select="agentdb/transaction">

        <xsl:variable name="trannum" select="num" />
        <xsl:if test="($repnum=$trannum)" >
          <agentreply>
            <num><xsl:value-of select="$prenum" /></num>
            <xsl:copy-of select="orderreply" />
            <xsl:copy-of select="$replysinagent" />
          </agentreply>
        </xsl:if>

      </xsl:for-each>

    </xsl:if>

  </messages>

</root>

</xsl:template>

```

```

<xsl:template match="root" mode="orderdb" >

  <xsl:variable name="maxno">

    <xsl:for-each select="agentdb/transaction/num">
      <xsl:sort data-type="number" order="descending"/>
      <xsl:if test="(position())=1">
        <xsl:value-of select="."/>
      </xsl:if>
    </xsl:for-each>

  </xsl:variable>

  <xsl:variable name="orderno" ><xsl:value-of select="$maxno+1" /> </xsl:variable>

  <root>

  <agentdb>

    <xsl:copy-of select="agentdb/transaction" />
    <transaction>
      <num><xsl:value-of select="$orderno" /></num>
      <xsl:copy-of select="userorder/order" />
    </transaction>
    <xsl:copy-of select="agentdb/agentdeployment" />

  </agentdb>

  <messages>

    <xsl:for-each select="agentdb/agentdeployment/service">

      <orders>

        <xsl:variable name="agentname" select="name" />
        <shop><xsl:value-of select="$agentname" /></shop>
        <url><xsl:value-of select="url" /></url>
        <xsl:variable name="pname" select="shop" />

        <payload>

          <xsl:if test="not ($agentname=$pname)">
            <userorder>

              <xsl:for-each select="shop" >
                <xsl:variable name="spname"><xsl:value-of select="."/></xsl:variable>
                <xsl:for-each select="key('shopname', $spname)" >
                  <order>
                    <num><xsl:value-of select="$orderno" /></num>
                    <xsl:copy-of select="shop" /><xsl:copy-of select="item" />
                  </order>
                </xsl:for-each>
              </xsl:for-each>
            </userorder>
          </xsl:if>

```

```

    <xsl:if test="$agentname=$pname">
        <xsl:for-each select="shop" >
            <xsl:variable name="spname"><xsl:value-of select="."/></xsl:variable>
            <xsl:for-each select="key('shopname', $spname)" >
                <order>
                    <num><xsl:value-of select="$orderno" /></num>
                    <xsl:copy-of select="shop" /><xsl:copy-of select="item" />
                </order>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:if>
</payload>
</orders>
</xsl:for-each>
</messages>
</root>
</xsl:template>
<xsl:template match="root" mode="childnodes" >
    <xsl:for-each select="child::node()" >
        <xsl:if test="contains(name(self::node()),'userorder')">
            <xsl:value-of select="'userorder'" />
        </xsl:if>
        <xsl:if test="contains(name(self::node()),'shopreply')">
            <xsl:value-of select="'shopreply'" />
        </xsl:if>
        <xsl:if test="contains(name(self::node()),'messages')">
            <xsl:value-of select="'messages'" />
        </xsl:if>
    </xsl:for-each>
</xsl:template>
</xsl:transform>

```

Java SOAP APIs

We implemented a travel agent system using Apache SOAP v2.3 over Resin web server [36, 37]. The travel agent and shops are SOAP services which is the early version of Web Services. Services talked to services by SOAP RPC in the manner of Web Service Architecture.

SOAP APIs (UserAgent.jsp)

```
<%@ page import="java.util.*, java.net.*, org.apache.soap.*,
org.apache.soap.encoding.*, org.apache.soap.encoding.soapenc.*,
org.apache.soap.rpc.*, javax.xml.transform.sax.SAXSource,
org.xml.sax.InputSource, javax.xml.transform.stream.StreamResult,
org.w3c.dom.*, javax.xml.transform.stream.StreamSource, java.io.*,
org.w3c.dom.Document, javax.xml.transform.*, javax.xml.transform.Source" %>

<%
    // Build the call
    URL url=new URL(request.getParameter("rpcurl"));

    Call call=new Call();
    call.setSOAPMappingRegistry(new SOAPMappingRegistry());
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI(request.getParameter("service"));
    call.setMethodName(request.getParameter("method"));

    Vector params=new Vector();
    params.addElement(new Parameter("parameter", String.class,request.getParameter("orders"),null));
    call.setParams(params);

    Response resp=call.invoke(url,"");

    boolean status=resp.generatedFault();
    Parameter ret=resp.getReturnValue();
    Object value=ret.getValue();

    //transform reply to html
    File xsltFile = new File(request.getParameter("display"));
    InputSource is = new InputSource(new StringReader(value.toString()));
    Source xmlSource = new SAXSource(is);
    Source xsltSource = new StreamSource(xsltFile);

    try{

        TransformerFactory transFact = TransformerFactory.newInstance();
```

```
        Transformer trans = transFact.newTransformer(xsltSource);
        trans.transform(xmlSource, new StreamResult(out));

    } catch(TransformerException e){

        e.printStackTrace();

    }

    %>
```

Appendix C

A JavaScript DFM Model

Why JavaScript?

JavaScript is a lightweight, interpreted programming language with object-oriented capabilities. The general-purpose of the language has been embedded in web browsers [28]. However, we are interested in the object literal, functions syntax and event model in the JavaScript. Syntactically, the core JavaScript language resembles C, C++, and Java, with programming constructs. Although JavaScript gives an object definition, it is not an object-oriented programming language.

Our experiments on JavaScript have two aims.

- Investigating the capability of modelling XML data structure using JavaScript object model.
- Build a tool to simulate or check our DFM model.

JavaScript Object Syntax

In JavaScript, an object is a collection of named values. These named values are usually referred to as properties of the object. An object literal syntax allows you to create an object and specify its properties. It consists of a comma-separated property/ value pairs, all enclosed within curly braces.

For example: `{x:2, y:8}`.

And Object literals can also be nested.

For example: `{x:2, y:{xx:2, yy:7}}`.

JavaScript Functions

“Function definition and invocation are syntactic features of JavaScript and of most other programming languages. However, in JavaScript, functions are not only syntax but also data, which means that they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on.” The function syntax allows our XSLT function model to cooperate with the JavaScript object syntax.

Event-Driven Programming Model

JavaScript provides support to event-driven programming models. Event handlers have been written as strings of JavaScript code that are used as the values of certain HTML attributes, such as `onClick`.

```
<input type="button" value="click here" onClick="alert('thanks');">
```

A JavaScript DFM Tool

In service-oriented applications, the actions (operations) are triggered by an incoming message. Therefore, we experimented to implement a JavaScript tool that simulates an incoming message event by an `onClick` event handler and model the XML data by JavaScript Object Syntax.

The tool is showed as the following figure. It successfully modelled the XML data structure and the event model. But it is week in simulating concurrent interactions. And the most important, it could not offer functions, like space searching and automatic executions. Therefore, we did not do further experiment on that.

The experiment is successful because we learned a lot on the JavaScript object syntax and create our document record data based on that.

The following figure is the GUI our JavaScript DFM model.

DFM in JavaScript

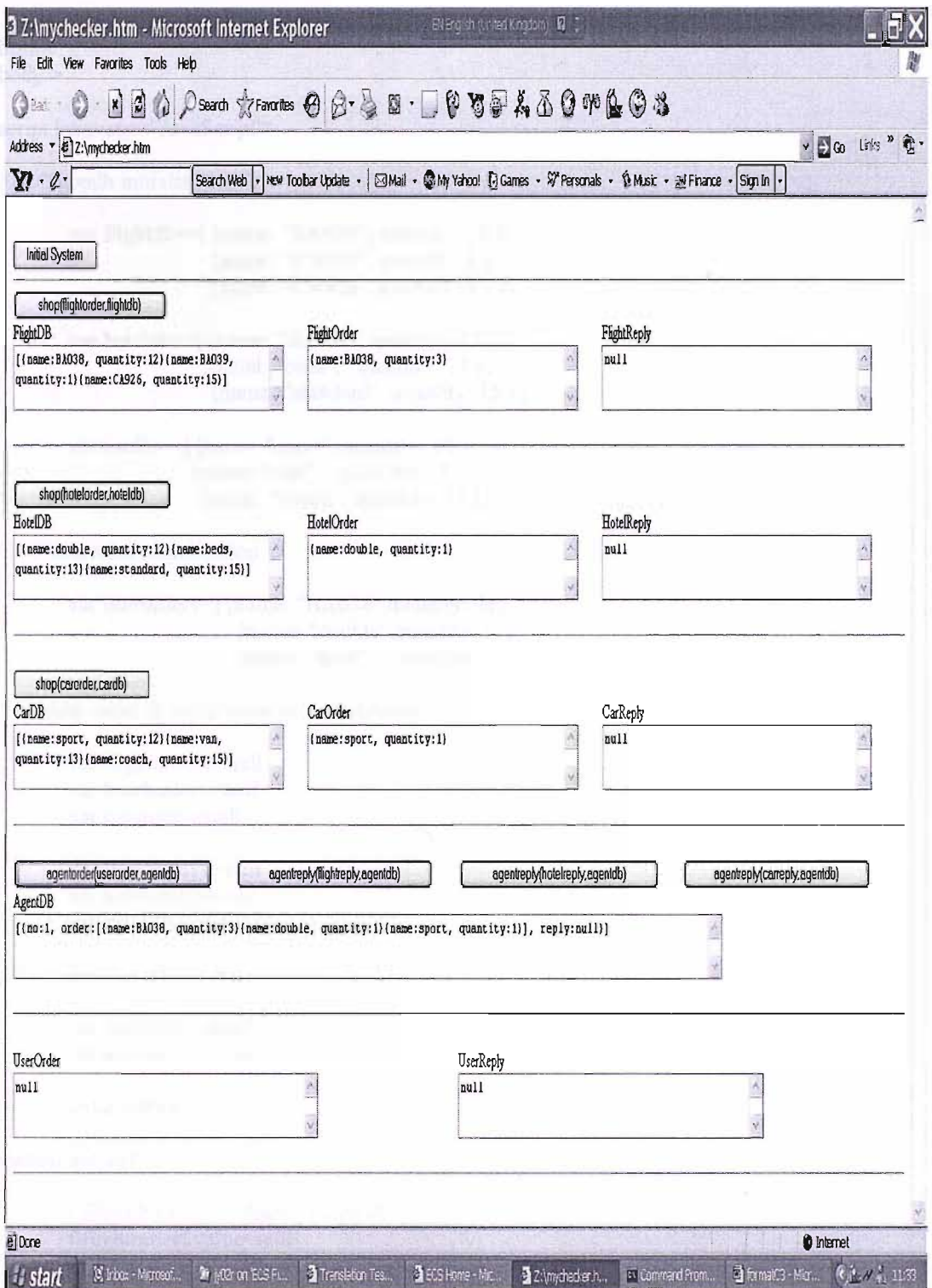


Figure C-1 A JavaScript DFM Checker

DFM in JavaScript

```
<html>
<body >

<script language="JavaScript">

//////////shopdb initialization

    var flightdbv=[{name: "BA038", quantity: 12 },
                   {name: "BA039", quantity: 1 },
                   {name: "CA926", quantity: 15 }]

    var hoteldbv=[{name: "double", quantity: 12 },
                  {name: "beds", quantity: 13 },
                  {name: "standard", quantity: 15 }]

    var cardbv= [{name: "sport", quantity: 12 },
                 {name: "van", quantity: 13 },
                 {name: "coach", quantity: 15 }]

//////////userorder initialization

    var userorderf=[{name: "BA038",quantity: 3},
                    {name: "double",quantity: 1},
                    {name: "sport", quantity: 1}]

//////////shop order & reply variable initialization

    var flightorderf=null
    var hotelorderf=null
    var carorderf=null

    var flightreplyf=null
    var hotelreplyf=null
    var carreplyf=null

////////// agent initialization

    var agentdbv=[null]
    var agentreplyv=[null]

////////// initialization

function initial(f){

    f.flightdbf.value = display(flightdbv)
    f.flightorderf.value=null
    f.flightreplyf.value=null

    f.hoteldbf.value = display(hoteldbv)
    f.hotelorderf.value=null
    f.hotelreplyf.value=null

    f.cardbf.value = display(cardbv)
```

```
f.carorderf.value=null
f.carreplyf.value=null

f.userorderf.value = display(userorderv)
f.userreplyf.value = null

f.agentdbf.value=displayagent(agentdbv)
}
```

```
function display(o){
  if(o==null){
    var returnstring=null
  }else{
    if(o.length>=1){
      if(o[0]==null){
        var returnstring="null"
      }else{
        var returnstring="["
        for( var i=0; i< o.length; i++)
          returnstring=returnstring+" {name:"+o[i].name+",
          quantity:"+o[i].quantity+"}"
        returnstring=returnstring+"]"
      }else{
        var returnstring=" {name:"+o.name+", quantity:"+o.quantity+"}"
      }
    }
    return returnstring
  }
}
```

```
function displayreply(o){
  if(o==null){
    var returnstring=null
  }else{
    if(o.length>=1){
      if(o[0]==null){
        var returnstring="null"
      }else{
        var returnstring="["
        for( var i=0; i< o.length; i++)
          returnstring=returnstring+" {name:"+o[i].name+",
          result:"+o[i].result+"}"
        returnstring=returnstring+"]"
      }else{
        var returnstring=" {name:"+o.name+", result:"+o.result+"}"
      }
    }
    return returnstring
  }
}
```

```
}

```

```
function displayagent(o){
    if(o.length>=1){
        if(o[0]==null){
            var returnstring="[null]"
        }else{
            var returnstring="["
            for( var i=0; i< o.length; i++){
                returnstring=returnstring+"{no:"+o[i].no+",
                order:"+display(o[i].order)+" ,reply:"+displayreply(o[i].reply)+"}"
            }
            returnstring=returnstring+"]"
        }
    }else{
        returnstring=null
    }
    return returnstring
}

```

```
function fagentorder(f){
    flightorderv=userorderv[0]
    f.flightorderf.value=display(flightorderv)

    hotelorderv=userorderv[1]
    f.hotelorderf.value=display(userorderv[1])

    carorderv=userorderv[2]
    f.carorderf.value=display(userorderv[2])

    f.userorderf.value=null

    agentdbv=[{no:1, order:userorderv, reply:null}]
    f.agentdbf.value=displayagent(agentdbv)
    userorderv=null
}

```

```
function shop(db,order){
    var newdb = db
    var reply = null

    for( var i=0; i< db.length; i++){
        if(db[i].name==order.name){
            if(db[i].quantity>=order.quantity){

```

```
        reply={name:order.name, result:"OK"}
        newdb[i]={name:db[i].name, quantity:(db[i].quantity-order.quantity)}
    } else{
        reply={name:order.name, result:"NOTOK"}
    }
    } else {
        newdb[i]=db[i]
    }
}
return {db:newdb, reply:reply}
}
```

```
function shopf(f){
    var temp=shop(flightdbv,flightorderv)
    flightdbv=temp.db
    flightreplyv=temp.reply
    flightorderv=null
    f.flightreplyf.value=displayreply(flightreplyv)
    f.flightdbf.value=display(flightdbv)
    f.flightorderf.value=null
}
```

```
function shoph(f){
    var temp=shop(hoteldbv,hotelorderv)
    hoteldbv=temp.db
    hotelreplyv=temp.reply
    hotelorderv=null
    f.hotelreplyf.value=displayreply(hotelreplyv)
    f.hoteldbf.value=display(hoteldbv)
    f.hotelorderf.value=null
}
```

```
function shopc(f){
    var temp=shop(cardbv,carorderv)
    cardbv=temp.db
    carreplyv=temp.reply
    carorderv=null
    f.carreplyf.value=displayreply(carreplyv)
    f.cardbf.value=display(cardbv)
    f.carorderf.value=null
}
```

```
function agentreply(shopreply){
```

```
    if (agentdbv[0].reply==null){
        agentdbv[0].reply=[shopreply]
    }else{
        agentdbv[0].reply[agentdbv[0].reply.length]=shopreply
    }
    if (agentdbv[0].reply.length==agentdbv[0].order.length){
        return "DONE"
    }else{
        return "NOTYET"
    }
}

function agentreplyf(f){
    if(f.flightreplyf.value!="null"){
        if(agentreply(flightreplyv)=="DONE"){
            agentreplyv=agentdbv[0].reply
            f.userreplyf.value=displayreply(agentreplyv)
        }
        f.flightreplyf.value=null
        f.agentdbf.value=displayagent(agentdbv)
    }
}

function agentreplyh(f){
    if(f.hotelreplyf.value!="null"){
        if(agentreply(hotelreplyv)=="DONE"){
            agentreplyv=agentdbv[0].reply
            f.userreplyf.value=displayreply(agentreplyv)
        }
        f.hotelreplyf.value=null
        f.agentdbf.value=displayagent(agentdbv)
    }
}

function agentreplyc(f){
    if(f.carreplyf.value != "null"){
        if(agentreply(carreplyv)=="DONE"){
            agentreplyv=agentdbv[0].reply
```

```

        f.userreplyf.value=displayreply(agentreplyv)
    }

    f.carreplyf.value=null
    f.agentdbf.value=displayagent(agentdbv)
}
}
</script>

<form name="agentform" >

<br><input type="button" value="Initial System" onclick="initial(this.form)"/>
<hr>
<table border="0" cellpadding="0" width="100%" id="AutoNumber1">

<tr>
<td width="100%">
<input type="button" value="shop(flightorder,flightdb)" onclick="shopf(this.form)"/></td>
</tr>

<tr>
<td width="33%">FlightDB</td>
<td width="33%">FlightOrder</td>
<td width="34%">FlightReply</td>
</tr>

<tr>
<td width="33%"><textarea name="flightdbf" rows=3 cols=44</textarea> </td>
<td width="33%"><textarea name="flightorderf" rows=3 cols=44</textarea></td>
<td width="34%"><textarea name="flightreplyf" rows=3 cols=44</textarea></td>
</tr>
</table>

<hr>
<table border="0" bordercolor="#111111" width="100%" id="AutoNumber1">

<tr>
<td width="100%" colspan="3">
<input type="button" value="shop(hotelorder,hoteldb)" onclick="shoph(this.form)"/></td>
</tr>

<tr>
<td width="33%">HotelDB</td>
<td width="33%">HotelOrder</td>
<td width="34%">HotelReply</td>
</tr>

<tr>
<td width="33%"><textarea name="hoteldbf" rows=3 cols=44</textarea> </td>
<td width="33%"><textarea name="hotelorderf" rows=3 cols=44</textarea> </td>
<td width="34%"><textarea name="hotelreplyf" rows=3 cols=44</textarea> </td>
</tr>
</table>

```

```
<p style="margin-top: 0; margin-bottom: 0">&nbsp;</p>
```

```
<hr>
```

```
<p style="margin-top: 0; margin-bottom: 0">&nbsp;</p>
```

```
<table border="0" bordercolor="#111111" width="100%" id="AutoNumber1">
```

```
<tr>
```

```
<td width="100%" colspan="3">
```

```
<input type="button" value="shop(carorder,cardb)" onclick="shopc(this.form)"/></td>
```

```
</tr>
```

```
<tr>
```

```
<td width="33%">CarDB</td>
```

```
<td width="33%">CarOrder</td>
```

```
<td width="34%">CarReply</td>
```

```
</tr>
```

```
<tr>
```

```
<td width="33%"><textarea name="cardbf" rows=3 cols=44</textarea> </td>
```

```
<td width="33%"><textarea name="carorderf" rows=3 cols=44</textarea> </td>
```

```
<td width="34%"><textarea name="carreplyf" rows=3 cols=44</textarea> </td>
```

```
</tr>
```

```
</table>
```

```
<p style="margin-top: 0; margin-bottom: 0">&nbsp;</p>
```

```
<hr>
```

```
<table border="0" bordercolor="#111111" width="100%" id="AutoNumber2">
```

```
<tr>
```

```
<td width="25%">
```

```
<input type="button" value="agentorder(userorder,agentdb)"
onclick="fagentorder(this.form)"/>
```

```
</td>
```

```
<td width="25%">
```

```
<input type="button" value="agentreply(flightreply,agentdb)"
onclick="agentreplyf(this.form)"/>
```

```
</td>
```

```
<td width="25%">
```

```
<input type="button" value="agentreply(hotelreply,agentdb)"
onclick="agentreplyh(this.form)"/>
```

```
</td>
```

```
<td width="25%"> <input type="button" value="agentreply(carreply,agentdb)"
onclick="agentreplyc(this.form)"/>
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td width="25%">AgentDB</td>
```

```
<td width="25%">&nbsp;</td>
```

```
<td width="25%">&nbsp;</td>
```

```
<td width="25%">&nbsp;</td>
```

```
</tr>
```

```
<tr>
```

```
<td><textarea name="agentdbf" rows=3 cols=120></textarea></td>
</tr>

</table>

<p style="margin-top: 0; margin-bottom: 0"> </p>

<hr>
<table border="0" bordercolor="#111111" width="100%" id="AutoNumber3">

  <tr>
    <td width="50%">UserOrder </td>
    <td width="50%">UserReply </td>
  </tr>

  <tr>
    <td width="50%"><textarea name="userorderf" rows=3 cols=50></textarea></td>
    <td width="50%"><textarea name="userreplyf" rows=3 cols=50></textarea></td>
  </tr>

</table>

  <p style="margin-top: 0; margin-bottom: 0">&nbsp;</p>

<hr>
</form>

</body></html>
```


Appendix D

Formal Verification of DFM using ARC

To establish our DFM, we formally verify our model by including complete (finite) state space search, using the formal model checking tool, ARC [38].

What is ARC?

“ARC is a software architecture modelling tool which provides the means to model systems of objects and processes. It is intended for modelling and validating distributed systems, service-based architectures and autonomous behaviour. Models, which follow a familiar state-transition paradigm, are coded in Java for the purposes of animation and model checking. A translator from ARC to Spin is under development” [38].

ARC models are coded in Java, as a collection of condition/action rules. The ARC implementation then allows the models to be animated using a simple interactive interface.

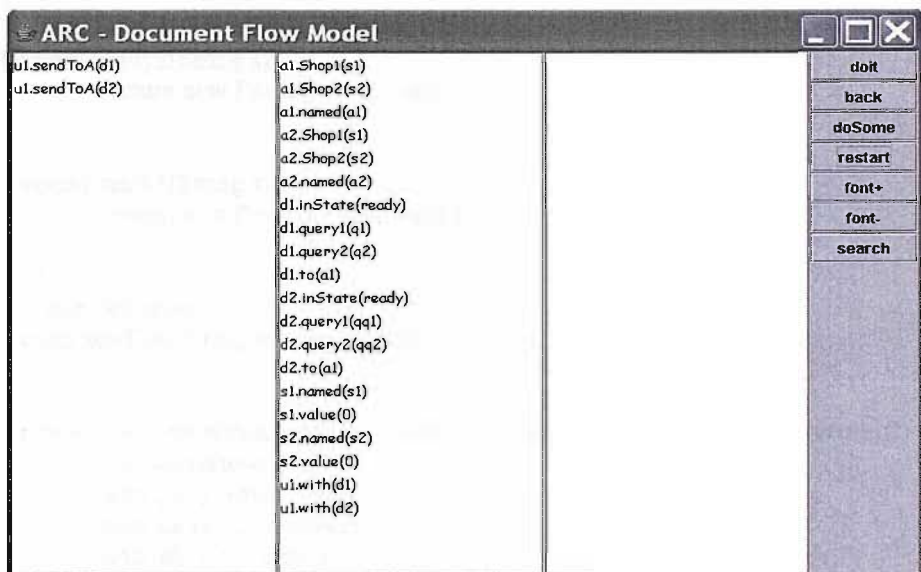


Figure D-1 ARC GUI

DFM ARC Model

DFN.java

```
import arc.*;

public class DFN extends Model {

    // document class
    class FlowDocument extends Entity{

        // properties definitions
        Property inState(String state){
            return new Property(this,"inState",state);
        }

        Property agent(String s){
            return new Property(this,"to",s);
        }

        Property from(String s){
            return new Property(this,"from",s);
        }

        Property query2(String s){
            return new Property(this,"query2",s);
        }

        Property query1(String s){
            return new Property(this,"query1",s);
        }

        Property reply2(String s){
            return new Property(this,"reply2",s);
        }

        Property reply1(String s){
            return new Property(this,"reply1",s);
        }

        // action definition
        Action newDoc(String tostring, String fromstring, String querysting1,String
            querysting2){

            return new Action(this,"newDoc",tostring, fromstring, querysting1, querysting2)
                .add(agent(tostring))
                .add(query1(querysting1))
                .add(query2(querysting2))
                .add(inState("ready"));
        }

        // methods definition
        Method addReply1(String reply){
```

```

        return new Method()
            .add(reply1(reply));
    }
    Method addReply2(String reply){

        return new Method()
            .add(reply2(reply));
    }
}

// shop class
class Shop extends Entity{

    Property inState(String state){
        return new Property(this,"inState",state);
    }

    Property named(String s){
        return new Property(this,"named",s);
    }

    Property value(Integer v){
        return new Property(this,"value",v);
    }

    Method onMessage(String s){
        return new Method();
    }

    Action newShop(String s){

        return new Action(this,"newShop",s)
            .add(named(s))
            .add(value(new Integer(0)));
    }

    Action shopping1(FlowDocument fd){

        return new Action(this,"shopping1",fd)
            .with(fd.inState("withs1")).with(named("s1"))
            .rem(fd.inState("withs1"))
            .with(value(new Integer(0))).rem(value(new Integer(0))).add(value(new
                Integer(1)))
            .add(fd.reply1("s1reply1"));
    }

    Action shopping11(FlowDocument fd){

        return new Action(this,"shopping11",fd)
            .with(fd.inState("withs1")).with(named("s1"))
            .rem(fd.inState("withs1"))
            .with(value(new Integer(1))).rem(value(new Integer(1))).add(value(new
                Integer(2)))
            .add(fd.reply1("s1reply2"));
    }
}

```

```

Action shopping2(FlowDocument fd){

    return new Action(this,"shopping2",fd)
        .with(fd.inState("withs2")).with(named("s2"))
        .rem(fd.inState("withs2"))
        .with(value(new Integer(0))).rem(value(new Integer(0))).add(value(new
                                                                 Integer(1)))
        .add(fd.reply2("s2reply1"));

}

Action shopping21(FlowDocument fd){

    return new Action(this,"shopping21",fd)
        .with(fd.inState("withs2")).with(named("s2"))
        .rem(fd.inState("withs2"))
        .with(value(new Integer(1))).rem(value(new Integer(1))).add(value(new
                                                                 Integer(2)))
        .add(fd.reply2("s2reply2"));

}

}

// agent class
class Agent extends Entity{

    Property inState(String state){
        return new Property(this,"inState",state);
    }

    Property named(String s){
        return new Property(this,"named",s);
    }

    Property wait(FlowDocument fd){
        return new Property(this,"wait",fd);
    }

    Property Shop1(Shop shop){
        return new Property(this,"Shop1",shop);
    }

    Property Shop2(Shop shop){
        return new Property(this,"Shop2",shop);
    }

    Method onMessage(String s){
        return new Method();
    }

    Action newAgent(Shop s1,Shop s2,String ss){

```

```

        return new Action(this,"newAgent")
        .add(Shop1(s1))
        .add(Shop2(s2))
        .add(named(ss));
    }

    Action sendToS(FlowDocument fd){

        return new Action(this,"sendToS",fd)
        .with(fd.inState("withA"))
        .with(fd.agent("a1")).with(named("a1"))
        .rem(fd.inState("withA")).add(fd.inState("withs1")).add(fd.inState("withs2"))
        .add(wait(fd));

    }

    Action sendToS1(FlowDocument fd){

        return new Action(this,"sendToS1",fd)
        .with(fd.inState("withA"))
        .with(fd.agent("a2")).with(named("a2"))
        .rem(fd.inState("withA")).add(fd.inState("withs1")).add(fd.inState("withs2"))
        .add(wait(fd));

    }

    Action sendReply(FlowDocument fd){

        return new Action(this,"sendReply",fd)
        .with(wait(fd))
        .without(fd.inState("withs1")).without(fd.inState("withs2"))
        .rem(wait(fd))
        .add(fd.inState("AgentDone"));

    }
}

// user class
class User extends Entity{

    Property inState(String state){
        return new Property(this,"inState",state);
    }

    Property named(String s){
        return new Property(this,"named",s);
    }

    Property wait(FlowDocument fd){
        return new Property(this,"wait",fd);
    }

    Property with(FlowDocument fd){
        return new Property(this,"with",fd);
    }
}

```

```

Action newUser(FlowDocument fd){
    return new Action(this,"newUser")
        .add(with(fd));
}

Action sendToA(FlowDocument fd){

    return new Action(this,"sendToA",fd)
        .with(fd.inState("ready"))
        .rem(fd.inState("ready")).add(fd.inState("withA"))
        .rem(with(fd)).add(wait(fd));
}

Action checkReply(FlowDocument fd){

    return new Action(this,"checkReply",fd)
        .with(fd.inState("AgentDone")).with(wait(fd))
        .rem(wait(fd)).rem(fd.inState("AgentDone"))
        .add(fd.inState("Done"));
}
}

DFN(){

    Shop s1=new Shop(); s1.setName("s1");

    Shop s2=new Shop(); s2.setName("s2");

    Agent a1=new Agent(); a1.setName("a1");

    Agent a2=new Agent(); a2.setName("a2");

    User u1=new User(); u1.setName("u1");

    FlowDocument d1=new FlowDocument(); d1.setName("d1");

    FlowDocument d2=new FlowDocument(); d2.setName("d2");

    doAction(s1.newShop("s1"));
    doAction(s2.newShop("s2"));

    doAction(a1.newAgent(s1,s2,"a1"));
    doAction(a2.newAgent(s1,s2,"a2"));

    doAction(u1.newUser(d1));
    doAction(u1.newUser(d2));
    doAction(d1.newDoc("a1","u1","q1","q2"));
    //doAction(d2.newDoc("a2","u1","qq1","qq2"));
    doAction(d2.newDoc("a1","u1","qq1","qq2"));

    putAction(u1.sendToA(d1));
    putAction(u1.checkReply(d1));
    putAction(u1.sendToA(d2));
    putAction(u1.checkReply(d2));
}

```

```
    putAction(a1.sendToS(d1));
    putAction(a1.sendToS1(d1));
    putAction(a1.sendReply(d1));

    putAction(a1.sendToS(d2));
    putAction(a1.sendToS1(d2));
    putAction(a1.sendReply(d2));

    putAction(a2.sendToS(d1));
    putAction(a2.sendToS1(d1));
    putAction(a2.sendReply(d1));
    putAction(a2.sendToS(d2));
    putAction(a2.sendToS1(d2));
    putAction(a2.sendReply(d2));

    putAction(s1.shopping1(d1));
    putAction(s1.shopping2(d1));
    putAction(s1.shopping1(d2));
    putAction(s1.shopping2(d2));
    putAction(s1.shopping11(d1));
    putAction(s1.shopping21(d1));
    putAction(s1.shopping11(d2));
    putAction(s1.shopping21(d2));

    putAction(s2.shopping1(d1));
    putAction(s2.shopping2(d1));
    putAction(s2.shopping1(d2));
    putAction(s2.shopping2(d2));
    putAction(s2.shopping11(d1));
    putAction(s2.shopping21(d1));
    putAction(s2.shopping11(d2));
    putAction(s2.shopping21(d2));
}

public static void main(String args[]) {

    new ArcGUI("Document Flow Model",new DFN());

}
}
```

DFM ARC Model Verification

The following figure shows the verification result of our DFM model by random path searching (doSome).

ARC - Document Flow Model			
a1.sendReply(d2)	a1.Shop1(s1)	u1.sendToA(d2)	doit
	a1.Shop2(s2)	u1.sendToA(d1)	back
	a1.named(a1)	a1.sendToS(d1)	doSome
	a1.wait(d2)	s1.shopping1(d1)	restart
	a2.Shop1(s1)	s2.shopping2(d1)	font+
	a2.Shop2(s2)	a1.sendReply(d1)	font-
	a2.named(a2)	a1.sendToS(d2)	search
	d1.inState(Done)	s1.shopping1(d2)	
	d1.query1(q1)	u1.checkReply(d1)	
	d1.query2(q2)	s2.shopping2(d2)	
	d1.reply1(s1reply1)		
	d1.reply2(s2reply1)		
	d1.to(a1)		
	d2.query1(qq1)		
	d2.query2(qq2)		
	d2.reply1(s1reply2)		
	d2.reply2(s2reply2)		
	d2.to(a1)		
	s1.named(s1)		
	s1.value(2)		
	s2.named(s2)		
	s2.value(2)		
	u1.wait(d2)		

Figure D-2 ARC Verification Result

Appendix E

A BEPL4WS Implementation using IBM BPWS4J

BPWS4J

IBM Business Process Execution Language for Web Services Java Runtime provides a platform upon which business processes written using BPEL4WS may execute. This version supports the BPEL4WS v1.1 (May 2003) specification [10].

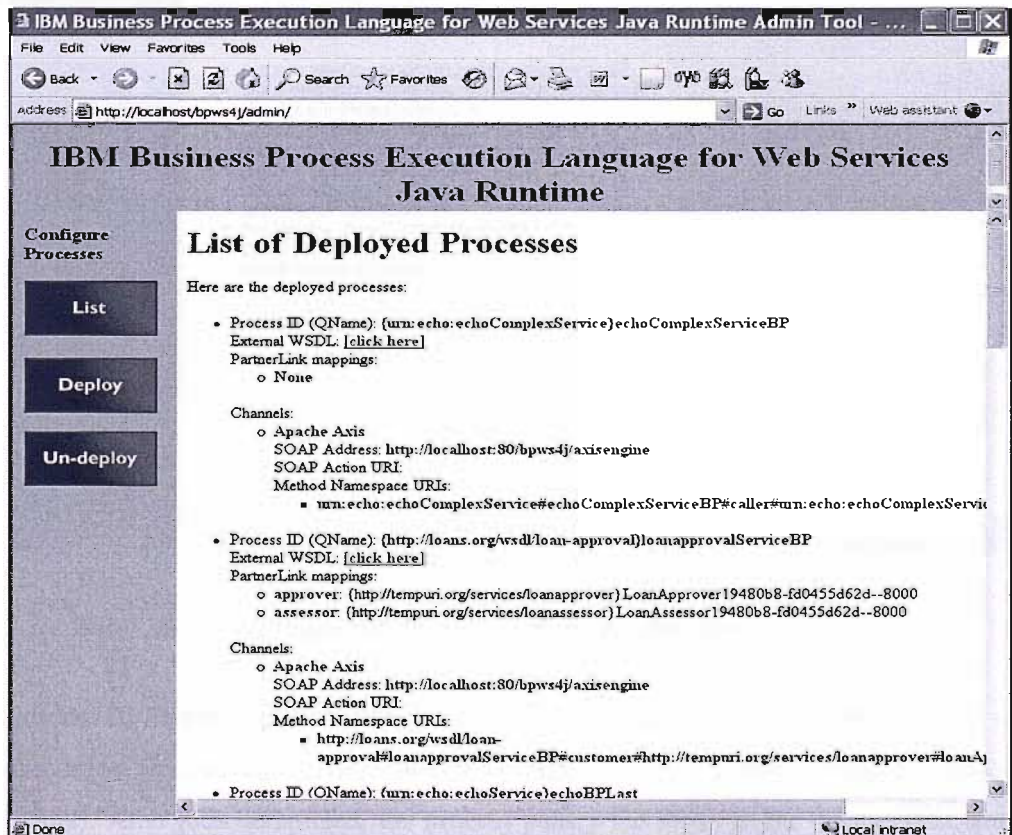


Figure E-1 A BPWS4J GUI

For each process, the engine takes in a BPEL4WS document which describes the process, a WSDL document (without binding information) which describes the interface that the process will present to clients, and WSDL documents (with binding information) which describe the services that the process may/will invoke during its execution. After deployment the process will be made available to outside consumers through a SOAP interface.

We installed the engine on Apache Tomcat under Windows XP. We deployed the following process.

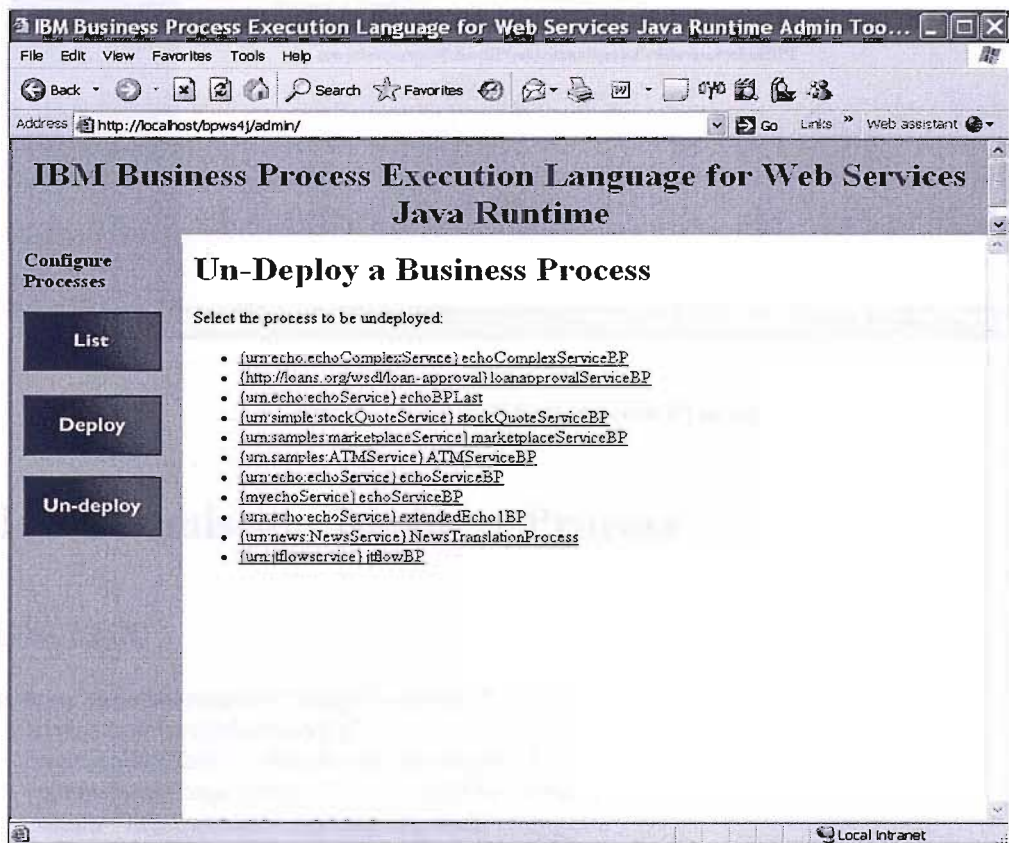


Figure E-2 Deployed Business Processes

To study the BPEL4WS specification and compare our model with it, we developed a simple job submission process. The process takes a job request and forwards them to two jobServices (it partners). After jobs completed, the process passes the result to the printservice, and then reply the job requester.

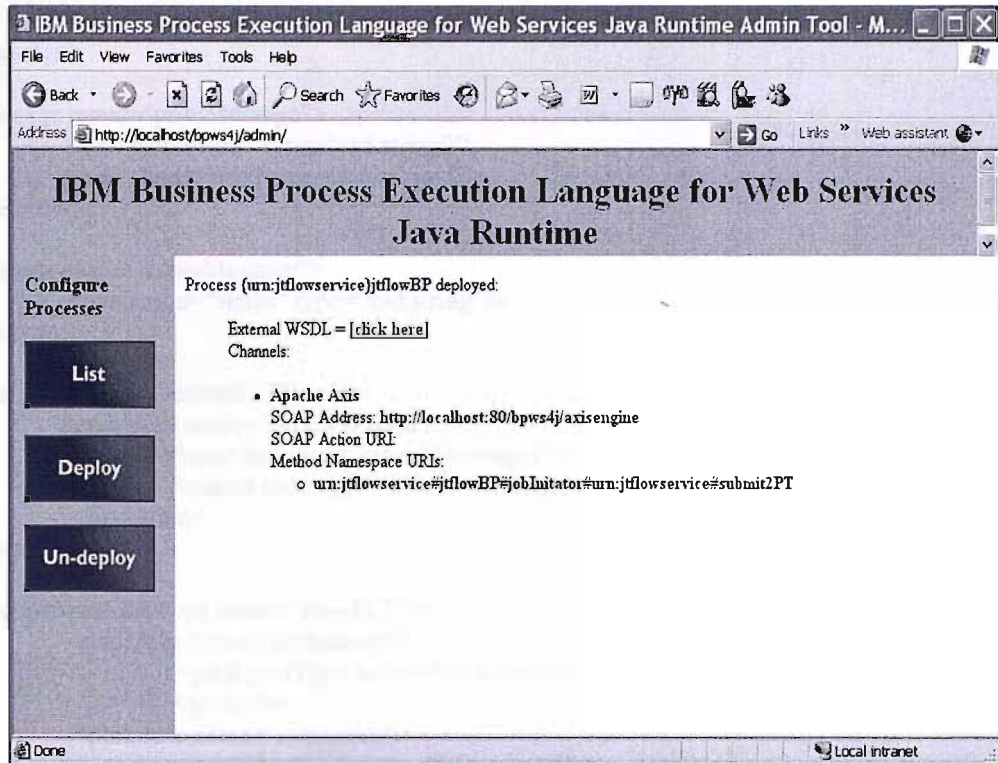


Figure E-3 Our Job Submission Process

A Job Submission Business Process

Job Flow WSDL

```

<definitions targetNamespace="urn:jtflowservice"
  xmlns:tns="urn:jtflowservice"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:job1ns="urn:jobService1"
  xmlns:job2ns="urn:jobService2"
  xmlns:job3ns="urn:jobService3">

  <message name="jobMessage">
    <part name="dir" type="xsd:string"/>
    <part name="job" type="xsd:string"/>
  </message>

  <message name="jobMessage1">
    <part name="dir1" type="xsd:string"/>
    <part name="job1" type="xsd:string"/>
    <part name="dir2" type="xsd:string"/>
  </message>

```

```

        <part name="job2" type="xsd:string"/>
    </message>

    <message name="printMessage">
        <part name="dir" type="xsd:string"/>
        <part name="file" type="xsd:string"/>
    </message>

    <message name="replyMessage">
        <part name="status" type="xsd:string"/>
    </message>

    <portType name="submit2PT">
        <operation name="submittingJob2" >
            <input message="tns:jobMessage1"/>
            <output message="tns:replyMessage"/>
        </operation>
    </portType>

    <plnk:partnerLinkType name="flowPLT">
        <plnk:role name="jobInitator">
            <plnk:portType name="tns:submit2PT"/>
        </plnk:role>
        <plnk:role name="jobPrinter">
            <plnk:portType name="tns:printPT"/>
        </plnk:role>
    </plnk:partnerLinkType>

    <service name="jtflowBP"/>
</definitions>

```

Job Flow BPEL

```

<process name="jtflow"
    targetNamespace="urn:flowProcess"
    xmlns:tns="urn:flowProcess"
    xmlns:lns="urn:jtflowservice"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:job1ns="urn:jobService1"
    xmlns:job2ns="urn:jobService2"
    xmlns:job3ns="urn:jobService3"
>
<variables>
    <variable name="jobs" messageType="lns:jobMessage1"/>
    <variable name="job1" messageType="lns:jobMessage"/>
    <variable name="job2" messageType="lns:jobMessage"/>
    <variable name="printvar" messageType="lns:printMessage"/>
    <variable name="reply" messageType="lns:replyMessage"/>
    <variable name="reply1" messageType="lns:replyMessage"/>
    <variable name="reply2" messageType="lns:replyMessage"/>

```

```

</variables>

<partnerLinks>
  <partnerLink name="jobInitator" partnerLinkType="Ins:flowPLT" />
  <partnerLink name="jobExecutor1" partnerLinkType="Ins:flowPLT" />
  <partnerLink name="jobExecutor2" partnerLinkType="Ins:flowPLT" />
  <partnerLink name="jobPrinter" partnerLinkType="Ins:flowPLT" />
</partnerLinks>

<sequence>

  <receive partnerLink="jobInitator" portType="Ins:submit2PT"
    operation="submittingJob2" variable="jobs"
    createInstance="yes" name="jobReceive"/>

  <assign>
    <copy>
      <from variable="jobs" part="dir1" /><to variable="job1" part="dir" />
    </copy>
    <copy>
      <from variable="jobs" part="job1" /><to variable="job1" part="job" />
    </copy>
  </assign>

  <assign>
    <copy>
      <from variable="jobs" part="dir2" /><to variable="job2" part="dir" />
    </copy>
    <copy>
      <from variable="jobs" part="job2" /><to variable="job2" part="job" />
    </copy>
  </assign>

  <flow>
    <invoke name="executejob1"
      partnerLink="jobExecutor1"
      portType="job2ns:jobPT"
      operation="submittingJob"
      inputVariable="job1"
      outputVariable="reply1">
    </invoke>

    <invoke name="executejob2"
      partnerLink="jobExecutor2"
      portType="job3ns:jobPT"
      operation="submittingJob"
      inputVariable="job2"
      outputVariable="reply2">
    </invoke>
  </flow>

  <assign>
    <copy>
      <from variable="jobs" part="dir1" />

```

```

                <to variable="printvar" part="dir" />
            </copy>
            <copy>
                <from variable="jobs" part="job1" />
                <to variable="printvar" part="file" />
            </copy>
        </assign>

        <invoke name="printjob"
            partnerLink="jobPrinter"
            portType="job1ns:jobPT"
            operation="printing"
            inputVariable="printvar" >
        </invoke>

        <assign>
            <copy><from expression=""OK"" /><to variable="reply" part="statu" /></copy>
        </assign>

        <reply partnerLink="jobInitator"
            portType="lns:submit2PT"
            operation="submittingJob2"
            variable="reply"
            name="jobReply"/>

    </sequence>

</process>

```

A Job Service WSDL

```

<definitions targetNamespace="urn:jobService1" xmlns:tns="urn:jobService1"
    xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
    xmlns:ejb="http://schemas.xmlsoap.org/wsdl/"
    xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/">

    <message name="jobMessage">
        <part name="dir" type="xsd:string"/>
        <part name="job" type="xsd:string"/>
    </message>

    <message name="jobMessage1">
        <part name="dir1" type="xsd:string"/>
        <part name="job1" type="xsd:string"/>
        <part name="dir2" type="xsd:string"/>
        <part name="job2" type="xsd:string"/>
    </message>

```

```

<message name="printMessage">
  <part name="dir" type="xsd:string"/>
  <part name="file" type="xsd:string"/>
</message>

<message name="replyMessage">
  <part name="status" type="xsd:string"/>
</message>

<portType name="jobPT">
  <operation name="submittingJob" >
    <input message="tns:jobMessage"/>
    <output message="tns:replyMessage"/>
  </operation>

  <operation name="submittingJob2" >
    <input message="tns:jobMessage1"/>
    <output message="tns:replyMessage"/>
  </operation>

  <operation name="printing" >
    <input message="tns:printMessage"/>
  </operation>
</portType>

<binding name="JavaBinding" type="tns:jobPT">
  <java:binding />
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>

  <operation name="submittingJob2">
    <java:operation methodName="submit2Jobs" parameterOrder="dir1 job1 dir2 job2" />
  </operation>

  <operation name="submittingJob">
    <java:operation methodName="submitJob" parameterOrder="dir job" />
  </operation>

  <operation name="printing">
    <java:operation methodName="printJobs" parameterOrder="dir file" />
  </operation>
</binding>

<!-- The service name and the TNS represent my service ID QName -->
<service name="JobService">
  <port name="JavaPort" binding="tns:JavaBinding">
    <java:address className="jtjobservice1.JobService"/>
  </port>
</service>
</definitions>

```

A BPEL Client

```

import java.io.*; java.net.*; java.util.*;
import org.apache.soap.*; org.apache.soap.rpc.*;

public class JobOwner {

    private static void printUsageAndTerminate() {

        System.err.println("Usage: java " + JobOwner.class.getName() +
            " SOAP-router-URL dir1 job1 dir2 job2");
        System.exit(1);
    }

    public static void main(String[] args) throws Exception {

        if (args.length != 5) {
            printUsageAndTerminate();
        }

        // Process the arguments.
        URL url = new URL(args[0]);
        String dir1 = args[1];
        String job1 = args[2];
        String dir2 = args[3];
        String job2 = args[4];

        // Build the call.
        Call call = new Call();
        Vector params = new Vector();

        call.setTargetObjectURI("urn:jtflowservice#jtflowBP#jobInitator#urn:jtflowservice#submit2PT");

        params.addElement(new Parameter("dir1",String.class,dir1,null));
        params.addElement(new Parameter("job1",String.class,job1,null));
        params.addElement(new Parameter("dir2",String.class,dir2,null));
        params.addElement(new Parameter("job2",String.class,job2,null));

        call.setMethodName("submittingJob2");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        call.setParams(params);

        // make the call: note that the action URI is empty because the
        // XML-SOAP rpc router does not need this. This may change in the
        // future.
        Response resp = call.invoke(/* router URL */ url, /* actionURI */ "");

        // Check the response.
        if (resp.generatedFault()) {

            Fault fault = resp.getFault();
            System.out.println("Ouch, the call failed: ");
            System.out.println(" Fault Code = " + fault.getFaultCode());
        }
    }
}

```



```
        System.out.println(" Fault String = " + fault.getFaultString());
        System.out.println(" Fault      = " + fault);
    } else {
        Parameter result = resp.getReturnValue();
        if (result != null)
            System.out.println(result.getValue());
        else
            System.out.println("No response was returned. Perhaps there was an error with the flow.");
    }
}
}
```

Bibliography

- [1] Martin Bichler, Aire Segev, J. Leon Zhao, *Component-based E-Commerce: Assessment of Current Practices and Future Directions*, SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 27 No4, pp. 7-14, 1998.
- [2] Tony Andrews, Francisco Curbera, et al., *Business Process Execution Language for Web Services Version 1.1*, <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>, 05 May 2003.
- [3] Erik Christensen, Francisco Curbera, et al., *Web Services Description Language (WSDL) 1.1*, <http://www.w3.org/TR/wsdl>, 15 March 2001.
- [4] Steve Graham, Doug Davis, Simeon Simeonov, al., *Building Web Services with Java - Making sense of XML, SOAP, WSDL, and UDD Second Edition*, Developer's Library, 2005.
- [5] Peter Henderson, *Laws for Dynamic Systems*, presented at International Conference on Software Re-Use (ICSR 98), Victoria, Canada, June 1998, IEEE Computer Society Press June 1998, IEEE Computer Society Press, pp. 330-336, 1998.
- [6] Object Management Group™ (OMG™), <http://www.uml.org>.
- [7] Sun Microsystems, *Sun Java System Message Queue 3 2005Q1 Technical Overview*, <http://docs.sun.com/app/docs/doc/819-0069>, 2005.
- [8] D.Cybok, *Grid Workflow Infrastructure*, presented at GGF 10, Berlin, March 2004.
- [9] T.Fahringer, J.Qin and S.Hainzer, *Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*, presented at IEEE International Symposium on Cluster Computing and the Grid 2005, Cardiff, UK, 2005.

- [10] IBM. alphaWorks, *BPWS4J - A platform for creating and executing BPEL4WS processes*, <http://www.alphaworks.ibm.com/>, 2004.
- [11] Peter Henderson, *Modelling Architectures for Dynamic Systems*: Eds Annabelle McIver and Carroll Morgan, Springer-Verlag New York Inc., pp. 732-737, 2003.
- [12] Mordechai Beizer, Chair, AIIM Accreditation Workflow Subcommittee, *Interesting Times For Workflow Technology*, http://www.e-workflow.org/White_Papers/index.htm.
- [13] David Hollingsworth, *The Workflow Reference Model Workflow Management Coalition*, <http://www.wfmc.org/standards/docs/tc003v11.pdf>. 19 Jan 1995.
- [14] WIKIPEDIA, the free encyclopedia
http://en.wikipedia.org/wiki/Model_Driven_Architecture
- [15] Workflow Management Coalition, *Terminology & Glossary Workflow Management Coalition*, Feb 1999.
- [16] Barry & Associates, Inc., *Service-oriented architecture (SOA) definition*, <http://www.service-architecture.com/index.html>.
- [17] UDDI.org, *Advancing Web Services Discovery Standard*.
<http://www.uddi.org/>.
- [18] Christ Pelts, *Web Services Orchestration, a review of emerging technologies, tools, and standards*, Jan 2003.
http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf
- [19] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, Sanjiva Weerawarana, *The next step in Web Services*, Communications of the ACM, vol. 46, pp. 29-34, 2003.
- [20] Assaf Arkin, et al., *Web Service Choreography Interface (WSCl) 1.0*,

- <http://www.w3.org/TR/wsci/>, August 2002.
- [21] Antonio Bucchiarone, ***Pattern-based Analysis of WSCI, BPML and BPEL4WS***.
- [22] Aissi, Selim, Malu, Pallavi, Srinivasan, Krishnamurthy, ***E-business process modeling: The next big step***, Computer, vol. 35, pp. 55-62, 2002.
- [23] Doug Bunting, Martin Chapman, et al., ***Web Services Composition Application Framework (WS-CAF) V1.0***, oasis.org, 2003.
- [24] Peter Henderson. ***Reasoning about Asynchronous Behaviour in Distributed Systems***, The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02), pp. 17-24, 2002
- [25] William Cox, Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, Tony Storey, Satish Thatte, ***Web Services Transaction (WS-Transaction)***. <http://xml.coverpages.org/WS-Transaction2002.pdf>.
- [26] Doug Bunting, Martin Chapman, Oisín Hurley, Mark Little, Jeff Mischkinsky, Eric Newcomer, Jim Webber, Keith Swenson, ***Web Services Context (WS-Context) V1.0***, oasis.org, 2003.
- [27] Michael Kay, ***XSLT Programmer's Reference 2nd Edition - Programmer's Reference***, Wrox Press Ltd, 2001.
- [28] David Flanagan, ***JavaScript The Definitive Guide - Fourth Edition***: O'Reilly & Associates, Inc., 2002.
- [29] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Ferris, David Orchard, ***Web Services Architecture- W3C Working Group Note 11 February 2004***, W3C, 2004.
- [30] Peter Henderson, Jingtao Yang, ***Reusable Web Services***, presented at 8th International Conference, ICSR 2004, Madrid, Spain, pp. 185-194, July 2004.

- [31] <http://is.tm.tue.nl/research/patterns/patterns.htm>.
- [32] IBM Web Services Architecture Team, *Web Services architecture overview*, 01 Sep 2000. <http://www-128.ibm.com/developerworks/webservices/library/w-ovr/?dwzone=webservices>.
- [33] Christ Peltz, *Web services orchestration and choreography*, Computer, vol. 36, pp. 46-52, 2003.
- [34] Sun Microsystem, *Java 2 Platform Standard Edition 5.0 API Specification*. <http://java.sun.com/j2se/1.5.0/docs/api/>.
- [35] Sun Microsystem, *Java 2 Platform, Enterprise Edition, v 1.3 API Specification*. http://java.sun.com/j2ee/sdk_1.3/techdocs/api/index.html.
- [36] Apache.org, *WebServices - SOAP project*, <http://ws.apache.org/soap/>.
- [37] caucho technology, *Resin open source application server project*, <http://www.caucho.com/>.
- [38] Peter Henderson, *ARC*, <http://www.ecs.soton.ac.uk/~ph/software.htm>.
- [39] Jingtao Yang, Corina Cirstea and Peter Henderson, *Document Flow Model: A Formal Notation for Modelling Asynchronous Web Services Composition*, in proceedings of OnTheMove workshops OTM2005, Agia Napa, Cyprus, pp 39-48, 2005.
- [40] Jingtao Yang, Corina Cirstea and Peter Henderson, *An operational semantics for DFM: a formal notation for modelling asynchronous web services*, in proceedings of 5th International Conference on Quality Software, QSIC2005, Melbourne, Australia, pp 446-451, 2005.

- [41] Sanjiva Weerawarana, et al., *Web Services Platform Architecture - SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*, Prentice Hall 2005.
- [42] WIKIPEDIA, the free encyclopedia <http://en.wikipedia.org/wiki/wsdl>
- [43] Howard Foster: *A Rigorous Approach to Engineering Web Service Composition*. PhD Thesis. <http://www.doc.ic.ac.uk/~hfl/phd/downloads/hf-thesis-ds-2006.pdf>
- [44] M. Mecella, and B. Pernici: *Designing wrapper components for e-services in integrating heterogeneous systems*. VLDB Journal, Volume 10, pp. 2-15, 2001
- [45] Jana Koehler, Giuliano Tirenni, and Santhosh Kumaran: *From Business Process Model to Consistent Implementation: A case for Formal Verification Methods*. In proceedings of 6th IEEE international Enterprise distributed object computing conference (EDOC). pp. 96-106, 2002
- [46] Xiang Fu, Tevfik Bultan, and Jianwen Su: *Analysis of Interacting BPEL Web Services*. In Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), pp. 621-630, New York, NY, May 17-22, 2004.
- [47] Qiming Chen and Meichun Hsu: *Inter-Enterprise Collaborative Business Process Management*. In proceedings of 17th International Conference on Data Engineering (ICDE), pp. 253-260, Heidelberg, Germany, 2001.
- [48] Tracy Gardner: *UML Modelling of Automated Business Process with a Mapping to BPELWS*, presented at 1st European Workshop on Web Services and Object Oriented ECOOP 2003, Darmstadt, Germany, July 2003.
- [49] David Skogan, Roy Gronmo, and Ida Solheim: *Web Service Composition in UML*, In proceedings of Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04), pp. 47-57, 2004.

- [50] Robin Milner: *A Calculus of Communication Systems*, LNCS. Springer-Verlag, Berlin, 1980.
- [51] Antonio Brogi, Carlos Canal, Ernesto Pimentel and Antonio Vallecillo: *Formalizing Web Service Choreographies*. In proceedings of First International Workshop on Web Services and Formal Methods, Pisa, Italy, pp. 73-94, February 2004. Published also in ENTCS, 105:73-94, 2004.
- [52] Andrea Ferrara: *Web services: a process algebra approach*. In proceedings of the 2nd international conference on Service oriented computing, New York, USA, pp 242-251, 2004
- [53] Rachid Hamadi, and Boualem Benatallah: *A Petri Net-based Model for web service Composition*. In proceedings of the Fourteenth Australasian database conference on Database technologies 2003, Adelaide, Australia, pp. 191 – 200, 2003.
- [54] Yu Tand, Lou Chen, Kai-Tao He, and Ning Jing: *SRN: An extended Petri-Net-Based Workflow Model for Web Service Composition*. In proceedings of IEEE International Conference on Web Services (ICWS'04), San Diego, California, pp. 591-599, 2004.
- [55] Aneesh Khetarpal, Sol M. Shatz, and Shengru Tu: *Applying an Object-Based Petri Net to the Modelling of Communication Primitives for Distributed Software*. In proceedings of the High Performance Computing Conference (HPC98), Boston, Mass., pp 404-409, 1998.
- [56] Nikola Milanovic and Miroslaw Malek: *Current Solutions for Web Service Composition*. IEEE Internet Computing, pp 51-59, November/December 2004.
- [57] Shin Nakajima: *Verification of web service flows with model-checking techniques*. In Proceedings of Cyber World 2002, pp.378-385, IEEE, Nov. 2002.
- [58] Aysu Betin-Can, Tevfik Bultan and Xiang Fu: *Design for Verification for Asynchronously Communicating Web Services.* In Proceedings of the Fourteenth

- International World Wide Web Conference (WWW 2005), pp. 750-759, Chiba, Japan, May 10-14, 2005.
- [59] Xiang Fu, Tevfik Bultan, and Jianwen Su: *WSAT: A Tool for Formal Analysis of Web Services*. In Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004), LNCS 3114, pp. 510-514, Boston, Massachusetts, July 13-17, 2004.
- [60] David L. Parnas: *Designing software for ease of extension and contraction*. IEEE Transaction on Software Engineering SE-5(2)12839. pp 128-137, 1979.
- [61] Daniel A. Menascé: *Composing Web Services: A QoS View*. Internet Computing, IEEE Volume 8, Issue 6, Nov.-Dec. 2004 , pp. 88 – 90, 2004.
- [62] Ali Arsanjani, Francisco Curbera, and Nirmal Mukhi: *Manners externalize semantics for on-demand composition of context-aware services*. In proceedings of the IEEE International Conference on Web Services, (ICWS'04), 2004, San Diego, California, USA, pp. 583-590, 2004.
- [63] Peter Henderson: *Modelling Architectures for Dynamic Systems*. In Programming Methodology, Eds Annabelle McIver and Carroll Morgan, Springer-Verlag New York Inc.; ISBN: 0387953493 , pp 161-174, 2003.
- [64] Manshan Lin, Jianshan Xie, Heqing Guo, and Hao Wang: *Solving QoS-driven Web service dynamic composition as fuzzy constraint satisfaction*. In proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service, San Diego, California, USA, pp 9-14, 2005.
- [65] Snehal Thakkar, Craig A. Knoblock, and Jose Luis Ambite: *A View Integration Approach to Dynamic Composition of web services*. In proceedings of ICAPS'03 workshop on Planning for web services, Trento, Italy, June 2003.
http://icaps03.itc.it/satellite_events/documents/WS/WS5/06/thakkar-icaps2003-p4ws.pdf

Other Related Web Site

- [66] <http://standards.ieee.org/>
- [67] <http://www.e-workflow.org/>
- [68] <http://www-106.ibm.com/developerworks/>
- [69] <http://java.sun.com/>
- [70] <http://www.bpmi.org/>
- [71] <http://www.ecs.soton.ac.uk/~ph>
- [72] <http://www.w3.org/>
- [73] <http://www.oasis-open.com>

Other Related Standards and Specification

- [74] *Web Services Conversation Language (WSCL) 1.0*
<http://www.w3.org/TR/wscl10/>
- [75] *Simple Object Access Protocol (SOAP) 1.1 – W3C Note 08 May 2000*
<http://www.w3.org/TR/SOAP/>
- [76] *Web Services Architecture – W3C Working Group Note 11 February 2004*
<http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>
- [77] *Web Service Flow Language (WSFL 1.0)*
<http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>
- [78] *XLANG – Web Services for Business Process Design*
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm