Behavioural Protocol Representation and Adaptation in a Service Composition Framework

Linh Duy Pham

Submitted in fulfilment of the requirements of the degree of Master of Science by Research

Faculty of Information and Communication Technologies
Swinburne University of Technology

November 2009
(Revised in July 2010)
Abstract

Service Oriented Architecture (SOA) is a paradigm for building software applications from loosely-coupled services. Building large applications often involves combining services into bigger ones. This process is known as service composition. In the dynamic, open and distributed environments of SOA, services are not always available and new services are introduced that might provide other characteristics that are more beneficial to the current system than the existing ones. In order to utilise different services, a service composition framework needs to allow its constituent services to be substituted and ensure that the services involved in the composition interact with each other in a compatible manner both in terms of functionality and sequence of exchanged messages (or behavioural protocol).

In this thesis, we propose a service composition framework that has three capabilities: (i) specifying behavioural protocols among services and offering support for behavioural protocols including: aggregation of protocols, consistency checking and run-time protocol monitoring, (ii) checking compatibilities of potential services, and (iii) proposing adaptor patterns to alleviate the potential incompatibilities between services and the specified required sequence of interactions. Our service composition framework is built based on the Role Oriented Adaptive Design (ROAD) framework. Required sequences of interactions are defined by using a declarative pattern-based specification, the Interaction Rule Specification (IRS).

The compatibilities of services are checked by a mechanism involving three steps: converting IRS descriptions to Finite State Automata (FSAs), converting services’ behavioural specifications (e.g. in OWL-S) to FSAs, and finally analysing the resulting FSAs to identify compatibilities of potential services. In the case that services are incompatible to the required sequence of interactions, we propose three adaptation patterns that can be used to alleviate mismatches. The run-time overheads of behavioural protocol’s conversion to FSA and analysis are also evaluated. By supporting automated checking of behavioural compatibilities between services and providing adaptor patterns to address the identified behavioural mismatches, this thesis presents a further step towards automated service composition.
Declaration

This is to certify that this thesis contains no material which has been accepted for the award of any other degree or diploma, and that to the best of my knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Linh Duy Pham
July, 2010
Melbourne, Australia
Acknowledgements

I would like to express my sincerest appreciation to my research advisors, Dr. Jean-Guy Schneider, Prof. Jun Han and Dr. Alan Colman, for their constant support, guidance and encouragement. Their constructive feedbacks and patience to spend many hours to discuss and review my work have made this thesis possible.

I am grateful to Tan Manh Phan for fruitful discussions and feedbacks on early drafts of this thesis although he has a busy schedule himself. I would like to thank Swinburne University of Technology for providing me the scholarships and the facility to carry out this work.

I would like to express a very special gratitude to my Mum, Dad and my wife Van Dao, for their endless love, understanding and encouragement.
I dedicate this thesis to my Mum, Phạm Thị Đức Lợi, for her love and sacrifice without which I might not be where I am today.
Contents

ABSTRACT ........................................................................................................................................... I
DECLARATION .......................................................................................................................................... II
ACKNOWLEDGEMENTS ....................................................................................................................... III
CONTENTS ................................................................................................................................................ V
LIST OF FIGURES ................................................................................................................................... VIII
LIST OF TABLES ....................................................................................................................................... X
LIST OF LISTINGS .............................................................................................................................. XI
LIST OF PUBLICATIONS ..................................................................................................................... XII
CHAPTER 1 : INTRODUCTION ............................................................................................................. 1
  1.1 MOTIVATION .................................................................................................................................. 2
  1.2 APPROACH AND CONTRIBUTIONS ............................................................................................... 3
  1.3 THESIS OUTLINE ......................................................................................................................... 6
CHAPTER 2 : BACKGROUND AND LITERATURE REVIEW ................................................................. 8
  2.1 SERVICE ORIENTED ARCHITECTURE (SOA) ............................................................................... 9
  2.2 SERVICE COMPOSITION ............................................................................................................... 10
  2.3 BEHAVIOURAL DESCRIPTION .................................................................................................... 12
     2.3.1 Constraint-based descriptions ............................................................................................ 13
     2.3.2 Process-based descriptions ................................................................................................ 14
     2.3.3 Web Services Approaches to Behavioural Description ....................................................... 16
           2.3.3.1 Web Service Semantics (WSDL-S) ........................................................................... 16
           2.3.3.2 Web Service Modeling Ontology (WSMO) ............................................................... 17
           2.3.3.3 Web Ontology Language for Services (OWL-S) ...................................................... 19
           2.3.3.4 Summary ................................................................................................................ 21
  2.4 FORMALISMS FOR EXPRESSING AND REASONING ABOUT BEHAVIOURAL PROTOCOLS .... 22
  2.5 MATCH-MAKING OF SERVICES .................................................................................................. 26
     2.5.1 Capability Matching .............................................................................................................. 26
     2.5.2 Sequence of Interaction Matching ..................................................................................... 27
     2.5.3 Summary ........................................................................................................................... 29
  2.6 MISMATCH ADAPTATION ............................................................................................................... 29
     2.6.1 Adaptation at Behavioural Level ......................................................................................... 30
  2.7 SUMMARY ....................................................................................................................................... 33
CHAPTER 3 : PRELIMINARIES ............................................................................................................. 35
  3.1 MOTIVATING SCENARIOS ............................................................................................................ 35
  3.2 THE ROLE ORIENTED ADAPTIVE DESIGN (ROAD) FRAMEWORK ....................................... 38
     3.2.1 Role and Player .................................................................................................................... 39
     3.2.2 Contract ............................................................................................................................... 40
           3.2.2.1 Control Communication Act (CCA) ......................................................................... 40
           3.2.2.2 Term ........................................................................................................................... 41
           3.2.2.3 Protocol Clause .......................................................................................................... 42
           3.2.2.4 Summary .................................................................................................................... 43
     3.2.3 Composite and Organiser ...................................................................................................... 43
     3.2.4 Flexibility and Adaptability of ROAD Framework .............................................................. 45
  3.3 BEHAVIOURAL PROTOCOL DESCRIPTION .............................................................................. 46
     3.3.1 Patterns .................................................................................................................................. 47
     3.3.2 Scopes ..................................................................................................................................... 48
     3.3.3 Parameters ............................................................................................................................ 49
3.4 FSA SEMANTICS OF CONSTRAINT PATTERNS ................................................................. 50
3.4.1 FSA Templates for Occurrence Patterns ............................................................... 51
3.4.2 FSA Templates for Order Patterns ....................................................................... 52
3.4.3 FSA Templates for Scopes .................................................................................. 53
3.4.4 Compound Constraints and Composition of FSAs ............................................. 55
3.5 SUMMARY .............................................................................................................. 58

CHAPTER 4 : BEHAVIOURAL PROTOCOL IN THE ROAD FRAMEWORK ....................... 60
4.1 USING IRS TO DEFINE BEHAVIOURAL PROTOCOLS IN CONTRACTS ................ 60
4.2 PROTOCOL DEPENDENCIES BETWEEN CONTRACTS ........................................... 65
4.3 ROLE-CENTRIC PROTOCOL .................................................................................. 69
4.3.1 Protocol Aggregation with One Contract ......................................................... 69
4.3.2 Protocol Aggregation with Multiple Contracts ................................................. 71
4.4 CONSTRAINTS COMPOSITION AND CONSISTENCY CHECKING ......................... 72
4.5 RUN-TIME PROTOCOL MONITORING .................................................................. 73
4.6 SUMMARY .............................................................................................................. 75

CHAPTER 5 : MATCH-MAKING OF WEB SERVICES TO ROLES ..................................... 76
5.1 THE PROCESS OF MATCH-MAKING ...................................................................... 76
5.2 CONVERTING OWL-S TO FSA ............................................................................ 79
5.3 MATCH-MAKING APPROACH AND MISMATCHES CLASSIFICATION .................. 86
5.3.1 Functional Capability Matching ....................................................................... 86
5.3.2 Behavioural Protocol Matching ....................................................................... 87
5.3.3 Considering both Functional and Behavioural Mismatches ................................ 89
5.4 SUMMARY .............................................................................................................. 90

CHAPTER 6 : MISMATCH ADAPTATION ....................................................................... 91
6.1 ADAPTOR COMPOSITE ......................................................................................... 91
6.2 CONVERSATION COORDINATOR ....................................................................... 93
6.3 FUNCTIONAL ROLE COMPOSITION ..................................................................... 97
6.4 TRUSTED MEDIATOR .......................................................................................... 99
6.5 APPLYING THE ADAPTOR PATTERNS AND DISCUSSION ................................... 101
6.6 SUMMARY ............................................................................................................. 103

CHAPTER 7 : IMPLEMENTATION ............................................................................... 104
7.1 SYSTEM DESIGN OVERVIEW ............................................................................... 104
7.2 THE ROAD FRAMEWORKS’ CORE ELEMENTS - THE ROAD,CORE PACKAGE ............ 106
7.3 PROTOCOL SUPPORT MECHANISM - THE ROAD,PROTOCOL PACKAGE .............. 107
7.3.1 Converting IRS constraints to FSAs ................................................................. 107
7.3.2 Converting OWL-S to FSA ............................................................................. 108
7.4 RUN-TIME MONITORING - THE ROAD,MONITORING PACKAGE ......................... 112
7.5 COMPATIBILITY CHECKING – THE ROAD,COMPATIBILITYCHECK PACKAGE ........ 114
7.5.1 Functional compatibility checking ..................................................................... 114
7.5.2 Behavioural compatibility check ...................................................................... 115
7.6 ADAPTATION - THE ROAD,ADAPTATION PACKAGE ........................................ 118
7.7 GUI DISPLAY - THE ROAD,GUI PACKAGE ................................................................ 119
7.8 SUMMARY .............................................................................................................. 121

CHAPTER 8 : ANALYSIS AND DISCUSSION .................................................................. 122
8.1 ADVANTAGES OF OUR BEHAVIOURAL PROTOCOL APPROACH ......................... 122
8.1.1 Behavioural Interface Description ..................................................................... 122
8.1.2 Behavioural Match-Making .............................................................................. 123
8.1.3 Mismatch Adaptation ....................................................................................... 124
8.2 LIMITATIONS ......................................................................................................... 124
8.3 PERFORMANCE ANALYSIS .................................................................................. 125
8.3.1 FSA Intersection Analysis ................................................................. 125
8.3.2 Conversion from OWL-S to FSA Overhead ............................... 131
8.3.3 Conclusion .................................................................................... 136
8.4 SUMMARY ..................................................................................... 137

CHAPTER 9: CONCLUSIONS AND FUTURE WORK ........................................ 138

9.1 CONTRIBUTIONS .............................................................................. 138
9.2 FUTURE WORK .................................................................................. 141

REFERENCES .......................................................................................... 143

APPENDIX A: AN EXAMPLE OF AGGREGATION OF MULTIPLE CONTRACTS .......... 151

APPENDIX B: OWL-S FILE USED IN PERFORMANCE TESTING FOR SEQUENCE OF ATOMIC PROCESSES .................................................................................................................. 157

APPENDIX C: OWL-S FILE USED IN PERFORMANCE TESTING SEQUENCE OF IF-THEN-ELSE CONSTRUCTS .................................................................................................................. 162
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1</td>
<td>Basic ROAD framework's elements</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2-1</td>
<td>A ceremony process definition (Casati and Shan 2001)</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2-2</td>
<td>WSMO Web service — general description as from (Roman et al. 2005)</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2-3</td>
<td>Top level of OWL-S service ontology (Martin et al. 2004a)</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2-4</td>
<td>FSA behavioural protocol specification</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3-1</td>
<td>Message ordering mismatch</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3-2</td>
<td>Missing functionalities</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3-3</td>
<td>There is no valid sequence of interaction</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3-4</td>
<td>Behavioural mismatches at interaction levels and missing of functionality</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3-5</td>
<td>Roles and players</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3-6</td>
<td>Summary of the contract concept</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3-7</td>
<td>Library book buying composite and its higher level composite</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3-8</td>
<td>The pattern hierarchy (Dwyer et al. 1999)</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3-9</td>
<td>Pattern scopes (Dwyer et al. 1999)</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3-10</td>
<td>FSA templates for occurrence patterns (Jin and Han 2005a)</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3-11</td>
<td>FSA templates for order patterns (Jin and Han 2005a)</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3-12</td>
<td>FSA templates for existence pattern with scope (Jin and Han 2005a)</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3-13</td>
<td>FSA intersection of “P precedes Q globally” and “P leads to Q globally”</td>
<td>57</td>
</tr>
<tr>
<td>Figure 4-1</td>
<td>Library book buying composite</td>
<td>61</td>
</tr>
<tr>
<td>Figure 4-2</td>
<td>The FSA of behavioural protocol that vendor services are required to support</td>
<td>61</td>
</tr>
<tr>
<td>Figure 4-3</td>
<td>Behavioural protocol FSA output from composing constraints</td>
<td>63</td>
</tr>
<tr>
<td>Figure 4-4</td>
<td>FSA representation of “P precedes Q” and “R precedes S”</td>
<td>64</td>
</tr>
<tr>
<td>Figure 4-5</td>
<td>The FSA of behavioural protocol that allows order cancellation</td>
<td>64</td>
</tr>
<tr>
<td>Figure 4-6</td>
<td>Behavioural protocol FSA output from composing constraints that allows order cancellation</td>
<td>65</td>
</tr>
<tr>
<td>Figure 4-7</td>
<td>Library book buying composite with broker role</td>
<td>67</td>
</tr>
<tr>
<td>Figure 4-8</td>
<td>Structure of the FSA generator in an organiser</td>
<td>72</td>
</tr>
<tr>
<td>Figure 4-9</td>
<td>The run-time protocol monitoring process</td>
<td>74</td>
</tr>
<tr>
<td>Figure 5-1</td>
<td>The creation of role-centric protocol and the problem of different notations</td>
<td>78</td>
</tr>
<tr>
<td>Figure 5-2</td>
<td>The steps required during match-making process</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5-3</td>
<td>Modelling OWL-S processes as FSA (Mokhtar et al. 2005)</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5-4</td>
<td>An illustration of converting a sequence control construct</td>
<td>83</td>
</tr>
<tr>
<td>Figure 5-5</td>
<td>The structure of an example OWL-S process specification</td>
<td>84</td>
</tr>
<tr>
<td>Figure 5-6</td>
<td>Non deterministic FSA result of OWL-S to FSA conversion</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Tables

TABLE 2-1: EVALUATION SUMMARY OF WSDL-S, WSMO AND OWL-S ................................................................. 22
TABLE 3-1: A SIMPLE SET OF CCA MESSAGE TYPES FOR THE LIBRARY BOOK BUYING SCENARIOS .................. 41
TABLE 3-2: A simple set of Terms for the Library Book Buying Scenarios ........................................................ 42
TABLE 5-1: FUNCTIONAL AND BEHAVIOURAL MISMATCHES AT THE SERVICE LEVEL ............................................. 89
TABLE 6-1: MISMATCH CLASSIFICATIONS AND ADAPTATIONS FOR THE LIBRARY BOOK BUYING SCENARIOS ........ 102
TABLE 8-1: MEAN EXECUTION TIME IN MILLISECONDS FOR COMPOSING CONSTRAINTS IN FOUR SCENARIOS ........ 129
TABLE 8-2: NUMBERS OF TRANSITIONS AND STATES OF THE RESULTING FSAS WHEN COMPOSING CONSTRAINTS IN FOUR SCENARIOS .................................................................................................. 129
TABLE 8-3: RESPONSE TIMES OF CONVERTING OWL-S SPECIFICATIONS NOT HAVING IF-THEN-ELSE CONSTRUCT TO FSAS ..................................................................................................................... 133
TABLE 8-4: RESPONSE TIMES OF CONVERTING OWL-S SPECIFICATIONS HAVING IF-THEN-ELSE CONSTRUCT TO FSAS .... 134
List of Listings

LISTING 3-1: WHERE-CLAUSE ADDS PARAMETER CONDITIONS TO CONSTRAINTS ............................................................... 50
LISTING 4-1: A SET OF CONSTRAINTS OF THE BEHAVIOURAL PROTOCOL THAT VENDOR SERVICES NEED TO SUPPORT .... 63
LISTING 4-2: A SET OF CONSTRAINTS OF THE LIBRARY THAT REQUIRE VENDOR SERVICES TO SUPPORT ORDER CANCELLATION OPERATION ......................................................................................................................... 65
LISTING 4-3: DEPENDENCIES BETWEEN BUYER-BROKER AND BROKER-VENDOR CONTRACTS .............................................. 68
LISTING 4-4: A DEPENDENCY CONSTRAINT BETWEEN BUYER-BROKER AND BROKER-VENDOR CONTRACTS WITH A WHERE-CLAUSE ........................................................................................................................................ 69
LISTING 4-5: PSEUDO-CODE OF PROTOCOL AGGREGATION WITH ONE CONTRACT .......................................................... 70
LISTING 4-6: VENDOR1’S ROLE-CENTRIC-PROTOCOL ............................................................................................................... 70
LISTING 4-7: PSEUDO-CODE OF PROTOCOL AGGREGATION WITH MULTIPLE CONTACTS .................................................. 71
LISTING 4-8: SUMMARY OF AGGREGATION OF BROKER’S ROLE-CENTRIC PROTOCOL .......................................................... 72
LISTING 5-1: CONVERTING NON-DETERMINISTIC FSA TO DETERMINISTIC FSA (BASED ON (HOPCROFT ET AL. 2001)) .... 85
LISTING 6-1: PSEUDO-CODE OF THE MESSAGE PROCESSING IN A COORDINATOR PLAYER FOR SCENARIO 1 ..................... 96
LISTING 6-2: PSEUDO-CODE OF THE MESSAGE PROCESSING IN A COORDINATOR PLAYER FOR SCENARIO 3 ..................... 96
LISTING 6-3: PSEUDO-CODE OF THE MESSAGE PROCESSING IN A COORDINATOR PLAYER FOR SCENARIO 2 ..................... 99
LISTING 6-4: PSEUDO-CODE OF THE MESSAGE PROCESSING IN A COORDINATOR PLAYER FOR SCENARIO 4 ................. 101
LISTING 7-1: PSEUDO-CODE OF CONVERTING AN OWL-S PROCESS DESCRIPTION TO FSA .................................................. 109
LISTING 7-2: PSEUDO-CODE OF CONVERTING A SEQUENCE CONSTRUCT TO FSA ......................................................... 110
LISTING 7-3: PSEUDO-CODE OF CONVERTING A CHOICE CONSTRUCT TO FSA .................................................................... 110
LISTING 7-4: PSEUDO-CODE OF CONVERTING AN IF-THEN-ELSE CONSTRUCT TO FSA ....................................................... 111
LISTING 7-5: PSEUDO-CODE OF CONVERTING A SPLIT/SPLIT+JOIN AND ANY-ORDER CONSTRUCTS TO FSA .................. 111
LISTING 7-6: PSEUDO-CODE OF CALCULATING COMBINATIONS OF FSAS FROM MULTIPLE LISTS OF FSAS .................. 111
LISTING 7-7: PSEUDO-CODE OF RUN-TIME PROTOCOL MONITORING IN CONTRACT AND ORGANISER ......................... 114
LISTING 7-8: FSA INTERSECTION PSEUDO-CODE (FROM (PHAN 2006)) .................................................................................... 116
LISTING 7-9: PSEUDO-CODE OF MATCHING A SERVICE’S FSAs TO A ROLE’S FSA ............................................................... 118
LISTING 8-1: RELATED CONSTRAINTS ........................................................................................................................................ 127
LISTING 8-2: UNRELATED CONSTRAINTS .................................................................................................................................. 127
LISTING 8-3: 5-EVENT SET AND 12-EVENT SET .......................................................................................................................... 127
LISTING 8-4: 12 UNRELATED CONSTRAINTS .................................................................................................................................... 128
LISTING 8-5: 12 RELATED CONSTRAINTS ...................................................................................................................................... 128
List of Publications


Chapter 1:
Introduction

In recent years, Service Oriented Architecture (SOA) has emerged as a standard technology to build distributed applications. SOA provides an architecture that allows heterogeneous applications to interact with each other by using various Web service standards such as WSDL\(^1\) for interface descriptions, HTTP and SOAP\(^2\) for transport and communication protocols, and UDDI\(^3\) for registration and discovery of services. Applications are exposed as services and can be invoked over the Internet. In SOA, building large applications often involves combining other services in a process consisting of service discovery, evaluation and composition. Services in a composition are required to be compatible not only in terms of functionality but also *behavioural protocols*. Behavioural protocols define the required sequences of exchanged messages among interacting parties. In other words, behavioural protocols specify sequencing constraints on service interactions in the services’ interfaces (Farias et al. 2002). In the rest of this thesis, the term “behavioural protocol” refers to the specification of sequence of interactions between services (for example, operation \(a\) is invoked before operation \(b\)).

The compatibility checking at the behavioural protocol level, at present, is mostly carried out manually by developers examining existing services. When services in a composition become unavailable (due to various reasons such as network outage, services permanently removed, etc.), or when services change their behavioural protocols, the entire evaluation process will need to be carried out again to replace un-operational services in the composition. This thesis reports on our research efforts

\(^{1}\) WSDL – Web Service Description Language - [http://www.w3.org/TR/wSDL](http://www.w3.org/TR/wSDL)

\(^{2}\) SOAP – Simple Object Access Protocol - [http://www.w3.org/TR/soap/](http://www.w3.org/TR/soap/)

\(^{3}\) UDDI – Universal Description Discovery and Integration - [http://www.uddi.org/pubs/uddi_v3.htm](http://www.uddi.org/pubs/uddi_v3.htm)
in developing a semi-automated service composition framework supporting compatibility checking of services against the required behavioural protocols and adaptation patterns to alleviate mismatches when services are not fully compatible.

In this chapter, we detail the background and motivation of our research in Section 1.1. An overview of our approach to address the problem is presented in Section 1.2. Finally, Section 1.3 gives an outline of the remainder of this thesis.

1.1 Motivation

Developing software applications from existing components is a goal that many researchers are trying to achieve (Szyperski 2002). Using existing software components offers reuse of not only the functionality, but also the quality of already-tested components (Meyer 1994). Following the same principle, Service Oriented Architecture (SOA) has recently emerged as a methodology of (re)using services as fundamental elements in the development process of service oriented applications (Milanovic and Malek 2004). A service is a well defined, self-contained module that provides certain coherent units of business functionality. A service exposes a standardised interface description to be invoked by other services.

In a business transaction, different services provided by business applications need to interact with each other in order to accomplish a given goal. SOA and networking infrastructure enable developers to build distributed applications which comprise of loosely-coupled services. SOA thus facilitates inter-organisation interactions. In SOA, a typical service implementation process often involves, among others, service discovery, evaluation and composition. Services can be composed to create new services thus providing more value in terms of reuse. This process is known as service composition.

To support composition of services, a variety of standards such as WSDL and SOAP have helped resolving the heterogeneity in implementation platforms. However, standardisations in the syntactic interface description of services (e.g. WSDL) alone are not sufficient to ensure the correct interoperation of services (Li et al. 2005). Services also need to be compatible in terms of behavioural protocols among services in order to interoperate. A behavioural protocol defines the required
sequence of exchanged messages among interacting parties. For example, in the case that an application interacts with a Web service to order books, clearly there is a required sequence of interactions from getting quote, placing order, payment, to sending delivery details, etc. The required sequence of interactions have to be specified and agreed upon in order to avoid wrong assumptions that components make about the environment and about other components. The problem caused by these wrong assumptions is not new but already existed in component based software engineering. Garlan et al. (1995) pointed out that these assumptions are often implicit and conflict with each other, which causes mismatches during composition. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained a certain level of maturity and adoption by using Web service technology such as WSDL, SOAP, etc., there remain open challenges when it comes to managing and coordinating service interactions (Barros et al. 2005). How to solve the incompatibility issues in service behavioural protocols of the participating services is among those challenges.

In SOA, services are in a very dynamic environment, where new services are introduced, or existing ones are no longer available. In such environment, the ability to adapt to different services is one of the solutions to ensure applications can still be functional by using other alternative services.

The work presented in this thesis addresses the problems mentioned above in three aspects: (i) specifying behavioural protocols of parties in an interaction and managing behavioural protocols by supporting aggregation of protocols, consistency checking and run-time protocol monitoring; (ii) compatibility checking of potential services in terms of behaviour; and (iii) proposing adaptor patterns to alleviate the incompatibilities between potential services and the required behavioural protocol.

1.2 Approach and Contributions

Services are software systems that can be accessed over a network. In this environment, one service generally does not have control over the existence of other services. In such a dynamic and open environment of services where new services are introduced and existing ones might become unavailable, we need an adaptive
framework that allows the exchange of services in an application. Developers specify the required functional and behavioural descriptions of services; and when a service becomes unavailable, the framework will search for a new service that matches the requirements.

The Role Oriented Adaptive Design (ROAD) framework developed by Colman and Han (Colman and Han 2005) presents an approach to adaptive software. The ROAD framework models an application as having a role-based organisational structure. Roles are associated by *contracts*. Roles are place holders which define abstract functions within a structured organisation, and a role can be played by a *player* that executes the required functions, e.g. a component, an agent, or a Web service. At run-time, to perform an application’s functionality, players interact with each other via their roles. When the need arises, a ROAD-based application will evaluate and change the players or modify the role structure; thus dynamically configuring its internal structure in order to adapt to the changing environment.

Figure 1-1 shows some basic elements of the ROAD framework. Consider the scenario where a library book buying application (*lib-BB*) interacts with a book vendor service to order books (such as the Amazon’s Web service). By using the ROAD framework, we model this system by having two roles: *Buyer* and *Vendor*; the *lib-BB* and the Amazon’s Web service act as players for the *Buyer* and *Vendor* roles respectively. Players do not know each other; their interactions are indirect via their roles. In this setup, players can be replaced by other suitable ones.

![Figure 1-1: Basic ROAD framework’s elements](image)

By employing Web services as players for roles, the ROAD framework provides an adaptive integration framework for Web services. Web services are players performing specific roles in an application. When a Web service is no longer available to play a role, a different Web service might be utilised to play that specific
role. In this thesis, we enhance the ROAD framework to offer support for behavioural protocols to ensure that players are compatible to the required sequence of interactions defined by developers.

This research will extend the work in the ROAD framework by adding a behavioural protocol aspect to the framework. This work involves:

- **Adding behavioural protocol support:** In a ROAD-based application, roles represent the functional and behavioural requirements that the application expects from services. These requirements are specified in contracts between roles. In order to specify behavioural protocols, we utilise a pattern-based declarative approach, *Interaction Rule Specification (IRS)* (Jin and Han 2005a). IRS has different patterns to define the temporal constraints between events of interest, and the authors expressed the semantics of IRS patterns by using Finite State Automaton (FSA) as the underlying formalism. By using these patterns to specify behavioural protocols, we have a formal foundation to support analysis and reasoning of behavioural protocols. In addition, being declarative, IRS supports incremental specification of protocols by adding and/or removing constraints. IRS also has an advantage in specifying only constraints on events of interest, rather than defining a rigid flow of events like other formalisms such as $\pi$-calculus or Petri nets. Since a role can be involved in multiple contracts, the specified requirements in contracts need to be aggregated to form role-centric requirements. The role-centric requirements are then checked for any inconsistencies. For a player to play a certain role, it has to conform to these role-centric requirements. Run-time protocol monitoring will intercept the interactions between parties to ensure the specified behavioural protocol is adhered to.

- **Compatibility checking of potential services in terms of behaviour:** A Web service that has the required functionality might have a different behavioural protocol from what a given role expects. This is called protocol mismatch. Before using a service, the behavioural protocol of that service will need to be checked against the required protocol. Currently, there are several proposed standards for specifying protocols and facilitating the Web service composition such as OWL-S (Martin et al. 2004a), WS-BPEL (Alves et al.
2007), WSDL-S (Akkiraju et al. 2005), etc. Different Web service providers can adopt different technologies. To perform protocol compatibility checking of Web services against roles’ requirements, Web services’ protocols and roles’ protocols will need to be converted to a common underlying formal expression such as FSA. If there are any mismatches found during compatibility checking, those mismatches will be analysed and classified so that adaptation can be applied to alleviate them.

- **Generating adaptors using adaptation patterns:** From the analysis performed in the previous step, different adaptation patterns can be applied to generate an adaptor that will coordinate the interactions and alleviate the mismatches between a role and its player. In our approach, the adaptor generation is not fully automated but rather it is a semi-automated process. Adaptor patterns provide templates of adaptors’ structures that developers can apply to create the required specific adaptors.

### 1.3 Thesis Outline

This rest of this thesis details our research work in 8 chapters:

Chapter 2 presents a detailed background and literature review. Firstly, the backgrounds of SOA and service composition are discussed. It then examines existing work in specifying behavioural protocols, compatibility checking and adaptation, respectively.

Chapter 3 introduces different business scenarios highlighting the problems that motivate this research. These scenarios are also used to illustrate the approach presented in this thesis. The concept of the Role Oriented Adaptive Design (ROAD) framework, which is the foundation that our work is built upon, is discussed in more detail. The behavioural protocol specification that we used, the Interaction Rule Specification (IRS) formalism, is then described. The chapter also discusses the advantages of IRS and the reasons for adopting it. The mapping between IRS patterns and Finite State Automata (FSAs) is also detailed.

Chapter 4 discusses our enhancement to the ROAD framework by adding behavioural protocol support. We show how IRS can be used to specify behavioural
protocols in contracts between roles. In addition to behavioural protocol constraints defined in contracts between roles, we show that dependencies between contracts also need to be defined. Roles are involved in different contracts; hence constraints related to a specific role in those contracts are required to be aggregated to produce a role-centric protocol. This role-centric protocol will represent the requirement that a Web service has to satisfy in order to become a player of this role. Since constraints are specified incrementally, there are possibilities that constraints are not consistent. The consistency checking mechanism is also detailed.

Chapter 5 presents how the compatibility checking between Web services and roles is carried out. The method to convert Web services’ protocol specifications in OWL-S descriptions to FSAs is detailed. Then the match-making process and a mismatch classification are presented.

Chapter 6 introduces our three adaptation patterns: conversation coordinator, functional role composition, and trusted mediator. How those three patterns can be used to alleviate identified mismatches is also presented.

Chapter 7 details the implementation of a prototype of the above concepts, including converting IRS to FSA, converting OWL-S to FSA, and behavioural protocol compatibility checking.

Chapter 8 discusses advantages and limitations of our approach. Performance test results are shown to give an indication of the overhead involved in composing and analysing behavioural protocols.

Finally, Chapter 9 concludes the thesis with a summarised discussion of our contributions to the field of service composition and future work to further enhance the support for behavioural protocols.
Chapter 2:  
Background and Literature Review

As stated in Chapter 1, this research aims at developing an adaptive service composition framework that supports:

- Defining behavioural protocols of interacting parties
- Compatibility checking behavioural protocols of the potential services
- Proposing adaptation patterns to alleviate the identified incompatibilities

This chapter provides the background of Service Oriented Architecture (SOA), service composition and existing research work that are closely related to our research. In Section 2.1, we summarise the SOA concept, the main context of our work. In SOA, services are often combined to realise users’ goals; this process is known as service composition. Service composition is presented in Section 2.2. At present, service composition is carried out mostly manually by developers, which does not scale well. Automated service composition was introduced to address this problem. For automated service composition to work, there are several areas that need to be addressed. In Section 2.3, we discuss the need to add extra information into service interface descriptions such as behavioural specification. Then we review different formalisms that are used to express and reason about behavioural protocols in Section 2.4. To find a suitable service given a behavioural requirement, a matchmaker will interpret behavioural specifications of advertised services in a service registry and match them against the formalised requirements. Match-making techniques are discussed in Section 2.5. To extend the number of potential services that a service consumer can interact with, adaptation to the identified incompatibilities is required. Section 2.6 presents several adaptation approaches. And finally, Section 2.7 will summarise this chapter.


2.1 Service Oriented Architecture (SOA)

As the network infrastructure gains advancement and provides a much higher speed, it opens new opportunities in distributed applications. Service Oriented Architecture (SOA) has emerged as an architecture that provides supports for seamless integration of distributed components. The key features of SOA are that it addresses the requirements of loosely coupled, standards-based, and protocol-independent distributed computing (Papazoglou and Heuvel 2007).

According to Papazoglou and Heuvel (2007), in an SOA, software resources are packaged as “service”. A service in SOA is an exposed piece of functionality with three essential properties:

- **Self-contained module**: Services are well defined, self-contained modules that provide business functionality and are independent of the state or context of other services.

- **Platform independence**: Services publish their interfaces in standard definition languages (e.g. WSDL – Web Service Description Language), and communicate with each other via standard protocols (e.g. SOAP – Simple Object Access Protocol). These standards enable communication between services implemented on heterogeneous platforms.

- **Dynamic invocation**: Services can be dynamically located, invoked and (re-) combined across a network.

Services often utilise but are not limited to Web services standards such as WSDL, SOAP and UDDI (Universal Description, Discovery and Integration). Web service is the most popular technology used to implement SOA.

SOA provides a number of advantages (Singh and Huhns 2005; Papazoglou and Heuvel 2007):

- **Increase reusability**: SOA promotes reuse at a service level rather than at an object or a package level.

- **Increase interoperability**: Standards enable the interoperation of software produced by different programmers and/or organisations.

- **Build distributed applications more effectively**: SOA together with Web services standards help building distributed applications effectively.
- **Facilitate inter-organisational interactions:** Software systems are more distributed; the software companies do not necessarily need to own and maintain the required infrastructures in order to use the services. It is also easier to invoke services provided by other companies.

- **Provide dynamism and more opportunities:** Since service consumers can discover and utilise new services, they have more choices among available services. Software systems become more dynamic; different services which offer similar functionalities can be selected and changed at run-time to suit the current context.

Although services can be invoked independently, services are often composed in order to meet more complicated users’ goals. The next section will discuss the current state of the arts in service composition.

### 2.2 Service Composition

The term “service composition” is used to describe the composition of services in a process flow, which contains *control* flow and *data* flow amongst services. Service composition is about providing an open, standard-based approach for connecting Web services together (Peltz 2003). There are two main streams in service composition techniques: **orchestration** and **choreography**.

- **Orchestration:** describes how Web services can interact with each other at the message level, including business logic and execution order of the interactions in a process model (Peltz 2003). Orchestration corresponds best to process languages such as Web Services Business Process Execution Language (WS-BPEL) (Alves et al. 2007).

- **Choreography:** tracks the sequence of messages that may involve multiple parties. Choreography is typically associated with the public message exchanges that occur between multiple Web services, rather than a specific business process that is executed by a single party (Peltz 2003). Choreography corresponds best to languages such as Web Services Choreography Description Language (WS-CDL) (Kavantzas et al. 2005).
As Barros et al. (2005) and Petlz (2003) point out: “a notable difference between orchestration and choreography is that **orchestration** refers to an **executable** business process that may interact with both internal and external services and the process is always controlled from the **perspective of one** of the business parties. Whereas **choreography** is more collaborative in nature, in which each party involved in the process describes the part they play in the interaction, interactions between services are seen from a **global** perspective”. The work presented in this thesis takes the approach of choreography to describe the required sequences of interactions between constituent services in a composition.

The major drawback of current service composition techniques is that they require developers to perform service composition manually by exploring what a service does in order to compose and execute a collection of services to achieve some objectives (Martin et al. 2007). Research work in **automated service composition** aims to address this problem. “Automated service composition is the process of automatically selecting, combining, integrating and executing Web services to achieve a user’s objective” (Martin et al. 2007).

As we have mentioned in Chapter 1, there are a number of problems that must be addressed for automated Web service composition to work:

- **Service interface description:** Current Web service interface description languages (e.g. WSDL) only describe the set of operations supported by a given Web service. WSDL does not describe the semantics of the inputs/outputs and the effects of operations. WSDL does not have the expressiveness to describe the order in which a service’s operations need to be invoked so that the service can be used correctly in a composition. In addition, all of that information needs to be documented in a computer-interpretable form.

- **Match-making:** In the match-making process, the requirements of functionality and behavioural protocols are matched against the ones offered by available services. The match-making process has to be performed automatically.

- **Adaptation:** In many cases, advertised services do not match perfectly to the requirements of a given client. Instead of rejecting these candidate services, if
the mismatches can be adapted, the number of services that can perform the client’s goal can be increased. The client will have more options in choosing the best candidate services based on their performance or other non-functional requirements.

The next section will review some of the research to the above-mentioned problems in more detail.

2.3 Behavioural Description

To support composition of services, a variety of standards such as WSDL and SOAP have helped resolving the heterogeneity in implementation platforms. However, standardisations in the syntactic interface description of services (e.g. WSDL) alone are not sufficient to ensure the correct interoperation of services. By examining the WSDL description of a service, we cannot unambiguously determine what the service does, and what the meanings of its inputs and outputs are. Two services can have the same syntactical specification but perform significantly different functions, whereas two syntactically dissimilar services can perform the same function (Akkiraju et al. 2005). A typical example is addition and subtraction operations which take the same inputs of two numbers, but they are significantly different in terms of semantics. Therefore, adding extra information into service interfaces to ensure the correct execution of services and to improve software reuse and discovery is important.

Previous research work suggests that there are four levels of component interface specifications: syntax, behaviour, synchronisation and Quality of Service (QoS) (Beugnard et al. 1999; Han 2000; Collet 2001).

- **Syntax:** Syntax describes signatures of operations and inputs/outputs data types. Syntactic information is checked at compile time.
- **Behaviour:** Behaviour specifies semantic description of data types and the sequence of interaction.
- **Synchronisation:** This level describes the concurrent feature of operations, i.e. synchronous, asynchronous, one-way, or call-back, whether it supports concurrent access or forces serialisation access to its services, and whether re-entrance is allowed.
- **Quality of Service (QoS):** This level describes the non-functional aspects of operations, e.g. security, response time, etc.

While the *syntactical* level of the interface is well-defined in both component-based software engineering (e.g. CORBA Interface Description Language\(^4\)) and SOA (e.g. WSDL), standard description languages in the other three levels are still immature. At the *behavioural* level, there is a lot of research effort addressing the behavioural interoperability, in particular the sequence of exchanged messages. Behavioural description is also the focus of our work.

There are different approaches to specify behavioural descriptions. In this chapter, we will review constraint-based, process-based descriptions and three main submissions to the World Wide Web Consortium (W3C) namely WSDL-S, WSMO and OWL-S, respectively.

### 2.3.1 Constraint-based descriptions

In constraint-based descriptions, behavioural protocols are specified by defining constraints on the sequence of operations invocations.

**Pre/post conditions:** Cho and McGregor (1999) used the UML Object Constraint Language (OCL) (Warmer and Kleppe 1999) to specify the pre and post-conditions for each method in the interface specification and used a state machine construct to describe protocols. The *interoperability testing framework* proposes a useful way to automatically generate test cases to verify the component interaction by exploring the execution tree of the protocols. In their approach, there are strong dependencies between interface specification and the protocol specification, which limits the modification of protocol and reuse of interface specification. The protocol specification is described in tuples of `<state : event ➔ state>`. If there are changes to the protocol, the entire specification has to be revised.

**Conditional message flows:** The SOAP Services Description Language (SSDL) (Kuo et al. 2006; Parastatidis et al. 2005) is an extensible XML-based language for specifying service contracts underpinned by the Conditional Message Flows (CMFs)

---

model. SSDL focuses on specifying messages and protocols to describe a SOAP-based Web service. CMFs define message behavioural protocols by using boolean conditions to specify when messages can be sent and received. One key benefit of specifying message behaviour by a set of sequential flows with conditions is that there is a clear separation between messages for the normal case from messages dealing with exceptions (Kuo et al. 2006). There are certain advantages in using CMF compared to existing process modelling languages such as WS-BPEL in terms of expressiveness. However, CMF does not support incremental specification of the protocols; hence the protocols have to be redefined if there are changes required at run-time.

Temporal constraints: For example, Han and Ker (2003) used temporal operators to specify the interaction constraints of a component relative to other components. This approach has an advantage of incremental specification of the component interaction protocols. There is also a run-time monitoring system that ensures the constraints are preserved during execution. At run-time, the operation invocations are intercepted and checked against the specified constraints. The drawback of this approach is that the source codes of the components have to be available and instrumented.

Interaction rule specification (IRS): Jin and Han (2005c) proposed a pattern-based approach to specify behavioural protocols called Interaction Rule Specification (IRS). The constraints are defined incrementally by using temporal constraints. A constraint has two parts: pattern and scope which are adopted from (Dwyer et al. 1999). Examples of pattern are: absence, existence, precedence, and response. Examples of scope are: globally, before, and after. These constraints are used to specify the required sequences of interactions.

2.3.2 Process-based descriptions

In process-based descriptions, a process specifies a complete sequence of operations invocations from the start to the end in order to achieve a business goal.

Abstract WS-BPEL process: An abstract process specifies the external message exchange between Web services and does not contain any internal details of the business process in each service. It focuses on specifying the publicly visible
messages exchanged between parties from the point of view of a single party; the internal process of each party involved in the interaction is not specified. In other words, it specifies the expected protocols between parties.

**Abstract process model:** Paolucci et al. (2004) defined an *abstract process model* of the conversation with an *exec* operation inside the mediator entity called ‘Broker’. The abstract process in *exec* operation will then be refined by the actual process model of the service at the discovery time. The notion of *abstract process model* is a versatile approach to define the conversation when the actual process models are unknown at design time. Similarly, Casati and Shan (2001) introduced *process schema* that composes of *service nodes*. Process schema defines the sequence of execution among service nodes in a process. The process schema is modelled by a graph (the flow structure) having notions such as decision nodes, parallel activities, etc., which is similar to the UML activity diagram. A service node defines the flow of method invocations on the selected service. There is also the notion of *generic service node*, which will be instantiated by a list of actual service nodes either at process instantiation time or at run-time. For example, Figure 2-1 models a ceremony process consisting of three actual service nodes: Data Collection, Restaurant Reservation, Billing; and one generic service node: Award Ceremony Services. At process instantiation time, depending on user’s selection from a service pool, the generic service node is configured. In this example, the Award Ceremony Service node is instantiated by four service nodes to be executed in parallel: Invitation, Registration, Personnel and Equipment.
2.3.3 Web Services Approaches to Behavioural Description

In SOA, there is a proliferation of specifications addressing the behavioural description of services, namely WSDL-S (Akkiraju et al. 2005), WSMO (Bruijn et al. 2005), OWL-S (Martin et al. 2004a). The main motivation of these approaches is the use of Semantic Web (Berners-Lee et al. 2001) and ontology (Noy and McGuinness 2001) to achieve computer-interpretable description, which facilitates the automated discovery and selection of services.

2.3.3.1 Web Service Semantics (WSDL-S)

WSDL-S (Akkiraju et al. 2005) was submitted to World Wide Web Consortium (W3C) in 2005. While WSDL standard operates at the syntactic level, WSDL-S aims at adding semantic annotations to the existing WSDL standard. The semantic information specified includes the definitions of the Inputs, Outputs, Preconditions and Effects (IOPEs) of Web service operations. According to Akkiraju et al. (2005), there are three main motivations of WSDL-S:
With the wide use of WSDL, adding semantic annotation to WSDL is a logical step. Users can describe, in an upwardly compatible way, both the semantics and operation level details in WSDL.

- WSDL-S uses external semantic domain models, which allows the Web service developers to annotate their Web services with their choice of ontology language (such as UML or OWL).
- It is relatively easy to update the existing tools supporting WSDL to support the annotation in WSDL-S. For example, the “Radiant: WSDL-S/SAWSDL Annotation Tool” was developed by the University of Georgia, USA (Gomadam et al. 2005)

It is not clear that WSDL-S supports the description of sequence of interactions. Currently, the sequence of interactions can be captured using preconditions by specifying the operations that are to be invoked before invoking an operation (Akkiraju et al. 2005). However, IOPEs specification is necessary but not sufficient to ensure correct interactions. The behavioural aspect in WSDL-S specification is subject to future work.

WSDL-S is still under early development. Nevertheless, it is a promising approach due to the current industrial strength and support of WSDL.

### 2.3.3.2 Web Service Modeling Ontology (WSMO)

WSMO (Bruijn et al. 2005) provides the conceptual underpinning and a formal language for semantically describing all relevant aspects of Web services in order to facilitate the automated discovering, combining and invoking services over the Web.

According to Roman et al. (2005), WSMO identifies four top-level elements as the main concepts for defining Semantic Web services:

- **Ontology element**: provides the common terminology used by other WSMO elements in order to make WSMO descriptions machine-processable.
- **Web services element**: provides a conceptual model for describing in an explicit and unified manner all the aspects of a Web service, including its non-functional properties, its functionality, and the interfaces to obtain it. This section includes the description of Inputs, Outputs, Preconditions and Effects (IOPEs) of a Web service. It further describes the *behaviour* of a service in...
both perspectives: orchestration and choreography by using an approach based on Abstract State Machines (ASM). The Web service element of WSMO is illustrated in Figure 2-2.

- **Goals**: provide the means to specify the requester-side’s objectives when consulting a Web service and describe at a high level the concrete tasks to be achieved. Goals are to be resolved by selecting from the available Web services which describes provision services that best satisfy the goals.

- **Mediators**: describe elements that handle interoperability problems between different Web services. Mediators are the core concept to resolve incompatibilities between WSMO components. WSMO includes a taxonomy of possible mediators based mainly on three levels: *Data Level Mediation, Protocol Level Mediation* and *Process Level Mediation*, i.e. in order to resolve mismatches between different used terminologies (data level), in how to communicate between Web services (protocol level) and on the level of Web services combination (process level).

![Figure 2-2: WSMO Web service – general description as from (Roman et al. 2005)](image)

In summary, WSMO provides a “heavy-weight” description which has most if not all the required information to interact with a Web service. According to Roman et al. (2005), WSMO proved to be a promising approach with a set of WSMO
compliant tools available\(^5\) and WSMO is funded by European Commission. However, WSMO places a lot of stress in the specification of mediators (Ankolekar et al. 2004). In addition, adding mediator components into the Web services infrastructure can be a burden to Web service providers when mediators can be created during the process of Web service composition.

### 2.3.3.3 Web Ontology Language for Services (OWL-S)

OWL-S (Martin et al. 2004a) is a successor of DARPA Agent Markup Language for Services (DAML-S) which was created as part of the DARPA Agent Markup Language (DAML) project. OWL-S is an ontology of services which provides a set of basic classes and properties for declaring and describing services. It is based on the ontology structuring mechanisms of OWL (Web Ontology Language).

OWL-S defines three elements to describe a service: Service Profile, Service Process Model and Service Grounding, as depicted in Figure 2-3.

![Figure 2-3: Top level of OWL-S service ontology (Martin et al. 2004a)](image)

- **Service Profile**: describes information about a service. This information is used by service consumers during service discovery. It describes three basic types of information: (i) the organisation provides a service, for example the organisation name, contact information, etc.; (ii) the capability of a service in terms of Inputs, Outputs, Preconditions and Effects (IOPEs) of the operations

\(^5\) A list of WSMO tools is available at: [http://www.wsmo.org/wsmo_tools.html](http://www.wsmo.org/wsmo_tools.html), last accessed in May 2009
of the service; and (iii) the characteristics of a service, for example the category of the service, quality rating of the service.

- **Service Process Model**: describes how to use the service. The service consumer will use the Service Process Model to control the interaction with the service. There are two types of processes that can be invoked: *atomic* and *composite*. Each process has a set of IOPEs. Ideally, IOPEs published by the Profile are a subset of those published by the Process Model. In addition, an *atomic* process has no internal structure, whereas a *composite* process consists of a set of component processes linked together by control flow and data flow structures. OWL-S also provides a *simple* process notation which is an abstracted, non-invocable view of atomic or composite process.

- **Service Grounding**: specifies the details of how to access/invoke a service including transport protocol, message formats, serialisation and addressing. A grounding maps each OWL-S atomic process to a concrete specification of those service description elements (e.g. a WSDL operation), and relates each OWL-S process input and output to elements of the XML serialisation of the input and output messages of that operation (Martin et al. 2007).

There is tool support for OWL-S, such as CMU’s OWL-S Development Environment (CODE) (Srinivasan et al. 2005) as a plug-in for Eclipse IDE\(^6\), and Protégé OWL-S Editor\(^7\). A complete list of OWL-S related tools is available at: [http://www.daml.org/services/owl-s/tools.html](http://www.daml.org/services/owl-s/tools.html). However, most of the tools are developed primarily as proof-of-concept research tools rather than robust industrial tools (Metcalf and Lewis 2006).

Metcalf and Lewis (2006) provided a useful evaluation of the capability of OWL-S. In order to utilise the potential of OWL-S during service discovery and invocation, there must be a well-defined ontology in place. OWL-S provides the foundation of automatic composition by providing the description of the semantics of service operation during discovery of services. However, this dynamic composition of

---


\(^{7}\) Protégé OWL-S Editor is available to download at: [http://owlseditor.semwebcentral.org/projects/owlseditor/](http://owlseditor.semwebcentral.org/projects/owlseditor/), last accessed in May 2009
services is not provided by OWL-S itself, it is a capability that must be implemented at the client level. Automatic composition would require more planning logic at the application layer.

In summary, OWL-S is a promising approach in describing IOPEs of a service in terms of ontology which supports dynamic discovery, composition and invocation of services. The process model of OWL-S supports description of manual composition of services into composite processes by defining control flow and data flow among processes.

2.3.3.4 Summary

It is widely recognised and agreed that the syntactic description (e.g. WSDL) of services is not enough for automated service composition. The above mentioned research aimed at adding semantic information into the service interface description by using the Semantic Web and the Web Ontology Language. Although there is great overlap in those approaches (such as all of them define semantics of IOPEs of Web services’ operations), they somewhat compensate and enhance each other.

From the above research in adding semantics to service descriptions, we have identified some requirements and evaluations of the three specifications (WSDL-S, WSMO and OWL-S) to be summed up as follows:

- **Description of IOPEs by using ontology**: Ontology facilitates computer-interpretation and automated reasoning. Ontology mappings can be utilised to evaluate concepts in different ontology domains. WSDL-S, WSMO and OWL-S all support descriptions of IOPEs by using ontology.

- **Description of a process model**: A process model describes the correct sequence of interactions of a service and also a list of other services that this service will invoke to fulfil its functionality. Service consumers need to be aware of this process model in order to invoke the service’s operations correctly. WSDL-S does not support specifying behavioural aspects of Web services.

- **Mediator/Adaptor description**: During service composition, if incompatibilities are encountered, having a mediator/adaptor specification will provide a foundation to resolve the incompatibilities. Among the three
specifications reviewed, only WSMO has the capability of specifying mediator descriptions.

- **Tool support**: Semantic Web and Ontology are designed for computer-interpretation. Even a simple process may become swamped in reefs of angle brackets that make it hard for people to read and understand what each process does (Martin et al. 2007). Tool support can save time and effort of developers to comprehend and write those descriptions. All the specifications reviewed in previous sections (OWL-S, WSDL-S and WSMO) have tool support.

- **Easy to be modified**: Services might change their functionality and process model during their life cycle. Depending on different services, changes can occur frequently or not at all. Considering the amount of information in the interface description identified in the above, it should be easy to change this information rather than the developers have to redefine it from scratch. However, the semantic descriptions reviewed above (OWL-S, WSDL-S and WSMO) do not have features that facilitate change.

Table 2-1 provides a summary of evaluation of WSDL-S, WSMO and OWL-S according to the five categories identified above.

<table>
<thead>
<tr>
<th>Standard</th>
<th>IOPE semantics</th>
<th>Behavioural protocol / Process model specification</th>
<th>Mediator/Adaptor description</th>
<th>Tool support</th>
<th>Easy to be modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSDL-S</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>WSMO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>OWL-S</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

2.4 **Formalisms for Expressing and Reasoning about Behavioural Protocols**

In addition to the different types of specifications reviewed in Section 2.3, there are approaches which use formalisms to describe behavioural protocols in order to facilitate automated analysis and evaluation. In this section, we will discuss approaches that utilise π-calculus, Petri nets and Finite State Automata (FSAs).
\textbf{\pi-calculus (Process algebra):} Canal et al. (2001) presented a method of using a subset of the \pi-calculus to describe object behavioural protocols and automated checking of protocol interoperability between CORBA objects. Although the \pi-calculus is expressive and powerful in describing the dynamic behaviour of processes, it is not practical to define all possible interactions that are allowed to happen in the interaction between two parties of interest.

\textbf{Petri nets:} Bastide et al. (1999) used Petri nets to model the behaviours of CORBA objects. The advantage of using Petri nets is that it has the capability to model concurrent behaviour of distributed system. There are existing tools to support verification and analysis of properties of the interaction such as liveness. There are also disadvantages of using Petri nets that it is lack of modularity and rigid to change.

\textbf{Finite State Automata (FSAs):} FSAs are easier to understand by software practitioners compared to other formalisms due to their simplicity. FSAs are effective in modelling sequential systems and support analysis and verification of protocol compatibility (Berardi et al. 2004; Canal 2004).

Due to its simplicity and effectiveness, we will utilise FSA as the formalism to express behavioural protocols in this thesis.

Finite State Automaton (FSA) is a model of behaviours consisting of states and transitions between those states. The transitions signify the allowable operations at those given states in a behavioural protocol specification. Final states denote acceptable states where an interaction can terminate correctly. It is desirable that after a set of operations was carried out, the FSA will reach a final state. Figure 2-4 shows a simple example of a Book Vendor’s Web service’s behavioural protocol FSA. The state with a triangle denotes the initial state, and double circles denote final states. In this example FSA, to interact with this Web service, it is required to obtain a quote first (the \texttt{quote} operation). The interaction can terminate here as the quote might be too high and customers do not proceed with an order, otherwise an order can be placed. Once the \texttt{order} operation is invoked, the service is expected to receive delivery detail and payment. Finally a delivery is dispatched, at which point the interaction will terminate successfully.
In the literature, there are approaches utilising FSA to specify behavioural protocols. For example, the COCOA (C Onversation-based service COmposition middlewAre) framework (Mokhtar et al. 2005; Mokhtar et al. 2006) used FSAs to define a sequence of user tasks. These user tasks’ FSAs will serve as required sequences of interactions. Wombacher et al. (2004a) proposed annotated FSA where each state might be assigned a propositional logical expression such as AND, OR. Benatallah et al. (2003; 2004) utilised FSA to define sequence of interactions. The authors added extra properties to the transitions between states. There are three additional properties of transition: activation, compensation and locking. An activation property describes the triggering features of a transition such as explicit transition from a service invocation or an implicit transition after a certain amount of time has elapsed. A compensation property specifies the effect of a transition and this effect can be undone by explicitly invoking a specified compensation operation. A locking property specifies temporary reservation of service resources for a service consumer when invoking a transition.

Formalisms such as π-calculus, Petri nets, FSA provide automated analysis and reasoning of behavioural protocols. It is often desirable to convert protocol descriptions into those formalisms so that verification can be performed.

Foster et al. (2003) described a set of steps to specify properties of a sequence of interaction, realise the interaction in a composition language such as WS-BPEL, and then verify the correctness of the WS-BPEL specification. The detailed description of those steps is as follows:

- Convert WS-BPEL to Finite State Processes (FSP). FSP is a textual notation for finite state machines developed by Magee and Kramer (1999). FSP is machine readable and allows automated analysis by tools such as the Labelled Transition System Analyser (LTSA).
- Use Labelled Transition System Analyser (LTSA) to verify whether the WS-BPEL FSP conforms to the required FSP.
Mokhtar et al. (2005; 2006) proposed a technique of converting OWL-S process models to FSAs. An OWL-S process is composed of control constructs such as sequence, choice, repeat-while, repeat-until, choice, split, split + join (more details in Section 2.3.3.3). Each OWL-S construct is mapped to an FSA. Then these FSAs are linked together to form a complete FSA representing the behavioural protocol. The resulting FSA is then served as a basis for composition of Web services to realise a user task.

Besides FSA, there are approaches that use other formalisms, such as Petri nets. Narayanan and McIlraith (2002) converted OWL-S description to Petri nets in order to perform analysis. The authors use the properties of Petri nets to verify reachability, liveness and existence of deadlocks in the process model described in OWL-S specification. Apart from verification, the proposed framework also allows user to simulate the execution of the service under different input conditions.

Ankolekar et al. (2005) described a mapping of OWL-S process model statements into equivalent PROMELA (Process Meta Language) statements. The authors defined a PROMELA process for each OWL-S control construct. The data flow is also modelled by PROMELA processes. The PROMELA processes are then evaluated by the SPIN model checker (Holzmann 2003) to check for liveness, absence of deadlocks and correctness of dataflow.

In summary, there are various approaches to specify and verify behavioural specifications by using different formalisms in order to provide automated analysis and reasoning about behavioural protocols. In this thesis, we utilise FSA as the underlying formalism to express the required sequences of interactions in a composition (behavioural protocol). Web services’ behavioural descriptions have to match this required behavioural protocol in order to be used in the composition. We based on the work of Mokhtar et al. (2005; 2006) to convert Web services’ behavioural descriptions in OWL-S to FSAs in order to perform automated analysis of Web services’ compatibilities in terms of behaviour to be used in a composition.
2.5 Match-making of Services

Having sequences of interactions defined in service interfaces, the next step is to develop match-makers that are aware of this extra information and correctly match advertised services against the service client’s requirements. Generally, services are registered in a service registry. A service registry allows businesses to register their Web services and allows service clients to search for suitable services. Match-making processing can be performed either on a centralised server (acting as a service registry) or on individual service consumers (when a service requester obtains the semantic enriched description of eligible services from a service registry and performs the matching process on its own) (Jaeger et al. 2005). However, in terms of the matching techniques, there is a great overlap between them; and generally, approaches in one area can be applied to the other one.

Theoretically, in order to be used in a composition, advertised services have to match exactly to the requirements at all levels identified in Section 2.3, namely syntactical level, behavioural level, synchronisational level, and Quality of Service (QoS) level. Medjahed et al. (2003) defined categories which aim at addressing all four levels. Each of the above levels is worth an entire research area in its own right. In this review section, we only concentrate on capability and behavioural matching, as this is the main area of research in this thesis.

2.5.1 Capability Matching

Capability matching is done by subsumption analysis of the semantics of the Inputs, Outputs, Preconditions and Effects (IOPEs) between the required capability of a service client and the advertised services’ capabilities with an assumption that mappings between different ontology systems exist. Paolucci et al. (2002) and Sycara et al. (2003) defined a match-making process that evaluates required inputs and outputs against advertised services’ inputs and outputs based on ontology subsumption. An advertised service satisfies the requested capabilities when (i) the outputs of the advertised service (outAdv) are equivalent or more general than the outputs of the request (outReq), i.e. the advertised service provides all the information that the requester needs, and (ii) the inputs of the request (inReq) are equivalent or
more general than the inputs of the advertised service ($inAdv$). The match-making process classifies the compatibilities into four categories:

- **exact**: when $outAdv = outReq$ and $inReq = inAdv$. The “=” means equivalent relation. For example, both outputs of the advertised capabilities and of the required capability refer to a *Car* concept in some ontology.

- **plugIn**: when ($outAdv$ subsumes $outReq$) and ($inReq$ subsumes $inAdv$). “Subsume” means more general. For example the *Car* concept subsumes the *Sedan* concept.

- **subsumed**: when ($outReq$ subsumes $outAdv$) and ($inAdv$ subsumes $inReq$). The advertised outputs are more specific than the outputs of the request, hence the advertised service can provide only some specific cases of what the requester desires (Paolucci et al. 2002).

- **failed**: when none of the above relations exists between the advertisement and the request.

The order of preference is: *exact* > *plug-in* > *subsumed* > *failed*. This subsumption analysis and variations of the above four degrees of matching are applied to a great extent in the literature (Martin et al. 2004b; Majithia et al. 2004; Jaeger et al. 2005; Mokhtar et al. 2006).

### 2.5.2 Sequence of Interaction Matching

In addition to the capability matching, the behavioural protocols of services also need to be checked against the required sequence of interaction. The work in Mokhtar et al. (2006) utilises FSA for behavioural matching between a specified user task against services’ behaviours. In their work, a required sequence of interaction is described as a user task’s FSA. A user task is realised by matching a fragment of user task’s FSA to a fragment of services’ behavioural protocols. Firstly, the capability required in the user task’s description is matched against advertised capabilities of services based on IOPEs as discussed in Section 2.5.1. The second step towards composition of a user task is the integration in terms of behaviour with the selected services. Behavioural protocols descriptions of services (e.g. in OWL-S) are converted to FSAs. Then the COCOA Conversation Integration (COCOA-CI) mechanism will analyse the FSA of the user task and match fragments of this FSA to services’ behavioural FSAs. The services with matching fragments are then composed to realise the user task. This
approach uses FSA in order to translate the problem of integrating conversations to an automata analysis problem (Mokhtar et al. 2006).

Wombacher et al. (2004a) performed matching of behavioural protocols by using an FSA intersection algorithm. Two FSAs match if their languages have a non-empty intersection. Their approach uses annotated FSA where each state can have a propositional logic expression. The original FSA intersection algorithm cannot be used as is; thus, the authors defined intersection algorithm for annotated FSA. Wombacher et al. (2004b) presented a translation of WS-BPEL to annotated FSA in order to match process specification in WS-BPEL to available services’ behavioural protocols.

Also using FSA, Benatallah et al. (2006) proposed a compatible composition operator which takes two protocols as inputs, one from a requester and the other from a potential service. It then computes a protocol called a compatible composition protocol, which describes the set of complete interaction trees between the two input protocols. The algorithm is based on the FSA intersection algorithm and it takes into consideration the mappings from output messages of one protocol to input messages of the other protocol. The direction of messages is called polarity where positive polarity (+) signifies incoming messages and negative polarity (-) signifies outgoing messages. If the result of a compatible composition protocol is empty, this means that no conversation is possible between the requester and the service. Otherwise, the compatible composition protocol is the identification of possible interactions between them.

During the matching of sequence of interaction, it is important to distinguish between compatibility and substitutability (in (Benatallah et al. 2006) the authors used the term replaceability), which are the two important aspects of the component interoperability (Vallecillo et al. 2000).

- **Compatibility:** is the ability of two components to work properly together if connected, i.e. all exchanged messages and data between them are understood by each other (Vallecillo et al. 2000). In the SOA context, Bordeaux et al. (2004) defined that two services A and B are compatible if they have opposite behaviours, i.e. in some reachable pair of states in their behavioural protocols,
a set of messages one service can send out match exactly to the set of messages the other can receive.

- **Substitutability**: is whether one protocol P can be replaced by the other protocol P’). In the SOA context, Bord eaux et al. (2004) defined that in a particular application made of two compatible services A and B, service A’ can substitute service A if A’ is also compatible with B. Substitutability checking is very important during evolution and maintenance of a service (Vallecillo et al. 2000; Bordeaux et al. 2004). When changing a service, we need to ensure that this change does not affect service consumers that are currently relying on this service.

### 2.5.3 Summary

In order to interact with each other, services have to match both in terms of capability and sequence of interaction (behaviour). Capability matching involves checking semantics of IOPEs of service requester against those of advertised services. Sequence of interaction matching involves checking that messages sent from one service are of correct sequence that the other service expects.

In an ideal case, a service consumer can find an advertised service that matches exactly in both required capability and sequence of interaction. However, most of the advertised services were developed without being aware of potential consumers. Therefore, it is unrealistic to expect to find an exact match to the consumer’s requirement. In the dynamic SOA context, advertised services can become unavailable, or change their behaviours should their implementation changed although not often. It is important to have adaptation mechanism to address and mitigate mismatches between service consumer’s requirements and advertised services. This will extend the number of services that a service can interact with to achieve its goal. The next section will review adaptation approaches.

### 2.6 Mismatch Adaptation

An adaptor is a module that sits between two components and compensates for the differences between their interfaces (Yellin and Strom 1997). As discussed in Section 2.3, there are four levels of interface specification namely *syntactic, behaviour,*
Chapter 2  Background and Literature Review

synchronisation, and Quality of Service (QoS). Adaptors need to resolve mismatches at all those levels for two services to be able to interact with each other. Adaptation at each level is a big research area in itself. As our research focuses on the adaptation at behavioural level, in this section we will concentrate on reviewing existing adaptation approaches at such level.

2.6.1 Adaptation at Behavioural Level

Yellin and Strom were considered to be one of the pioneers to formally describe and tackle the problem of behavioural mismatch adaptation. In (Yellin and Strom 1997), the authors proposed method of synthesising adaptors from interface mappings. An interface mapping consists of a set of rules which relate messages and parameters in the two interfaces. From the mapping rules, the adaptor synthesis algorithm will either produce a valid adaptor or will correctly conclude that no such adaptor exists. The interface mapping requires human intervention to create and/or verify its correctness. Therefore, this approach only provides semi-automated adaptation.

Motahari-Nezhad et al. (2007) adopted this notion of interface mapping between component interfaces for Web services adaptation. The authors extended the original work from Yellin and Strom (1997) by proposing a way to identify and handle deadlocks. A state is a deadlock state if there is no possible message exchange between the services in that state. When a deadlock is reached, an analysis is performed by assuming that each of the messages engaged in the deadlock can be generated, then it records how the message exchanges between two protocols proceed until reaching final states. If another deadlock is reached, the same analysis is carried out. Based on this analysis, a mismatch tree is built. A mismatch tree represents all possible deadlocks between two protocols, and the messages that are engaged in each deadlock. A suggestion deadlock resolution is then generated. The combination of the tree representation and the suggestions for deadlock resolution assist the user in the decision makings leading to the generation of the final adaptor. The advantage of this technique is that it takes possible future messages into consideration in order to resolve the current deadlock while minimising the number of deadlocks in the future.

Tao and Yang (2008) proposed a concept which allows a single service to have different interface variations to meet different interaction requirements. A service’s
process is an “abstract business process (ABP)” when it consists of both concrete activity and abstract activity. Concrete activities remain unchanged for all usage contexts, whereas abstract activities are bound to specific usage contexts defined by different policies. Depending on policies of the current usage context, abstract activities are replaced with different tasks; as a result, differentiated functions can be provided to users. The mappings between abstract activities to concrete tasks for particular policies are defined in context configured business processes (CCBPs).

From the list of concrete tasks, different service interfaces can be generated and presented to users of the service. This technique provides a mechanism to adapt interaction mismatches by changing and presenting different service’s interfaces to meet interaction requirements. However, how to identify mismatches and how to overcome them are not discussed.

Brogi and Popescu (2006) proposed an adaptation framework based on YAWL (Yet Another Workflow Language) (van der Aalst and Ter Hofstede 2005). The framework attempts to adapt two WS-BPEL processes, called C (consumer) and S (service). There are four phases in the adaptation process:

- **Service Translation:** This phase is in charge of translating the WS-BPEL processes of C and S into corresponding YAWL workflows.

- **Adaptor Generation:** Firstly the service execution trees (SET) of C with respect to S and of S with respect to C are generated from their YAWL workflows. Next, the SET of the adaptor is obtained by suitably merging the duals of the above two SETs. (A dual of a SET of a service X is a SET satisfying that when X outputs a message, there is a corresponding input of that message in the dual SET and vice-versa). Finally, the YAWL workflow of the adaptor is derived from the adaptor’s SET.

- **Lock Analysis:** this phase verifies whether the YAWL-based aggregation of C, adaptor and S has any lock or not. If it does, the adaptation has failed. Otherwise, it is successful.

- **Adaptor Deployment:** If the adaptation is successful, this phase deploys the YAWL workflow of the adaptor as a WS-BPEL process, which can be used as a service-in-the-middle between C and S.
The job of an adaptor is to mediate between two protocols (Benatallah et al. 2005). This requires performing activities such as receiving messages, storing messages, transforming message, and invoking service operations. Benatallah et al. (2005) proposed different adaptor templates in order to generate adaptors to resolve various mismatch patterns at the behavioural protocol level: message ordering, extra message, missing message, message split and message merge.

- **Message Ordering Mismatch**: is classified as Ordering Constraint Pattern. The adaptor can temporarily store given information and forward the information to services when they are ready to accept it.

- **Extra Message Mismatch**: is classified as Extra Message Pattern. The adaptor will discard extra messages that are not expected by the intended recipient. It should be noted that such an adaptation makes sense only if the discarding extra messages does not affect the semantics of the intended recipient’s protocol. Typical messages that can be discarded are acknowledgement messages when the intended recipient does not need it. This pattern corresponds to the Hide operator in (Dumas et al. 2006).

- **Missing Message Mismatch**: is classified as Missing Message Pattern. The adaptor will generate messages based on available information. This is a reverse of Extra Message Pattern. It is also noted that the extra messages should not affect the semantics of the protocol. An example of this is when a service does not explicitly send an acknowledgement, while the service requester requires. The adaptor will base on the current interaction and generate an acknowledgement message to send to the service requester. In (Dumas et al. 2006), the authors avoided introducing any operators that handle this pattern where an action from the target protocol is needed but is not provided by the source protocol. It is argued that this pattern requires the introduction of business logic in the adaptor, which is undesirable from a software maintenance perspective.

- **Message Split Mismatch**: is classified as One To Many Messages Pattern. It is required to have XQuery functions for parameters extraction in order to generate adaptors. The adaptor will split the information given in one message to many messages as required by the recipient. This pattern corresponds to the Scatter operator in (Dumas et al. 2006).
• **Message Merge Mismatch:** is classified as Many To One Message Pattern. It is required to have XQuery functions for parameter computation in order to generate adaptors. The adaptor will combine information given in multiple messages to generate one message as required by the recipient. This pattern corresponds to the Gather operator in (Dumas et al. 2006).

Benatallah et al. (2005) do not address the problem of automatically identifying differences between two protocols and how to classify the differences into the above mismatch patterns. In addition, the XQuery functions used for splitting or merging information have to be specified.

Wu et al. (2003) proposed a method of service composition by using SHOP2 planner which is a Hierarchical Task Network (HTN) planner. The HTN planning is an Artificial Intelligent planning methodology that creates plan by task decomposition. A given task is decomposed into smaller and smaller subtasks, until primitive tasks are found that can be performed directly. The service composition is carried out by converting OWL-S descriptions of services to SHOP2. The user’s request is then analysed and realised by a composition of SHOP2 tasks into a plan. Then this SHOP2 plan will be converted to OWL-S which can be directly executed by an OWL-S executor.

### 2.7 Summary

In this chapter, we discussed the background of Service Oriented Architecture and service composition to reuse existing services in order to realise user tasks. However, manual service composition is laborious and does not scale well. Automated service composition was introduced to address these shortcomings. We reviewed various areas required by automated service composition: (i) adding behavioural protocol specification to services’ interface descriptions; (ii) using formalisms to express and reason about behavioural protocols; (iii) match-making of behavioural requirements against available services; and (iv) adapting the mismatches identified in the match-making process.
The subsequent chapters will present our research efforts in addressing the above-mentioned areas of automated service composition in an adaptive service composition framework.
Chapter 3:
Preliminaries

In the previous chapter, we reviewed the research effort in service composition addressing various areas: adding behavioural specification to service interface description, using formalisms to specify and reasoning about behavioural specifications, match-making of services against specified requirements, and mismatch adaptation.

In this chapter, we present a preliminary which forms the basis for further discussion in later chapters. In particular, Section 3.1 provides motivating scenarios highlighting the need for a semi-automated service composition; Section 3.2 presents the Role Oriented Adaptive Design (ROAD) framework that our proposed approach to service composition framework is based on. In Section 3.3, we present how behavioural protocols can be specified by using Interaction Rule Specification (IRS). The IRS has Finite State Automaton (FSA) as its underlying semantics. In Section 3.4, we show how IRS constraints are mapped to FSAs. Finally, the chapter is summarised in Section 3.5.

3.1 Motivating Scenarios

In this section, we present scenarios of a library that buys books from different book vendors. In order to streamline the ordering process, the library book ordering system (lib-BB, which stands for “library book buyer”) will need to interact with various book vendors’ Web services (vendor-WS) to ask for quotes and to order books. For the lib-BB to successfully interact with various vendor-WS, clearly the book vendors’ WS have to match the requirements of the lib-BB at four different levels as discussed in Section 2.3 namely syntactic, behaviour, synchronisation and Quality of Service. In particular, we focus on the matching of interaction behaviour, i.e. the sequences of
interactions for example payment before delivery, or delivery before payment. There can be different types of mismatches between the behavioural protocol expected by the lib-BB (behavioural requirements) and the one provided by the vendor-WS. Figures 3-1 to 3-4 show some mismatching scenarios with the behavioural protocols depicted in finite state automata (FSA), where a state with a triangle indicates an initial state and a double circle indicates a final state. The final states denote the points at which the transactions can terminate correctly. In general, the two interacting parties should reach final states at the end of a transaction. If a final state cannot be feasibly reached, there is some mismatch(es) in their behavioural protocols. The behavioural mismatches are highlighted by the squares enclosing the operations in figures 3-1 to 3-4.

- **Scenario 1** (Figure 3-1): The lib-BB will retrieve a quote for books from the vendor-WS-1. If it accepts the quote, it will place an order for those books and make a payment before delivery detail is sent. However, our book seller expects to receive the delivery detail before accepting payment.

- **Scenario 2** (Figure 3-2): The requirement of the lib-BB is still similar to Scenario 1; however in this scenario, the vendor-WS-2 does not do delivery. The purchasers are required to come to the store to pick up the books.

- **Scenario 3** (Figure 3-3): The vendor-WS-3 has all the functionality required by the lib-BB. However it requires an explicit quote acknowledgement whether the buyer accepts the quote and continues with an order, or ends the
transaction with a quoteNotAccepted message. Although the protocols are quite similar, it should be noted that the two protocols do not share any common sequence of operations from the initial state to a final state. The mismatch arises from incompatibilities in protocols rather than Vendor-WS-3 has extra functionality, i.e. quoteAccepted and quoteNotAccepted operations.

![Diagram](image)

**Figure 3-3: There is no valid sequence of interaction**

- **Scenario 4** (Figure 3-4): Similar to Scenario 3, but in this scenario, the book buyer requires its partner to send receipts of the payments for tax purposes, which is not supported by the vendor’s WS. In addition, the book buyer changes its behavioural protocol requirement; it now requires receiving delivery before it makes any payment, while the service expects payment before delivery. Unlike scenario 1, this mismatch is not merely a difference in the order of messages, but arises from incompatible at the business level of the respective parties.

![Diagram](image)

**Figure 3-4: Behavioural mismatches at interaction levels and missing of functionality**

Currently, to identify the services with matching behavioural protocol, developers/designers have to perform a manual check. This may be time consuming particularly if the number of Web services is large. If the lib-BB changes its behavioural protocol requirements (although it should only happen rarely), the entire
process will need to be carried out manually again. To address this issue, this thesis presents our approach to provide automated protocol compatibility checking and semi-automated protocol mismatch adaptation for use in service compositions. Our approach is built upon and extends the Role Oriented Adaptive Design (ROAD) framework (Colman 2006; Colman and Han 2007). The next section will introduce the ROAD framework.

### 3.2 The Role Oriented Adaptive Design (ROAD) Framework

Software systems are becoming more open, distributed, pervasive, mobile and connected (Colman 2006). The environments where these software systems are operating are becoming more diverse and dynamic, in which existing components might become unavailable or new components are introduced. In addition, the performance of individual component might also be changing. In order to sustain pre-defined goals and performance, software systems will need to be able to adapt to these changing environments. In this context, the ROAD framework (Colman 2006; Colman and Han 2007) provides an approach to adaptive architecture. The adaptability that the ROAD framework provides makes it a suitable candidate for modelling and implementing behavioural adaptation.

In ROAD, components of a software system can be changed dynamically. These components are governed by their relationships with other components. Relationships between components are modelled as first class entities which contain predefined functional and non-functional requirements. These relationships will be monitored and controlled at run-time in order to maintain the system’s viability.

The ROAD framework models a software system as an organisational structure consisting of roles, players (that play specific roles) and contracts (relationships between roles). These elements constitute a self-managed composite which is maintained by an organiser. An organiser manages the bindings between roles and players (known as indirection of instantiation) and creates/destroys contracts between roles (known as indirection of association). The next sections will describe these elements in details and show how adaptability can be achieved.
3.2.1 Role and Player

Roles or functional roles (as distinct from organiser-roles, see Section 3.2.3) are abstract functional and non-functional descriptions that require players to fulfil. A player is a self-contained component that is able to perform some functionalities (e.g. objects, components, services, agents, humans, etc).

The difference between a role and a player is that functional roles define a ‘position’ within an organisation, while role-players “do the work” (Colman 2006). In order to be able to play a role, a player has to satisfy the functional and non-functional requirements of the role. These requirements are defined by the relationships (or contracts) the role has with other roles (see Section 3.2.2). A role can only be played by one player at a time; however a role can be played by various suitable players at run-time. Players do not interact with each other directly, but via their roles. There are two main interaction types in a system: role-to-role interaction and role-and-player interaction. To illustrate this role-player relationship, consider a university having a Lecturer role. The Lecturer role contains required functionality of preparing lectures, delivering lectures and marking assignments and exams as specified in the relationship between Lecturer and Student roles. The Lecturer role will be played by a human who has qualifications and is able to perform those functionalities required by the role.

To model the scenarios in Section 3.1 in the simplest way, there are two roles: Buyer and Vendor roles. Buyer will be played by the lib-BB, whereas Vendor will be played by various book services such as Amazon’s Web service. This is illustrated in Figure 3-5.

The lib-BB and vendor-WS do not interact with each other directly but via their roles. For example, the lib-BB sends a quote request to Buyer role, the Buyer role then sends that request to the Vendor role. The request is fulfilled by the Vendor role’s player (vendor-WS). The quote response is then sent back to the Buyer role and subsequently to lib-BB.
3.2.2 Contract

Contract is a first class entity that represents relationships between roles. A contract is a rich connector between two roles. More than just a binding, it stores the required sequence of interactions (behavioural protocol) between roles and provides a mechanism to intercept the messages exchanged between two roles during run-time in order to verify the interaction. Any messages that do not follow the behavioural protocol will be rejected. In addition, contract stores pre-defined non-functional requirements of each party with respect to those interactions, measures and compares the actual performances against the pre-defined ones to check for compliance. Contract thus encapsulates both the coordination and performance management of interactions (Colman 2006). Contract achieves these tasks by the following elements: Control Communicative Act (CCA), Term and ProtocolClause.

3.2.2.1 Control Communication Act (CCA)

CCAs are used to define the control communication in contracts in terms of allowable communication between roles in a contract. CCAs abstract the control aspects of the communication from the functional aspects (Colman and Han 2007). CCA defines message types that can be exchanged between roles, abstract away from the lower level detail of communication between roles, such as method of communication (whether sending messages or method invocation), operation name and parameter values. In this sense, ROAD’s CCA is one level lower than the communicative act defined in Williams et al. (2005). Williams et al. define communicative act as a single domain level communication, but may correspond to an exchange of one or more lower-level messages or Web service operations. For example, in Williams et al. (2005), the informReadyForCollection communicative corresponds to request and confirmation messages, with time-out retries of request messages. This
communicative act concept corresponds to our Term concept (see Section 3.2.2.2). Table 3-1 represents a simple set of CCA message types to model the scenarios in Section 3.1.

Table 3-1: A simple set of CCA message types for the Library Book Buying Scenarios

<table>
<thead>
<tr>
<th>CCA</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuoteRequest</td>
<td>A role requests a quote from the other role.</td>
</tr>
<tr>
<td>QuoteResponse</td>
<td>A role replies with a quote to the other role.</td>
</tr>
<tr>
<td>Order</td>
<td>A role places an order for books to the other role.</td>
</tr>
<tr>
<td>Ack</td>
<td>A role replies with an acknowledgement of either success or failure to the other role.</td>
</tr>
<tr>
<td>Payment</td>
<td>A role sends payment detail to the other role.</td>
</tr>
<tr>
<td>DeliveryDetail</td>
<td>A role sends delivery detail to the other role.</td>
</tr>
</tbody>
</table>

3.2.2.2 Term

CCAs define only general message types. A contract associates two roles, partyA and partyB of the contract. Messages initiated from partyA have direction AtoB (from partyA to partyB), and vice versa for messages initiated from partyB i.e. direction BtoA (from partyB to partyA). A term defines a domain level communication between two parties. It is a sequence of valid CCAs taking direction of messages into account. The directions of CCAs are important because in a valid contract, each party can only be able to send certain message types to the other party. For example, in a Buyer-Vendor contract, Buyer can send QuoteRequest message requesting for a quote to Vendor, but Vendor cannot. A term only define a sequence at an elementary level and does not allow alternatives in its sequence. For example, a term consisting of two CCAs: QuoteRequest and QuoteResponse, the term is completed when both QuoteRequest and QuoteResponse were exchanged in that order and had the right directions. If the QuoteRequest CCA was not previous exchanged and the QuoteResponse CCA is encountered, it will be rejected.

In addition, terms and CCAs do not consider parameters of messages. In later sections, we will show how parameters are used to correlate messages from the same instances of a behavioural protocol at run-time.

ROAD’s term is similar to communicative act defined in Williams et al. (2005). However, a term defines both valid message types and directions of messages. Table 3-2 represents a simple set of terms to model the scenarios in Section 3.1, where
partyA is the Buyer role and partyB is the Vendor role. In this table, it is shown that a term is composed of two CCAs; however, a term can have any number of CCAs, an example could be that a term consists of four CCAs: Request, Response, Commit, and Acknowledgement.

**Table 3-2: A simple set of Terms for the Library Book Buying Scenarios**

<table>
<thead>
<tr>
<th>Term</th>
<th>CCA</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quote</td>
<td>QuoteRequest</td>
<td>AtoB</td>
</tr>
<tr>
<td></td>
<td>QuoteResponse</td>
<td>BtoA</td>
</tr>
<tr>
<td>Order</td>
<td>Order</td>
<td>AtoB</td>
</tr>
<tr>
<td></td>
<td>Ack</td>
<td>BtoA</td>
</tr>
<tr>
<td>Payment</td>
<td>Payment</td>
<td>AtoB</td>
</tr>
<tr>
<td></td>
<td>Ack</td>
<td>BtoA</td>
</tr>
<tr>
<td>DeliveryDetail</td>
<td>DeliveryDetail</td>
<td>AtoB</td>
</tr>
<tr>
<td></td>
<td>Ack</td>
<td>BtoA</td>
</tr>
</tbody>
</table>

Based on valid CCAs and directions, a set of terms in a contract also defines the *required functionality* of each party involved. For example, in Table 3-2, partyB (which is a Vendor role) is required to have capability to process all messages of direction AtoB, which are of CCA types QuoteRequest, Order, Payment and DeliveryDetail.

Each term has zero or more non-functional requirements associated with it, for example response time, security requirement, price, quality of service, etc. In order to monitor these performance parameters, each term might have one or more *utility functions* that measure the performance of players currently playing the roles involved in the contract. If underperformance is detected, the contract is notified so that remedy actions can be carried out.

### 3.2.2.3 Protocol Clause

A term only defines a simple sequence of CCAs. A Term is a domain level communication consisting of a few message exchanges. Terms do not allow alternatives in its sequence. To specify the sequence of terms, *protocol clauses* are used. Protocol clauses define the behavioural protocol of a contract as a sequence of terms to be followed by the parties. At run-time, interactions between roles will be intercepted by their contract and are verified against pre-defined protocol clauses.
An example of protocol clauses of the terms defined in Table 3-2 is:

- Quote occurs before Order, but Order might not necessarily occur (as the quote might be too high)
- Order is to be followed by Payment
- Payment is to be followed by Delivery

### 3.2.2.4 Summary

The Contract concept can be summarised as an UML diagram in Figure 3-6.

![UML Diagram](image)

**Figure 3-6: Summary of the Contract concept**

#### 3.2.3 Composite and Organiser

A *composite* is a cluster of functional roles and their relationships (contracts). Players can be part of a composite (internal players); however they can be outside of a composite (external players) such as an external service. A composite can be a player of a role in a bigger composite, thus forms a recursive structure of a role-based organisation (Colman and Han 2007). This is illustrated in Figure 3-7. In Figure 3-7, the Library composite contains three roles representing three activities of the library: BookLoaner has the functionality to loan books, BookReturner has the functionality to process a return of books, and BookBuyer has the functionality to order new books for the library. BookReturner is contracted to BookLoaner in order to finalise previous loan, and BookLoaner is contracted to BookBuyer so that any unavailable books will be ordered at some stage. All those three roles are played by internal players; however player of BookBuyer role is a Library Book Buying composite which contains Vendor and Buyer roles. Both players of these two roles are external to the composite as shown in the figure.
Each composite has an *organiser-role*. An organiser-role provides an overall management over roles and contracts within its composition. An organiser-role can create new or destroy existing role instances. It can create and revoke contracts between roles. In addition, it can bind or unbind roles and their players.

An organiser-role also provides a management interface that allows the non-functional requirements of the composite to be set. The management interface is also a communication channel to pass non-functional requirements from a higher composite to lower ones. Upon receiving non-functional requirements from a higher level of management, the organiser-role of a composite will set the non-functional requirements to its contracts. These new requirements will be used during monitoring process and a notification will be generated if underperformance is detected. This is
illustrated as big dotted arrows in Figure 3-7. An organiser-role is also a management interface of composites. Organiser of a composite will communicate the required settings (e.g. non-functional requirements) to organisers of the lower level composites, and it will report back to organisers of the higher level composites (e.g. current performance parameters). Thus, organisers form dependency relationships between composites.

An organiser-role itself is a role, and it is required to have a player to perform its functionality. The intelligence of a player of an organiser-role knowing how to break a high level non-functional requirement to lower level obligations that are written into contracts, when to create new roles and contracts as well as when to destroy them, how to react to performance violation, etc. depends on its capability. A player of an organiser-role can be an artificial intelligent agent having access to a knowledge base, or a human being. Typically, an organiser player is a person/entity or a software system that owns and manages the composite.

3.2.4 Flexibility and Adaptability of ROAD Framework

In the ROAD framework, relationships between entities (role-to-role, or role-to-player) are dynamically defined. Different players can perform functionality of a role at different time. The functionality of a system is defined by roles and their relationships (contracts). Which player actually carries out the pre-defined functionality (i.e. players) is unknown and can be changed. This is particularly suitable to be applied in SOA, mobile and pervasive systems where components are not always available and new components are dynamically discovered.

Non-functional requirements are defined and embedded in contracts. At run-time, if any terms violate these requirements, the organiser-role will be notified and mitigation strategy will be carried out. The mitigation strategy could be: looking for new players, utilising extra players that are available by creating new roles, or re-negotiate the previously defined non-functional requirements.

The flexibility and adaptability of the ROAD framework are the reasons we adopt it as the underlying framework on which to build our service composition framework.
3.3 Behavioural Protocol Description

The original ROAD framework (Colman 2006) defines extension point where behavioural protocols can be defined (Term and Protocol Clause), however the exact format and mechanism to support behavioural protocols have not been previously defined. One of the contributions of this work is to provide this extension to the ROAD framework so that behavioural protocols can be supported. This is achieved by defining behavioural protocol description in a formalism called Interaction Rule Specification (IRS). In this section, we will introduce IRS in detail.

Interaction Rule Specification (Jin and Han 2005a) is a declarative pattern-based approach to specify behavioural protocol constraints. Declarative approaches have certain advantages over imperative approaches. According to Gottesdiener (1997), imperative languages, such as C and C++, describe a set of rules in terms of sequence of actions; whereas declarative languages, such as Prolog, describes ‘what’ the rules are, rather than ‘how’ they are enforced. As such, declarative approaches create simpler descriptions, requires one-third to one-sixth of the number of statements to represent rules compared to imperative approaches (Gottesdiener 1997). In addition, declarative description is relatively easier to understand and modification requires less effort (Li et al. 2005).

IRS is an extension of Specification Pattern System (SPS) (Dwyer et al. 1999). The SPS provides a “high level specification abstraction” to formally specify temporal properties of system’s interactions. This temporal approach has some advantages over other formalisms (e.g. π-calculus, Petri nets) in that we only need to define the order of occurrence or absence of the events of interest (Jin and Han 2005a). Dwyer et al. (1999) show that SPS patterns can be directly mapped to formalisms such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and Quantified Regular Expression (QRE). SPS enables practitioners who are not experts in those formalisms to read and write formal specifications.

In IRS, each pattern specifies a recurring occurrence or absence of events in an interaction protocol. A behavioural protocol is represented by a collection of patterned rules, where each rule represents an interaction constraint of the behavioural protocol. Each rule is specified separately and can be independent from other rules. Thus, this approach enables incremental specification; and modification is carried out
by merely adding or removing rules, which is relatively easier than other imperative approaches.

Given the above advantages, we have adopted IRS to specify interaction constraints of behavioural protocols in the contracts between roles. The next sections detail the structure of IRS constraints.

### 3.3.1 Patterns

As discussed previously, SPS is a declarative pattern-based approach to specify interaction constraints. SPS uses two main patterns: *occurrence* and *order* patterns. Occurrence patterns specify the occurrence of a given event/operation during system execution. Order patterns specify the relative order in which multiple events/operations occur during system execution. Figure 3-8 illustrates the hierarchy of patterns as defined in Dwyer et al. (1999).

![Figure 3-8: The pattern hierarchy (Dwyer et al. 1999)](image)

**Occurrence** patterns:
- **Absence**: describes the absence of an event/operation.
- **Existence**: describes the existence of certain events/operations.
- **Universality**: describes the existence of an event/operation that is desirable. This pattern is often used to look for the existence of a positive event/operation, whereas the Absence pattern is used to look for the absence of a negative event/operation.
- **Bounded Existence**: describes a number of times an event/operation can occur at most or at least.

**Order** patterns:
- **Precedence**: describes relationships between a pair of events/operations where the occurrence of the first event/operation is a necessary pre-condition for an occurrence of the second one.
- **Response**: (also known as leads to) describes cause-and-effect relationships between a pair of events/operations. The first event/operation is ‘the cause’; the second event/operation is ‘the effect’. An occurrence of the first event/operation must be followed by an occurrence of the second one.

- **Chain Precedence**: describes precedence relationships of two sequences of events/operations, for example, a sequence of events/operations P₁,...,Pₙ must be preceded by a sequence of events/operations Q₁,...,Qₙ. This pattern is a generalisation of the Precedence pattern.

- **Chain Response**: describes cause-and-effect relationships of two sequences of events/operations, for example, a sequence of events/operations P₁,...,Pₙ must be followed by a sequence of events/operations Q₁,...,Qₙ. This pattern is a generalisation of the Response pattern.

For example, in the Library Book Buying scenarios presented in Section 3.1, the Vendor has a constraint that “a quote has to be obtained before ordering”. This constraint can be specified as: “Vendor.quote precedes Vendor.order”. Note that we cannot specify that “Vendor.quote leads to Vendor.order”, as the event of ordering might not occur after getting a quote in case the quote is too high.

### 3.3.2 Scopes

In addition to patterns, each pattern has a **scope** which is the temporal extent of a system’s execution over which a pattern must hold. A scope divides a system’s execution into different portions where a given constraint is effective. There are five basic kinds of scopes: global, before, after, between, and after-until.

- **Globally**: describes the scope of the entire system’s execution.
- **Before**: describes the scope up to the existence of a given event/operation.
- **After**: describes the scope after the existence of a given event/operation.
- **Between**: describes the scope by two events/operations. The specified scope is between the first event/operation to the second event/operation.
- **After-until**: describes the scope by two events/operations. The specified scope is after the first event/operation and up to the second event/operation. However, this scope pattern is different from **Between** scope in that the second event/operation might not occur.
Figure 3-9 illustrates how a system’s execution trace is divided into different portions by various scopes.

As illustrated in Figure 3-9, the “Before Q” and “After Q” scopes start with Q and completes at the occurrence of R. The “Between Q and R” scope requires both Q and R to occur in order to form the scope. Note that Q opens the scope and R closes the scope that is currently open. If there are two consecutive occurrences of Q before R, the first Q will open the scope and the next Q does not have any effect. Hence the event sequence in Figure 3-9 forms two portions of the “Between Q and R” scope: QR-QQR, instead of having just one portion formed by the first Q and the last R. The “After Q until R” scope requires only Q to form the scope and R will close the scope (unlike the Between scope where both Q and R are required). If R does not occur, the scope does not close. Therefore, there are three portions of “After Q until R” scope in Figure 3-9.

Consider the case that we want to specify the “Between Q and R” scope formed by the first Q and the last R in Figure 3-9, we cannot specify this scope since SPS does not take instances of events into consideration. The next section presents the work of Jin and Han (2005b) that addresses this limitation.

### 3.3.3 Parameters

In previous sections, we have discussed the use of Specification Pattern System (SPS) to describe interaction constraints. However, SPS does not consider the instances of
event types. It is argued that the effect of different instances of an event should be taken into consideration in behavioural protocol specification (Jin and Han 2005b; Li et al. 2005). For example, a book buyer should be able to place multiple orders while waiting for deliveries of previous orders. For each book order, the transaction is expected to follow the specified protocol.

Interaction Rule Specification (IRS) addresses the above-mentioned limitation of SPS by using parameters to correlate between instances of events (Jin and Han 2005b). IRS uses where-clauses to associate events/operations with particular parameters. A where-clause specifies the condition of when a constraint is applied based on the values of parameters of the event/operation in the constraint. For example, to specify a rule “once a payment is made, a delivery will be sent out”, we use two constraints as in Listing 3-1: “Vendor.payment precedes Vendor.delivery” and “Vendor.payment leads to Vendor.delivery”. These two constraints mean that the payment operation is a pre-condition to delivery; and once the payment is made, eventually delivery will occur. In addition, the where-clause in Listing 3-1 associates orderNo to these constraints which means that these two constraints only apply to the same order. The operator “::” in Listing 3-1 denotes that the parameter orderNo belongs to operations Vendor.payment and Vendor.delivery.

```plaintext
Vendor.payment precedes Vendor.delivery
where Vendor.payment::orderNo = Vendor.delivery::orderNo;

Vendor.payment leads to Vendor.delivery
where Vendor.payment::orderNo = Vendor.delivery::orderNo;
```

Listing 3-1: Where-clause adds parameter conditions to constraints

### 3.4 FSA Semantics of Constraint Patterns

In order to enable automated reasoning and analysis on behavioural protocols, it is essential to define formal semantics for constraint patterns. In this section, we describe the work of Jin and Han (2005a) which provide a mapping of SPS constraint patterns to Finite State Automata.

Dwyer et al. (1999) proposed a direct mapping of SPS constraint patterns to formalisms such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and Quantified Regular Expression (QRE) as in. However, according to Jin and Han
(2005a), these formalisms have not gained wide acceptance in the industry, because of the lack of familiarity of software practitioners to those formalisms.

Jin and Han (2005a) proposed a mapping of SPS constraint patterns to Finite State Automaton (FSA) formalism which is relatively simpler and is used more widely compared to other formalisms. FSA models a behavioural protocol as a sequence of transitions. The advantages of using FSA are that FSA is effective in modelling sequential systems and verification of protocol compatibility (Berardi et al. 2004; Canal 2004).

3.4.1 FSA Templates for Occurrence Patterns

The occurrence patterns under investigation are absence, existence and bounded existence. In Jin and Han (2005a), the authors did not consider Universality, as it can be viewed as an absence of its negation.

Figure 3-10 illustrates the FSA templates of occurrence patterns: absence, existence and bounded existence. State 0 is an initial state, indicated by having a hollow triangle. A double circled state is a final state. A transaction is expected to be terminated at a final state in order to be considered correctly conforming to the specified behavioural protocol. An event P is the event of interest, O indicates all other events besides P. The detailed description is as follows:

- Figure 3-10(a) presents the FSA template for the absence pattern – “P is absent globally”. At the initial state 0, any other events can occur. The FSA does not accept sequences that have the event P. This is what we expect, as the constraint specifies that P does not occur.
- Figure 3-10(b) presents the FSA template for the existence pattern – “P exists globally”. At the initial state 0, any other events can occur. However, for the FSA to reach its final state, the event P is expected to occur.
- Figure 3-10(c) and Figure 3-10(d) present the FSA templates for the bounded existence pattern. We distinguish between “at most” and “at least” number of times. The sample constraint in Figure 3-10(c) – “P exists at most 2 times” – indicates that the number of times the event P occurs has to be smaller than or equal to 2. Hence, the states 0, 1 and 2 in Figure 3-10(c) are all final states. When the FSA reaches state 2, it does not accept the event P anymore. The
sample constraint in Figure 3-10(d) – “P \textbf{exists at least} 2 \textbf{times}” – indicates the number of times the event P occurs has to be greater than or equal to 2. Hence, the states 0 and 1 in Figure 3-10(d) are non-final states. The FSA is in final state only after P occurs at least 2 times.

(a) P is absent globally  
(b) P exists globally  
(c) P exists at most 2 times globally  
(d) P exists at least 2 times globally

![Figure 3-10: FSA templates for occurrence patterns (Jin and Han 2005a)](image)

### 3.4.2 FSA Templates for Order Patterns

The order patterns under investigation are precedence and response (also known as leads to). Chain-precedence and chain-response are not considered, as they are combinations of finer grained constraints such as precedence, response and other occurrence patterns.

Figure 3-11 illustrates the FSA templates of precedence and response (leads-to) patterns. Similar to Figure 3-10, the letter O represents a set of all other events besides the events of interest, i.e. events other than P and Q in this case.

(a) P precedes Q globally  
(b) P leads to Q globally

![Figure 3-11: FSA templates for order patterns (Jin and Han 2005a)](image)

- Figure 3-11(a) presents the FSA template for the precedence pattern – “P \textbf{precedes} Q \textbf{globally}”. This constraint means that P is a pre-condition for Q to
occur. Before P occurs, Q cannot occur. This is indicated by the fact that at the initial state 0, only event P and all other events besides P and Q are accepted and Q cannot occur. When P occurs, the FSA is transitioned into state 1 where the event Q can occur. Note that neither P nor Q is required to occur for the transaction to be valid. This is why states 0 and 1 are both final states.

- Figure 3-11(b) presents the FSA template for the response (or leads to) pattern – “P leads to Q globally”. This constraint means that when P occurs, eventually Q has to occur. State 0 accepts Q and other events. When P occurs, the FSA is transitioned into a non-final state, state 1. It is required that event Q occur in order to transition the FSA back to a final state (state 0 in this case). Note that at the initial state, the transaction is still valid even when P never occurs and Q can occur without P.

### 3.4.3 FSA Templates for Scopes

The scopes do not stand by themselves but are incorporated with occurrence and order patterns. In this section, we will illustrate the FSA templates for scopes with the one of the occurrence patterns: the existence pattern. The scopes before, after, between and after-until are discussed below. The scope globally indicates an entire system’s execution history, examples of usages of the scope globally were shown in Sections 3.4.1 and 3.4.2.

Figure 3-12 illustrates the FSA templates of the existence pattern with four different scopes: before, after, between-and and after-until. In Figure 3-12(a) and Figure 3-12(b), letter O represents events other than P and Q. And in Figure 3-12(c) and (d), letter O represents events other than P, Q and R.

- Figure 3-12(a) presents the FSA template for an existence pattern with the before scope – “P exists before Q”. The constraint “P exists” is only valid within the specified scope “before Q”. If Q does not occur, the scope is not formed; hence, we are not interested in the occurrence of P. Similarly, after Q occurs, we are also not interested in the occurrence of P. This is indicated by having states 0 and 1 as final states. Note that this constraint produces the same effect as the constraint “P precedes Q globally”.

---

53
Figure 3-12(b) presents the FSA template for an existence pattern with the **after** scope – “P exists after Q”. After Q occurs, P has to occur. At the initial state, the occurrence of Q will transition the FSA into state 1 which is a non-final state. Only when P occurs, the FSA will be transitioned back to a final state (state 2).

Figure 3-12(c) presents the FSA template for an **existence** pattern with **between-and** scope – “P exists between Q and R”. For the scope to be in effect, both Q and R have to occur. Event Q opens the scope and event R closes the scope. If any of those two events are missing, the scope is not formed; hence the constraint “P exists” is not in consideration. When there are both Q and R, P has to occur in between them. At state 1, after Q occurred, the FSA does not accept R. At this state, if R happens, the scope will close when P has not occurred yet, hence it will violate the constraint. Only after P occurred, then R can occur given that Q already happened previously in order to form the scope. The occurrence of R will transition the FSA back to the initial state. Some examples of valid sequences accepted by the FSA are: PRQ (when the sequence does not form the scope Q and R), PRQQP (when R does not exist after Q, hence the scope is not formed), QPQQR (there is a P between the first Q and the ending R), QPQR-OO-QPR (there are Ps in two scopes formed by Q and R separated by ‘-’ for readability). An example of invalid sequence when P does not exists between Q and R: QQRPR; this sequence is not accepted by the FSA. Note that the scope opens with the first Q and closes with the first R in the sequence, instead of the last R matches with the first Q.

Figure 3-12(d) presents the FSA template for an **existence** pattern with **after-until** scope – “P exists after Q until R”. As explained in Section 3.3.2, the after-until scope is similar to the between-and scope. The difference is that it does not require an occurrence of R to form the scope. Hence, after Q occurs, P is required to occur. This is indicated by having state 1 as a non-final state. Some examples of valid sequences accepted by the FSA are: PR (Q does not exist, hence the scope is not in effect), PRQP (R does not exist, however P exists after Q, hence it is valid), QQPRR (P is between Q and R), QPQR-OO-QPR (there are Ps in two scopes formed by Q and R separated by ‘-’ for
readability). Examples of invalid sequences: PRQ (existence of Q starts the scope and P is absent after Q - note that this sequence is accepted by the between-and scope), PRRQR (there is no P in between Q and R). These invalid sequences are not accepted by the FSA.

![FSAs for existence pattern](image)

(a) P exists before Q  
(b) P exists after Q  
(c) P exists between Q and R  
(d) P exists after Q until R

Figure 3-12: FSA templates for existence pattern with Scope (Jin and Han 2005a)

### 3.4.4 Compound Constraints and Composition of FSAs

A constraint can be a simple pattern constraint consisting of only one pattern, or it can be a compound constraint consisting of multiple patterns joint by composition operators. Yu et al. (2006) defined five composition Boolean operators: AND, OR, and XOR. For example, to specify that “an order is followed by a payment or an order cancellation”, we can use an OR operator:

```
order leads to payment globally
or order leads to orderCancellation globally;
```

In order to analyse the behavioural protocol of a contract as a whole, a composition of those individual constraints is needed. Each constraint is directly mapped to an FSA as described in previous sections. For a simple constraint, its pattern is mapped to its corresponding FSA. For a compound constraint, each pattern is converted to its FSA, the constituent patterns are then composed based on its composition operator. Finally, the FSAs of all constraints are then composed by using AND composition to form an FSA representing the behavioural protocol of a
An FSA $A$ is denoted as: $A = (Q, \Sigma, \delta, q^0, F)$ where:

- $Q$ is a finite set of states
- $\Sigma$ is a finite set of events (called the input alphabet)
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. We will use the triple of ($\text{fromState}$, $\text{event}$, $\text{toState}$) to denote $\delta$
- $q^0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states

Given two FSAs $A_1 = (Q_1, \Sigma_1, \delta_1, q_{1}^0, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, q_{2}^0, F_2)$, the composition of $A_1$ and $A_2$ is an FSA $A = (Q, \Sigma, \delta, q^0, F)$ where its states are a Cartesian product of two sets of states of $A_1$ and $A_2$. And for every transition function in the intersection FSA, there exist transition functions in both operands’ FSAs that accept the corresponding constituent states. Final state is determined by the composition operators used (i.e. AND, OR, and XOR). More formally:

- $Q = Q_1 \times Q_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta = \{(q_{1}^i, q_{2}^j, \alpha, (q_{1}^k, q_{2}^m)) | (q_{1}^i, \alpha, q_{1}^k) \in \delta_1 \land (q_{2}^j, \alpha, q_{2}^m) \in \delta_2 \}$ where $\alpha \in \Sigma$
- $q^0 = (q_{1}^0, q_{2}^0)$

For AND operator: $F = \{(f_1, f_2) | f_1 \in F_1 \text{ AND } f_2 \in F_2 \}$
For OR operator: $F = \{(f_1, f_2) | f_1 \in F_1 \text{ OR } f_2 \in F_2 \}$
For XOR operator: $F = \{(f_1, f_2) | f_1 \in F_1 \text{ XOR } f_2 \in F_2 \}$

We will illustrate the above FSA composition concept by the following example of composing two constraints: “$P$ precedes $Q$ globally” AND “$P$ leads to $Q$ globally”. The FSAs corresponding to these two constraints are presented in Figure 3-11; they are also shown in Figure 3-13 for convenience.
Chapter 3  Preliminaries

(a) P precedes Q globally  (b) P leads to Q globally  (c) FSA composition of (a) and (b)

Figure 3-13: FSA intersection of “P precedes Q globally” AND “P leads to Q globally”

Let \( A_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1) \) is the FSA of the constraint “P precedes Q globally”.

This FSA is depicted in Figure 3-13(a).

- \( Q_1 = \{0; 1\} \)
- \( \Sigma_1 = \{P; Q; O\} \)
- \( \delta_1 = \)

<table>
<thead>
<tr>
<th>FromState</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>Q, P, O</td>
</tr>
</tbody>
</table>

- \( q_1^0 = 0 \)
- \( F_1 = \{0; 1\} \)

Let \( A_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2) \) is the FSA of the constraint “P leads to Q globally”.

This FSA is depicted in Figure 3-13(b).

- \( Q_2 = \{0; 1\} \)
- \( \Sigma_2 = \{P; Q; O\} \)
- \( \delta_2 = \)

<table>
<thead>
<tr>
<th>FromState</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q, O</td>
<td>Q</td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>P, O</td>
</tr>
</tbody>
</table>

- \( q_2^0 = 0 \)
- \( F_2 = \{0\} \)

The FSA \( A = (Q, \Sigma, \delta, q^0, F) \) is the FSA composition of \( A_1 \) and \( A_2 \), which is depicted in Figure 3-13(c), where:

- \( Q = \{ (0, 0); (0, 1); (1, 0); (1, 1) \} \)
Chapter 3  Preliminaries

- \( \Sigma = \{ P; Q; O \} \)
- \( \delta = \)

\[
\begin{array}{c|cccc}
\text{ToState} & 0, 0 & 0, 1 & 1, 0 & 1, 1 \\
\hline
0, 0 & O & - & - & - \\
0, 1 & - & O & - & - \\
1, 0 & - & - & Q, O & Q \\
1, 1 & - & - & - & P, O \\
\end{array}
\]

- \( q^0 = (0, 0) \)
- \( F = \{ (0, 0); (1, 0) \} \) (since in \( A_1 \) states 0 and 1 are final states, and in \( A_2 \) state 0 is a final state)

From the initial state (0,0), there is no sequence of transitions to get to state (0,1), therefore this state and the transitions going out of this state can be removed from the FSA for simplification.

3.5 Summary

In this chapter, we presented four Library Book Buying scenarios that highlight different cases of incompatibilities between a library book ordering system and book vendors’ Web services. In subsequent chapters, these scenarios will illustrate our approach to a semi-automated service composition framework that identifies and addresses incompatibilities between interacting parties. We then discussed the Role Oriented Adaptive Design (ROAD) framework which is the foundation of our proposed framework. The ROAD framework models software systems by using a role-based architecture. The adaptability that ROAD provides is the ability to have different services playing roles at different times, to create new role instances and to revoke/create relationships (contracts) between roles, thus the structure of a composition can be changed dynamically. However, the ROAD framework previously lacked the ability to specify behavioural protocols formally. We then discussed the Interaction Rule Specification (IRS) as a formal declarative pattern-based approach to specify behavioural protocols. IRS is a divide-and-conquer approach, where constraints can be added incrementally. In addition, interaction constraints can be specified independently from other constraints. IRS has FSA as its underlying semantics to allow reasoning and analysis of constraints. Each constraint
pattern has a corresponding FSA template. In the next chapter, we will describe our approach to utilise IRS to specify behavioural protocols in the ROAD framework.
Chapter 4: 
Behavioural Protocol in the ROAD Framework

In the previous chapters, we have illustrated how a software system can be modelled as an organisational structure consisting of roles and contracts (relationships) between roles in the Role Oriented Adaptive Design (ROAD) framework. We have also discussed how behavioural protocols can be specified by using Interaction Rule Specification (IRS). In this chapter, we describe how to apply IRS to specify behavioural protocols in ROAD contracts (Section 4.1). We show that besides the protocols of contracts, it is also required to specify dependencies between contracts (Section 4.2). For a service to be able to play a certain role, the service has to match the role’s requirements. In order to perform protocol compatibility checking between service and role, a role-centric protocol is required. We discuss how an aggregation of protocols in contracts and their dependencies can be carried out to generate role-centric protocols in Section 4.3. Since protocols are specified incrementally, it is possible that those constraints are inconsistent with each other. Section 4.4 details a mechanism that checks those constraints to ensure that the protocol is consistent. In addition to protocol specification, it is required that the protocols are adhered to during run-time. In Section 4.5, we discuss how the interactions between roles will be monitored and checked against the specified protocols in contracts and their dependencies by a run-time protocol monitoring mechanism. Finally, we summarise the chapter in Section 4.6.

4.1 Using IRS to Define Behavioural Protocols in Contracts

In this section, we show how Interaction Rule Specification (IRS) presented in Section 3.3 can be applied to define the library’s behavioural protocol requirement of book vendors that it will interact with, as in scenario 1 in Section 3.1. We model the
composition of the library book buyer (lib-BB) and its book vendor’s service by a composite consisting of Buyer, Vendor roles and their contract as in Figure 4-7. Organiser role and its player are omitted for simplicity.

![Library Book Buying composite](image)

**Figure 4-1: Library Book Buying composite**

The required behavioural protocol between lib-BB and potential book vendor’s services is specified in contract between Buyer and Vendor roles. Figure 4-2 depicts the FSA representing the behavioural protocol that the library book buyer (lib-BB) requires.

![Library's Requirements of Vendor](image)

**Figure 4-2: The FSA of behavioural protocol that Vendor services are required to support**

Each transition in this figure represents a set of business level operational messages between interacting parties. For example, a `quote` operation might correspond to a quote request message and a quote response message; a `payment` operation might correspond to payment detail and acknowledgement messages. We assume the `payment` operation will handle invalid payment and ask for a different payment until a valid one is received. Each of these operations corresponds to a term in Buyer and Vendor contract, for example the `quote` operation corresponds to the `quote` term.

To interact with the library, a compatible service represented by the Vendor role is required to have this behavioural protocol. Therefore, we prefix operation names with the word “Vendor”. The interaction starts with a Vendor.quote operation. The notation `entity.operation` implies the entity provides the operation that can be invoked by other
parties. For example, Vendor.quote means that the Vendor role offers a quote operation to be invoked to obtain quotes for books.

After getting a quote response, the interaction proceeds with a Vendor.order operation. However, if a quote is too high, an order might not be placed. We have our first constraint:

\[
\text{Vendor.quote precedes Vendor.order globally;}\]

Note that we did not have constraint “Vendor.quote leads to Vendor.order globally”, as Vendor.order does not necessarily occur.

Once an order is placed, a payment is sent out. Our next two constraints are:

\[
\begin{align*}
\text{Vendor.order precedes Vendor.payment globally;} \\
\text{Vendor.order leads to Vendor.payment globally;}
\end{align*}
\]

Note that we use both precedes and leads to constraints in this case since the payment operation has to occur after the order operation.

Similarly, we specify constraints for deliveryDetail and delivery operations:

\[
\begin{align*}
\text{Vendor.payment precedes Vendor.deliveryDetail globally;} \\
\text{Vendor.payment leads to Vendor.deliveryDetail globally;} \\
\text{Vendor.deliveryDetail precedes Vendor.delivery globally;} \\
\text{Vendor.deliveryDetail leads to Vendor.delivery globally;}
\end{align*}
\]

In this scenario, each operation will occur at most once (0 or 1 time); therefore we add the exist constraints for each operation.

\[
\begin{align*}
\text{Vendor.quote exists at most once globally;} \\
\text{Vendor.order exists at most once globally;} \\
\text{Vendor.payment exists at most once globally;} \\
\text{Vendor.deliveryDetail exists at most once globally;} \\
\text{Vendor.delivery exists at most once globally;}
\end{align*}
\]

All of the above constraints are applied for a given ordering process in which a quote operation is also part of. Hence, we associate all of the above constraints with a parameter orderNo by using where-clauses. A complete set of constraints of the required behavioural protocol is presented in Listing 4-1.

The above constraints are then composed to produce an FSA representing the library’s requirement of behavioural protocol that Vendor services have to support. The composed FSA contains only the paths from the initial state leading to one of the final states, all the paths that do not lead to final states are removed. Figure 4-3 shows
the composed FSA generated by our proof-of-concept implementation (see Chapter 8).

Listing 4-1: A set of constraints of the behavioural protocol that Vendor services need to support

```plaintext
Vendor.quote precedes Vendor.order globally  
   where Vendor.quote::orderNo = Vendor.order::orderNo;

Vendor.order precedes Vendor.payment globally  
   where Vendor.order::orderNo = Vendor.payment::orderNo;

Vendor.order leads to Vendor.payment globally  
   where Vendor.order::orderNo = Vendor.payment::orderNo;

Vendor.payment precedes Vendor.deliveryDetail globally  
   where Vendor.payment::orderNo = Vendor.deliveryDetail::orderNo;

Vendor.payment leads to Vendor.deliveryDetail globally  
   where Vendor.payment::orderNo = Vendor.deliveryDetail::orderNo;

Vendor.deliveryDetail precedes Vendor.delivery globally  
   where Vendor.deliveryDetail::orderNo = Vendor.delivery::orderNo;

Vendor.deliveryDetail leads to Vendor.delivery globally  
   where Vendor.deliveryDetail::orderNo = Vendor.delivery::orderNo;

Vendor.quote exists at most once globally  
   where Vendor.quote::orderNo;

Vendor.order exists at most once globally  
   where Vendor.order::orderquote::orderNo;

Vendor.payment exists at most once globally  
   where Vendor.payment::orderNo;

Vendor.deliveryDetail exists at most once globally  
   where Vendor.payment::orderNo;

Vendor.delivery exists at most once globally  
   where Vendor.orderCancellation::orderNo;
```

Listing 4-1: A set of constraints of the behavioural protocol that Vendor services need to support

Figure 4-3: Behavioural protocol FSA output from composing constraints

In Listing 4-1, there is a large number of constraints. This is because we tried to model all transitions acceptable by the FSA representing the lib-BB’s required behavioural protocol as in Figure 4-2. This implies the set of constraints listed and the FSA are equivalent in their set of allowable behaviour. In such an extreme case, using FSA directly might be shorter (even though using the IRS constraints is more intuitive). However, in the case where we only want to model the sequence of certain events, IRS has a clear advantage because with IRS we can incrementally specify the constraints without having to enumerate all the possible allowable behaviours as in
the case of using FSA. For example, it is required that “P occurs before Q” and “R occurs before S”, the order between P and (R, S) and the order between Q and (R, S) are not important. By using IRS, we only need to specify two constraints: “P precedes Q” and “R precedes S”; while it is more complex to specify the same constraints by using FSA with 4 states and 12 transitions (Figure 4-4).

Figure 4-4: FSA representation of “P precedes Q” and “R precedes S”

Back to our Book Buying scenario, assuming that the library changes the behavioural protocol so that it now requires Vendor services to also support order cancellation. The library can cancel an order before the payment is made. Figure 4-5 shows the FSA of this new requirement.

Figure 4-5: The FSA of behavioural protocol that allows order cancellation

IRS allows easy modification of constraints by removing constraints that are no longer applicable and adding new ones. In this example, we replace the constraint “Vendor.order leads to Vendor.payment globally” with the new constraints: “Vendor.order leads to Vendor.payment globally OR Vendor.order leads to Vendor.orderCancellation globally”. The OR composition operator signifies the alternative paths. In addition, we need to specify that the payment operation cannot occur after the orderCancellation operation and vice versa. We add two more constraints: “Vendor.payment is absent after Vendor.orderCancellation” and “Vendor.orderCancellation is absent after Vendor.payment”. To show how constraints in Listing 4-1 are modified, Listing 4-2 presents a complete set of constraints specifying the new behavioural protocol in which newly added constraints are in italic and

64
removed constraint(s) are in strike-through. Where-clauses are omitted for simplicity. Figure 4-6 shows the FSA output from composing the constraints in Listing 4-2.

![Figure 4-6](image)

Listing 4-2: A set of constraints of the library that requires Vendor services to support order cancellation operation

<table>
<thead>
<tr>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor.quote precedes Vendor.order globally;</td>
</tr>
<tr>
<td>Vendor.order precedes Vendor.payment globally;</td>
</tr>
<tr>
<td>Vendor.order precedes Vendor.orderCancellation globally; //added</td>
</tr>
<tr>
<td>Vendor.order leads to Vendor.payment globally;</td>
</tr>
<tr>
<td>OR Vendor.order leads to Vendor.orderCancellation globally; //added</td>
</tr>
<tr>
<td>Vendor.payment is absent after Vendor.orderCancellation; //added</td>
</tr>
<tr>
<td>Vendor.orderCancellation is absent after Vendor.payment; //added</td>
</tr>
<tr>
<td>Vendor.payment precedes Vendor.deliveryDetail globally;</td>
</tr>
<tr>
<td>Vendor.payment leads to Vendor.deliveryDetail globally;</td>
</tr>
<tr>
<td>Vendor.deliveryDetail precedes Vendor.delivery globally;</td>
</tr>
<tr>
<td>Vendor.deliveryDetail leads to Vendor.delivery globally;</td>
</tr>
<tr>
<td>Vendor.quote exists at most once globally;</td>
</tr>
<tr>
<td>Vendor.order exists at most once globally;</td>
</tr>
<tr>
<td>Vendor.orderCancellation exists at most once globally; //added</td>
</tr>
<tr>
<td>Vendor.payment exists at most once globally;</td>
</tr>
<tr>
<td>Vendor.deliveryDetail exists at most once globally;</td>
</tr>
<tr>
<td>Vendor.delivery exists at most once globally;</td>
</tr>
</tbody>
</table>

Figure 4-6: Behavioural protocol FSA output from composing constraints that allows order cancellation

The examples of IRS constraints shown in Listing 4-1 and Listing 4-2 would be written by developers to define interaction constraints in a composite. At the moment, there is no tool support for inputting the constraints; developers enter constraints by using a text editing tool. To make the input easier, a more advanced editing tool is required. This tool could support dragging-and-dropping, re-arranging events graphically, and generating the corresponding constraints from the order of the graphical representations of events. We intend to investigate how to implement such a tool in future work.

### 4.2 Protocol Dependencies between Contracts

In previous sections, we presented our approach to using Interaction Rule Specification (IRS), a declarative pattern-based specification, to specify behavioural
protocols of contracts. In this section, we will show that the dependencies between contracts are also required.

The description of point-to-point protocols in a composition is necessary but may not be enough to ensure that a business transaction is carried out correctly. It may be necessary to model the interactions between multiple (three or more) services in a composition. Consider, for example, a holiday booking scenario. While a hotel room might be initially reserved, the finalisation of the hotel booking can only be done after the booking of air tickets is confirmed. There is therefore a dependency between the client-to-hotel booking protocol and the client-to-airline booking protocol. This requires dependencies between protocols to be specified and enforced.

In an orchestrated composition, the dependencies between protocols are typically captured in the sequencing of service invocations (e.g. as in a WS-BPEL description). However, in the ever-changing world of Web services, this imperative approach to protocol specification is brittle if the protocols need to change. Protocols may change as the business processes change to better suit customers’ needs. Consequently, the service composition has to update its protocols and their dependencies to reflect such changes.

To facilitate the specification of protocol dependencies and the protocol modification process, we also use IRS to specify the protocol dependencies between contracts. The protocol dependencies will be maintained by the organiser of the composite. We will illustrate this concept in the context of the Library Book Buying scenario presented previously in Section 3.1. Consider the scenario where there is a broker that facilitates the buying process between the library and book selling services. The broker will search for the cheapest quotes and order books from that service. This extra complexity is hidden from the library. In order to model this scenario, we need to add a Broker role into the composite. Figure 4-7 depicts the structure of this new composite. There are multiple instances of the Vendor role (Vendor1 and Vendor2) to represent different services that the broker will interact with. There are also multiple Broker-Vendor contracts (BrokerVendor1 and BrokerVendor2) to keep track of the interactions with different sellers. These contracts can have different behavioural protocols if required, for example the BrokerVendor1 contract requires
payment before delivery and the BrokerVendor2 contract mandates delivery before payment.

From the Buyer’s point of view, the Broker acts as a Vendor. The complexity of acquiring the cheapest quote is hidden from the Buyer. Therefore, the behavioural protocol between Buyer and Broker is similar to the protocol between Buyer and Vendor in Section 4.1: a sequence of quote, order, payment, deliveryDetail, and delivery.

Similarly, from the Vendor’s point of view, the Broker is like a proxy that forwards requests from the Buyer. The only difference is that the Broker gets various quotes from different book sellers’ services via their Vendor roles; and it replies to the Buyer with the cheapest one. Therefore, the behavioural protocol between Broker and Vendor is similar to the protocol between Buyer and Vendor in Section 4.1: a sequence of quote, order, payment, deliveryDetail, and delivery. The distinction in this scenario is the dependencies between the Buyer-Broker contract and each of the Broker-Vendor contracts.

The Buyer-Broker contract’s protocol and the Broker-Vendor contracts’ protocols are specified independently. If the dependencies between these protocols are not specified, the Broker will be able to ask the Vendor for quote before it receives a quote request from the Buyer. In this situation, the individual protocols are still adhered to; however, the correctness of the transaction as a whole has been violated. In addition, in a given transaction, the Broker can obtain quotes from multiple Vendors, but it can
only order from one Vendor. Therefore, we have to add dependency constraints between Buyer-Broker and Broker-Vendor contracts. In a transaction, each operation (quote, order, payment, deliveryDetail or delivery) in the interaction between the Buyer and the Broker has to occur before the corresponding operation between the Broker and the Vendors; in addition, order, payment, deliveryDetail and delivery operations can only be invoked from one Vendor for a given order. Listing 4-3 shows the constraints to specify these dependencies; an operation is denoted by a triple of contract name, role name and operation name separated by a dot ‘.’: 

\[
\text{contractName.roleName.operationName}
\]

These constraints in Listing 4-3 are applied for parameter orderNo, and where-clauses are omitted for simplicity (Listing 4-4 shows an example of a dependency constraint with where-clause). In the Listing 4-3, we assume that Buyer-Broker interactions and Broker-Vendor interactions use the same orderNo, or there is a mapping mechanism to associate the two parameters if they are different. Protocol dependencies are specified in the \textit{organiser} of the composite by developers.

```
protocol dependencies {
  Buyer-Broker.Broker.quote precedes Broker-Vendor1.Vendor1.quote globally;
  Buyer-Broker.Broker.quote leads to Broker-Vendor1.Vendor1.quote globally;
  Buyer-Broker.Broker.quote precedes Broker-Vendor2.Vendor2.quote globally;
  Buyer-Broker.Broker.quote leads to Broker-Vendor2.Vendor2.quote globally;
  Buyer-Broker.Broker.order precedes Broker-Vendor1.Vendor1.order globally;
  Buyer-Broker.Broker.order precedes Broker-Vendor2.Vendor2.order globally;
  Buyer-Broker.Broker.order leads to Broker-Vendor1.Vendor1.order globally
  OR Buyer-Broker.Broker.order leads to Broker-Vendor2.Vendor2.order globally;
  Buyer-Broker.Broker.payment precedes Broker-Vendor1.Vendor1.payment globally;
  Buyer-Broker.Broker.payment precedes Broker-Vendor2.Vendor2.payment globally;
  Buyer-Broker.Broker.payment leads to Broker-Vendor1.Vendor1.payment globally
  OR Buyer-Broker.Broker.payment leads to Broker-Vendor2.Vendor2.payment globally;
  Buyer-Broker.Broker.deliveryDetail precedes Broker-Vendor1.Vendor1.deliveryDetail globally;
  Buyer-Broker.Broker.deliveryDetail precedes Broker-Vendor2.Vendor2.deliveryDetail globally;
  Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor1.Vendor1.deliveryDetail globally
  OR Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor2.Vendor2.deliveryDetail globally;
  Buyer-Broker.Broker.delivery precedes Broker-Vendor1.Vendor1.delivery globally;
  Buyer-Broker.Broker.delivery precedes Broker-Vendor2.Vendor2.delivery globally;
  Buyer-Broker.Broker.delivery leads to Broker-Vendor1.Vendor1.delivery globally
  OR Buyer-Broker.Broker.delivery leads to Broker-Vendor2.Vendor2.delivery globally;
}
```

Listing 4-3: Dependencies between Buyer-Broker and Broker-Vendor contracts
Chapter 4  Behavioural Protocol in the ROAD Framework

4.3 Role-centric Protocol

For a player to be able to play a role, it has to be compatible to the role’s requirements in both functionality and behavioural protocols. As the role may be involved in the various contracts, in order to verify the compatibility between services’ protocols and the role’s required protocol, an aggregation of contract protocols that the role is involved in and their dependencies is required. A role-centric protocol is such an aggregation. It is a role’s view or projection of the protocols in the contracts and their dependencies that this role is associated with. A role-centric protocol is the behavioural protocol requirements to be matched against candidate services to play this role. We will discuss match making in detail in Chapter 5.

The organiser of a composite will initiate this aggregation when there is a change in the contracts’ protocols within its composite. Then roles are responsible to perform the aggregation required. Each role has references to the contracts that it is associated with. During the aggregation process, it will query the contracts and the organiser in order to retrieve the constraints and their related dependencies. There are two cases when performing aggregation: (i) role that is associated with only one contract (e.g. in Figure 4-7, the Buyer role is associated with one Broker role); and (ii) role that is associated with multiple contracts (e.g. in Figure 4-7, the Broker role is associated with multiple Vendor roles).

4.3.1 Protocol Aggregation with One Contract

For role that is associated with only one contract, such as the Buyer role or the Vendor roles in Figure 4-7, role-centric protocol is a set of constraints in the contract that involve its role’s name. In other words, it is a projection of a contract protocol on a specific role. Listing 4-5 shows the pseudo-code to achieve this protocol aggregation.

The following example illustrates how the protocol aggregation is done for the Vendor1 role in the Library Book Buying composite in Figure 4-7. The Broker-Vendor1

Listing 4-4: A dependency constraint between Buyer-Broker and Broker-Vendor contracts with a where-clause

```plaintext
Buyer-Broker.Broker.quote precedes Broker-Vendor1.Vendor1.quote globally
where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor1.Vendor1.quote::orderNo;
```
contract protocol is parsed and only constraints involving Vendor1 are extracted; and similarly for constraints in protocol dependencies. Listing 4-6 shows the Vendor1’s role centric protocol as the result of the aggregation.

```plaintext
Listing 4-5: Pseudo-code of protocol aggregation with one contract

```
### 4.3.2 Protocol Aggregation with Multiple Contracts

For a role that is associated with multiple contracts, such as the Broker role in Figure 4-7, the aggregation of the role-centric protocol is more complicated. It involves the aggregation of each contract this role is associated with and the dependencies between contracts.

Consider the case where the Broker role is contracted to multiple Vendor roles as in Figure 4-7. Each Broker-Vendor contract can have different protocols to other Broker-Vendor contracts. A Broker’s player is required to support all the protocols in the contracts that it is associated with. For this reason, the composition of constraints among contracts is an AND composition. The result of this composition is then aggregated with the dependencies constraints by also using an AND composition. Listing 4-7 shows the pseudo-code for the aggregation of protocol with multiple contracts. The AND composition was discussed in Section 3.4.4. The process of aggregation of protocols for the Broker role is summarised in Listing 4-8 and the full listing of the Broker’s role-centric protocol is shown in Appendix A.

```python
01 function calculate protocol aggregation of a role {  
02 if (role has only one contract) {  
03     //the case of aggregation of a single contract in Section 4.3.1  
04 }  
05 else {  
06     protocolAggregation = new FSA  
07     for each contract that this role is associated with {  
08         protocolAggregation = AND composition of protocolAggregation and  
09         projection of each contract to this role  
10     }  
11     role-centric-protocol = AND composition of protocolAggregation and  
12     projection of protocol dependencies to this role  
13 }  
14 function projection of a contract to role {  
15     //as in Listing 4-5  
16 }  
17 function projection of protocol dependencies to role {  
18     for each constraint in the protocol dependencies {  
19         if (constraint involves the role) {  
20             add constraint to the projection  
21         }  
22 }  
23 }  
24 }
```

**Listing 4-7: Pseudo-code of protocol aggregation with multiple contacts**
4.4 Constraints Composition and Consistency Checking

In previous sections, we described that protocol is specified incrementally by using Interaction Rule Specification (IRS). Since a protocol is specified incrementally, there are chances that the newly added constraints are inconsistent with the previous ones. This is particularly so for role-centric protocols as they are compositions of multiple protocols. A simple example is the two contradicting constraints: “P precedes Q” and “Q precedes P”. Therefore, it is necessary to perform protocol consistency checking for contract protocols, role-centric protocols, and protocol dependencies.

Before consistency checking can be carried out, IRS constraints of a protocol are converted and composed to form a single FSA representing the protocol. The resulting FSA is then analysed for consistency. The FSA composition and consistency checking are done by the FSA Generator in the organiser of a composite. Figure 4-8 shows the structure of the FSA Generator.
The constraints are input into the Parser and FSA Translator (PFT) which is part of the FSA Generator as in Figure 4-8. The PFT will parse the compound constraints into elementary constraints. The implementation of the parser is done by utilising the ANTLR (ANother Tool for Language Recognition) Parser Generator (Parr 2006). The IRS grammar is input to the ANTLR; it then outputs Java code that can parse IRS constraints. Each elementary constraint pattern has a corresponding FSA template as discussed in Section 3.4. After each constraint is parsed and its pattern is recognised, the PFT will convert the constraint to its FSA based on the constraint’s FSA template.

Once the FSA of each constraint is created, the composition and consistency checking are then carried out by the FSA Composer and Consistency Checker (FCCC). The FCCC will compose the FSAs of all constraints by using the AND composition to form an FSA representing all the constraints (more details in Section 3.4.4). In the case that the resulting FSA is empty (there is no path from the initial state leading to one of the final states), this signifies that there are conflicting constraints. If such a conflict exists, an inconsistency exception will be thrown and handled by the organiser player. The level of intelligence of specific organiser players can vary. If the organiser’s player cannot handle the exception, it can throw the exception which will be handled by players who plays the Organiser roles of composites at higher levels (see Section 3.2.3 for the discussion about the dependency between organisers). The inconsistency is handled at the organiser level instead of the role level because an organiser is responsible for managing a composite and has an overall knowledge of the composite, whereas a normal role does not have enough information to make decisions of how to deal with inconsistencies. If constraint inconsistencies cannot be dealt with by organisers’ players, human intervention is required. How an organiser player specifically deals with the exception is out of scope of this thesis.

### 4.5 Run-time Protocol Monitoring

The behavioural protocol is specified at the design time. During run-time, an interaction might not follow what were previously specified in the protocol. This
leads to erroneous and unsuccessful interactions. Therefore, run-time protocol monitoring is crucial.

In our approach, the run-time protocol monitoring is performed by the Protocol Monitor (PM) in a contract and in the organiser of the composite. Since roles’ interactions occur via contracts, contracts will intercept messages exchanged between their associated roles. The intercepted messages will then be checked for protocol compliance by the PMs. A message is considered valid when it satisfies both contract’s protocol and organiser’s protocol dependencies. The monitoring process is illustrated in Figure 4-9.

![Figure 4-9: The run-time protocol monitoring process](image)

A contract’s behavioural protocol and organiser’s protocol dependencies constraints are converted to FSAs by the FSA Composer and Consistency Checker as described in Section 4.4. When a message is intercepted by a contract (Step 1), the message will be passed to PMs of the contract and the organiser (Step 2 and 3). The PMs of the contract and the organiser will check the parameters of the message according to the specification in the where-clauses (Step 4 and 5). A new coordination context instance is created by the PM every time a new value of the specified parameter is encountered. Each coordination context contains a stateful instance of the composed FSA, which is used to keep track of the state of a conversation. On the other hand, if the parameter value is already encountered, it will update the existing coordination context’s FSA accordingly. The interaction is allowed if it is consistent with the current state’s allowable transitions in the
coordination context’s FSA. Otherwise the interaction will be rejected and the composite’s organiser is informed of the interaction violation. The player who initiated the non-conforming message is also notified. In our opinion, the purpose of the run-time monitoring is to check whether the interactions conform to the pre-defined constraints or not, hence the organiser will just stop the non-conforming messages and inform interested parties and will not try to adapt those messages. The implementation of the run-time protocol monitoring mechanism is detailed in Section 7.4.

4.6 Summary

In this chapter, we have presented an example to show how IRS can be applied to define behavioural protocols in contracts for the Library Book Buying scenario 1 in Section 3.1. The example also illustrated that modification to a constraint specification is relatively easy by removing unwanted constraints and adding new ones. We then discussed that protocol dependencies between contracts are as important as the contracts’ behavioural protocols themselves. We showed how the contracts’ behavioural protocols and protocol dependencies can be aggregated to form role-centric protocols. Role-centric protocols will be used to check the compatibility of services’ behavioural protocols, should such services become the roles’ players. Other mechanisms to support behavioural protocol in the framework were also described, including: consistency checking and run-time protocol monitoring.

In the next chapter, we will show how protocol compatibility between a role-centric protocol and services’ behavioural protocols can be checked in the match-making process. The outcome of the match-making process is then categorised into different levels of compatibilities between the services and the role.
Chapter 5:
Match-Making of Web Services to Roles

Previous chapters showed how the behavioural protocol of a system can be specified by using Interaction Rule Specification (IRS), and the behavioural protocols between roles are stored in their contracts. We then discussed that dependencies between contracts are also important and the model of such dependencies are specified in the organiser of a composite. The behavioural protocols in contracts and their dependencies are aggregated to form so-called role-centric protocols. These protocols will be used in compatibility checking between roles and their players (e.g. Web services). However, our role-centric protocol is specified as IRS constraints; whereas the service’s protocol may be in a different specification language, such as OWL-S. In this chapter, we will discuss this match-making process and how we overcome the differences in behavioural protocol notations.

In Section 5.1, we will give an overview of the steps involved in match-making Web services against their prospective roles. These steps include: (i) converting a role’s protocol specification and Web services’ protocol specifications to the same underlying formalism such as Finite State Automaton (FSA) (Section 5.2), and (ii) match-making of protocols by using the chosen underlying formalism and classifying their mismatches (Section 5.3). The classification of mismatches is required because the Web services were developed externally; it is unlikely that their protocols fully match roles’ protocols. The classification of mismatches is then used to adapt services to match roles’ requirements. Section 5.4 summarises the chapter.

5.1 The Process of Match-Making

For a Web service to become a player of a certain role, its protocol has to be compatible with the role-centric protocol. The role-centric protocol is specified by
using Interaction Rule Specification (IRS) as discussed in Chapter 4. However, the Web service’s protocol specification may be in a different notation than IRS. In Section 2.3.3, we reviewed three different approaches to specify Web services’ interaction protocols, namely OWL-S (Martin et al. 2004a), WSDL-S (Akkiraju et al. 2005) and WSMO (Bruijn et al. 2005). For the purpose of this thesis, we chose OWL-S as the Web service’s protocol specification since it is relatively more mature than WSDL-S and not as complex as WSMO (more details in Section 2.3.3). In this thesis, we assume that the service’s behavioural description is in OWL-S. However, our match-making process and approach are not limited to any particular behavioural description.

To recap, OWL-S describes a service’s behavioural protocol via a process model. The service’s process model is defined by composite processes consisting of either atomic processes or other composite processes. These processes are linked together by different control constructs such as sequence, choice, if-then-else, repeat-while, repeat-until, split, split+join, etc. A composite process defines the behavioural protocol of a service by control flow (different control constructs) and data flow (how information is passed from one process to another). On the other hand, atomic process, as the name suggests, cannot be broken into sub-processes. It is grounded to a concrete operation of service description elements (e.g. a WSDL operation).

Figure 5-1 summaries the creation of a role-centric protocol and highlights the problem of having different notations during match-making Web services to their prospective roles. It depicts a simple composite in the Library Book Buying context presented in Section 3.1. The Library Book Buying composite has two roles: Buyer and Vendor. The Buyer role represents the lib-BB; whereas the Vendor role represents a book seller’s service. The required sequence of interactions is set in Buyer-Vendor contract’s protocol (Step 1). The protocol is then aggregated by the organiser’s player to form a role-centric protocol for the Vendor role (Step 2). This protocol is specified in IRS. For a book seller’s service to play the Vendor role, it has to be compatible with the Vendor’s role-centric protocol. However, the match-making process is not straightforward as the service’s behavioural protocol is specified in a different notation, such as OWL-S. To overcome this problem, we will need to convert both the role-centric protocol (in IRS) and the service’s behavioural description (in OWL-
S) into a common formalism that allows analysis and reasoning. We chose Finite State Automaton (FSA) as the formalism. The main reasons are that we already have the mechanism to translate IRS to FSA and FSA formalism is effective in verification of protocol compatibility (Canal 2004; Berardi et al. 2004).

The process of translation to FSA is depicted in Figure 5-2 (the Buyer and Organiser roles are omitted for simplicity). Firstly, we need to convert IRS constraints to FSA. This is already described in Section 3.4. Secondly, the services’ behavioural description (e.g. in OWL-S) will also be converted to FSA. Finally, the service behavioural protocol’s FSA will be matched against the role-centric protocol’s FSA to determine their compatibility. In the next section, we will discuss our technique to convert OWL-S to FSA.
5.2 Converting OWL-S to FSA

To convert OWL-S to FSA, we have drawn on the work by Mokhtar et al. (2005; 2006). Each process whether atomic or composite in an OWL-S process model will be parsed and converted to an FSA. This approach is illustrated in Figure 5-3 and is described in detail below. For illustration purpose, we only show a composite process consisting of two sub-processes P1 and P2.

Figure 5-3: Modelling OWL-S processes as FSA (Mokhtar et al. 2005)
For an atomic process, the conversion is straightforward. The corresponding FSA has an initial state and a final state. The transition from the initial state to the final state is formed by the occurrence of the atomic process operation. This FSA is depicted as “Atomic Process ap” in Figure 5-3.

A composite process in OWL-S is composed of other processes (either atomic or composite) based on different control constructs such as sequence, choice, if-then-else, repeat-while, repeat-until, split, split+join, and any-order. For a composite process, the conversion to FSA is done by recursively converting its constituent processes into FSAs and combining those FSAs based on the control construct that they are in. The recursion is stopped when the atomic processes of those constituent processes are reached. How the FSAs are combined for different control constructs is illustrated in Figure 5-3 and is also discussed below. The combination process of FSAs makes use of ε transitions. An ε transition is a transition from one state to another state with an empty set of events.

- **Sequence(P1, P2):** A sequence is a list of processes to be executed in the given order. For a sequence of two processes P1 and P2, we convert each process to its corresponding FSA, and then combine them together by linking final states of P1’s FSA to the initial state of P2’s FSA by ε transitions as illustrated in Figure 5-3. The final states of P1 are no longer final states which are depicted as former final states in Figure 5-3. The final states of the newly combined FSA are the final states of P2’s FSA. The initial state of the newly combined FSA is the initial state of P1’s FSA.

- **Choice(P1, P2):** This construct consists of a collection of processes P1 and P2. Only one process in the collection (either P1 or P2) is chosen to be executed. The processes’ FSAs are combined by linking initial state of the resultant FSA to initial states of P1’s and P2’s FSAs by ε transitions as depicted in Figure 5-3. The final states of the combined FSA are the final states of P1’s and P2’s FSAs.

- **If(condition)-Then(P1)-Else(P2):** If the condition is true, P1 is executed, else P2 is executed. Depending on the value of the condition at run-time, P1 or P2 can be executed. Since the condition cannot be evaluated at design time, both branches of the If-Then-Else construct have to be considered separately.
during the compatibility checking. This is different from the Choice construct where either P1 or P2 can be chosen to be executed and there is no condition governing which one will be executed. For this reason, an If-Then-Else construct is converted to two FSAs as illustrated in Figure 5-3. For an OWL-S specification that has many If-Then-Else constructs, the conversion will result in $2^n$ FSAs in the worst case scenario, where $n$ is the number of the If-Then-Else constructs. The worst case scenario occurs when all the If-Then-Else constructs have both the If-Then and the Else branches and those constructs are placed one after the other (sequential) rather than in nested If-Then-Else constructs.

- **Repeat-While(P1):** The process P1 is executed as long as the condition holds true. The number of occurrences of P1 is unknown at design time as the value of the condition cannot be evaluated. This expression is considered unbounded number of occurrences of P1, whereas IRS specifies bounded occurrences. Therefore, Web services with OWL-S specifications having Repeat-While construct(s) are not considered matched and their OWL-S will not be converted to FSAs.

- **Repeat-Until(P1):** The process P1 is executed until the condition becomes true. Similar to the Repeat-While construct, OWL-S specifications having Repeat-Until construct(s) are not considered matched.

- **Split(P1, P2):** The specified collection of processes will be executed concurrently. The process is completed when all processes are scheduled to be executed. According to Wombacher et al. (2004b), automata do not provide means to model parallel execution, thus the resulting execution sequences must be enumerated. A technique to generate such enumeration is the shuffle product (Matz et al. 1995). The shuffle product keeps the message order within each message sequence, but combines two message sequences in all possible combinations. In other words, Split(P1, P2) can be treated as Choice(Sequence(P1, P2), Sequence(P2, P1)), although the concurrency semantics is not correctly modelled.

- **Split+Join(P1, P2):** This control construct is similar to Split(P1, P2); however there is a specification of barrier synchronisation, which means that the Split+Join process is completed when all its sub-processes have completed
(Martin et al. 2004a). Since concurrency is not supported, this construct is treated as Split(P1, P2) for the purpose of conversion to FSA.

- **Any-order(P1, P2):** The collection of processes can be executed in some unspecified order. All processes in the collection have to be executed. To convert this construct to FSA, a choice of sequences of all permutations of constituent processes is created. In this example of Any-order(P1,P2), the result is Choice(Sequence(P1, P2), Sequence(P2, P1)).

Our approach to the conversions of the If-Then-Else, Repeat-While and Repeat-Until constructs is different from the one suggested by Mokhtar et al. (2005; 2006). For the If-Then-Else construct, Mokhtar et al. (2005; 2006) considers it as similar to the Choice construct and the condition in the If-Then-Else construct is practically ignored. We argue that those two constructs are fundamentally different and both branches of the If-Then-Else construct have to be considered separately. The conversion of the If-Then-Else construct is one of our contributions. For the Repeat-While construct, Mokhtar et al. (2005; 2006) use an $\varepsilon$ transition to connect from the start of the loop to the end to bypass the loop; and for the Repeat-Until construct, an $\varepsilon$ transition is used to connect from the end of the loop to the start of the loop. The conditions are not considered. If party A and party B are interacting with each other, and in certain conditions, party A is expecting party B to perform an operation 5 times, while party B will only perform it 3 times, the two parties should be considered ‘not matched’. However, they would be regarded as compatible with each other by using Mokhtar et al.’s approach. We do not fully address the conversions of Repeat-While and Repeat-Until constructs in this thesis and will investigate the conversions in more detail in future work.

Since the conversion of If-Then-Else constructs can result in multiple FSAs, converting other control constructs (such as Sequence, Choice, Split, Split+Join and Any-order) has to take these FSAs into account. As an example, Figure 5-4 illustrates the conversion of a Sequence construct consisting of two If-Then-Else constructs, denoted as: Sequence(If-Then(P1)-Else(P2), If-Then(P3)-Else(P4)). To convert this Sequence construct, the two constituent If-Then-Else constructs are converted first. The first If-Then-Else construct is converted into two FSAs, FSA1 corresponds to process P1, and FSA2 corresponds to P2; similarly the second If-Then-Else construct
is converted to FSA3 and FSA4. The next step is to combine the FSAs of these two If-Then-Else constructs as part of the Sequence construct by using $\varepsilon$ transitions. We create four combinations, each of which contains one FSA from the first If-Then-Else construct and one FSA from the second construct: \{FSA1, FSA3\}, \{FSA1, FSA4\}, \{FSA2, FSA3\} and \{FSA2, FSA4\}. For each combination, we join the FSAs in the combination by using $\varepsilon$ transitions to link the final states of a given FSA to the initial state of the next one since they are part of a Sequence construct. As a result, the Sequence construct in this example is converted into four FSAs as illustrated in Figure 5-4.

![Figure 5-4: An illustration of converting a Sequence control construct](image)

A Web service’s OWL-S description can be converted to multiple FSAs (service-FSAs) by the OWL-S to FSA conversion process. All the service-FSAs need to match to the FSA representing a given role-centric protocol in order for the Web service to be considered matched to the role’s behavioural requirement (see Section 5.3.2). The implementations of the OWL-S parser, OWL-S to FSA conversion mechanism and the compatibility checking are detailed in Chapter 7.

Since $\varepsilon$ transitions are added to the resulting FSAs during the conversions, the resulting FSAs are non-deterministic FSAs. However, our FSA model of IRS does not have such transitions and it is a deterministic FSA. This would lead to some problems during the compatibility checking. Hence, we convert non-deterministic OWL-S’s FSAs to deterministic ones by using the subset construction algorithm (Hopcroft et al. 2001). This algorithm is detailed in Listing 5-1.

Figure 5-5 shows an example structure of an OWL-S process specification. This example process model contains a Sequence of an atomic process quote and a Choice construct. The Choice construct contains a Sequence of two atomic processes (order and payment) and an atomic process rejectQuote. Figure 5-6 illustrates the resulting
FSA of OWL-S to FSA conversion. As part of the conversion, this FSA contains $\varepsilon$ transitions. Figure 5-7 shows the result of converting this non-deterministic FSA into a deterministic FSA. In our implementation, state numbers are reindexed for each FSA and are only used as state labels during the visualisation. This is why the states in the FSAs depicted in figures 5-6 and 5-7 have different numbering.

Figure 5-5: The structure of an example OWL-S process specification

Figure 5-6: Non deterministic FSA result of OWL-S to FSA conversion

Figure 5-7: A result of converting non-deterministic FSA to deterministic FSA
In this section, we discussed the conversion of OWL-S to FSA. Our approach is to define a mapping between elementary constructs of one specification to those of the other. A common challenge is that there is no direct one to one mapping from one specification to the other. For example, OWL-S has the notions of conditions and concurrency, whereas an FSA does not support those notions. It is important to ensure that the usage of the outputs from the conversion does not depend on the differences in the two specifications. For example, the Split and Split+Join constructs are modelled as Choice constructs, since the FSAs output from the conversions are used for the purpose of match-making only and do not depend on the concurrency execution of the operations.
Once the OWL-S processes are converted to FSAs, we then carry out match-making of all the FSAs against the FSA representing the role-centric protocol. The next section details the match-making process.

5.3 Match-making Approach and Mismatches Classification

The purpose of the service match-making is to find a Web service that is capable of fulfilling the requirements. The match making has to be performed at all the levels of service description; however, in this thesis, we are only concerned with the functional and behavioural levels. During match-making, the mismatches need to be analysed and identified, so that appropriate adaptation can be carried out.

5.3.1 Functional Capability Matching

Functional capability matching compares a role’s required functionality to a service’s offered functionality. As introduced in Section 3.2.2, the role’s required functionality is identified by terms in contracts that it is associated with. Terms are specified by Control Communication Acts (CCAs) and the valid directions of CCAs. For example, a contract involves a Buyer role (partyA of the contract) and a Vendor role (partyB of the contract), and it has a term specifying Order CCA with an AtoB direction, then the partyB of the contract (which is the Vendor role) has to be able to process such Order messages. Such required processing capability is then aggregated to form the role’s required functional capability.

For a service described in OWL-S, its offered functionality is specified in a Service Profile and a Service Process Model. The capability of a service is specified in terms of Inputs, Outputs, Preconditions and Effects (IOPEs) of the operations of the service. These descriptions are specified in ontology, which enables automated interpretation and reasoning.

As part of the functional capability matching, we need to map the CCA message types to service’s operations. Since the focus of this thesis is on behavioural protocol, we used a simple CCA-to-operation mapping mechanism which maps a CCA directly to a given service operation. For example, the CCA QuoteRequest will be mapped to the quote process of a book selling service described in OWL-S. A more formal
approach could be using ontology subsumption technique such as the one proposed by Paolucci et al. (2002) as discussed in Section 2.5.1.

The above capability matching is at the operation level. At the service level, we compare all the required operations to the provided operations of a candidate service. We define three categories of matching at the service functional level:

- **Sufficient match**: The service matches the required functionality when it can match all the required operations. A special case of this category is the *exact match* where the service’s functionality match exactly all the required operations.
- **Partial match**: We have a partial match when the service matches only some but not all of the required operations.
- **No match**: When the service’s operations do not match any of the required ones.

Let us consider the scenarios from Figure 3-1 to Figure 3-4 in Section 3.1. In Scenarios 1 and 3, the vendor-WS matches *sufficiently* at the service functional level to their respective requirements. In Scenarios 2 and 4, the vendor-WS matches *partially* to the requirements as there are missing functionality (e.g. delivery or payment receipt). The *no match* situation occurs when the Web service offers completely different functionality, for example hotel booking instead of selling books.

Note that in Scenarios 3 and 4, the vendor-WS has extra operations (*quoteAccepted* and *quoteNotAccepted*) than the required functionality. These extra operations do not affect the result of the match-making as they are not part of the required functionality and are not considered during the match-making process.

### 5.3.2 Behavioural Protocol Matching

Even if the lib-BB’s functional requirements are met by the vendor-WS (e.g. Scenario 3 in Figure 3-3), there may exist potential mismatches at the behavioural level. After the functionality matching is completed, the behavioural matching can be carried out. The behavioural matching compares the required protocol’s FSA (required-FSA) against the service behavioural description’s FSAs (service-FSAs) by using the FSA intersection algorithm (see Section 7.5.2). The service-FSAs are the FSAs resulted from the conversion of the service’s OWL-S specification to FSA as discussed in
Section 5.2. Since the conversion of If-Then-Else constructs might produce multiple FSAs, the conversion process of an OWL-S specification to FSA might also result in multiple FSAs (service-FSAs). These service-FSAs will be matched against the required-FSA and the results of the matching are classified into the following categories as illustrated in Figure 5-8.

- **Compatible**: All the required sequences of interactions are supported by the Web service. In terms of FSA, this means for every path from the initial state to a final state in the required-FSA, there is an identical path in all the service-FSAs. In general, the Web service may or may not support other sequences of interaction than the required; these extra sequences of interactions are of no interest from the role’s perspective. When the Web service does not support other sequences, it gives rise to the special case of precise behavioural match.

- **Partially compatible**: Only some but not all the required sequences of interactions are supported by the Web service. In terms of FSA, this means there is only a subset of the paths from the initial state to final states in the required-FSA is supported by the service-FSAs, but not all.

- **Incompatible**: The Web service does not support any sequence of interactions required. In terms of FSA, this means that the required-FSA and the service-FSAs do not have any common path between the initial state and the final states.

Each of the service-FSAs is matched against the required-FSA. If one of the service-FSA is considered incompatible, the result is incompatible. For the compatible case, all the service-FSAs have to be compatible. For other cases, the result is partially compatible.

From the above classification, a service satisfies the requirements in terms of behaviour when it is in the compatible class; and no adaptation is required. With the partially compatible and incompatible cases, some adaptations will be required in order to utilise the Web service. In the scenarios discussed in Section 3.1, Scenarios 1 and 2 represent partially compatible cases because both parties (vendor-WS and lib-BB’s requirements) support the sequence of interaction that ends right after the quote operation (denoted by a final state), but the vendor-WSs do not support all the sequences required by the lib-BBs. Scenarios 3 and 4 represent incompatible cases.
because the vendor-WSs do not support any acceptable sequence required by the lib-BBs.

<table>
<thead>
<tr>
<th>Functional</th>
<th>Behavioural</th>
<th>Sufficient</th>
<th>Partial</th>
<th>No match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatible</td>
<td>Case 1: No adaptation required</td>
<td>Case 4: Impossible to occur</td>
<td>Case 7: Impossible to occur</td>
<td></td>
</tr>
<tr>
<td>Partially Compatible</td>
<td>Case 2: (Scenario 1) Adaptation required</td>
<td>Case 5: (Scenario 2) Adaptation required</td>
<td>Case 8: Impossible to occur</td>
<td></td>
</tr>
<tr>
<td>Incompatible</td>
<td>Case 3: (Scenario 3) Adaptation required</td>
<td>Case 6: (Scenario 4) Adaptation required</td>
<td>Case 9: No interest</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-8: Behavioural protocol mismatches classifications

5.3.3 Considering both Functional and Behavioural Mismatches

By combining the functional and behavioural matching at the service level, we have the cases of mismatches as shown in Table 5-1.

Table 5-1: Functional and behavioural mismatches at the service level

Case 1 in Table 5-1 shows a perfect match when the Web service satisfies both the functional and behavioural requirements. In this case, the Web service can be used as is, and there is no adaptation needed. Cases 7, 8 and 9 in the ‘No match’ column either will never occur or are of no interest since the Web service does not offer any of the required operations. Case 4 will also never occur because a role’s FSA only contains paths from the initial states leading to final states, so a compatible behavioural match is predicated on all required operations being present.

The services in the remaining four cases (i.e. Cases 2, 3, 5 and 6) can be potentially adapted, should we want to utilise them. Table 5-1 also shows the classification of scenarios 1 to 4 to their corresponding mismatch types. In Chapter 7,
we will introduce some compositional adaptation patterns to handle these types of mismatches.

5.4 Summary

In this chapter, we have discussed the need to have a translation mechanism to convert role-centric protocol specification in Interaction Rule Specification (IRS) and service behavioural specification in OWL-S to the same underlying formalism (such as Finite State Automaton) in order to carry out the match-making process. The match-making is carried out by two separate phases: functionality matching and behavioural protocol matching. Instead of just a pass and fail criteria, we developed a classification of mismatches so that an adaptation mechanism can alleviate those mismatches. In the next chapter, we will discuss the adaptation mechanism in more detail.
Chapter 6:
Mismatch Adaptation

In Chapter 5, we have illustrated how the functional capability and behavioural protocol of a Web service can be matched against those of a role’s requirements. As part of the match-making process, mismatches are identified and categorised. Table 5-1 showed different types of mismatches. When a matched service cannot be found, adaptation is required should a mismatched Web service be used as a player for a given role. In this chapter, we introduce three adaptation patterns to overcome the functional and behavioural mismatches. In these patterns, a composite acts as an adaptor between a role and a Web service to overcome the functional and behavioural discrepancies between them. In Section 6.1, we evaluate two approaches to modifying a composite to adapt a mismatched Web service to a role’s requirements. We then discuss the three adaptation patterns used to create adaptor composites: Conversation Coordinator, Functional Role Composition, and Trusted Mediator in Section 6.2, 6.3 and 6.4 respectively. Section 6.5 discusses the advantages and disadvantages of our approach. Finally, Section 6.6 summarises the chapter.

6.1 Adaptor Composite

For a Web service to be a player of a role, it has to satisfy the role’s functional and behavioural specifications. If the Web service matches the role’s requirements (sufficient match in terms of functionality and compatible match in terms of behaviour), the Web service can be used as is, otherwise adaptation is required. We considered two typical approaches to adapt a mismatched Web service:

- **Structural change:** In order to adapt a mismatched Web service, the role’s requirements have to be changed so that the Web service can be compatible with the role’s new requirements. To change this role’s requirements without
affecting the overall original interaction requirements, we need to introduce intermediate roles and players between the role the Web service is to play and other roles. This approach modifies existing relationships between roles and also modifies the interaction constraints between roles in a composite. Figure 6-1 illustrates this approach. In Figure 6-1, the playerB did not satisfy the RoleB’s role-centric protocol aggregated from the contract between the RoleA and the RoleB. To adapt the playerB, intermediate roles and players are introduced so that interactions with the RoleB are based on the playerB’s required protocol and similarly for interactions with the RoleA. The players for adaptor roles are defined by developers.

**Figure 6-1: Adaptation by changing the structure of the composite**

- **Adaptor composite**: Instead of modifying the current composite, an intermediate composition, called an “adaptor composite”, is introduced between the Web service player and its role to alleviate the mismatches. The structure of an adaptor composite consists of intermediate roles and players. The adaptor composite also contains a role representing the Web service that it is adapting. Figure 6-2 illustrates an adaptor composite alleviating the mismatches between the playerB and the RoleB’s required protocol.

**Figure 6-2: Adaptation by adding an adaptor composite (legend as in Figure 6-1)**
The structural change approach might result in a simpler setup compared to the adaptor composite approach as it does not add an extra composite. However, it is quite intrusive by directly modifying the existing composite. Changing a composite’s structure involves creating and destroying contracts between roles. As a role can be bound to multiple contracts, this adaptation process can affect other roles and contracts as well. In the adaptor composite approach, the structure of the original composite is unchanged. From a role’s point of view, an adaptor composite is similar to any other player; therefore the adaptation process is transparent to the original composite. For these reasons, the adaptor composite approach is chosen as the adaptation technique presented in this thesis.

Depending on the type of mismatches as presented in Section 5.3.3, an adaptor composite’s structure can be set up in different ways. To address different mismatch types, adaptor composites are instantiated based on our proposed three adaptation patterns: Conversation Coordinator, Functional Role Composition, and Trusted Mediator. These patterns are discussed in more detail in the next sections.

### 6.2 Conversation Coordinator

When a Web service can provide all the required functionality (a sufficient match in terms of functionality), but has a partially compatible or incompatible match to the behaviour requirements of a role (Case 2 and 3 in the classification of mismatches described in Section 5.3.3), the adaptation will need to reorder the interaction messages between the role and the Web service. In these cases, the Conversation Coordinator adaptation pattern can be applied. This pattern produces adaptor composites that have the capability to reorder, synthesise, hide, split and merge messages.

- **Reorder**: Messages are buffered when they are not expected by the recipient and are delivered when the recipient is ready. For example, the recipient expects to a message containing delivery address at a later stage, an adaptor can store the message and send it to the recipient when the recipient is ready to process this message.
- **Synthesise:** Messages are generated based on information in previously sent messages. An example of a synthesised message is a notification of the number of rejected payments in a transaction.

- **Hide:** Messages are hidden from the recipient as the recipient does not expect to receive such messages. For example, acknowledgement messages can be discarded when the recipient does not need them.

- **Split:** Messages are split into two or more messages and are delivered separately to the recipient as required. For example, a message carrying both payment and delivery detail can be split into two messages.

- **Merge:** Two or more messages are merged into one message. The generated message has all the information in the messages that are being merged. For example, a single message contains the sum of a quote for an item and a delivery quote rather than two separate messages.

A Conversation Coordinator adaptor composite has a **MessageCoordinator** role and another role representing the mismatched player that it is adapting. The **MessageCoordinator** role has a player. The structure of an adaptor composite is generated by the framework with a **MessageCoordinator** role, its player and a role for the mismatched player. The framework also automatically binds roles to their corresponding players. The logic of what to do with each message is encoded in the **MessageCoordinator**’s player; and this configuration is performed by developers. Figure 6-3 illustrates an example of a Conversation Coordinator adaptor composite.

---

**Figure 6-3: An example of the conversation coordinator pattern (legend as in Figure 6-1)**

There are cases where the Conversation Coordinator pattern cannot be applied to resolve the message ordering mismatches: when introducing extra messages or hiding messages can affect the semantics of the interaction and when information available
is not enough to synthesise new messages. For example, receipt messages cannot be generated from previously exchanged information as they are related to invoice and taxation purposes.

The Conversation Coordinator pattern provides capabilities similar to those described by Benatallah et al. (2005) as discussed in Section 2.6.1. However, in our approach, the logic of message handling can be swapped by using a different MessageCoordinator’s player. In addition, the adaptor composite is more than just a message mediator, it is a structural entity providing us the flexibility to consider the cases where messages cannot be synthesised. In our approach, extra roles and players can be added to the composite, thus providing capabilities to address this issue. Section 6.3 and 6.4 will discuss this in more detail.

As an example, in Scenario 1 (Figure 3-1) presented in Section 3.1, there are mismatches between the message sequence expected by the Vendor role and the message sequence provided by the Web service. Because all the required functionalities are provided by the Web service, to mitigate the mismatch, the Conversation Coordinator pattern is applied and an adaptor composite consisting of the MessageCoordinator and Supplier roles is introduced between the Vendor role and the vendor-WS service (see Figure 6-3). The Supplier role represents the vendor-WS and will have the same behavioural requirement as the vendor-WS’s protocol. The MessageCoordinator role is responsible for buffering messages and delivering them (reordering of messages) in the sequence expected by the Supplier role in the same composite and by the Vendor role in the upper composite. In Figure 6-3, the LibraryBookBuying composite is the ‘upper composite’ of the adaptor composite, as the LibraryBookBuying composite contains the Vendor role which is played by the adaptor composite. The MessageCoordinator role is responsible for managing the flow of messages into and out of the adaptor composite. This capability is provided by the computational entity playing the role, e.g. the coordinator in Figure 6-3. In this scenario, upon receiving messages from the Vendor role via the MessageCoordinator role, the coordinator will forward those messages to the Supplier role. Those messages are then processed by the vendor-WS. When the coordinator encounters the payment message sent from the Vendor role, it will store this message and forward it to the
Supplier when the message is required. Listing 6-1 shows the pseudo-code of the message processing in the coordinator player in this example.

For Scenario 3 (Figure 3-3) in Section 3.1, the coordinator is designed to synthesise the quoteNotAccepted message when it does not receive any messages from the Vendor role in the upper composite after a specified time-out. In the case that the lib-BB accepts the quote and sends the order message, the coordinator will generate a quoteAccepted message before forwarding the order message to the Supplier role, which will then forward these messages to the vendor-WS. Listing 6-2 shows the pseudo-code of the message processing in the coordinator player in this example.

Listing 6-1: Pseudo-code of the message processing in a coordinator player for Scenario 1

```java
01  receive a quote message from the Vendor role, forward it to the Supplier role
02  forward the quote reply message from the Supplier role to the Vendor role
03  wait for an order message from the Vendor role
04  if (timeout) {
05      create a quote not accepted message and send it to the Supplier role
06  }
07  else {
08      forward the order message to the Supplier role
09      receive a payment message from the Vendor role, verify that the payment
10      information is correct, return a successful acknowledgement to the Vendor
11      role and store the message to the buffer
12      send the stored payment message to the Supplier role
13      receive a delivery detail message from the Vendor role, forward it to the
14      Supplier role
15      receive a delivery message from the Vendor role, forward it to the Supplier
16      role
17      receive a payment message from the Vendor role, verify that the payment
18      information is correct, return a successful acknowledgement to the Vendor
19      role and store the message to the buffer
20      send the stored payment message to the Supplier role
21      receive a delivery detail message from the Vendor role, forward it to the
22      Supplier role
23      receive a delivery message from the Vendor role, forward it to the Supplier
24      role
25      receive a payment message from the Vendor role, verify that the payment
26      information is correct, return a successful acknowledgement to the Vendor
27      role and store the message to the buffer
28      send the stored payment message to the Supplier role
29      receive a delivery detail message from the Vendor role, forward it to the
30      Supplier role
31      receive a delivery message from the Vendor role, forward it to the Supplier
32      role
33  }
```

Listing 6-2: Pseudo-code of the message processing in a coordinator player for Scenario 3

```java
01  receive a quote message from the Vendor role, forward it to the Supplier role
02  forward the quote reply message from the Supplier role to the Vendor role
03  wait for an order message from the Vendor role
04  if (timeout) {
05      create a quote not accepted message and send it to the Supplier role
06  }
07  else {
08      store the order message
09      create a quote accepted message and send it to the Supplier role
10      send the stored order message to the Supplier role
11      receive a payment message from the Vendor role, forward it to the Supplier
12      role
13      receive a delivery detail message from the Vendor role, forward it to the
14      Supplier role
15      receive a receive delivery message from the Vendor role, forward it to the
16      Supplier role
17  }
```
6.3 Functional Role Composition

When a candidate Web service does not offer all the required functionality but only has a *partial* match in terms of functionality, it corresponds to Case 5 and 6 in the mismatch classifications discussed in Section 5.3.3. In these cases, to adapt the Web service, other Web services are sought to provide the missing functionality. Once found, these Web services are combined in a composition; the Functional Role Composition pattern can be applied to provide such compositions.

A Functional Role Composition’s adaptor composite contains a `MessageCoordinator` role and other roles representing the Web services in the composition. The `MessageCoordinator` role and its player provide the necessary interaction coordination between other roles in the composite and between this adaptor composite and the upper composite. The framework will search for available Web services offering the missing functionality in a service directory and perform match-making analysis on those potential services. The results are listed for developers to choose the most appropriate ones. The structure of the adaptor composite is generated with binding among roles and from roles to their players. The interaction coordination among roles in the composite is defined by developers.

For example, in Scenario 2 (Figure 3-2) presented in Section 3.1, since the vendor-WS does not do delivery, a delivery service is searched for to fulfill the missing functionality. Assume that such a service is found (delivery-WS), an adaptor composite with the `MessageCoordinator`, `Supplier` and `Delivery` roles is instantiated (see Figure 6-4). The vendor-WS is bound to the Supplier role, while the delivery-WS is bound to the Delivery role. The message flow among the roles is controlled by the coordinator performing the `MessageCoordinator` role. Note that the complexity of the adaptor composite is abstracted away from the Vendor role and consequently the `lib-BB`. 
As the delivery-WS belongs to a different business entity, there is a cost for delivering the goods. Hence the quote and payment messages sent from the Vendor role will need to be split and routed to both the delivery-WS and vendor-WS appropriately. Upon receiving the quote message from the Vendor role, the coordinator will send two separate quote messages to the vendor-WS and delivery-WS. It then adds up both the quote replies and responds to the Vendor role. Similarly, the coordinator will split the order and payment messages to send to both the delivery-WS and the vendor-WS. The logic of how to manage the messages is encoded in the coordinator by developers. The pseudo-code of an example message processing in the coordinator player to resolve mismatches in Scenario 2 (Figure 3-2) is shown in Listing 6-3.
6.4 Trusted Mediator

In previous sections, we have discussed how the Conversation Coordinator and Functional Role Composition patterns can be used in trying to alleviate the mismatches between a Web service and a role’s requirements. However, there are cases when certain messages cannot be mediated by an adaptor or when certain functionalities have to be provided by the same Web service. For example, a “payment” message contains financial information and the payment liability of a specified amount, which cannot be generated by an adaptor, and a “receipt” message has to be originated from the Web service that previously received the payment. In these situations, the Conversation Coordinator and Functional Role Composition patterns will not be able to resolve the mismatches. In such cases, the Trusted Mediator pattern can be applied.

Similar to the Functional Role Composition pattern, a Trusted Mediator pattern’s adaptor composite also consists of a MessageCoordinator role and other roles.
Chapter 6  Mismatch Adaptation

representing the Web services in the composition. Unlike the Functional Role Composition pattern, even if all the required functionality is provided by a service, additional services may also be needed in order to resolve the mismatch.

For example, in Scenario 4 (Figure 3-4) in Section 3.1, the two library and the book vendor do not trust each other, and are unable to carry out the transaction (the library requires receiving delivery before payment, while the book vendor requires payment before delivery). The mismatch in the sequence of the delivery and payment messages cannot be overcome by a simple message mediator and cannot be generated by an adaptor, as those messages imply business liability (i.e. paying the other party and delivery of books). In addition, the vendor-WS does not provide the sending receipt functionality. Unlike the case that the delivery functionality can be provided by a separate service, if a service providing the sending receipt functionality is found, during service composition, we have to consider the sending receipt functionality as part of the payment process, not as an independent functionality. As such, developers will specify the required functionalities in each additional Web service before the framework searches for the required Web services for the adaptation. Then, similar to the Functional Role Composition pattern, the framework lists the search results so that developers can choose the most appropriate Web services to be used in the adaptor composite. The framework generates the structure of the composite with roles, their players and the corresponding bindings between them. Finally, developers define the flow of messages within the adaptor composite.

To resolve the mismatch in the above mentioned example, we would need to have a trusted third party payment service to pay the book vendor (before the library makes the payment) so that the transaction can continue. This payment service also provides receipts on payments made by the library. Figure 6-5 presents an adaptor composition to resolve this mismatch. The composite has the MessageCoordinator, Supplier and PaymentFacilitator roles. The trusted payment-WS is bound to the PaymentFacilitator role, and the vendor-WS to the Supplier role. After the books are ordered, the coordinator will request the payment-WS to pay the vendor-WS so that the books can be delivered to the library. Upon receiving the books, the lib-BB sends the payment message; then the coordinator forwards it to reimburse the payment-WS. Finally, the payment-WS sends the receipt message to the coordinator which will then forward it to the lib-BB for the
payment. Listing 6-4 shows the pseudo-code of the message processing in the coordinator player in this example.

```plaintext
Listing 6-4: Pseudo-code of the message processing in a coordinator player for Scenario 4
01 receive a quote message from the Vendor role, forward it to the Supplier role
02 forward the quote reply message from the Supplier role to the Vendor role
03 wait for an order message from the Vendor role
04 if (timeout) {
05 create a quote not accepted message and send it to the Supplier role
06 } else {
07 store the order message
08 create a quote accepted message and send it to the Supplier role
09 send the stored order message to the Supplier role
10 receive a delivery detail from the Vendor role, store the message
11 retrieve information of the Supplier’s player such as an ABN
12 create a request for a payment for an amount equivalent to the book order
13 value, payable to the Supplier’s player
14 send the payment request to the PaymentFacilitator role
15 receive a payment message from the PaymentFacilitator role, forward it to
16 the Supplier role
17 send the stored delivery detail message from the Vendor role to the Supplier
18 role
19 receive a receive delivery message from the Vendor role, forward it to the
20 Supplier role
21 receive a payment message from the Vendor role, forward it to the
22 PaymentFacilitator role
23 receive a receipt message from the PaymentFacilitator role, forward it to
24 the Vendor role
25 }
```

6.5 Applying the Adaptor Patterns and Discussion

In previous sections, we presented three adaptor patterns that can be used to alleviate the identified mismatches between available services and roles. These patterns are: Conversation Coordinator, Functional Role Composition and Trusted Mediator. The three adaptor patterns can be applied separately or in combination. In the Functional
Role Composition pattern, there are cases where message coordination is also needed; and in the Trusted Mediator pattern, both trusted facilitation and message coordination are required. Table 6-1 provides an overview of the mismatch classifications and their corresponding adaptation patterns for all the scenarios presented in Section 3.1.

Table 6-1: Mismatch classifications and adaptations for the Library Book Buying scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Mismatch classification</th>
<th>Adaptation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>Case 2</td>
<td>Conversation coordinator</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>Case 5</td>
<td>Functional role composition</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>Case 3</td>
<td>Conversation coordinator</td>
</tr>
<tr>
<td>Scenario 4</td>
<td>Case 6</td>
<td>Trusted mediator</td>
</tr>
</tbody>
</table>

The following provides a comparison of our approach with some adaptation approaches reviewed in Section 2.6.1 such as Yellin and Strom (1997), Benatallah et al. (2005) and Casati and Shan (2001). Yellin and Strom (1997) present adaptation by synthesising an adaptor from an interface mapping. An interface mapping consists of a set of rules which relate messages and parameters in the two given interfaces of two interacting parties. An adaptor can queue messages until being requested by the recipient. It can also synthesise new messages from the available interfaces and mapping information. In essence, the adaptor specification corresponds to our MessageCoordinator role and coordinator player. In our approach, adaptors are structural entities that can relate multiple parties and their internal structures can be changed at run-time by adding/removing roles and creating/revoking relationships (contracts) between roles. Although adaptors have to be encoded by developers and the adaptor patterns only provide a guide, our approach has an advantage in addressing different mismatches in a systematic manner and in a non-intrusive way thanks to the adaptability of composites.

Benatallah et al. (2005) introduce a number of specific patterns to resolve mismatches in message ordering: message ordering mismatch, extra message mismatch, missing message mismatch, message split mismatch and message merge mismatch. These patterns are all covered by our Conversation Coordinator pattern. However, their approach does not consider the cases where messages cannot be
generated or re-ordered as it might violate the business logic of services or compositions. Our approach addresses this problem by incorporating a third party service in the composition which will create the required messages and be responsible for any business liability implied by the messages such as payment messages.

Casati and Shan (2001) introduce process schema into service composition, where a process schema defines the order of execution among the service nodes in a process. Within a service node, a flow of method invocation nodes defines the actual operations to be invoked on the service and their execution dependencies. Compared to our approach, a service node roughly corresponds to a player or a composite (for a generic service node). A generic service node is instantiated to contain a list of other service nodes to achieve a task; while a composite can contain other roles and players which will perform the required work. In addition, our three adaptor patterns also provide a guide to search for the required services based on the mismatched in functionality and behaviour. Our adaptor composites are more flexible and their structures can be changed dynamically at run-time.

The disadvantage of our approach is that it does not provide a fully automated adaptor generation mechanism. It only provides a guide to specify the structures of adaptor composites. Developers would need to instantiate an adaptor composite and encode the message coordination between entities in the adaptor composite.

### 6.6 Summary

In this chapter, we have presented three adaptation patterns that are used to create adaptor composites: Conversation Coordination, Functional Role Composition and Trusted Mediator. The Conversation Coordination pattern is used to rearrange messages in interactions. The Functional Role Composition is applied to create a composition of additional services offering missing functionality. The Trusted Mediator is used when adapting mismatches in sequence of messages carrying business liability information that needs to be considered during message generation and functionality composition, such as messages carrying payment details. We then discussed the advantages and disadvantages of our approach. In the next chapter, we will describe a prototype implementation of the concepts presented in this thesis.
Chapter 7: Implementation

Previous chapters have given the theoretical background on different aspects of behavioural protocols that our proposed framework supports. These aspects can be summarised as: (i) behavioural protocols and conversations representations in the Role Oriented Adaptive Design (ROAD) framework, (ii) run-time protocol monitoring, (iii) match-making of protocols based on their FSA representations, and (iv) adaptation of mismatches. In this chapter, we will discuss how these aspects are implemented. In Section 7.1, we will discuss an overview of the framework’s design and the main packages. We then discuss each package in more detail in later sections. Particularly, in Section 7.2, we will present the core classes of the ROAD framework and their associations. The method to convert IRS and OWL-S to FSAs are detailed in Section 7.3. How the run-time protocol monitoring is achieved is presented in Section 7.4. We then show how the compatibility checking and adaptation are implemented in Sections 7.5 and 7.6; respectively. The implementation of the graphical representation of composites and behavioural protocols is discussed in Section 7.7. Finally, the chapter is summarised in Section 7.8.

7.1 System Design Overview

The concepts presented in this thesis are implemented by three major components: ROADMaker, ROADAdaptor and ROADGui; and the implementation is written in Java. Figure 7-1 shows each component, its packages and the dependencies between packages. In this section, we will summarise each of the component’s responsibilities. The implementation of each package will be discussed in later sections.
Figure 7-1: The dependencies between packages in UML version 2.0

- **ROADMaker**: This component is responsible for creating composite instances.
  - **road.core**: This package defines the core classes of the ROAD framework.
  - **road.protocol**: This package is responsible for representing and handling FSAs in the system. Its main functionality is to convert IRS constraints and OWL-S descriptions to FSAs.
  - **road.monitoring**: This package implements the monitoring of the conversation between roles at run-time. It checks the exchanged messages and stops messages that do not follow the constraints specified in the corresponding contracts and contract dependencies.

- **ROADAdaptor**: This component performs the compatibility checking and adapts the specified services in order to be able to play a role in a composite.
  - **road.compatibilitycheck**: It performs the functional and behavioural matching to see if services are compatible with role’s requirements.
• **road.adaptation**: Based on the result of the compatibility checking of the services for adaptation, this package will allow user to choose an adaptor pattern that can be used in order to adapt the mismatched services.

• **ROADGui**: This component is responsible for displaying composite structures, compatibility checking results and adaptor composites in a graphical user interface (GUI). It also handles user interaction in viewing the specified protocols and the results of the match-making process.

• **road.gui**: This package implements the above mentioned functionality.

### 7.2 The ROAD Framework’s Core Elements - the road.core package

The **road.core** package implements the core part of the ROAD framework. The **road.core** package defines the elementary classes such as Composite, Organiser, Role, Contract, Term, CCA and UtilityFunction as defined in Section 3.2. These classes were implemented by Colman (2006) and Pham (2006). The ROAD framework supports adaptability by two mechanisms: *indirection of association* and *indirection of instantiation*. Indirection of association is the ability to create and revoke contracts between roles in a system, thus modifying the relationships and interactions between components of a system. Indirection of instantiation is the ability to modify the binding between roles and players. If a player’s performance is below expectation, this player can be replaced by a different player offering the same functionality and behaviour.

In order to support these two mechanisms, the ROAD framework is built on a message based communication architecture. When the contract between roles is temporarily changed (*indirection of association*) or when the role’s player is replaced by a different player (*indirection of instantiation*), their bindings are temporarily broken. During this time, messages can be queued in the role and be forwarded to
new parties when the change is finalised. The messaging mechanism is realised by implementing a queue in the Role class. This is illustrated in Figure 7-2.

![Figure 7-2: The messaging mechanism](image)

A role communicates with other roles by sending messages via the contracts between them (Step 1 in Figure 7-2). The contracts then add the messages onto the queue of the other roles (Step 2). In each role, there is a message processing thread that waits for messages in the queue (Step 3). After retrieving a message off the queue, the message processing thread dispatches the messages to the role’s player if such player is attached to the role (Step 4). The message is then processed by the player and the result is sent back to the originating role by following the same process.

### 7.3 Protocol Support Mechanism - the road.protocol package

As discussed in Chapter 6, both the IRS constraints and the OWL-S descriptions have to be converted to FSAs before the match-making process can be performed. This conversion mechanism is implemented in the `road.protocol` package.

#### 7.3.1 Converting IRS constraints to FSAs

The implementation of the conversion of IRS constraints to FSA is based on the work of Jin and Han (2005a) and Phan (2006). In order to convert IRS constraints to FSAs, first of all, we need to write a parser that can interpret IRS constraints. We utilised the ANTLR (ANother Tool for Language Recognition) Parser Generator (Parr 2006) to generate the IRS constraint parser. A grammar of IRS is defined as input to the ANTLR Parser Generator engine. It then produces Java code that can parse IRS
constraints. During the parsing process, the parser will refer to a file containing a definition of the semantics of IRS elementary constraints in FSAs so that each elementary constraint can be translated to its corresponding FSA. The reader is referred to Section 3.4 for all IRS elementary constraints and their FSA semantics. For example, upon encountering an elementary constraint “P exists globally”, the parser will convert it to an FSA as illustrated in Figure 7-3.

![Figure 7-3: The FSA for elementary IRS constraint “P exists globally”](image)

After the elementary FSAs are created, they are composed to create a single FSA representing the specified protocol. The FSA composition is done by the FSA intersection algorithm (Hopcroft et al. 2001) as described in Section 3.4.4.

### 7.3.2 Converting OWL-S to FSA

In order to parse OWL-S descriptions, we utilised the Mindswap OWL-S API (Sirin 2005). This API was developed by the Mindswap group (Maryland Information and Network Dynamics Lab Semantic Web Agents Project) at the University of Maryland. The OWL-S API provides a Java API for programmatic access to read, execute and write OWL-S service descriptions.

The Mindswap OWL-S API allows us to access to each process defined in an OWL-S file in order to convert the OWL-S process description to FSA. For an Atomic Process, the conversion is straightforward; the corresponding FSA has one initial state and one final state. The transition from the initial state to the final state is the occurrence of the Atomic Process operation. For a Composite Process formed by other processes in a control construct (such as Sequence, Choice, etc.), each constituent process is converted to an FSA and is then joined together based on the control constructs they are in. As specified in Section 5.2, conditions cannot be evaluated at design time, Repeat-While and Repeat-Until control constructs are treated as unbounded occurrences; hence OWL-S descriptions having these control constructs will not be considered and the conversion to FSAs for these OWL-S descriptions will terminate. For an If-Then-Else control construct, we convert both
the If-Then-branch and the Else-branch into two separate FSAs. For this reason, converting a composite process can return a list of FSAs. To convert a control construct consisting of sub-processes, firstly we convert each sub-process to FSAs. We then create different combinations of sub-processes’ FSAs by taking one FSA from the list of FSAs of each sub-process. Within a combination, each FSA represents its corresponding sub-process in the control construct. The FSAs in each combination are then joined together based on the control construct that their processes are in. For the concept of how the constituent processes’ FSAs in a control construct are joined together and an example illustrating the conversion process, please refer to Section 5.2. Listings 7-1 to 7-6 present the pseudo-codes of these conversions.

```
01 function convertProcessToFSA(p) {
02     if process p is an AtomicProcess {
03         call to convertAtomicProcess function
04     }
05     else if process p is a CompositeProcess {
06         call to convertCompositeProcess function
07     }
08 }
09
10 function convertAtomicProcess(p) {
11     create an FSA having one initial state and one final state
12     create transition from the initial state to the final state having p as the transition event
13 }
14
15 function convertCompositeProcess(p) {
16     retrieve the control construct of p
17     case: control construct of p is in {
18         perform a process : call to convertProcessToFSA function
19         sequence : call to convertSequenceConstruct function
20         choice : call to convertChoiceConstruct function
21         ifThenElse : call to convertIfThenElseConstruct function
22         repeat-while : throw an exception, stop the process
23         repeat-until : throw an exception, stop the process
24         split : call to convertSplitConstruct
25         split+join : call to convertSplitConstruct
26         anyorder : call to convertAnyOrderConstruct
27     }
28     convert all non-deterministic FSAs in the resultList to deterministic FSAs as in Listing 5-1
29 }
```

Listing 7-1: Pseudo-code of converting an OWL-S process description to FSA
Chapter 7  Implementation

01 function convertSequenceConstruct {
02    resultList = create a new list to store FSAs
03    get all the constituent processes, convert each constituent process to FSA by
calling the convertProcessToFSA function, each constituent process can
return a list of FSAs
04    combinationLists = createCombination(constituent processes' FSA lists) //
//createCombination() is defined in Listing 7-6
05    for each list of FSAs (listA) in the combinationLists {
06        anFSA = concatenateSequenceOfFSAs(listA)
07        add anFSA to resultList
08    }
09    return the resultList
10 }

11 function concatenateSequenceOfFSAs(listA) {
12    resultFSA = create a new FSA with an initial state
13    get the first FSA off the listA, assign to resultFSA
14    for each FSA (anFSA) from the second FSA in the listA {
15        create epsilon (empty) transitions from final states of resultFSA to
initial state of anFSA
16        set final states of resultFSA and initial state of anFSA to normal states
17    }
18    return resultFSA
19 }

Listing 7-2: Pseudo-code of converting a Sequence construct to FSA

01 function convertChoiceConstruct {
02    resultList = create a new list to store FSAs
03    get all the constituent processes, convert each constituent process to FSA by
calling the convertProcessToFSA function, each constituent process can
return a list of FSAs
04    combinationLists = createCombination(constituent processes' FSA lists) //
//createCombination() is defined in Listing 7-6
05    for each list of FSAs (listA) in the combinationLists {
06        anFSA = concatenateChoiceOfFSAs(listA)
07        add anFSA to resultList
08    }
09    return the resultList
10 }

11 function concatenateChoiceOfFSAs(listA) {
12    resultFSA = create a new FSA with an initial state
13    get the first FSA off the listA, assign to resultFSA
14    for each FSA (anFSA) from the second FSA in the listA {
15        create epsilon (empty) transitions from final states of resultFSA to
initial state of anFSA
16        set final states of resultFSA and initial state of anFSA to normal states
17    }
18    return resultFSA
19 }

Listing 7-3: Pseudo-code of converting a Choice construct to FSA
Listing 7-4: Pseudo-code of converting an If-then-else construct to FSA

```java
function convertIfThenElseConstruct {
    resultList = create a new list to store FSAs
    get the then-process and the else-process
    convert the then-process to FSAs by calling the convertProcessToFSA function defined in Listing 7-1, add all the converted FSAs to resultList
    if there is no else-process {
        return the resultList
    }
    else {
        convert else-process to FSAs, add all the converted FSAs to resultList
    }
    return resultList
}
```

Listing 7-5: Pseudo-code of converting a Split/Split+Join and Any-order constructs to FSA

```java
function convertSplitConstruct {
    get all the processes of the construct
    create permutations of all the processes
    for each permutation {
        create a sequence of the processes in the permutation
        create a choice construct consisting of all sequences of permutation processes
        call to convertChoiceConstruct
    }
    function convertAnyOrderConstruct {
        // similar to the convertSplitConstruct
    }
```

Listing 7-6: Pseudo-code of calculating combinations of FSAs from multiple lists of FSAs

```java
// function to calculate combinations of FSAs by taking one FSA from each list
// of FSAs
function createCombination(multiple lists of FSAs) {
    listA = get the first list of FSAs
    for each FSA (anFSA) in listA {
        contentList = create a new list to store FSAs
        add anFSA to contentList
        add contentList to the resultLists
    }
    for each list of FSAs (listB) from the second list to the last list {
        resultLists = createCombinationWithNextList(resultLists, listB)
    }
    return resultLists
}

function createCombinationWithNextList(lists, listB) {
    resultLists = create a new list to store lists of FSAs
    for each list of FSAs (listA) from lists {
        for each FSA (fsaB) from listB {
            copyOfListA = create a copy of listA
            add fsaB to copyOfListA
            add copyOfListA to resultLists
        }
    }
    return resultLists
}
```
7.4 Run-time Monitoring - the road.monitoring package

This package implements the monitoring of the conversation between roles at run-time. The monitoring capability is implemented by the CoordinationContext class. There are coordination contexts in each contract and organiser. A CoordinationContext instance contains a stateful FSA of constraints defined in a contract or contract dependencies in the case of organisers, and a list of terms corresponding to the specified constraints. CoordinationContext instances are grouped in a CoordinationContext group for each parameter specified in the where-clauses of constraints. For example, if there are orderNo and deliveryNo parameters specified in the where-clauses of constraints, there will be two CoordinationContext groups keeping track of CoordinationContext instances.

The messaging mechanism of the ROAD framework makes it easy to intercept messages and verify them since messages are sent via Contracts. When a message is intercepted by a Contract, the Contract and the Organiser will identify the corresponding CoordinationContext instances based on the value of the parameters specified in the where-clauses of the constraints. Firstly, the CoordinationContext group is identified based on each of the message’s parameter. If there is no matching group, the parameter is skipped. Then the CoordinationContext instance within a group is identified based on the value of the parameter. When there is no matching CoordinationContext instance identified due to a new value of the parameter is encountered, a new instance of CoordinationContext is created. The CoordinationContext will then verify the message. The message is only allowed to go through when it satisfies three conditions:

- The message contains a valid Control Communication Act (CCA) and direction as specified in the Term. For example, QuoteRequest CCA message with direction AtoB or QuoteResponse CCA message with direction BtoA. If the message is QuoteRequest but the direction is BtoA, the message will be rejected.

- The CCA of the message is expected by the Term. For example, if a Quote Term consists of a sequence of QuoteRequest and QuoteResponse CCAs and the first message is QuoteResponse, the Term will reject the message.
The corresponding Term of the message is mapped to an event that will be accepted by the current state of the FSA for the next transition. For example, if the stateful FSA is at initial state and it only accepts the Quote event as the next transition, it will reject all other messages except the QuoteRequest and QuoteResponse messages of the Quote Term.

When a message is rejected by the CoordinationContext, the sender will be notified and the message is discarded. Otherwise, the message will be delivered to the recipient role and the Term will advance to the next sequence. Once the Term is completed, the stateful FSA will be transitioned to the next state. For example, for the FSA in Figure 7-4, at the initial state, it expects the quote event which is mapped to the Quote Term consisting of QuoteRequest (AtoB) and QuoteResponse (BtoA). At this state, only QuoteRequest CCA message with direction AtoB is allowed. Upon receiving the QuoteRequest CCA message, the Term is updated to expect the QuoteResponse CCA message. After receiving QuoteResponse CCA message with direction BtoA, the Term is completed, and the FSA will be transitioned along the quote transition to the next state. Listing 7-7 details the pseudo-code of the run-time protocol monitoring.

**Figure 7-4: An FSA of a Contract's constraints**
Chapter 7  Implementation

Listing 7-7: Pseudo-code of run-time protocol monitoring in Contract and Organiser

```
01 function processMessage {
02 for each message parameter {
03     get the coordination context group corresponding to the parameter
04     in the coordination context group, get the coordination context instance
05         for the parameter value
06         if there is no corresponding coordination context instance {
07             create a new coordination context for this parameter value
08             assign the constraints FSA to coordination context with current state
09             set to initial state
10             add the coordination context to the group
11         }
12         check the message and direction against the coordination context
13     }
14 }

14 function check message and direction against the coordination context {
15     get the corresponding Term from the message’s CCA and direction
16     if the next CCA and direction of Term == the message’s CCA and direction {
17         get the Term’s operation name
18         if statefulFSA accepts Term’s operation {
19             advance the Term to the next sequence
20             if Term is completed {
21                 advance the FSA to the next state following the transition
22                 corresponding to the Term’s business operation
23             }
24         }
25         else {
26             reject the message and inform that the message is not accepted by FSA
27         }
28     }
29     else {
30         reject the message and inform that the message is not accepted by Term
31     }
```

7.5 Compatibility Checking – the road.compatibilitycheck package

The road.compatibilitycheck package is responsible for checking if a given service matches a role’s requirement in terms of both functional and behavioural properties. The compatibility checking takes services’ OWL-S descriptions and roles’ requirements as input; and it returns the checking result. The compatibility checking process has two steps: functional checking and behavioural checking.

7.5.1 Functional compatibility checking

The functional compatibility checking concentrates on functional matching between services’ capabilities and roles’ requirements. The matching result is output based on three different matching degrees: sufficient, partial or no-match as defined in Section 5.3.1.
Before the functional capability matching can be carried out, we need to map the CCA message types to services’ operations. In this thesis, the focus is on behavioural protocols. We assume such a mapping exists by describing inputs, outputs, preconditions and effects (IOPEs) of CCA message types in terms of ontology and the matching between roles’ required capability and services’ offered capability is realised by the *ontology subsumption* reasoning such as the technique proposed by Paolucci et al. (2002). In the proof of concept implementation, we have implemented a simple CCA-to-operation mapping mechanism which maps a CCA directly to a given service operation. For example, the CCA QuoteRequest will be mapped to the quote process of a book selling service described in OWL-S.

### 7.5.2 Behavioural compatibility check

The behavioural compatibility checking performs the *behavioural* matching of services’ capability and roles’ requirements. There are three different degrees of matching: *compatible*, *partially compatible* or *incompatible* as defined in Section 5.3.2. To carry out the matching process, we firstly convert a service’s OWL-S process description to FSAs (ref. Section 7.3.2). The resulting FSAs of this conversion, known as ‘the service’s FSAs’, are then compared to a role’s FSA to determine the compatibility of the service to the role. Each of the service’s FSAs is matched against the role’s FSA by using an FSA intersection algorithm (Hopcroft et al. 2001). Listing 7-8 shows the pseudo-code of an implementation of the FSA intersection algorithm by Phan (2006).
Chapter 7  Implementation

Listing 7-8: FSA intersection pseudo-code (from (Phan 2006))

```java
01  function fsaIntersect(FSA fsaOne, FSA fsaTwo) {
02      create an FSA, assign to fsaIntersection
03      initState1 = initial state of fsaOne
04      initState2 = initial state of fsaTwo
05      combinedInitState = create a state with a combined name of initStatel and
06      initStatet2
07      add combinedInitState to fsaIntersection, set it as initial state
08      create an empty processedStateSet to keep track of the processed states as a
09      triple of <state, state, event>
10      call to recursiveIntersectionCalc(fsaIntersection, fsaOne, fsaTwo,
11      combinedInitState, initStatet1, initStatet2, processedStateSet)
12      return fsaIntersection
13  }
14
15  function recursiveIntersectionCalc(FSA fsaIntersection, FSA fsaOne, FSA fsaTwo,
16      State combinedState, State stateOne, State stateTwo, Set processedStateSet) {
17      eventSetOne = outgoing transitions' events of stateOne in fsaOne
18      eventSetTwo = outgoing transitions' events of stateTwo in fsaTwo
19      eventIntersection = calculate set intersection of eventSetOne and
20      eventSetTwo
21      for each event (e) in eventIntersection {
22          if processedStateSet does not have a triple of (stateOne, stateTwo, e) {
23              newStateOne = the next state of fsaOne by following transition of
24              event e
25              newStateTwo = the next state of fsaTwo by following transition of
26              event e
27              combinedNewState = create a combined state of (newStateOne,
28              newStateTwo)
29              add combinedNewState to fsaIntersection if it is not already added
30              create a new transition from combinedState to combinedNewState with
31              event e, add this transition to fsaIntersection
32              add a triple of (stateOne, stateTwo, e) to processedStateSet
33              call to recursiveIntersectionCalc(fsaIntersection, fsaOne, fsaTwo,
34              combinedNewState, newStateOne, newStateTwo, processedStateSet)
35          }
36      }
37  }
```

The result of the intersection between the role FSA and a service’s FSA is an
FSA that we refer to as $FSA_{intersection}$. In the case that the $FSA_{intersection}$ does
not have any path from the initial state to final states (i.e. the $FSA_{intersection}$ is
empty), it corresponds to the incompatible case.

If the $FSA_{intersection}$ has some paths from the initial state to final states (i.e.
the $FSA_{intersection}$ is not empty), we will evaluate if these paths are exactly the
same as all the paths in the role’s FSA, which corresponds to the compatible case. We
compute the intersection of the role’s FSA with the complement of $FSA_{intersection}$.
A complement of an FSA is calculated by firstly convert the FSA to its total FSA. A
total FSA contains a state called trap state, and we create a transition to the trap state
for every rejected event at every state of the original FSA. At the trap state, there is a
self-reflective transition consisting of all events in the set of input alphabet. It is
called trap state because once the FSA is in this state, it cannot be transitioned out of
this state. For example, consider the FSA in Figure 7-5(a) and a set of input alphabet
consisting of \{A,B,C\}. At state 0, the FSA only accepts event A. In the corresponding total FSA in Figure 7-5(b), there is a transition consisting of events B and C to the trap state at state 0. Similarly for the states 1, 2 and 3. At the trap state, there is a self-reflective transition containing all events in the input alphabet: A, B and C.

Once the total FSA is obtained, the complement of the original FSA is calculated by setting all non-final states of the total FSA to final states and vice versa. For example, in the total FSA in Figure 7-5(b), we set all non-final states: 0,1,2 and Trap to become final states, and set the final state 3 to become a non-final state. The result is the FSA shown in Figure 7-5(c).

If the result of the intersection of the role’s FSA with the complement of FSA_intersection is an empty FSA, we will conclude that this is the compatible case. If the resultant FSA is not empty, it corresponds to the partially compatible case.

The above description discusses how the matching for one of the service’s FSAs to a role’s FSA is carried out. To match a service to a role, all of the service’s FSAs need to be matched against the role’s FSA. A service is considered compatible to a role when all of its FSAs are compatible with the role’s FSA. When one of the service’s FSA is incompatible with the role’s FSA, the matching process is terminated and outputs that the service is incompatible with the role. Otherwise, the service is considered partially compatible to the role. The above steps are summarised in the algorithm in Listing 7-9.
7.6 Adaptation - the road.adaptation package

The implementation of road.adaptation does not aim at fully automating the adaptation process, but developers can design such adaptors based on three patterns: Conversation Coordinator, Functional Role Composition and Trusted Mediator as defined in Chapter 7.

The compatibility checking result between potential services from a service registry against a role’s requirements is displayed to the user (see Section 7.7). When there is a functional mismatch, the user can select which services he/she would like to use. Based on the missing functionality, a suggestion of services that are the best matches to role’s requirements will be displayed during the selection process. Once the functional matching is done, behavioural compatibility checking is carried out. When there is a mismatch, the user can select the adaptor pattern (Conversation Coordinator, Functional Role Composition, or Trusted Mediator) to be applied. The

Listing 7-9: Pseudo-code of matching a service’s FSAs to a role’s FSA

```plaintext
01 function fsaMatching(roleFSA, serviceFSA) {
02     calculate intersection of roleFSA and serviceFSA, the result is stored in fsaIntersection
03     if fsaIntersection is empty {
04         this is the incompatible case
05     } else {
06         calculate the complement of fsaIntersection
07         calculate the intersection of roleFSA and complement of fsaIntersection, the result is stored in resultFSA
08         if resultFSA is empty {
09             this is the compatible case
10         } else {
11             this is the partially compatible case
12         }
13     }
14 }
15
16 function compatibilityChecking(roleFSA, listOfServiceFSAs) {
17     for each of FSA (aFSA) in listOfServiceFSAs {
18         result = fsaMatching(roleFSA, aFSA)
19         if (result == “incompatible”) {
20             output “incompatible”
21         } else {
22             store result to a map where the key is aFSA, the value is result
23         }
24     }
25     if all the values of the map is “compatible” {
26         output “compatible”
27     } else {
28         output “partially compatible”
29     }
30 }
31
32 }
33
34 }
```
chosen adaptor pattern will be applied and the user is directed to the design process where the adaptation composite is designed and message sequences between roles are defined. This process is illustrated in Figure 7-6.

![Figure 7-6: Overview of the adaptation process](image)

### 7.7 GUI Display - the road.gui package

The main functionality of the **road.gui** package is to visualise composites in a graphical user interface (GUI) and to handle user interaction in viewing the specified protocols and the results of the match-making process. This package uses the JGraph open source library (JGraph Ltd 2007). JGraph is an open source graph visualisation library written in Java. JGraph is designed based on the MVC (Model View Controller) pattern. The BasicMarqueeHandler class in the library provides us an easy way to handle MouseEvent when interacting with various nodes in the graph. Figure 7-7 shows the visualisation of two composites.

To visualise FSAs (FSA intersection results are also FSAs), we describe FSAs declaratively by using the DOT language (Gansner et al. 2006) and Graphviz (Graph Visualization Software - [http://www.graphviz.org/](http://www.graphviz.org/)) to render it. The advantage of using Graphviz is that we are not concerned about the layout, we only define the
structure of the graph (or FSA) and Graphviz will do the layout and rendering. Once 
FSAs are rendered to images, these images are then displayed in the GUI.

Figure 7-7: Visualisation of composites by using JGraph

Figure 7-8 shows a compatibility checking result of a service against the Vendor 
role’s requirements. The top section shows a summary of the compatibility checking 
result in functional and behavioural categories as defined in Section 5.3. The next 
section of the figure presents the service’s functional and behavioural capabilities, 
respectively. Note that the image shown is the FSA representing the service’s 
behavioural protocol after converting the service’s OWL-S description to FSA. The 
last section shows the compatibility checking result of the service against the role’s 
requirements. The image shown in this section is the FSA intersection of the role’s 
behavioural protocol and the service’s protocol.
Chapter 7  Implementation

Figure 7-8: Visualising FSAs and compatibility checking results

7.8 Summary

The behavioural protocol extension to the ROAD framework presented in this thesis provides a consistent way to define constraints of the required interactions between roles and services, to monitor the interaction at run-time, to analyse compatibility of candidate services, and to adapt the identified mismatches. This chapter showed the implementation of a proof of concept prototype. The prototype has been developed in Java by using tools and libraries such as ANTLR (Parr 2006), Mindswap OWL-S API (Sirin 2005) and JGraph (JGraph Ltd 2007). The implementation has three main components: ROADMaker, ROADAdaptor and ROADGui. The ROADMaker is responsible for the core aspect of the ROAD framework, protocol representation and run-time monitoring of interactions. The ROADAdaptor performs compatibility checking and mismatch adaptation. Finally, the ROADGui implements the GUI and user interaction. In the next chapter, we will present the analysis and discussion of our behavioural protocol approach.
Chapter 8: Analysis and Discussion

Previous chapters presented the concept and implementation of our behavioural protocol framework. In this chapter, we will discuss its advantages and limitations in Section 8.1 and 8.2. To show the capability of our framework in terms of response time, results of performance testing are detailed in Section 8.3. Finally, Section 8.4 summarises the chapter.

8.1 Advantages of Our Behavioural Protocol Approach

In the literature review in Chapter 2, we have identified three main areas that a behavioural protocol framework has to support:

- Behavioural interface description
- A well defined match-making process
- Adaptation to identified mismatches

In this section, we recap on how the above three criteria are achieved by the behavioural protocol approach presented in this thesis and discuss its advantages.

8.1.1 Behavioural Interface Description

The Interaction Rule Specification (IRS) that we used is a declarative approach to specify interaction constraints. The IRS was developed by Jin and Han (2005a) and is based on the Specification Pattern System (SPS) by Dwyer et al. (1999). The main benefits of using IRS are:

- **Declarative description**: A declarative approach results in simpler descriptions and requires one-third to one-sixth the number of statements to represent rules compared to other imperative approaches (Gottesdiener 1997).
A declarative approach is easier to understand and to be evaluated for correctness.

- **Incremental specification:** Each pattern in IRS defined a temporal constraint specifying a recurring occurrence or absence of events in an interaction protocol. We only need to specify the constraints of events of interest rather than specifying everything in other formalisms such as the $\pi$-calculus. In addition, we can incrementally specify constraints. When an imperative approach such as WS-BPEL and OWL-S is used to specify a business process, it is required that the interaction processes are well-defined and understood. However, in the case when interaction processes are not foreseen, an incremental approach is more suitable.

- **Relatively easy to modify:** This is a direct result of an incremental specification. With an imperative approach such as WS-BPEL or OWL-S, the protocol and its dependencies are hard-coded into the orchestration specification. This imperative specification has limited support for changes. By using IRS, in the case there are changes to the interaction protocol, we only need to remove obsolete constraints and add new ones as shown in Section 4.1.

- **Expressiveness:** The expressiveness of the constraint patterns has been shown by Dwyer et al. (1999). The authors have collected a large sample of specifications that suggest that most property specifications are instances of constraint patterns.

### 8.1.2 Behavioural Match-Making

In our behavioural match-making approach, we converted IRS constraints and Web services’ protocol specifications (in OWL-S) to FSA and the match-making is carried out based on results of FSA intersection. The result of behavioural match-making is then classified into three degrees of matching: compatible, partially compatible and incompatible instead of just pass/fail criteria. The classified match-making results provide more information so that adaptation can be carried out more easily.

Our approach provides a consistent way of evaluating behavioural compatibility based on the FSA formalism. In the current heterogeneous world of services where each service might be specified in different specifications (e.g. OWL-S, WSMO,
WSDL-S, etc), it is necessary to have a common base in order to evaluate them. However, our approach is not fixed to any specific specification; for different specifications, converters to FSA can be written to convert a specific specification to FSA before the evaluation in FSA can be carried out. For example, Wombacher et al. (2004b) proposed technique to convert WS-BPEL to annotated FSA.

8.1.3 Mismatch Adaptation

In cases where mismatches happen, existing services can be recomposed to meet the requirements (in contrast to the more costly or even prohibitive option of developing new services). The adaptations needed may differ depending on the type and degree of mismatch. We proposed three adaptor patterns that can be used: conversation coordinator, functional role composition, and trusted mediator. Section 6.5 provided a discussion of the advantages and disadvantages of our adaptation approach compared to other adaptation approaches such as (Yellin and Strom 1997; Benatallah et al. 2005; Casati and Shan 2001).

8.2 Limitations

Section 8.1 highlights the advantages of the behavioural protocol approach proposed in this thesis. This section discusses the limitations.

IRS is expressive in specifying ordering and occurrence of events of interest, however, for a business transaction to be viable, real-time constraints must also be considered (Benatallah et al. 2003). An example of a real-time constraint is “the payment has to be made within 10 days of delivery”.

As identified in Chapter 5, functional matching is also very important in addition to behavioural matching. In this thesis, we concentrated on behavioural matching and only defined an elementary functional matching. During behavioural matching between potential services against roles, services’ OWL-S descriptions are converted to FSAs. Since the values of conditions in Repeat-While and Repeat-Until constructs cannot be evaluated at design time, these constructs are considered unbounded occurrences and cannot be matched against roles’ centric protocols. OWL-S descriptions having these constructs will be considered not matched and will not be evaluated further. For If-Then-Else constructs, the If-Then-branch and the Else-
branch of an If-Then-Else construct are converted to two separate FSAs. All the generated FSAs have to match to a role’s centric protocol during match-making in order for the corresponding Web service to be considered a match. This matching requirement is quite strict, as all the branches of the If-Then-Else have to match to a role’s protocol. In addition, in the worst case scenario, this approach results in $2^n$ FSAs, where $n$ is the number of the If-Then-Else constructs.

Roles’ centric protocols are also converted to FSAs which are matched against FSAs of potential services’ OWL-S specifications. Both the conversion process of roles’ centric protocols to FSAs and behavioural compatibility checking are based on FSA intersection algorithm. Under extreme cases, the FSA intersection can result in state space explosion and the execution time of the FSA intersection computation can be a bottleneck to the whole process (see Section 8.3.1).

The adaptor patterns provide an architectural approach to adaptation. However, the adaptation process is only semi-automated and still requires a developer’s interactions to define message flows within an adaptor composite and to choose the appropriate Web services to be used in an adaptor composite.

### 8.3 Performance Analysis

In this section, we will analyse the overhead imposed by the behavioural protocol support mechanism presented in this thesis in a prototype implementation. The overhead is analysed in two processes: (i) FSA composition of behavioural protocols’ constraints and behavioural compatibility checking, and (ii) conversion from OWL-S to FSA.

#### 8.3.1 FSA Intersection Analysis

The FSA intersection overhead is introduced during composition of constraints and compatibility checking. As presented in the FSA intersection algorithm in Listing 7-8, an intersection of two FSAs is built up by walking the two operands’ FSAs from the initial states following all common transitions. Therefore, the time taken to compute the FSA intersection depends on the number of states and the number of common transitions between two FSAs.
Chapter 8

Analysis and Discussion

The FSA intersection algorithm produces a Cartesian product of states; this could result in a state space explosion problem. In fact, in the worst case scenario, the number of states increases at the rate of $2^n$, where $n$ is the number of FSAs being composed; and the growth rate of the number of transitions exceeds by far the growth rate of states (Markus et al. 2008). However, when the number of common transitions of two operands’ FSAs is small, the number of states and transitions in the intersection FSA can be significantly reduced. For example, for two FSAs that do not have any common transitions (i.e. the best case), the FSA intersection process terminates at the initial states.

IRS constraints are converted to FSAs and are then composed. The number of common transitions between two FSAs is affected by two factors: (i) the number of events in an event set; and (ii) whether constraints are related to each other or not. These two factors are explained below.

- **The number of events in FSAs’ event set:** The number of transitions depends on the number of events in the event set of FSAs. For example, in the FSA corresponding to a constraint “P precedes Q globally” shown in Figure 8-1, the event $O$ represents a set of all other events besides the events of interest (i.e. P and Q in this case). For an event set containing four events \{P, Q, R, S\}, the set of other events will be \{R, S\}; hence there are two transitions from state 0 to state 0 and are denoted by a triple of (fromState, event, toState): (0, R, 0) and (0, S, 0). Similarly there are four transitions from state 1 to state 1: (1, P, 1), (1, Q, 1), (1, R, 1) and (1, S, 1). Hence, a larger event set will result in a larger number of transitions.

  ![Figure 8-1: The FSA of “P precedes Q globally”](image)

- **Constraints are related:** When constraints are related to each other, each constraint reduces the number of transitions in the FSA. The FSAs generated during composition will have less common transitions compared to the case of composing unrelated constraints. For example, it is expected that composing two related constraints in Listing 8-1 will be completed sooner than
composing two unrelated constraints in Listing 8-2; and the resulting FSA from composing constraints in Listing 8-1 has smaller numbers of transitions and states compared to the one from composing constraints in Listing 8-2.

**Listing 8-1: Related constraints**

```
Vendor.order precedes Vendor.deliveryDetail globally;
Vendor.order leads to Vendor.deliveryDetail globally;
```

**Listing 8-2: Unrelated constraints**

In order to quantify the performance, a number of tests were run and the execution times were recorded. To quantify the impact of the number of events in an event set, we chose 5 and 12 event sets as shown in Listing 8-3. The 5-event set is likely to be used in a composite containing a Vendor and a Buyer role; and the 12-event set is likely from a composite containing a Vendor, a Delivery and a PaymentFacilitator role. To analyse the effect of unrelated constraints, we chose only exist globally constraints to ensure that these constraints are not related. Hence, from the 12-event set, we had 12-exist constraints as shown in Listing 8-4. Using the same number of constraints, the 12-related constraints are shown in Listing 8-5. These constraints were used for testing only; they did not necessarily model a business scenario.

**Listing 8-3: 5-event set and 12-event set**

(a) 5-event set

```
Vendor.quote
Vendor.order
Vendor.deliveryDetail
Vendor.payment
Vendor.delivery
```

(b) 12-event set

```
Vendor.quote
Vendor.order
Vendor.deliveryDetail
Vendor.payment
Vendor.delivery
Delivery.quote
Delivery.order
Delivery.payment
Delivery.delivery
PaymentFacilitator.quote
PaymentFacilitator.payment
PaymentFacilitator.receipt
```

**Listing 8-4:**

```
Vendor.quote
Vendor.order
Vendor.deliveryDetail
Vendor.payment
Vendor.delivery
Delivery.quote
Delivery.order
Delivery.payment
Delivery.delivery
PaymentFacilitator.quote
PaymentFacilitator.payment
PaymentFacilitator.receipt
```

**Listing 8-5:**

```
Vendor.quote
Vendor.order
Vendor.deliveryDetail
Vendor.payment
Vendor.delivery
Delivery.quote
Delivery.order
Delivery.payment
Delivery.delivery
PaymentFacilitator.quote
PaymentFacilitator.payment
PaymentFacilitator.receipt
```
We were interested in the growth of composition time versus the number of constraints in the following four scenarios:

- **Scenario 1**: analysed composition from 2 to 12 related constraints using the 5-event set. We used the 5-event set in Listing 8-3(a), and gradually adding more constraints to the composition, starting from 2 and increasing to 12 constraints from those listed in Listing 8-5.

- **Scenario 2**: analysed composition from 2 to 5 unrelated constraints using the 5-event set in Listing 8-3(a). Since there were only 5 events, we could use only 5 exist constraints in this case. In this scenario, we used the first 5 constraints in Listing 8-4.

- **Scenario 3**: analysed composition from 2 to 12 related constraints in Listing 8-5 using the 12-event set in Listing 8-3(b).

- **Scenario 4**: analysed composition from 2 to 12 unrelated constraints in Listing 8-4 using the 12-event set in Listing 8-3(b).

The tests were run on a machine with the following specification: Intel Core 2 Duo 2.4 GHz CPU, 3 Gb RAM, Windows Vista Ultimate SP1, and Java JDK version 1.6.0_07. Each test result was recorded based on calculating the average of the execution times of 50 executions. The results are shown in Table 8-1.
In Scenario 1 and Scenario 3, the constraints that were composed were the same; the only difference was the number of events in the event-set. We can see from the results in Table 8-1, adding more events to the FSAs’ event set increases composition time. In Scenario 1 and 3, after 8 constraints, the growth of the composition time is reduced; and for 5-event set in Scenario 1, execution times are almost unchanged. This is because up to a certain number of constraints, adding more related constraints will make the resulting FSA simpler (the numbers of transitions and states are smaller) and composition time does not increase as much. Table 8-2 shows the numbers of transitions and states indicating the relative complexity of the FSAs. Figure 8-2 shows a graph of response times in Scenario 1 and Scenario 3.

Table 8-1: Mean execution time in milliseconds for composing constraints in four scenarios

<table>
<thead>
<tr>
<th>No. of constraints</th>
<th>Scenario 1: Response time for composing related constraints (in ms)</th>
<th>Scenario 2: Response time for composing unrelated constraints (in ms)</th>
<th>Scenario 3: Response time for composing related constraints (in ms)</th>
<th>Scenario 4: Response time for composing unrelated constraints (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9.8</td>
<td>13.3</td>
<td>27.3</td>
<td>34.4</td>
</tr>
<tr>
<td>3</td>
<td>14.7</td>
<td>26.6</td>
<td>42.3</td>
<td>66.8</td>
</tr>
<tr>
<td>4</td>
<td>23.6</td>
<td>42.2</td>
<td>58.4</td>
<td>131.1</td>
</tr>
<tr>
<td>5</td>
<td>30.7</td>
<td>72.7</td>
<td>82.3</td>
<td>254.6</td>
</tr>
<tr>
<td>6</td>
<td>37.9</td>
<td>115.8</td>
<td>115.8</td>
<td>513.6</td>
</tr>
<tr>
<td>7</td>
<td>44.1</td>
<td>155.9</td>
<td>155.9</td>
<td>1,233.3</td>
</tr>
<tr>
<td>8</td>
<td>48.1</td>
<td>210.3</td>
<td>210.3</td>
<td>3,428.3</td>
</tr>
<tr>
<td>9</td>
<td>48.5</td>
<td>214.4</td>
<td>214.4</td>
<td>13,775.6</td>
</tr>
<tr>
<td>10</td>
<td>48.6</td>
<td>221.1</td>
<td>221.1</td>
<td>81,068.2</td>
</tr>
<tr>
<td>11</td>
<td>48.8</td>
<td>239.6</td>
<td>239.6</td>
<td>623,335.8</td>
</tr>
<tr>
<td>12</td>
<td>49.6</td>
<td>254.8</td>
<td>254.8</td>
<td>5,217,238.5</td>
</tr>
</tbody>
</table>

Table 8-2: Numbers of transitions and states of the resulting FSAs when composing constraints in four scenarios

<table>
<thead>
<tr>
<th>No. of constraints</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td># transitions: 12 # states: 3</td>
<td># transitions: 20 # states: 4</td>
<td># transitions: 33 # states: 3</td>
<td># transitions: 48 # states: 4</td>
</tr>
<tr>
<td>3</td>
<td># transitions: 17 # states: 4</td>
<td># transitions: 40 # states: 8</td>
<td># transitions: 45 # states: 4</td>
<td># transitions: 96 # states: 8</td>
</tr>
<tr>
<td>4</td>
<td># transitions: 19 # states: 5</td>
<td># transitions: 80 # states: 16</td>
<td># transitions: 54 # states: 5</td>
<td># transitions: 192 # states: 16</td>
</tr>
<tr>
<td>5</td>
<td># transitions: 29 # states: 7</td>
<td># transitions: 160 # states: 32</td>
<td># transitions: 78 # states: 7</td>
<td># transitions: 384 # states: 32</td>
</tr>
</tbody>
</table>
### Table 8-1: Comparison of Response Times for Different Scenarios

<table>
<thead>
<tr>
<th>No. of constraints</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td># transitions: 34&lt;br&gt;# states: 9</td>
<td># transitions: 97&lt;br&gt;# states: 9</td>
<td># transitions: 768&lt;br&gt;# states: 64</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td># transitions: 54&lt;br&gt;# states: 13</td>
<td># transitions: 145&lt;br&gt;# states: 13</td>
<td># transitions: 1,536&lt;br&gt;# states: 128</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td># transitions: 42&lt;br&gt;# states: 13</td>
<td># transitions: 133&lt;br&gt;# states: 13</td>
<td># transitions: 3,072&lt;br&gt;# states: 256</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td># transitions: 17&lt;br&gt;# states: 8</td>
<td># transitions: 73&lt;br&gt;# states: 8</td>
<td># transitions: 6,144&lt;br&gt;# states: 512</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td># transitions: 8&lt;br&gt;# states: 6</td>
<td># transitions: 50&lt;br&gt;# states: 6</td>
<td># transitions: 12,288&lt;br&gt;# states: 1024</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td># transitions: 6&lt;br&gt;# states: 6</td>
<td># transitions: 48&lt;br&gt;# states: 6</td>
<td># transitions: 24,576&lt;br&gt;# states: 2048</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td># transitions: 5&lt;br&gt;# states: 6</td>
<td># transitions: 47&lt;br&gt;# states: 6</td>
<td># transitions: 49,152&lt;br&gt;# states: 4096</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 8-2: Comparing response times between Scenario 1 and Scenario 3

In the 12-event set scenarios, comparing Scenario 3 (composing related constraints) and Scenario 4 (composing unrelated constraints) results, we can see a big jump in response times. The graphs of the response times are shown in Figure 8-3. During composition of unrelated constraints (Scenario 4), all transitions in an FSA are accepted by the next constraint; therefore, we will have a state space explosion. Log scale are used to show the exponential growth of response times. By composing related constraints (Scenario 3), adding more constraints will restrict the
allowable transitions. Hence the number of common transitions needed to iterate through during composition will be significantly less than the case of composing unrelated constraints.

In the results shown in Table 8-1, the response time quickly exceeds the acceptable limit of under an hour response time when composing more than 11 unrelated constraints in a 12 event-set. To compose 12 unrelated constraints in a 12-event set, it took almost an hour and a half. In our approach, an event-set does not contain all possible events but it contains only events that we are interested in and specify their relative order in constraints. In addition, in a behavioural protocol, we define a sequence between constraints; hence constraints are specified in relation to each other. Composing only unrelated constraints is not likely.

![Figure 8-3: Comparing response times from Scenario 3 and Scenario 4 (graph in log scale)](image)

8.3.2 Conversion from OWL-S to FSA Overhead

Before the compatibility checking between a Web service and a role’s behavioural protocol can be carried out, the Web service’s process description in an OWL-S specification is converted to FSA. In this section, we will analyse the performance overhead of this operation. To characterise the performance overhead of converting an OWL-S description to FSA, we consider two distinct cases: (i) OWL-S
descriptions that have control constructs other than If-Then-Else, and (ii) OWL-S descriptions having If-Then-Else constructs.

Firstly, we analyse the performance overhead of converting OWL-S descriptions having control constructs other than If-Then-Else. By examining the algorithm to convert OWL-S to FSA in Listing 7-1, the time taken to compose FSA is linearly proportional to the total number of atomic processes that a service’s OWL-S specification has (including the atomic processes which are constituents of the composite processes in the specification). The performance testing is carried out by gradually increasing the total number of atomic processes, from 1 to 12 processes, in a Sequence construct in an OWL-S description and measuring the time taken to convert these process models to FSAs. The OWL-S descriptions are edited by using Protégé OWL-S Editor (Elenius et al. 2005). Figure 8-4 shows the structure of the process model described in the OWL-S file used in the test (which is included in Appendix B); it has a Sequence construct containing 12 atomic processes and does not have any If-Then-Else constructs. The process model shown in Figure 8-4 is used for testing purpose only. The conversion from an OWL-S description to FSA has two major steps: (i) parsing XML structure of the OWL-S file by using Mindswap’s OWL-S parser (Sirin 2005), and (ii) converting the parsed process definitions to FSA. Table 8-3 shows the response times of parsing and converting OWL-S specifications having from 1 to 12 processes.

![Figure 8-4: The structure of an example OWL-S composite process](image-url)
Table 8-3: Response times of converting OWL-S specifications not having If-Then-Else construct to FSAs

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>Parsing OWL-S file response time (ms)</th>
<th>Lines of code in the OWL-S files used in testing</th>
<th>Converting to FSA response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1068.5</td>
<td>71</td>
<td>7.5</td>
</tr>
<tr>
<td>2</td>
<td>1061.8</td>
<td>80</td>
<td>9.0</td>
</tr>
<tr>
<td>3</td>
<td>1048.0</td>
<td>90</td>
<td>10.2</td>
</tr>
<tr>
<td>4</td>
<td>1071.5</td>
<td>100</td>
<td>10.7</td>
</tr>
<tr>
<td>5</td>
<td>1063.8</td>
<td>110</td>
<td>12.5</td>
</tr>
<tr>
<td>6</td>
<td>1047.5</td>
<td>120</td>
<td>15.0</td>
</tr>
<tr>
<td>7</td>
<td>1064.3</td>
<td>130</td>
<td>15.7</td>
</tr>
<tr>
<td>8</td>
<td>1065.7</td>
<td>140</td>
<td>17.7</td>
</tr>
<tr>
<td>9</td>
<td>1060.8</td>
<td>150</td>
<td>19.2</td>
</tr>
<tr>
<td>10</td>
<td>1063.8</td>
<td>160</td>
<td>21.0</td>
</tr>
<tr>
<td>11</td>
<td>1050.5</td>
<td>169</td>
<td>22.3</td>
</tr>
<tr>
<td>12</td>
<td>1076.3</td>
<td>179</td>
<td>23.8</td>
</tr>
<tr>
<td>Standard deviation</td>
<td></td>
<td></td>
<td>9.06</td>
</tr>
</tbody>
</table>

From the results in Table 8-3, the response time of the OWL-S file parsing is fluctuating and does not change a lot; the recorded results have a standard deviation of 9.06 ms for 12 samples. The response time of converting the parsed processes to FSA has a linear growth relative to the number of processes as shown in the graph in Figure 8-5.

![Figure 8-5: Graph of response times of converting process definitions to FSAs for the case that the OWL-S descriptions do not have If-Then-Else constructs](image)
The next step is to analyse the performance overhead of converting OWL-S descriptions that have If-Then-Else constructs. As discussed in Section 5.2, converting an OWL-S specification having \( n \) number of If-Then-Else constructs will result in \( 2^n \) FSAs in the worst case scenario when all the If-Then-Else constructs have both the If-Then and the Else branches and those constructs are placed one after the other rather than in nested If-Then-Else constructs. To characterise this overhead, we consider different OWL-S descriptions having from 1 to 12 If-Then-Else constructs and measure the time taken to convert these OWL-S descriptions to FSAs. Figure 8-6 shows the structure of the process model used in the test, the OWL-S file of this process model is included in Appendix C. Table 8-4 shows the response times of parsing and converting OWL-S specifications having from 1 to 12 If-Then-Else constructs to FSAs.

<table>
<thead>
<tr>
<th>No. of If-Then-Else constructs</th>
<th>Parsing OWL-S file response time (ms)</th>
<th>Lines of code in the OWL-S files used in testing</th>
<th>Converting to FSA response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1015.2</td>
<td>78</td>
<td>7.5</td>
</tr>
<tr>
<td>2</td>
<td>1017.7</td>
<td>98</td>
<td>11.5</td>
</tr>
<tr>
<td>3</td>
<td>1042.0</td>
<td>117</td>
<td>19.7</td>
</tr>
<tr>
<td>4</td>
<td>1037.7</td>
<td>138</td>
<td>29.2</td>
</tr>
<tr>
<td>5</td>
<td>1035.5</td>
<td>158</td>
<td>50.8</td>
</tr>
<tr>
<td>6</td>
<td>1044.0</td>
<td>179</td>
<td>97.8</td>
</tr>
<tr>
<td>7</td>
<td>1065.3</td>
<td>202</td>
<td>187.8</td>
</tr>
<tr>
<td>8</td>
<td>1020.8</td>
<td>224</td>
<td>429.5</td>
</tr>
<tr>
<td>9</td>
<td>1029.7</td>
<td>245</td>
<td>999.2</td>
</tr>
<tr>
<td>10</td>
<td>1042.5</td>
<td>265</td>
<td>2347.2</td>
</tr>
<tr>
<td>11</td>
<td>1033.2</td>
<td>285</td>
<td>5760.8</td>
</tr>
<tr>
<td>12</td>
<td>1074.3</td>
<td>306</td>
<td>12662.2</td>
</tr>
</tbody>
</table>

Table 8-4: Response times of converting OWL-S specifications having If-Then-Else construct to FSAs
Figure 8-6: The structure of an OWL-S description having 12 If-Then-Else constructs in sequence

Similar to the results in Table 8-3, the results of parsing OWL-S files shown in Table 8-4 also fluctuate and do not change significantly. However, the response times of converting the OWL-S processes to FSAs have an exponential growth against the number of If-Then-Else constructs in an OWL-S description, as shown in Figure 8-7.
Once a Web service’s OWL-S description is converted to FSA, the resulting FSA will be checked against the FSA of a role’s behavioural protocol in order to determine the compatibility of the Web service against the role. The checking is carried out by using the FSA intersection. The analysis of the overhead of the FSA intersection process was already discussed previously in Section 8.3.1.

8.3.3 Conclusion

Previous sections have characterised the run-time overheads of the FSA intersection and converting OWL-S to FSA. The FSA intersection has a limitation regarding the state space explosion problem when a large number of unrelated constraints are composed and when an event set having a large number of events is used. However, given that constraints are used to add restriction to the sequence of interactions, constraints are often related to each other. In addition, an event set only contains events of interest; hence, the size of an event set is not very large. As shown in Table 8-1, composing 12 related constraints using a 12-event set took under 300 ms to complete.

During conversion of OWL-S descriptions to FSA, the parsing times of OWL-S files do not change significantly. For OWL-S descriptions that do not have If-Then-Else constructs, converting a parsed OWL-S description to FSA has a linear growth
against the number of atomic processes defined in an OWL-S description. In our experiment of converting an OWL-S file having 12 atomic processes, the OWL-S to FSA conversion process was completed in just over a second (on average, OWL-S file parsing took 1076.3 ms and FSA conversion took 23.8 ms as in Table 8-3). For OWL-S descriptions having If-Then-Else constructs, the conversion of a parsed OWL-S description to FSA has an exponential growth against the number of If-Then-Else constructs in the worst case scenarios. Converting an OWL-S description having 12 If-Then-Else constructs took less than 14 seconds to complete (on average, OWL-S file parsing took 1074.3 ms and FSA conversion took 12,662 ms as in Table 8-4). 

From the experiments shown above, we can conclude that our proposed behavioural protocol support mechanism does not impose a significant overhead.

### 8.4 Summary

In this chapter, we discussed the advantages and limitations of the proposed behavioural protocol framework. The behavioural protocol specification used in our framework, Interaction Rule Specification (IRS), is a declarative pattern based approach which supports incremental specification of behavioural protocols. IRS is limited in specifying other constraints such as real-time constraints. Our proposed behavioural protocol match-making and classification provides more information to be used during adaptation rather than just pass/fail criteria. The functional match-making is quite rudimentary as it is not the focus of this thesis. In future work, we plan to implement the functional match-making by using techniques such as ontology subsumption. Our proposed adaptors are structural compositions which are more than a usual adaptor in that they have a flexible and adaptive architecture which can be changed at run-time. However, human interaction is required to define adaptors.

The run-time overheads of the FSA intersection and converting OWL-S to FSA were also examined by recording response times from executing different scenarios. From the results gathered, it is concluded that the overhead of our behavioural protocol support mechanism is insignificant.
Chapter 9:
Conclusions and Future Work

In this thesis, we have presented our research in providing a behavioural protocol support framework in order to specify the required behavioural protocols of interacting parties, perform compatibility checking between required protocols and services’ provided protocols, and adapt any possible mismatches between protocols. In Section 9.1 below, we summarise our contributions, followed by a discussion of future work in Section 9.2.

9.1 Contributions

Service Oriented Architecture (SOA) and Web services have created open and standard-based approaches for connecting services together. These standards (including WSDL, SOAP, etc.) enable communication between services implemented on heterogeneous platforms at the message level. However, for them to be successfully composed, they have to be compatible in both the types of messages and the sequence of interactions. WSDL-based service descriptions do not support the description of the interaction sequences of a service, also known as its behavioural protocol. As a result, various approaches have been proposed to describe behavioural aspects of Web services, including OWL-S (Martin et al. 2004a) and WSMO (Bruijn et al. 2005). These approaches do not provide formalisms whereby behaviour can be easily reasoned about and composed. This means the current techniques for behavioural service composition require developers to evaluate candidate services and compose them manually (Martin et al. 2007). To address these issues, we have proposed a semi-automated service composition framework with support for behavioural protocols. The research presented in this thesis has three main contributions:
Applying a behavioural protocol specification approach (Interaction Rule Specification), which has Finite State Automaton (FSA) as its underlying formalism, into an adaptive service composition framework (the Role Oriented Adaptive Design framework) and implementing various mechanisms to support behavioural protocols in the framework including: protocol aggregation, consistency checking and run-time protocol monitoring.

Developing a mismatch classification based on both functional and behavioural protocols.

Proposing adaptation patterns to resolve mismatches between the required protocols and the provided ones.

By using the Interaction Rule Specification (IRS) (Jin and Han 2005a) in the way presented in this thesis, developers can define behavioural specifications in a declarative manner. Each constraint is an encapsulated unit which is meaningful and does not depend on other constraints. Constraints are added incrementally and the order of constraints specified does not affect their combined meaning. IRS has its underlying semantics defined in FSA so that it can be manipulated, composed and reasoned about. FSA was chosen since it is effective in modelling sequential systems and verification of protocol compatibilities (Berardi et al. 2004; Canal 2004), and it is simple and widely known by software practitioners. An important contribution of our work is the development of mechanisms to manipulate behavioural protocols: aggregation, consistency checking and run-time protocol monitoring. Once the required behavioural constraints are defined, they are aggregated to produce different views of protocols which form the basis to match potential Web services. Because behavioural constraints are added incrementally, the computed aggregation of protocols can have inconsistencies. The consistency checking mechanism performs verification to ensure that constraints are consistent and do not violate each other. Finally, the run-time protocol monitoring intercepts messages exchanged between interacting parties and checks that the interactions actually follow the specified behavioural protocols.

Having required behavioural constraints defined, potential Web services exposing behavioural protocol specifications (for example in OWL-S) can be matched against the required behavioural constraints. Although functional specification and functional
Chapter 9  Conclusions and Future Work

mismatch are out of scope for this thesis, we believe that behavioural aspects can only be addressed adequately in conjunction with functional aspects. Although there is existing work for match-making and classifications of functional capability such as Paolucci et al. (2002) and Sycara et al. (2003), one of the contributions of our work is the match-making and mismatch classifications taking both functional and behavioural aspects into consideration. Functional matching is done by comparing all the required operations to the provided operations of a candidate service, and results in three categories: sufficient match, partial match and no match. As part of the behavioural match making process, behavioural protocol specifications in different notations (such as IRS and OWL-S) are converted to the same formalism, FSA. The behavioural match-making is then carried out by computing FSA intersection of the converted behavioural protocols’ FSAs. The match-making result is classified into three categories: compatible, partially compatible and incompatible.

In situations where mismatches happen and can be adapted, adaptors can be used to (re-)compose existing services to meet the requirements (in contrast to the more costly or even prohibitive option of developing new services). This thesis proposes three adaptation patterns: conversation coordinator, functional role composition and trusted mediator. In the literature, there are some efforts in addressing service/component mismatches and adaptation such as Yellin and Strom (1997), Wombacher et al. (2004a), Benatallah et al. (2005), Brogi and Popescu (2006) and Mokhtar et al. (2006). However, they mostly address specific types of mismatches in specific contexts, focusing on simple message ordering and operation selection. In general, service adaptation needs to be flexible and systematic in order to deal with a wide range of mismatches. A contribution of our work is the development of adaptation patterns which produces adaptors as architectural entities that can be dynamically introduced and changed at run-time. This allows us to address a greater range of mismatches in a systematic manner.

It is important that the Web services involved in a service composition interact with each other in a compatible manner both in terms of functionality and behaviour. Generally, Web services were developed without service consumers in mind; it is difficult to find Web services that are a perfect match to each other in terms of both required/provided functionality, and in terms of compatible sequences of interactions.
Our research efforts have contributed to the overall task of automated service composition where users can specify the functional and behavioural requirements, a composition framework then looks for the matched or partially matched services and composes them to achieve the predefined goal. We also presented a way to resolve the mismatches in the protocols between what is expected and what is actually provided, and provided service oriented applications the ability to adapt to changing environments. Our work represents a further step towards automated service composition.

9.2 Future Work

In some business transactions, constraints related to real-time are required to be specified. An example of a real-time constraint is “the payment has to be made within 10 days of delivery”. Although IRS is expressive in specifying ordering and occurrence of business events, IRS lacks the capability to specifying real-time constraints. Real-time constraints are an important aspect that needs to be added to IRS so that it can be more expressive. In addition, the process of specifying IRS behavioural constraints is done manually by developers. This process could be better supported by a tool that provides features such as ‘drag and drop’ of constraint patterns and operation names, and displaying graphical previews of the specified sequence of interactions.

During conversions of OWL-S descriptions to FSAs, conditions in Repeat-While and Repeat-Until constructs cannot be evaluated at design time; hence these constructs are treated as unbounded occurrences and OWL-S specifications having these constructs are considered not matched against roles’ centric protocols. For conditions in If-Then-Else constructs, both the If-Then-branch and the Else-branch are considered separate FSAs and both of them have to match to a role’s centric protocol during match-making in order for the corresponding OWL-S specification to be considered matched. This approach to deal with conditions in OWL-S specifications for Repeat-While and Repeat-Until constructs is quite restricted as certain potential matches may be regarded as “not matched”; and for If-Then-Else constructs, it can result in an explosion of FSAs at the rate of $2^n$ in the worst case.
scenario where \( n \) is the number of If-Then-Else constructs. Addressing this limitation is one of the topics of future work.

The FSA intersection algorithm is used to compute behavioural constraint compositions and compatibility checking. Under extreme cases, the FSA intersection can result in a state space explosion, which impacts execution times. As part of future work, we plan to optimise our implementation and investigate methods to reduce the state space in order to improve the execution time of the behavioural constraint evaluation process.

The proposed adaptation patterns provide a flexible and systematic framework to address mismatches. Nevertheless, the adaptation process is not fully automated. Developers are included “in the loop” to define message flows within an adaptor composite. To increase the level of automation during the adaptation process, intelligent agent or automatic adaptor synthesis can be considered.

This work has only concentrated on behavioural aspects. To complete the picture, a thorough functional match-making will need to be investigated and implemented. In order to consider a wide spread of meanings in inputs/outputs and the semantics of functionality of services, ontology and cross-ontology mappings can be used. Then functional match-matching techniques based on ontology evaluation, such as techniques presented by Paolucci et al. (2002) and Sycara et al. (2003), can be integrated to our work to address functional compatibility matching in a more formal way.
References


Colman, A. (2006) *Role Oriented Adaptive Design (ROAD)*. PhD thesis. Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia


Appendix A: An Example of Aggregation of Multiple Contracts

```plaintext
protocol dependencies
{
  Buyer-Broker.Broker.quote precedes Broker-Vendor1.Vendor1.quote globally
  where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor1.Vendor1.quote::orderNo;
  Buyer-Broker.Broker.quote leads to Broker-Vendor1.Vendor1.quote globally
  where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor1.Vendor1.quote::orderNo;

  Buyer-Broker.Broker.quote precedes Broker-Vendor2.Vendor2.quote globally
  where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor2.Vendor2.quote::orderNo;
  Buyer-Broker.Broker.quote leads to Broker-Vendor2.Vendor2.quote globally
  where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor2.Vendor2.quote::orderNo;

  Buyer-Broker.Broker.order precedes Broker-Vendor1.Vendor1.order globally
  where Buyer-Broker.Broker.order::orderNo = Broker-Vendor1.Vendor1.order::orderNo;
  Buyer-Broker.Broker.order leads to Broker-Vendor1.Vendor1.order globally
  where Buyer-Broker.Broker.order::orderNo = Broker-Vendor1.Vendor1.order::orderNo;
  (Buyer-Broker.Broker.order leads to Broker-Vendor1.Vendor1.order globally where Buyer-Broker.Broker.order::orderNo = Broker-Vendor1.Vendor1.order::orderNo)  OR (Buyer-Broker.Broker.order leads to Broker-Vendor2.Vendor2.order globally where Buyer-Broker.Broker.order::orderNo = Broker-Vendor2.Vendor2.order::orderNo);

  Buyer-Broker.Broker.payment precedes Broker-Vendor1.Vendor1.payment globally
  where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo;
  Buyer-Broker.Broker.payment leads to Broker-Vendor1.Vendor1.payment globally
  where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo;
  (Buyer-Broker.Broker.payment leads to Broker-Vendor1.Vendor1.payment globally where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo)  OR (Buyer-Broker.Broker.payment leads to Broker-Vendor2.Vendor2.payment globally where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor2.Vendor2.payment::orderNo);

  Buyer-Broker.Broker.deliveryDetail precedes Broker-Vendor1.Vendor1.deliveryDetail globally
  where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor1.Vendor1.deliveryDetail::orderNo;
  Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor1.Vendor1.deliveryDetail globally
  where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor1.Vendor1.deliveryDetail::orderNo;
  (Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor1.Vendor1.deliveryDetail globally where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor1.Vendor1.deliveryDetail::orderNo)
```

(Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor2.Vendor2.deliveryDetail globally where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor2.Vendor2.deliveryDetail::orderNo); 38

Buyer-Broker.Broker.delivery precedes Broker-Vendor1.Vendor1.delivery globally where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor1.Vendor1.delivery::orderNo; 40
Buyer-Broker.Broker.delivery precedes Broker-Vendor2.Vendor2.delivery globally where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor2.Vendor2.delivery::orderNo; 42
(Buyer-Broker.Broker.delivery leads to Broker-Vendor1.Vendor1.delivery globally where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor1.Vendor1.delivery::orderNo) 44
OR (Buyer-Broker.Broker.delivery leads to Broker-Vendor2.Vendor2.delivery globally where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor2.Vendor2.delivery::orderNo); 46

Buyer-Broker protocol
50{
52
Broker.quote precedes Broker.order globally where Broker.quote::orderNo = Broker.order::orderNo; 54

Broker.order precedes Broker.payment globally where Broker.order::orderNo = Broker.payment::orderNo; 56
Broker.order leads to Broker.payment globally where Broker.order::orderNo = Broker.payment::orderNo; 58

Broker.payment precedes Broker.deliveryDetail globally where Broker.payment::orderNo = Broker.deliveryDetail::orderNo; 60
Broker.payment leads to Broker.deliveryDetail globally where Broker.payment::orderNo = Broker.deliveryDetail::orderNo; 62

Broker.deliveryDetail precedes Broker.delivery globally where Broker.deliveryDetail::orderNo = Broker.delivery::orderNo; 64
Broker.deliveryDetail leads to Broker.delivery globally where Broker.deliveryDetail::orderNo = Broker.delivery::orderNo; 66

Broker.quote exists at most once globally; 68
Broker.order exists at most once globally; 70
Broker.payment exists at most once globally; 72
Broker.deliveryDetail exists at most once globally; 74
Broker.delivery exists at most once globally; 76
}
Broker-Vendor1 protocol
{
  Vendor1.quote precedes Vendor1.order globally
  where Vendor1.quote::orderNo = Vendor1.order::orderNo;
  Vendor1.order precedes Vendor1.payment globally
  where Vendor1.order::orderNo = Vendor1.payment::orderNo;
  Vendor1.order leads to Vendor1.payment globally
  where Vendor1.order::orderNo = Vendor1.payment::orderNo;
  Vendor1.payment precedes Vendor1.deliveryDetail globally
  where Vendor1.payment::orderNo = Vendor1.deliveryDetail::orderNo;
  Vendor1.payment leads to Vendor1.deliveryDetail globally
  where Vendor1.payment::orderNo = Vendor1.deliveryDetail::orderNo;
  Vendor1.deliveryDetail precedes Vendor1.delivery globally
  where Vendor1.deliveryDetail::orderNo = Vendor1.delivery::orderNo;
  Vendor1.deliveryDetail leads to Vendor1.delivery globally
  where Vendor1.deliveryDetail::orderNo = Vendor1.delivery::orderNo;
  Vendor1.quote exists at most once globally;
  Vendor1.order exists at most once globally;
  Vendor1.payment exists at most once globally;
  Vendor1.deliveryDetail exists at most once globally;
  Vendor1.delivery exists at most once globally;
}

Broker-Vendor2 protocol
{
  Vendor2.quote precedes Vendor2.order globally
  where Vendor2.quote::orderNo = Vendor2.order::orderNo;
  Vendor2.order precedes Vendor2.payment globally
  where Vendor2.order::orderNo = Vendor2.payment::orderNo;
  Vendor2.order leads to Vendor2.payment globally
  where Vendor2.order::orderNo = Vendor2.payment::orderNo;
  Vendor2.payment precedes Vendor2.deliveryDetail globally
  where Vendor2.payment::orderNo = Vendor2.deliveryDetail::orderNo;
  Vendor2.payment leads to Vendor2.deliveryDetail globally
}
where Vendor2.payment::orderNo = Vendor2.deliveryDetail::orderNo;

Vendor2.deliveryDetail precedes Vendor2.delivery globally
where Vendor2.deliveryDetail::orderNo = Vendor2.delivery::orderNo;
Vendor2.deliveryDetail leads to Vendor2.delivery globally
where Vendor2.deliveryDetail::orderNo = Vendor2.delivery::orderNo;

Vendor2.quote exists at most once globally;
Vendor2.order exists at most once globally;
Vendor2.payment exists at most once globally;
Vendor2.deliveryDetail exists at most once globally;
Vendor2.delivery exists at most once globally;
}

Broker role-centric protocol
{

//projection of Buyer-Broker contract protocol on Broker
Buyer-Broker.Broker.quote precedes Buyer-Broker.Broker.order globally
where Buyer-Broker.Broker.quote::orderNo = Buyer-Broker.Broker.order::orderNo;
Buyer-Broker.Broker.order precedes Buyer-Broker.Broker.payment globally
where Buyer-Broker.Broker.order::orderNo = Buyer-Broker.Broker.payment::orderNo;
Buyer-Broker.Broker.order leads to Buyer-Broker.Broker.payment globally
where Buyer-Broker.Broker.order::orderNo = Buyer-Broker.Broker.payment::orderNo;
Buyer-Broker.Broker.payment precedes Buyer-Broker.Broker.deliveryDetail globally
where Buyer-Broker.Broker.payment::orderNo = Buyer-Broker.Broker.deliveryDetail::orderNo;
Buyer-Broker.Broker.payment leads to Buyer-Broker.Broker.deliveryDetail globally
where Buyer-Broker.Broker.payment::orderNo = Buyer-Broker.Broker.deliveryDetail::orderNo;
Buyer-Broker.Broker.deliveryDetail precedes Buyer-Broker.Broker.delivery globally
where Buyer-Broker.Broker.deliveryDetail::orderNo = Buyer-Broker.Broker.delivery::orderNo;
Buyer-Broker.Broker.deliveryDetail leads to Buyer-Broker.Broker.delivery globally
where Buyer-Broker.Broker.deliveryDetail::orderNo = Buyer-Broker.Broker.delivery::orderNo;

Buyer-Broker.Broker.quote exists at most once globally;
Buyer-Broker.Broker.order exists at most once globally;
Buyer-Broker.Broker.payment exists at most once globally;
Buyer-Broker.Broker.deliveryDetail exists at most once globally;
Buyer-Broker.Broker.delivery exists at most once globally;

} AND

//projection of Broker-Vendor1 contract protocol on Broker
//empty

} AND

//projection of Broker-Vendor2 contract protocol on Broker
//empty

) AND

//projection of protocol dependencies on Broker

Buyer-Broker.Broker.quote precedes Broker-Vendor1.Vendor1.quote globally
where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor1.Vendor1.quote::orderNo;
Buyer-Broker.Broker.quote leads to Broker-Vendor1.Vendor1.quote globally
where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor1.Vendor1.quote::orderNo;
Buyer-Broker.Broker.quote precedes Broker-Vendor2.Vendor2.quote globally
where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor2.Vendor2.quote::orderNo;
Buyer-Broker.Broker.quote leads to Broker-Vendor2.Vendor2.quote globally
where Buyer-Broker.Broker.quote::orderNo = Broker-Vendor2.Vendor2.quote::orderNo;

Buyer-Broker.Broker.order precedes Broker-Vendor1.Vendor1.order globally
where Buyer-Broker.Broker.order::orderNo = Broker-Vendor1.Vendor1.order::orderNo;
Buyer-Broker.Broker.order leads to Broker-Vendor1.Vendor1.order globally where Buyer-Broker.Broker.order::orderNo = Broker-Vendor1.Vendor1.order::orderNo;
Buyer-Broker.Broker.order precedes Broker-Vendor2.Vendor2.order globally
where Buyer-Broker.Broker.order::orderNo = Broker-Vendor2.Vendor2.order::orderNo;

Buyer-Broker.Broker.payment precedes Broker-Vendor1.Vendor1.payment globally
where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo;
Buyer-Broker.Broker.payment leads to Broker-Vendor1.Vendor1.payment globally where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo;
Buyer-Broker.Broker.payment precedes Broker-Vendor2.Vendor2.payment globally

Buyer-Broker.Broker.order leads to Broker-Vendor2.Vendor2.order globally where Buyer-Broker.Broker.order::orderNo = Broker-Vendor2.Vendor2.order::orderNo;
where Buyer-Broker.Broker.payment::orderNo = Broker-Vendor2.Vendor2.payment::orderNo;
(Buyer-Broker.Broker.payment leads to Broker-Vendor1.Vendor1.payment globally where Buyer-
Broker.Broker.payment::orderNo = Broker-Vendor1.Vendor1.payment::orderNo)
OR (Buyer-Broker.Broker.payment leads to Broker-Vendor2.Vendor2.payment globally where Buyer-
Broker.Broker.payment::orderNo = Broker-Vendor2.Vendor2.payment::orderNo);

Buyer-Broker.Broker.deliveryDetail precedes Broker-Vendor1.Vendor1.deliveryDetail globally
where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor1.Vendor1.deliveryDetail::orderNo;
Buyer-Broker.Broker.deliveryDetail precedes Broker-Vendor2.Vendor2.deliveryDetail globally
where Buyer-Broker.Broker.deliveryDetail::orderNo = Broker-Vendor2.Vendor2.deliveryDetail::orderNo;
(Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor1.Vendor1.deliveryDetail globally where Buyer-
Broker.Broker.deliveryDetail::orderNo = Broker-Vendor1.Vendor1.deliveryDetail::orderNo)
OR (Buyer-Broker.Broker.deliveryDetail leads to Broker-Vendor2.Vendor2.deliveryDetail globally where Buyer-
Broker.Broker.deliveryDetail::orderNo = Broker-Vendor2.Vendor2.deliveryDetail::orderNo);

Buyer-Broker.Broker.delivery precedes Broker-Vendor1.Vendor1.delivery globally
where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor1.Vendor1.delivery::orderNo;
Buyer-Broker.Broker.delivery precedes Broker-Vendor2.Vendor2.delivery globally
where Buyer-Broker.Broker.delivery::orderNo = Broker-Vendor2.Vendor2.delivery::orderNo;
(Buyer-Broker.Broker.delivery leads to Broker-Vendor1.Vendor1.delivery globally where Buyer-
Broker.Broker.delivery::orderNo = Broker-Vendor1.Vendor1.delivery::orderNo)
OR (Buyer-Broker.Broker.delivery leads to Broker-Vendor2.Vendor2.delivery globally where Buyer-
Broker.Broker.delivery::orderNo = Broker-Vendor2.Vendor2.delivery::orderNo);
Appendix B:

OWL-S File Used in Performance Testing for Sequence of Atomic Processes

```xml
<?xml version="1.0"?>
<rdf:RDF
   xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
   xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
   xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
   xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:expr="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:swrl="http://www.w3.org/2003/11/swrl#"
   xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
   xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:time="http://www.isi.edu/~pan/damltime/time-entry.owl#"
   xml:base="http://localhost:8080/owls/services/directory01/BooksToYourDoorService.owl#"`
   <owl:Ontology rdf:about="">
     <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Profile.owl#"/>
     <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"/>
     <owl:imports rdf:resource="http://www.w3.org/2003/11/swrl#"/>
     <owl:imports rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
     <owl:imports rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#"/>
     <owl:imports rdf:resource="http://localhost:8080/owls/services/directory01/BooksToYourDoorService.owl#"/>
   </owl:Ontology>
   <process:Perform rdf:ID="Perform_23">
     <process:process>
       <process:AtomicProcess rdf:ID="paymentInstalment07"/>
     </process:process>
   </process:Perform>
   <process:ControlConstructList rdf:ID="ControlConstructList_10">
     <list:first>
       <process:Perform rdf:ID="Perform_9">
         <process:process>
           <process:AtomicProcess rdf:ID="order"/>
         </process:process>
       </process:Perform>
     </list:first>
   </process:ControlConstructList>
```
```xml
<process:controlConstructList rdf:id="ControlConstructList_12">
  <list:rest>
    <process:controlConstructList rdf:id="ControlConstructList_14">
      <list:rest>
        <process:controlConstructList rdf:id="ControlConstructList_16">
          <list:first>
            <process:perform rdf:id="Perform_15">
              <process:process>
                <process:atomicProcess rdf:id="paymentInstalment03"/>
              </process:process>
            </process:perform>
          </list:first>
          <list:rest>
            <process:controlConstructList rdf:id="ControlConstructList_18">
              <list:first>
                <process:perform rdf:id="Perform_17">
                  <process:process>
                    <process:atomicProcess rdf:id="paymentInstalment04"/>
                  </process:process>
                </process:perform>
              </list:first>
              <list:rest>
                <process:controlConstructList rdf:id="ControlConstructList_20">
                  <list:first>
                    <process:perform rdf:id="Perform_19">
                      <process:process>
                        <process:atomicProcess rdf:id="paymentInstalment05"/>
                      </process:process>
                    </process:perform>
                  </list:first>
                  <list:rest>
                    <process:controlConstructList rdf:id="ControlConstructList_22">
                      <list:first>
                        <process:perform rdf:id="Perform_21">
                          <process:process>
                            <process:atomicProcess rdf:id="paymentInstalment06"/>
                          </process:process>
                        </process:perform>
                      </list:first>
                    </process:controlConstructList>
                  </list:rest>
                </process:controlConstructList>
              </list:rest>
            </process:controlConstructList>
          </list:rest>
        </process:controlConstructList>
      </list:rest>
    </process:controlConstructList>
  </list:rest>
</process:controlConstructList>
```
<process:ControlConstructList rdf:ID="ControlConstructList_24">
  <list:rest>
    <process:ControlConstructList rdf:ID="ControlConstructList_26">
      <list:rest>
        <process:Perform rdf:ID="Perform_25">
          <process:process>
            <process:AtomicProcess rdf:ID="paymentInstalment08"/>
          </process:process>
        </process:Perform>
      </list:rest>
    </process:ControlConstructList>
    <list:rest>
      <process:ControlConstructList rdf:ID="ControlConstructList_28">
        <list:rest>
          <process:Perform rdf:ID="Perform_29">
            <process:process>
              <process:AtomicProcess rdf:ID="delivery"/>
            </process:process>
          </process:Perform>
        </list:rest>
      </process:ControlConstructList>
      <list:rest rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#nil"/>
      <process:Perform rdf:ID="Perform_27">
        <process:process>
          <process:AtomicProcess rdf:ID="deliveryDetail"/>
        </process:process>
      </process:Perform>
    </list:rest>
  </list:rest>
</process:ControlConstructList>
<list:first rdf:resource="#Perform_23"/>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
<list:first>
<process:Perform rdf:ID="Perform_13">
<process:process>
<process:AtomicProcess rdf:ID="paymentInstalment02"/>
</process:process>
</process:Perform>
</list:first>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
<list:first>
<process:Perform rdf:ID="Perform_11">
<process:process>
<process:AtomicProcess rdf:ID="paymentInstalment01"/>
</process:process>
</process:Perform>
</list:first>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
<process:AtomicProcess rdf:ID="quote"/>
<expr:Condition rdf:ID="Condition_3"/>
<expr:Condition rdf:ID="Condition_10"/>
<process:AtomicProcess rdf:ID="rejectQuote"/>
<process:Sequence rdf:ID="Sequence_1">
<process:components>
<process:ControlConstructList rdf:ID="ControlConstructList_3">
<list:first>
<process:Perform rdf:ID="Perform_Quote">
<process:process rdf:resource="#quote"/>
<process:Perform>
</list:first>
</process:ControlConstructList>
</process:components>
</process:Sequence>
</profile:Profile rdf:ID="BooksToYourDoorProfile">
<service:presentedBy>
<service:Service rdf:ID="BooksToYourDoorService">
<service:presents rdf:resource="#BooksToYourDoorProfile"/>
<service:describedBy>
<process:CompositeProcess rdf:ID="booksOrderProcess">
<process:composedOf rdf:resource="#Sequence_1"/>
<service:describes rdf:resource="#BooksToYourDoorService"/>
</process:CompositeProcess>
</service:describedBy>
</service:Service>
</service:presentedBy>
<profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
A book service and delivery</profile:textDescription>
<profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
BooksToYourDoorService</profile:serviceName>
<profile:has_process rdf:resource="#booksOrderProcess"/>
</profile:Profile>
<process:AtomicProcess rdf:ID="payment"/>
<expr:Condition rdf:ID="Condition_6"/>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.2.1, Build 365)  http://protege.stanford.edu -->
Appendix C:

OWL-S File Used in Performance Testing Sequence of If-Then-Else constructs

<?xml version="1.0"?>
<rdf:RDF
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:expr="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:time="http://www.isi.edu/~pan/damltime/time-entry.owl#"
  xmlns:base="http://localhost:8080/owls/services/directory01/BooksToYourDoorService.owl#"
  xml:base="http://localhost:8080/owls/services/directory01/BooksToYourDoorService.owl#"
>
<owl:Ontology rdf:about=""
  owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Profile.owl#"/>
<owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"/>
<owl:imports rdf:resource="http://www.w3.org/2003/11/swrl#"/>
<owl:imports rdf:resource="http://www.w3.org/2003/11/swrlb#"/>
<owl:Ontology rdf:ID="Perform_36">
  <process:process rdf:ID="rejectQuote"/>
</owl:Ontology>
<process:Perform rdf:ID="Perform_36">
  <process:process rdf:ID="quote"/>
<process:ControlConstructList rdf:ID="ControlConstructList_10">
  <list:rest>
    <process:ControlConstructList rdf:ID="ControlConstructList_15">
      <list:rest>
        <process:ControlConstructList rdf:ID="ControlConstructList_20">
          <list:rest>
            <process:ControlConstructList rdf:ID="ControlConstructList_25">
              <list:first>
                <process:If-Then-Else rdf:ID="If-Then-Else_29">
                  <process:ifCondition>
                    <expr:Condition rdf:ID="Condition_31"/>
                  </process:ifCondition>
                  <process:then>
                    <process:Perform rdf:ID="Perform_32">
                      <process:process>
                        <process:AtomicProcess rdf:ID="order"/>
                      </process:process>
                    </process:Perform>
                  </process:then>
                  <process:else>
                    <process:Perform rdf:ID="Perform_35">
                      <process:process rdf:resource="#rejectQuote"/>
                    </process:Perform>
                  </process:else>
                </process:If-Then-Else>
              </list:first>
              <list:rest>
                <process:ControlConstructList rdf:ID="ControlConstructList_2">
                  <list:first>
                    <process:If-Then-Else rdf:ID="If-Then-Else_1">
                      <process:else>
                        <process:Perform rdf:ID="Perform_9">
                          <process:process rdf:resource="#rejectQuote"/>
                        </process:Perform>
                      </process:else>
                      <process:then>
                        <process:Perform rdf:ID="Perform_6">
                          <process:process rdf:resource="#order"/>
                        </process:Perform>
                      </process:then>
                    </process:If-Then-Else>
                  </list:first>
                </process:ControlConstructList>
              </list:rest>
            </process:ControlConstructList>
          </list:rest>
        </process:ControlConstructList>
      </list:rest>
    </process:ControlConstructList>
  </list:rest>
</process:ControlConstructList>
</process:Perform>
</process:then>
<process:ifCondition>
<expr:Condition rdf:ID="Condition_5"/>
</process:ifCondition>
</process:If-Then-Else>
</list:first>
</list:rest>
<process:ControlConstructList rdf:ID="ControlConstructList_17">
<list:first>
<process:If-Then-Else rdf:ID="If-Then-Else_16">
<process:else>
<process:Perform rdf:ID="Perform_20">
<process:process rdf:resource="#rejectQuote"/>
</process:Perform>
</process:else>
<process:then>
<process:Perform rdf:ID="Perform_19">
<process:process rdf:resource="#order"/>
</process:Perform>
</process:then>
</process:If-Then-Else>
</list:first>
<list:rest>
<process:ControlConstructList rdf:ID="ControlConstructList_22">
<list:rest>
<process:ControlConstructList rdf:ID="ControlConstructList_27">
<list:rest>
<process:ControlConstructList rdf:ID="ControlConstructList_32">
<list:rest>
<process:ControlConstructList rdf:ID="ControlConstructList_38">
<list:first>
<process:If-Then-Else rdf:ID="If-Then-Else_37">
<process:then>
<process:Perform rdf:ID="Perform_40">
<process:process rdf:resource="#order"/>
</process:Perform>
<process:then>
  <process:ifCondition>
    <expr:Condition rdf:ID="Condition_39"/>
  </process:ifCondition>
  <process:else>
    <process:Perform rdf:ID="Perform_41">
      <process:process rdf:resource="#rejectQuote"/>
    </process:Perform>
  </process:else>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
<list:first>
  <process:If-Then-Else rdf:ID="If-Then-Else_31">
    <process:ifCondition>
      <expr:Condition rdf:ID="Condition_33"/>
    </process:ifCondition>
    <process:else rdf:resource="#Perform_36"/>
  </process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
<list:first>
  <process:If-Then-Else rdf:ID="If-Then-Else_26">
    <process:ifCondition>
      <expr:Condition rdf:ID="Condition_28"/>
    </process:ifCondition>
    <process:else>
      <process:Perform rdf:ID="Perform_30">
        <process:process rdf:resource="#rejectQuote"/>
      </process:Perform>
    </process:else>
  </process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
<process:then>
  <process:Perform rdf:ID="Perform_29">
    <process:process rdf:resource="#order"/>
  </process:Perform>
</process:then>
<process:then>
  <process:else>
    <process:Perform rdf:ID="Perform_28">
      <process:process rdf:resource="#rejectQuote"/>
    </process:Perform>
  </process:else>
</process:else>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
</list:first>
<process:If-Then-Else rdf:ID="If-Then-Else_19">
  <process:then>
    <process:Perform rdf:ID="Perform_22">
      <process:process rdf:resource="#order"/>
    </process:Perform>
  </process:then>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
</list:first>
<process:If-Then-Else rdf:ID="If-Then-Else_14">
  <process:then>
    <process:Perform rdf:ID="Perform_17">
      <process:process rdf:resource="#order"/>
    </process:Perform>
  </process:then>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
<process:ifCondition rdf:resource="#Condition_16"/>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
</list:rest>
</process:ControlConstructList>
<process:If-Then-Else rdf:ID="If-Then-Else_9">
<process:then>
<process:Perform rdf:ID="Perform_12">
<process:process rdf:resource="#order"/>
</process:Perform>
</process:then>
<process:ifCondition>
<expr:Condition rdf:ID="Condition_11"/>
</process:ifCondition>
<process:else>
<process:Perform rdf:ID="Perform_13">
<process:process rdf:resource="#rejectQuote"/>
</process:Perform>
</process:else>
</process:If-Then-Else>
</list:first>
</process:ControlConstructList>
<expr:Condition rdf:ID="Condition_3"/>
<process:AtomicProcess rdf:ID="delivery"/>
<process:Sequence rdf:ID="Sequence_1">
<process:components>
<process:ControlConstructList rdf:ID="ControlConstructList_6">
<list:first>
<process:If-Then-Else rdf:ID="If-Then-Else_5">
<process:ifCondition rdf:resource="#Condition_3"/>
<process:then>
<process:Perform rdf:ID="Perform_7">
<process:process rdf:resource="#order"/>
</process:Perform>
</process:then>
<process:else>
<process:Perform rdf:ID="Perform_8">
<process:process rdf:resource="#rejectQuote"/>
</process:Perform>
<process:else>
  </process:IF-Then-Else>
</list:first>
</process:ControlConstructList>
</process:components>
</process:Sequence>
<expr:Condition rdf:ID="Condition_12"/>
<expr:Condition rdf:ID="Condition_4"/>
<service:Service rdf:ID="BooksToYourDoorService">
  <service:presents>
    <profile:Profile rdf:ID="BooksToYourDoorProfile">
      <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        A book service and delivery
      </profile:textDescription>
      <service:presentedBy rdf:resource="#BooksToYourDoorService"/>
      <profile:has_process>
        <process:CompositeProcess rdf:ID="booksOrderProcess">
          <process:composedOf rdf:resource="#Sequence_1"/>
          <service:describes rdf:resource="#BooksToYourDoorService"/>
        </process:CompositeProcess>
      </profile:has_process>
      <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        BooksToYourDoorService
      </profile:serviceName>
    </profile:Profile>
    <service:describedBy rdf:resource="#booksOrderProcess"/>
  </service:presents>
</service:Service>
<expr:Condition rdf:ID="Condition_10"/>
<process:AtomicProcess rdf:ID="payment"/>
<expr:Condition rdf:ID="Condition_6"/>
<process:AtomicProcess rdf:ID="deliveryDetail"/>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.2.1, Build 365)  http://protege.stanford.edu -->