Semantic Web Enabled Composition of Web Services

Brahim Medjahed

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Dr. Athman Bouguettaya, Chair,
Dr. Reza Barkhi
Dr. Ing-Ray Chen
Dr. Mohamed Eltoweissy
Dr. Naren Ramakrishnan

January 19th, 2004
Falls Church, Virginia, USA

Keywords: Semantic Web - Web Service - Ontology - Service Composition.

Semantic Web Enabled Composition of Web Services

Brahim Medjahed

(ABSTRACT)

In this dissertation, we present a novel approach for the automatic composition of Web services on the envisioned Semantic Web. Automatic service composition requires dealing with three major research thrusts: semantic description of Web services, composability of participant services, and generation of composite service descriptions. We first propose an ontology-based framework for organizing and describing semantic Web services. We introduce the concept of *community* to cluster Web services based on their domain of interest. Each community is defined as an instance of an ontology called community ontology. We then propose a composability model to check whether semantic Web services can be combined together, hence avoiding unexpected failures at run time. The model defines formal safeguards for meaningful composition through the use of composability rules. We also introduce the notions of composability degree and τ-composability to cater for partial and total composability. Based on the composability model, we propose a set of algorithms that automatically generate detailed descriptions of composite services from high-level specifications of composition requests. We introduce a Quality of Composition (QoC) model to assess the quality of the generated composite services. The techniques presented in this dissertation are implemented in *WebDG*, a prototype for accessing e-government Web services. Finally, we conduct an extensive performance study (analytical and experimental) of the proposed composition algorithms.

This work is supported by the NSF Digital Government Program under grant 9983249-EIA.

To my Mother and Father,
Wife and Daughter,
Sisters and Brothers.

Acknowledgments

I would like to thank my advisor Athman Bouguettaya for his full support during my studies. I am very fortunate to have had the opportunity to work under his supervision. Athman instilled a thirst for excellence in me, taught me how to do scholarly research, and helped me think creatively and independently. His guidance and patience during my Ph.D. research are greatly appreciated. I will never forget my enjoyable experience working with Athman. I would like also to express my gratitude to Reza Barkhi, Ing-Ray Chen, Mohamed Eltoweissy, and Naren Ramakrishnan for serving on my Thesis committee and for their helpful comments.

I thank my co-authors: Salman Akram, James Beard, Boualem Benatallah, Athman Bouguettaya, Jerry Cameron, Ahmed Elmagarmid, Weiping He, Lily Hendra, Hao Long, Yao Meng, Anne Ngu, Mourad Ouzzani, Abdelmounaam Rezgui, and Xu Yang for their productive and enjoyable collaborations. I would like also to thank the anonymous reviewers for their comments on earlier drafts of my papers.

If I were to name two persons who deserve the most thanks, it would be my mother and father. I would like to thank them for their constant love and support. I am indebted to my wife for her tireless encouragement. Her presence, at happy and hard times, and invaluable moral help were instrumental to the conduct of my research. I thank her for all her patience and support. For continuous encouragement, my sister receives my everlasting thanks. She always pushed me to pursue a Ph.D. and did not hesitate to provide me with the necessary help when I decided to do so. I also offer my thanks to my daughter Lina, my sisters, and brothers for their love and support.

I owe special thanks to my friend, brother, officemate, and ex-roommate Mourad Ouzzani. He is the one who helped me start my Ph.D. study. I would like to thank him and my dear friend Abdelmounaam Rezgui for the countless hours we spent together discussing research issues and other matters of life. I also enjoyed the time with my officemates and friends Salman Akram, Weiping He, Zaki Malik, Mourad Ouzzani, Abdelmounaam Rezgui, and Xu Yang. Last but not least, I would

like to thank Charmaine Carter, Program Support Technician at the Department of Computer Science within Virginia Tech, for making all administrative business transparent to me.

Contents

List of Figures **x**

List of Tables **xiv**

1 Introduction **1**

1.1 Semantic Web Services 2

1.2 Web Service Composition: Motivation 3

1.3 Case Study: E-Government Web Services 4

1.4 Thesis Statement 7

1.5 Automatic Composition of Semantic Web Services 8

1.5.1 Different Types of Composition 10

1.5.2 Research Issues 11

1.6 Major Contributions 14

1.7 Dissertation Organization 16

2 Interactions on the Web: A Taxonomic Perspective **18**

2.1 Architecture of a Web-based Interaction Framework 20

2.2 A Taxonomy for Semantic Web Interactions 22

2.2.1 Interaction Layers 22

2.2.2 Dimensions for Semantic Web Interactions 24

2.3 Interactions in the Pre Semantic Web Era 27

2.3.1 Electronic Data Interchange (EDI) 28

2.3.2 Software Components 33
2.3.3 Workflows 39
2.4 Trends in Supporting Semantic Web Interactions 43
2.4.1 Ontologies 43
2.4.2 Web Services 48
2.4.3 Software Agents 54
2.4.4 XML-based Interaction Standards 57
2.5 Summary and Discussion 62
2.5.1 Comparison of Semantic Web Interaction Technologies 63
2.5.2 Web Services and Related Technologies 65
2.5.3 The Role of Web Services in the Semantic Web Landscape . . 67

3 An Ontological Framework for Web Services 71
3.1 The Proposed Model for Semantic Web Services 72
3.1.1 Ontological Support for Web Services 72
3.1.2 Structure of a Community 74
3.1.3 Generic Operations 78
3.1.4 Community Members 80
3.2 Operational Description of Communities 81
3.2.1 Syntactic Attributes 81
3.2.2 Static Semantic Attributes 83
3.2.3 Dynamic Semantics 86
3.2.4 Qualitative Properties 92
3.3 Registering Web Services With Communities 95
3.3.1 The Web Service Registration Process 95
3.3.2 Importing Generic Operations 98
3.4 A Peer-to-Peer Approach for Managing Communities 101
3.4.1 Propagating Changes Initiated by Community Providers . . . 101
3.4.2 Propagating Changes Initiated by Service Providers 104

4 A Composability Model for Semantic Web Services 111
4.1 The Proposed Model for Composability . . . 112
4.1.1 Horizontal, Vertical, and Hybrid Composition . . . 114
4.1.2 Properties of a Composability Rule . . . 116
4.1.3 Composability Degree . . . 118
4.1.4 τ-Composability . . . 121
4.2 Syntactic Composability Rules . . . 122
4.2.1 Composability at the Operation Granularity . . . 122
4.2.2 Composability at the Message Granularity . . . 124
4.3 Static Semantic Composability Rules . . . 125
4.3.1 Composability at the Operation Granularity . . . 125
4.3.2 Composability at the Message Granularity . . . 127
4.4 Dynamic Semantic Composability . . . 131
4.5 Qualitative Composability . . . 135
4.6 Business Process Composability . . . 137
4.6.1 Composition and Stored Templates . . . 138
4.6.2 Composition Soundness . . . 139

5 Automatic Composition of Semantic Web Services 141
5.1 Specification of Composition Requests . . . 143
5.1.1 Orchestration Model . . . 143
5.1.2 Describing Composition Sub-Requests . . . 145
5.1.3 Customization via Composer Profiles . . . 147
5.2 Outsourcing Web Services in the Matchmaking Phase . . . 148
5.2.1 One-to-One Vertical Composition . . . 149
5.2.2 One-to-Many Vertical Composition . . . 162
5.3 Generating Composite Service Descriptions . . . 167
5.3.1 Replacing Sub-requests by Composition Plans . . . 168
5.3.2 Inserting Pre and Post-Operations . . . 168
5.3.3 Quality of Composition . . . 170

6 Implementation and Performance Study **173**
6.1 WebDG Prototype . . . 174
6.1.1 WebDG Services . . . 174
6.1.2 Architecture . . . 176
6.1.3 WebDG Scenario . . . 178
6.2 Web Service Benchmarking Framework . . . 180
6.2.1 Statistical Distribution Models in WSBF . . . 181
6.2.2 Generation Flow . . . 183
6.2.3 Architecture . . . 185
6.3 Performance Study . . . 188
6.3.1 Analytical Model . . . 189
6.3.2 Experiments . . . 200
6.3.3 Summary . . . 208

7 Related Work **210**
7.1 Automatic Composition of Web Services . . . 210
7.2 Research Prototypes . . . 214
7.3 Standardization Efforts . . . 218
7.4 Deployment Platforms . . . 222

8 Conclusions **230**
8.1 Summary . . . 230
8.2 Directions for Future Research . . . 232

Bibliography **236**

A Vita **251**

List of Figures

1.1 Web Services Market Growth 2
1.2 Case Study - Government Social and Welfare Services 5
1.3 Composing E-Government Web Services 6
1.4 Automatic Composition of Web Services 9

2.1 Architecture of a Web-based Interaction Framework 20
2.2 Semantic Web Interactions: A Running Example 21
2.3 EDI-based Interactions 28
2.4 Component-based Interactions 34
2.5 Workflow System Characteristics 40
2.6 The Web Service Reference Model 50
2.7 XML-based Interaction Standards 58
2.8 Technologies in a Semantic Web Interaction Framework 68

3.1 The Proposed Web Service Model 73
3.2 Subset of the DAML+OIL Specification for the Community Ontology 75
3.3 The General Structure of a Community 77
3.4 Operation Execution States 87
3.5 An Example of OPD Diagram 90
3.6 The Web Service Registration Process 96
3.7 Service Registration: Member Agent Algorithm 97
3.8 Service Registration: Community Agent Algorithm 98

3.9 Propagating Changes Initiated by a Community Provider to its Members . . . 102
3.10 Propagating Changes Initiated by Community Providers to their Peers 103
3.11 Generic Operation Modification: Community Agent Algorithm 105
3.12 Propagating Changes Initiated by Service Providers 106
3.13 Changes Issued by Service Providers: Member Agent Algorithm . . . 107
3.14 Changes Issued by Service Providers: Community Agent Algorithm . 109
3.15 Reaction to Changes Issued by Community Providers: Member Agent Algorithm . . . 110

4.1 Composability Stack . . . 113
4.2 Horizontal Composition . . . 115
4.3 Vertical Composition . . . 116
4.4 Hybrid Composition . . . 117
4.5 B-Composability Rules . . . 132
4.6 Relationships Between B-Composability Rules . . . 136
4.7 Example of Composition Template . . . 138
4.8 Example of Stored Template . . . 139

5.1 Overview of the Proposed Approach for Service Composition . . . 142
5.2 Modeling Composition Request through Activity Diagrams . . . 144
5.3 Composition Request with one Single Sub-request . . . 145
5.4 Description of a Composition Request in CSL . . . 146
5.5 Main Matchmaking Algorithm . . . 148
5.6 Operation-Centric Algorithm for 1:1 Vertical Composition . . . 150
5.7 Syntactic and Operation Semantic Composability . . . 151
5.8 Message Composability . . . 152
5.9 Behavioral Composability . . . 153
5.10 Community-Centric Algorithm for 1:1 Vertical Composition . . . 157
5.11 Automaton States and Transitions Definition . . . 159
5.12 Updating Input Automata . . . 160

5.13 Message-Centric Algorithm for 1:1 Vertical Composition 161
5.14 The Flow Graph 163
5.15 Updating the Flow Graph 164
5.16 Flow Graph for the Driving Directions Example 165
5.17 Flow Graph for the Language Translation Example 165
5.18 One-to-many Vertical Composition 167
5.19 Replacing a Sub-request by its Composition Plan 169
5.20 Replacing a Sub-request by its Composition Plan 169
5.21 Inserting Pre and Post Operations 170
5.22 Algorithm for Composition Soundness 171

6.1 WebDG Architecture 177
6.2 Discovering E-Government Services 179
6.3 Mappings of the Composite Service Operations 180
6.4 UML Class Diagram for Statistical Models 182
6.5 UML Activity Diagram for Web Service Generation 184
6.6 WSBF Architecture 186
6.7 WSBF Generation Interface 187
6.8 WSBF Analyzer Interface 188
6.9 WSBF Publisher Interface 189
6.10 Composition Time for the Operation Centric Algorithm 194
6.11 Composition Time for the Community Centric Algorithm 195
6.12 Community-centric Algorithm for Different Numbers of Communities 196
6.13 Composition Time for the Message Centric Algorithm 198
6.14 Composition Time for Operation, Community, and Message Centric Algorithms 199
6.15 Static Semantic Composition for Operation, Community, and Message Centric Algorithms 201
6.16 Experiments - Syntactic and Semantic Composability for the Operation Centric Algorithm 203

6.17 Experiments - Syntactic and Semantic Composability for the Community Centric Algorithm 204
6.18 Experiments - Syntactic and Semantic Composability for the Message Centric Algorithm 205
6.19 Experiments - Community Centric Algorithm for Various Numbers of Communities 206
6.20 Experiments - Total Composition Times for the Different Algorithms 207
6.21 Experiments - Static Semantic Composition Times for the Different Algorithms 208

List of Tables

1.1 Different Types of Composition 11

2.1 Interaction Layers 23
2.2 Interaction Dimensions 24
2.3 Agent Properties 54
2.4 Pre Semantic Web Technologies vs. Interaction Layers 64
2.5 Semantic Technologies vs. Interaction Layers 64
2.6 Pre Semantic Web Technologies vs. Interaction Dimensions 69
2.7 Semantic Web Technologies vs. Interaction Dimensions 70

3.1 Assigning Values to Generic Operation Attributes 79
3.2 Quality of Operation Model 93

4.1 Composability Rules Properties 118
4.2 Instantiations of Generic B-Composability 135

5.1 Inference Rules for the “$>$” Operator 154
5.2 Inference Rules for the “$\geq$” Operator 154
5.3 Inference Rules for the “$<$” Operator 155
5.4 Inference Rules for the “$\leq$” Operator 155
5.5 Inference Rules for the “$\neq$” Operator 155
5.6 Inference Rules for the “$=$” Operator 156

6.1 Enabling Technologies 175

6.2 Symbols and Parameters 190
6.3 Simulation settings 200
6.4 Experiments - Syntactic and Semantic Composability for the Operation Centric Algorithm 202
6.5 Experiments - Syntactic and Semantic Composability for the Community Centric Algorithm 203
6.6 Experiments - Syntactic and Semantic Composability for the Message Centric Algorithm 204
6.7 Experiments - Community Centric Algorithm for Various Numbers of Communities 205
6.8 Experiments - Total Composition Times for the Different Algorithms 206
6.9 Experiments - Static Semantic Composition Times for the Different Algorithms 207

7.1 Prototypes vs. Interaction Layers 218
7.2 Prototypes vs. Interaction Dimensions 219
7.3 Deployment Platforms 229

Chapter 1

Introduction

The Web has been an excellent tool to elicit connectivity to a wealth of information sources that had been previously inaccessible. While initially aimed to enable the sharing of information among scientists, the Web has since then evolved to cater for governments, businesses, and individuals to make their data and applications Web accessible. The original design of the Web served its purpose and went beyond anticipated predictions. However, as the number of data and applications available on the Web increased tremendously, it has become apparent that the Web could no longer sustain its growth in its present form. Indeed, a large proportion of data on the "current" Web is mostly "understandable" by humans or custom developed applications. The main impediment has been the lack of *semantics* to enable machines to *"understand"* and *automatically* process the data that they now merely display. The *Semantic Web* is an emerging paradigm shift to fulfill this goal. It is defined as an extension of the existing Web, in which information is given a well-defined *meaning* [13]. The ultimate goal of the envisioned Semantic Web is to transform the Web into a medium through which data and applications can be *automatically understood* and *processed*.

1.1 Semantic Web Services

The development of technologies for supporting the envisioned Semantic Web has been the priority of various research communities (e.g., database, artificial intelligence). A major player in enabling the Semantic Web is the concept of *Web service* [5, 74, 77]. A *Web service* is a set of related functionalities that can be programmatically accessed through the Web. Examples of Web services span several application domains including *e-government* (e.g., e-tax preparation) and *B2B E-commerce* (e.g., stock trading). Web services are gradually taking root because of the convergence of business and government efforts to making the Web the place of choice for all types of activities. The maturity of XML-based Web service technologies such as SOAP, UDDI, and WSDL is a prominent factor contributing to the large adoption of Web services in the near future [37]. Figure 1.1 summarizes the growth of the Web services market as predicted in [122]. The prediction conforms to *Cringely's law*: "short-term adoption of new technologies never occurs as quickly as we expect, but the long-term impact is far greater than we realize".

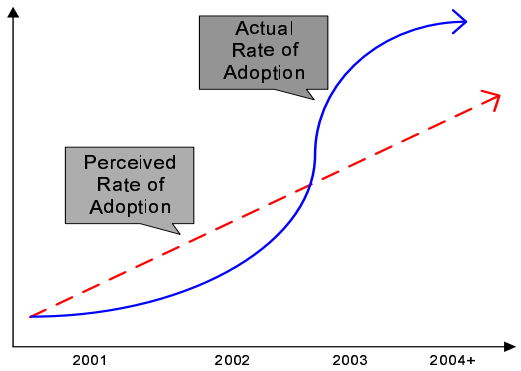

Figure 1.1: Web Services Market Growth

Another key player that is taking the spotlight in the envisioned Semantic Web is the concept of *ontology*. An *ontology* is defined as a *formal* and *explicit* specification of a *shared conceptualization* [13, 133]. Ontologies were first developed in the artificial intelligence community to facilitate knowledge sharing and reuse [48].

Nowadays, they are increasingly seen as key to enabling semantics-driven data access and processing. Ontologies are expected to play a central role to empower Web services with semantics. The combination of these powerful concepts (i.e., Web services and ontologies) has resulted in the emergence of a new generation of Web services called *Semantic Web services*. Semantic Web services are poised to be the building blocks of tomorrow's Web, i.e., the Semantic Web [74]. Applications "exposed" as Web services would be *understood*, *shared*, and *invoked* by *automated* tools.

Semantic Web services have spurred an intense activity in industry and academia to address challenging research issues such as the *automatic selection*, *monitoring*, and *composition* of Web services. The diversity of these issues calls for the design and development of a comprehensive *Web Service Management System* (WSMS), where Web services would be treated as first-class objects that can be manipulated as if they were pieces of data. A WSMS includes the architectural components necessary to tackle Web service research issues. This would introduce more convenience, flexibility, and effectiveness in managing Web services. In our PhD dissertation, we focus on the *automatic composition of semantic Web services*.

1.2 Web Service Composition: Motivation

Web service composition refers to the process of combining several Web services to provide a *value-added* service [24, 124]. It is emerging as *the* technology of choice for building cross-organizational applications on the Web [5, 77]. This is mainly motivated by three factors. First, the adoption of XML-based messaging over well-established and ubiquitous protocols (e.g., HTTP) enables communication among disparate systems. Indeed, major existing environments are able to communicate via HTTP and parse XML documents. Second, the use of a document-based messaging model in Web services caters for loosely coupled relationships among organizations' applications. This is in contrast with other technologies (e.g., *software components* [121]) which generally use object-based communication, thereby yielding systems where the coupling between applications is tight. Third, tomorrow's

Web is expected to be highly populated by Web services [28]. Almost every "asset" would be turned into a Web service to drive new revenue streams and create new efficiencies.

We identify two types of Web services: *simple* and *composite*. *Simple* services are Internet-based applications that do not rely on other Web services to fulfill consumers' requests. A *composite* service is defined as a conglomeration of outsourced Web services (called *participant services*) working in tandem to offer a *value-added* service. `Tax Preparator` is an example of composite service used by citizens to file their taxes. It combines simple Web services such as financial services at citizens' companies to get W2 information, banks' and investment companies' services to retrieve investment information, and electronic tax filing services provided by state and federal revenue agencies.

From a business perspective, Web service composition offers several advantages [122]. First, composite services allow organizations to minimize the amount of work required to develop applications, ensuring a rapid time-to-market. Second, application development based on Web services reduces business risks since reusing existing services avoids the introduction of new errors. Third, composing Web services enables the reduction of skills and effort requirements for developing applications. Finally, the possibility of *outsourcing* the "best-in-their-class" services allows companies to increase their revenue.

1.3 Case Study: E-Government Web Services

While the outcomes of our research are generic enough to be applicable to a wide range of applications, we use the area of *e-government* as a case study. One of the major concerns of e-government is to improve government-citizen interactions using information and communication technologies [83, 16, 18, 102]. In the *WebDG* (Web Digital Government) project, we have teamed up with *Indiana*'s *Family and Social Services Administration* (FSSA) and *Virginia Department for the Aging* (VDA). The FSSA provides welfare programs to assist low income citizens, strengthen families

and children, and help elderly and disabled people. VDA offers a large spectrum of programs and services to assist senior citizens. However, collecting social benefits is currently a frustrating and cumbersome task in both FSSA and VDA. Citizens must often visit different offices located within and outside their home town. Additionally, case officers must delve into a wealth of proprietary applications to access welfare programs that best meet citizens' needs.

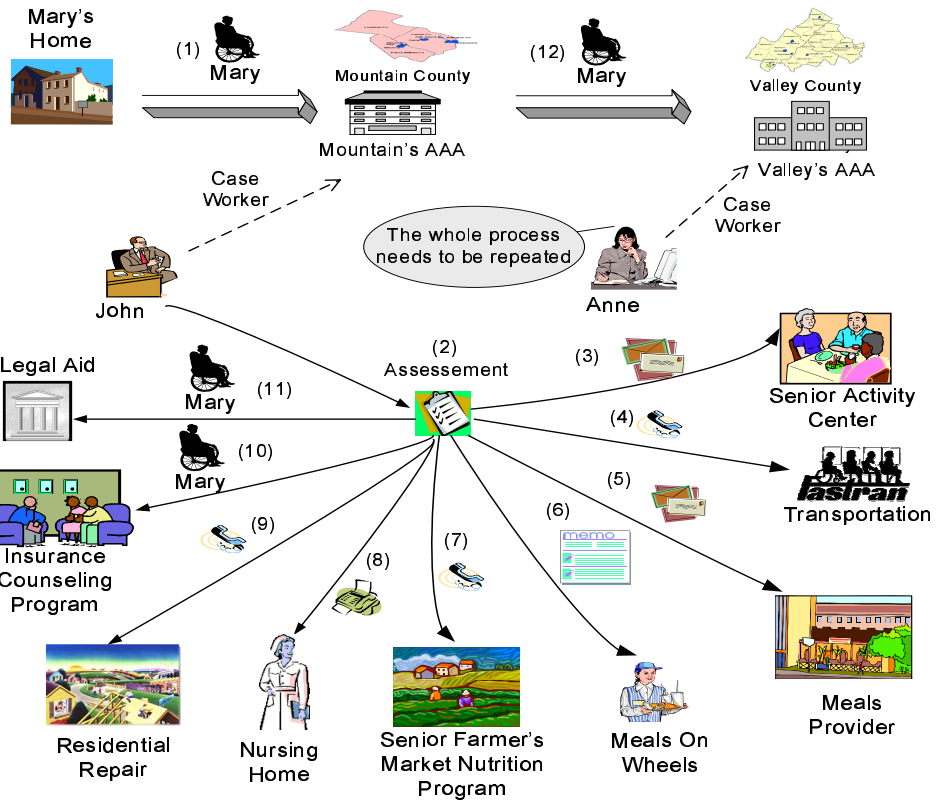

Figure 1.2: Case Study - Government Social and Welfare Services

Let us consider the following scenario typical to VDA application domain (Figure 1.2). Assume that citizen *Mary*, a handicapped and indigent retiree, wants to receive services from an Area Agency on Aging (AAA). Typically, she would have to travel to *Mountain* County's AAA for an interview. In this case, John, a case worker at the agency, would assess the kind of services Mary would need. He would

delve into a large number of social services and match the features of those services with Mary's particular needs. John determines that Mary may qualify for the following services: *FastTran* (transportation for the elderly and handicapped), *Meals on Wheels*, *Meals Providers*, *Senior Activity Center*, *Residential Repair*, *Nursing Home*, *Senior Market Nutrition Program*, *Insurance Counseling Program*, and *Legal Aid*. Mary's information is transmitted using different means of communication, including email, snail mail, fax, and phone. Mary may also have to visit some of the agencies such as the insurance counseling agency. Delay in processing is usually the rule and not the exception in these cases. To further illustrate the inadequacy of the current system, assume that Mary decides to move to *Valley* county because she developed high altitude sickness. The case worker at Valley's AAA would then initiate the same highly manual and error-prone process.

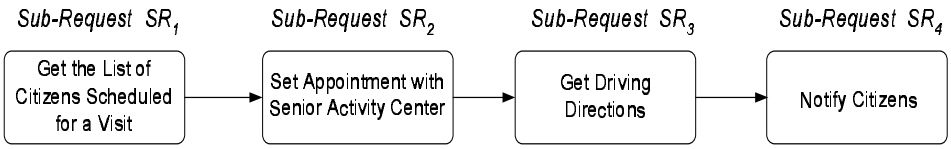

Figure 1.3: Composing E-Government Web Services

This difficulty in collecting social benefits prevents senior citizens from becoming self-dependent with a consequent harmful impact on their welfare and health. To facilitate the use of VDA applications and hence expeditiously satisfy citizens' needs, we organize these applications into Web services. Those services may be used "individually" or combined together to provide value-added services. Assume that *John* is planning to organize a visit to a Senior Activity Center (SAC). *John*'s request includes several *sub-requests*. Each sub-request would typically be performed by executing one or more Web services (Figure 1.3). *John* first retrieves the list of citizens interested in visiting an SAC (SR_1). We assume that *John* gets the names and zip codes of those citizens instead of their full addresses. *John* then sets an appointment to visit a senior activity center (SR_2). Once a visit is scheduled, *John* gets the driving directions from each citizen's location to the SAC (SR_3). He finally

notifies each citizen about the date and time of the visit and the driving directions to the SAC (SR_4).

1.4 Thesis Statement

Web service composition has recently taken a central stage as an emerging research area. Several techniques have been proposed [10, 25, 70, 90, 113]. Standardization efforts are under way for supporting Web service composition (e.g., *BPEL4WS* [9]). However, these techniques and standards provide little or no support for the semantics of participant services, their messages, and interactions. Additionally, they generally require dealing with low level programming details which may lead to unexpected failures at run-time.

To illustrate the challenges raised by Web service composition, let us consider the composition request depicted in Figure 1.3. Assume that sub-requests SR_2 and SR_3 are executed by invoking the Web services `Lookup-SAC` and `Schedule-Visit`, respectively. In this case, social worker (composer) should understand the exact format, content, and semantics of messages exchanged between `Lookup-SAC` and `Schedule-Visit`. He must also check that `Lookup-SAC` and `Schedule-Visit` "can" actually be combined and "manually" specify the way their messages are mapped to each other. Additionally, he needs to identify the way to invoke `Lookup-SAC` and `Schedule-Visit` (e.g., protocols supported by `Lookup-SAC` and `Schedule-Visit`). He should finally determine the way `Lookup-SAC` and `Schedule-Visit` can together define an overall business process (e.g., order of messages, semantics of interactions).

A promising approach to dealing with the aforementioned issues is the *automation* of the composition process [74]. This tedious process would then be conducted without human intervention. The less efforts are required from users, the easier and faster Web services are composed. We propose a framework for the *automatic composition of Semantic Web services*. Composers would specify the *what* part of the desired composition (i.e., the tasks to be performed), but will not concern themselves with the *how* part (e.g., which services will be outsourced). They would

provide "abstract" definitions of the actions they would like to perform. The process of composing Web services (selecting Web services, plugging their operations, and so forth) would be transparent to users. Detailed descriptions of composite services would be automatically generated from composers' specifications.

Several characteristics of Web service environments entangle the automatic composition process. First, the number of services available on the Web is growing at a very fast pace [28]. Service composers must delve into the potentially vast amount of available services, find services of interest, check whether they can interact with each other, and then compose them. Second, the Web service space is highly dynamic. New services are expected to avail themselves on the Web. This requires the ability to select the "best" and "relevant" available participants in a composite service at any given time [25]. Third, participant services are generally deployed in heterogeneous environments. Heterogeneity occurs at different levels including syntactic (e.g., communication) and semantic (e.g., content, business logic) levels. Composite services need to "understand" and deal with the peculiarities of each participant service. Finally, the execution of a composite service typically spans organizational boundaries and requires the capability of interacting with Web services that are autonomous. Participant services cannot be considered to be "subservient" to other services [117]. They should instead be perceived as interacting independently with each other.

1.5 Automatic Composition of Semantic Web Services

To illustrate the major research issues for developing a Semantic Web enabled service composition approach, let us consider our e-government scenario (Figure 1.3). The *composition engine* would delve into the service space to determine participants that "best" serve each sub-request (Figure 1.4). The following simple services are found relevant to sub-requests SR_1, SR_2, SR_4, respectively: `Get-Citizens-List`,

`Schedule-Visit`, and `Notify-Citizens`.

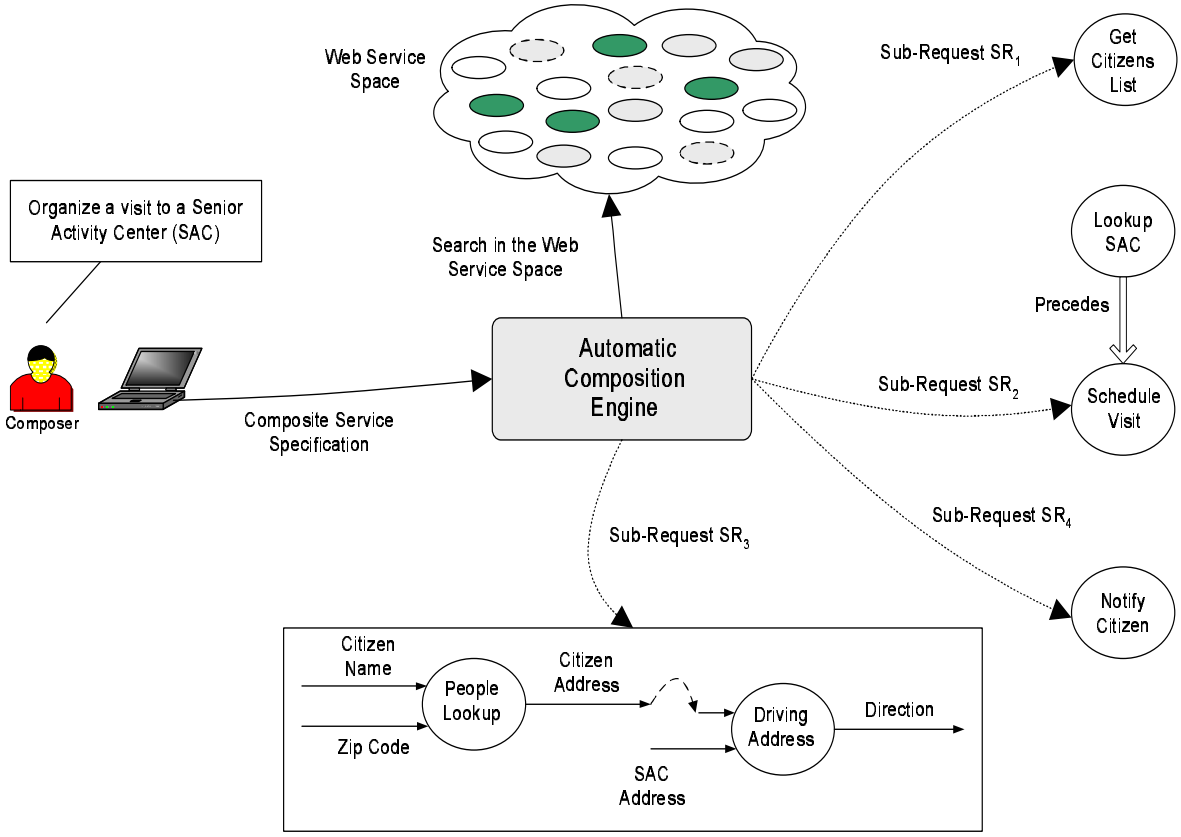

Figure 1.4: Automatic Composition of Web Services

The "Get Driving Directions" sub-request (SR_3) returns the driving directions, given a citizen's name, zip code, and the address of the SAC. Since there is no simple service that offers such functionality, one solution would be to compose existing Web services in a way that would transparently fulfill the desired objective (i.e., sub-request SR_3). The composition engine finds the following two simple services as relevant: `People-Lookup` and `Direction-From-Address`. `People-Lookup` returns citizens' addresses, given their names and zip codes. `Direction-From-Address` returns the driving directions, given an initial and final address. The composition engine would then automatically compose `People-Lookup` and `Direction-From-Address`

to execute the SR_3 (Figure 1.4).

To make our scenario even more challenging, let us consider relationships that may exist between Web services. For example, the invocation of the `Schedule-Visit` service *requires* the invocation of the `Lookup-SAC` service to get the list of senior activity centers. Such *pre-execution* relationships are generally dictated by the business logic of Web services (e.g., `Lookup-SAC` and `Schedule-Visit`). They may also reflect government regulations. For example, applying for certain welfare programs (e.g., unemployment benefits) may require access to the applicant record with a taxation office. Note that Web services may also be linked by *post-execution* relationships. The composition engine should be able to automatically include pre- and post-execution relationships in the generated composite service.

1.5.1 Different Types of Composition

The example depicted in Figure 1.4 introduces the notions of composition *cardinality* and *mode*. The composition *cardinality* refers to the number of participants selected for each sub-request. We define two cardinalities: *one-to-one* and *one-to-many*. In the *one-to-one* cardinality, a sub-request is mapped to one participant service. For example sub-request SR_1 is mapped to one service that is, `Get-Citizens-List`. Another example is that of sub-request SR_4. This sub-request is mapped to the service `Notify Citizen`. In the *one-to-many* cardinality, a sub-request is mapped to a set of participants executed in a specific order. For example, sub-request SR_3 is mapped to two services namely, `People-Lookup` and `Direction-From-Address`.

The composition *mode* refers to the way participants are combined. We define three composition types: *horizontal*, *vertical*, and *hybrid*. *Horizontal* composition refers to a "supply chain"-like combination of Web services. For example, `People-Lookup` and `Direction-From-Address` are horizontally combined in the order `People-Lookup` $\rightarrow$ `Direction-From-Address`. *Vertical* composition refers to the "outsourcing" of a Web service by a sub-request or another Web service. For example, SR_2 is vertically composed with `Lookup-SAC`. *Hybrid* composition combines

horizontal and vertical composition. For example, SR_3 is vertically composed with two services, `People-Lookup` and `Direction-From-Address`, that are horizontally composed.

	One-to-One	One-to-Many
Horizontal	NA	Yes
Vertical	Yes	NA
Hybrid	Yes	Yes

Table 1.1: Different Types of Composition

Based on the notions of composition mode and cardinality, several types of composition are possible. We summarize in Table 1.1, the different types of compositions. "NA" (Not Applicable) means that the corresponding combination of composition mode and cardinality is not feasible. Horizontal composition implies the combination of several Web services. It is hence used with one-to-many cardinality. Vertical composition refers to the outsourcing of one Web service. It is hence combined with one-to-one cardinality. Hybrid composition refers to the most general mode. It may be used with both horizontal and vertical compositions. For example, the composition depicted in Figure 1.4 is hybrid. It combines horizontal (e.g., SR_3) and vertical (e.g., SR_4) compositions. In this dissertation, we consider the four types of composition mentioned in Table 1.1.

1.5.2 Research Issues

As illustrated in the aforementioned example, the automatic composition of Semantic Web services raises the following challenging issues:

- *Specification of Composers' Requests*: Composers should *specify* their *requests* for composition (e.g., "organize a visit to a senior activity center") in an unambiguous way. A composer's request may include several *sub-requests* (e.g., lookup for a senior activity center). The issues that need to be addressed are as follows: (i) whether composers specify all, some, or no participant services;

(ii) how should the composition engine "interpret" each composer's request; and (iii) how would the different sub-requests be *orchestrated*. *Orchestration* refers to the execution order (sequential, parallel, etc.) of the different sub-requests and the condition under which a certain sub-request may or may not be executed.

- *Understanding the Semantics of Web Services*: Once the composition engine has received and "interpreted" a request for composition, it should delve into the Web service space to locate "potential" participants. Because of the large size of this space and sheer heterogeneity of Web services, there is a need to define a "meaningful" organization of that space to filter interactions and accelerate service searches. Web services should be described in a way that captures their semantics. In our e-government scenario, Web services may be located in the same county (local agencies), different counties within a state (state agencies), or different states (federal agencies). Additionally, those services may be offered by heterogeneous providers such as state and federal government agencies (e.g., Department of Health and Human Services), businesses (e.g., restaurants participating in a subsidized government program), volunteer centers (e.g., meal deliverers), and non-profit organizations (e.g., American Red Cross). The composition engine should be able to limit its search to Web services that are relevant to the composition request. It should also "understand" that `Direction-From-Address` provides "driving direction from one location to another".

- *Checking the Composability of Participant Services*: Let us now assume that the composition engine is able to understand the capabilities of Web services. The next step would be to select participant services. The selection process should be done while making sure that participants "can" actually interact with each other. We refer to such a task as *composability*. In our e-government scenario, the composition engine must verify that `People-Lookup` is *composable with* `Direction-From-Address` as depicted in Figure 1.4. The issue is to

develop a model that clearly defines the Web service features that need to be compared for composability. For example, there is a need to make sure that the message parameter returned by `People-Lookup` is "similar" (e.g., in terms of their semantics) to the parameter required by `Direction-From-Address`. The composability process should compare Web service features at different "*granules*" (e.g., messages, operations, services) and levels (e.g., syntactic, semantic, and qualitative). Because of the heterogeneity of Web services, it would be unrealistic to assume that participants are "fully" composable. For example, `People-Lookup` and `Direction-From-Address` may agree on the semantics of their message parameters but use different communication protocol such as SOAP/HTTP and SOAP/MIME. The composition engine should not return a boolean type of answers regarding the composability of Web services. It should be flexible enough to cater for *partial* and *total* composability.

- *Generating Composite Service Descriptions*: The composition engine finally generates a *description* of the composite service. This description should include details such as the list of participants services, their orchestration (i.e., execution order), the way they are interconnected, and the mappings between their messages. The generation process should consider the four types of composition mentioned in Table 1.1. The *orchestration* of participant services is an important issue that needs to be addressed during the generation process. We define two types of orchestration: *composer-defined* and *system-generated*. The composer-defined orchestration is specified by users in their composition requests. For example, Figure 1.3 states the execution order of sub-requests SR1, SR2, SR3, and SR4. The system-generated orchestration is automatically derived during the composition process. It includes services that are horizontally composed (e.g., sub-request SR3), vertically composed (e.g., sub-request SR1), and pre/post-execution relationships (e.g., sub-request SR2). Several composite service descriptions may be generated for a given composition request. The composition engine should, in this case, be able to assess

the "quality" of the generated composite services.

1.6 Major Contributions

We propose a generic approach for the automatic composition of Web services on the Semantic Web. We provide an implementation of our approach in the *WebDG* prototype. More precisely, our research contribution focuses on the following:

- *Semantics-aware Description and Organization of the Web Service Space* [76, 78, 81, 11, 4]: We propose an ontology-based framework for organizing and describing Web services on the Semantic Web. This framework provides the architectural foundation for the automatic composition of Web services. We introduce the concept of *community* to cater for an ontological organization of Web services. Web Services are clustered into communities based on their domain of interest. Service providers identify a community of interest and register their service with it. Each community is defined as an instance of an ontology called *community ontology*. The community ontology includes a set of *generic* operations that can be used "as is" or customized by underlying services. A generic operation is identified by a set of attributes including *syntactic* (e.g., message parameters), *semantic* (e.g., purpose), *behavioral* (e.g., business logic), and *qualitative* (e.g., cost, time) properties.

- *Multilevel Composability Model for Semantic Web Services* [83, 79]: We propose a *composability model* to check whether Web services can be combined together, hence avoiding unexpected failures at run time. Composability is checked through a set of *rules* organized into five *levels*: *syntactic*, *static semantic*, *dynamic semantic*, *qualitative*, and *business process* levels. The first four levels check composability of service messages and operations. Each rule in a given level compares a specific pair of *attributes* of interacting Web services. The *business process* level checks composability at the composite service "granule". The corresponding rules verify the "meaningfulness" of a given compo-

sition of Web services, that is, whether a composition provides a *value-added* service. Each composability rule specifies the constraints and requirements for checking horizontal, vertical, and hybrid composability. We also define the notions of *composability degree* and τ-*composability* to cater for *partial* and *total* composability.

- *Automatic Composition of Semantic Web Services* [80, 82]: We propose an approach for the automatic composition of semantic Web services. The proposed approach consists of three conceptually separate phases: *specification*, *matchmaking*, and *generation*. The *specification* phase enables high level and customized descriptions of the desired compositions. Composers' specifications include constructs for the orchestration and semantic description of composition sub-requests. The *matchmaking* phase uses the composability model to generate *composition plans* that conform to composers' specifications. By *composition plan*, we refer to the list of participant services and the way those services interact with each other (plugging operations, mapping messages, etc.) to "realize" the corresponding sub-request. We develop a set of algorithms for checking composability and generating composition plans. The algorithms consider both *one-to-one* and *one-to-many* composition cardinalities. The *generation* phase returns detailed composite service descriptions. Such descriptions include the *orchestration* (composer-defined and system-generated) of participant services. We also define a *Quality of Composition* (*QoC*) model to assess the *quality* of the generated composite service.

- *Implementation and Performance Study* [83, 17]: We provide an implementation of the proposed techniques in the *WebDG* prototype. We adopt emerging Web service standards including WSDL, UDDI, and SOAP. We define an analytical model for studying the performance of the proposed algorithms for composition. We also conduct a set of experiments to evaluate the performance and scalability of these algorithms. For that purpose, we define a testbed for Web services called *Web Services Benchmarking Framework* (*WSBF*). The

testbed *mimics* characteristics of real Web service environments such as dynamics, size, and heterogeneity (i.e., different service capabilities). The main features of *WSBF* include customized generation (i.e., user-controlled generation), use of statistical distribution models (e.g., *Poisson* for service arrival rate) and extensibility (e.g., adding new service attributes). Finally, we compare the results obtained through the analytical model with those returned by the experimental study.

1.7 Dissertation Organization

The remainder of this dissertation is organized as follows.

In Chapter 2, we present an in-depth study of interaction technologies on the pre Semantic Web and Semantic Web eras. We propose a framework for comparing Semantic Web interaction technologies. The framework identifies the interaction layers and proposes a set of dimensions to study interaction solutions. We compare major interaction technologies (e.g., workflows, software components, software agents, Web services), and illustrate the role of Web services in enabling interactions on the Semantic Web.

In Chapter 3, we propose an *ontological* framework for organizing and describing Web services on the Semantic Web. We introduce the concept of *community* to cater for an ontological organization and description of Web services. We develop an ontology, called *community ontology*, that serves as a "template" for describing communities and semantic Web services. We also propose a peer-to-peer approach for managing communities in highly dynamic environments. In particular, we present techniques for registering Web services with communities and coping with changes that occur in the Web service space.

In Chapter 4, we propose a *composability model* for Semantic Web services. We provide formal safeguards for meaningful composition through the use of *composability rules*. Composability rules are organized into five levels: *syntactic*, *static semantic*, *dynamic semantic*, *qualitative*, and *business process* levels. We introduce

the notions of *composability degree*, and τ-*composability* to cater for partial and total composability.

In Chapter 5, we present a generic approach for the automatic composition of Semantic Web services. The composition process is conducted in three separate phases: *specification*, *matchmaking*, and *generation*. We define constructs for the high-level specification of composition requests. We then propose a set of algorithms for checking composability and matching composers' requests with "relevant" compositions of Web services. We define a technique for automatically generating detailed descriptions of a composite service. We finally introduce a Quality of Composition (QoC) model for assessing the generated descriptions.

In Chapter 6, we describe the implementation of our approach for service composition in the *WebDG* prototype. We also present a framework for benchmarking Web services called *Web Services Benchmarking Framework* (*WSBF*). We finally conduct an extensive performance study using two approaches: analytical model and simulation experiments. We use *WSBF* as a testbed for conducting our experiments.

In Chapter 7 we describe the major techniques, standards, and platforms for Web service composition that are most closely related to our research.

In Chapter 8, we provide concluding remarks and discuss directions for future research.

Chapter 2

Interactions on the Web: A Taxonomic Perspective

The growth of the Web is revolutionizing the way organizations interact with their partners and customers. Businesses and government agencies are moving or have already moved their main operations to the Web to take advantage of the potential of more automation, efficient business processes, and global visibility [42, 43]. This has elicited the formation of alliances in which different partners join their applications and systems to share costs, skills and resources in offering value-added services. The ultimate goal is to have inter and intra-organization applications evolve independently, yet allow them to effectively and conveniently *interact* with each other. *Interaction* is defined as consisting of *interoperation* and *integration* with both internal and external enterprise applications.

Interactions among loosely coupled and tightly coupled systems has been, over the past twenty years, an active research topic in areas such as databases, knowledge-based systems, and digital libraries [15, 101]. However, the emerging Semantic Web has opened new research avenues because of issues such as semantics, heterogeneity, scalability, and automation. The Semantic Web requires the integration and interoperation of both applications and data. Disparate data representations between partner's systems must be dealt with. Interaction is also required at a higher level for

connecting (i) front-end with back-end systems, (ii) proprietary/legacy data sources, applications, processes, and workflows to the Web, and (iii) partners' systems.

In this chapter, we survey the main issues and concepts to interactions on the Semantic Web [77]. We propose a framework for comparing Semantic Web interaction technologies. The framework identifies the interaction layers, i.e., communication, content, and business process. It also proposes a set of dimensions to study interaction solutions. We present an in-depth study of interaction technologies on the pre Semantic Web and Semantic Web eras. We compare major Semantic Web interaction technologies (e.g., EDI, workflows, software components, software agents) using the proposed framework. Previous work dealing with interoperation in loosely coupled systems mostly focused on databases and digital libraries [115, 103]. Recent surveys addressing interactions on the Semantic Web (e.g., [2, 20, 41, 44, 69, 116]) were mostly fragmented and lacked a holistic view of the problem.

The Chapter's organization reflects the historical evolution of interaction technologies in the pre-Semantic Web and Semantic Web eras. In Section 2.1, we present a typical architecture for a Semantic Web Interaction framework. In Section 2.2, we define the different interaction layers in B2B E-commerce. We then identify a set of dimensions for comparing interaction solutions across these layers. In Section 2.3, we study several popular interaction solutions for the pre-Semantic Web, namely, EDI, components, and workflows. These solutions are evaluated against a pre-defined set of dimensions. In Section 2.4, we survey and evaluate the trends in supporting interactions in the Semantic Web. These include ontologies, Web services, agents, and XML-based standards. Finally, Section 2.5 provides a tabular comparison summary of the existing solutions for Semantic Web interactions.

2.1 Architecture of a Web-based Interaction Framework

Web-based applications generally involve several partners that interact via computerized systems (e.g., Web servers, networking services, databases) for conducting their daily business (e.g., exchanging documents, selling products, filing taxes) [19]. The building blocks for enabling such applications are provided through an *interaction framework* (Figure 2.1). These include modules for (1) defining and managing internal and external *business processes*, and (2) integrating those processes, and (3) supporting interactions with back-end application systems such as ERPs (*Enterprise Resource Planning*) [20]. A *business process* is defined as a multi-step activity that supports an organization's mission such as manufacturing a product and processing insurance claims [20].

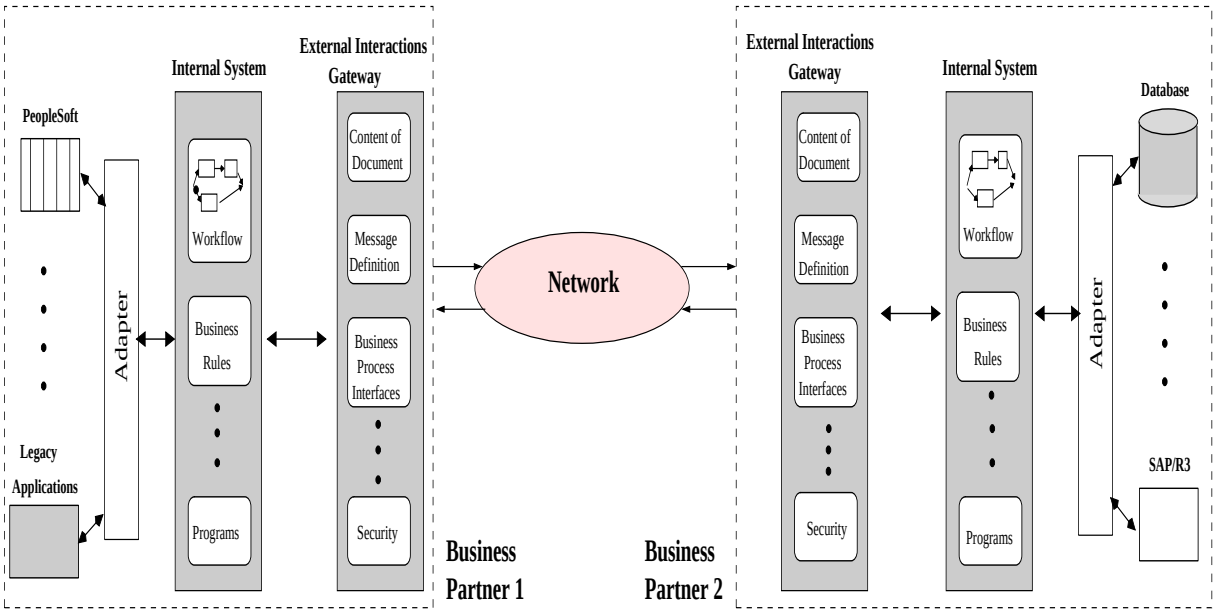

Figure 2.1: Architecture of a Web-based Interaction Framework

We depict in Figure 2.1 the main components of an interaction framework. Translation facilities (e.g., application adapters) may be used to interconnect back-end systems (e.g., databases, ERPs) and internal business processes (e.g., workflows, applications). An external business process implements the business logic of an organization with regard to its external partners such as processing messages sent by trading partners' systems. Interactions between partners' external business processes may be carried out based on a specific *standard* (e.g., EDI [92, 34], *RossettaNet* [33]) or bilateral agreements. *Interaction standards* define the format and semantics of messages (e.g., request for quote), bindings to communication protocols (e.g., HTTP, FTP), business process conversations (e.g., joint business process), security mechanisms (e.g., encryption, non-repudiation), etc. Interaction frameworks may have to support several standards and proprietary interaction protocols.

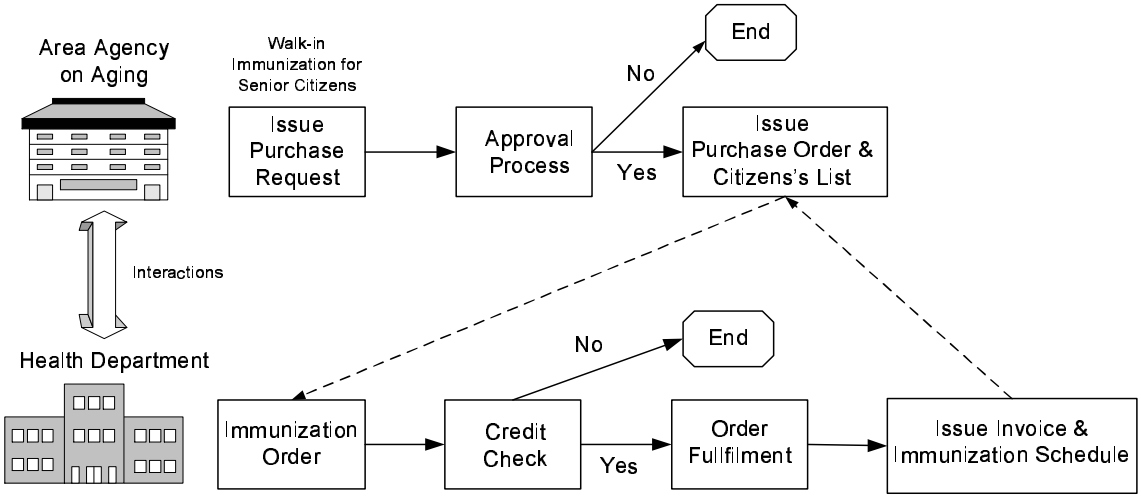

Figure 2.2: Semantic Web Interactions: A Running Example

Figure 2.2 depicts an example of interaction between the `AreaAgencyAging` and `HealthDepartment` (Section 1.3). Assume that *John*, `AreaAgencyAging`'s case worker, wants to organize a walk-in immunization for a group of disabled senior citizens. Such immunization is fee-based and provided by a separate agency, namely the

`HealthDepartment`. *John* first issues a *request for purchase*. Upon approval of this request, a *purchase order* is issued and sent to `HealthDepartment` along with the list of citizens interested by the immunization campaign. The purchase order is transformed into an *immunization order* at `HealthDepartment`'s order processing system. After satisfactory *credit check*, an *order fulfillment* is issued by `HealthDepartment`. An *invoice* and *immunization schedule* is finally sent to `AreaAgencyAging`.

2.2 A Taxonomy for Semantic Web Interactions

In the first part of this section, we identify the different layers that make up an interaction framework on the Semantic Web. We then define the dimensions for assessing interactions across these layers. These dimensions are used as a benchmark for evaluating Semantic Web interaction solutions.

2.2.1 Interaction Layers

Interactions on the Semantic Web occur in three layers: *communication*, *content*, and *business process* layers (Table 2.1). For example, `AreaAgencyAging` and `HealthDepartment` need to agree on their joint business process: *John* expects to receive an invoice and immunization schedule from the `HealthDepartment` after sending a purchase order. `HealthDepartment` needs also to "understand" the content of the purchase order sent by `AreaAgencyAging`. Finally, there must be an agreed upon communication protocol to exchange messages between `AreaAgencyAging` and `HealthDepartment`.

The communication layer provides protocols for exchanging messages among remotely located partners (e.g., HTTP, SOAP). It is possible that partners use different proprietary communication protocols. In this case, gateways should be used to translate messages between heterogeneous protocols. For example, `AreaAgencyAging` and `HealthDepartment` may use *Java RMI* (*Remote Method Invocation*) [88] and IBM's *MQSeries* [63] respectively for internal communications.

Layer	*Definition*	*Examples of Enabling Technologies*
Communication	Protocols for exchanging messages among remotely located partners	HTTP, SOAP, CORBA ORB, etc.
Content	Languages and models to describe and organize information in such a way that it can be understood and used	Ontologies, XML-based standards (e.g., eCO, cXML), etc.
Business Process	Enable autonomous and heterogeneous partners to engage in peer-to-peer interactions with each other	Web services (e.g., BPEL4WS), XML-based standards (e.g., RosettaNet), Inter-enterprise Workflows

Table 2.1: Interaction Layers

The objective of integration at this layer is to achieve a seamless integration of the communication protocols.

The content layer provides languages and models to describe and organize information in such a way that it can be understood and used. Content interactions require that the involved systems understand the semantics of content and types of business documents. For instance, if `HealthDepartment` receives a message that contains a document, it must determine whether the document represents a purchase order or request for quotation. Information translation, transformation, and integration capabilities are needed to provide for reconciliation among disparate representations, vocabularies, and semantics. The objective of interactions at this layer is to achieve a seamless integration of data formats, data models, and languages. For example, if `AreaAgencyAging` uses xCBL (*XML Common Business Library*) [32] to represent business documents and `HealthDepartment` expects documents in cXML (*Commerce XML*) [38], there is a need for a conversion between these two formats.

The business process layer is concerned with the conversational interactions (i.e, joint business process) among services. Before engaging in a transaction, `AreaAgencyAging` and `HealthDepartment` need to agree on the procedures of their joint business process. The semantics of interactions among `AreaAgencyAging` and `HealthDepartment` must be well defined, such that there is no ambiguity as to what a message may mean, what actions are allowed, what responses are expected, etc. The objective of interactions at this layer is to allow autonomous and heterogeneous partners to come online, advertise their terms and capabilities, and engage in peer-

to-peer interactions with any other partners. Interoperability at this higher level is a challenging issue because it requires the understanding of the semantics of partner business processes.

2.2.2 Dimensions for Semantic Web Interactions

The Semantic Web covers a wide spectrum of interactions among different partners. The type of interactions depend on the usage scenarios, involved parties, and business requirements. Each framework makes specific tradeoffs with regard to the requirements of Semantic Web interactions. It is therefore important to determine the relevant requirements and understand the related tradeoffs when evaluating models of interactions. In this section, we identify the following set of dimensions to study interaction issues the Semantic Web (Table 2.2):

Dimension	*Definition*
Coupling among Partners	Degree of *tightness* and *duration* of coupling among business partners
Heterogeneity	Degree of dissimilarity among business partners
Autonomy	Degree of compliance of a partner to the global control rules
External Manageability	Degree of external visibility and manageability of partners' applications.
Adaptability	Degree to which an application is able to quickly adapt to changes
Security	Measures must be in place to boost partners' confidence that their transactions are safely handled
Scalability	Ability of a system to grow in one or more dimensions such the number of relationships that can be supported

Table 2.2: Interaction Dimensions

- *Coupling among partners:* this dimension refers to the degree of *tightness* and *duration* of coupling among business partners. Two partners are *tightly coupled* if they are strongly dependent on each other. For example, one partner may control the other, or they may control one another. *Loosely coupled* partners exchange business information on demand. The *duration* of a relationship may be *transient* (also called *dynamic*) or *long term*. In *transient* relationships, businesses may need to form a fast and short term partnership (e.g., for one transaction), and then disband when it is no longer profitable to stay together.

Partners need to dynamically discover partners to team up with to deliver the required service. In *long term* relationships, businesses assume an *a priori* defined partnership.

- *Heterogeneity:* heterogeneity refers to the degree of dissimilarity among business partners. The need to access data across multiple types of systems has arisen due to the increased level of connectivity and increased complexity of the data types. Applications use different data structures (e.g., XML, relational databases), standard or propriety semantics (e.g., standardized ontologies). There may also be structural heterogeneity at the business process layer (e.g., use of APIs, document exchange protocols, inter-enterprise workflows). In addition, organizations may, from a semantic point of view, use different strategies for conducting business that depend on business laws and practices [23].

- *Autonomy:* autonomy refers to the degree of compliance of a partner to the global control rules. Partner systems may be autonomous in their design, communication, and execution. This means that individual partners select the process and content description models, programming models, interaction models with the outside world, etc. In a fully autonomous collaboration, each partner is viewed as a black box that is able to exchange information (i.e., send and receive messages). Partners interact via well-defined interfaces allowing them to have more local control over implementation and operation of services, and flexibility to change their processes without affecting each other. Usually, a completely autonomous collaboration may be difficult to achieve because it may require sophisticated translation facilities.

- *External Manageability:* this dimension refers to the degree of external visibility and manageability of partners' applications. In order to be effectively monitored by external partners, an application must be defined in a way that facilitates the supervision and control of its execution, measure-

ment of its performance, and prediction of its status and availability. For example, `AreaAgencyAging` may need to get the status (e.g., pending, approved) of the purchase order sent to `HealthDepartment`. This requires that `HealthDepartment` exposes sufficient information pertaining to measurements and control points to be used by `AreaAgencyAging`. While desirable in principle, high visibility may require complex descriptions of partners' applications. However, the overhead to provide such descriptions may be well justified if it provides other advantages such as *Quality of Service* (QoS).

- *Adaptability:* adaptability refers to the degree to which an application is able to quickly adapt to changes. Semantic Web applications operate in a highly dynamic environment where new services could come on-line, existing services might be removed, and the content and capabilities of services may be updated. Businesses must be able to respond rapidly to changes whereby both operational (e.g., server load) and market (e.g., changes of availability status, changes of user's requirements) environment are not predictable. For example, if `HealthDepartment` decides to stop its walk-in immunization activities, `AreaAgencyAging` would then need to adapt to such change. Changes may be initiated to adapt applications to actual business climate (e.g., economic, policy, or organizational changes). They may also be initiated to take advantage of new business opportunities. Since applications interact with both local back-end systems and partner applications, it is important to consider the impact of changes in both local and external applications to ensure local and global consistency. In general, the impact of changes depends on the degree of tightness among applications.

- *Security:* security is a major concern for inter-enterprise applications. Before Semantic Web applications reach their real potential, sophisticated security measures must be in place to boost partners' confidence that their transactions are safely handled [136]. For instance, `HealthDepartment` may need to check the authenticity of the purchase order before processing it. Semantic Web

applications must support mutual authentication, fine grain authentication, communication integrity, confidentiality, non-repudiation, and authorization. Interactions may be based on limited mutual trust, little or no prior knowledge of partners, and transient collaborative agreements. Shared information may include limited capabilities of services.

- *Scalability:* scalability refers to the ability of a system to grow in one or more dimensions such as the volume of accessible data, the number of transactions that can be supported in a given unit of time, and the number of relationships that can be supported. More importantly, changes in business climate are forcing organizations to merge to be effective in the global market. Thus, the cost and effort to support new relationships is an important criterion to consider when evaluating interaction solutions in the Semantic Web. Clearly, a low cost establishment of new relationships is desirable. However, in case of long-term relationships, the cost of establishing a new relationship is not of great significance.

2.3 Interactions in the Pre Semantic Web Era

Technologies for interactions on the pre-Web era have been around for almost three decades providing businesses, such as the banking industry, with a secure framework for sharing and exchanging data electronically. The most widely used and earliest framework is the *Electronic Data Interchange* (EDI) standard that runs on dedicated computer networks. Later, advances in software technology gave rise to a new breed of affordable software for distributed messaging and computing that can securely run on public computer networks: *component-based frameworks*. With corporate takeovers and consolidations coupled with the need of agile, just-in-time inter-enterprise cooperation on the Web, pressure mounted to provide solutions for enabling *inter-enterprise workflows*. Tomorrow's silver bullet applications such as *Virtual Enterprises* [11, 52, 53], will heavily draw on these solutions. In this section,

we describe major interaction frameworks developed in the pre Semantic Web era: EDI, components, and workflows.

2.3.1 Electronic Data Interchange (EDI)

EDI [92, 34] is commonly defined as the inter-organizational application-to-application transfer of business documents (e.g., purchase orders, invoices, shipping notices) between computers in a compact form. Its primary aim is to minimize the cost, effort, and time incurred by the paper-based transfer of business documents [1]. EDI documents are structured according to a standard (e.g., ANSI X12 [34] and UN/EDIFACT [92]) and machine-processable format.

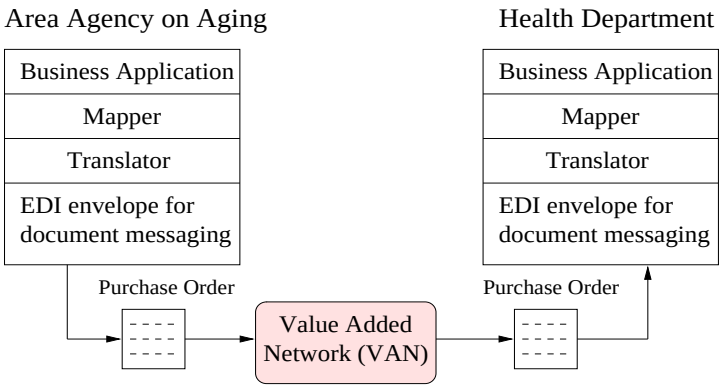

Figure 2.3: EDI-based Interactions

Figure 2.3 depicts two trading partners `AreaAgencyAging` and `HealthDepartment` exchanging business documents via a *Value-Added Network* (VAN). The document (e.g., purchase order) must be created in the business application of the sender (i.e., `AreaAgencyAging`). The *mapper* software is used to describe the relationship between the information elements in the application and the EDI standard. The EDI *translator* software converts the document into an EDI message according to the standard used. The translator wraps the

EDI message in an electronic envelope that has an identifier for the receiver (i.e., `HealthDepartment`). The actual transmission of the electronic envelope is performed by the *communication* software. This software maintains the trading partners' phone numbers to dial-up and exchange operations. The communication software can be a separate application or part of the translator. The VAN reads the identifier on the envelope and places it in the mailbox of `HealthDepartment`. At the `HealthDepartment` side, the reverse process takes place.

2.3.1.1 Interactions in EDI-based Solutions

EDI focuses mostly on interoperability at the communication and content layers. VANs are used to handle message delivery and routing among business partners. EDI standards provide a single homogeneous solution for content interoperability. They define a set of types for describing business documents. However, there is a limited (albeit large) number of predetermined documents supported by EDI standards. While these documents represent a large number of business transactions (e.g., shipping invoices, health care claim status reports), companies are limited to that set of EDI documents for which standards already exist [1]. It would be difficult for trading partners to conduct transactions whose parameters are not included in an EDI document. In that regard, EDI is hardly flexible in its ability to expand the set of supported document types. The introduction of a new type or changing an existing type of business transaction is complex and time consuming [1]. This kind of changes requires modification to the configuration of the translation software and must be validated in the related standard or EDI guideline committee which usually takes a long time [1]. For example the *EDI Guideline Consistency Subcommittee* (EGCS) is responsible for the content and maintenance of all TCIF (*Telecommunications Industry Forum*) EDI-maintained code lists [6]. Any modification to these code lists has to be reviewed by the EGCS. The EGCS is also responsible for notifying the *TCIF Secretariat* of any changes in the electronic documentation. Interoperability at the business process layers is supported through

pre-defined business processes. For example, if `AreaAgencyAging`'s purchase order is accepted then the `AreaAgencyAging` expects a purchase order acknowledgment, an invoice, and the immunization schedule.

The EDI approach is particularly strong along the criteria of security and heterogeneity. EDI is based on document exchange over private or value-added networks. Business partners do not concern themselves with those security issues encountered in public networks. Moreover, business partners do not need to directly reference each other systems. Therefore, critical security issues are bypassed. All partners are required to comply with the EDI standard. As a result, heterogeneity is not a problem. However, understanding all information in an EDI document is not a simple task. For example, there are data elements (UNH and UNT) in EDI document whose sole purpose is to indicate the start and end of a message. The impact of local changes is limited as partners do not directly reference each others' systems.

Although several EDI implementations have shown impressive results as set in the example of SEWP [91], the cost of establishing a new relationship usually requires a significant overhead. Because EDI is based on proprietary and expensive networks, organizations, predominantly small and medium, could not afford EDI. They were, *de facto*, excluded from being partners with larger organizations that mandate the use of EDI [1, 68]. Typically, VAN services entail three types of costs: account start-up costs, usage of variable costs, and VAN-to-VAN interconnect costs for the number of characters in each document [68]. The final cost of an EDI solution depends on several factors such as the expected volume of documents, economics of the EDI translation software, and implementation time. Maintenance fees and VAN charges can vary considerably and as such affect the cost of EDI systems. Some VAN providers do their billing on a per document basis. Others charge based on the number of characters in each documents [68]. It has been reported that 90% of the Fortune 500 companies in the United States uses EDI; only 6% of the other 10 million companies can make that claim [1]. Efforts to reduce the cost of using VAN networks include Internet-based EDI solutions such as EDIINT [66] and OBI [97].

Each EDI deployment involves negotiation and agreement on a set of implemen-

tation conventions describing the extensions to the standard documents and actual formats that would be exchanged. This negotiation and agreement process represents a significant cost in EDI deployment. To address this issue, EDIFACT and ANSI X.12 have undertaken an effort to standardize sets of documents for various industries. For example, ANSI X.12 has recently released a set of standard EDI document definitions for the health care industry. Using these industry standard document definitions, the customizations required per relationship can be reduced, although per-relationship work is generally still required. Additionally, once implementation conventions are decided upon, custom integration work must be performed at both partner organizations for the existing enterprise systems to process the EDI documents. This typically involves purchasing a commercial EDI system, integrating it with the enterprise systems, and writing custom code to translate the EDI system document definitions to the corresponding enterprise system records.

2.3.1.2 Internet-based EDI Initiatives

EDI has been extended in many directions. For instance, business documents in EDI standards have been mapped to XML documents (e.g., XML/EDI [35]). More specifically, the combination of EDI and Internet technologies seems to overcome several shortcomings of the traditional EDI (e.g., VAN charges). Indeed, several organizations are already using EDI for transacting over the Internet. For example, EDI purchase orders and invoices are now routinely exchanged via the Internet by NASA, Sun Microsystems, and Cisco systems. Major Internet-based EDI initiatives include EDIINT (*EDI over the Internet*) [66] and OBI (*Open Buying on the Internet*) [97].

EDIINT [66] – EDIINT is essentially the same as traditional EDI, but uses the Internet as a communication medium instead of VANs. The aim is mainly to reduce EDI communication charges due to the use of VANs. EDIINT was initiated by the *Uniform Code Council* (UCC) to standardize the method to exchange EDI

documents over the Internet. EDIINT is similar to EDI in terms of interoperability at the content and business process layers. At the communication layer, the first EDIINT standard (emerged in 2000) was EDIINT AS1 (*Applicability Statement 1*). EDIINT AS1 set the rules to exchange EDI documents using SMTP protocol. The second standard (completed in 2001) was EDIINT AS2 standard. It supported communication of EDI documents using the HTTP protocol.

Initially, there was reluctance to use the Internet for exchanging critical business information due to concerns about security. To deal with this problem, EDIINT AS2 specifies standard mechanisms for securing documents using PGP (*Pretty Good Privacy*) encryption and digital signatures [67]. The standards referenced by EDIINT AS2 include RFC1847 and MIME Security with PGP [67]. EDIINT offers lower entry cost than EDI since it is Internet-based. However, the quality of service (e.g., automatic error detection and correction) associated with VANs is lost. EDIINT offers similar characteristics as EDI with respect to the other dimensions (i.e, coupling, heterogeneity, autonomy, external manageability, and adaptability).

OBI [97] – OBI is a standard that leverages EDI to define an Internet-based procurement framework. It is clearly stated that OBI aims to complement EDI standards, not replace them. OBI is intended for high-volume, low-dollar amount transactions, which account for 80% of the purchasing activities in most organizations. At the communication level, OBI uses HTTP protocol for exchanging messages. OBI relies on the ANSI X12 EDI standard to describe the content of order documents. Order documents are encapsulated in OBI objects. OBI objects also encapsulate other non-EDI messages such as buyers' and sellers' digital signatures. OBI does not introduce a specific model for describing locally maintained information (e.g., product and price information). This information may be described in the partner's database. At the business process level, OBI defines a simple and pre-defined operational protocol for Internet-based purchasing. This protocol consists of a number of commonly agreed upon activities (e.g., select a supplier, create order) for purchasing non-strategic material (e.g., office supplies, laboratory supplies). In fact, this

protocol only specifies the way partner OBI systems interact. It is the responsibility of each partner to integrate its internal applications (catalogs, inventory and order management systems, etc) with OBI servers.

OBI makes a strong attempt to provide a robust security infrastructure. It uses the SSL (*Secure Sockets Layer*) [93] over HTTP for securing communications. It also uses digital signatures and digital certificates for ensuring messages authenticity and integrity. OBI rates higher than EDI with regard to the scalability and adaptability dimensions. First, the extensibility of order documents is not an important requirement. OBI targets simple and pre-defined purchasing transactions. Second, it offers lower entry cost as it is an Internet-based framework. OBI offers similar properties as EDI and EDIINT with regard to the other dimensions (i.e., coupling, heterogeneity, autonomy, and external manageability).

2.3.2 Software Components

Software components (simply, *components*) are program modules that can be independently developed and delivered [14, 121]. They may be newly developed or wrap existing functionalities provided by databases, legacy systems or packages. Although most of the fundamental ideas that define object technology are applicable to components, components are not necessarily created using object-oriented tools and languages [84, 58]. For example, components may be realized using a functional language, an assembly language, or any other programming language [121].

The development of component-based applications generally requires the interconnection of geographically distributed components. The availability of a *middleware* that provides more effective ways of programming is important to the development of distributed component-based applications. A *component middleware* is an infrastructure that supports the creation, deployment, and interactions among components [125]. Figure 2.4 depicts `AreaAgencyAging`'s and `HealthDepartment`'s applications assembled from components. Each component represents an independent unit of a business functionality such as *payment*, *purchasing*, *privacy*, and *security*.

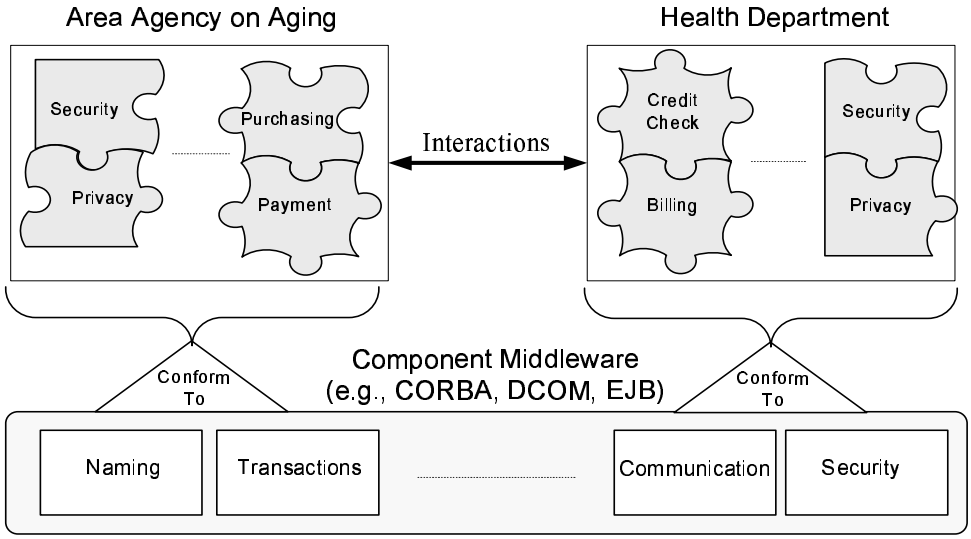

Figure 2.4: Component-based Interactions

The different components work together to serve the needs of `AreaAgencyAging`'s and `HealthDepartment`'s business processes. They are built on top of a set of basic services. Functions provided by these services include distributed communication, security, transactions, and naming schema.

Three major component middleware frameworks have been developed during the past decade:

- CORBA (*Common Object Request Broker Architecture*) [99]: CORBA is the standard promoted by the OMG (*Object Management Group*), an international industry consortium. It is part of a general architecture called the *Object Management Architecture* (OMA). The backbone of CORBA is the *Object Request Broker* (ORB) which allows communication between client and server components.

- DCOM (*Distributed Component Object Model*) [85]: DCOM is Microsoft's technology for distributed components. It is an extension of COM, the Mi-

crosoft's component software architecture. COM and its DCOM extension are merged into a single runtime which provides both local and remote access.

- EJB (*Enterprise Java Beans*) [109]: EJB is one of several technologies which make up Sun's J2EE (*Java 2 Platform, Enterprise Edition*) specification. It provides a component model for the Java programming language. In EJB, pieces of business logic or functions can be written in Java and encapsulated to become components known as *beans*. The *container* is the core of EJB component model. It provides a runtime environment that hosts and controls the beans.

The component-based approach is more appropriate for a small number of partners within an enterprise [30]. However, with companies being merged and acquired at the current rate, there is a need to address interactions within an enterprise. Components mainly cover interactions at the communication layer. They exhibit limited capabilities dealing with interactions at the content layer. They focus on the syntactic integration to wrap heterogeneous applications. At the business process layer, applications (e.g., scheduling an immunization campaign for `AreaAgencyAging`) may be assembled from independently developed components (e.g., payment, purchasing, privacy, security) [119, 51]. However, businesses generally would need to develop *ad hoc* solutions for defining intra and inter-enterprise business processes.

2.3.2.1 CORBA-based Interactions

At the communication layer, the use of ORBs in CORBA hides the underlying complexity of network communications from application developers. When a client issues a method invocation on a server component, the ORB intercepts the invocation and routes it across the network to the appropriate server. It is also possible that components distributed on different ORBs communicate over the Internet through the *Internet Inter-ORB Protocol* (IIOP).

CORBA provides a *trader service* through which businesses can find each other by assigning a set of properties to each component. However, these properties are

simply defined as (name,value) pairs. They do not provide support for semantic description of components. Recent efforts have been made to add semantic features to CORBA through the ECDTF (*Electronic Commerce Domain Task Force*) reference model which includes a *semantic data* facility [99]. However, the model is still at its very early stage. Additionally, very little work has been done so far to define a specification for the semantic data facility.

CORBA enable tightly coupled and long term business relationships between components. Once interfaces are expressed in IDL (*Interface Definition Language*), they are compiled by an IDL compiler into *stubs* and *skeletons*. The *stub*, used on the client side, invokes remote operations via the ORB to the corresponding skeleton on the server side. The *skeleton* gets the call parameters, invokes the actual operation implementation, collects results, and returns values back to the client through the ORB. Efforts are being made to add messaging capabilities to CORBA [30]. The new messaging specification defines a number of asynchronous and time-independent invocation modes for CORBA. It allows both static and dynamic invocations to use all modes. The use of message driven interactions among components allows the support of loosely coupled relationships. CORBA components are mostly based on static operation invocation. Although the *Dynamic Invocation Interface* (DII) in CORBA allows components to learn about other components' operations at run time, the utility of DII is yet to be exploited due to its complexity.

Components shield application developers from implementation details. Interfaces are the only considerations businesses must make when interacting with each other. Business partners have the latitude to implement their interfaces in ways that best fit their internal needs and requirements. Each CORBA component has an IDL that includes the name of the operations to be called by clients together with the name and types of all parameters and return values. However, all participants in a certain market need to agree on a predefined interface. This means that businesses are bound to interfaces published by their trading partners. In terms of heterogeneity, CORBA was designed to be independent of implementation languages, operating systems, and other factors that normally affect interactions. Components can be

implemented using diverse programming language such Java, C++, and Smalltalk.

External manageability is partially addressed in CORBA through the *event service*. The CORBA *event service* allows components to inform each other of the occurrence of specific events. It divides components into suppliers and consumers. Suppliers generate notifications of events while consumers register to be notified about the occurrence of events so that they can perform specific actions in response of those events.

CORBA provides little or no support for adaptability. As mentioned before, businesses are tightly bound to interfaces published by their trading partners. Hence, any change to a partner's interface may need the corresponding interface to be re-compiled. To date, CORBA does not provide mechanism to respond rapidly to changes in component interfaces.

Security is addressed in CORBA through the CORBA *security service*. This service provides a number of mechanisms such as authentication, authorization, and encryption of messages to build secure Web-based applications. Major CORBA vendors provide implementations of the security service.

The complexity of CORBA development increases the cost of entry in CORBA-based solutions. For example, developers in CORBA must generate binary code packages and deploy them on client sides when building new applications or when modifying the interfaces of existing applications. Although the dynamic invocation interface in CORBA alleviates this problem, programming calls with such interface is fairly complicated.

2.3.2.2 DCOM-based Interactions

Similarly to CORBA, DCOM-based solutions mainly deal with interactions at the communication layer. They present little or no support for interactions at the content and business process layers. For a DCOM client to access an operation of another component at the communication layer, it must use virtual lookup tables to obtain a pointer to that operation. The DCOM runtime environment ensures that

the pointer is local to the invoking process by using proxies [71].

DCOM components enable tightly coupled and long term business relationships. Proxies need to be created at the client side to communicate with stubs on the serving end [71]. The operation invocation process is static in DCOM which prevents establishing dynamic relationships among components. In terms of heterogeneity, current DCOM implementations are mostly based on Windows platforms although some experimentation have been done to port DCOM to other platforms (e.g., UNIX). Also, the languages that are mostly used to write DCOM components are Microsoft J++ (Microsoft's implementation of Java), C, C++, and Visual Basic. Additionally, DCOM's IDL is neither CORBA nor DCE (*Distributed Computing Environment*) compliant [71]. Security in DCOM relies on the Windows NT security model. Although this allows developers to build secure applications on Windows platforms, it is not clear how security will be provided when DCOM is used on other platforms. DCOM has similar characteristics as CORBA with respect to autonomy, external manageability, adaptability, and scalability.

2.3.2.3 EJB-based Interactions

At the communication layer, EJB uses the *Java RMI* [88] to enable interactions among beans. The use of RMI makes the location of the server transparent to the client. Similarly to CORBA and DCOM, EJB is fairly limited in terms of interactions at the content and business process layers.

Similarly to CORBA and DCOM, EJB caters for tightly coupled and long term business relationships. Developers must define an RMI remote interface for each bean. The RMI compiler generates a stub for each remote interface. The stub is installed on the client system and provides a local proxy for the client. The stub implements all the remote interfaces and transparently delegates all method calls across the network to the remote bean. A new specification of EJB (EJB Version 2) has recently been made available. It uses JMS (*Java Messaging Service*) to add support for message driven beans, extending the EJB component model to support

both tightly and loosely coupled applications [30]. Static operation invocation is found in most EJB implementations. However, some implementations such as *JBoss* integrate more dynamic features.

In EJB, each bean has a *remote* interface which defines the methods that carry out the business logic of the bean. The EJB remote interface provides functions that are similar to those provided by CORBA and DCOM IDL. Hence, EJB is similar to CORBA and DCOM in terms of autonomy. EJB does not support heterogeneous platforms although it is fully based on Java. Indeed, most of the current EJB implementations do not offer direct interoperability with non-Java platforms. In addition, communicating between components deployed on heterogeneous application servers, such as invoking a *BEA WebLogic* component from an *IBM WebSphere* server, requires operations in degraded mode.

Several implementations of an event service have also been provided for EJB to support external manageability. An example of EJB's event service is the *Drala Event Broker* [46]. EJB provides some support for adaptability by associating a *deployment descriptor* to each bean. The descriptor describes the way in which a bean interacts with its environment. Application developers declaratively define contracts in their descriptors. This contract describes the type of services (such as the form of transaction management to be used) required by the bean. It can be changed independently of the business logic.

The EJB container provides security features to EJB components. Each deployment descriptor contains declarations about the access control for the corresponding enterprise bean. When a client calls an operation of that bean, the container is responsible for checking that the requester has the right to invoke that operation by accessing an access control list. Finally, EJB offers similar properties as CORBA and DCOM with respect to scalability.

2.3.3 Workflows

Workflow management is concerned with the declarative definition, enactment, ad-

ministration and monitoring of *business processes*. A *business process* (or workflow process) consists of a collection of activities related by data and control flow relationships (Figure 2.5). An activity is typically performed by executing a program, enacting a human/machine action, or invoking another process (called sub-process). Programs, persons, machines, and data used to perform workflow processes are called *workflow resources*. For example, `AreaAgencyAging`'s business process includes several activities such as issuing a purchase request, approving it, and issuing a purchase order. The information sent from `AreaAgencyAging`'s *Issue_Purchase_Request* activity to the *Approval_Process* activity includes the number of citizens eligible for immunization service. The scripting of activities and resource policies through *business process analysis, modeling, and definition tools* defines a *business process definition* (workflow schema) [39]. The *workflow enactment service* enables different parts of the business process to be enacted by providing interfaces to users, applications, and databases distributed across the workflow domain (Figure 2.5).

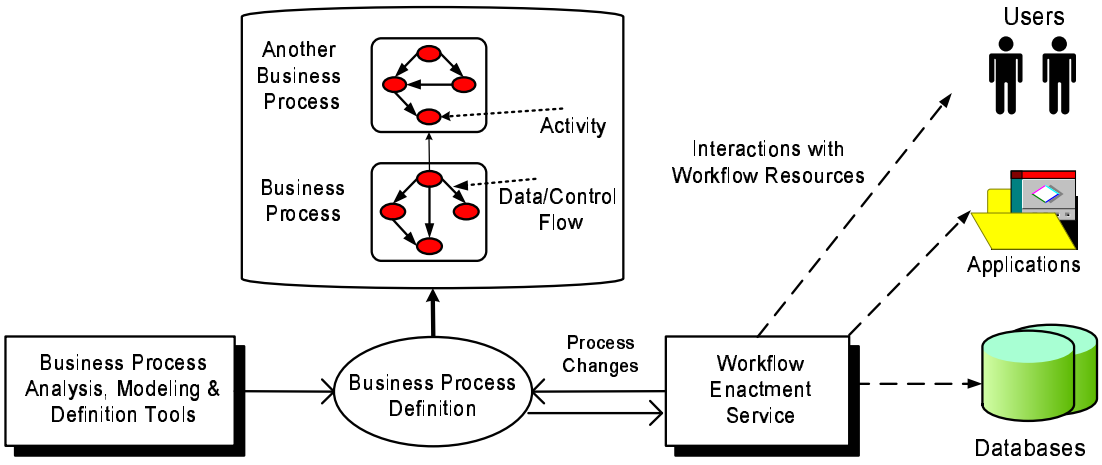

Figure 2.5: Workflow System Characteristics

Workflow is a key technology for automating business processes that involve access to several applications. However, traditional workflow systems are ineffective

when we consider the needs of Web-based applications, with their complex partnerships, possibly among a large number of highly evolving processes. Current efforts (e.g., the Business Process Initiative - BPMI.org) promise to deliver a next generation workflow systems (*Inter-Enterprise Workflow Systems* - IEWSs) that have the ability to thread together cross-organizational business processes, supporting the integration of diverse users, applications, and systems [136]. IEWSs focus mainly on interactions at the business process layer. Their purpose is to automate business processes that interconnect and manage communication among disparate systems.

2.3.3.1 Distributed Workflow Systems

The emphasis in *Distributed Workflow Systems* (DWSs) is on partitioning the overall workflow specification into several sub-workflows, each encompassing all the activities that are to be executed by a given entity within an organization [90]. DWSs impose that each organization participating in a distributed workflow deploy a full-fledged execution engine, capable of interpreting the workflow definition. The same workflow model must be adopted by each participant in the global workflow. This approach assumes that global and sub-business processes use the same process definition and data exchange model. This is a quite restrictive assumption in the context of Web-based applications where: (i) partners may use disparate data and process representation models (e.g., `AreaAgencyAging` uses EDI and `HealthDepartment` uses RosettaNet), and (ii) private business processes may require access to proprietary/legacy data sources and applications (e.g., Oracle database for `AreaAgencyAging` and SAP application for `HealthDepartment`). In addition, DWSs assume a tight coupling model among the distributed sub-workflows. Thus, modifications to back-end applications, sub-workflows, and global workflow need to be coordinated. The cost of establishing a new relationship may be significant as business processes must be modeled and deployed in concert across all participants. DWSs are appropriate for the development of a business process of a single organization that needs to integrate multiple distributed sub-workflows.

2.3.3.2 Collaborative Process Management

Inter-enterprise business processes management features the separation between *public* and *private processes* [20, 39]. A *public process* defines an external message exchange of an organization with its partners according to a message exchange protocol such as EDI and RosettaNet. A *private process* describes internal executable activities that support the activities of public processes. Public and private processes interact through *process wrappers*. Process wrappers consist of pre-defined activities that can be used in a private business process to send/receive messages to/from public business processes. For example, if a public process uses *xCBL* [32] to represent business documents, and the private business process expects documents in *cXML* [38], the conversion between these two formats is handled by a wrapper. Private processes may also interact with back-end applications through *application adapters*. In this approach there is no requirement that local process management engines (e.g., engines which are responsible for managing private business processes) be identical. It is possible for example, that one engine is based on IBM's *MQSeries* [63] and another based on HP's *Process Manager* [60].

The separation between back-end applications, public, and private processes has the advantage that local changes (i.e, those that concern only private processes) have no impact on public processes and back-end applications. However, changes related to interactions (e.g., changing the formats of incoming and outgoing messages) between a public process (or a back-end application) and a private business process may require the modification of some wrappers. The separation between components of a Web-based application (public processes, private processes, business rules, and back-end systems) contributes to the scalability of this approach. The support of a new interaction protocol (e.g., EDI) requires only the creation of a new public process and process wrappers. The support of new a back-end application requires the creation of new application adapters. The creation of a new relationship with a new partner may require a few adjustments. If the new partner does not comply to an already supported interaction protocol, a new public process must be created

to support the protocol used by the new partner. The support of a new back-end application requires only the creation of a new application adapter. The above discussion shows that the addition of interaction protocols, back-end applications, or partners does not require the modification of private business processes.

The separation between public and private business process provides for a greater degree of autonomy and bridging of heterogeneity. With regard to security, IEWs may leverage techniques used in other frameworks (i.e, document-based or component-based frameworks). External manageability can be provided by adding specific activities in public processes.

2.4 Trends in Supporting Semantic Web Interactions

The traditional approaches for Web-based interactions were not devised for Semantic Web. They provide little or no support for capturing the semantics of messages exchanged among partners or partners' business processes. Therefore, recent research has focused on developing concepts and techniques for enabling interactions on the Semantic Web. In this section, we overview major Semantic Web technologies, namely, ontologies, Web services, agents, and XML-based interaction standards.

2.4.1 Ontologies

The Web is evolving from a set of single isolated application systems into a World wide network of disparate systems interacting with each other via information exchange. This requires means to represent the semantics of the exchanged information so that it could be automatically understood. This is where *ontologies* would play a crucial role, providing a machine processable semantics of the information communicated between heterogeneous systems.

Ontologies were initially developed in Artificial Intelligence to facilitate knowledge sharing and reuse. They have since then been recognized as a popular research

topic in various research communities such as knowledge engineering, e-commerce, natural language processing, cooperative information systems, and information integration. In this section, we define the concept of ontology and give the motivations behind it. We then overview major languages and standards for describing ontologies on the Web. Finally, we evaluate ontology-based systems using interaction layers and dimensions.

2.4.1.1 Modeling Semantics Through Ontologies

An *ontology* is a *formal* and *explicit* specification of a *shared conceptualization* [40, 100]. "Conceptualization" refers to an abstraction of a domain that identifies the relevant concepts in that domain. "Shared" means that an ontology captures *consensual* knowledge. The development of ontologies is often a cooperative process involving different entities possibly at different locations (e.g., businesses, government agencies). All entities that agree on using a given ontology commit themselves to the concepts and definitions within that ontology. "Explicit" means that the concepts used in an ontology and the constraints on their use are explicitly defined. "Formal" intends that the ontology should be machine understandable and described using a well-defined model or language called *ontology language*.

An ontology typically consists of a hierarchical description of important concepts in a domain, along with descriptions of the properties of each concepts. Formally, an ontology Ω contains a set of *concepts* (also called *classes*) $\{c_1,...,c_n\}$ which constitutes the core of the ontology. The notion of concept in ontologies is similar to the notion of class in object-oriented programming. Each concept c_i has a set of *properties* $P_i = \{p_{i1},...p_{im}\}$ associated to it. This set describes the different features of the class. Each property p_{ik} has a *range* R_{ik} (also called *type*) indicating a restriction on the values p_{ik} can take. An ontology relates classes to each other through *ontology relationships*. Examples of relationships include "subclassof", "superclassof". Properties are also related through similar relationships such as "subpropertyof" and "suprepropertyof".

We identify three different types of ontologies depending on their generality level: *vertical*, *horizontal*, and *metadata* ontologies. Other types of ontologies such as *representational*, *method* and *task* ontologies also exist but are out of the scope of our research [48]. *Vertical* ontologies capture the knowledge valid for a particular domain such as medical, mechanic, chemistry, and electronic. *Horizontal* ontologies describe general knowledge that is valid across several domains. They define basic notions and concepts (e.g., time, space) applicable in many technical domains. *Metadata* ontologies provide concepts that allow the description of other concepts. For example, *Dublin Core* is an ontology for describing the content of on-line information source.

2.4.1.2 Web Ontology Languages

Several ontology-based approaches have been developed for Web applications during the past few years. A common claim for all these approaches is the need for models and languages to specify ontologies. An obvious solution would be to use XML for describing ontologies. However, XML provides a syntax to encode data. It does not specify the data's use and semantics. The parties that use XML for their data exchange must agree beforehand on the vocabulary, its use, and meaning. As consequence, many Web ontology languages have been developed. In this section, we overview a representative set of these languages: RDF, RDF Schema, and DAML+OIL. An exhaustive list of ontology languages for the Semantic Web is presented in [54].

RDF and RDF Schema: RDF (*Resource Description Framework*) provides a domain-neutral framework to model metadata about *resources* on the Web. It is recommended by *World Wide Web Consortium* (W3C) working groups. A *resource* is any "thing" on the Web that has a *Uniform Resource Identifiers* (URI) such as Web pages and files. Besides resources, RDF's basic model includes *properties* and *statements*. A *property* is a characteristic, attribute, or relation that describes a re-

source. A *statement* consists of a specific resource with a named property plus that property's value for that resource. This value can be another resource or a literal value (free text). An RDF description is a list of triples: *(resource, property, value)*. It is syntactically represented in XML. Below is an example of triple represented in XML. It states that a specific Web page (resource) was created (property) by "John" (values). We use XML namespaces "rdf" and "dc" to identify elements. Simply put, XML namespaces provide a method for qualifying element and attribute names used in XML documents by associating them with URI references:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/DC/">
 <rdf:Description about="http://www.w3.org/">
   <dc:Creator> John </dc:Creator>
 </rdf:Description>
</rdf:RDF>
```

RDF does not provide mechanisms for defining relationships between properties and resources. This is the role of *RDF Schema*. RDF Schema is a type system for RDF. It provides a mechanism to define domain-specific properties and classes of resources to which we can apply those properties. Basic primitives of RDF include *class* and *subclass* which together allow the definition of class hierarchy, *property* and *subproperty* to build property hierarchies, and *domain* and *range* to restrict the possible combinations of properties and classes.

DAML+OIL: DAML+OIL is the result of a merger between DAML-ONT, a language developed as part of the US DARPA Agent Markup Language (DAML) and OIL (the Ontology Inference Layer), developed by a group of (mostly) European researchers. It builds on earlier Web ontology standards such as RDF and RDF Schema and extends those languages with richer modeling primitives (e.g., cardinality). DAML+OIL adopts an object oriented approach, describing ontologies in terms of classes, properties, and axioms (e.g., subsumption relationships between

classes or properties) [59]. When a resource R is an instance of a class C, we say that R has type C.

From a formal point of view, DAML+OIL can be seen to be equivalent to description logic (DL), with a DAML+OIL ontology corresponding to a DL terminology (Tbox). As in a DL, DAML+OIL classes can be names (URIs) or expressions, and a variety of constructors are provided for building class expressions. Example of DAML+OIL constructors include *intersectionOf*, *unionOf*, *complementOf*, *oneOf*, *minCardinalityQ*, *maxCardinalityQ*. The meaning of the first three constructors (intersectionOf, unionOf and complementOf) is relatively self-explanatory: they are the standard boolean operators that allow classes to be formed from the intersection, union and negation of other classes. The oneOf constructor allows classes to be defined existentially, i.e., by enumerating their members. The minCardinalityQ, maxCardinalityQ, cardinalityQ constructors are known in DLs as qualified number restrictions. The minCardinalityQ (maxCardinalityQ, cardinalityQ) constructor gives the class whose instances are related via a given property to at least (at most, exactly) n different resources of type C. The number n and class C are specified in the constructor. The following DAML+OIL example defines two classes "HomePage" and "Person". These classes are related by the property "belongsTo" with a cardinality 1. This means that each home page belongs to one person:

```
<daml:Class rdf:ID="HomePage"> </daml:Class>
<daml:Class rdf:ID="Person"> </daml:Class>
<daml:ObjectProperty rdf:ID="belongsTo">
    <daml:domain rdf:resource="#HomePage"/>
    <daml:range rdf:resource="#Person"/>
</daml:ObjectProperty>
<daml:Class rdf:about="#HomePage">
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
       <daml:onProperty rdf:resource="#belongsTo"/>
```

```
        </daml:Restriction>
      </rdfs:subClassOf>
    </daml:Class>
```

2.4.1.3 Interactions in Ontology-based Systems

Ontologies are mostly developed to support interactions at the content layer. At the communication layer, they may rely on existing communication protocol (e.g., HTTP). Ontologies may be used, at the business process layer, to provide a set of pre-defined business documents to be exchanged among partners. Each document would contain the actions that should take place during message exchange.

Participants in an ontology-based system are bound to the concepts and properties defined within the ontology. This may impact their autonomy and heterogeneity. Ontologies are often not stable definitions. Indeed, a shared conceptualization of a domain has to be reached in a process of social communication. Changes in the domain generally trigger the modification of the ontology. The evolution of ontologies causes operability problems, which hamper their effective reuse. Solutions are still required to allow changes to ontologies without making current use invalid. The cost of entry in an ontology-based system may be significant since participants must describe their applications according to a given ontology. Finally, the issues of external manageability and security are not addressed in ontology-based systems.

2.4.2 Web Services

The precise definition of Web services is still evolving as witnessed by the various definitions in the literature. One such definition is that a Web service is a "business function made available via the Internet by a service provider, and accessible by clients that could be human users or software applications" [28]. It is also defined as "loosely coupled applications using open, cross-platform standards and which interoperate across organizational and trust boundaries" [124]. The W3C (*World Wide Web Consortium*) defines a Web service as a "software application identified

by a URI (*Uniform Resource Identifier*), whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols". The aforementioned definitions can be seen as complementary. Each definition emphasizes some part of the Web service characteristics (discovery, invocation, etc). In this section, we define Web services as business functionalities that are:

- *Programmatically accessible*: Web services are mainly designed to be invoked by other Web services and applications. They are distributed over the Web and accessible via widely deployed protocols such as HTTP and SMTP. Web services must describe their capabilities to other services including their operations, input and output messages, and the way they can be invoked.
- *Loosely coupled*: Communication among Web services is *document-based*. Web services generally communicate with each other by exchanging XML documents. The use of a document-based communication model caters for loosely coupled relationships among Web services.

2.4.2.1 The Web Service Reference Model

Interactions among Web services involve three types of participants: *service providers*, *service registry*, and *service consumers* (Figure 2.6). *Service providers* are the parties that offer services. They define descriptions of their services and publish them in the *service registry*, a searchable repository of service descriptions. Each description contains details about the corresponding service such as its data types, operations, and network location. *Service consumers* use a *find* operation to locate services of interest. The registry returns the description of each relevant service. The consumer uses this description (e.g., network location) to invoke the corresponding Web service. For example, `HealthDepartment` may advertise a Web service that includes a `scheduleImmunization` operation. `AreaAgencyAging`'s

provider would then access the registry, discover `HealthDepartment`'s service, and invoke its operations.

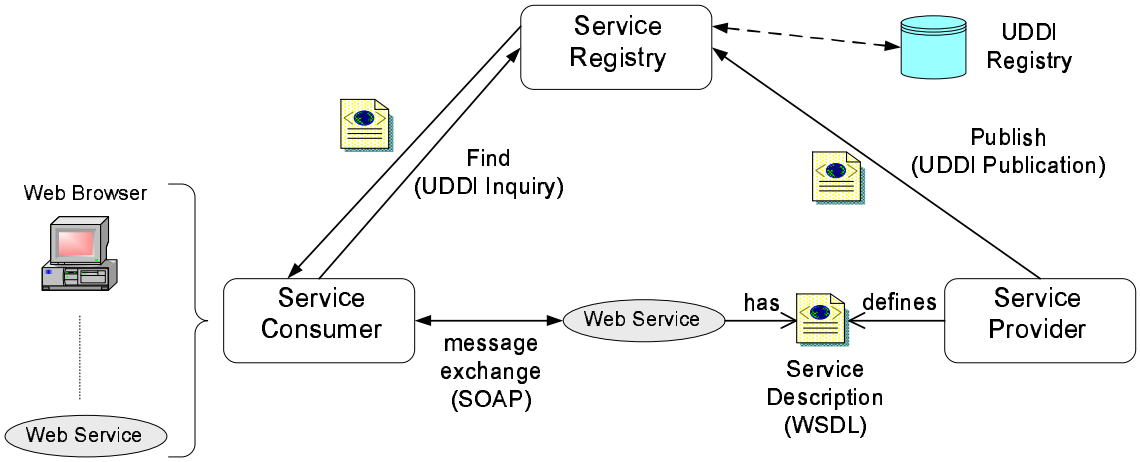

Figure 2.6: The Web Service Reference Model

Three major standardization initiatives have been submitted to the W3C consortium to support interactions among Web services (Figure 2.6):

- WSDL (*Web Services Description Language*) [131]: WSDL is an XML-based language for describing operational features of Web services. WSDL descriptions are composed of *interface* and *implementation* definitions. The *interface* is an abstract and reusable service definition that can be referenced by multiple *implementations*. The *implementation* describes how the interface is implemented by a given service provider.

- *UDDI* (*Universal Description, Discovery and Integration*) [130]: UDDI defines a programmatic interface for publishing (*publication API*) and discovering (*inquiry API*) Web services [130]. The core component of UDDI is the *business registry*, an XML repository where businesses advertise services so that other businesses can find them. Conceptually, the information provided in a UDDI

business registration consists of *white pages* (contact information), *yellow pages* (industrial categorization), and *green pages* (technical information about services).

- SOAP (*Simple Object Access Protocol*) [129]: SOAP is a lightweight messaging framework for exchanging XML formatted data among Web services. SOAP can be used with a variety of transport protocols such as HTTP, SMTP, and FTP. A SOAP message has a very simple structure: an XML element (called *envelope*) with two child elements. The first element, the *header* includes features such as security and transactions. The second element, the *body* includes the actual exchanged data.

2.4.2.2 Interactions in Web Services

Web services allow interactions at the communication layer by using SOAP as a messaging protocol. The adoption of an XML-based messaging over well-established protocols (e.g., HTTP, SMTP, and FTP) enables communication among heterogeneous systems. For example, major existing environments are able to communicate via HTTP and parse XML documents. However, SOAP protocol is still at its early stage; current implementation do not yet meet the reliability and workload constraints needed in enterprises.

At the content layer, Web services use WSDL language. WSDL recommends the use of XML Schema as a canonical type system (to associate data types to message parameters). However, the current version of WSDL does not model semantic features of Web services. For example, no constructs are defined to describe document types (e.g., whether an operation is a request for quotation or a purchase order). Recent efforts towards dealing with semantic interoperability include the development of content markup languages such as DAML-S [74]. However, such efforts are still in their infancy.

Web services are still at a maturing stage. Hence, they still lack the support for interactions at the business process layer. To date, enabling interactions among

Web services has largely been an *ad hoc* process involving repetitive low level programming. Standardization efforts such as BPEL4WS (*Business Process Execution Language for Web Services*) [9] are underway for enabling the definition of business processes through Web service composition.

The use of a document-based messaging model in Web services caters for loosely coupled relationships. Additionally, Web services are not statically bound to each other. New partners with relevant features can be discovered and invoked. However to date, dynamic discovery of Web services takes place mostly at development time. Heterogeneous applications (e.g., Java, CORBA objects) may be wrapped and exposed as Web services. For example, the *Axis's Java2WSDL* utility in *IBM's Web Services Toolkit* enables the generation of WSDL descriptions from Java class files. *IONA*'s *Orbix E2A Web Services Integration Platform* may be used to create Web services from existing EJBs or CORBA objects. In terms of autonomy, Web services are accessible through published interfaces. Partners interact with Web services without having to be aware of what is happening behind the scene. They are not required to know how the operations provided by the service are internally implemented. Some operations can even be transparently outsourced from third parties.

WSDL does not currently include operations for monitoring Web services such as checking the availability of an operation or the status of a submitted request. Additionally, neither UDDI nor WSDL currently define quality of service parameters such as cost and time. In terms of adaptability, changes may occur in operation signatures (e.g., name), messages (e.g., number of parameters, data types), service access (e.g., port address), and service and operation availability. The process of dealing with changes is currently *ad hoc* and manually performed. More efforts need to be done to cater for automatic detection and handling of changes.

Security in Web services needs to be addressed at different levels including communication, description, and firewall. At the communication level, enabling security in XML and HTTP is an important factor towards securing Web services. Current standardization efforts include securing XML-based messages through the creation

of an XML digital signature standard and S-HTTP, a protocol for transmitting data securely over the Web. Other work is also being done to extend SOAP to include a security specification at the receiving endpoints (e.g., specify which users are authorized to receive and process messages). At the service description level, WSDL does not include security-oriented information such as role-based access control and other authorization information. Finally, since SOAP messages carried over HTTP traverse firewalls, network administrators would need to configure their firewalls so that malicious requests (e.g., code embedded in SOAP messages) are not tunneled through SOAP messages. For example, *application firewalls*, sitting behind networks firewalls, have been developed (e.g., *iSecureWeb*). Application firewalls check all HTTP traffic to validate and authorize messages based on security policies.

The emergence of tools to describe, advertise, and invoke Web services facilitates the development of Web service-based solutions. However, the use of a tagged language such as XML increases the volume of information to be exchanged among Web services. This might overload the network in presence of a large number of services, hence penalizing the scalability of the Web service approach. Additionally, SOAP defines only simple data types. Using complex data types may require the XML parser to get the corresponding XML Schema definitions from remote locations. This might add an overhead for processing SOAP messages. The registry presents another scalability issue. A centralized registry might result in a single point of failure and bottleneck for accessing and publishing Web services. A distributed registry would cater for a more reliable and scalable solution. However, this incurs an additional overhead of managing distributed repositories. An intermediary solution is adopted in UDDI where the registry is physically replicated over multiple nodes. This solution solves the problem of centralized registry. However, it still requires the nodes to exchange data with each other to maintain registry consistency.

2.4.3 Software Agents

The concept of *software agent* (or simply *agent*) emerged as an important paradigm for organizing many classes of distributed applications such as e-commerce, digital libraries, information retrieval, and data mining. It is defined as a piece of software capable of acting exactingly to accomplish tasks on behalf of its consumers. Consumers may be end-users, other agents, or applications using the agent. A more comprehensive definition would be to view an agent as an umbrella term which covers a range of properties. Agents typically possess several (or all) of the properties summarized in Table 2.3. Ideally, an agent should exhibit all the properties equally well, but this is an aspiration rather than the reality [94]. Other properties such as *emotion* and *character* (knowledge, belief, intention, etc.) have also been defined [135]. They are mainly used in artificial intelligence applications and so, are out of the scope of our research.

Property	*Synonyms*	*Definition*
Continuous	Not Available	An agent is a continuously running process
Autonomous	Not Available	Agents operate without the direct intervention of humans or others and exercise control over their own actions
Co-operative	Communicative Socially able	Agents interact with each other via some kind of agent-communication language
Reactive	Sensing and acting	Agents perceive their environment and respond in a timely fashion to changes that occur in it
Adaptive	Learning	Agents change their behavior based on their previous experience
Pro-active	Goal-oriented Purposeful	Agents do not simply act in response to their environment, they are able to take initiatives to achieve a certain goal
Mobile	Not Available	Agents are able to transport themselves from one site (in a network) to another

Table 2.3: Agent Properties

2.4.3.1 Agent Communication Language

A multi-agent system is a collaborating set of agents in which the membership may change over time. Individual agents in the community view each other as peers and are responsible for providing specific services according to their role. The processing of a request in such an environment is distributed among the different

agents. The success of a multi-agent system relies on how the agents interact and interoperate. This involves the ability of agents to communicate with each other via an *Agent communication language* (ACL). An ACL consists of three major parts: a *vocabulary*, an inner language called *KIF* (*Knowledge Interchange Format*) and an outer language called *KQML* (*Knowledge Query and Manipulation Language*). An ACL message is a KQML expression in which the arguments are terms in KIF formed from words in the ACL vocabulary.

Agents should share a common understanding of what is being communicated. Ontologies facilitate agent communication by providing a common vocabulary. The vocabulary is an open-ended dictionary or list of words appropriate to common application areas. Each word has an English description for use by humans and formal annotations (written in KIF) for use by programs. The dictionary allows for the addition of new words within both existing and new application areas. KIF is a prefix version of first order predicate calculus, with various extensions to enhance its expressiveness. It provides for the encoding of simple data, constraints, negations, and quantified expressions. A KQML message is a piece of dialogue between a sender and a receiver. Each message is a list of participants enclosed in matching parentheses. The first word in the list indicates the type of dialogue (ask, tell, recommend, bid, etc.). The subsequent entries are the arguments, that is KIF expressions appropriate to that dialogue.

Below is an example of KQML message where a sender (Mary) is asking the receiver (Bookstore) to send information regarding all books related to "Agents" which costs below $30.

```
(ask
 :sender Mary
 :receiver Bookstore
 :content "select * from book where topic = ‘‘Agents’’ and price <30"
 :language   "Oracle SQL"
```

```
 :ontology   "online bookstore"
)
```

2.4.3.2 Interactions in Multi-Agent Systems

Multi-agent systems cater for interactions at the communication layer by having a standard communication language, namely ACL. This assumes that the underlying communication and transport protocols are already established. At the content layer, ontologies are used to facilitate communication between agents and add semantics to their messages. Multi-agent systems lack support for specifying complex business processes. A process is generally spread through the implementation code of every agent involved in the process. This *ad hoc* definition of business processes does not scale because of the size of partners that may be involved in business processes. One solution would be to have a separate *control* agent that handles the business process. However, the control agent's design is time consuming. Additionally, the business logic is embedded in the control agent, making it hard to reuse for different business processes [106].

Multi-agent systems assume that relationships between different partners have to be pre-established. Additionally, data to be interchanged among agents generally needs to be manually or semi-automatically mapped to a common ontology. This makes agents suitable for long-term and tightly coupled relationships. Agents may be added to existing applications to add new functions or customize the execution of existing functions with little impact on the autonomy of those applications. However, they must adhere to the concepts and properties of a common ontology which may limit their autonomy.

Agents may wrap proprietary and heterogeneous applications. The adoption of an ontology enables the abstraction from heterogeneous information representations. Event subscriptions and monitoring agents may be used to address the issue of external manageability. However, adaptability is not adequately addressed in multi-agent systems. Agents are typically pre-programmed with certain function-

alities and can hardly be configured to switch roles or change interaction patterns. Agents interact to carry out sophisticated tasks on behalf of users. In the course of this interaction, they may exchange sensitive information about these users. More research is needed to ensure that multi-agent systems support the security features required in Semantic Web interactions. The cost of entry in an agent-based system is significant. Agents must be able to exchange information via Agent Communication Language. They also need to formulate queries or requests based on the common ontology. They finally must have a way of matching queries/requests with their known ontologies (i.e. reason over its repository of ontologies).

2.4.4 XML-based Interaction Standards

A large number of contemporary interaction standards are based on XML. The vision behind this approach is to allow the use of services on the Web without dedicated transformation and mediation facilities or custom integration of partners' systems. Business partners would form a *trading community* based on their capabilities to produce and consume those XML documents. The business process of the trading community is specified by the shared document definitions. The partners are interconnected in terms of largely agreed upon documents. The business logic implementation at a partner side is invisible to other trading partners. In general, a complete XML-based integration requires standardized domain-specific ontologies (such as an agreed upon DTD or XML Schema), mappings between different ontology descriptions, and means for processing XML documents and invoking appropriate services (e.g., workflows and legacy systems) to handle requests.

Most of the existing XML-based interaction standards were developed for B2B interactions. However, they may be extended to deal with other types of applications such as e-government and digital library. In what follows, we describe a representative set of those standards (Figure 2.7). The proposed standards generally deal with enabling interactions at the content (e.g., *eCO*, *cXML*) and business process (e.g., *RosettaNet*) layers. However they sometimes overlap or even compete

with each other [20]. The issue of interoperability has thus shifted from the level of applications to the level of standards. A trading partner has to deal with several standards at the same time. In case one trading partner exchanges messages across industries, the variety of standards is likely to increase even more [20].

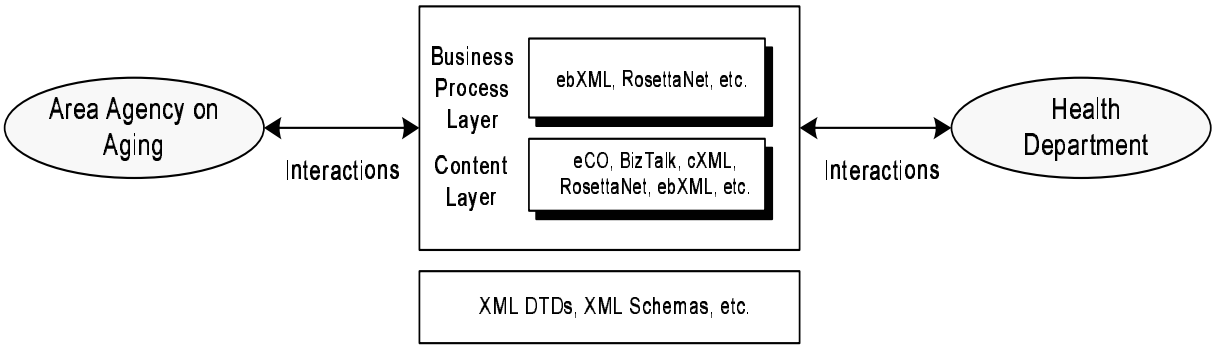

Figure 2.7: XML-based Interaction Standards

2.4.4.1 eCO

eCO [32] aims at providing means to businesses to discover and access services regardless of the standards and protocols each potential partner adopts. At the content level, eCO introduces xCBL (*XML Common Business Library*) to define business documents. xCBL consists of a set of XML *core documents* that are used to represent common interactions in business transactions. It does not target vertical industry domains. It attempts to provide a generic framework for describing the content of core business documents. The main motivation for establishing core documents is that some concepts are common to all business domains and thus can be expressed in a common format. Examples of such core documents are: *purchase orders*, *invoices*, *date*, *time*, and *currencies*. Business partners may use and extend these documents (e.g., adding new elements) to develop their own business documents. For example, `HealthDepartment` can use xCBL to create an XML

document of its service catalog by customizing the generic *xCBL catalog DTD* with specific information about the search method (e.g., immunization services by type). Businesses are not limited to a specific set of pre-defined documents. However, this may hamper interoperability since companies would need to be aware of newly created documents.

At the business process level, eCO focuses more on providing a common building blocks for interactions among businesses. Businesses can advertise their online services as *Business Interface Definitions (BIDs)*. BIDs are XML descriptions that specify business services in terms of documents they accept and produce. It does not mandate a global business process definition. eCO uses xCBL as a basis to define both the interfaces of processes and content of data elements. Since every partner is forced to use the same tag to define the same type of information, the structural heterogeneity is not a problem. As is any standard, there is however a non-trivial issue: the meaning and types of services and their interfaces can vary among businesses although a group of partners in a specific marketplace may select to adopt common conventions. In generic frameworks such as eCO, it is difficult to address semantic heterogeneity because of the wide range of E-commerce applications. One solution is to use several schemas (or *ontologies*) including horizontal (i.e., across domains such as computer manufacturing and healthcare) and vertical (i.e., within a specific domain) domains. This solution requires the support of data normalization, mapping and conversion between schemas or ontologies.

Although, eCO requires that services be described using XML schemas, it does address, albeit in a limited way, the issue of autonomy. eCO separates the description of services and their implementations. Note that a marketplace may adopt some common conventions for describing services. This may as a result, negatively impact on the partners' autonomy. For example, a change in the name of a tag, requires all partners to make that specific change at the same time. In eCO, the use of security mechanisms is optional. However, business partners do not need to directly gain access to each other systems. The establishment of a new relationship with an existing partner does not require any additional work. The creation of a new service

requires the provision of its description (types, interfaces, etc). It also requires the integration of the interfaces of the service with internal applications. The integration cost in an XML-based approach tends to be less significant than other approaches because of widely available XML processing tools. The eCO framework rates high in adaptability. The impact of local changes is limited as partner systems are loosely coupled. In addition, eCO offers extensibility to accommodate changes.

2.4.4.2 cXML

cXML (*Commerce XML*) [38] consists of an XML-based schema language and a protocol for online purchasing transactions. It targets business transactions that involve non-production Maintenance, Repair, and Operating (MRO) goods and services. In a nutshell, cXML can be considered as a simplified XML and Internet-based version of EDI. cXML assumes the existence of intermediaries (*E-commerce hubs*) that act as trusted third parties between procurement systems and supplier systems. The functions provided by an E-commerce hub (e.g., Ariba Network, Extricity Software) are similar to those provided by the BizTalk repository. However, cXML does not prescribe a specific intermediary architecture.

cXML supports two communication models: *request-response* and *one-way*. The request-response provides for synchronous communication through HTTP. The one-way provides for asynchronous communication through HTTP or other protocols. Currently, the one-way model supports HTTP and URL Form Encoding.

At the content level, cXML defines a set of XML DTDs to describe procurement documents in the same spirit as xCBL (e.g., order request, order response). It provides the following elements for describing product catalogs: *Supplier*, *Index*, and *Contract*. The supplier element describes general information about a supplier (e.g., address, ordering methods). The index element describes the supplier's inventory (e.g., product description, part numbers, classification codes). The contract element describes the negotiation agreements between a buyer and a supplier on product attributes (e.g., price, quantity). Catalogs can be static or dynamic. In the cXML

terminology a dynamic catalog is called a *punchout*.

At the business process level, the cXML approach is similar to OBI's (see Section 2.3.1.2). cXML defines a generic procurement protocol. This protocol consists of a number of commonly agreed upon online procurement activities (e.g., product selection, order request, order approval, order transmission, order routing). E-commerce hubs provide means for catalog and purchase order management (e.g, catalog publishing and subscription, automated purchase order routing and tracking).

cXML offers similar properties to those in OBI, namely, heterogeneity, autonomy, and adaptability. cXML appears to rate higher than OBI with regard to scalability because the integration cost in an XML-based approach tends to be less significant than other approaches. cXML addresses security by including authentication information message headers. One advantage of cXML approach is economy of scale and ease of managing business relationships. Both suppliers and buyers only need to manage relationships with the trusted intermediary rather than with all their business partners.

2.4.4.3 RosettaNet

RosettaNet [33] aims at standardizing product descriptions and business processes in information technology supply chain applications. RosettaNet's supply chain include information technology products (e.g., boards, systems, peripherals, finished systems) and electronic components (e.g., chips, connectors). RosettaNet focuses on three key areas of standardization to automate Web-based interactions. First, the vocabulary needs to be aligned. The *RosettaNet Business Dictionary* contains vocabulary that can be used to describe business properties (e.g., business name, address, tax identifier). The *RosettaNet Technical Dictionary* contains properties that can be used to describe characteristics of products (e.g., computer parts) and services (e.g., purchase order). Second, the way in which business messages are wrapped and transported must be specified. The *RosettaNet Implementation Framework* specifies content of messages, transport protocols (HTTP, CGI, email, SSL) for communica-

tion and common security mechanism (digital certificates, digital signatures). Third, the business process governing the interchange of the business messages themselves must be harmonized and specified. *RosettaNet*'s PIPs (*Partner Interface Processes*) are pre-defined XML-based *conversations*. A *conversation* consists of a set of business documents (e.g., purchase order, purchase order acknowledgment) and message exchange logic (e.g., the sequencing of the actions that take place during a product quote request). A PIP is defined using a combination of textual and graphical (UML-based state machine) representations.

At the communication layer, common Internet transport protocols are supported. At the content layer, RosettaNet uses an XML-based schema as document content model. The use of a vertical ontology (i.e, common vocabulary with information technology supply chain domain) contributes to solving the problem of semantic heterogeneity. At the business process layer, RosettaNet focuses on providing a common basis for public interactions via PIPs. The integration of PIPs with internal business processes is performed by partners. RosettaNet does not provide means to define arbitrary global business processes. RosettaNet offers similar properties as OBI with regards to security. It offers similar properties as eCO with regard to autonomy, adaptability, scalability, coupling, and external manageability.

2.5 Summary and Discussion

In this section, we compare the different approaches to interactions on the pre-Semantic Web and Semantic Web eras. We discuss the commonalities and differences between Web services and related technologies such as components, agents, and workflows. Finally, we illustrate the role of Web services on the Semantic Web.

2.5.1 Comparison of Semantic Web Interaction Technologies

We identify two sets of parameters that together exhaustively define how applications interact on the Web. The first set (applicable to enabling technologies and prototypes) consists of the following parameters: communication layer, content layer, and business process layer. The second set (applicable to enabling technologies and prototypes) consists of the following parameters: coupling, autonomy, heterogeneity, external manageability, adaptability, security, and scalability.

In Table 2.4, enabling technologies technologies for the pre Semantic Web era are compared using the most important architectural layers. For example, communications in EDI are conducted through VANs. The content layer is supported through ANSI X12 and EDIFACT formatted documents. Interactions at the business process layer are limited to pre-defined business processes. In Table 2.5, key Semantic Web technologies are compared using the most important architectural layers. For example, *Web Services*' communication layer is typically provided by SOAP. The content layer is supported by using WSDL language. However, WSDL currently provides little support for semantic description of business documents. One of the current trends to support semantic interoperability is the use of ontologies (e.g., DAML-S). WSFL, XLANG, and BPEL4WS languages provide support for interactions at the business process layer. However, these languages are still at their early stage.

In Table 2.6, enabling technologies For the pre Semantic Web era are now compared using salient interaction parameters (dimensions). For example, *EJB-based* B2B's coupling is tight and long term. Autonomy is provided by separating the interface and the implementation using the EJB remote interface. Heterogeneity is supported at the platform levels (e.g., Unix and Windows) but only Java is supported. The use of event services provides support for external manageability. Adaptability is partially addressed through the separation between contracts and business logic. Security is provided through the EJB container. Scalability is not much of a concern for intra-enterprise applications. In Table 2.7, Semantic Web interaction

	Communication Layer	*Content Layer*	*Business Process Layer*
EDI	VANs	ANSI X12 and EDIFACT formatted documents	Pre-defined business processes
EDIINT	SMTP (for EDIINT AS1) and HTTP (for EDIINT AS2)	ANSI X12 and EDIFACT formatted documents	Pre-defined business processes
OBI	HTTP	ANSI X12 formatted documents	Pre-defined protocol for Internet procurement
CORBA	ORBs and IIOP	Not Addressed	Ad hoc: hand-coded programming of the integration logic
DCOM	DCOM runtime environment	Not Addressed	Ad hoc: hand-coded programming of the integration logic
EJB	RMI/JMS	Not Addressed	Ad hoc: hand-coded programming of the integration logic
Workflows	Not Addressed	Not Addressed	Inter-enterprise business processes (public and private)

Table 2.4: Pre Semantic Web Technologies vs. Interaction Layers

	Communication Layer	*Content Layer*	*Business Process Layer*
Ontologies	Not Addressed	Shared Conceptualization	Not Addressed
Web Services	SOAP	WSDL but little support for semantic description. Use of ontologies for semantic interoperability	WSFL, XLANG, and BPEL4WS
Agents	Agent Communication Language	Common Ontology	Not Addressed
eCO	HTTP	xCBL	Not Addressed
cXML	HTTP and URL form encoding	XML DTDs	Pre-defined protocol for Internet procurement
RosettaNet	HTTP, E-mail, etc.	RosettaNet Business Dictionary and RosettaNet Technical Dictionary	Partner Interface Processes (PIPs). Pre-defined protocol for Internet procurement

Table 2.5: Semantic Technologies vs. Interaction Layers

technologies are compared using the same parameters. For example, Web services support both transient and long-term as well as loose relationships. The separation between WSDL interface and implementation definition caters for the autonomy of participants. WSDL descriptions allow the wrapping of heterogeneous applications. The issues of external manageability and adaptability are currently not addressed. Research efforts are still on-going to provide security mechanisms for Web services.

2.5.2 Web Services and Related Technologies

Web services share some features with the component, agent, and workflow-based approaches. However, several characteristics differentiate Web services from components, agents, and workflows.

Web Services and Components – Web services use document-based communication. This in in contrast with component-based frameworks which use object-based communication, thereby yielding systems where the coupling between components is tight [127]. Additionally, by using HTTP as a communication protocol, Web services enables much more firewall-friendly computing than component-based systems. For example there is no standard port for IIOP, so it normally does not traverse firewalls easily. Although a specification has been submitted to OMG to dealing with IIOP firewall traversal, ORB implementations are still using their own proprietary solutions such as *VisiBroker's GateKeeper* and *IONA's WonderWall*.

Web Services and Agents – As in Web services, agent-based systems provide "directories" where an agent advertise their distinct functionalities and other agents search to locate it [61]. However, agents are inherently communicative, whereas Web services are passive until invoked. Web services use XML formatted message over ubiquitous protocols such as HTTP to interact with each other. However, agents use ACL language to format their messages. As discussed in Section 2.4, agents are loosely coupled while agents are tightly coupled. All these reasons make Web service suitable for enabling cross-organizational interactions. Agents

generally provide alerts and updates when new events become available.

Web Services and Workflows – Service composition shares some features with workflows. For example, to define a composite service, the provider needs to specify the flow of service invocations (i.e., services to be invoked, their input and output data, and their execution dependencies). Similarly, in a workflow, the designer must specify the flow of work (i.e., work items to be invoked, their input and output data, and their execution dependencies). However, several features distinguish between composite services and workflows. We list below the main differences:

- Distributed workflow systems require that each participating enterprise deploys a full-fledged execution engine, capable of interpreting the workflow definition. The same workflow model must be adopted by all the participants in the global workflow. They assume a tight coupling among the distributed sub-workflows. Modifications to sub-workflows need to be coordinated. Hence, distributed workflow systems are appropriate for the development of business processes within a single enterprise.

- In workflows, input and output data are typically specified by a set of variable names [27]. The values of the input variables are passed to a selected resource at the time an activity is started. Activity execution results are inserted into the output variables. Communications between the workflow system and resources is done through *adapters* which deal with data mapping issues. However, the development of adaptors is a difficult and tedious task [27]. In contrast, Web services interact by exchanging messages, thereby avoiding the need for adapters.

- A number of XML-based interaction standards (e.g., RosettaNet, cXML) are being defined to support B2B interactions. Many applications that support such standards are being or have been developed. It is likely that composite services will interact with services that follow one of these standards [27].

However, many workflow systems do not support such standards, although some vendors are moving in this direction [27].

2.5.3 The Role of Web Services in the Semantic Web Landscape

The current technologies for Web-based interactions may be viewed as complementary. In Figure 2.8, we summarize these technologies and the way they co–exist in a Semantic Web interactions framework. Component middleware, workflows, Web services, and agents define the building blocks for developing Semantic Web applications. Ontologies, EDI, OBI, and XML-based standards (eCO, RosettaNet, etc.) focus on defining the semantics of interactions among businesses. Component middleware (CORBA, DCOM, and EJB) are suitable for building robust and secure applications *within* an enterprise. Intra-enterprise business processes are managed using systems such as ERPs (e.g., *SAP/R3*) and workflows (e.g., IBM's *MQSeries*). Inter-enterprise workflows mostly focus on interactions at the business process layer. Their aim is to automate business processes that span the boundaries of disparate enterprises.

Web services take components a step further by enabling inter-enterprise interactions. They define the entry points to enterprises' internal systems. Web services may wrap intra-enterprise components to provide connectivity between autonomous and heterogeneous inter-enterprise applications. They may also wrap other applications developed, for example, in Java or Visual Basic. Agents may be used add new functions to Web services (e.g., change monitoring) or customize existing functions based on operating conditions or observations on users' behavior. Web services may use a workflow-based approach or service composition techniques and standards (e.g., BPEL4WS) to support inter-enterprise business processes. Web services adopt ontologies and XML-based standards to capture the semantics of documents and business processes. For example, Web services may use *RosettaNet* or *cXML* to carry out interactions among businesses according to these frameworks. *EDI*

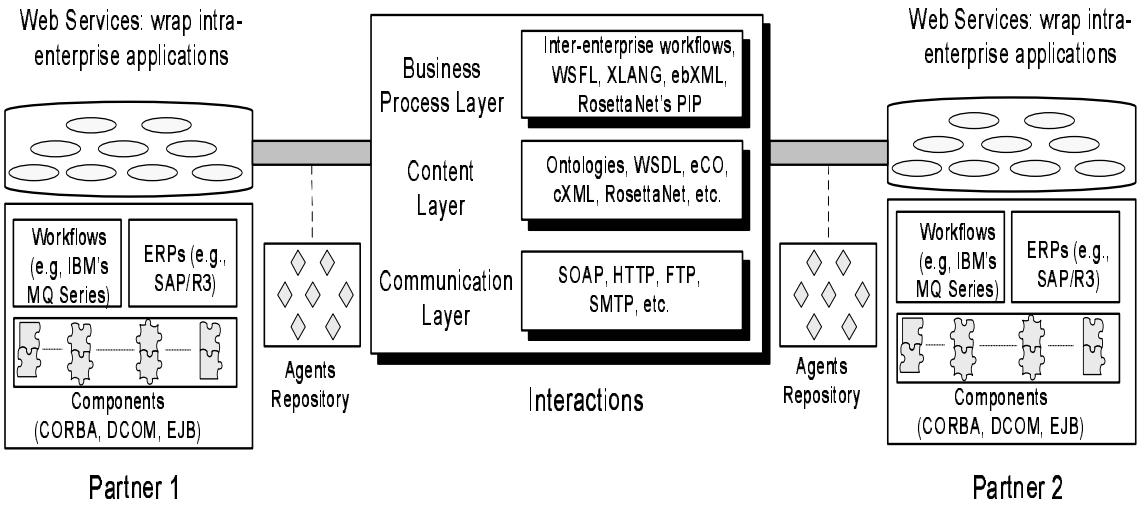

Figure 2.8: Technologies in a Semantic Web Interaction Framework

and *OBI* standards may also be used to enable interactions at the content layer. However, they provide little support in terms of expanding the set of supported document types.

	Coupling	*Autonomy*	*Heterogeneity*	*Ext. Manag.*	*Adaptability*	*Security*	*Scalability*
EDI	Loose and long term	New documents must be approved by EDI guideline committees	Support of heterogeneous applications thanks to translator software	Not Addressed	Impact of local changes limited. New documents must be approved	Private VANs	Expensive networks. Need to agree on implementation conventions
EDIINT	Loose and long term	New documents must be approved by EDI guideline committees	Support of heterogeneous applications thanks to translator software	Not Addressed	Impact of local changes limited. New documents must be approved	RFC 1847, MIME security with PGP, etc.	Lower entry-cost than EDI (Internet-based)
OBI	Loose and long term	Higher than EDI (document extensibility is not an important requirement)	Support of heterogeneous applications thanks to translator software	Not Addressed	Higher than EDI (document extensibility is not an important requirement)	SSL/HTTP, digital signatures, and digital credentials	Lower entry-cost than EDI (Internet-based)
CORBA	Tight and long term	Separation between interface and implementation (IDL)	Different languages (e.g., Java, C++) and platforms (Unix, Windows)	Event service	Not Addressed	Supported by CORBA security service	Suitable for intra-enterprise applications. Participants need to have a stub for each component server
DCOM	Tight and long term	Separation between interface and implementation (IDL)	Different languages (e.g., Microsoft J++, C++) but Windows platform	Event service	Not Addressed	Based on Windows NT security model	Suitable for intra-enterprise applications. Proxies needed at client side
EJB	Tight and long term	Separation between interface and implementation (EJB remote interface)	Java language and different platforms (Unix, windows)	Event service	Contracts can be changed independently of the business logic	Security features provided in EJB container	Suitable for intra-enterprise applications. Participants must define a remote interface for each bean
Workflows	System-specific	Separation between public and private business processes	Different workflow engines	Adding appl-specific states	Process wrappers and adapters help localize changes	Not Addressed	May require creation of new process wrappers and adapters

Table 2.6: Pre Semantic Web Technologies vs. Interaction Dimensions

	Coupling	*Autonomy*	*Heterogeneity*	*Ext. Manag.*	*Adaptability*	*Security*	*Scalability*
Ontologies	Loose and long-term	Participants bound to concepts defined within the ontology	Participants bound to concepts defined within the ontology	Not Addressed	Not Addressed	Not Addressed	Participants need to describe their applications according to a given ontology
Web Services	Loose, transient, and long-term	Separation between WSDL interface and implementation definitions	WSDL descriptions to wrap underlying applications	Not Addressed	Not Addressed	On-going efforts (e.g., XML digital signatures, S-HTTP)	Availability of development tools. XML tags increase the volume of information. Replicated registries must exchange data to maintain coherence
Agents	Tight and long-term	Participants must adhere to concepts defined within a common ontology	Agents may be used to wrap heterogeneous applications	Events and monitoring agents	Not Addressed	Not Addressed	Agents need to formulate requests based on the common ontology. They must have a way of matching requests with their known ontologies
eCO	Loose, transient, and long term	Separation between description and implementation. Marketplaces may hinder autonomy	XML Schemas. Marketplaces may help addressing semantic heterogeneity.	Not Addressed	Impact of local changes limited. Support of extensibility documents	Optional	Establishment of a relationship with a partner does not require additional work from this partner
cXML	Loose, transient, and long term	Higher than EDI because document extensibility is not an important requirement	XML DTDs	Not Addressed	Impact of local changes limited. Extensibility of documents is not required	Authentication information. Trusted intermediaries	Cost of entry is lower than in OBI thanks to the use of XML
Rosetta Net	Loose and long term	Separation between description and implementation. Marketplaces may hinder autonomy	RosettaNet business and technical dictionaries. Use of vertical technology	Not Addressed	Impact of local changes limited. Support of extensibility documents	SSL/HTTP. Digital certificates and signatures	Establishment of a relationship with a partner does not require additional work from this partner

Table 2.7: Semantic Web Technologies vs. Interaction Dimensions

Chapter 3

An Ontological Framework for Web Services

The semantic organization and description of Web services is an important requirement for enabling the automatic composition of Web services. The number of Web services available on the Web is large and continuously changing. Additionally, service providers may be located in different places over the world. Finally, the large scale, dynamics, and heterogeneity of Web services may hinder any attempt for "understanding" their semantics and hence outsourcing them. This calls for techniques to organize Web services in a way they can be efficiently "understood" and outsourced.

In this chapter, we propose a Semantic Web centered framework for organizing and describing semantic Web services [76, 78, 81, 11, 4]. We introduce the concept of *community* to cater for an ontological organization and description of Web services. We develop an ontology, called *community ontology*, that serves as a "template" for describing communities and semantic Web services. We also propose a peer-to-peer approach for managing communities in highly dynamic environments. In particular, we present techniques for registering Web services with communities and coping with changes that occur in the Web service space.

This chapter is organized as follows. In Section 3.1, we give an overview of the

proposed model for semantic Web services. In Section 3.2, we describe operational features of communities via *generic operations*. In Section 3.3, we present the technique used for registering Web services with a community. In Section 3.4, we propose algorithms for managing communities. We focus on dealing with changes issued by community and service providers.

3.1 The Proposed Model for Semantic Web Services

Ontologies are poised to play a central role to empower Web services with semantics. They are increasingly seen as key to enabling semantics-driven data access and processing. In this section, we give an overview of the proposed model for organizing and describing Web services on the Semantic Web.

3.1.1 Ontological Support for Web Services

Combining Web services and ontologies is at the core of the proposed model. Central to our model is the concept of *community*. A *community* is a "container" that clumps together Web services related to a specific area of interest (e.g., disability, adoption). All Web services that belong to a given community share the same area of interest. Communities provide descriptions of desired services (e.g., providing interfaces for insurance services) without referring to any actual service.

Communities are defined using a *metadata* ontology, called *community ontology*, as a template. As defined in Section 2.4.1, metadata ontologies provide concepts that allow the description of other concepts (communities and Web services in our case). We summarize in Figure 3.1 the process of creating a community and registering Web services with it. Communities are defined by *community providers* as instances of the community ontology. Community providers are generally groups of government agencies, non-profit organizations, and businesses that share a common domain of interest. For example, the Department for the Aging and other related agencies, such

as the Department of Health, would define a community that provides healthcare benefits for senior citizens.

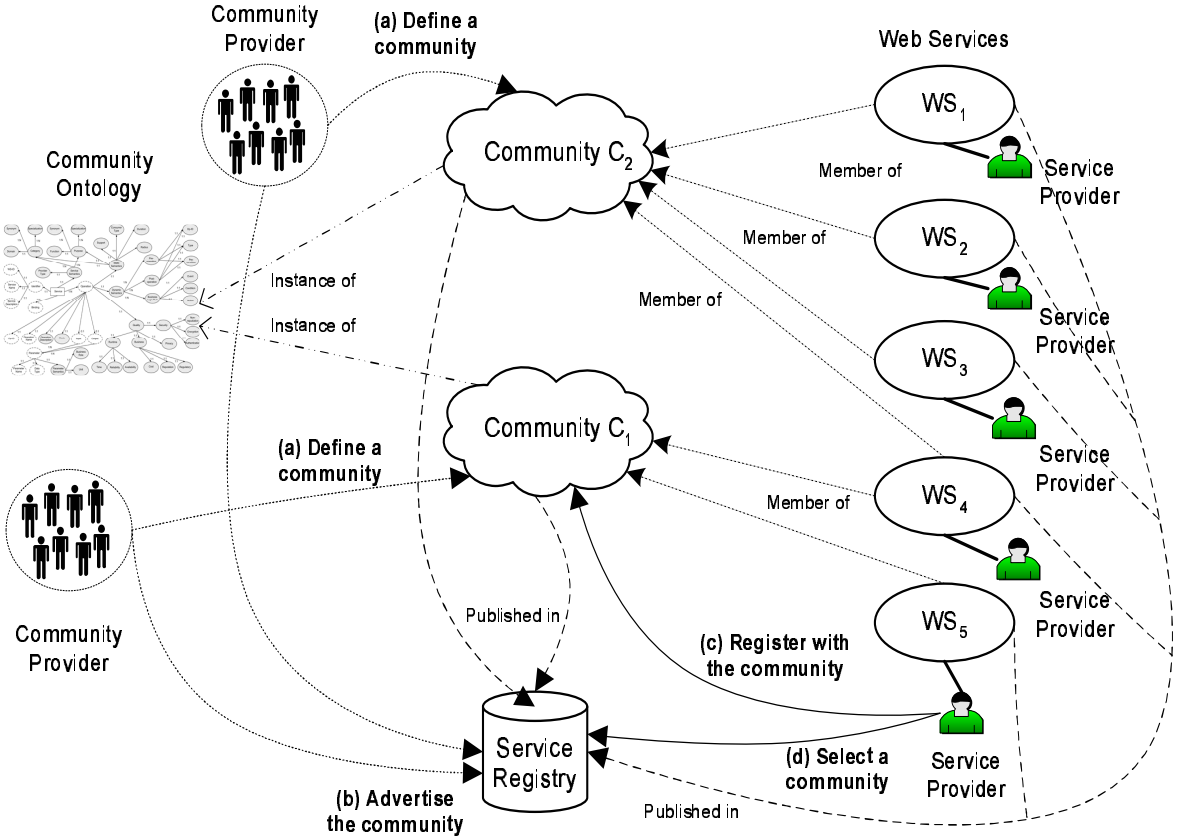

Figure 3.1: The Proposed Web Service Model

A community is itself a service that is created, advertised, discovered, and invoked in the same way "regular" Web services are. The providers of a community assign values to the concepts of the community ontology (Figure 3.1 - step a). Examples of concepts defined in the community include *category* and *generic operation*. Each concept is defined by a set of attributes. For example, the *category* concept contains four attributes: *domain*, *synonyms*, *specialization*, and *overlapping*. Communities are published in a registry (e.g., UDDI) so that they can be discovered by service providers (Figure 3.1 - step b). *Service providers* (e.g., `medicare` provider)

identify the community of interest (Figure 3.1 - step c) and register their services with it (Figure 3.1 - step d). During the registration of a service *WS* with a community C_i, the service provider specifies the concepts of C_i that are "inherited" by *WS*. For example, *WS* may "inherit" only some of the generic operations defined in C_i. A Web service may belong to different communities. For example, a composite service (WS_4 in Figure 3.1) may outsource operations that have different domains of interest (e.g., "healthcare" and "elderly"). Since these operations belong to two different communities, the composite service is registered with the "healthcare" and "elderly" communities (C_1 and C_2 in Figure 3.1). End-users (e.g., case officers) select a community of interest and invoke its operations. Each invocation of a community operation is translated into the invocation of a community member's operation.

We use the emerging DAML+OIL language for describing the proposed ontology [59]. It is noteworthy that the proposed model is not dependent on DAML+OIL. Other Web ontology standards could be used to describe the community ontology. We give in Figure 3.2 a subset of the community ontology specified in DAML+OIL. The first four elements define the *Community*, *Identifier*, *Category*, and *Generic-Operation* classes. The three following elements define relationships (or properties) between *Community* and the other classes. The last element specifies cardinality constraints on the different properties. For example, the first constraint indicates that a community has one single identifier.

3.1.2 Structure of a Community

A community C_i is formally defined by a tuple *(Identifier$_i$, Category$_i$, G-operation$_i$, Members$_i$)*. The *identifier$_i$* clause contains a unique *name* and a text *description* that summarizes C_i's features (Figure 3.3). *Category$_i$* describes the area of interest of the community. All Web services that belong to C_i have the same category as C_i's. C_i is accessible via a set of operations called *generic operations*. Those are specified in the *G-operation$_i$* clause. Generic operations are "abstract" operations that summarize the major functions needed by C_i's members. Community providers define generic

```
<daml:Class rdf:ID="Community"> </daml:Class>
<daml:Class rdf:ID="Identifier"> </daml:Class>
<daml:Class rdf:ID="Category"> </daml:Class>
<daml:Class rdf:ID="GenericOperation"> </daml:Class>
<daml:ObjectProperty rdf:ID="isIdentifiedBy">
    <daml:domain rdf:resource="#Community"/>
    <daml:range rdf:resource="#Identifier"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="hasCategory">
    <daml:domain rdf:resource="#Community"/>
    <daml:range rdf:resource="#Category"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="hasGenericOperation">
    <daml:domain rdf:resource="#Community"/>
    <daml:range rdf:resource="#GenericOperation"/>
</daml:ObjectProperty>
<daml:Class rdf:about="#Community">
    <rdfs:subClassOf>
      <daml:Restriction daml:cardinality="1">
        <daml:onProperty rdf:resource="#isIdentifiedBy"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:Restriction daml:cardinality="1">
        <daml:onProperty rdf:resource="#hasCategory"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:Restriction daml:mincardinality="1">
        <daml:onProperty rdf:resource="#hasGenericOperation"/>
      </daml:Restriction>
    </rdfs:subClassOf>
</daml:Class>
```

Figure 3.2: Subset of the DAML+OIL Specification for the Community Ontology

operations based on their expertise on the corresponding area of interest that is, C_i's

category. The term "abstract" means that no implementation is provided for generic operations. Community providers only define an interface for each generic operation op_{ik}. This interface could subsequently be used and implemented by community members (i.e., actual Web services) interested in offering op_{ik}. We say that those members *support* or *import* op_{ik}. The execution of op_{ik} hence refers to the execution of an actual operation offered by a member that support op_{ik}. The $Members_i$ clause refers to the list of C_i's members. By being members of C_i, Web service providers "promise" that they will be *supporting* one or several of C_i's generic operations. In this section, we focus on describing the category clause and community ontology. Details about generic operations and community members are given in Sections 3.2 and 3.3 respectively.

The *category* of a community C_i is formally defined by a tuple *(*$Domain_i$*,* $Synonyms_i$*,* $Specialization_i$*,* $Overlapping_i$*)*. $Domain_i$ gives the area of interest of the community (e.g., "healthcare"). It takes its value from a taxonomy for domain names. For flexibility purposes, different communities may adopt different taxonomies to specify their category. We use XML namespaces to prefix categories with the taxonomy in which they are defined. Simply put, XML namespaces provide a method for qualifying element and attribute names used in XML documents by associating them with URI references. $Synonyms_i$ contains a set of alternative domain names for C_i. For example "medical" is a synonym of "healthcare". Values assigned to this attribute are taken from the same taxonomy as the one used for domains. $Specialization_i$ is a set of characteristics of the C_i's domain. For example, "insurance" and "children" are specialization of "healthcare". This means that C_i provides health insurance services for children. Communities are generally not independent. They are linked to each other via *inter-ontology* relationships. These relationships are specified in the $Overlapping_i$ attribute. $Overlapping_i$ contains the list of categories that *overlap* with C_i's category. It is used to provide a peer-to-peer topology for connecting communities with "related" categories. We say that $category_i$ *overlaps with* $category_j$ if composing C_i's operations with C_j's is "meaningful". By meaningful, we mean that the composition provides a *value-added* (in

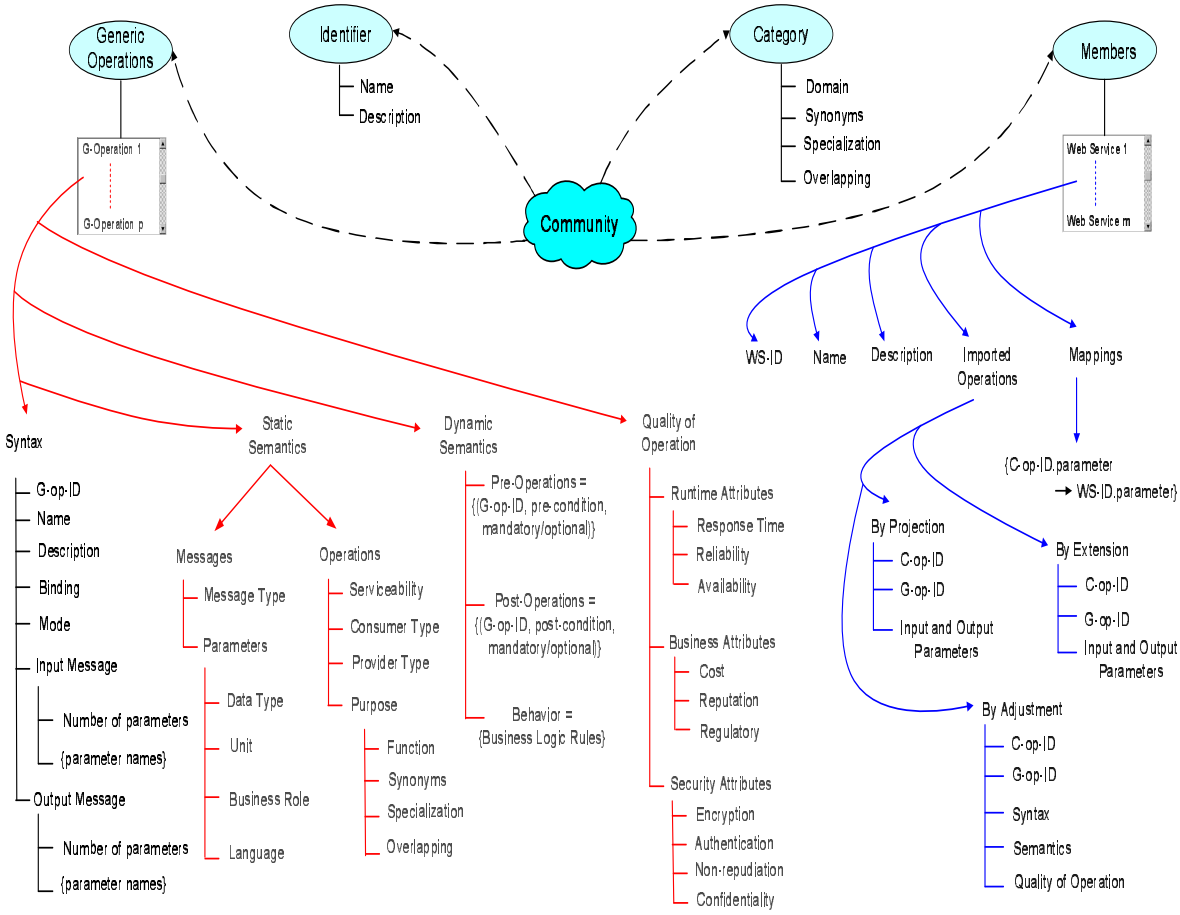

Figure 3.3: The General Structure of a Community

terms of categories). For example, an operation that belongs to a community whose domain is *family* may be composed with another operation that belong to a community whose domain is *insurance*. This would enable providing health insurance for needy families. It should be noted that it is the responsibility of the community providers to identify related categories and assign them to the overlapping attribute.

3.1.3 Generic Operations

A generic operation is defined by a set of *functional* and *non-functional* attributes. *Functional* attributes describe *syntactic* and *semantic* features of generic operations. *Syntactic* attributes represent the structure of a generic operation. An example of syntactic attribute is the list of input and output parameters that define the operation's messages. *Semantic* attributes refer to the meaning of the operation or its messages. We consider two types of semantic attributes: *static* and *dynamic* semantic attributes. *Static* semantic attributes (or simply static attributes) describe non-computational features of generic operations. Those are semantic attributes that are generally independent of the execution of the operation. An example of static attribute is the operation's category. *Dynamic* semantic attributes (or simply dynamic attributes) describe computational features of generic operations. They generally refer to the way and constraints under which the operation is executed. An example of dynamic attribute is the business logic of the operation i.e., the results returned by the operation given certain parameters and conditions.

Non-functional attributes, also called *qualitative* attributes, include a set of metrics that measure the quality of the operation. Examples of such attributes include time, availability, security, and cost. Two service providers that support the same generic operation may have different values for their qualitative attributes. Non-functional attributes model in fact the competitive advantage that competitors (i.e., Web services that support the same generic operation) may have on each other.

While defining a community, community providers assign values to part of the attributes of their generic operations. The rest of the attributes are assigned either by service providers or third parties during the registration of Web services with C_i. Table 3.1 summarizes the different attributes and mentions the parties that are responsible of assigning their values (using the $\surd$ symbol). For example, the types of input and output messages (e.g., purchase order, registration confirmation) are defined by community providers. The cost (dollar amount) of executing an operation is service-specific and hence defined by the service provider. The other

Group	Sub-Group	Attribute	Community Provider	Service Provider	Third Party
Syntax	Message	Input	√	√	
		Output	√	√	
		Number of param.	√	√	
	Operation	G-op-ID	√		
		Name	√		
		Description	√		
		Mode	√		
		Binding		√	
Static Semantics	Message	Message Type	√		
		Business Role	√		
		Unit	√	√	
		Language	√	√	
	Operation	Serviceability	√		
		Consumer Type	√		
		Provider Type	√		
		Purpose	√		
		Category	√		
Dynamic Semantics	Inter-operation	Pre-operations	√	√	
		Post-operations	√	√	
	Intra-operation	Behavior	√	√	
Quality of Operation	Run-time	Response Time			√
		Reliability			√
		Availability			√
	Business	Cost		√	
		Reputation			√
		Regulatory			√
	Security/Privacy	Encryption			√
		Authentication			√
		Non-repudiation			√
		Confidentiality			√

Table 3.1: Assigning Values to Generic Operation Attributes

qualitative attributes (e.g., response time, availability) are assigned by third parties (e.g., trusted parties, monitoring agencies). The way those parties determine the values to be assigned (e.g., through monitoring) is out of the scope of our research. It is worth noting that the values of some attributes may be assigned by both community and service providers. For example, the content of an input and output message is given by community providers. However, service providers may modify this content by adding and/or removing parameters to input and output messages.

3.1.4 Community Members

Service providers can, at any time, select a community of interest (based on categories) and register their services with it. We say that those services are members of that community. The registration process requires giving an *identifier* (*WS-ID*), *name*, and *description* for the Web service. The *identifier* takes the form of a unique UUID. The *description* summarizes the main features of the Web service. Service providers specify the list of generic operations supported by their services through the *imported* attribute. We define three constructs for importing generic operations: *projection*, *extension*, and *adjustment*. The *projection* and *extension* constructs allow the addition and deletion of message parameters respectively. *Adjustment* enables the modification of the content of operation attributes. The invocation of an imported operation is translated into the invocation of an "actual" service operation. The correspondence between imported and "actual" operations is done through the *mapping* attribute. For each imported operation, the provider gives the ID of the corresponding "actual" operation. It also defines a one-to-one mapping between the imported operation's parameters and "actual" operation's parameters. Defining mappings between parameters enables the support of "legacy" Web services. Providers do not need to change the message parameters in their actual service codes.

Assume that a service provider *SP* offers a given operation *op*. The follwowing three cases are then possible: (i) If there is a community C_i that contains a generic operation op_{ik} similar to *op*, *SP* would import op_{ik} "as is"; (ii) If there is a community C_i that contains a generic operation op_{ik} "closely" similar to *op* (e.g., *op* has less input parameters than defined in op_{ik}), *SP* would import op_{ik} using *projection*, *extension*, and/or *adjustment* technique; (iii) If no community has an operation similar or "closely" similar to *op*, *SP* would define a new community C_j that has *op* as a generic operation and *SP*'s service as a member. The latter case is similar to the "traditional" WSDL/UDDI/SOAP Web service model where service providers create descriptions for their services. The difference is that, in our case, *SP* instantiates

the attributes and concepts of the community ontology while in the "traditional" model, providers define their service descriptions from scratch.

3.2 Operational Description of Communities

As mentioned previously, a generic operation is described at four different levels: *syntactic*, *static semantic*, *dynamic semantic* and *qualitative* levels. In this section, we give a detailed description of generic operation attributes for each of those levels.

3.2.1 Syntactic Attributes

We define two levels for syntactically describing a generic operation: *message* and *operation* levels. Attributes at the message level describe the structure of the messages defined within the operation such as the number of parameters within a message. Attributes at the operation level describe general-purpose features of the generic operation such as the name and ID of the operation.

3.2.1.1 Message Syntax

Generic operations have *input* and *output* messages. Each input or output message contains one or more parameters defined by their *names*. The name of a parameter is unique within a given message. Let us for example consider a generic operation `checkEligibilityWIC` which checks citizen's eligibility for a nutrition program (WIC stands for Women Infant and Children). The input message of this operation contains *income*, *familySize*, and *zipCode* as parameter names. The output message of `checkEligibilityWIC` has *approved* and *duration* as parameter names. Although message parameters are predefined by community providers, service providers have the ability to add new parameters or remove pre-defined ones. Threfore the number of parameters within a messages may be changed by service providers.

We define two sets $In(op_{ik})$ and $Out(op_{ik})$ for each generic operation op_{ik}. $In(op_{ik})$ and $Out(op_{ik})$ contain the list of input parameters' and output parameters'

names of op_{ik} respectively. We also define two attributes NI_{ik} and NO_{ik} that give the number of input and output parameters in op_{ik} respectively. For example, in the case where op_{ik} = `checkEligibilityWIC`, $In(op_{ik}) = \{income, familySize, zipCode\}$, $NI_{ik} = 3$, $Out(op_{ik}) = \{approved, duration\}$, and $NO_{ik} = 2$.

3.2.1.2 Operation Syntax

A generic operation has a unique identifier, called *G-op-ID*, that takes the form of a *Universally Unique ID* (UUID). A UUID is an identifier that is unique across both space and time [5]. The operation has also a *name* and a text *description* that summarizes the operation's features. The *binding* defines the message formats and protocols used to interact with the operation. An operation may be accessible using several bindings such as *SOAP/HTTP* and *SOAP/MIME*. The binding of an operation is assigned by the service provider. This is in contrast to the rest of syntactic attributes whose values are pre-defined by community providers. Indeed, the binding attribute is dependent on the way the generic operation is implemented at the service provider side. A provider may offer SOAP/HTTP access to a generic operation supported by its Web service while another provider may prefer to use SOAP/MIME for the same operation.

The *mode* of an operation refers to the order according to which its input and output messages are sent and received. It states whether the operation initiates interactions or simply replies to invocations from other services. We define two operation modes: *In/Out* or *Out/In*. One of these values is assigned by community providers to each operation. *In/Out* operation first receives an input message from a client, process it (locally or forward it to another service), and then returns an output message to the client. *Out/In* first sends an output message to a server and receives an input message as a result. `checkEligibilityWIC` is an example of *In/Out* operation. As specified in WSDL standard, some operations may be limited to an input or an output message [5]. For example, `expirationWIC` is an operation that automatically notifies citizens about the termination of their eligibility period

for WIC programs. This operation obviously does not require any input message. However, such operations may be considered as *In/Out* or *Out/In* operations where the input or output message is empty. Hence, without loss of generality, we focus in the proposed framework on the aforementioned operation modes.

3.2.2 Static Semantic Attributes

The static semantics of a generic operation describe semantic properties that are independent of the execution of the operation. It specifies the semantics of the operation itself (e.g., what does the operation do) as well as the semantics of input and output messages defined within the operation.

3.2.2.1 Message Semantics

Messages must be semantically described so that they can be "correctly" interpreted by service providers and consumers. For that purpose, we associate a *message type* $\mathcal{MT}$ to each message. $\mathcal{MT}$ gives the general semantics of the message. For example, a message may represent a "purchase order" or an "invoice". *Vertical* ontologies are the ideal concept to describe the type of message. An example of such ontology is *RosettaNet's PIPs* (*Partner Interface Processes*).

The message type does not capture the semantics of message parameters. We define the following attributes to model the semantics of message parameters: *data type*, *business role*, *unit*, and *language*. The *data type* gives the range of values that may be assigned to the parameter. We use XML Schema's *built-in* data types as the typing system. *Built-in* types are pre-defined in the XML schema specification. They can be either *primitive* or *derived*. Unlike *primitive* types (e.g., *string*, *decimal*), *derived* types are defined in terms of other types. For example, *integer* is derived from the *decimal* primitive type.

The *business role* gives the type of information conveyed by the message parameter. For example, an `address` parameter may refer to the first (street address and unit number) or second (city and zip code) line of an address. Another exam-

ple is that of a `price` parameter. It may represent a total price or price without taxes. Business roles take their values from a pre-defined taxonomy. Every parameter would have a well-defined meaning according to that taxonomy. An example of such taxonomy is *RosettaNet*'s business dictionary. It contains a common vocabulary that can be used to describe business properties. For example, if the *price* parameter has an "extendedPrice" role (defined in *RosettaNet*), then it represents a "total price for a product quantity". For flexibility purposes, different community providers may adopt different taxonomies to specify their parameters' business roles. As for categories, we use XML namespaces to prefix business roles with the taxonomy according to which they are defined.

The *unit* attribute refers to the measurement unit in which the parameter's content is provided. For example, a *weight* parameter may be expressed in "Kilograms" or "Pounds". A *price* parameter may be in "US Dollars", "Canadian Dollars", or "Euro". An *eligibility period* parameter may be specified in days, weeks, or months. We use standard measurement units (length, area, weight, money code, etc.) to assign values to parameters' units. If a parameter does not have a unit (e.g., *address*), its unit is equal to "none". The content of a message parameter may be specified in different languages. For example, a `profession` parameter may be expressed in English or Spanish. An `English-Urdu-translation` operation takes as input, an English word (input parameter) and returns as output, its translation in Urdu (output parameter). We adopt the standard taxonomy for languages to specify the value of this attribute. We give below a formal definition of a message parameter. Each message (input or output) is defined as a set of such parameters.

Definition 3.1 – *Message Parameter.* A message parameter P is defined as a tuple $(\mathcal{T},\mathcal{R},\mathcal{U},\mathcal{L})$ where $\mathcal{T}$ is the parameter's data type (in XML Schema), $\mathcal{R}$ is its business role taken from a taxonomy for business roles, $\mathcal{U}$ gives P's unit of measurement, and $\mathcal{L}$ is the language according to which P is expressed. $\diamondsuit$

The content of static semantic attributes is assigned by community providers. The data type, unit, and language attributes may be changed by service providers.

This is in contrast to the message type and business role which model the core of the message semantics and hence cannot be altered. Service providers have the flexibility to support a data type, unit, or language different from those specified by community providers. For example, a service provider may decide to support a *weight* parameter in "Kilograms" although the community providers specified "Pounds" as the measurement unit for this parameter.

3.2.2.2 Operation Semantics

The static semantic of an operation is defined by the following attributes: *serviceability*, *provider type*, *consumer type*, *purpose*, and *category*. These attributes model the core of the operation's semantics. Hence, they are exclusively assigned by community providers.

The *serviceability* attribute gives the type of assistance provided by the operation. Examples of values for this attribute are "cash", "in-kind", "informational", and "educational". TANF (Temporary Assistance for Needy Families) is an example of welfare program that provides financial support to needy families. A food stamp is an example of in-kind support available to indigent citizens. Returning the list of senior activity centers is an example of informational support. Enhancing communication skills of visually impaired people is an example of educational support. Other types of support may be mentioned by assigning the value "other" to this attribute.

A generic operation may be supported via one or several *provider types*. A provider may be governmental ("federal", "state", "local", and "tribal") or non-governmental ("non-profit" and "business") agencies. For example, `nursingHome` may be provided by the Department for the Aging (government agency) and Red Cross (non-profit). The *consumer type* property specifies the group of citizens (e.g., children, pregnant women) that are eligible to the operation's welfare program. Different groups may be eligible for the same benefit. For example, WIC (Women, Infant, and Children) is a program for pregnant women, lactating mothers, and

children.

Each generic operation performs a certain functionality for a specific area of interest. This is specified through the *purpose* and *category* attributes respectively. An operation inherits the category of the community in which it is defined. Hence, all operations that belong to the same community share the same category. The *purpose* attribute describes the goal of the operation. It is defined by four attributes: *function*, *synonyms*, *specialization*, and *overlapping*. The *function* describes the business functionality offered by the operation. Examples of functions are "eligibility", "registration", and "mentoring". *Synonyms* and *specialization* attributes work as they do for categories. *Overlapping* contains the list of purposes that *overlap* with the purpose of the current operation. Let op_{ik} and op_{jl} be two generic operations. We say that $purpose_{ik}$ *overlaps with* $purpose_{jl}$ if composing op_{ik} with op_{jl} is "meaningfull". By meaningfull, we mean that the composition provides a *value-added* (in terms of purposes). For example, two operations that have "eligibility" and "registration" as respective purposes may be combined to first check whether citizens are eligible for a given social a program and then register them for that program. As for categories, it is the responsibility of the community providers to identify related purposes and assign them to the overlapping attribute.

3.2.3 Dynamic Semantics

Dynamic semantics allows the description of attributes related to the execution of generic operations. Those attributes may relate the execution of an operation op_{ik} to the execution of other operations (*inter-operation* attributes) or describe features inherent to the execution of op_{ik} (*intra-operation* attributes). Inter-operation attributes define the execution order of op_{ik} with respect to other operations. We identify two inter-operation attributes: *pre-operation* and *post-operation* which give the list of operation whose execution precedes and follows op_{ik}'s execution respectively. Intra-operation attribute, also called *behavior* defines the internal business logic of op_{ik}. The definition of the aforementioned attributes is based on the notion

of *execution state* described below.

3.2.3.1 Operation Execution States

The execution of an operation op_{ik} generally goes through four major observable states (Figure 3.4): *Ready*, *Start*, *Active*, and *end*. The execution of op_{ik} is in the *Ready* state if the request for executing op_{ik} has not been made yet. The *Start* state means that op_{ik} execution has been initiated. It refers to one of the following events: (i) an input message is sent to op_{ik} if op_{ik}'s mode is *In/Out*; or (ii) an output message has been sent from op_{ik} if op_{ik}'s mode is *Out/In*. We say that op_{ik} is in the *Active* state if op_{ik} has already been initiated and the corresponding request is being processed. After processing the request, the operation reaches the *End* state during which results are returned. It refers to one of the following events: (i) an output message is sent to the client if op_{ik}'s mode is *In/Out*; or (ii) an input message is received from the server if op_{ik}'s mode is *Out/In*.

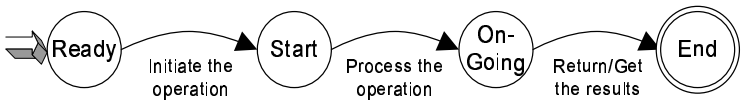

Figure 3.4: Operation Execution States

We define a precedence relationship between states, noted $\longrightarrow_t$, as follows: $S_1 \longrightarrow_t S_2$ if S_1 occurs before S_2. The execution states are totally ordered according to $\longrightarrow_t$ as follows: *Ready* $\longrightarrow_t$ *Start* $\longrightarrow_t$ *Active* $\longrightarrow_t$ *End*

3.2.3.2 Pre-Operations

Executing a Web service operation may require going through a pre-defined process that involves the execution of several operations called *pre-operations*. This pre-defined process is dictated by government regulations or the internal business process of the Web service. For example, senior citizens must first register with an Area Agency on Aging (AAA) via `checkRegistration` operation before applying for

any welfare program. They may also reflect the business logic of the Web service. For example, senior citizens must order a meal from a participating restaurant via the `orderMeal` operation before requesting its delivery through the `mealsOnWheels` operation.

Let us consider two generic operations op_{ik} and op_{jl} that belong to the same or different communities. We say that op_{ik} is a *pre-operation* of op_{jl} if the invocation of op_{jl} is preceded by the execution of op_{ik}. We call op_{ik} and op_{jl} *source* and *target* operations respectively. An operation may have several pre-operations. It may also be the source (i.e., pre-operation) of several operations. We give below a formal definition of the *pre-operation* relationship.

Definition 3.2 – *Pre-operation.* Let op_{ik} and op_{jl} be two generic operations. op_{ik} is a *pre-operation* of op_{jl} if $End(op_{ik}) \longrightarrow_t Ready(op_{jl})$. $\diamondsuit$

The definition of a pre-operation relationship includes a source operation op_{ik}, target operation op_{jl}, and the *condition* and *mandatory* attributes. The *condition* is a predicate over op_{ik}'s input and output parameters. op_{jl} can be invoked only if all its pre-operations reached their End state and their conditions are true. If no condition is specified for a given pre-operation then the default value is "true". For example, `mealsOnWheels` is executed only if the `orderMeal` operation has been approved. The *mandatory* attribute takes boolean values and specifies whether executing the source operation is mandatory or optional. If this attribute is true then the relationship between op_{ik} and op_{jl} is obligatory. Otherwise, it is *recommended*. For example, senior citizens *must* have ordered meals from a restaurant before requesting home delivery from a volunteer center. However, it is recommended to get the list of participating (`participatingRestaurants`) restaurants before ordering a meal. Indeed, users can directly order a meal from a restaurant if they already know that restaurant's name.

3.2.3.3 Post-Operations

The execution of a given operation may trigger the invocation of other operations called *post-operations*. For example, a senior citizen that registers successfully for a food check program (`registerFoodCheck`) is required to register for a nutritional counseling course (`registerNutritionCourse`). We say that op_{ik} is a *post-operation* of op_{jl} if the termination of op_{jl} precedes the invocation of op_{ik}. We call op_{jl} and op_{ik} *source* and *target* operations respectively. An operation may have several post-operations. It may also be the target (i.e., post-operation) of several operations. Note that if op_{ik} is a pre-operation of op_{jl} then op_{jl} is not necessarily a post-operation of op_{ik}. For example, `checkRegistration` is pre-operation of `orderMeal`. However, `orderMeal` is not a post-operation of `checkRegistration`. Indeed, users do not need to order meals whenever their registration with the Department on the Aging is checked. We give below a formal definition of the *post-operation* relationship.

Definition 3.3 – *Post-operation.* Let op_{ik} and op_{jl} be two generic operations. op_{ik} is a *post-operation* of op_{jl} if $End(op_{jl}) \longrightarrow_t Ready(op_{ik})$. $\diamondsuit$

As for pre-operations, we associate a *condition* and *mandatory* attribute to each post-operation relationship. A target operation enters the initiation state if *at least* one of its source operations has reached its End state and the corresponding condition is true. A post-operation may also be mandatory or optional. For example, a pregnant women that registers for a food check program women (`registerFoodCheck`) *must* also register for nutritional counseling course by invoking `registerNutritionCourse` (mandatory = true). The post-operation `register` is optional (mandatory = false). Indeed, citizens do not necessarily have to register with the Department for the Aging if they are not willing to do so.

3.2.3.4 Operation Dependency Diagram

Pre and post-operations provide means to specify pre-defined business processes within a community. In the case of e-government, those business processes are

mostly driven by government regulations and laws. We use *UML activity diagrams* to model pre-operation and post-operation relationships [56]. We refer to such diagrams as a *operation dependency diagrams* (OPD). Activity diagrams show the flow of *activities* in a business process. In an OPD diagram, each activity represents a generic operation. Generic operation within a diagram may belong to different communities. In this case, the community name prefaces the G-op-IDs. We refer to such pre/post operations as *remote* pre/post operations. OPD diagrams are defined by community providers. Service providers may also add pre/post-operations while importing a generic operation to model business processes inherent to their Web services.

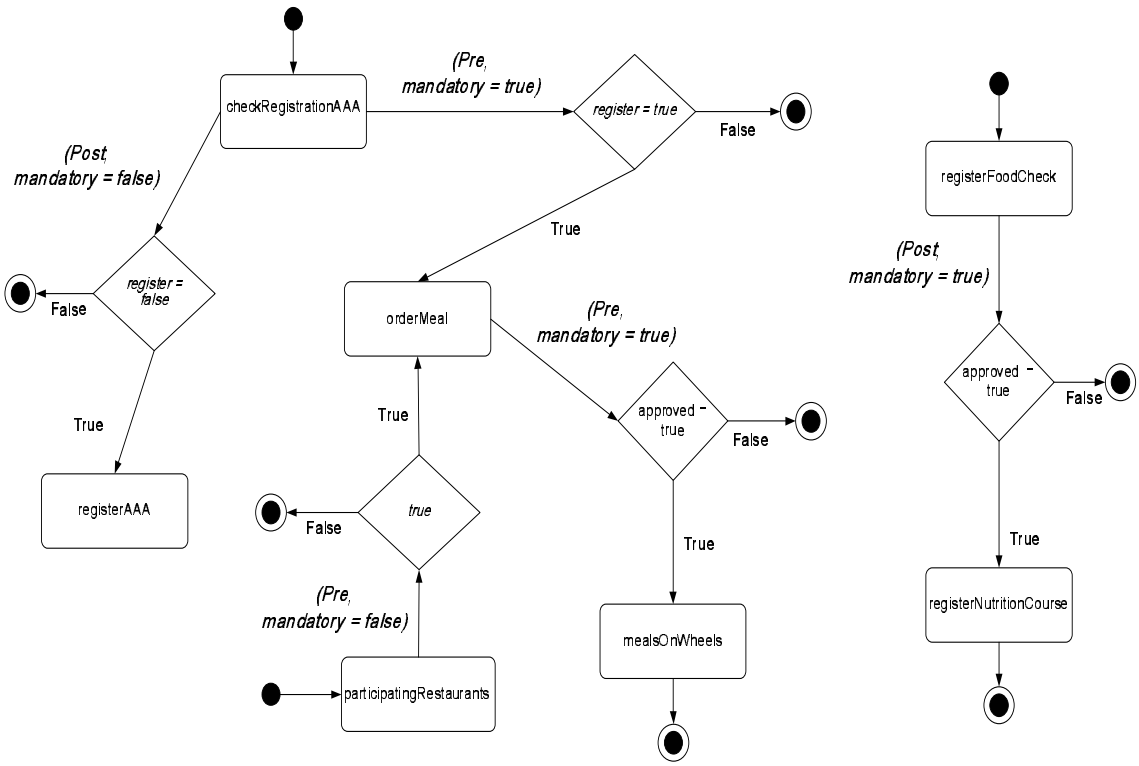

Figure 3.5: An Example of OPD Diagram

We depict in Figure 3.5 part of the diagram for the `elderly` community. The filled circle is a starting point of the diagram. The filled circle with a bor-

der is an ending point. Each edge in the diagram is labeled with a *relationship* attribute. This attribute takes one of the values "Pre" or "Post" to specify whether the edge models a pre or post-operation. For example, the edge `checkRegistration` → `orderMeal` models a pre-operation relationship. It specifies that users should execute `checkRegistration` before initiating the execution of `orderMeal`. Edges are also labeled with the *mandatory* attribute. For example, `orderMeal` is a mandatory pre-operation of `mealsOnWheels`. However, `orderMeal` may be invoked even if `participantRestaurants` has not previously been executed.

OPD diagrams may also indicate that one operation *conditionally* follows another. For example, the *diamond* between the `checkRegistration` and `orderMeal` operations states that the the value returned by the `register` parameter should be "true" before invoking `orderMeal`. The edge `registerFoodCheck` → `registerNutritionCourse` models a post-operation relationship. It mentions that all citizens applying for a food check program should register for a nutrition course. Conditional constructs (i.e., diamonds) are labeled with a *mandatory* attribute. For example, `orderMeal` is a mandatory pre-operation of `mealsOnWheels`. However, `orderMeal` may be invoked even if `participantRestaurants` has not previously been executed.

3.2.3.5 Behavior

The *behavior* of a generic operation op_{ik} refers to the outcome expected after executing op_{ik} given a specific condition. It is defined by a set of *business logic rules* where each rule R_{ik}^{m} has the following format:

$$R_{ik}^{m} = \frac{(PreParameters_{ik}^{m}, PreCondition_{ik}^{m})}{(PostParameters_{ik}^{m}, PostCondition_{ik}^{m})}$$

$\text{PreParameters}_{ik}^{m}$ and $\text{PostParameters}_{ik}^{m}$ are sets of parameters. Each parameter is defined by name, data type, business role, unit, and language as stated in Definition 3.1. The elements of $\text{PreParameters}_{ik}^{m}$ and $\text{PostParameters}_{ik}^{m}$ generally

refer to op_{ik}'s input and output parameters. However, they may in some cases refer to parameters that are neither input nor output of op_{ik}. For example, assume that the *address* of every citizen registered with the Department on the Aging is stored in the department's database. In this case, this parameter should not be required as input for the `orderMeal` operation since its value could be retrieved from the database. $PreCondition_{ik}^{m}$ and $PostCondition_{ik}^{m}$ are predicates over the parameters in $PreParameters_{ik}^{m}$ and $PostParameters_{ik}^{m}$ respectively. The rule R_{ik}^{m} specifies that if $PreCondition_{ik}^{m}$ holds when the operation op_{ik} starts, then $PostCondition_{ik}^{m}$ holds after op_{ik} reaches its End state. If $PreCondition_{ik}^{m}$ does not hold, there are no guarantees about the behavior of the operation. Preconditions generally specify relationships between input values. Similarly, post-conditions generally specify relationships between the returned values. The following is an example of the pre- and post-condition of a rule associated with the operation `registerFoodCheck`:

$$\frac{(income < 22,090 \wedge familySize \geq 2 \wedge zipCode = 22044)}{(approved = true \wedge duration = 6)}$$

The rule uses the parameters *income* (unit = {year, US dollar}), *familySize*, and *zipcode* in the pre-condition (income < 2,090) $\wedge$ (familySize $\geq$ 2) $\wedge$ (zipCode = 22044). The attributes *approved* and *duration* (unit = {month}) are used in the post-condition (approved = true) $\wedge$ (duration = 6). The rule specifies that citizens with a yearly income less than 22,090 US dollars and a minimum household size 2 are eligible for food checks for a 6-month period. Business logic rules are pre-defined by community providers. However, service providers may add new rules to reflect a behavior specific to their Web services.

3.2.4 Qualitative Properties

Multiple Web services that belong to the same community may import the same generic operation. It is hence important to define a set attributes that help select the "best" Web service supporting a given functionality. For this purpose, we define a *Quality of Operation* (QoP) model based on a set of *qualitative* attributes that are

transversal to all operations such as the cost and response time.

QoP Group	QoP Attribute	Definition
Run-time	Response Time	$\text{Time}_{process}(op_{ik})$ + $\text{Time}_{results}(op_{ik})$ where $\text{Time}_{process}$ is the time to process op_{ik} and $\text{Time}_{results}$ is the time to transmit/receive the results
	Reliability	$\text{N}_{success}(op_{ik})/\text{N}_{invoked}(op_{ik})$ where $\text{N}_{success}$ is the number of times that op_{ik} has been successfully executed and $\text{N}_{invoked}$ is the total number of invocations
	Availability	UpTime(op_{ik})/TotalTime(op_{ik}) where UpTime is the time op_{ik} was accessible during the total measurement time TotalTime
Business	Cost	Dollar amount to execute the operation
	Reputation	$\sum_{u=1}^{n} Ranking_u(op_{ik})/n$, $1 \leq$ Reputation ≤ 10 where $Ranking_u$ is the ranking by user u and n is the number of the times op_{ik} has been ranked
	Regulatory	Compliance with government regulations, $1 \leq$ Regulatory ≤ 10
Security	Encryption	A boolean equal to *true* iff messages are encrypted
	Authentication	A boolean equal to *true* iff consumers are authenticated
	Non-repudiation	A boolean equal to *true* iff participants cannot deny requesting or delivering the service
	Confidentiality	List of parameters that are not divulged to external parties

Table 3.2: Quality of Operation Model

The international quality standard ISO 8402 describes *quality* as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" [138, 108]. We define QoP as a set of non-functional attributes that may impact the quality of the operations imported by a Web service. There are many QoP attributes important to Web services operations. We organize them into three groups of quantifiable attributes based on type of measurement performed by each attribute: *run-time*, *business*, and *security* (Table 3.2).

Run-time Attributes – These attributes enable the measurement of properties that are related to the execution of an operation op_{ik}. We identify three run-time attributes: *response time*, *reliability*, and *availability*. The *response time* measures the expected delay in seconds between the moment when op_{ik} enters the Start state (i.e., op_{ik} is initiated) and reaches the End state (i.e., op_{ik} gets or sends the results). Time(op_{ik}) is computed using the expression $\text{Time}_{process}(op_{ik})$ + $\text{Time}_{results}(op_{ik})$. This means that the response time includes the time to process the operation ($\text{Time}_{process}$) and the time to transmit or receive the results ($\text{Time}_{results}$). The *reliability* of op_{ik} is the ability of the operation to be

executed within the maximum expected time frame. Reliability(op_{ik}) is computed based on historical data about previous invocations of the operation using the expression $\mathrm{N}_{success}(op_{ik})/\mathrm{N}_{invoked}(op_{ik})$ where $\mathrm{N}_{success}(op_{ik})$ is the number of times that the operation has been successfully executed within maximum expected time frame and $\mathrm{N}_{invoked}(op_{ik})$ is the total number of invocations. The *availability* is the probability that the operation is accessible. Availability(op_{ik}) is measured by the expression UpTime(op_{ik})/TotalTime(op_{ik}) where UpTime is the time op_{ik} was accessible during the total measurement time TotalTime.

Business Attributes – These attributes allow the assessment of an operation op_{ik} from a business perspective. We identify three business attributes: *cost*, *reputation*, and *regulatory*. The *cost* gives the dollar amount required to execute op_{ik}. The *reputation* of op_{ik} is a measure of the operation's trustworthiness. It mainly depends on users' experiences on invoking op_{ik}. Users are given a range to rank Web service operations (e.g., between 1 and 10). The lowest value refer to the best ranking. Different users may have different opinions on the same operation. The reputation of op_{ik} is defined by the average ranking given by users to the operation. Reputation(op_{ik}) is computed by the expression $\sum_{u=1}^{n} Ranking_u(op_{ik})/n$, where $Ranking_u$ is the ranking by user u and n is the number of the times the operation has been ranked. The *regulatory* property is a measure of how well op_{ik} is aligned with government regulations. Regulatory(op_{ik}) is value within a range (e.g., between 1 and 10). The lowest value refer to an operation that is highly compliant with government regulations.

Security Attributes – These attributes describe whether the operation op_{ik} is compliant with security requirements. Indeed, service providers collect, store, process, and share information about millions of users who have different preferences regarding security of their information. We identify four properties related to security and privacy: *encryption*, *authentication*, *non-repudiation*, and *confidentiality*. *Encryption* is a boolean that indicates whether op_{ik}'s message are securely exchanged (us-

ing encryption techniques) between servers and clients. *Authentication* is a boolean that states whether op_{ik}'s consumers (users and other services) are authenticated (e.g., through passwords). *Non-repudiation* is a boolean that specifies whether participants (consumers and providers) can deny requesting or delivering the service after the fact. The *confidentiality* attribute indicates which parties are authorized to access the operation's input and output parameters. Confidentiality(op_{ik}) contains op_{ik}'s input and output parameters that should not be divulged to external entities (i.e., other than the service provider). If a parameter does not belong to Confidentiality(op_{ik}), then no confidentiality constraint is specified on that parameter. Assume that *confidentiality(*op_{ik}) = {*SSN*, *salary*} where *SSN* and *salary* are two op_{ik}'s input parameters. The content of this attribute states that those two parameters are kept private by op_{ik}'s provider.

3.3 Registering Web Services With Communities

Registering a Web service with a community refers to the process of importing generic operations. The invocation of an imported operation is translated into the invocation of an "actual" service operation. The correspondence between imported and "actual" operations is done through the *mapping* attribute. For each imported operation, the provider gives the ID of the corresponding "actual" operation. It also defines a one-to-one mapping between the imported operation's parameters and "actual" operation's parameters. Defining mappings between parameters enables the support of "legacy" Web services. Providers do not need to modify the message parameters in their actual service codes.

3.3.1 The Web Service Registration Process

The registration process is handled by a network of software agents associated to service and community providers. *Member* and *community agents* (MA_j and CA_i) are attached to each service and community provider (SP_j and CP_i) respectively

(Figure 3.6). MA_j handles the registration of SP_j's Web services with the community C_i. SP_j registers its service *WS-ID* using the following *registration statement*:

Register Service *WS-ID* **With Community** C_i
Name *service-name*
Description *service-description*
[**Imported Generic** *G-op-ID*
<*importing statements*>
Mappings With Actual *Op-ID*
[*G-op-ID.<parameter>* **Maps To** *Op–ID.<parameter>*]$^+$]$^+$

The clauses in the aforementioned statement correspond to the different attributes defined for service members within the community ontology. The <*importing statements*> is a sequence of statements for importing generic operation. A Web service may import several generic operations as stated by the "+" iteration symbol.

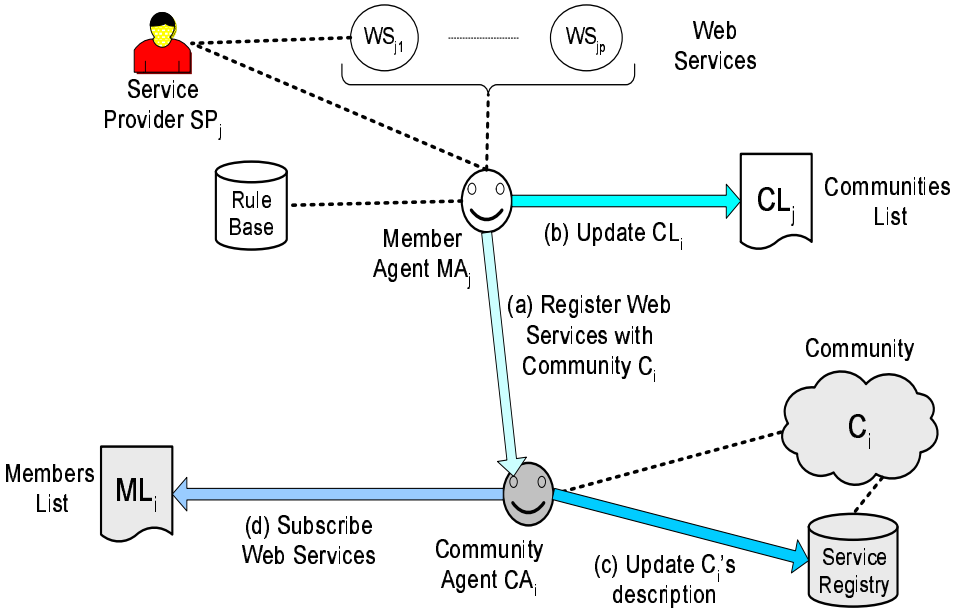

Figure 3.6: The Web Service Registration Process

MA_j parses the registration statement and sends a registration message *SP_Register(WS-ID,name,desc,imported)* to CA_i (Figure 3.6, step (a)). The message includes the service ID (*WS-ID*), name, description, and the list of imported operations. The *imported* set is equal to {*(G-op-ID,mappings,AttSet,NotImpSet)*}. It includes the mappings of G-op-ID (ID of the imported operation) with actual service operation. *AttSet* is the set of G-op-ID's attributes with their values as assigned by SP_j. It is defined by the set {*(attribute,value)*}. *NotImpSet* is the list of non-imported attributes. MA_j maintains a *Communities List* CL_j and *Rule Base* RB_j. CL_j is the list of communities with which SP_j's Web services are registered. Each entry in this list contains the G-op-ID of an imported operation, the WS-ID of the service that imported it, and the ID (CA-ID) of the community agent CA_i. To enable fast access to CL_j, we sort it on the G-op-ID column using *Counting Sort* algorithm [36]. RB_j contains a set of rules that enable MA_j to react to changes issued by community providers.

Upon reception of the registration message, CA_i updates the content of C_i's *members list* ML_i (Figure 3.6, steps (c) and (d)). Each entry in ML_i contains the ID of the imported operation (op-ID), the ID of the importing service (*WS-ID*), the ID of *WS-ID*'s agent (*MA-ID*), *WS-ID*'s *status* ("available ", "unavailable", or "unsubscribed"), and the list *NotImp* of *G-op-ID*'s attributes not imported by *WS-ID*. The *NotImp* column is assigned with the content of *NotImpSet* included in the registration message. We sort ML_i on the op-ID column using *Counting Sort* algorithm to enable fast access to ML_i [36].

(00) `Member Agent` MA `Algorithm` {
(01) `Upon Reception of`
(02) **Register (WS-ID,name,desc,imported,**C_i**)** statement `From` SP
(03) `Parse` the registration statement;
(04) `Send` SP_Register (WS-ID,name,desc,imported) `To` CA_i;
(05) }

Figure 3.7: Service Registration: Member Agent Algorithm

Figures 3.7 and 3.8 give the algorithm executed by a community agent CA_i and member agent MA during registration. SP_Register(WS-ID,name,desc,imported) describes the actions executed by CA_i as a reaction to a registration message sent by MA. The *Insert_Member()* function allows the insertion of a new member in C_i's description. *Insert_ML()* function allows the insertion of a new entry in the ML_i list for members.

```
(00) Community Agent CA_i Algorithm {
(01) Upon Reception of
(02) SP_Register (WS-ID,name,desc,imported) message From MA-ID
(03)  Insert_Member (C_i,WS-ID,name,desc,imported);
(04)  for each G-op-ID ∈ imported
(05)  do Insert_ML (G-op-ID,WS-ID,MA-ID,''available'',NotImpSet);
(06) }
```

Figure 3.8: Service Registration: Community Agent Algorithm

3.3.2 Importing Generic Operations

Service providers use generic operations as "templates" to define their operations. A Web service may offer all or some of the generic operations defined within a community. The provider specifies the G-op-IDs of the operations imported by its service. By adopting a generic operation, the service provider "promise" to abide by all attributes (syntactic, semantic, and behavioral) of that operation except those changed explicitly during importation. Providers may customize generic operations to best fit their capabilities via *importing statements*. Customization has the important advantage of enabling flexible and personalized Web service descriptions. It is important to note that the customization process does not affect the description of generic operations. Two service providers may import the same generic operation in different ways.

We define three importing statements: *projection*, *extension*, *adjustment*. Importing statements are defined within member agents. Projection, extension,

and adjustment may be combined to define imported operations. For example, a service provider may use projection and extension to remove existing parameters from a generic operation and add new ones.

Projection – A generic operation G-op-ID imported by *projection* uses a subset of the input/output parameters defined in G-op-ID. The rest of the message parameters are not imported by the service and hence, are included in the *NotImpSet* sent by the member agent to the community agent. Assume that a `checkRegistrationAAA` operation includes `register` (boolean) and `registrationDate` (date) as output parameters. A service provider may customize this operation by keeping only the `register` parameter if it is not interested in returning citizen's registration date. In what follows, we give the general form of a *Project* statement. The input and output clauses give the subset of G-op-ID's parameters supported by the imported operation:

Project *G-op-ID*
Input <*list-of-parameter-names*>
Output <*list-of-parameter-names*>

The *AttSet* submitted with the registration message takes the form {*(att,value)*} where *att* is a projected parameter and the content of *value* is "null" since no new value is assigned to the input or output parameter. We refer to *AttSet.Attributes* as the list of projected parameters. The list of not imported attributes *NotImpSet* is defined by the expression $(In(\mathrm{op}_{ik}) \cap Out(\mathrm{op}_{ik}))$ – *AttSet.Attributes*.

Extension – An imported operation defined by *extension* adds input and/or output parameters to the corresponding generic operation. The new parameters and their values are included in the registration messages sent by the member agent to the community agent. For example, a service provider (e.g., a volunteer center) may extend the `mealsOnWheels` operation by adding the deliverer's cell phone number as an output parameter. Service providers must assign values to the attributes of

each new message parameters, namely data type, unit, business role, and language. Below is the general form of an extension statement:

Extend *G-op-ID*
  Input [*(<name>, <data-type>, <unit>, <business-role>, <language>)*]*
  Output [*(<name>, <data-type>, <unit>, <business-role>, <language>)*]*

The *AttSet* submitted with the registration message takes the form {*(att,value)*} where *att* is a new attribute added by the extension statement and *value = (<name>, <data-type>, <unit>, <business-role>, <language>)*. The list of not imported attributes *NotImpSet* is empty since all message parameters are imported.

Adjustment – The aim of an adjustment statement is to modify the content of a generic operation's attributes. Service providers may assign values to attributes whose content is undefined (e.g., qualitative attributes) or change the content of previously assigned attributes (e.g., language attribute). Adjustment is done by adding a value to an attribute (Add clause) or deleting an existing value from it (Delete clause). For example, service providers may modify the language attribute if their operation supports a language different from the one specified by community providers. The Add and Delete clauses may be combined to remove and add values to an attribute. The new and deleted values are included in the registration messages sent by the member agent to the community agent. The general form of an adjustment statement is given below:

Adjust *G-op-ID*
  Add [<*value*> **To** <*attribute*>]*
  Delete [<*value*> **From** <*attribute*>]*

The *AttSet* submitted with the registration message takes the form {(att,["+"/"-"],value)} where *att* is an attribute modified by the adjustment statement. The *value* is preceded by a "+" or "-" symbol depending on whether that value is added or deleted from *att* respectively. The list of not imported attributes *NotImpSet* is empty since all message parameters are imported.

3.4 A Peer-to-Peer Approach for Managing Communities

Communities and their members operate in a highly dynamic environment where changes can be launched to adapt to actual business climate (e.g., economic, politic, organizational). Changes are initiated by community or service providers. At the community providers side, generic operations may be dynamically added, deleted, and modified. If a generic operation G-op-ID is deleted or modified, then all members that are supporting G-op-ID should be notified to ensure global consistency. At the service provider side, a Web service may cancel its membership with a community, make its operations temporarily unavailable, or modify the definition of its imported operations. The community provider should in this case be notified to avoid references to inexistent or obsolete imported operations.

In our approach, all changes are introduced through member and community agents. Agents automatically interact with their peers to manage changes. We consider two types of changes based on the party that launched them: community or service providers.

3.4.1 Propagating Changes Initiated by Community Providers

Community providers (CPs) may modify the definition of their generic operations (Figure 3.9). For example, they may change the pre-operation attributes to reflect new government regulations. For this purpose, each CP_i executes a *Modify* statement defined in its CA_i agent (Figure 3.9, step 1). The statement includes the G-op-ID of the operation to be modified by CP_i and a *ModifySet* that contains the list of attributes to be modified along with their new content. *ModifySet* is defined by the set {(<attribute>,["+"/"-"],<value>)}. We use the notation *ModifySet.Att* to refer to the set of modified attributes. CA_i will then access the service registry and update C_i's description by changing the content of *ModifySet.Att*'s at-

tributes (Figure 3.9, step 2). In the third step, CA_i accesses ML_i list to determine the list $\mathcal{L}$ of members *subscribed* with CP_i's changes. A member WS-ID is *subscribed* with CP_i's changes if WS-ID imports G-op-ID and at least one attribute in *ModifySet.Att* is imported by WS-ID. CA_i assigns the value "unavailable" to the *status* of each subscribed member. This prevents references to members that imported "obsolete" generic operations (Figure 3.9, step 3). Finally, CA_i sends a *CP-Modify(G-op-ID, WS_j, ModifySet)* notification to each subscribed member's MA_j (Figure 3.9, step 4). We give below a formal definition of a subscription.

Definition 3.4 – *Subscription.* Let C_i be a community and WS_j be a member that imported an operation op_{ik}. WS_j is *subscribed* with changes specified in *CP-Modify(G-op-ID,ModifySet)* if (i) G-op-ID = op_{ik} and (ii) *ModifySet.Att* – *ML_i[k].NotImp* $\neq \emptyset$ where k is the entry corresponding to G-op-ID and WS_j. $\diamondsuit$

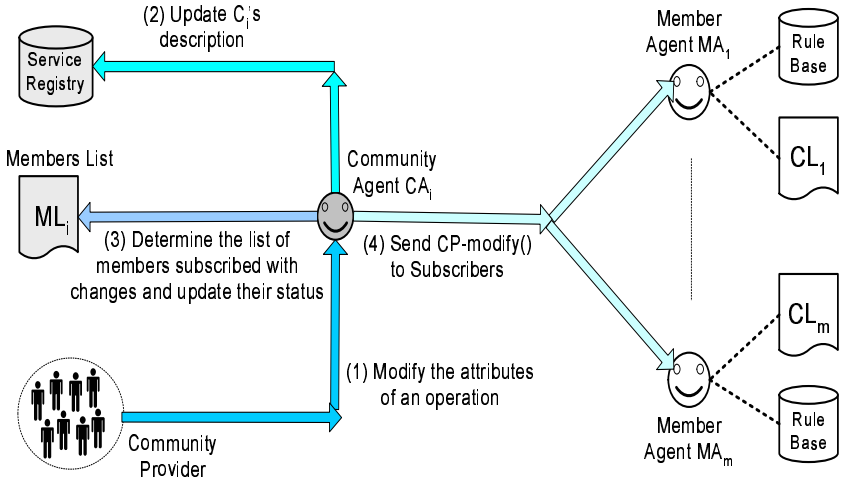

Figure 3.9: Propagating Changes Initiated by a Community Provider to its Members

Because pre and post-operations for a given operation op_{ik} may belong to different communities, we associate two "dual" subscription lists $PPCS_i$ and $PPMS_i$ to each community agent CA_i. These lists are used to notify relevant parties (communities and members) about changes that are related to remote pre and post-operations. $PPCS_i$ (*Pre and Post-operation Community Subscription*) contains the list of com-

munities C_j that use an operation op_{ik} of C_i as a remote pre or post-operation. Each entry in PPCS$_i$ contains the ID (op-ID) of op_{ik} and the ID of C_j's agent (CA-ID). Such entry is created at C_j's definition time; at that time, CA_j sends a *CP-PP-subscribe-CP(op-ID)* to CA_i (CP stands for Community Provider and PP for Pre/Post operation). PPCS$_i$ is used to notify C_j about changes that occur in op_{ik}. Each time CP$_i$ executes a *Modify* statement on op_{ik}, CA_i accesses PPCS$_i$ list and sends a notification *CP-PP-Alert-CP(op_{ik})* to each community agent CA_j that uses op$_{ik}$ as a remote pre or post-operation (Figure 3.10). We sort $PPCS_i$ on the op-ID column using *Counting Sort* algorithm to enable fast access to the list [36]. Once the *CP-PP-Alert-CP(op_{ik})* message is received by CA_j, CA_j notifies C_j's members using the PPMS$_j$ list.

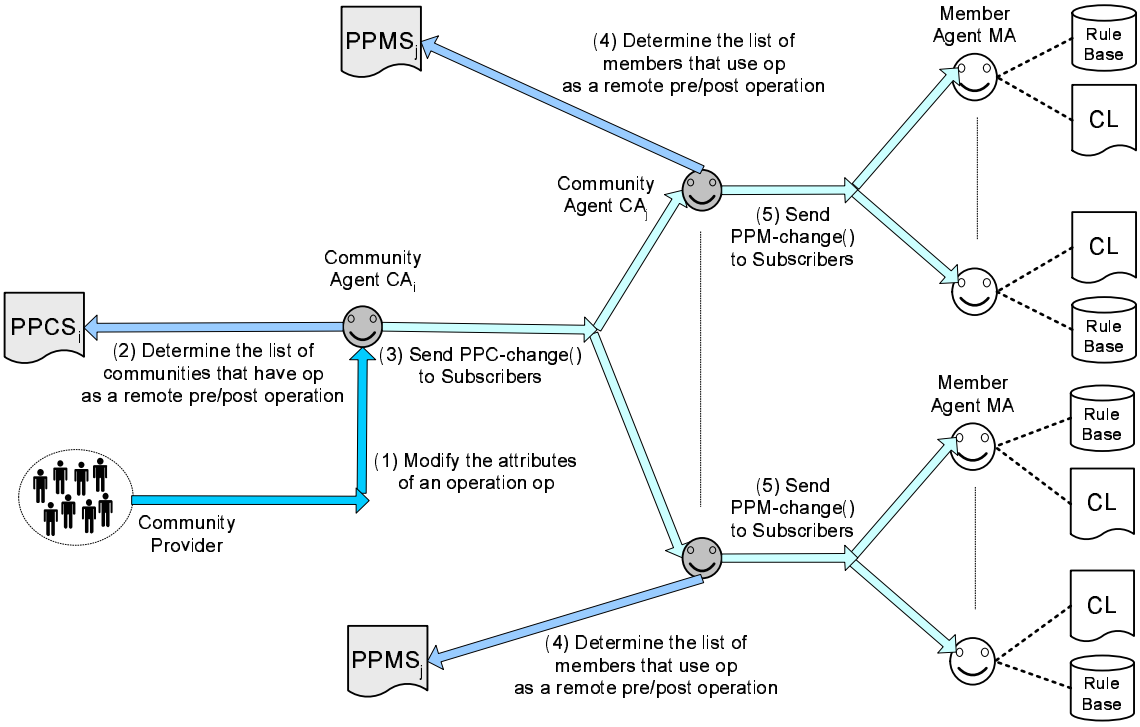

Figure 3.10: Propagating Changes Initiated by Community Providers to their Peers

PPMS$_j$ (*Pre and Post-operation Member Subscription*) contains the list of C_j's members that use the operations op_{ik} of other communities C_i as remote pre or

post-operations. This list is created at C_j's definition time. It is used to notify members about changes that occur in their pre and post-operations. Each entry in PPMS_j contains the ID (CA-ID) of the community agent CA_i, the ID (PP-ID) of the remote pre or post-operation op_{ik}, and the ID (MA-ID) of the agent of a C_j's member that uses op_{ik} as a remote pre or post-operation. CA_j accesses the PPMS_j list and sends a notification message *CP-PP-Alert-SP(op_{ik})* (SP stands for Service Provider) to each member agent MA-ID that uses op_{ik} as a remote pre-operation (Figure 3.10). We sort $PPMS_i$ on the CA-ID column using *Counting Sort* algorithm to enable fast access to the list [36]. To prevent references to members that imported "obsolete" generic operations, CA_j assigns the value "unavailable" to the *status* of MA-ID entries in ML_j list. We only consider entries in which MA-ID imported operations that have op_{ik} as a remote pre or post-operation.

C_j's provider may remove the remote pre/post operation op_{ik} from its generic operation definitions. To propagate this change, CA_j removes from PPMS_j the entry that corresponds to CA_i and op_{ik}. CA_j also sends a *CP-PP-Remove-CP(op_{ik})* to CA_i. Upon reception of this message, CA_i removes from PPCS_i the entry corresponding to CA_j and op_{ik}. CA_i will hence send no *CP-PP-Alert-CP(op_{ik})* messages to CA_j since C_j does not use op_{ik} as a remote pre- or post-operation. Figure 3.11 summarizes the algorithm executed by CA_i for managing the modification of generic operations. The *Update_Member()* function allows the update of the description of a community description within the service registry.

3.4.2 Propagating Changes Initiated by Service Providers

The provider of a Web service *WS-ID* may initiate changes that should be sent to their communities (Figure 3.12). A community C_i is notified about a change if the change is made on an operation imported from C_i. All changes are introduced through member agents and automatically forwarded to community agents which reflect those changes at the community level. We define the following service providers changes:

(00) `Community Agent` CA_i `Algorithm` {
(01) `Upon Reception of`
(02) **Modify (G-op-ID,ModifySet) statement From** CP_i
(03) `Update_Member (`C_i`,G-op-ID,ModifySet);`
(04) `for each` ML_i[k].op-ID |
(05) (ML_i[k].op-ID=G-op-ID)∧(ModifySet-ML_i[k].NotImp≠∅)
(06) `do` ML_i[k].status = "unavailable";
(07) `Send CP_Modify (G-op-ID,`ML_i`[k].WS-ID,ModifySet) To` ML_i[k].MA-ID;
(08) `if` ModifySet `contains` pre or post-operation attributes
(09) `then For each` $PPCS_i$[k].op-ID | $PPCS_i$[k].op-ID==G-op-ID
(10) `do Send CP-PP-Alert-CP`(G-op-ID) `To` $PPCS_i$[k].CA-ID;
(11) **CP_PP_Alert_CP (G-op-ID) message From** CA_j
(12) `for each` $PPMS_i$[k] | $PPCS_i$[k].op-ID==G-op-ID
(13) `do` Let p be the entry in ML_i | ML_i[p] has G-op-ID as pre/post-operation
(14) ML_i[p].status = "unavailable";
(15) `Send CP-PP-Alert-SP`(G-op-ID) `To` *ML_i[p].MA-ID*;
(16) **CP_PP_Remove_CP (G-op-ID) message From** CA_j
(17) Let k be | $PPCS_i$[k].op-ID==G-op-ID ∧ $PPCS_i$[k].CA-ID==CA_j;
(18) `remove` entry k from $PPCS_i$;
(19) }

Figure 3.11: Generic Operation Modification: Community Agent Algorithm

- *Modifying operations* – *WS-ID*'s provider may modify attributes (e.g., remove a message parameter) of a previously imported operation through modification statements defined in *WS-ID*'s agent *MA* (Figure 3.12, step 1). A modification statement includes the *G-op-ID* of the operation to be modified, *WS-ID* of the service that imported it, and the importing statement *I-statement* used to do the modification. Once *MA* gets a modification statement, it sends an *SP-Modify(G-op-ID,WS-ID,IType,ModifySet)* message to C_i's agent CA_i (Figure 3.12, step 2). ModifySet is defined as in *CP-Modify()* messages. *IType* gives the type of I-statement ("projection", "extension", "adjust add", and

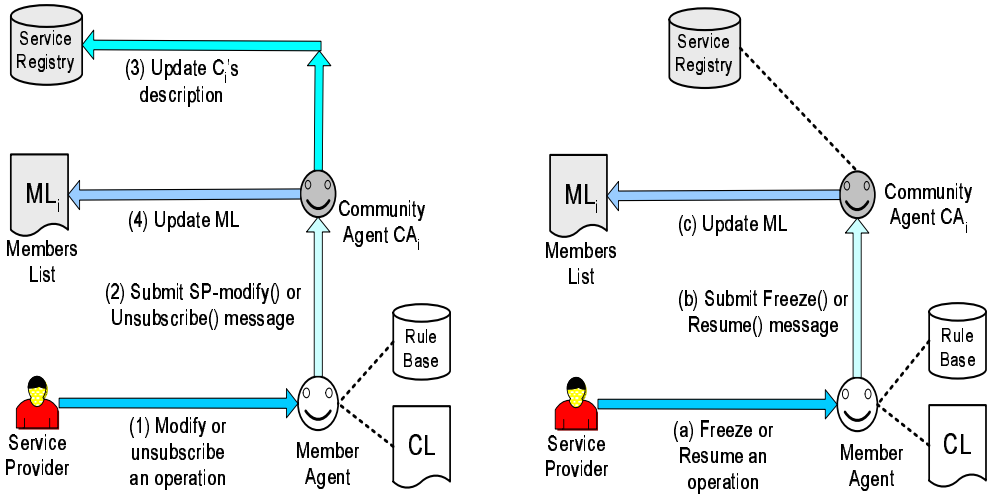

Figure 3.12: Propagating Changes Initiated by Service Providers

"adjust delete"). Upon reception of the *SP-Modify()* message, CA_i updates C_i's description (Figure 3.12, step 3). If the message concerns a projection statement, CA_i updates the *NotImp* column in the ML_i 's entry that correspond to *G-op-ID* and *WS-ID* (Figure 3.12, step 4). It assigns the content of *ModifySet.Att* to this column. If the message concerns an "adjust delete" statement on a remote pre or post operation op_{jl}, then CA_i removes the entry in PPMS_i corresponding to CA_j, op_{jl}, and *WS-ID*'s agent. If the message concerns an "adjust add" statement on a remote pre or post operation op_{jl}, then CA_i adds a new entry in PPMS_i with the values CA_j, op_{jl}, and *WS-ID*'s agent *MA-ID*.

- *Freezing operations* – An imported operation may be "available", "unavailable" (e.g., due to network problem), or "unsubscribed". *WS-ID*'s provider may temporarily make an imported operation *G-op-ID* by executing the *Freeze* statement (Figure 3.12, step a) defined in *MA*. The statement includes the G-op-ID of the operation to be frozen and the WS-ID of the service that imported it. As a consequence, *MA* sends an *SP-Freeze(G-op-ID,WS-ID)* message to CA_i (Figure 3.12, step b). CA_i then assigns the "unavailable" value to the

```
(00) Member Agent MA Algorithm {
(01) Upon Reception of
(02) Modify (G-op-ID,WS-ID,I-Statement) statement From SP
(03) Parse the importing statement I-Statement;
(04) Get ModifySet From I-Statement;
(05) IType = type of the I-Statement;
(06) CA_i = Lookup_community(CL,G-op-ID);
(07) Send SP_Modify (G-op-ID,WS-ID,IsProjection,ModifySet) To CA_i;
(08) Freeze (G-op-ID,WS-ID) statement From SP
(09) CA_i = Lookup_community(CL,G-op-ID);
(10) Send SP_Freeze (G-op-ID,WS-ID) To CA_i;
(11) Resume (G-op-ID,WS-ID) statement From SP
(12) CA_i = Lookup_community(CL,G-op-ID);
(13) Send SP_Resume (G-op-ID,WS-ID) To CA_i;
(14) Unsubscribe (G-op-ID,WS-ID) statement From SP
(15) CA_i = Lookup_community(CL,G-op-ID);
(16) Send SP_Unsubscribe (G-op-ID,WS-ID) To CA_i;
(17) }
```

Figure 3.13: Changes Issued by Service Providers: Member Agent Algorithm

status column of the ML_i 's entry that correspond to *G-op-ID* and *WS-ID* (Figure 3.12, step c).

- *Resuming operations* – *WS-ID*'s provider may re-activate an operation G-op-ID that has previously been frozen through the *Resume* statement defined in *MA*. The statement includes a reference to WS-ID and G-op-ID. As a consequence, *MA* sends an *SP-Freeze(G-op-ID,WS-ID)* to G-op-ID's community agent CA_i (Figure 3.12, step b). CA_i then assigns the "available" value to the *status* column of the ML_i 's entry that correspond to *G-op-ID* and *WS-ID* (Figure 3.12, step c).

- *Unsubscribing operations* – *WS-ID*'s provider may decide not to support an imported operation any more. For that purpose, it executes *MA*'s *Unsubscribe* statement (Figure 3.12, step 1). *MA* informs CA_i about this change (Figure 3.12, step 2) by sending an *Unsubscribe(G-op-ID,WS-ID)* message to it. CA_i then updates C_i's description (Figure 3.12, step 3). It also assigns the "unsubscribed" value to the *status* of the ML_i 's entry that correspond to *G-op-ID* and *WS-ID* (Figure 3.12, step 4). CA_i periodically checks the *status* column in ML_i and remove all entries associated with unsubscribed operations.

Figures 3.13 and 3.14 depict the algorithm executed by community and member agents for managing changes initiated by service providers. The *Lookup_Community()* function (lines 14) is executed on the communities list *CL*. It returns the CA_i's ID of the community C_i to which the operation G-op-ID belongs.

Each member's agent MA_j also needs to react to change notifications sent by a community agent via *CP-Modify(G-op-ID,WS$_j$,ModifySet)* or *CP-PP-Alert-SP(op_{ik})* messages. The actions to be performed by MA_j as a result of change notification are captured using ECA (*Event Condition Action*) rules [31]. Briefly, the basic semantics of an ECA rule is as follows: when an event occurs, an action is executed if the corresponding condition is true. Event-driven systems are becoming the paradigm of choice for organizing many classes of loosely coupled and dynamic applications. Members react to changes using their own change control policies via local rule specified in their agent. Hence, the reaction to changes can be customized to the peculiarities of each member. Below is an example of ECA rule specified within a member agent:

Rule R_1

Event CP-Modify(G-op-ID,WS_j,ModifySet)

Condition Source = $C_1 \wedge$ *pre-operations* $\in$ *Modify.Att*

Action <notify the service provider to change its internal business logic>;

send *Resume(G-op-ID,WS$_j$)* message to **source**

(00) `Community Agent` CA_i `Algorithm` {
(01) `Upon Reception of`
(02) **SP_Modify (G-op-ID,WS-ID,IType,ModifySet) message From** MA-ID
(03) `Update (`C_i`,G-op-ID,ModifySet)`;
(04) Let k be the entry in ML_i |
(05) ML_i[k].op-ID==G-op-ID $\wedge$ ML_i[k].WS-ID==WS-ID;
(06) `Case` IType
(07) "Projection": ML_i[k].NotImp = Parameters(G-op-ID) - ModifySet.Att;
(08) "Adjust Delete": `if` ModifySet `contains` pre or post-operations
(09) `then remove` from PPMS$_i$ the entry related to G-op-ID and MA-ID;
(10) "Adjust Add": `if` ModifySet `contains` pre or post-operations
(11) `then add` to PPMS$_i$ an entry for G-op-ID and MA-ID
(12) **SP_Freeze (G-op-ID,WS-ID) message From** MA-ID
(13) Let k be the entry in ML_i |
(14) ML_i[k].op-ID==G-op-ID $\wedge$ ML_i[k].WS-ID==WS-ID;
(15) ML_i[k].status = "unavailable";
(16) **SP_Resume (G-op-ID,WS-ID) message From** MA-ID
(17) Let k be the entry in ML_i |
(18) ML_i[k].op-ID==G-op-ID $\wedge$ ML_i[k].WS-ID==WS-ID;
(19) ML_i[k].status = "available";
(20) **SP_Unsubscribe (G-op-ID,WS-ID) message From** MA-ID
(21) Let k be the entry in ML_i |
(22) ML_i[k].op-ID==G-op-ID $\wedge$ ML_i[k].WS-ID==WS-ID;
(23) ML_i[k].status = "unsubscribed';
(24) }

Figure 3.14: Changes Issued by Service Providers: Community Agent Algorithm

R_1 states that whenever the member receives a change notification issued by C_1's provider and if the change concerns the *pre-operations* attribute, the service provider should reflect the change by modifying its internal business logic. The member agent then sends a *Resume(G-op-ID,*WS_j*)* message to C_1's agent. The

message is a confirmation that the member has locally reflected the changes done by CP_1. Figure 3.15 summarizes the change reaction algorithm executed by member agents.

```
(00) Member Agent MA Algorithm {
(01) Upon Reception of
(02) CP_Modify (G-op-ID,WS-ID,ModifySet)
(03) for each rule=(event,condition,actions) in the rule base RB
(04)   | event==CP_Modify(G-op-ID,WS-ID,ModifySet) ∧ condition==.T
(05)   do Execute actions;
(06) CP_PP_Alert_SP(G-op-ID) From CA_i
(07) for each rule=(event,condition,actions) in the rule base RB
(08)   | event== CP_PP_Alert_SP(G-op-ID) ∧ condition==.T
(09)   do Execute actions; (10) }
```

Figure 3.15: Reaction to Changes Issued by Community Providers: Member Agent Algorithm

Chapter 4

A Composability Model for Semantic Web Services

Existing techniques for Web service composition generally require dealing with low level programming details, thus making the process of composing services demanding for composers [10, 25, 70, 90, 113]. Composers need to identify the way operations are interconnected, services are invoked, and messages are mapped to one another. To deal with this issue, we propose a model, called *composability model*, to check whether operations can be combined together, hence avoiding unexpected failures at run time. *Composability* refers to the process of checking if Web service operations to be composed can actually interact with each other [12].

The proposed model is defined by a set of rules called *composability rules*. Each rule compares a specific feature of interacting Web services. Since those features are relevant to different aspects of Web services (syntactic, semantic, behavioral, and qualitative), we organize composability rules into several *levels*. Each rule specifies the constraints and requirements for checking horizontal, vertical, and hybrid composability. We also define rules, called *business process composability* rules, to check whether a given composition of Web service operations provides a *value-added*.

Due to the sheer heterogeneity of Web services, it is not always possible to find services that are "fully" composable. Composers may, in this case, select services

that are "partially" composable and then "adapt" their requests based on the results returned by the composability process. For that purpose, we introduce the notions of *composability degree* and *τ-composability* to cater for partial and total composability.

In this chapter, we give a detailed description of the proposed composability model [83, 79]. In Section 4.1, we define the basic concepts of our model namely, composability *rule*, *level*, *degree*, and *τ-composability*. In Section 4.2, we describe syntactic composability rules. In Section 4.3 and 4.4, we define composability rules for comparing static and dynamic semantic features of Web services. Section 4.5 is devoted to composability rules related to quality of operation features. In Section 4.6, we describe business process composability.

4.1 The Proposed Model for Composability

The proposed model for composability contains a set of rules organized into four levels (Figure 4.1). Each rule CR_{pq} at a level CL_p (p=0,4) compares interacting operations with respect to a specific *attribute* within CL_p. The first level CL_0 compares syntactic attributes such as the number of parameters (CR_{00}) and bindings (CR_{01}). The second level CL_1 compares static semantic attributes. We define two groups of rules at this level. The first group (CR_{10},...,CR_{14}) compares the static semantics of messages. The second group (CR_{15},...,CR_{18}) compares the static semantics of operations. The third level CL_2 compares dynamic semantic attributes (CR_{20},...,CR_{24}).

The fourth composability level CL_3 focuses on quality of operation attributes. It contains three groups of rules. The first group (CR_{30},...,CR_{33}) compares security attributes. The second group (CR_{34},...,CR_{36}) checks business attributes. The third group (CR_{37},...,CR_{39}) of qualitative composability rules deals with runtime attributes. The fourth level CL_3 deals with business process composability. It contains rules (CR_{40},CR_{41},CR_{42}) that checks the *soundness* of a composite service, that is, whether that service provides a *value-added*.

There are two main differences between business process composability rules

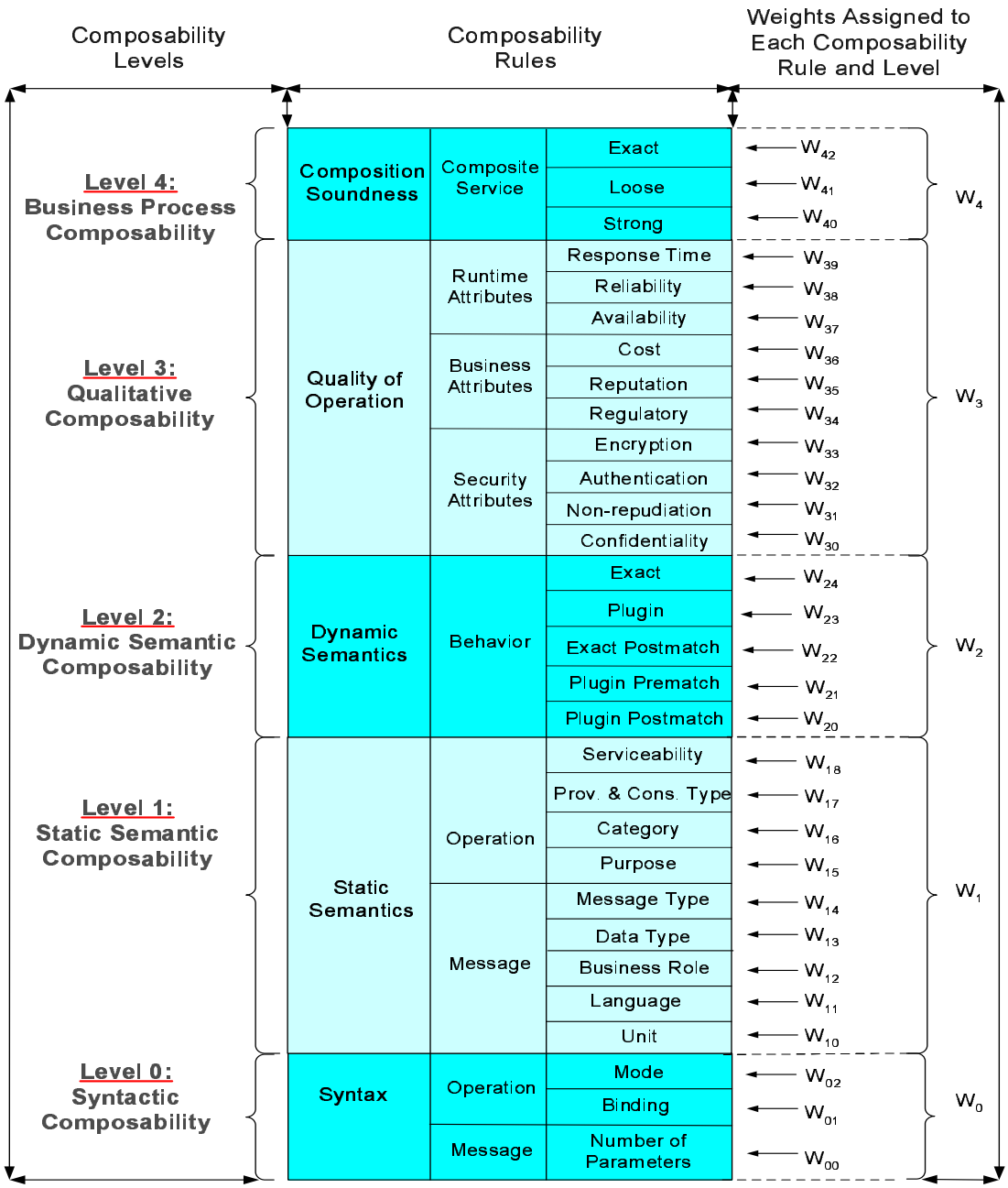

Figure 4.1: Composability Stack

and the rules at lower levels. First, business process composability deals with features that are related to composite services while the other levels' rules compare operation and message attributes of participant services. Second, business process composability checks composability *a posteriori*, i.e., after a given composition was performed. Rules at lower levels check composability *a priori*, i.e., while composing Web services.

4.1.1 Horizontal, Vertical, and Hybrid Composition

We define three ways of combining Web service operations: *horizontal*, *vertical*, and *hybrid*. *Horizontal* composition models a "supply chain"-like combination of operations. Let op_{ik} and op_{jl} be two operations that are horizontally composed. We call op_{ik} and op_{jl} *source* and *target* operations respectively. Figure 4.2 depicts horizontal composition in the case where op_{ik}'s mode is *In/Out* (case (a)) and *Out/In* (case (b)). The operation op_{ik} is first executed, followed by op_{jl}'s execution. op_{ik}'s messages are used to *feed* op_{jl}'s input message. Let $\mathcal{M}$ be a set of messages and $Input_{jl}$ the input message of op_{jl}. We say that $\mathcal{M}$ *feeds* $Input_{jl}$ if parameters in $\mathcal{M}$'s messages are used as $Input_{jl}$'s parameters. As depicted in Figure 4.2, In_{ik} and Out_{ik} messages feed In_{jl}.

As example of the horizontal composition, assume that op_{ik} provides translation from English to French and op_{jl} provides translation from French to Chinese. The operations op_{ik} and op_{jl} may be horizontally composed to provide translation from English to Chinese. In this case, the result of op_{ik} (French word) is used as input by op_{jl}. The precedence relationships between op_{ik}'s and op_{jl}'s states are given below:

$$Start_{ik} \longrightarrow_t Active_{ik} \longrightarrow_t End_{ik} \longrightarrow_t Start_{jl} \longrightarrow_t Active_{jl} \longrightarrow_t End_{jl}$$

Vertical Composition models the "subcontracting" of an operation op_{jl} by another operation op_{ik}. Figure 4.3 depicts vertical composition in the case where op_{ik}'s mode is *In/Out* (case (a)) and *Out/In* (case (b)). Let us consider the first case where

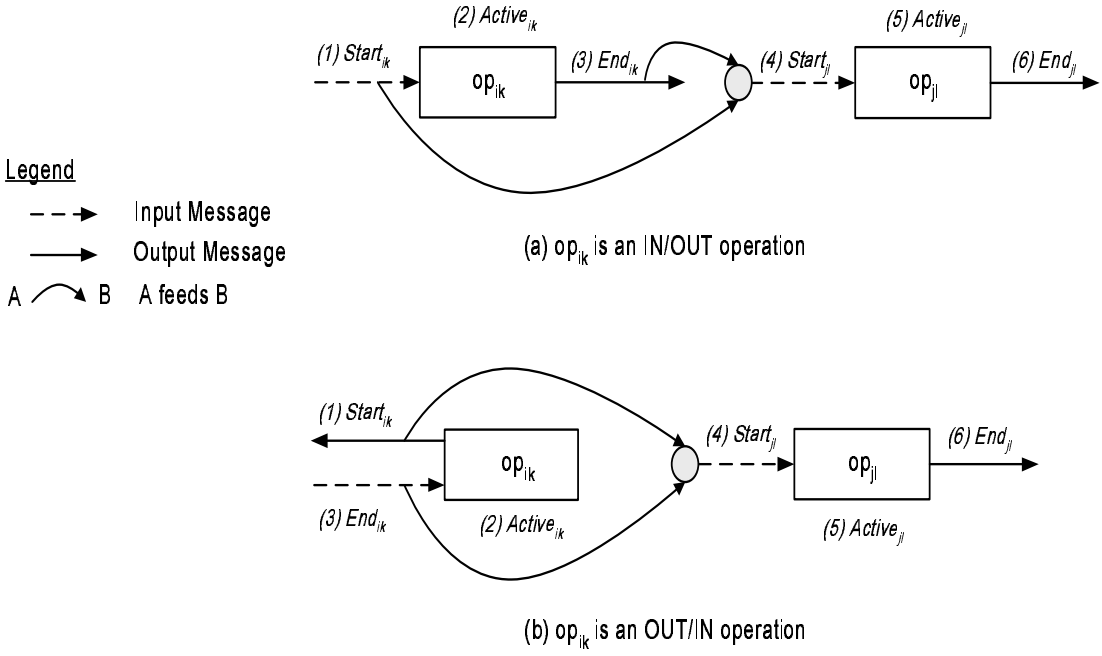

Figure 4.2: Horizontal Composition

op_{ik}'s mode is *In/Out*. Whenever op_{ik} is invoked, it transparently sends an input message to op_{jl}. op_{jl} then performs the requested function on behalf of op_{ik} and returns an output message to op_{ik}. op_{ik} will finally send the results to its invoker. Assume now that op_{ik}'s mode is *Out/In*. op_{ik} starts its execution by invoking op_{jl}. After op_{jl} terminates its execution, it sends results to op_{ik} which receives them as an input message.

An example of vertical composition is that of a personal computers (PC) reseller offering an operation `requestQuotes`. This operation allows customers to request quotes. The execution of `requestQuotes` requires the invocation of another operation provided by a PC manufacturer to get the latest prices and delivery schedule. We summarize below the precedence relationships between op_{ik}'s and op_{jl}'s states:

$$Start_{ik} \longrightarrow_t Start_{jl} \longrightarrow_t Active_{jl} \longrightarrow_t End_{jl} \longrightarrow_t End_{ik}$$

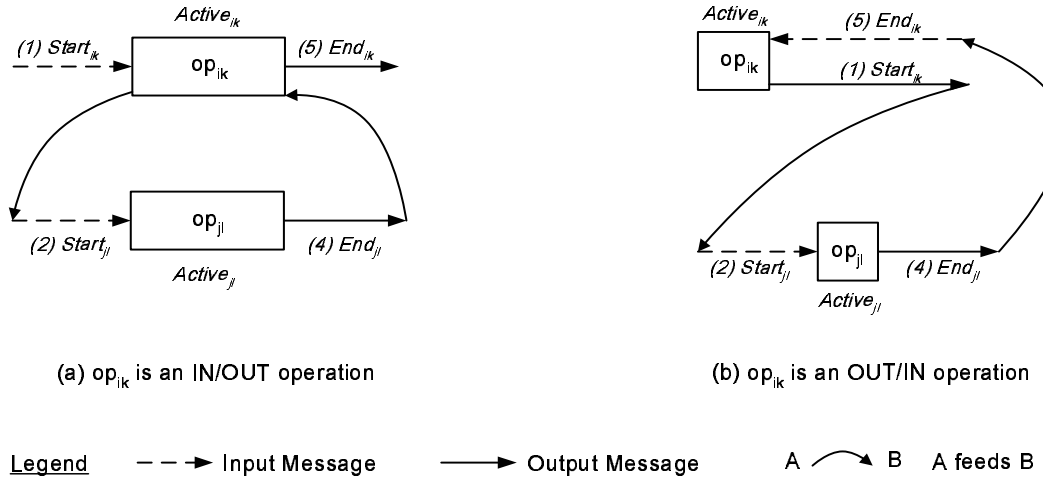

Figure 4.3: Vertical Composition

and $Start_{ik} \longrightarrow_t Active_{ik} \longrightarrow_t End_{ik}$

A composite service *CS* may include operations that are horizontally composed and others that are vertically composed. We refer to this type of composition as *hybrid composition*. The first example in Figure 4.4 depicts a hybrid composition of three operations op_1, op_2, and op_3. In this case, op_1 is vertically composed with op_2. This means that op_1 "subcontracts" from op_2. Additionally, op_1 is horizontally composed with op_3, meaning that op_1's input and output messages feed op_3's input. The second example of Figure 4.4 depicts another hybrid composition of op_1, op_2, op_3, and op_4. op_1 subcontracts from two horizontally composed operations op_2 and op_3. Furthermore, op_1 is horizontally composed with op_4.

4.1.2 Properties of a Composability Rule

Composability rules check whether two operations op_{ik} and op_{jl} are composable from different perspectives. We identify five *properties* that characterize composability rules: *level*, *granularity*, *attribute*, *symmetry*, and *composition type*.

Table 4.1 summarizes the different composability rules and their properties. We

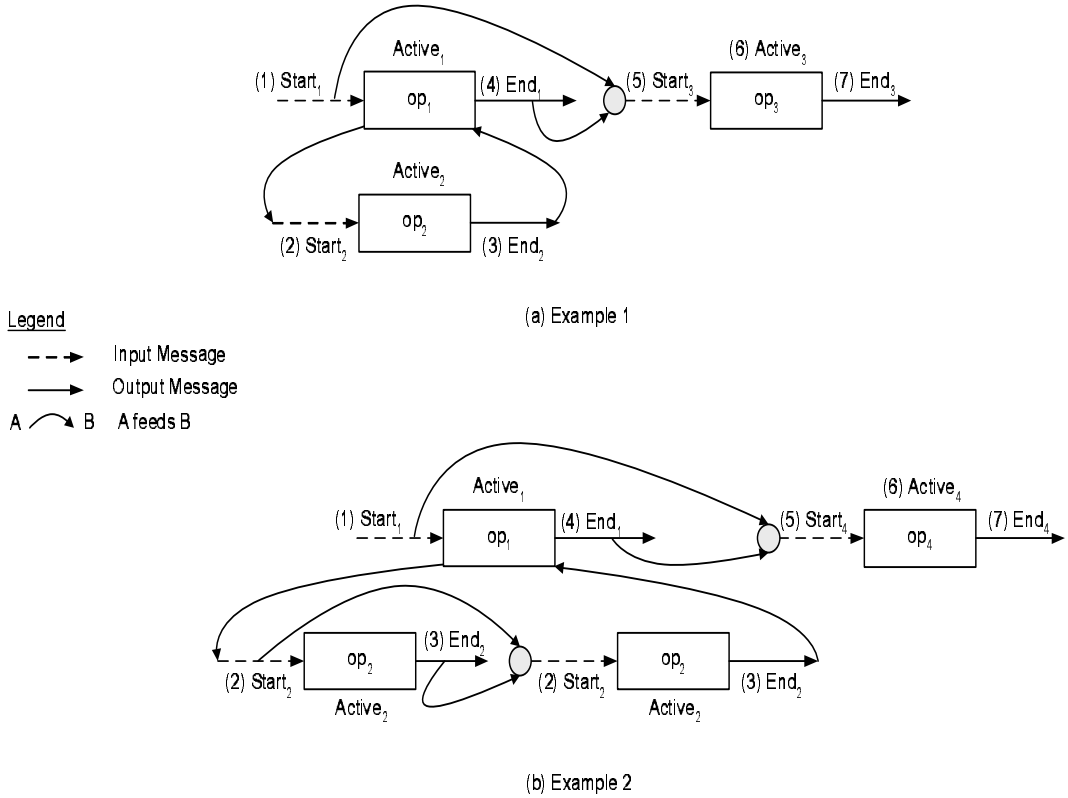

Figure 4.4: Hybrid Composition

organize these rules into four *levels*: *syntactic*, *static semantic*, *dynamic semantic*, and *qualitative*. These levels check composability at the message and operation *granularity*. Each rule in a given level compares a specific pair of *attributes* of op_{ik} and op_{jl} (e.g., mode, binding, purpose, and cost). A rule is either *symmetric* or *asymmetric*. It is *symmetric* if the order in which it is checked (from op_{ik} to op_{jl} or op_{jl} to op_{ik}) is not important. This is in contrast with *asymmetric* rules; if an asymmetric rule is satisfied from op_{ik} to op_{jl} then it is not necessarily satisfied from op_{jl} to op_{ik}. Finally, a rule may be applicable for horizontal composition (e.g., *Plugin Prematch*), vertical composition (e.g., *Exact Postmach*), both (e.g., *purpose* and *category*), or hybrid (e.g., composition soundness).

Level	Granularity	Attribute	Symmetry	Horizontal	Vertical	Hybrid
Syntactic	Message	Number of Parameters	√	√	√	
	Operation	Binding	√	√	√	
		Mode		√	√	
Static Semantics	Message	Data type	√	√	√	
		Unit	√	√	√	
		Language	√	√	√	
		Business Role	√	√	√	
		Message Type	√		√	
	Operation	Purpose		√	√	
		Category		√	√	
		Prov. & Cons. Type		√	√	
		Serviceability	√		√	
Dynamic Semantics	Operation	Plugin Postmatch			√	
		Plugin Prematch		√		
		Exact Postmatch	√		√	
		Plugin			√	
		Exact	√		√	
Qualitative	Operation	Confidentiality			√	
		Non-repudiation	√		√	
		Authentication	√		√	
		Encryption	√		√	
		Regulatory			√	
		Reputation			√	
		Cost			√	
		Availability			√	
		Reliability			√	
		Response Time			√	
Business Process	Composite	Exact				√
		Loose				√
		Strong				√

√: The rule has the corresponding property.

Table 4.1: Composability Rules Properties

4.1.3 Composability Degree

Composers may have different views on composability rules. One may, for example, give higher importance to syntactic composability while another may focus more on semantic rules. To capture this aspect, we associate a *weight* W_p to each level CL_p. We also define a *weight* W_{pq} for each rule CR_{pq} in that level. A weight is an estimate of the significance of the corresponding level or rule from the composer's point of view. Composers assign a weight to each level and rule. This gives composers more control over the composability process. The higher is a weight, the more important is the corresponding level or rule. W_p (≥ 0 and ≤ 1) compares CL_p to the other levels

in terms of their importance. The total of weights assigned to the different levels equals 1. Similarly, W_{pq} (≥ 0 and ≤ 1) compares CR_{pq} to the other rules at level CL_p. The total of weights assigned to rules within a level equals 1. Formally, the different weights must respect the following constraints, where $|CL_p|$ is the number of rules at level p:

1. $\forall$ p,q: $(0 \leq W_p \leq 1) \wedge (0 \leq W_{pq} \leq 1)$; and
2. $(\sum_{p=0}^{4} W_p = 1) \wedge (\forall$ p: $\sum_{q=0}^{|CL_p|-1} W_{pq} = 1)$.

Due to the heterogeneity of Web services, it is not always possible to find operations that are "fully" composable with source operations. Composers may, in this case, select operations that are "partially" composable with their operation and then, "adapt" their operations based on the results returned by the composability process. For example, the composer may modify the data type of an operation parameter if it is not compatible with the data type of the corresponding target operation's parameter. For that purpose, we introduce the notion of composability *degree* between source and target operations.

The *degree* of op_{ik} and op_{jl} gives the ratio of composability rules that are satisfied between op_{ik} and op_{jl}. It takes its values from 0 to 1 (≥ 0 and ≤ 1). We define a boolean function $satisfied_{pq}(op_{ik},op_{jl})$ that returns true iff the rule CR_{pq} is satisfied between op_{ik} and op_{jl}. To reflect the composer's view on each rule CR_{pq}, we adjust the value returned by the function $satisfied_{pq}(op_{ik},op_{jl})$ with the weight W_{pq}. The degree at a given level CL_p is obtained by adding the adjusted values returned by the function *satisfied* applied on each CL_p's rule. Once the degree at CL_p is computed, we adjust it with the weight W_p assigned to CL_k. As specified below, the *degree* of op_{ik} and op_{jl} is obtained by summing composability degrees at all levels CL_p (p=0,4):

$$\text{Degree } (op_{ik},op_{jl}) = \sum_{p=0}^{4}(W_p \times (\sum_{q=0}^{|CL_p|-1} (W_{pq} \times satisfied_{pq}(op_{ik},op_{jl}))))$$

During a composition process, the composer assigns weights to each level and rule by providing a vector called *level weight* (LW) and matrix called *rule weight*

(RW). The element LW_p (p=0,3) gives the weight assigned to level CL_p. The element CW_{pq} gives the weight assigned to rule CR_{pq}. If a rule CR_{pq} is undefined, then CW_{pq} is automatically assigned the value 0. For example, since the rules CR_{03}, ...,CR_{09} are not defined, the entries RW_{03}, ...,RW_{09} are equal to 0. As an illustration, assume that case officer *John* provides the weights given below to his source operation `scheduleTransportation`:

$$LW = \begin{pmatrix} 0.2 & 0.3 & 0.2 & 0.3 & 0.2 \end{pmatrix}$$

$$RW = \begin{pmatrix} 0.25 & 0.25 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.15 & 0.15 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.6 & 0 & 0 & 0 & 0 & 0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This example shows that the composer gives more importance to static semantic composability since the corresponding weight is greater than the other levels' weights. Among dynamic semantic properties, the composer gives higher priority to the *exact* behavioral rule. Assume now that `scheduleTransportation` is compared to `fasTransportation` using the composability model and that all composability rules are satisfied, except the ones that correspond to the following rules: CR_{00}, CR_{01}, CR_{02}, CR_{10}, CR_{15}, CR_{16}, CR_{20}, CR_{21}, CR_{22}, CR_{23}, CR_{24}, CR_{33}, CR_{35}, CR_{36}. The composability degree is computed as follows:

Degree (`scheduleTransportation`,`fasTransportation`)
$= 0.2 \times (0.25 + 0.25 + 0.3) + 0.3 \times (0.1 + 0.1 + 0.1)$

$$+ 0.2 \times (0.1 + 0.1 + 0.1 + 0.1 + 0.6) + 0.1 \times (0.1 + 0.2 + 0.2) = 0.57 = 57\%.$$

Based on the degree of op_{ik} and op_{jl}, we can decide about the composability of those operations. If *degree* = 0 then no rule is satisfied and the operations are *non composable*. If *degree* = 1 then all composability rules are satisfied and the operations are *fully* composable. Otherwise, only a subset of rules are satisfied. In this case, op_{ik} and op_{jl} are *partially* composable:

If degree (op_{ik},op_{jl}) = 0
  Then op_{ik} and op_{jl} are non composable
  Else If degree (op_{ik},op_{jl}) = 1
    Then op_{ik} and op_{jl} are fully composable
    Else op_{ik} and op_{jl} are partially composable

4.1.4 τ-Composability

Service composers may have different expectations about the composability degree of their operations. For that purpose, composers provide a *composability threshold* τ ($0<\tau\leq 1$). The *threshold* gives the minimum allowed value of a composability degree. All operations op_{jl} so that *degree*(op_{ik},op_{jl}) $\geq \tau$ are potential candidates to be composed with op_{ik}. If the threshold is greater than degree (op_{ik},op_{jl}) then op_{ik} will not be composed with op_{jl}. Assume that the composer assigns the value 0.75 to the threshold (τ=0.75). Since the composability degree of `scheduleTransportation` and `fasTransportation` equals 0.57, then `scheduleTransportation` and `fasTransportation` are not τ-composable. We introduce below the definition of *τ-composability*. This definition compares the composability degree and threshold to decide whether an operation is composable with another:

Definition 4.1 – An operation op_{ik} is *τ-composable* with op_{jl} if *degree*(op_{ik},op_{jl}) $\geq \tau$. ♢

We summarize below (Lemma 1) the relationship between τ-Composability and composability degree. If op_{ik} and op_{jl} are fully composable then *degree*(op_{ik},op_{jl}) = 1 $\geq \tau$. Hence, they are τ-composable for all values of τ. If op_{ik} and op_{jl} are non composable then *degree*(op_{ik},op_{jl}) = 0 $< \tau$. Hence, they are non composable for all values of τ. If the two operations are partially composable, then op_{ik} and op_{jl} may be τ-composable or not depending on the value of τ.

Lemma 4.1 – Let op_{ik} and op_{jl} be source and target operations respectively:

- If op_{ik} and op_{jl} are fully composable then $\forall\ \tau$, op_{ik} and op_{jl} are τ-composable.
- If op_{ik} and op_{jl} are non composable then $\forall\ \tau$, op_{ik} and op_{jl} are not τ-composable.
- If op_{ik} and op_{jl} are partially composable then op_{ik} and op_{jl} τ-composability depends on the value of τ. $\diamondsuit$

4.2 Syntactic Composability Rules

We define two group of syntactic composability rules. The first group contains rules that compare attributes at the operation granularity. Such rules include *binding* and *mode* composability. The second group includes rules that check attributes at the message granularity. Such rules include *PN-Composability* which compares the number of parameters of operation messages.

4.2.1 Composability at the Operation Granularity

The first syntactic attribute that needs to be checked is *binding*. Two operations op_{ik} and op_{jl} may support different binding protocols (e.g., SOAP/HTTP or SOAP/MIME). It is hence important to insure that they "understand" each other at the message format and protocol level. At least one of the protocols expected by an operation must be supported by the other. For example, it would be difficult for an operation that expects to receive messages in SOAP/MIME protocol to interact

with another service that format its messages in SOAP/HTTP. The following rule, called *binding composability*, checks that operations support at least one common binding protocol.

Definition 4.2 – Let op_{ik} and op_{jl} be two operations. We say that op_{ik} and op_{jl} are *binding composable* if $Binding_{ik} \cap Binding_{jl} \neq \emptyset$. $\diamondsuit$

Assume now that op_{ik} and op_{jl} are binding composable. For these operations to be "plugged" together, they must have "composable" modes. *Mode composability* depends on two criteria: the composition type (vertical or horizontal) and the source operation's mode $Mode_{ik}$. We consider the following possible cases:

1. *The composition is vertical and* $Mode_{ik}$ = *"In/Out"* (Figure 4.3-(a)): the input of op_{ik} feeds op_{jl}'s input and the output of op_{jl} feeds op_{ik}'s output. Hence op_{jl} is an *In/Out* operation.

2. *The composition is vertical and* $Mode_{ik}$ = *"Out/In"* (Figure 4.3-(b)): op_{ik} starts by sending an output message which should be received by op_{jl}. After processing the message, op_{jl} replies to op_{ik} through an output message. Hence op_{jl} is an *In/Out* operation.

3. *The composition is horizontal and* $Mode_{ik}$ = *"In/Out"* (Figure 4.2-(a)): op_{ik}'s output message is the result returned by the execution of op_{ik}. It is used in part or whole to invoke op_{jl} by sending an input message to it. Hence op_{jl} is an *In/Out* operation.

4. *The composition is horizontal and* $Mode_{ik}$ = *'Out/In"* (Figure 4.2-(b)): op_{ik}'s input message is the result of a subcontracting performed by op_{ik} (via vertical composition). It is used to invoke op_{jl} by sending an input message to it. Hence op_{jl} is an *In/Out* operation.

As illustrated in our discussion above, the target operation's mode $Mode_{jl}$ should be *In/Out*, independently of the composition type and the model of the source

operation. We give below a formal definition of mode composability:

Definition 4.3 - Let op_{ik} and op_{jl} be two operations. We say that op_{ik} is *mode composable* with op_{jl} if $Mode_{jl}$ = *"In/Out"*. $\diamondsuit$

4.2.2 Composability at the Message Granularity

Syntactic composability also requires comparing op_{ik} and op_{jl} at the message level. There is a need to compare the input and output messages of both operations. For this purpose, we define a composability rule, called *Parameters Number Composability* (*PN-Composability*). *PN-Composability* compares the number of parameters in every pair of "plugged" messages. The definition of *PN-Composability* depends on the composition type and operations' modes. We identify the following four cases:

1. *The composition is vertical Composition and* $Mode_{ik}$ = *"In/Out"*: In_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with Out_{ik} (Figure 4.3-(a)). In_{ik} (Out_{jl}) must provide at least as much information as expected by In_{jl} (Out_{ik}). Hence, the number of parameters in In_{ik} (Out_{jl}) is at least equal to the number of parameters in In_{jl} (Out_{ik}).

2. *The composition is vertical and* $Mode_{ik}$ = *"Out/In"*: Out_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with In_{ik} (Figure 4.3-(b)). The number of parameters in Out_{ik} (Out_{jl}) is at least equal to the number of parameters in In_{jl} (In_{ik}).

3. *The composition is horizontal and* $Mode_{ik}$ = *"In/Out"*: in this case, In_{ik} and Out_{ik} are "plugged" with In_{jl} (Figure 4.2-(a)). Consequently, the total number of parameters in In_{ik} and Out_{ik} should be at least equal to In_{jl}'s.

4. *Horizontal Composition and* $Mode_{ik}$ = *'Out/In"*: As illustrated in Figure 4.2-(b)), this case is similar to case (3).

Based on the discussion above, it is clear that *PN-Composability* is not symmetric. We give below the formal definitions of this rule:

Definition 4.4 – Let op_{ik} and op_{jl} be two operations. We say that op_{ik} is *PN-composable* with op_{jl} if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $|In_{jl}| \leq |In_{ik}|$, and (iv) $|Out_{ik}| \leq |Out_{jl}|$.
2. (i) the composition is vertical, (i) $Mode_{ik}$ = "Out/In", (i) $|In_{jl}| \leq |Out_{ik}|$, and (iv) $|In_{ik}| \leq |Out_{jl}|$.
3. (i) the composition is horizontal and (ii) $|In_{jl}| \leq |In_{ik}| + |Out_{ik}|$. $\diamondsuit$

4.3 Static Semantic Composability Rules

For two operations op_{ik} and op_{jl} to be "plugged" together, they must be semantically "compliant". For example, it would be semantically "incorrect" to interconnect an operation that provides "counseling" with another that checks "eligibility" since these operations have different purposes. Additionally, two message parameters cannot be mapped together if they have diffrent business roles. In the following, we present static semantic rules that deal with composability at the operation and message granularity.

4.3.1 Composability at the Operation Granularity

Operations are described at the static semantics level via *serviceability*, *provider type*, *consumer type*, *category*, and *purpose* attributes (Chapter 3). The first composability rule compares op_{ik}'s and op_{jl}'s *serviceability*. The content of both attributes must be similar if op_{ik} and op_{jl} are vertically composed. For example, an operation providing cash support cannot "subcontract" in-kind operations. Horizontal composition does not require comparing *serviceability* since no operation will "service" the other.

Definition 4.5 - Let op_{ik} and op_{jl} be two vertically composed operations. We say that op_{ik}'s and op_{jl}'s serviceability are composable if $serviceability_{ik} = serviceability_{jl}$.$\diamondsuit$

The second rule compares op_{ik}'s and op_{jl}'s *provider* and *consumer types*. The way this comparison is done depends on the type of composition between op_{ik} and op_{jl}. If the composition is horizontal, op_{ik} and op_{jl} must have at least one common provider type and one common consumer type. For example, if op_{ik} expects to outsource from a federal agency's operation then op_{jl}'s agency should include the type "federal". Additionally, if op_{ik} provides benefits for children and pregnant women then op_{jl} should provide benefits for at least those two groups. If op_{ik} is horizontally composed with op_{jl}, then it should be viewed as a consumer of op_{jl}. Hence, op_{jl}'s consumer type should include at least one value from op_{ik}'s provider types. In the following, we give a formal definition of provider and consumer type composability, called *PC-Composability*:

Definition 4.6 - Let op_{ik} and op_{jl} be two operations. We say that op_{ik} is *PC-Composabile* with op_{jl} if one of the following conditions is true:

1. (i) the composition is vertical, (ii) $provider_{ik} \cap provider_{jl} \neq \emptyset$, and (iii) $consumer_{ik} \cap consumer_{jl} \neq \emptyset$.
2. (i) the composition is horizontal and (ii) $provider_{ik} \cap consumer_{jl} \neq \emptyset$. $\diamondsuit$

Operations' *categories* are compared in the third composability rule. Assume that op_{ik} and op_{jl} are vertically composed. Since op_{ik} is meant to "replace" op_{jl}, the following two conditions should be true: (i) op_{ik}'s and op_{jl}'s domains of interest are similar or synonyms, and (ii) all characteristics (i.e., elements of the specialization attribute) of op_{ik}'s category are provided by op_{jl}'s. For example, assume that op_{ik}'s category provides health insurance for children (i.e., $Name_{ik}$="healthcare" and $Specialization_{ik}$={"children", "insurance"}). The operation op_{jl} should not only deal with healthcare but also *at least* provide insurance for children as well.

Assume now that op_{ik} and op_{jl} are horizontally composed. $Category_{ik}$ and $category_{jl}$ should be defined so that op_{ik} and op_{jl} "can" be combined. This is captured by the *overlapping* attribute of a category. Hence, $category_{ik}$ is composable with $category_{jl}$ if $overlapping_{ik}$ contains $category_{jl}$. This leads us to defining the *category composability* rule. We say that op_{ik} is *category composable* with op_{jl}

if category$_{ik}$ is *compatible with* category$_{jl}$. We give below the definition of *category composability*:

Definition 4.7 – We say that category$_{ik}$ is *composable with* category$_{jl}$ if one of the following two conditions is true:

1. (i) the composition is vertical, (ii) specialization$_{ik}$ $\subseteq$ specialization$_{jl}$, and (iii) (domain$_{ik}$ = domain$_{jl}$) or (domain$_{ik}$ $\in$ synonyms$_{jl}$) or (domain$_{jl}$ $\in$ synonyms$_{ik}$) or (synonyms$_{ik}$ $\cap$ synonyms$_{jl}$ $\neq \emptyset$).
2. (i) the composition is horizontal and (ii) category$_{jl}$ $\in$ overlapping$_{ik}$. $\diamondsuit$

The last rule compares operations' *purposes*. The *purpose composability* rule is defined in the same way as *category composability* where domain$_{ik}$ and domain$_{jl}$ are replaced by function$_{ik}$ and function$_{jl}$ respectively:

Definition 4.8 – We say that purpose$_{ik}$ is *composable with* purpose$_{jl}$ if one of the following two conditions is true:

1. (i) the composition is vertical, (ii) specialization$_{ik}$ $\subseteq$ specialization$_{jl}$, and (iii) (function$_{ik}$ = function$_{jl}$) or (function$_{ik}$ $\in$ synonyms$_{jl}$) or (function$_{jl}$ $\in$ synonyms$_{ik}$) or (synonyms$_{ik}$ $\cap$ synonyms$_{jl}$ $\neq \emptyset$).
2. (i) the composition is horizontal and (ii) category$_{jl}$ $\in$ overlapping$_{ik}$. $\diamondsuit$

4.3.2 Composability at the Message Granularity

Composability requires checking the semantics of op$_{ik}$'s and op$_{jl}$'s messages. The first message composability rule compares message types. This rule is applicable only for vertical composition since horizontal composition does not involve replacing op$_{ik}$'s messages with op$_{jl}$'s or vice versa. Given a vertical composition of op$_{ik}$ and op$_{jl}$, we identify two cases based on the mode of op$_{ik}$. If op$_{ik}$ is *In/Out*, then In$_{ik}$'s (resp. Out$_{ik}$'s) and In$_{jl}$'s (Out$_{jl}$'s) types should be similar. Otherwise, Out$_{ik}$'s (resp. In$_{ik}$'s) and In$_{jl}$'s (Out$_{jl}$'s) types should be similar.

Definition 4.9 - Let op_{ik} and op_{jl} be two vertically composed operations. The message type of op_{ik} is composable with op_{jl}'s if one of the following conditions is true:

1. (i) $mode_{ik}$ = *In/Out*, (ii) $\mathcal{MT}(In_{ik}) = \mathcal{MT}(In_{jl})$, and (iii) $\mathcal{MT}(Out_{ik}) = \mathcal{MT}(Out_{jl})$.
2. (i) $mode_{ik}$ = *Out/In*, (ii) $\mathcal{MT}(Out_{ik}) = \mathcal{MT}(In_{jl})$, and (iii) $\mathcal{MT}(Out_{jl}) = \mathcal{MT}(In_{ik})$. $\diamondsuit$

The second rule, called *Data Type Composability* (*DT-Composabilty*), checks parameters' data types. *DT-Composabilty* is based on the notion of *compatibility* between data types (XML Schema). We define two data type compatibility techniques: *name* and *derivation* compatibility. Name compatibility means that both parameters have the same type. A parameter p_1 is compatible by derivation with p_2 if p_1's data type (e.g., `long`) is a super-type of p_2's (e.g., `integer`).

The definition of *DT-Composabilty* depends on the composition type and operations' modes. We distinguish the following four cases:

1. *The composition is vertical and* $Mode_{ik}$ = *"In/Out"*: In_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with Out_{ik} (Figure 4.3-(a)). The data type of each parameter in In_{jl} (Out_{ik}) should be compatible with the data type of a corresponding parameter in In_{ik} (Out_{jl}).

2. *The composition is vertical and* $Mode_{ik}$ = *"Out/In"*: Out_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with In_{ik} (Figure 4.3-(b)). The data type of each parameter in In_{jl} (resp. In_{ik}) should be compatible with the data type of a corresponding parameter in Out_{ik} (Out_{jl}).

3. *The composition is horizontal and* $Mode_{ik}$ = *"In/Out"*: in this case, In_{ik} and Out_{ik} are "plugged" with In_{jl} (Figure 4.2-(a)). The data type of each parameter in In_{jl} should be compatible with the data type of a corresponding parameter in In_{ik} or Out_{ik}.

4. *The composition is horizontal and* Mode_{ik} = *'Out/In"*: As illustrated in Figure 4.2-(b)), this case is similar to case (3).

Based on the discussion above, it is clear that *DT-Composability* is not symmetric. We give below the formal definition of this rule:

Definition 4.10 – An operation op_{ik} is *DT-composable* with op_{jl} if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \mid p'$ is data type compatible with p, and (iv) $\forall\ p \in \text{Out}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid p'$ is data type compatible with p.
2. (i) the composition is vertical, (ii) $Mode_{ik}$ = "Out/In", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{Out}_{ik} \mid p'$ is data type compatible with p, and (iv) $\forall\ p \in \text{In}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid p'$ is data type compatible with p.
3. (i) the composition is horizontal and (ii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid p'$ is data type compatible with p. $\diamondsuit$

The remaining three rules compare parameters' *business role*, *language*, and *unit* respectively. These rules as defined similarly to DT-composability, except that the data type is replaced by *business role*, *language*, or *unit*, respectively. We give below the rules definitions:

Definition 4.11 – The business role of op_{ik} is composable with op_{jl}'s if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \mid role(p') = role(p)$, and (iv) $\forall\ p \in \text{Out}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid role(p') = role(p)$.
2. (i) the composition is vertical, (ii) $Mode_{ik}$ = "Out/In", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{Out}_{ik} \mid role(p') = role(p)$, and (iv) $\forall\ p \in \text{In}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid role(p') = role(p)$.
3. (i) the composition is horizontal and (ii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid role(p') = role(p)$. $\diamondsuit$

Definition 4.12 – The language of op_{ik} is composable with op_{jl}'s if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $\forall\ p \in In_{jl}\ \exists\ p' \in In_{ik}$ | *language(p')* = *language(p)*, and (iv) $\forall\ p \in Out_{ik}\ \exists\ p' \in Out_{jl}$ | *language(p')* = *language(p)*.
2. (i) the composition is vertical, (ii) $Mode_{ik}$ = "Out/In", (iii) $\forall\ p \in In_{jl}\ \exists\ p' \in Out_{ik}$ | *language(p')* = *language(p)*, and (iv) $\forall\ p \in In_{ik}\ \exists\ p' \in Out_{jl}$ | *language(p')* = *language(p)*.
3. (i) the composition is horizontal and (ii) $\forall\ p \in In_{jl}\ \exists\ p' \in In_{ik} \cup Out_{ik}$ | *language(p')* = *language(p)*. $\diamondsuit$

Definition 4.13 – The unit of op_{ik} is composable with op_{jl}'s if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $\forall\ p \in In_{jl}\ \exists\ p' \in In_{ik}$ | *unit(p')* = *unit(p)*, and (iv) $\forall\ p \in Out_{ik}\ \exists\ p' \in Out_{jl}$ | *unit(p')* = *unit(p)*.
2. (i) the composition is vertical, (ii) $Mode_{ik}$ = "Out/In", (iii) $\forall\ p \in In_{jl}\ \exists\ p' \in Out_{ik}$ | *unit(p')* = *unit(p)*, and (iv) $\forall\ p \in In_{ik}\ \exists\ p' \in Out_{jl}$ | *unit(p')* = *unit(p)*.
3. (i) the composition is horizontal and (ii) $\forall\ p \in In_{jl}\ \exists\ p' \in In_{ik} \cup Out_{ik}$ | *unit(p')* = *unit(p)*. $\diamondsuit$

Based on the previous definitions for static semantic composability at the message granularity, we define the notion of composability between message parameters. Let p_1 and p_2 be two message parameters. We say that p_1 is composable with p_2 if (i) p_1 is DT-composable with p_2, (ii) p_1's and p_2's roles, languages, and units are composable. We also define a rule, called *message composability*, that compares the static semantic attributes of source and target operations' messages.

Definition 4.14 – An operation op_{ik} is *message composable* with op_{jl} if one of the following three conditions is true:

1. (i) the composition is vertical, (ii) $Mode_{ik}$ = "In/Out", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \mid p'$ is *data type compatible* with p, *role(p') = role(p)*, *language(p') = language(p)*, *unit(p') = unit(p)*, and (iv) $\forall\ p \in \text{Out}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid p'$ is *data type compatible* with p, *role(p') = role(p)*, *language(p') = language(p)*, *unit(p') = unit(p)*.
2. (i) the composition is vertical, (ii) $Mode_{ik}$ = "Out/In", (iii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{Out}_{ik} \mid p'$ is *data type compatible* with p, *role(p') = role(p)*, *language(p') = language(p)*, *unit(p') = unit(p)*, and (iv) $\forall\ p \in \text{In}_{ik}\ \exists\ p' \in \text{Out}_{jl} \mid p'$ is *data type compatible* with p, *role(p') = role(p)*, *language(p') = language(p)*, *unit(p') = unit(p)*.
3. (i) the composition is horizontal and (ii) $\forall\ p \in \text{In}_{jl}\ \exists\ p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid p'$ is *data type compatible* with p, *role(p') = role(p)*, *language(p') = language(p)*, *unit(p') = unit(p)*. $\diamondsuit$

4.4 Dynamic Semantic Composability

Behavioral composability (*B-Composability*) compares the business logic rules of source and target operations. Let us consider two rules R^n_{ik} =($PreC^n_{ik}$,$PostC^n_{ik}$) and R^m_{jl} =($PreC^m_{jl}$,$PostC^m_{jl}$) that belong to op_i and op_j respectively. B-composability relates $PreC^n_{ik}$ to $PreC^m_{jl}$ and $PostC^n_{ik}$ to $PostC^m_{jl}$. We define several forms of B-composability depending on the relationships between post- and pre-conditions. Each form is an instantiation of the general form of B-composability, called *generic B-composability*, given below. The relations $\mathcal{R}_1$ and $\mathcal{R}_2$ relate preconditions and postconditions, respectively, but need not be the same. They are either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or *nil* (meaning that the corresponding term is dropped). In some cases, we may want to include some information about the postcondition in the precondition clause. To allow this flexibility, we let $\widetilde{PreC}_{ikn}$ be either $PreC^n_{ik}$ or $PreC^n_{ik} \wedge PostC^n_{ik}$ in the generic B-composability rule.

Definition 4.15 – We say that op_{ik} is *Generically B-composable* with op_{jl} if:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |$$
$$(\widetilde{PreC}_{ikn}\ \mathcal{R}_1\ PreC^m_{jl}) \wedge (PostC^m_{jl}\ \mathcal{R}_2\ PostC^n_{ik})$$
$$\text{where: } \mathcal{R}_1, \mathcal{R}_2 \in \{\Leftrightarrow, \Rightarrow, nil\} \quad \text{and}$$
$$(\widetilde{PreC}_{ikn} = PreC^n_{ik}) \text{ or } (\widetilde{PreC}_{ikn} = PreC^n_{ik} \wedge PostC^n_{ik})\ \diamondsuit$$

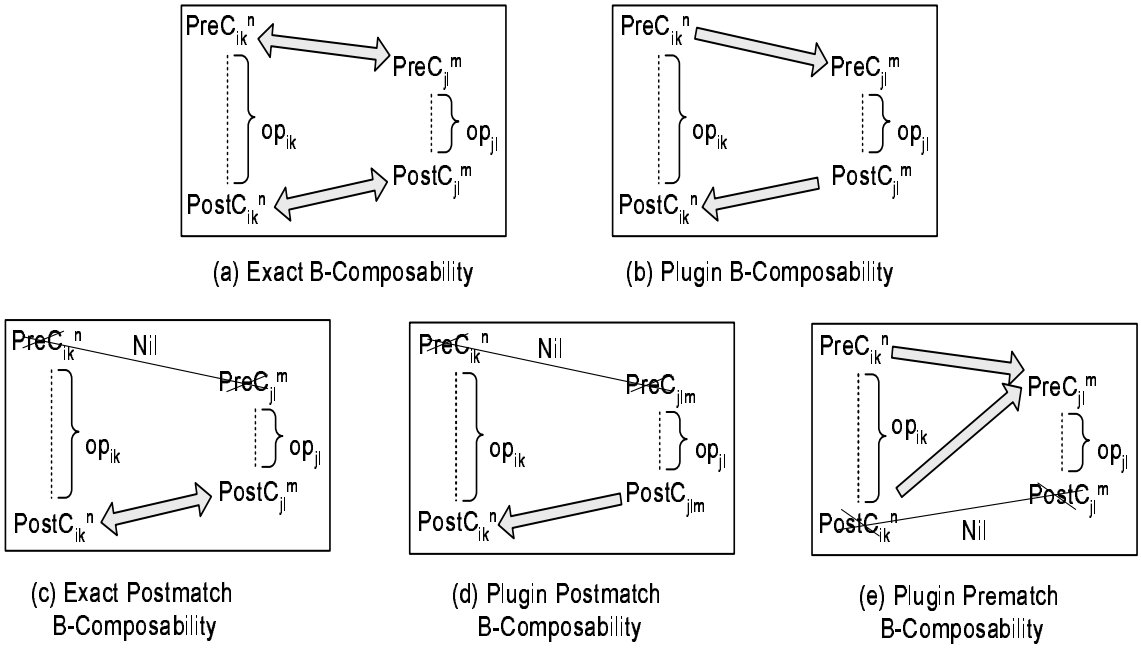

Figure 4.5: B-Composability Rules

Figure 4.5 depicts the different forms of B-Composability rules, beginning with the strongest rule and weakening the rule by relaxing the relations $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$, and *nil*. We also vary $\widetilde{PreC}_{ikn}$ from $PreC^n_{ik}$ to $PreC^n_{ik} \wedge PostC^n_{ik}$. Relaxing the B-composability rule allows making comparison between less closely related operations, but weakens the guarantees about the relationship between the two operations.

Exact B-Composability – Exact B-composability instantiates both $\mathcal{R}_1$ and $\mathcal{R}_2$ to $\Leftrightarrow$ and $\widetilde{PreC}_{ikn}$ to $PreC^n_{ik}$. If exact B-composability holds for two operations, then they are behaviorally equivalent and thus completely interchangeable. Whenever one

operation is used, it could be replaced by the other with no change in observable behavior. As depicted in Figure 4.5.a, this rule is suitable for vertical composition since op_{ik} and op_{jl} are in their *active* state simultaneously. The condition for exact B-composability can be expressed as follows:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |\\ (PreC^n_{ik} \Leftrightarrow PreC^m_{jl}) \wedge (PostC^m_{jl} \Leftrightarrow PostC^n_{ik})$$

Plugin B-Composability – Plugin B-composability relaxes both $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$. It also instantiates $\widetilde{PreC}_{ikn}$ to $PreC^n_{ik}$. The rule R^n_{ik} is matched by any rule R^m_{jl} whose precondition is weaker to allow *at least* all of the conditions that R^n_{ik} allows. The post-condition of R^m_{jl} is stronger than R^n_{ik}'s to provide a guarantee *at least* as strong as R^n_{ik}'s. As depicted in Figure 4.5.b, this rule is suitable for vertical composition since op_{ik} and op_{jl} are in their *active* state simultaneously. The condition for Plugin B-composability is given below:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |\\ (PreC^n_{ik} \Rightarrow PreC^m_{jl}) \wedge (PostC^m_{jl} \Rightarrow PostC^n_{ik})$$

Let us now show that if op_{jl} is "plugged in" op_i, then the business logic of op_j according to R^n_{ik} is preserved. If $PreC^n_{ik}$ holds then $PreC^m_{jl}$ holds because of the first conjunct of Plugin B-composability. Since $PreC^m_{jl} \Rightarrow PostC^m_{jl}$ by definition, we can assume that $PostC^m_{jl}$ will hold after executing the "plugged-in" op_{jl}. Finally, since $PostC^m_{jl} \Rightarrow PostC^n_{ik}$ from the second conjunct of B-composability, $PostC^n_{ik}$ must hold. We say that op_i is behaviorally equivalent to op_j, since we can plug in op_j for op_i and have the same observable behavior. Note, however, that this is not a true equivalence because it is not symmetric; we cannot necessarily plugin op_i for op_j and get the same guarantees.

Exact Postmatch B-Composability – In some cases, composers are concerned only with the effects of operations. For example, a composer may be interested in an operation that provides a social benefit independently of any precondition of that operation. Thus, a useful relaxation of the exact B-composability is to

consider only the postcondition part of the conjunction. *Exact postmatch* is also an instance of the generic B-composability, with $\mathcal{R}_2$ instantiated to $\Leftrightarrow$ and dropping both $\widetilde{PreC}_{ikn}$ and $PreC^m_{jl}$ (Figure 4.5, case (c)). Since only equivalence relationship is used, the exact Postmatch is symmetric. Because $op_{ik}$and op_{jl} are in their *active* state simultaneously (Figure 4.5.c), this rule is suitable for vertical composition. The condition for exact Postmatch B-composability is defined as follows:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |\ PostC^m_{jl} \Leftrightarrow PostC^n_{ik}$$

Plugin Postmatch B-Composability – *Plugin Postmatch* is a relaxation of exact Postmatch where the relationship between postconditions is equal to $\Rightarrow$. Thus, Plugin Postmatch is an instance of the generic B-composability, with $\mathcal{R}_1$ and $\mathcal{R}_2$ instantiated to *nil* and $\Rightarrow$ respectively (Figure 4.5, case (d)). In contrast to exact Postmatch, Plugin Postmatch is asymmetric because of the use of implication between postconditions. As stated in Figure 4.5.d, this rule is suitable for vertical composition since op_{ik} and op_{jl} are in their *active* state simultaneously. We give below the condition for Plugin Postmatch B-composability:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |\ PostC^m_{jl} \Rightarrow PostC^n_{ik}$$

Plugin Prematch B-Composability – Plugin Prematch includes information about op_i's postcondition in the precondition and drops the relationship between postconditions. This is particularly useful to check horizontal composability that is, whether the execution of op_i can be followed by the execution of op_j. Since op_i is executed (according to R^n_{ik}) before op_j, $PreC^n_{ik}$ and $PostC_{kl}$ are by definition true. In order for op_j to be executable according to R^m_{jl}, its precondition $PreC^m_{jl}$ should be true. One way to ensure this is to check that the implication $PreC^n_{ik} \wedge PostC^n_{ik}) \Rightarrow PreC^m_{jl}$ is true. Note that Plugin Prematch is asymmetric because of the implication relationship. Figure 4.5.e shows that op_{ik} and op_{jl} enter their *active* states sequentially ($Active_{ik} \longrightarrow_t Active_{jl}$). Hence, this rule is used for horizontal composition. As defined below, plugin prematch is an instantiation of generic B-composability where $\mathcal{R}_1$ is instantiated to $\Rightarrow$, $\mathcal{R}_2$ to *nil*, and $\widetilde{PreC}_{ikn}$ to

$PreC^n_{ik} \wedge PostC^n_{ik}$:

$$\forall\ R^n_{ik} \in Rules(\text{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\text{op}_{jl}) \mid (PreC^n_{ik} \wedge PostC^n_{ik}) \Rightarrow PreC^m_{jl}$$

Table 4.2 summarizes the way $\mathcal{R}_1$, $\mathcal{R}_2$, and $\widetilde{PreC}_{ikn}$ are instantiated for each of the B-composability rule. For example, for *Plugin B-composability*, $\mathcal{R}_1$ and $\mathcal{R}_2$ are both $\Rightarrow$ and $\widetilde{PreC}_{ikn}$ is $PreC^n_{ik}$. The last column shows that *Plugin Prematch* is suitable for horizontal composition while the other rules are applicable for vertical composition.

Rule	$\mathcal{R}_1$	$\mathcal{R}_2$	$\widetilde{PreC}_{ikn}$	Symmetry	Composition
Exact	$\Leftrightarrow$	$\Leftrightarrow$	$PreC^n_{ik}$	Yes	Vertical
Plugin	$\Rightarrow$	$\Rightarrow$	$PreC^n_{ik}$	No	Vertical
Exact Postmatch	Nil	$\Leftrightarrow$	$PreC^n_{ik}$	Yes	Vertical
Plugin Postmatch	Nil	$\Rightarrow$	$PreC^n_{ik}$	No	Vertical
Plugin Prematch	$\Rightarrow$	Nil	$PreC^n_{ik} \wedge PostC^n_{ik}$	No	Horizontal

Table 4.2: Instantiations of Generic B-Composability

B-composability rules are related according to the graph depicted in Figure 4.6. An arrow from a B-composability rule BR_1 to BR_2 indicates that BR_1 is stronger than BR_2, that is, if op_{ik} is composable with op_{jl} according to BR_1, then it is it is also composable with op_{jl} according to BR_2. For example, *Exact* $\Rightarrow$ *Plugin* $\Rightarrow$ *Plugin Prematch*.

4.5 Qualitative Composability

Composers have generally preferences regarding the quality of operations they would like to compose. For that purpose, we define a set of composability rules, called *qualitative rules*, to compare qualitative features of interacting operations. Assume that

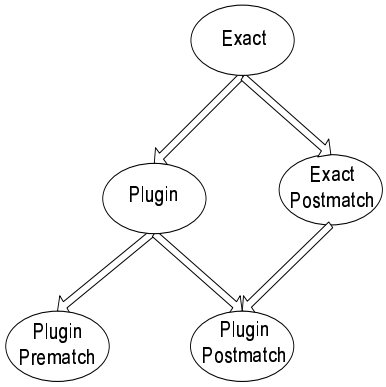

Figure 4.6: Relationships Between B-Composability Rules

op_{ik} and op_{jl} are vertically composed. Since op_{ik} "subcontracts" op_{jl}, it is important to make sure that op_{jl}'s quality is at least "similar" to op_{ik}'s quality. However, if op_{ik} and op_{jl} are horizontally composed, then their quality are independent. Hence, qualitative rules are applicable for vertical composition.

Each rule compares the content of a given qualitative attribute (noted *Q-attribute*) in op_{ik} and op_{jl}. Each rule has the general form $\mathcal{R}(Q\text{-}attribute_{ik}, Q\text{-}attribute_{jl})$ where $\mathcal{R}$ is a relationship between $Q\text{-}attribute_{ik}$ to $Q\text{-}attribute_{jl}$ (e.g., $=$, $\subseteq$, $\leq$, $\geq$). The definition of $\mathcal{R}$ differs from one rule to another. The first qualitative rule compares the *encryption* attribute. It guarantees that either both op_{ik} and op_{jl} encrypt their messages or none of them does. *Authentication* and *Non-repudiation* attributes are compared in the same way. The last security-related rule compares the *confidentiality* attribute. The privacy preferences of op_{ik} should be preserved by the privacy features exposed by op_{jl}. If op_{ik}'s provider does not want a parameter p to be divulged (i.e., $p \in confidentiality_{ik}$) then p should also belong to $confidentiality_{jl}$. We summarize below the subset of qualitative rules dealing with security attributes:

Definition 4.16 – We say that the security attributes of op_{ik} are composable with op_{jl}'s security attributes if (i) $\text{Encryption}_{ik} = \text{Encryption}_{jl}$; and (ii) $\text{Authentication}_{ik} = \text{Authentication}_{jl}$; and (iii) $\text{Non-repudiation}_{ik} = \text{Non-repudiation}_{jl}$; and (iv)

$\text{Privacy}_{ik} \subseteq \text{Privacy}_{jl}$. $\diamondsuit$

The second subset of qualitative rules compare op_{ik}'s and op_{jl}'s business attributes (*cost*, *reputation*, and *regulatory*). Since op_{ik} subcontracts from op_{jl}, then the cost of executing op_{jl} should at most be equal to op_{ik}. Similarly, the ranking of op_{jl} in terms of its *reputation* and *regulatory* features should be at least as good as op_{ik}'s ranking. The following definition describes the composability rules related to business attributes:

Definition 4.17 – We say that the business attributes of op_{ik} are composable with op_{jl}'s business attributes if: (i) $\text{Cost}_{ik} \geq \text{Cost}_{jl}$; and (ii) $\text{Reputation}_{ik} \geq \text{Reputation}_{jl}$; and (iii) $\text{Regulatory}_{ik} \geq \text{Regulatory}_{jl}$. $\diamondsuit$

The last subset of qualitative rules compare run-time attributes: *response time*, *reliability*, and *availability*. The time to executed a subcontracted operation op_{jl} should be less or equal to the response time expected expected by op_{ik}. Additionally, op_{jl} should be at least as reliable and available than op_{ik}. Below is the definition of run-time related composability rules:

Definition 4.18 – We say that the run-time attributes of op_{ik} are composable with op_{jl}'s run-time attributes if (i) $\text{Time}_{ik} \geq \text{Time}_{jl}$; and (ii) $\text{Reliability}_{ik} \leq \text{Reliability}_{jl}$; and (iii) $\text{Availability}_{ik} \leq \text{Availability}_{jl}$. $\diamondsuit$

4.6 Business Process Composability

Service composition involves combining a set of generic operations in a specific way. One important issue to consider is whether such combination provides an added value. To address this issue, we define a rule, called *composition soundness*, to check whether a given composition of generic operations is *sound*. By *sound*, we mean that the way operations are combined provides an added value.

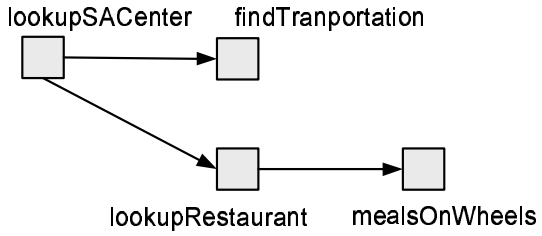

Figure 4.7: Example of Composition Template

4.6.1 Composition and Stored Templates

The definition of *composition soundness* is based on the notions of *composition* and *stored templates* defined below. A *composition template* is built for each composite service *CS* and gives its general structure. It is modeled by a directed graph *(V, E)* where *V* is a set of generic operation IDs (G-op-IDs) and *E* is a set of edges. If an operation *op* is vertically composed with another operation *op'* in CS, then *op* and *op'* represent the same node in *V* since the execution of *op* is "replaced" by the execution of *op'*. Edges in *E* model horizontal composition relationships between *E*'s operations. An edge (op_{ik}, op_{jl}) belongs to E if op_{ik} is horizontally composed with op_{jl}. Figure 4.7 shows an example of composition template for a composite service CS that collects social benefits for senior citizens. CS uses four operations `lookupSACenter`, `findTransportation`, `loookupMealsProviders`, and `mealsOnWheels`. CS first looks for senior activity center. Then, based on the location of that center, it provides for transportation from citizen's residence to the center. It also looks for a participant restaurant around the center and then, arranges for a delivery from the restaurant to center location. The template shows that `lookupSACenter` is horizontally composed with `findTransportation` and `loookupMealsProviders`. The `loookupMealsProviders` operation is horizontally composed with `mealsOnWheels`.

Stored templates are defined by directed graphs similar to those used for composition templates. The difference between stored and composition templates is twofold. First, stored templates are saved in a *stored template repository* (*ST-*

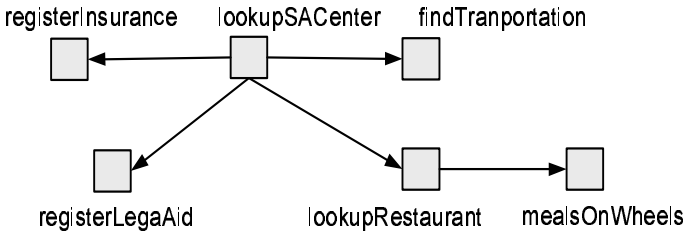

Figure 4.8: Example of Stored Template

repository) while composition are computed for each composite service and then discarded. Second, the interpretation given to composition and stored templates are different. Composition templates model actual composite services as defined by users. Hence they may or may not provide added values. Stored template model "potential" composite services. They are generally pre-defined by domain experts (i.e, community providers). Hence, they inherently provide added values. For example, government agencies dealing with social services would agree that one composite service that collects benefits for senior citizens would combine the following operations: `lookupSACenter`, `findTransportation`, `loookupMealsProviders`, `mealsOnWheels`, `registerInsurance`, and `registerLegalAid` (Figure 4.8). Stored templates may also be "learned" by the system. Each time a composite service is defined by a user, the system saves the corresponding composition template in the *ST-repository* if the template does not already exist in the repository.

4.6.2 Composition Soundness

Because stored templates intrinsically provide added values, they can prove or disprove the soundness of a composite service *CS*. The idea is to compare the composition template of *CS* (*template(CS)*) with the existing stored templates. The following four cases are then possible:

1. *Case 1:* If *template(CS)* is equal to a given stored template *ST*, then *CS* provides *exactly* the same functionalities as the functionalities modeled by *ST*.

We say that *CS* is *exactly sound* with respect to *ST*.

2. *Case 2:* If *template(CS)* is a *subgraph* of a given stored template *ST*, then *CS* provides a subset of the functionalities modeled by *ST*. For example, the composition template depicted in Figure 4.7 is a subgraph of the stored templates depicted in Figure 4.8. We say that *CS* is *loosely sound* with respect to *ST*.

3. *Case 3:* If a given stored template *ST* is a *subgraph* of *template(CS)*, then *CS* provides all the functionalities modeled by *ST*. *CS* also provides functionalities not offered by *ST*. We say that *CS* is *strongly sound* with respect to *ST*.

4. *Case 4:* If none of the previous cases is possible, then *CS* is not *sound*.

We give below the definition of composition soundness rules.

Definition 4.19 – Let *CS* be a composite service. We define three composition soundness rules as follows:

- *CS* is *exactly sound* if $\exists$ $ST \in$ ST-repository $\mid$ *template(CS)* $= ST$.
- *CS* is *loosely sound* if $\exists$ $ST \in$ ST-repository $\mid$ *template(CS)* $\subset ST$.
- *CS* is *strongly sound* if $\exists$ $ST \in$ ST-repository $\mid$ *template(CS)* $\supset ST$. $\diamondsuit$

Chapter 5

Automatic Composition of Semantic Web Services

The *automatic* composition of Web services is a recent trend to unburden composers from the complexity of the composition process (e.g., checking composability, orchestrating Web services) [12, 74]. It is slated to play a major role in enabling the envisioned *Semantic Web Services* paradigm [133]. In this chapter, we present a Semantic Web centered approach for the automatic composition of Web services [80, 82]. The proposed approach builds on the semantic description framework and composability model described in chapters 3 and 4 respectively. It consists of three conceptually separate phases (Figure 5.1): *specification*, *matchmaking*, and *generation*.

The *specification* phase (phase 1) enables high-level descriptions of composition requests. For that purpose, we define a *Composition Specification Language* (*CSL*). CSL extends *UML activity diagrams* [56] with an ontological description of the composition request. Using CSL, composers specify the *what* part of the desired composition but will not concern themselves with the *how* part. They are not required to be aware of the full technical details such as the list of participant services and the way they are executed and plugged together. The *matchmaking* phase (phase 2) uses the composability model to generate *composition plans* that conform to composers'

specifications. One or several composition plans are generated for each composer's sub-request. By *composition plan*, we refer to the list of imported operations to be outsourced, their orchestration order, and the way they interact with each other (plugging operations, mapping messages, etc.) to "realize" the corresponding sub-request. In the *generation phase* (phase 3), detailed descriptions of the composite service are generated. We propose a *Quality of Composition* (*QoC*) model to assess the quality of the generated descriptions.

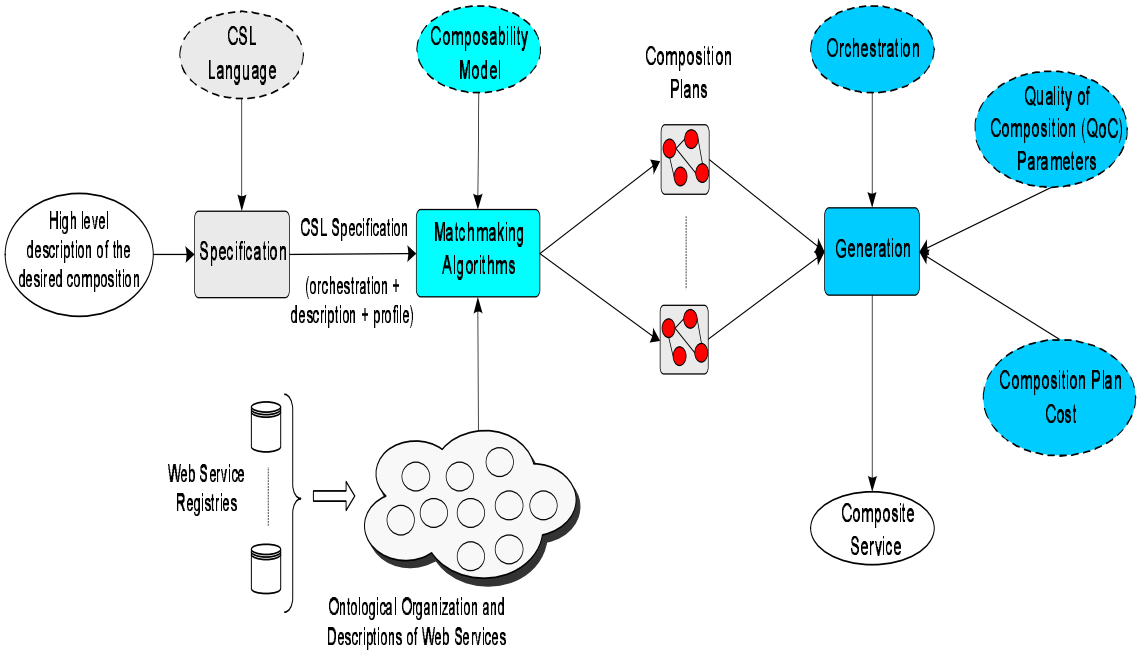

Figure 5.1: Overview of the Proposed Approach for Service Composition

This chapter is organized as follows. In Section 5.1, we define the model and techniques used for the specification of composition requests. In Section 5.2, we propose a set of algorithms for checking composability and the automatic generation of composition plans. In Section 5.3, we illustrate the process of generating composite service descriptions.

5.1 Specification of Composition Requests

The specification of a composition request requires dealing with two major issues: *orchestration* and *description*. A composer's request generally includes one or several *sub-requests*. For example, the request "visit a senior activity center" includes several sub-requests such as "lookup for senior activity center" and "set appointment with a senior activity center". The sub-requests should be executed in a specific order and if certain conditions are true (e.g., look for senior activity center *then* set up an appointment). Such order and conditions are specified through an *orchestration model*. Additionally, the automatic generation of composite services requires a clear and unambiguous *description* of the request and its sub-requests. For example, the composition engine should understand what "lookup for senior activity center" means. In this section, we define a language, called *Composition Specification Language* (*CSL*), for the specification of composition requests.

5.1.1 Orchestration Model

We define two types of orchestration in our approach: *composer-defined* and *system-generated* orchestration. *Composer-defined* is provided by the composer in the specification phase. It gives the execution order of the composition sub-requests. We refer to it as *inter-subrequest* orchestration since it represents the orchestration of sub-requests. *System-generated* orchestration is automatically generated by the composition engine. It gives the execution order of the participant services outsourced for each sub-request. We refer to it as *intra-subrequest* orchestration since it represents the orchestration within sub-requests. The orchestration model presented in this section deals with the first type. Details about system-generated orchestration are given in Section 5.2.

Several orchestration models have been proposed in the literature such as *UML activity diagrams*, *Petri-nets*, *statecharts*, and *π-calculus*. In our approach, we adopt *UML activity diagrams* as an orchestration model [56]. Activity diagrams are the most widely used process modeling techniques both in conventional interaction tech-

nologies (e.g., workflows) and Web services [5]. The reason for their success is their ease-of-use and simplicity for modeling business processes. Several tools (e.g., *Rational Rose*) are available for designing business processes using activity diagrams. Additionally, the *Unified Modeling Language* (*UML*) has become the *de facto* standard for representing application architectures and design models. Finally, activity diagrams model orchestrations by specifying which actions should be performed, from the beginning of the execution to the end. This seems to be the most natural way in which users think of a process and developers code their applications [5].

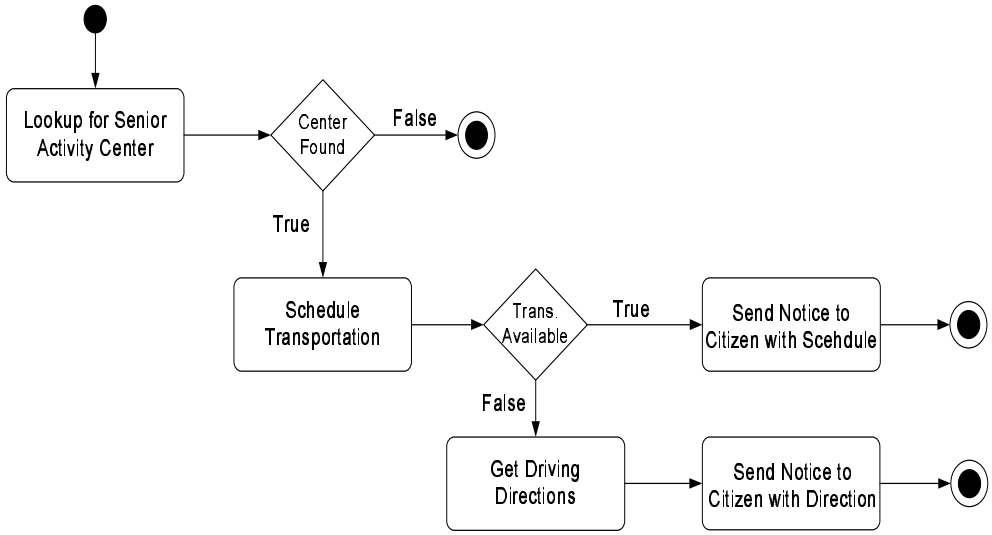

Figure 5.2: Modeling Composition Request through Activity Diagrams

Activity diagrams show the flow of *activities* in a business process. In our approach, each activity represents a composition sub-request. A sub-request is modeled as a capsule (i.e., as a rectangle with rounded corners) with the sub-request name enclosed. We depict in Figure 5.2, an example of orchestration for a composition request within our e-government scenario. The case officer specify the composition request that is, "organizing a visit to a senior activity center" through an activity diagram. The filled circle is the starting point of the diagram. The filled circle with a border is the ending point. The composition request includes five sub-requests

(e.g., `schedule transportation`). The case officer has also the possibility to indicate that one sub-request *conditionally* follows another. For example, the *diamond* labeled "Trans. Available" tests whether a government-funded transportation is available. In this case, a notification should be sent to the citizen with schedule. Otherwise, the case officer performs a `get driving directions` from the citizen's home to the activity center location. Activity diagrams may also model *parallelism* via *fork* and *join* constructs. A *fork* is represented by a black bar with one flow going into it and several leaving it. It denotes the beginning of parallel processing. A *join* is depicted by a black bar with several flows entering it and one leaving it. It denotes the end of parallel processing.

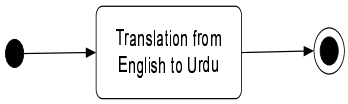

Figure 5.3: Composition Request with one Single Sub-request

A composition request specification may consist of only one sub-request. For example, Figure 5.3 depicts a composition request that "provides translation from English to Urdu". The request contains one sub-request label `translation from English to Urdu`. It is important to note that, during the composite service generation process, each sub-request may be mapped to a set of Web service operations combined in a specific way. For example, the `translation from English to Urdu` sub-request would be mapped to the combination of Web service operations in the following order: `English-French` → `French-Chinese` → `Chinese-Urdu`.

5.1.2 Describing Composition Sub-Requests

Activity diagrams enable the specification of the execution order of the composition sub-requests. They provide no means for describing sub-requests. Such description concerns both syntactic and semantic features. To deal with this issue, we define an ontology for sub-requests called *sub-request ontology*. All sub-requests are described

according to this ontology. Since a sub-request is meant to be mapped to one or several generic operations, we adopt the ontology defined for generic operations (Figure 3.3). Each sub-request is hence described by syntactic, static semantic, dynamic semantic, and qualitative attributes.

```
<request>
    <name> ..... </name>
    <description>  ..... </description>
    <Parameters>  .....  <Parameters>
    <subRequest>
       <syntax> ..... </syntax>
       <staticSemantics> ..... </staticSemantics>
       <dynamicSemantics> ..... </dynamicSemantics>
       <quality> ..... </quality>
   </subRequest>
   .....
</request>
```

Figure 5.4: Description of a Composition Request in CSL

The ontology-based description of a request and its sub-requests produces an XML document. We give in Figure 5.4 the general form of this document. For the sake of clarity, we omit references to XML namespaces. The top element (<*request*>) contains the name of the request (<*name*> element), a text description (<*description*>) of the request, a <*parameters*> element, and several <*subRequest*> elements. The <*parameters*> element gives the input and output parameters used by composition sub-requests. Similarly to profile inputs, sub-request parameters (input and output) may be defined according to a specific taxonomy. The <*subRequest*> element contains six sub-elements. The first sub-element (<*syntax*>) includes syntactic attributes. The binding of a sub-request is similar to the binding of the composition request defined in the composer profile. The sub-elements <*staticSemantics*> and <*dynamicSemantics*> describe the static and dynamic semantics of the sub-request. The dynamic semantics of sub-request is limited to behavioral attributes. Pre-operations and post-operations represent pre-defined business processes given by

community providers and are hence specific to generic operations. The <*quality*> sub-elements contains qualitative attributes of the sub-request. The <*input*> and <*output*> sub-elements describe the input and output parameters required by the sub-request. A sub-request may use as input/output, the parameters defined within the composer profile. It may also use the parameters defined within the <*parameters*> element. In both cases, the <*input*> and <*output*> sub-elements contain references to the parameters' names.

5.1.3 Customization via Composer Profiles

Customization is a key requirement of the automatic composition of Web services. Composers must be able to personalize the composition process according to their preferences. Such preferences make up the composer's *profile*. The general form of the profile definition construct is given below:

Define Profile
Levels Assign <vector> **To** *LW*
Rules Assign <matrix> **To** *RW*
Degree Assign <value> **To** <τ-composability>
Binding Assign <value> **To** <binding>
Plans Assign <value> **To** <max-plans>
Input [<parameter>]$^+$

As part of their profile definition, composers assign values to the level weights vector (LW) and rule weights matrix (RW). They also provide the τ-composability threshold and binding protocols supported by their composite services. Additionally, they give the maximum number of composition plans to be generated for each composition sub-request. They finally provide a list of input parameters (e.g., citizen's name, address) to be used during the execution of the generated composite service. Each parameter is defined by its name, data type, unit, business role, and

language (Section 3.2). To facilitate the definition of such parameters, composers may use pre-defined parameters defined in any existing taxonomy. An example of such taxonomy is the *RosettaNet Technical Dictionary*. This dictionary contains properties that can be used to describe characteristics of a product or service [33].

5.2 Outsourcing Web Services in the Matchmaking Phase

Once a CSL specification (i.e., orchestration, profile, and description) is provided, the next step is to generate *composition plans* relevant to each sub-request using matchmaking algorithms. The general premise of such algorithms is to consider each sub-request as an *"abstract" operation*. Abstract operation would then be vertically composed with one or a combination of generic operations. Each vertical composition returns a set of composition plans. Let n be the number of sub-requests and *Nb_CP(r)* be the number of plans generated for the r^{th} sub-request. The number of plans generated for the whole composition request would be $\sum_{r=1}^{n} Nb\_CP(r)$.

```
(00) Matchmaking(Profile) {
(01) for each sub-request SR_p do {
(02)  Vert_Comp1:1(SR_p, Registry, Plans1:1[p], Profile, nb_plans);
(03)  if nb_plans ≤ Profile.max_plans
(04)    then Vert_Comp1:N(SR_p, Registry, Plans1:n[p], Profile, nb_plans);
(05) }  }
```

Figure 5.5: Main Matchmaking Algorithm

The generation of composition plans is based on checking composability rules between each sub-request and the imported operations. The main matchmaking algorithm is depicted in Figure 5.5. For each sub-request, the algorithm first tries to vertically compose it with one generic operation. We refer to such composition as a *one-to-one* (1:1) composition. Each entry *Plans1:1[p]* contains the list of composition plans generated for sub-request SR_p. If the number of generated plans is less or equal

to the maximum number of plan per request, then the algorithm tries to vertically compose the sub-request with a combination of generic operations. We refer to such composition as a *one-to-many* (1:N) composition.

5.2.1 One-to-One Vertical Composition

We propose three algorithms for performing 1:1 vertical composition. The first algorithm is *operation-centric*. It compares each sub-request *SR* with all operations in the registry. The second algorithm is *community-centric*. It compares each sub-request *SR* with the imported operations op_{ik} of each relevant community C_i. The third algorithm adopts a *message-centric* approach for checking composability. It compares the vertical composability of *SR*'s message with the messages in C_i and then determines the imported operations with which *SR* is composable.

5.2.1.1 Operation-Centric Matchmaking Algorithm

The *operation-centric* algorithm for 1:1 vertical composition uses operations as a basis for checking composability. It performs an exhaustive search in the service registry. For that purpose, it browses the registry to determine all operations that could "replace" each sub-request *SR* (Figure 5.6). The algorithm checks the vertical composability of *SR* with every operation op_{ik} in the registry. Syntactic and static semantic (at the operation level) composability are checked as illustrated in Chapter 4. Static semantic composability at the message level is checked by comparing *SR*'s input and output parameters with the input and output parameters of op_{ik}. The composability degree is computed after checking each group of composability rules. If the degree is greater or equal to τ-composability (specified in the composer profile), then op_{ik} is a potential candidate to "replace" *SR*. In this case, op_{ik} is added to the set of *SR*'s composition plans.

The operation-centric algorithm uses two procedures for checking syntactic and static semantic composability (at the operation level). We give in Figure 5.7 the algorithms executed by these procedures. A rule is checked if the corresponding

```
(00) Operation1:1(SR, Registry, Plans1:1, Profile, nb_plans) {
(01)  nb_plans = 0
(02)  degree = 0
(03)  Plans1:1 = ∅
(04)   for each operation op_ik ∈ Registry | (nb_plans < Profile.max_plans) do {
(05)    syntactic_vertical(SR, op_ik, degree)
(06)    if degree > Profile.τ-composability
(07)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(08)           nb_plans = nb_plans + 1
(09)           continue   }
(10)    static_semantics_vertical(SR, op_ik, degree)
(11)    if degree > Profile.τ-composability
(12)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(13)           nb_plans = nb_plans + 1
(14)           continue   }
(15)    message_vertical(SR, op_ik, degree)
(16)    if degree > Profile.τ-composability
(17)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(18)           nb_plans = nb_plans + 1
(19)           continue   }
(20)    behavioral(SR, op_ik, degree)
(21)    if degree > Profile.τ-composability
(22)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(23)           nb_plans = nb_plans + 1
(24)           continue   }
(25)  }  }
```

Figure 5.6: Operation-Centric Algorithm for 1:1 Vertical Composition

weight (assigned by the composer) is positive. For that purpose, we use a function *get_weightR(CR)* that returns the weight of the current rule *CR* from the RW matrix. We also use a function *get_weightL(CL)* that returns the weight of the current level *CL* from the LW vector. The *Static_semantics_vertical* procedure uses two procedures *composable(category$_{ik}$,category$_{jl}$)* and *composable(purpose$_{ik}$,purpose$_{jl}$)* that check the conditions specified in Definition 4.7 and 4.8, respectively.

The *message_vertical()* procedure checks that *SR* is message composable with

```
(00) Syntactic_vertical(op_ik,op_jl,degree) {
(01)     d = 0
(02)     if get_weightR(CR) > 0 and mode_ik = "In/Out"
(03)          then d = d + get_weightR(CR);
(04)     if get_weightR(CR) > 0 and Binding_ik ∩ Binding_jl ≠ ∅
(05)          then d = d + get_weightR(CR);
(06)     if get_weightR(CR) > 0 and mode_ik = "In/Out" and
(07)                    |In_jl| ≤ |In_ik| and |Out_ik| ≤ |Out_jl|
(08)          then d = d + get_weightR(CR);
(09)     if get_weightR(CR) > 0 and mode_ik = "Out/In" and
(10)                    |In_jl| ≤ |Out_ik| and |In_ik| ≤ |Out_jl|
(11)          then d = d + get_weightR(CR);
(12)     degree = degree + d × get_weightL(CL);
(13) }

(00) Static_semantics_vertical(op_ik,op_jl,degree) {
(01)     d = 0
(02)     if get_weightR(CR) > 0 and serviceability_ik = serviceability_jl
(03)          then d = d + get_weightR(CR);
(04)     if get_weightR(CR) > 0 and provider_ik ∩ provider_jl ≠ ∅ and
(05)                    consumer_ik ∩ consumer_jl ≠ ∅
(06)          then d = d + get_weightR(CR);
(07)     if get_weightR(CR) > 0 and composable(category_ik,category_jl)
(08)          then d = d + get_weightR(CR);
(09)     if get_weightR(CR) > 0 and composable(purpose_ik,purpose_jl)
(10)          then d = d + get_weightR(CR);
(11)     degree = degree + d × get_weightL(CL);
(12) }
```

Figure 5.7: Syntactic and Operation Semantic Composability

op_{ik} (Figure 5.8). It uses a function *Is_message_composable(M_1,M_2)* which returns *true* or *false* depending on whether a message M_1 is composable with M_2. To allow a one-to-one mapping between M_1's and M_2's parameters, we use the *matched* set. A parameter p_{2l} of M_2 is inserted in this set, if we already found a parameter p_{1k} of M_1 that is composable with p_{2l}.

Behavioral composability (the *behavioral* procedure) compares *SR*'s business

```
(00) Message_vertical(op_ik,op_jl,degree) {
(01)        if mode_ik = "In/Out" and Is_message_composable(In_ik,In_jl) and
(02)              Is_message_composable(Out_jl,Out_ik)
(03)                    then degree = degree + get_weightL(CL);
(04)        if mode_ik = "Out/In" and Is_message_composable(Out_ik,In_jl) and
(05)              Is_message_composable(Out_jl,In_ik)
(06)                    then degree = degree + get_weightL(CL);
(07) }

(00) Is_message_composable(M1,M2):boolean {
(01)        matched = ∅
(02)             for each param p_1k ∈ M_1 do
(03)        { found = false
(04)              for each param p_2l ∈ M_2 | p_2l ∉ matched do
(05)         if composable(p_1k,p_2l)
(06)           then { found = true
(07)                  matched = matched ∪ {p_2l}
(08)                  break }
(09)         if ¬found then return false
(10)         } /* for in line (03) *
(11)       return true
(12) }
```

Figure 5.8: Message Composability

logic rules with op_{ik}'s. Since B-composability rules are hierarchically organized (Figure 4.6), we adopt a bottom-up approach for checking these rules. Indeed, if the *Plugin Postmatch* rule is not satisfied, then the *Plugin* rule is necessarily not satisfied since *Plugin* $\Rightarrow$ *Plugin Postmatch*. We present in Figure 5.9, the algorithm executed for checking behavioral composability. The algorithm uses four functions *check_plugin_postmatch()*, *check_plugin*, *check_exact_postmatch()*, and *check_exact*. These functions return true if the corresponding B-composability rules are satisfied.

B-composability is based on proving implications between business logic rules' conditions (pre- and post-conditions). However, proving expressions such as $Cond_1 \Rightarrow Cond_2$ is a NP-complete problem [36, 110]. To deal with this issue we

```
(00) Behavioral(op_ik,op_jl,degree) {
(01)     d = 0;
(02)     isPluginPost = check_Plugin_Postmatch(op_ik,op_jl);
(03)     if ¬isPluginPost
(04)           then isPlugin = false
(05)           else isPlugin = check_Plugin(op_ik,op_jl);
(06)     isExactPost = check_Exact_Postmatch(op_ik,op_jl);
(07)     if ¬isPlugin or ¬isExactPost
(08)           then isExact = false
(09)           else isExact = check_Exact(op_ik,op_jl);
(10)     if get_weightR(CR) > 0 and isPluginPre
(11)           then d = d + get_weightR(CR);
(12)     if get_weightR(CR) > 0 and isPluginPost
(13)           then d = d + get_weightR(CR);
(14)     if get_weightR(CR) > 0 and isPlugin
(15)           then d = d + get_weightR(CR);
(16)     if get_weightR(CR) > 0 and isExactPost
(17)           then d = d + get_weightR(CR);
(18)     if get_weightR(CR) > 0 and isExact
(19)           then d = d + get_weightR(CR);
(20)     degree = degree + degree × get_weightL(CL);
(21) }

(00) Prover(Cond_1,Cond_2) :boolean {
(01)     while true do
(02)      if ∃ x_1^1,...,x_n^1 in Cond_1 and x_1^2,...,x_m^2 in Cond_2 | composable(x_i^1,x_j^2)
(03)        then replace x_1^1,...,x_n^1 and x_1^2,...,x_m^2 by x_1^2
(04)        else break
(05)     for each term t_2p in Cond_2 do
(06)      if ∃ term t_1q in Cond_1 | t_1q ⇒ t_2p
(07)        then return true
(08)        else return false
(09) }
```

Figure 5.9: Behavioral Composability

define an approximate solution. The proposed solution is based on the assumption that each condition is a conjunction of *terms*. Each *term* has the form x <r> v where

x is parameter, v is a constant value, and <r> is a relational operator that belongs to $\{=,\neq,<,>,leq,\geq\}$. The theorem prover first *unifies* the parameters in $Cond_1$ and $Cond_2$. The unification step works as follows: if there are parameters $x_1^1,...,x_n^1$ in $Cond_1$ that are composable with parameters $x_1^2,...,x_m^2$ in $Cond_2$, then replace $x_1^1,...,x_n^1$ and $x_1^2,...,x_m^2$ by the same parameter name (say x_1^2). The second step of the prover is to *match* each term t_{2p} of $Cond_2$ with a term t_{1q} in $Cond_1$. We say that t_{2p} *matches* with t_{1q} if $t_{1q} \Rightarrow t_{2p}$. Proving the matching between terms is done by applying one of the *inference rules* for relational operators (Table 5.1, 5.2, 5.3, 5.4, 5.5, 5.6). If a given term in $Cond_2$ matches with no term in $Cond_1$, then $Cond_1 \Rightarrow Cond_2$ is false. If all terms in $Cond_2$ are matched with a term in $Cond_1$, then $Cond_1 \Rightarrow Cond_2$ is false.

$$\frac{x > a \ \wedge \ a = b}{x > b} \qquad \frac{x > a \ \wedge \ a > b}{x > b} \qquad \frac{x > a}{x \geq a}$$

$$\frac{x > a \ \wedge \ a \geq b}{x \geq b}$$

Table 5.1: Inference Rules for the "$>$" Operator

$$\frac{x \geq a \ \wedge \ a = b}{x \geq b} \qquad \frac{x \geq a \ \wedge \ a > b}{x > b} \qquad \frac{x \geq a \ \wedge \ a \geq b}{x \geq b}$$

Table 5.2: Inference Rules for the "$\geq$" Operator

$$\frac{x < a \ \wedge \ a = b}{x < b} \qquad \frac{x < a \ \wedge \ a < b}{x < b} \qquad \frac{x < a}{x \leq a}$$

$$\frac{x < a \ \wedge \ a \leq b}{x \leq b}$$

Table 5.3: Inference Rules for the "<" Operator

$$\frac{x \leq a \ \wedge \ a = b}{x \leq b} \qquad \frac{x \leq a \ \wedge \ a < b}{x < b} \qquad \frac{x \leq a \ \wedge \ a \leq b}{x \leq b}$$

Table 5.4: Inference Rules for the "≤" Operator

$$\frac{x \neq a \ \wedge \ a = b}{x \neq b}$$

Table 5.5: Inference Rules for the "≠" Operator

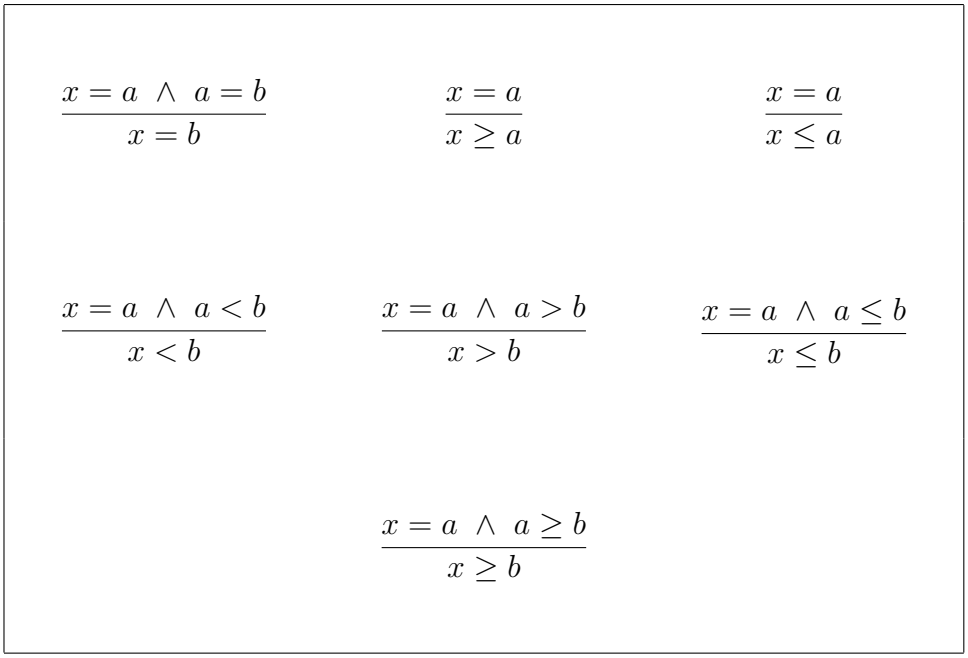

Table 5.6: Inference Rules for the "=" Operator

5.2.1.2 Community-Centric Matchmaking Algorithm

The *community-centric* algorithm for 1:1 vertical composition uses communities as a basis for selecting service operations. It browses "relevant" communities to determine all imported operations that could "replace" a sub-request *SR* (Figure 5.6).

To accelerate the search process, the algorithm focuses on communities C_i whose category may be composed with *SR*'s category. The remaining communities and their imported operations are pruned from the service space. The algorithm then checks the vertical composability of *SR* with every imported operation op_{ik} in C_i. It considers imported operations instead of generic ones because imported operation are the actual operations offered by community members. Additionally, imported operations may change the description of generic operations (e.g., removing input/output parameters). The algorithms for *syntactic_vertical()*, *static_semantics_vertical()*, *message_vertical()*, and *behavioral()* procedures are simi-

```
(00) Vert_Comp_Operation1:1(SR, Registry, Plans1:1, Profile, nb_plans) {
(01)  nb_plans = 0
(02)  degree = 0
(03)  Plans1:1 = ∅
(04)  for each community C_i ∈ Registry | category_composable_vertical(C_i, SR)
(05)      and (nb_plans < Profile.max_plans) do
(06)   for each imp.  operation op_ik ∈ C_i | (nb_plans < Profile.max_plans) do {
(07)    syntactic_vertical(SR, op_ik, degree)
(08)    if degree > Profile.τ-composability
(09)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(10)           nb_plans = nb_plans + 1
(11)           continue   }
(12)    static_semantics_vertical(SR, op_ik, degree)
(13)    if degree > Profile.τ-composability
(14)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(15)           nb_plans = nb_plans + 1
(16)           continue   }
(17)    message_vertical(SR, op_ik, degree)
(18)    if degree > Profile.τ-composability
(19)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(20)           nb_plans = nb_plans + 1
(21)           continue   }
(22)    behavioral(SR, op_ik, degree)
(23)    if degree > Profile.τ-composability
(24)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(25)           nb_plans = nb_plans + 1
(26)           continue   }
(27)  }  }
```

Figure 5.10: Community-Centric Algorithm for 1:1 Vertical Composition

lar to those presented in the operation-centric algorithm. Similarly to the operation-centric algorithm, the composability degree is computed after each composability rule. If the degree is greater or equal to τ-composability, then op_{ik} is a potential candidate to "replace" *SR*.

5.2.1.3 Message-Centric Matchmaking Algorithm

The main characteristic of the community-centric algorithm is that it compares sub-requests with all operations in the community to generate composition plans. However, message parameters in different operations may be "similar" (in terms of composability rules) and hence do not have to be compared for each operation. The *message-centric* algorithm approaches the matchmaking problem from another perspective. It checks the composability of messages instead of operations and then derives the operations that could be vertically composed with a sub-request.

The design of the message-centric matchmaking algorithm raises two important issues: (i) how should community messages be modeled (ii) how can outsourced operations be deduced from messages. To deal with these issues we use the concept of *finite automaton* [118]. A *finite automaton* has several parts including a set of *states* and rules or *transitions* for going from one state to another, depending on the input symbol. It also has an input *alphabet* that indicates the allowed input symbols. It finally has a *start* state and a set of *accept* or *final* states.

Formally, a *finite automaton* is a 5-tuple $(Q,\sum,\delta,q_0,F)$ where Q is a finite set of *states*, $\sum$ is a finite set called *alphabet*, δ: $Q\times\sum\rightarrow Q$ is the *transition* function, $q_0 \in Q$ is the *start* state, and $F \subseteq Q$ is the set of *final states*. The transition function δ defines the rules for moving from a state to another. If the automaton has an arrow from a state S_1 to a state S_2 labeled with the input symbol x, that means that, if the automaton is in state S_1 when it reads a x, it then moves to state S_2. We generally indicate this with the transition function by saying that $\delta(S_1,x) = S_2$. An *empty* transition (noted ε) between S_1 and S_2 means that we can move from S_1 to S_2 without reading any input. When the automaton receives an input string, it processes that string and produces an output. The output is *accept* if all the string is parsed and the automaton reaches a final state. Otherwise, the output is *reject*.

A) Defining the Automaton for Community Messages

Since checking message composability from sub-requests to an imported operations concerns only the input messages of *In/Out* operations, we define an *Input Automaton* ($\mathcal{IA}_i$) for each community C_i. $\mathcal{IA}_i$ models C_i's input messages related to *In/Out* operations. $\mathcal{IA}_i$ has iq_0 as an initial state. The alphabet of $\mathcal{IA}_i$ is the set of XML Schema data types. The use of data types as alphabet has the important advantage of limiting the alphabet size since the number of data types is finite. $\mathcal{IA}_i$ is updated each time that an *In/Out* generic operation is imported.

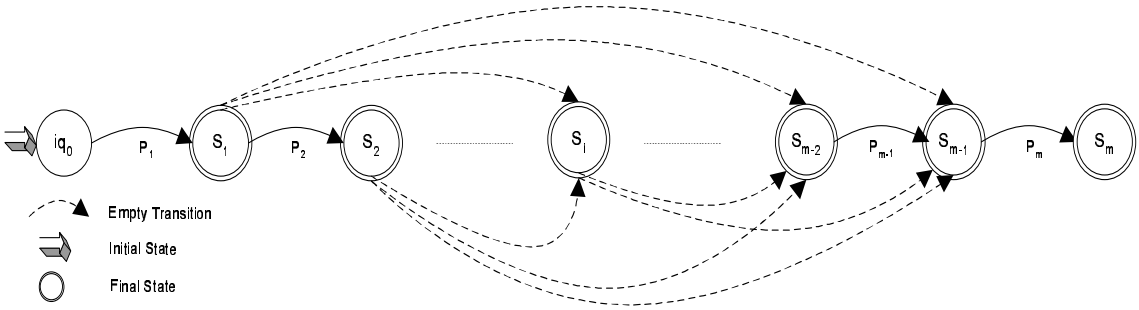

Figure 5.11: Automaton States and Transitions Definition

Let us now illustrate the algorithm used for building this automata. Assume that an input message $M = \{P_1, P_2, ..., P_m\}$ is defined by a community member (by projection, extension, or adjustment). The algorithm for adding M to $\mathcal{IA}_i$ includes two steps: (i) defining states and transitions, and (ii) checking *determinism*. In the first step, the message parameters are sorted according to their XML Schema data type. Let P_1, P_2,..., P_m be the order obtained for M parameters. New states S_1, S_2,...., S_m are then created (Figure 5.11). Each state S_i corresponds to a parameter P_i (Figure 5.11). A transition labeled with P_1 data type is first created from iq_0 to S_1. A transition labeled with P_{i+1}'s data type is then created from S_i to S_{i+1} (i=1,m-1). Since an *SR*'s input message may map to a subset of M parameters, then all states S_i (i=1,m) are made final. Each final state contains the ID of M's operation. Additionally, we create an empty transition ε from iq_0 to S_i (i=2,m-1). An empty transition is also created from each state S_i to another state S_j where

$1 \leq i \leq m\text{-}2$, $j \leq m$, and $j \geq i+2$. If, for example, *SR*'s input message contains only P_3 and P_5, then it would be accepted by the automaton.

The second step of the automaton definition algorithm is *determinism*. An automaton is *deterministic* if (i) it does not contain empty transitions and (ii) each state has zero or one exiting arrow for each alphabet symbol. If at least one of those conditions is not true, then the automaton is *nondeterministic*. Determinism is an important feature of automata. Every step of computation in a deterministic automaton follows in a unique way from the preceding way. When the automaton is in a given state and reads the next input symbol, it determines without ambiguity what the next state will be. $\mathcal{IA}_i$ is, by definition, nondeterministic because of the existence of empty transitions. To facilitate the process of comparing *SR*'s messages, we transform $\mathcal{IA}_i$ into a deterministic automaton. Indeed, every nondeterministic automaton can be converted into an equivalent deterministic automaton [118]. We adopt the algorithm presented in [118] to eliminate empty transition in a finite automaton. We summarize in Figure 5.12 the algorithm for building $\mathcal{IA}_i$. This automaton is incrementally updated each time that an imported operation with an input message M is defined.

```
(00) UpdateAutomaton(IA_i, M) {
(01)   Let IA_i = (Q,∑,δ,q_0,F)
(02)   Let M = {P_1, P_2,..., P_m}
(03)   ∑ = ∑ ∪ {S_1, S_2,..., S_m}
(04)   F = F ∪ {S_1, S_2,..., S_m}
(05)   δ(q_0, P_1.type) = S_1
(06)   for i=2 to m-1 do
(07)     δ(q_0, ε) = S_i
(08)   for i=1 to m-1 do
(09)     δ(S_i, P_{i+1}.type) = S_{i+1}
(10)   for i=1 to m-2 do
(11)     for j=i+2 to m do
(12)       δ(S_i, ε) = S_j
(13)   eliminate_empty_transitions(A) }
```

Figure 5.12: Updating Input Automata

```
(00) Vert_Comp_Message1:1(SR, Registry, Plans1:1, Profile, nb_plans) {
(01)  degree = 0
(02)  Plans1:1 = ∅
(03)  for each community C_i ∈ Registry | category_composable_vertical(C_i, SR) ∧
(04)         (nb_plans < Profile.max_plans) do {
(05)   if SR.mode = "In/Out"
(06)      then accepted_automaton(IA_i, SR.input, Final, degree)
(07)      then accepted_automaton(IA_i, SR.output, Final, degree)
(08)   for each imported operation op_ik | (op_ik ∈ Operations(Final)) ∧
(09)         (nb_plans < Profile.max_plans) do {
(10)    if SR.mode = "I/O" and ¬Is_message_composable(SR.output, op_ik.output)
(11)      then continue;
(12)    if SR.mode = "O/I" and ¬Is_message_composable(SR.input, op_ik.output)
(13)      then continue;
(14)    if get_weightR(CR) > 0 and Binding_ik ∩ Binding_jl ≠ ∅
(15)        then degree = degree + get_weightR(CR);
(16)    if degree > Profile.τ-composability
(17)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(18)           nb_plans = nb_plans + 1
(19)           continue   }
(20)    static_semantics_vertical(SR, op_ik, degree)
(21)    if degree > Profile.τ-composability
(22)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(23)           nb_plans = nb_plans + 1
(24)           continue   }
(25)    behavioral(SR, op_ik, degree)
(26)    if degree > Profile.τ-composability
(27)      then { Plans1:1 = Plans1:1 ∪ {op_ik}
(28)           nb_plans = nb_plans + 1
(29)           continue   }
(30)  }  }  }
```

Figure 5.13: Message-Centric Algorithm for 1:1 Vertical Composition

B) Algorithm for Vertical Composition

The message-centric algorithm for 1:1 vertical composition uses $\mathcal{IA}_i$ of relevant communities C_i to select service operations (Figure 5.13). To avoid comparison

with all operations, the algorithm prunes the Web service space at three levels. First, it focuses on communities C_i whose category may be composed with *SR*'s category. Second, it considers only "In/Out" operations in C_i. Third, it checks composability only with operations that are message composable with *SR*.

The algorithm considers two cases depending on the mode of *SR*. If *SR*'s mode is "In/Out", then we check whether *SR*'s input is accepted by $\mathcal{IA}_i$ via the function *accepted_automaton($\mathcal{IA}_i$,SR.input,Final,degree)*. We then consider only operations that belong to the *Final* set (i.e., accepted by $\mathcal{IA}_i$). Let us assume that *SR*'s input is mapped with the input of an operation op_{ik} (i.e., op_{ik} belongs to *Final*). In this case, op_{ik}'s output is compared (in terms of message composability) with *SR*'s output. If *SR*'s mode is "Out/In", then $\mathcal{IA}_i$ is used to test whether *SR*'s output is accepted via the function *check_automaton($\mathcal{OA}_i$, SR.output, Final, degree)*. Let us assume that *SR*'s output is mapped with the input of an operation op_{ik}. In this case, op_{ik}'s output is compared with *SR*'s input. The algorithm then checks binding composability since mode and PN-composability are checked in the previous steps. The algorithms for *static_semantics_vertical()*, *Is_message_composable()*, and *behavioral()* procedures are similar to those used in the operation-centric and community-centric algorithms.

5.2.2 One-to-Many Vertical Composition

The one-to-many vertical composition algorithm determines the set of imported operations which, combined in certain way, could be vertically composed with a sub-request. For example, the `English-Urdu` sub-request is composable with the combination of three operations: `English-French`, `French-Chinese`, and `Chinese-Urdu`. To support 1:N vertical composition, we introduce the notion of *flow graph*. The *flow graph* links together all operations that are *horizontally composable*. The nodes of this graph are of two types: *operation* and *compound* nodes (Figure 5.14). Arrows in this graph relate nodes of different types. Operation nodes may be linked to compound nodes and vice versa. However, no operation node is linked with another

operation node. Similarly, no compound node is linked to another compound node.

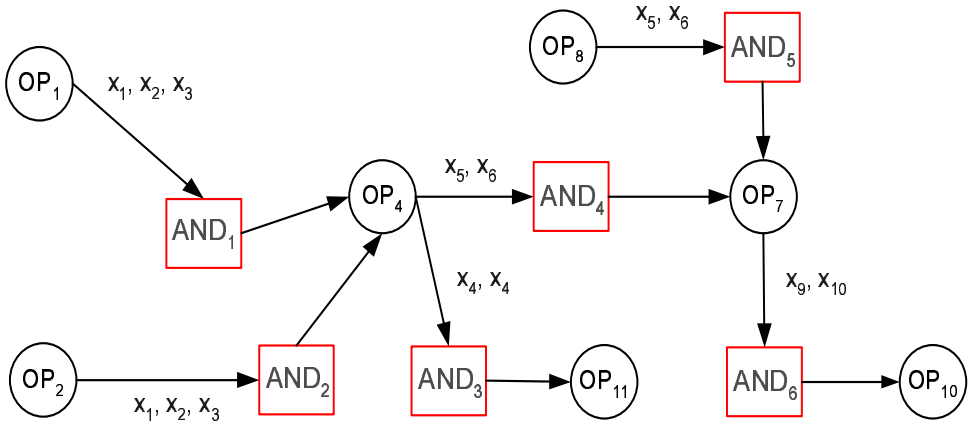

Figure 5.14: The Flow Graph

Each compound node *And* in the flow graph has one predecessor op_i and one successor op_j (e.g. And_2). This situation models the fact that op_i is horizontally composable with op_j. This means that op_j can be invoked after the execution of op_i. The arrow $op_i \rightarrow And$ is labeled with op_i's outputs that could be used as op_j's inputs. op_j is labeled with its output parameters. The compound node *And* is labeled with op_i's inputs. If the same op_i's output occurs m times on the arrow, then op_i should be executed m times before invoking op_j (e.g., x_4 in op_4). Each compound node is labeled with all input parameters defined in its predecessors (i.e., operations).

The flow graph is built in an incremental fashion, each time an operation *op* is imported (Figure 5.15). This assumes that all operations in the service registry are already modeled in the flow graph. To update this graph, we compare *op* with each operation node op_i in the graph. If *op* is horizontally composable with op_i, then a new operation node is created for *op*. A new compound node And_k is also inserted in the graph. Horizontal composability is checked by evaluating the conditions defined in Chapter 4 for horizontal composition. The procedure *composable_horizontal()* returns the set of parameters $\mathcal{P}$ of *op* that are horizontally composable with op_i's parameters. The nodes *op*, And_k, and op_i are then linked by the following edges: $op \rightarrow \text{And}_k \rightarrow op_i$. The edge $op \rightarrow \text{And}_k$ is labeled with $\mathcal{P}$. The operation node *op*

is labeled with its outputs. The compound node is labeled with the inputs of *op*.

```
(00) UpdateFlowGraph(FG,op) {
(01)  for each operation node op_i ∈ FG do
(02)   if composable_horizontal(op,op_i,P)
(03)    then { add_arrow(op,And_k);
(04)           add_arrow(And_k,op_i);
(05)           add_label_arrow(op,And_k,P);
(06)           add_label_node(op,op.output);
(07)           add_label_node(And_k,op.input);
(08)           k = k + 1;    };
(09) }
```

Figure 5.15: Updating the Flow Graph

To illustrate the definition of a flow graph, we consider the example depicted in Figure 5.16. The example is related to the `get driving directions` sub-request (Figure 5.2). The operation `People Lookup` returns the *address* and *phone* number of a person (outputs), given her/his *name* and *state* of residence (inputs). The operation `Driving Directions Address` returns the *driving direction* (output) from a *source* to a *destination address* (outputs). Assume that `People Lookup` is horizontally composable with `Driving Directions Address`. Since `Driving Directions Address` requires two *address* parameters, it could be executed after executing `People Lookup` twice. The first execution of `People Lookup` returns the *source address* and the second execution gives the *destination address*.

The combination of `People Lookup` and `Driving Directions Address` as depicted in Figure 5.16 defines an "abstract operation" that returns the driving directions given the name and state of residence of two persons. The flow graph contains two operation nodes and one compound node. The arrow from `People Lookup` to And_1 contains the output parameter *address* twice. The compound node is labeled with the parameters *name*, *state*, *name*, and *state*. The operation node `Driving Directions Address` is labeled with the parameter *driving direction*.

We depict in Figure 5.17 the flow graph of the language translation example. We assume that `English-French` is horizontally composable with `French-Chinese`

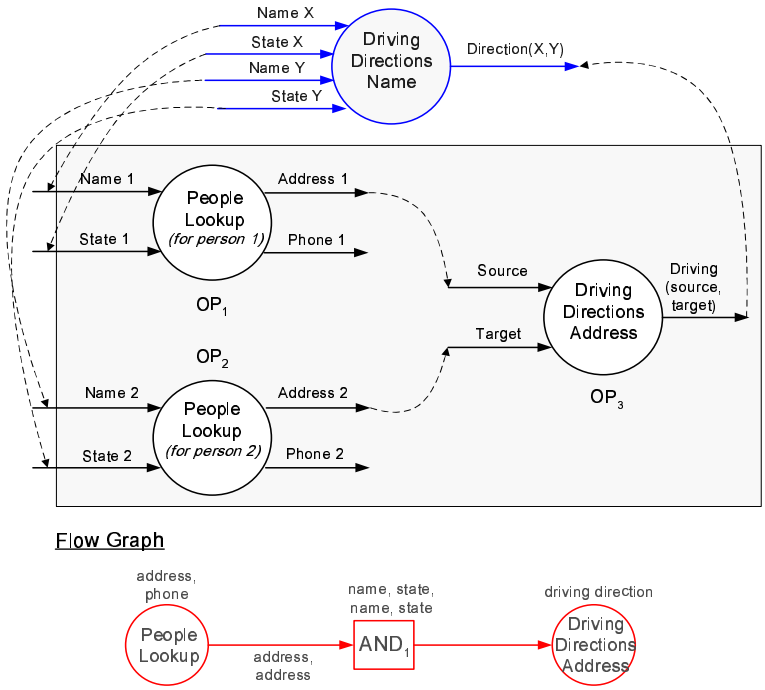

Figure 5.16: Flow Graph for the Driving Directions Example

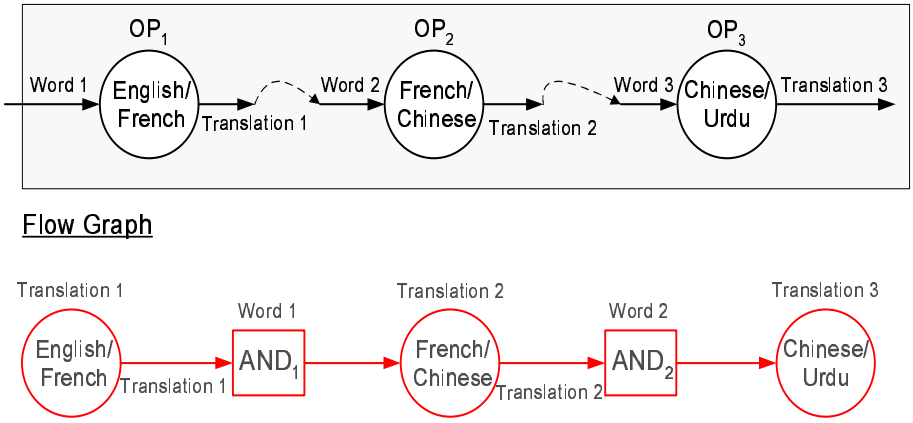

Figure 5.17: Flow Graph for the Language Translation Example

and that `French-Chinese` is horizontally composable with `Chinese-Urdu`. The flow graph corresponding to these operations contains two compound nodes And_1 and And_2 and three operation nodes. And_1 and And_2 are labeled with the parameters *English word* and *French word* respectively. `English-French`, `French-Chinese`, and `Chinese-Urdu` are labeled with the parameters *French word*, *Chinese word*, and *Urdu word* rspectively. The combination of the three operations as depicted in Figure 5.17 defines three abstract operations. The first operation provides translation from English to Chinese, the second operation enables translation from English to Urdu, and the third operation allows translation from French to Urdu.

The flow graph is used to determine composition plans through 1:N vertical composition (Figure 5.18). Let *SR* be a composition sub-request. The issue is how to determine the operations to be outsourced from *SR*. We identify two cases based on the mode of *SR*. If *SR* is an *In/Out* sub-request, then the matchmaking algorithm looks for a compound node, say $And_{initial}$, so that *SR*'s input is composable with $And_{initial}$'s input message. It also looks for an operation node, say op_{final}, so that op_{final}'s output message is composable with *SR*'s output message. The second case of the algorithm concerns *Out/In* sub-requests. The matchmaking algorithm looks, in this case, for a compound node, say $And_{initial}$, so that $And_{initial}$'s input message is message composable with *SR*'s output message. It also looks for an operation node, say op_{final}, so that op_{final}'s output message is message composable with *SR*'s input message. In both cases (*In/Out* and *Out/In* modes), $And_{initial}$'s predecessor is the initial operation of *SR*'s composition plan. Additionally, op_{final} is the final operation in that plan. To determine the intermediary operations, we calculate a *path* from $And_{initial}$ to op_{final} in the flow graph. We use the *breadth-first* search algorithm for this purpose [36].

To illustrate the 1:N composability algorithm, we consider the language translation example (Figure 5.17). Assume that *SR* (*In/Out* sub-request) aims at providing translation from English to Urdu. *SR*'s input and output are equal to the parameters *English word* and *Urdu word* respectively. *SR*'s input is message compatible with And_1's input and `Chinese-Urdu`'s output is message com-

```
(00) Vert_Comp1:N(SR_p, Registry, Plans1:n, Profile, nb_plans) {
(01)  while .true.  do {
(02)   if SR.mode = "In/Out"
(03)      then { compound = search_compound_composable(FG,SR.input,found1);
(04)             operation = search_operation_composable(FG,SR.output,found2); }
(06)   if SR.mode = "Out/In"
(07)      then { compound = search_compound_composable(FG,SR.output,found1);
(08)             operation = search_operation_composable(FG,SR.input,found2); }
(09)   if ¬found1 or ¬found2 then break;
(10)   path = breadth_first_search(FG,compound,operation);
(11)   Plans1:n = Plans1:n ∪ {path}
(12)   nb_plans = nb_plans + 1
(13) }   }
```

Figure 5.18: One-to-many Vertical Composition

patible with *SR*'s output. The algorithm then determines the path from And_1 to `Chinese-Urdu` using the *breadth-first* search algorithm. This gives the path And_1 → `French-Chinese` → And_2 → `Chinese-Urdu`. By replacing And_1 by its predecessor and eliminating And_2, the composition plan generated for *SR* is `English-French` → `French-Chinese` → `Chinese-Urdu`.

5.3 Generating Composite Service Descriptions

At the end of the matchmaking phase, one or several composition plans are generated for each composition sub-request. The aim of the generation phase is to replace each sub-request by a composition plans. Composers first select a composition plan for each sub-requests. Based on the selected plans, we generate a composite service description. The generation process is conducted in three steps. We first replace each sub-request by the selected composition plan. We then insert the pre-operations and post-operations of each operation of the plan. We finally compute *Quality of Composition* (*QoC*) parameters to assess the quality of the generated plan. Composers have the possibility to select other plan if they do not like the

quality of the generated composite service description.

5.3.1 Replacing Sub-requests by Composition Plans

Assume now that a composition plan is selected for each composable sub-request. Using those plans and the activity diagram (noted AD_s) specified by the composer, we generate an activity diagram (noted AD_g) for the composite service. Each sub-request SR is substituted by the operation in the corresponding plan. If the plan is obtained through 1:N composition, then the execution order of those operations in AD_g is similar to their execution order in the plan. Figure 5.19 illustrates the technique used for replacing a request in AD_s by its composition plan.

We sketch in Figure 5.20 the algorithm executed for replacing a subrequest SR by a plan P. Let SR_i and SR_j be a predecessor and succesor of SR in AD_s, respectively. We first create an activity A_i in AD_g for each operation op_i in P. For each arrow $op_i \rightarrow op_j$ in P, we then create a transition $A_i \rightarrow A_j$ in AD_g. The activity SR is then discarded. Finally, a transition is created from SR_i to the activity corresponding to the first operation in P. Likewise, a transition is created from the activity corresponding to the last operation in P to SR_j.

5.3.2 Inserting Pre and Post-Operations

Once, all sub-requests are replaced by their composition plans in the activity diagram, the final step of the generation phase is to include pre- and post-operations in AD_g. We determine the pre-operations and post-operations of every operation contained in AD_g. The idea is to "augment" each operation inserted in AD_g by its pre and post-operation relationships. We summarize in Figure 5.21, the algorithm used for that purpose.

Let op_i be an operation inserted in AD_g. The first step is to search for op_i in the OPD diagram. We use the well-known breadth-first algorithm for such task [36]. If op_i is found, we determine all operations that are related to op_i via pre and post operation relationships. For that purpose, we determine the set $\mathcal{P}$ of all OPD

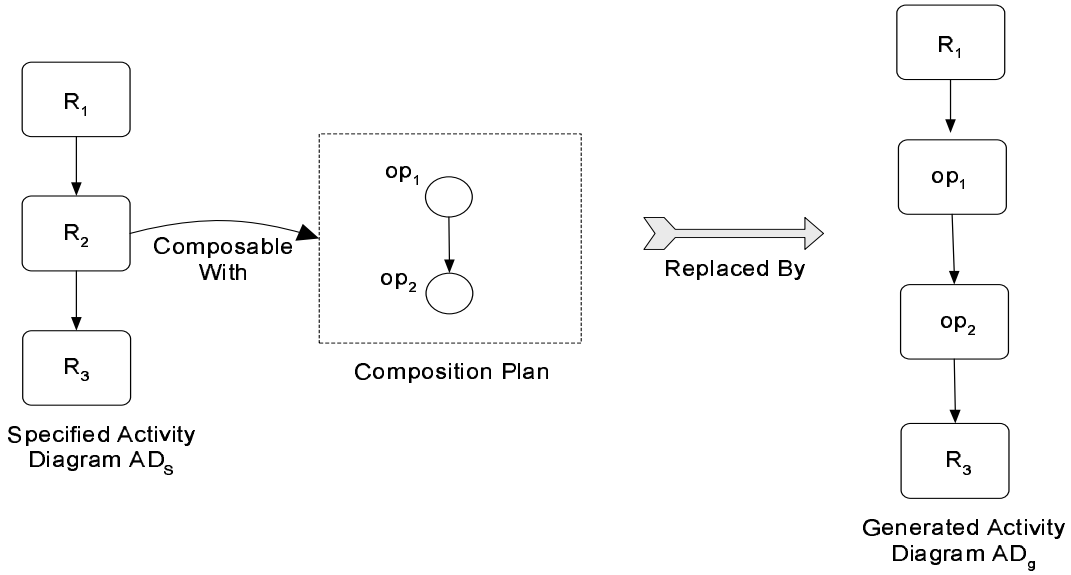

Figure 5.19: Replacing a Sub-request by its Composition Plan

(00) **Replace(*SR*,*Plan*)** {
(01) Let SR_i be a predecessor of SR in AD_s;
(02) Let SR_j be a succesor of SR in AD_s;
(03) Let op_1 be the first operation in *Plan*;
(04) Let op_n be the last operation in *Plan*;
(05) `for each` operation $op_i \in Plan$ `do`
(06) create an activity A_i;
(07) `for each` $op_i \rightarrow op_j$ in *Plan* `do`
(08) create a transition $A_i \rightarrow A_j$;
(09) create a transition $SR_i \rightarrow A_1$;
(10) create a transition $A_n \rightarrow SR_j$;
(11) }

Figure 5.20: Replacing a Sub-request by its Composition Plan

diagram's paths that contain op_i as a vertex. We use the Floyd-Warshall dynamic programming algorithm for computing these paths [36]. The OPD sub-diagram that contain all paths in $\mathcal{P}$ needs then to be inserted in AD_g. For that purpose, each arrow that exists in the sub-diagram is inserted into AD_g.

```
(00) Insert(AD_g,OPD) {
(01)    for each op_i ∈ AD_g do
(02)     if found(op_i, OPD)
(03)       then { sub-diagram = compute_paths(op_i, OPD);
(04)            for each op_i ∈ sub-diagram
(05)             create an activity A_i;
(06)            for each op_i → op_j ∈ sub-diagram
(07)             create a transition A_i → A_j in AD_g; }
(08) }
```

Figure 5.21: Inserting Pre and Post Operations

5.3.3 Quality of Composition

Once the activity diagram of a composite service is generated, the next step consists in checking its *quality*. We define a *Quality of Composition* (QoC) model for composite services. The model includes three parameters: *composition soundness*, *composition ranking*, and *composition completeness*.

Checking Composition Soundness – The first quality criterion we check is composition soundness (Chapter 4). A composition template CT is computed for the generated composite service. We use adjacency-matrix graph representation to model templates. Each composition template CT is represented by a $|OP(CT)| \times |P(CT)|$ matrix T where $OP(CT)$ is the set of operations in the template. The entry T_{ij} corresponds to the operations op_i and op_j. It is equal to 1 *iff* there is an arrow from op_i to op_j. It is equal to 0 otherwise. Stored templates are represented in the same way as composition templates.

We propose an algorithm for checking soundness based on comparing CT and ST matrices (Figure 5.22). The first step of the algorithm is the *unification* CT and ST vertices. This consists in adding to CT all vertices that belong to ST-CT and adding to ST all vertices that belong to CT-ST. In this way, both CT and ST will have the same set of vertices. The resulting templates are then represented by the matrices as explained previously. The next step consists in computing the classical

```
(00) Sound(CT,Stored,result) {
(01)     result = "not sound";
(02)     for each ST ∈ Stored do {
(03)       unification(CT,ST);
(04)       R = matrix(CT) - matrix(ST);
(05)       if R = 0 then result = "exactly sound";
(06)       if R > 0 then result = "strongly sound";
(07)       if R < 0 then result = "loosely sound";
(08)       if result ≠ "not sound" then break;
(09) }   }
```

Figure 5.22: Algorithm for Composition Soundness

substraction between matrices: R = matrix(CT) - matrix(ST). Four cases are then possible. If all elements in the matrix R are equal to 0, then CT and ST are similar. The composite service is hence exactly sound with respect to ST. If all elements in R are equal to 1/0, then ST is a subgraph of CT. The composite service is hence strongly sound with respect to ST. If all elements in R are equal to -1/0, then CT is a subgraph of ST. The composite service is hence loosely sound with respect to ST. If all elements in R are equal to -1, 1 or 0, then CT is not comparable with ST. CT should then be compared to another stored template (if any). If no stored template is left, then CT is not sound.

Composition Ranking – Assume that a composite service is sound with respect to stored template ST. We define a QoC parameter called *ranking* that gives an approximation of ST "importance". For each composite service, we determine its composition template CT. Assume that CT is *sound* with respect to a stored template ST_i. We define a function $\mathcal{R}$ ($\mathcal{R}$ stands for *reference*) so that $\mathcal{R}(\mathrm{ST}_i)$ gives the number of times that composite services that are sound with respect to ST_i have been created. This assumes that a ranking coefficient is maintained for each stored template. The *ranking* of CT with respect to ST_i is the proportion of references to ST. If the composite service is not sound, then its ranking is equal to zero. We give below the formula used to compute the ranking of a sound composite service (s is

the number of stored templates):

$$Ranking(CT, ST_i) = \frac{\mathcal{R}(ST_i)}{\sum_{k=1}^{s} \mathcal{R}(ST_k)}$$

Composition Completeness - This parameter, noted *CC*, is used to assess composite services that are not sound. It gives the proportion of composition sub-requests that are composable with participant service operations. *CC* allows the generation of composite service descriptions whose composition sub-requests are not "fully" composable with participant services. The value of *CC* is set by service composers in their profile and depends on their level of expertise. Indeed, if the value *CC* is relatively low (e.g., 25%), the algorithm might return composite services for which 75% of the composition sub-requests are not composable with participant service operations. In this case, composers may need to change their specification (e.g., data types) so that the desired sub-request can deal with participant services' features. The following formula defines the *CC* parameter for a composite service CS:

$$CC(CS) = \frac{\mid Composable(SubRequests) \mid}{\mid Sub - Requests \mid}$$

where *Composable*(SubRequests) is the set of sub-requests that are composable with participant services.

Chapter 6

Implementation and Performance Study

This chapter is devoted to the *implementation* and *performance study* of the proposed approach for composition [83, 17]. We provide an implementation of our techniques in *WebDG* prototype. We use social and welfare services within the *Family and Social Services Administration* (FSSA). Then, we provide an extensive performance study of our composition approach. For that purpose, we first define an analytical model for the proposed composition algorithms. We compare the performance of these algorithm using that model.

Another way to study the performance is through experiments. However, it is difficult to do such experiments in "real" Web service environments or prototypes (e.g., *WebDG*). Indeed, the availability of large and various numbers of services is not always guaranteed. Additionally, it is often desirable to measure performance for specific and various configurations of Web services (e.g., services with more than a certain number of operations) which is not always simple to achieve. To deal with this issue, we define a testbed for Web services called *Web Service Benchmarking Framework* (*WSBF*). We conduct an extensive experimental performance study using WSBF testbed.

This chapter is organized as follows. In Section 6.1, we describe our implemen-

tation in *WebDG* prototype. In Section 6.2, we present the salient features of the proposed Web service benchmarking framework. In Section 6.3, we describe our performance study. We first define our analytical model and then we describe the simulation experiments and their results.

6.1 WebDG Prototype

WebDG is built with a *holistic* perception of e-government Web services. It aims at providing a Web service middleware that enables citizens and case officers to access government e-services via a rich, uniform, and flexible interface. The design and development of *WebDG* is directed by two key guidelines: the use of Web service standards (e.g., WSDL, SOAP, and UDDI) and extensibility (i.e., ability to add new functionalities). Our focus in *WebDG* is on implementing the composition techniques proposed in this dissertation.

6.1.1 WebDG Services

As a proof of concept, we implemented several FSSA applications in *WebDG*. These include *WIC*, *Medicaid*, and *TOP*. *WebDG* applications, developed in Java (JDK 1.3), access databases (Oracle 8.0.5) to retrieve/store government and citizens' information. *WIC* (Women, Infant, and Children) provides Federal grants to States for supplemental food, health care referrals, and nutrition education for low-income pregnant, breastfeeding, and non-breastfeeding postpartum women. It also provides nutritional assistance to infants and children who are found at nutritional risk. *Medicaid* is a jointly-funded (Federal and State) health insurance program for specific groups of low-income and needy people. It covers 36 million individuals including children, senior citizens, blind, and/or disabled, and people eligible for federally assisted payments. *TOP* (Teen Outreach Pregnancy) provides pregnant teens with childbirth and postpartum educational support, housing information, and adoption services.

To enable access to the aforementioned applications in *WebDG*, we "wrapped" them into Web services. We use state-of-the-art technologies for implementing *WebDG*. Table 6.1 gives a summary of these technologies. *WebDG* services are deployed using *Apache SOAP* (2.2). *Apache SOAP* provides not only server-side infrastructure for deploying and managing service, but also client-side API for invoking those services. Each service has a *deployment descriptor*. The descriptor includes the unique identifier of the Java class to be invoked, session scope of the class, and operations in the class available for the clients. Each service is deployed using the *service management client* by providing its descriptor and the URL of the *Apache SOAP servlet rpcrouter*.

Product	*Version*	*Role*
Oracle	8.0.5	WIC, Medicaid, and TOP databases. Used also as a template repository.
Java	JDK 1.3	WIC, Medicaid, and TOP "legacy" applications.
JDBC	2.0	Database connection.
Apache SOAP	2.2	SOAP server.
Systinet WASP UDDI	Standard 3.1	UDDI server.
J2EE Cloudscape	4.0	UDDI registry.
IBM WSTK	2.0	Convert applications into WSDL descriptions.
Tomcat	3.2.1	Web server and Java Servlet engine.

Table 6.1: Enabling Technologies

We use WSDL language to describe *WebDG* services. WSDL descriptions are extended with semantic features defined in the community ontology (e.g., purpose, business role). We use IBM Web Service Tool Kit (WSTK) to automatically generate WSDL files for Web services from Java class files. These WSDL files are accessible through tModels stored in the UDDI registry. We adopt *Systinet's WASP UDDI Standard 3.1* as our UDDI toolkit. *Cloudscape* (4.0) database is used as a UDDI registry. WASP UDDI provides client-side API (Application Programming Interface) for publishing and inquiring about Web services. Each Web service has one or more operations. For example, WIC service has an operation `check_eligibility` which takes a citizen's eligibility information as input (e.g., salary, family size) and checks this information against categorical (e.g., pregnant women), financial (e.g.,

household income), and nutritional (e.g., children at risk) criteria. If the citizen's eligibility information satisfies all of the three criteria, the operation updates the eligibility status for this citizen to "eligible, wait for approval". Otherwise, the operation updates the status to "not eligible". In both cases, a message is returned to the customer.

6.1.2 Architecture

WebDG system is implemented across a network of *Solaris* workstations. We present the architecture of *WebDG* in Figure 6.1. The architecture is organized into four layers. The first layer contains a set of Oracle databases that store government and citizens' data. The second layer includes "proprietary" applications (e.g., WIC application) developed in Java. Each application access databases from the first layer. The third layer contains Web services. The Web service are either simple or composite. Simple Web services wrap proprietary applications defined in the lower layer. Composite services are automatically generated by the *WebDG manager*. The upper layer includes a *Graphical User Interface* (GUI) and *WebDG manager*. Citizens and case officers access *WebDG* via a GUI implemented using HTML/Servlet. Three types of requests are submitted through the GUI: discovery, invocation, and composition.

The *WebDG manager* is at the core of *WebDG* system. It is composed of several modules: *Request Handler*, *Matchmaker*, *Service Locator*, *One-to-One Composer*, *One-to-Many Composer*, *generator* and *QoC Manager*. The request handler is the router of the WebDG manager. The task it performs depends on the type of request it receives. If the request type is "discovery", it forwards it to the service locator which implements *UDDI Inquiry Client* using WASP UDDI API. If the request type is "invocation", the request handler invokes the corresponding operation through *SOAP Binding Stub* which is implemented using Apache SOAP API.

If the request type is "composition", the request handler forwards the composition specification (XML document) to the matchmaker. The matchmaker uses *JAXP*

(*Java API for XML Processing*) to parse CSL specifications, check their correctness, and decompose them into sub-requests. For each sub-request, the matchmaker interacts with the service locator to get operations from the service registry. Then, it forwards these operations to the one-to-one composer which implements one-to-one algorithms for vertical composition. We currently consider the operation-centric algorithm.

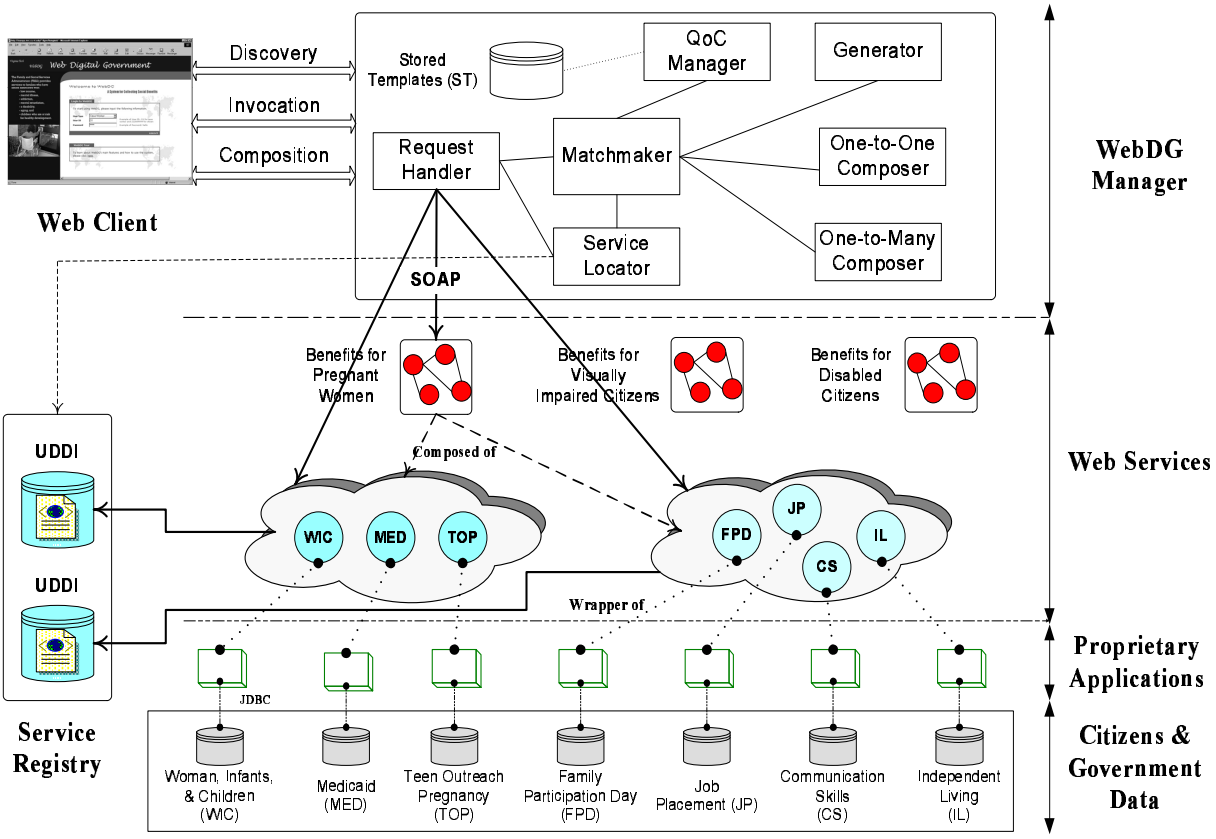

Figure 6.1: WebDG Architecture

At the end of its execution, the one-to-one composer returns a set of operations. Each operation could be used as a composition plan. If all operations in the registry are checked and the maximum number of plans is not reached, the matchmaker in-

vokes the one-to-many composer which implements the flow graph-based algorithm. At the end of its execution, the one-to-many composer returns a set of horizontally composed operations that could be used as composition plans. The matchmaker returns all generated plans to the user via the request handler. The user selects a composition plan for each sub-request and returns it to the matchmaker. The matchmaker sends the selected plans to the generator which augments them with pre and post-operations. The resulting composite service is forwarded to the QoC manager which checks QoC parameters. The composite service along with the values of its QoC parameters are finally returned to the user.

6.1.3 WebDG Scenario

We present a scenario that illustrates the main features of *WebDG*. We consider the case of a pregnant teen *Mary* visiting case officer *John* to collect social benefits to which she is entitled. *Mary* would like to apply for a government funded health insurance program. She also needs to consult a nutritionist to maintain an appropriate diet during her pregnancy. As *Mary* will not able to take care of the future newborn, she is interested in finding a foster family.

The fulfillment of the teen's needs requires accessing different services scattered in and outside the local agency. For that purpose, the case manager may either look for simple Web services that fit specific needs of the teen or specify all teen's needs through one single composite Web services. In the following, we describe the main steps for discovering and composing Web services using *WebDG*.

- **Step 1: Web Service Discovery** – To locate a specific Web service, *John* could provide either the service name, if known, or properties. This is achieved by selecting the "By Program Name" or "By Program Properties" nodes respectively (Figure 6.2). *WebDG* currently supports two properties: *Category* and *Agency*. Assume *John* is interested in a service that provides help in finding foster families. He would select the *adoption* and *pregnancy* categories and the *Division of Family and Children* agency. *WebDG* would return the `Teen`

`Outreach Pregnancy` (`TOP`) service. `TOP` offers childbirth and postpartum educational support for pregnant teens.

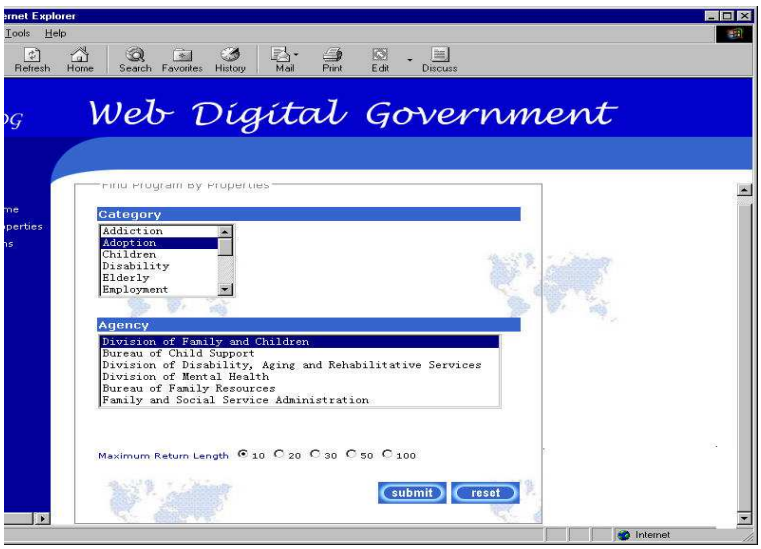

Figure 6.2: Discovering E-Government Services

- **Step 3: Composing Web Services** – Assume now *John* wants to specify *Mary*'s needs through one single composite service called `Pregnancy Benefits` (`PB`). For that purpose, he selects the "Advanced Programs" node (Figure 6.3). Examples of `PB`'s operations include `Find Available Nutritionist`, `Find PCP Providers` (which looks for primary care providers), and `Find Pregnancy Mentors`. After checking composability rules, *WebDG* would return *composition plans* that conform to `BP` specification. Each plan has an ID (number). *John* would click on the plan's ID to display the list of outsourced services. In our scenario, `WIC` (a federally funded food program for `Women, Infants, and Children`), `Medicaid` (a healthcare program for low income citizens and families), and `TOP` services would be outsourced by `PB`.

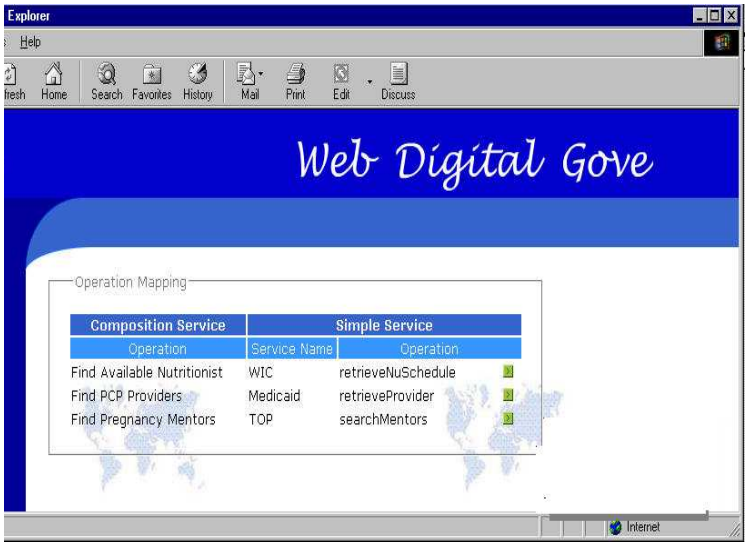

Figure 6.3: Mappings of the Composite Service Operations

6.2 Web Service Benchmarking Framework

WSBF provides simple and efficient means for generating a large number of service descriptions based on the WSDL standard. It uses as input a set of parameters provided by the user (e.g., number of services to be generated and their arrival rate). It returns a list of XML files containing the generated WSDL descriptions. These descriptions are stored in a service registry specified by the user. The main features of WSBF include the following:

- *Customized Generation*: WSBF allows user-controlled generation of Web service descriptions. Users have the ability to specify the modes of service operations. They also give as input the taxonomy to be used for categorizing Web services. This allows the generation of Web services for various application domains such as E-commerce and e-government. Finally, users specify the data type system (e.g., XML Schema) to be used for message parameters.

- *Use of Statistical Models*: WSBF generates service descriptions at an arrival rate that follows a statistical model specified by the user. Statistical models are also used to remove services from the registry. We use *Systinet's WASP UDDI Standard 3.1* as our UDDI toolkit. *Cloudscape* (4.0) database is used as a UDDI registry. Additionally, WSBF enables users to specify the range and statistical models for several quantitative attributes such as the number of operations per service, messages, and parameters per message.

- *Extensibility*: WSBF is extensible at different levels. For example, new statistical models such as *Geometric* and *Poisson* can be plugged into WSBF. Additionally, new service attributes that are not in the current WSDL version can be added to the generation process (e.g., semantic attributes defined for generic operations).

6.2.1 Statistical Distribution Models in WSBF

WSBF currently includes four (4) statistical distribution models for randomly generating quantitative attributes: *Gaussian*, *Poisson*, *Uniform*, and *Binomial* [105]. The algorithms used in WSBF to implement the aforementioned statistical models are mostly adopted from [105]. In the models given below, we use the function *Random()* from Java's *Random* class to generate a random real number between 0 and 1:

- *Gaussian Distribution*: In this model (also known as *Normal Distribution*), the generated data follows a bell-shaped distribution. This distribution is and completely determined by its *mean* and *standard* Deviation. We use the Java's built-in random number generator (*nextGaussian()* method of *Random* class) to generate numbers according to this model.

- *Binomial Distribution*: This model measures the number of times that a particular event will occur in a sequence of observations. The binomial distribu-

tion is specified by the number of observations and the probability of event occurrence.

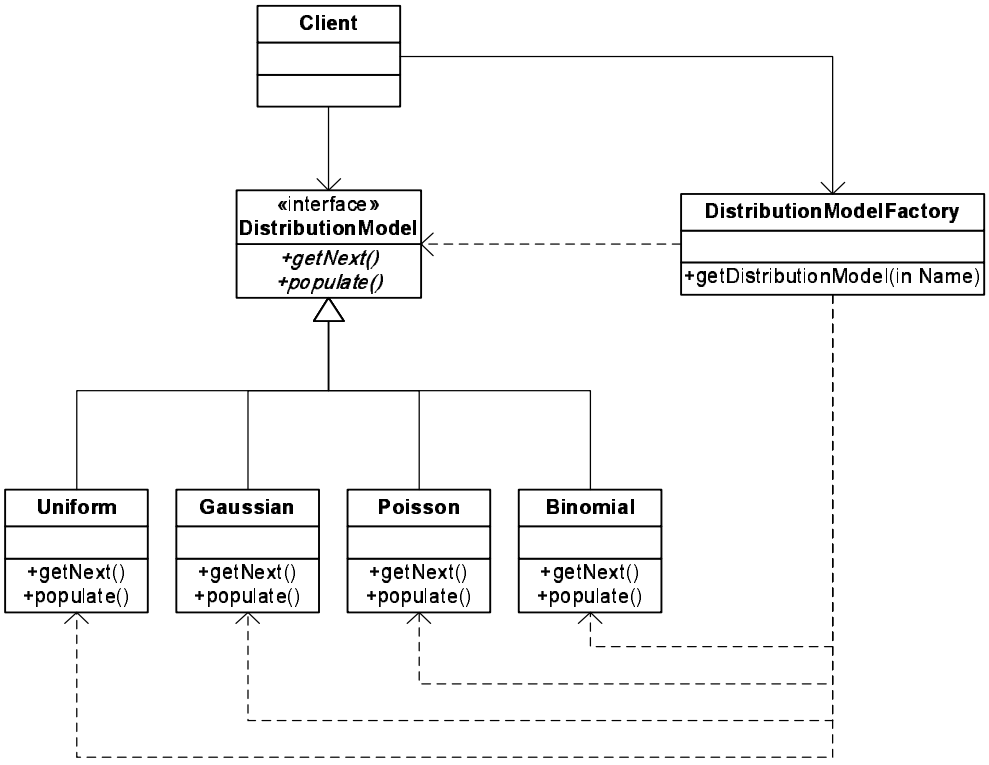

Figure 6.4: UML Class Diagram for Statistical Models

- *Poisson Distribution*: This model is often used to measure the number of events in a specific time period. It is often used to describe situations in which the probability of an event is small and the number of opportunities for the event is large. *Poisson* is an extension of the binomial distribution in which the number of samples is infinite.

- *Uniform Distribution*: A uniform distribution is one for which the probability of occurrence is the same for all integer values within an interval of values.

We depict in Figure 6.4 the UML class diagram for WSBF's distribution models. The *Client*, which may be the *Generator* or *Publisher*, interacts only with *Distribu-*

tionModel and *DistributionModelFactory*. *DistributionModel* is an abstract interface of concrete distribution models, such Uniform, Gaussian and Poisson. *DistributionModelFactory* creates concrete distribution model depending on the model's name. Note that more distribution models can be used without changing the *Client* class.

6.2.2 Generation Flow

The process of generating Web services includes two phases (Figure 6.5): the *input* phase and *generation* phase. In the *input* phase, users provide a set of parameters related to *services* as well as their *operations*, *port types*, and *messages*. The *generation* phase uses these parameters to randomly produce WSDL service descriptions. These descriptions are automatically stored in a service repository. The location of the repository is given by users as part of their input parameters. *Service* parameters include the number of WSDL descriptions to be generated and the arrival rate of Web services. The arrival rate represents the duration (in seconds) between the generation of two consecutive services. This parameter is particularly important to simulate the dynamics of Web service environments. To enable random arrival times, users specify the minimum and maximum durations. After each service generation, the arrival time for the next service is computed by randomly selecting a value between the minimum and maximum durations. This selection is based on a statistical distribution model (e.g., *Poisson*) specified by users.

Operation parameters include the number of operations within a service. To generate services with various numbers of operations, users specify the range (minimum and maximum) of possible values for this parameter. A random number is generated within this range for each service based on a user-specified distribution model. For the operation modes, users have the choice between specifying a fixed value for all operations (e.g., *In/Out*) or requesting a random generation of modes. The first *port type* parameter is the number of port types. This parameter is randomly generated using a range and statistical model similarly to the number of operation. Another *port type* parameter specified by users is the binding protocols supported by the

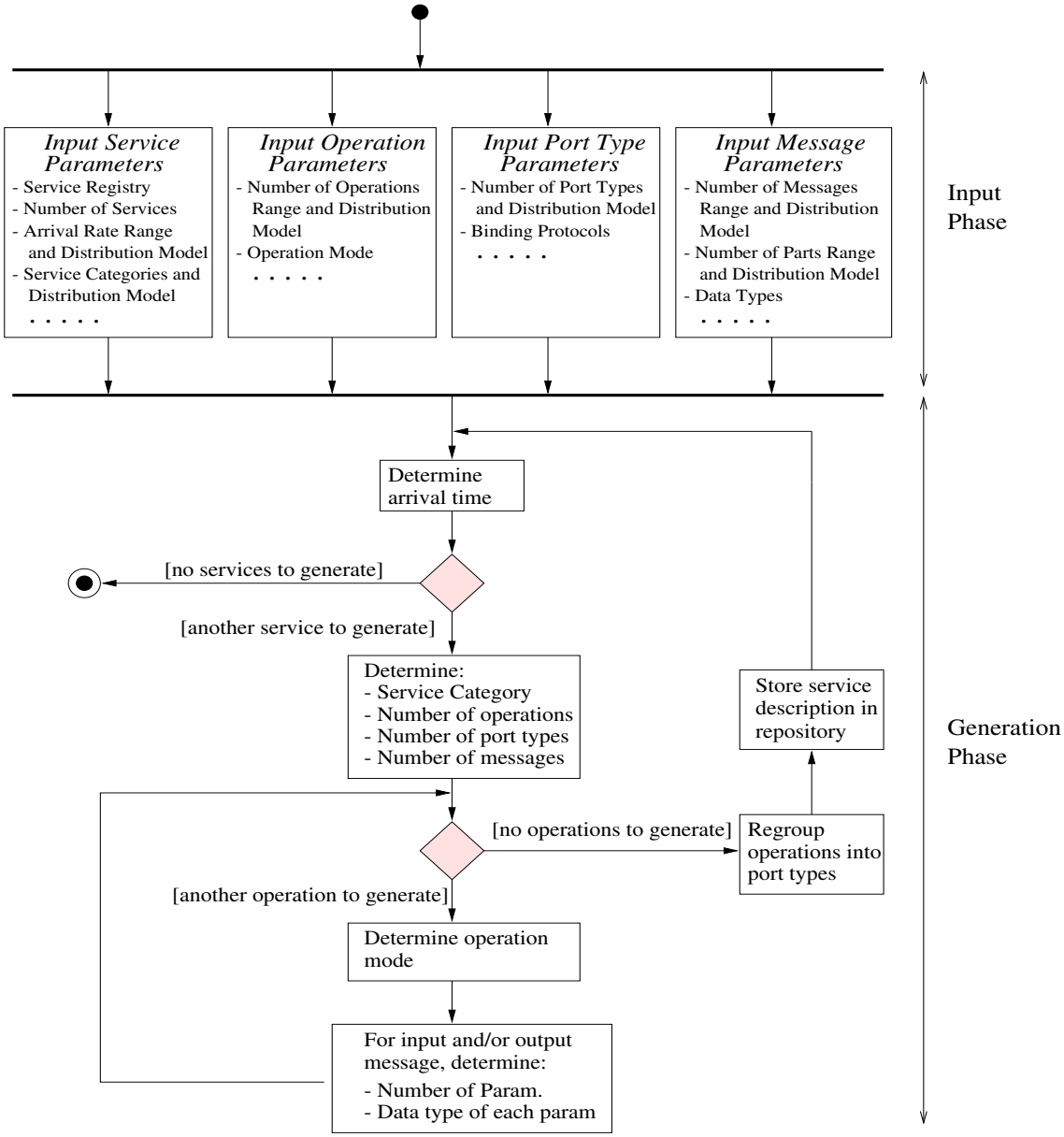

Figure 6.5: UML Activity Diagram for Web Service Generation

generated services. Finally, *message* parameters contain the number of messages within a service, the number of parameters per messages, and the parameters' data types. The number of messages and parameters are randomly generated similarly to the number of operations and port types. For the parameters' data types, users provide a list of types (e.g., XML Schema). A data type is then randomly selected and assigned to each parameter.

The *generation* phase starts once all input parameters are given. For more flexibility, users have the possibility to leave some input parameters unspecified. A default value is then assigned to those parameter. For example, *Uniform* is the default statistical distribution model. During the second phase, a category, number of operations, number of port types, and number of messages are first randomly generated for each service. Then, the generator determines the mode of each operation either randomly or based on the value set by the user. It also randomly selects the number of parameters and data types within input and output messages. Once all operations are generated, they are randomly grouped into port types. The whole WSDL description is stored in the repository. Finally, an arrival time is computed before generating the next service description.

6.2.3 Architecture

WSBF is fully implemented in Java and includes four components (Figure 6.6): the *Random Number Generator*, *Generator*, *Analyzer*, *Publisher*, and GUI (*Graphical User Interface*). The *Random Number Generator* consists of four modules corresponding to different statistical distribution models: *Gaussian*, *Poisson*, *Binomial*, and *Uniform*.

The *Generator* is the core of WSBF. It is composed of an *interface*, *service definer*, and *history log*. The *interface* takes user's input from the GUI and obtains necessary random numbers from the random number generator (e.g., number of operations for each service). User specified values (e.g., number of services) are directly forwarded to the *service definer*. This module is in charge of generating

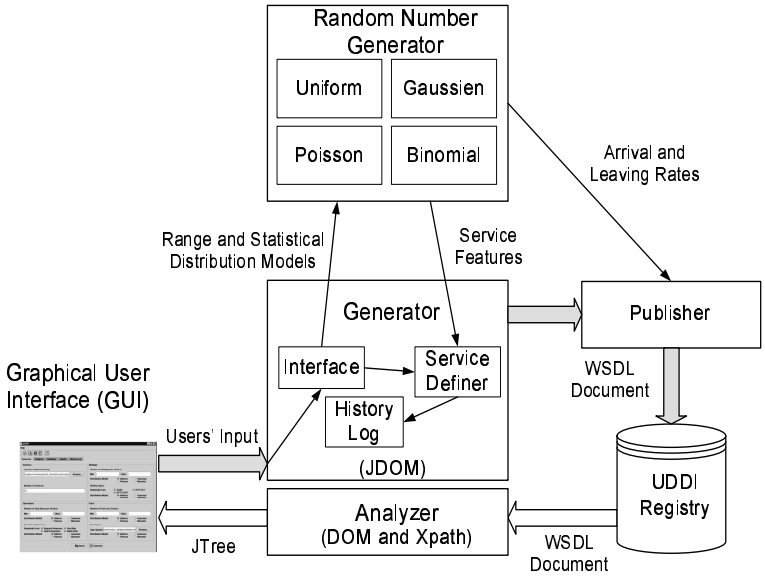

Figure 6.6: WSBF Architecture

WSDL descriptions and ensuring their syntactic correctness. WSDL descriptions are manipulated using *JDOM*, a Java-based (*Document Object Model*) for XML documents. *JDOM* provides means to represent XML documents for easy manipulation. Information, such as the time to generate each WSDL description, is stored in the *history log*. This information is useful for monitoring the generation process. Once a WSDL description has been generated, it is forwarded to the *publisher* which, based on the service arrival rate, stores a tModel of the description into the UDDI registry. The publisher also uses the leaving rate to remove service description's tModel from the registry.

Each generated description can be viewed through the *WSDL analyzer*. The aim of the *analyzer* is twofold. First, it extracts service attributes such as category, operations, and input/output messages. Second, it provides statistical information about each service description, such as the number of operations and the average number of parts per message. The analyzer uses *DOM* (*Document Object Model*)

and *XPath* technologies. A WSDL document is first parsed by a *DOM* handler. Then a tree-like structure is built and parsed by an *XPath* engine (*Jaxen*). As a result, an *XPath*-ready structure is constructed. Finally, the analyzer uses *XPath* to parse the service description.

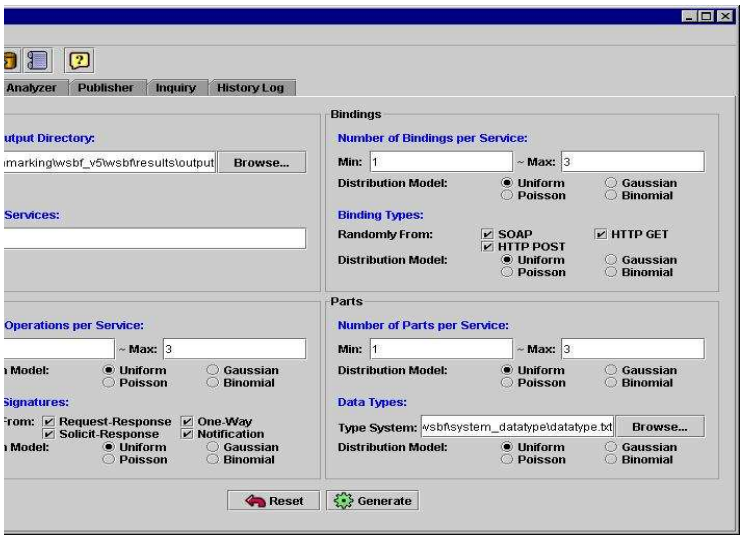

Figure 6.7: WSBF Generation Interface

The GUI was developed using *Java Swing*. It includes the *generator*, *analyzer*, and *publisher tabs*. Figure 6.7 depicts the service generator *tab*. It includes four panels: *Services*, *Operations*, *Bindings*, and *Parts*. These panels provide means for the specification of users' input.

The service analyzer *tab* displays WSDL documents in a user-friendly way (*JTree*, a tree-like structure (Figure 6.8). Service descriptions may also be displayed in text format. The analyzer also returns several statistics such as the number of operations per mode and the average number of parameters per message. The statistical analysis is currently done during generation. The statistical results are then stored in statistical files. Clicking on the analyzer *tab* allows the display of those statistics in a tabular format.

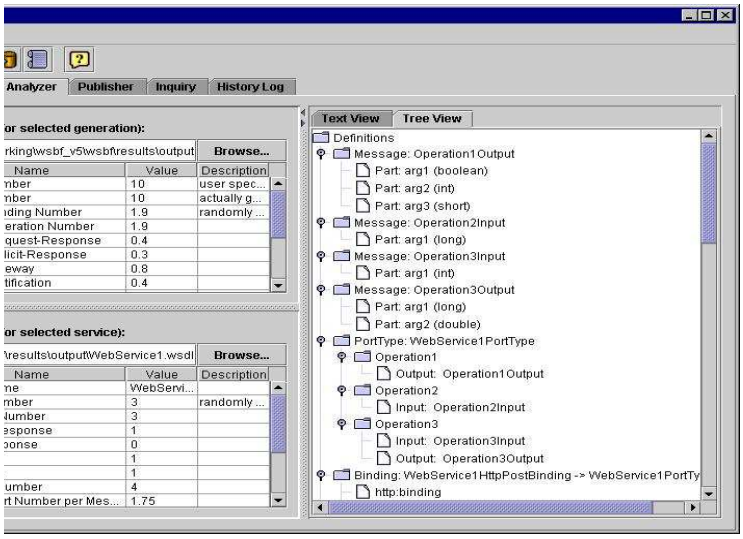

Figure 6.8: WSBF Analyzer Interface

The publisher *tab* enables users to select the service registry that would store the generated descriptions (Figure 6.9). Users may also specify the list of service categories and the number of business services in the registry. The dynamic aspect of Web services is "simulated" by specifying the arrival and leaving times for a Web service in the registry.

6.3 Performance Study

The aim of our performance study it twofold. First, we show the scalability of our approach by computing the composition time for large number of Web services. Second, we compare the performance of the different composition algorithms. Our focus in this section is on one-to-one composition since one-to-many composition does not involve access to the service registry and searching in the service space. We conduct our performance study using the following methodology. We first define an analytical model for the proposed algorithms. We compare the algorithms using

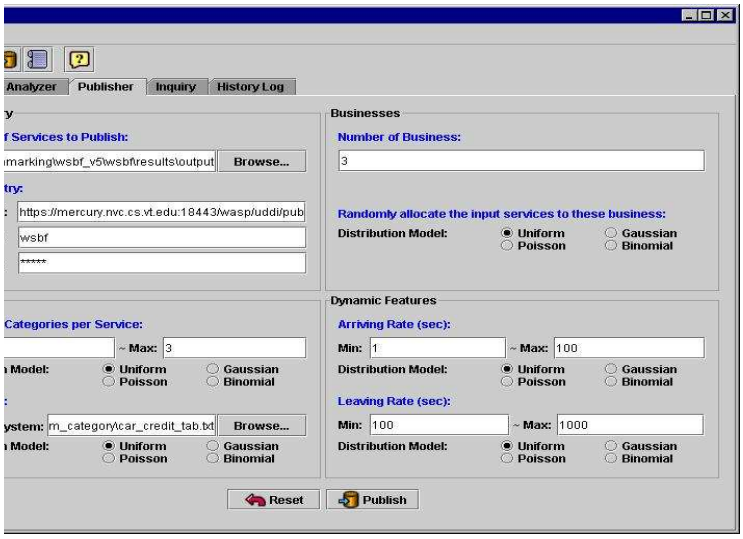

Figure 6.9: WSBF Publisher Interface

that model. Then, we compare the same algorithms based on simulation. We use WSBF testbed for that purpose. We finally compare the results obtained in the analytical and experimental studies. Comparing them can prove/disprove the correctness of our testbed and the predictiveness of the proposed analytical model.

6.3.1 Analytical Model

In this section, we present the analytical model for each composition algorithm. The total composition time for each algorithm will be the sum of the global time for checking syntactic composability (T_{ST}), global time for checking static semantic composablity (T_{SS}), and global time for checking dynamic semantic composability. Table 6.2 defines the parameters and symbols used in this section. The composition time can be expressed as follows:

$$T = T_{ST} + T_{SS} + T_{DS}$$

$$T_{min} = T_{ST}^{min} + T_{SS}^{min} + T_{DS}^{min}$$
$$T_{max} = T_{ST}^{max} + T_{SS}^{max} + T_{DS}^{max}$$

Variables	
N_{op}	Number of operations in the registry
N_C	Number of communities
N_{opc}	Number of operations per community
P_{max}	Maximum number of plans
C_{SR}	Number of sub-request in a composition
P_M	Number of parameters per message
B_{op}	Number of business rules per operations
TC	Number of terms per condition
A_F	Number of final states in the automaton
I_R	Number of inference rules
Performance measurement parameters and functions	
U_t	Time to obtain description from UDDI registry
O_t	Time to parse a service description
T	Total composition time
T_{min}	Minimum total composition time
T_{max}	Maximum total composition time
T_{ST}	Total syntactic composability time
T_{SS}	Total static semantic composability time
T_{DS}	Total dynamic semantic composability time
T_{ST}^{min}	Minimum total syntactic composability time
T_{SS}^{min}	Minimum total static semantic composability time
T_{DS}^{min}	Minimum total dynamic semantic composability time
T_{ST}^{max}	Maximum total syntactic composability time
T_{SS}^{max}	Maximum total static semantic composability time
T_{DS}^{max}	Maximum total dynamic semantic composability time
T_{msg}	Time to check message composability for one operation
t_{ST}	Time to check syntactic composability for one operation
t_{SS}	Time to check static semantic composability (operation level) for one operation

Table 6.2: Symbols and Parameters

For each composition algorithm, we compute the average execution time. Thus, T is equal to $(T_{min} + T_{max})/2$. To simplify the analysis, we assume that the times to a description from a UDDI registry and parse that description are fixed values. It is also reasonable to assume that times to check syntactic and static semantic (at the operation level) composability for an operation are fixed values. In contrast, message and behavioral composability times depend on the number of message parameters and business logic rules, respectively. Thereby, U_t, O_t, t_{ST}, and t_{SS} are constants.

In the rest of this section, we will show how to derive composition time for each algorithm.

Operation-Centric Algorithm

Let start by computing the minimum composition time T_{min}. This time corresponds to the case where P_{max} iterations of the algorithm are executed. This means that we get a composition plan after each iteration. The total syntactic composability time T_{ST}^{min} is then equal to $P_{max} \times t_{ST}$. The total static semantic composability time T_{SS}^{min} is equal to $P_{max} \times (t_{SS} + T_{msg})$. Let us now compute the time T_{msg} for checking message composability. T_{msg} refers to the time of comparing a pair of message twice. At minimum, each parameter in a message would be compared to one parameter of the dual message. Hence, T_{msg} is equal to $2 \times P_M$ and T_{SS}^{min} is equal to $P_{max} \times (t_{SS} + 2 \times P_M)$. The last time to compute is the total time for dynamic semantic composability T_{DS}^{min}. Since the algorithm performs vertical composition, dynamic semantic composability does not check plugin prematch rule. Based on the definition of B-composability rules, there is a need to execute the theorem prover nine (9) times: one for plugin postmatch, two for exact postmatch, two for plugin, and four for exact B-composability. At minimum, each business rule would be proved using the first inference rule. T_{DS}^{min} is then equal to $P_{max} \times (9 \times B_{op} \times TC)$. Based on the above analysis, we have:

$$T_{min} = P_{max} \times (U_t + O_t + t_{ST} + t_{SS} + 2 \times P_M + 9 \times B_{op} \times TC)$$

The maximum composition time refers to the case where all operations in the registry are checked. This means that the number of iterations executed by the algorithm is equal to N_{op}. Consequently, T_{ST}^{max} and T_{SS}^{max} are equal to $N_{op} \times t_{ST}$ and $N_{op} \times (t_{SS} + T_{msg})$, respectively. Let us now compute the formula for T_{msg}. The first parameter of each message should be compared to P_M parameters of the other

message. The next parameter needs to be compared to P_M - 1 parameters, and so on. Hence, P_M is equal to the following:

$$T_{msg} = 2 \times (P_M + (P_M - 1) + ... + 1) = P_M \times (P_M + 1)$$

We need now to compute the total time for dynamic semantic composability T_{DS}^{max}. As mentioned previously, we execute the theorem prover nine (9) times to check the different B-composability rules. We compare each rule in the sub-request with all rules of the current operations. The total of comparisons is then B_{op}^2. For each term, we need to go through all inference rules for a total of $TC \times I_R$. T_{DS}^{max} is then equal to $N_{op} \times (9 \times B_{op}^2 \times TC \times I_R)$. The maximum composition time is then given below:

$$T_{max} = N_{op} \times (U_t + O_t + t_{ST} + t_{SS} + P_M \times (P_M + 1) + 9 \times B_{op}^2 \times TC \times I_R)$$

The previous formulas give composition times for one sub-request. In what follows, we specify the total execution time for the operation-centric algorithm when executed on C_{SR} sub-requests. In this case, we consider the times to access the registry and parse operation descriptions:

$$\begin{aligned}
T_{ST} &= \frac{1}{2} \times t_{ST} \times C_{SR} \times (N_{op} + P_{max}) \\
T_{SS} &= \frac{1}{2} \times N_{op} \times C_{SR} \times (t_{SS} + P_M \times (P_M + 1)) + \frac{1}{2} \times P_{max} \times C_{SR} \times (t_{SS} + 2 \times P_M) \\
T_{DS} &= \frac{9}{2} \times N_{op} \times C_{SR} \times (B_{op}^2 \times TC \times I_R) + \frac{9}{2} \times P_{max} \times C_{SR} \times B_{op} \times TC \\
T &= \frac{1}{2} \times N_{op} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + P_M \times (P_M + 1) + 9 \times B_{op}^2 \times TC \times I_R) \\
&\quad + \frac{1}{2} \times P_{max} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + 2 \times P_M + 9 \times B_{op} \times TC)
\end{aligned}$$

We compare in Figure 6.10 the times for syntactic (T_{ST}), static semantic (T_{SS}), and dynamic semantic composition (T_{DS}). To enable a better visualization of the figure, we represent T_{ST}, T_{SS}, and T_{DS} using logarithm function. The graph shows that most of the composition time is spent on checking static and dynamic composability. Indeed, B-composability (dynamic semantics) compares all business logic rule of each sub-request with the business logic rules of target operations. Similarly, static semantics compares all message parameters of each sub-request with message parameters of target operations. This is in contrast with syntactic composability which performs three comparisons for each pair (sub-request,target): binding, mode, and number of parameters.

Community-Centric Algorithm

The minimum composition time T_{min} for the community-centric algorithm corresponds to the case where all outsourced operations are obtained during the first P_M iterations. It is thereby similar to that of the operation-centric algorithm:

$$T_{min} = P_{max} \times (U_t + O_t + t_{ST} + t_{SS} + 2 \times P_M + 9 \times B_{op} \times TC)$$

The maximum composition time T_{max} refers to the case where all operations are checked for composability. However, in contrast to the operation centric algorithm, only the operations within a specify community are checked. This means that the number of iterations executed by the algorithm equals N_{opc}. Let us now derive the relationship between N_{opc} and N_{op}. We assume that the operations within a registry are uniformly distributed across communities. Hence, $N_{opc} = \frac{N_{op}}{N_C}$. T_{max} can now be derived from the formula of the operation-centric algorithm by replacing N_{op} by $\frac{N_{op}}{N_C}$:

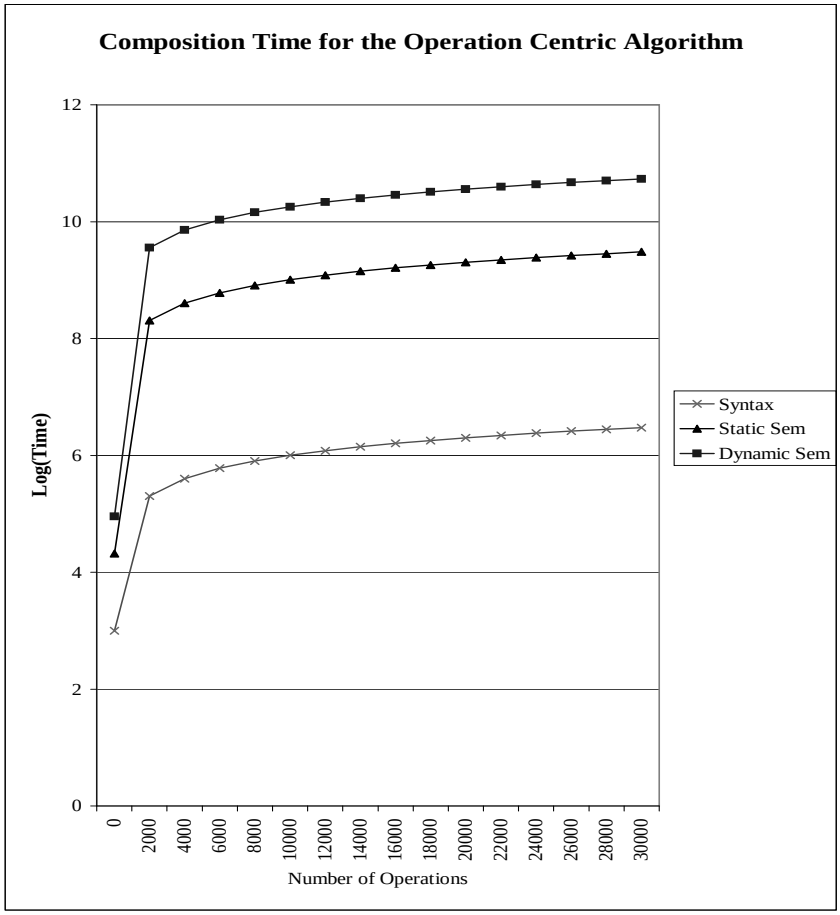

Figure 6.10: Composition Time for the Operation Centric Algorithm

$$T_{max} = \frac{N_{op}}{N_C} \times (U_t + O_t + t_{ST} + t_{SS} + P_M \times (P_M + 1) + 9 \times B_{op}^2 \times TC \times I_R)$$

Finally, the composition times are as follows:

$$T_{ST} = \frac{1}{2} \times t_{ST} \times C_{SR} \times (\frac{N_{op}}{N_C} + P_{max})$$

$$T_{SS} = \frac{N_{op}}{2 \times N_C} \times C_{SR} \times (t_{SS} + P_M \times (P_M + 1)) + \frac{1}{2} \times P_{max} \times C_{SR} \times (t_{SS} + 2 \times P_M)$$

$$T_{DS} = \frac{9 \times N_{op}}{2 \times N_C} \times C_{SR} \times (B_{op}^2 \times TC \times I_R) + \frac{9}{2} \times P_{max} \times C_{SR} \times B_{op} \times TC$$

$$T = \frac{N_{op}}{2 \times N_C} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + P_M \times (P_M + 1) + 9 \times B_{op}^2 \times TC \times I_R) + \frac{1}{2} \times P_{max} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + 2 \times P_M + 9 \times B_{op} \times TC)$$

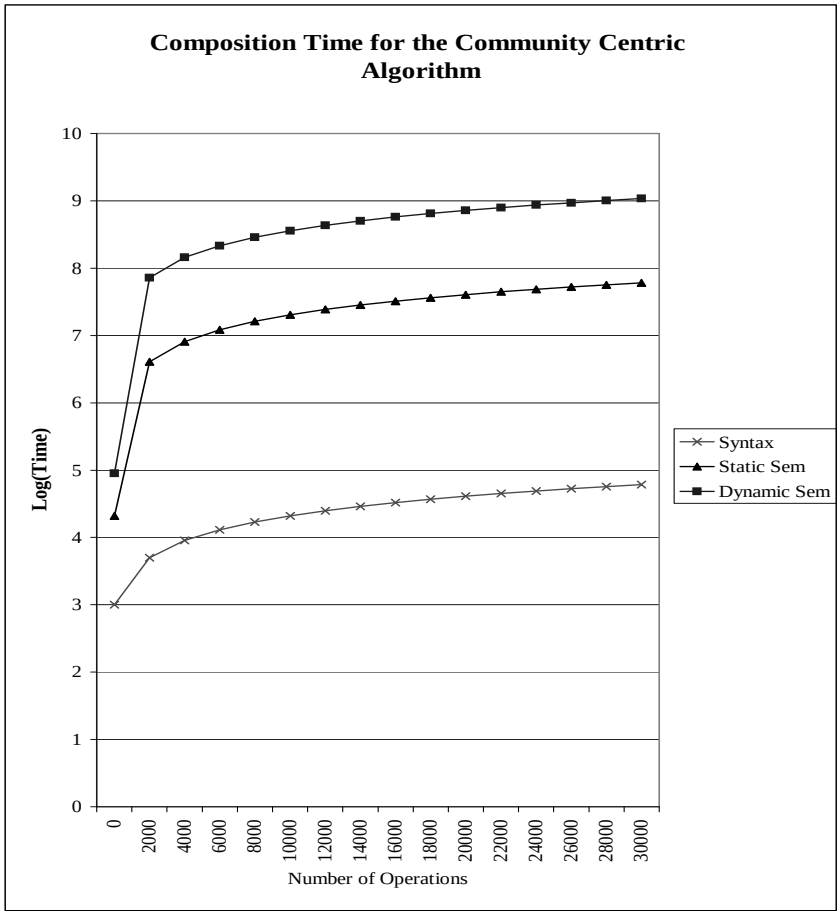

Figure 6.11: Composition Time for the Community Centric Algorithm

We compare in Figure 6.11 the times for syntactic (T_{ST}), static semantic (T_{SS}), and dynamic semantic composition (T_{DS}). As for the operation-centric algorithm, the algorithm spends most of the time in checking static and dynamic semantic com-

posability rules. Figure 6.12 shows that the number of communities has an impact on the composition time. We consider two cases where the number of communities is 50 and 500. The figure states that the composition time for $N_C = 50$ is higher than then composition time for $N_C = 500$. The bigger is the number of communities, the smaller is the composition time. Indeed, the number of operations with which the algorithm checks composability is smaller in the case of $N_C = 50$.

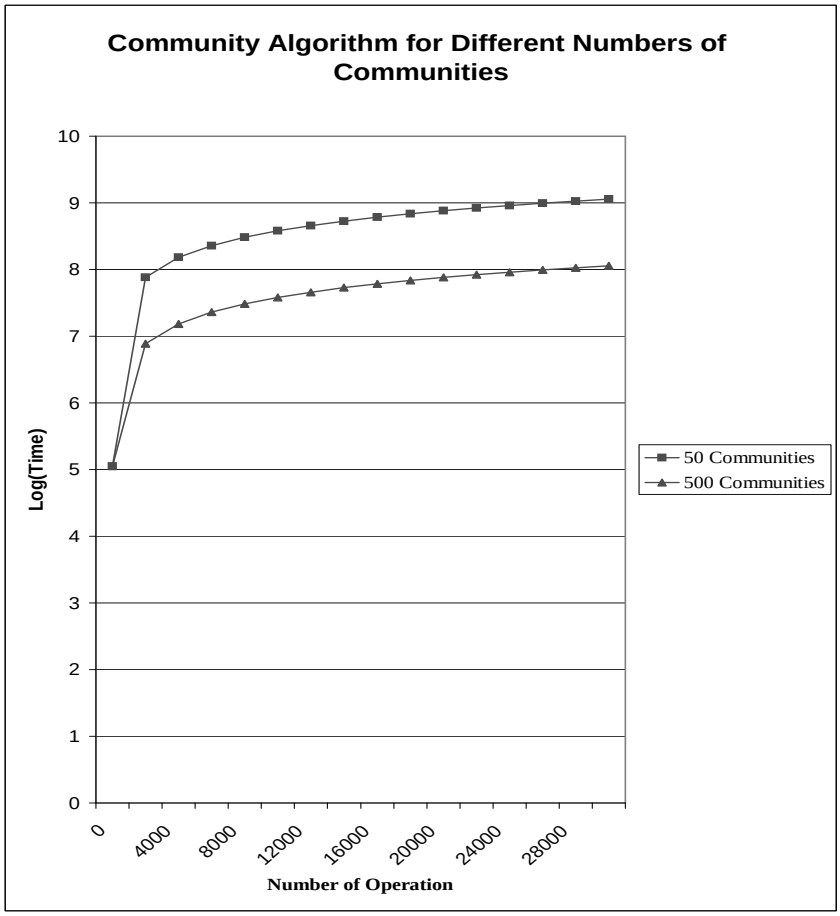

Figure 6.12: Community-centric Algorithm for Different Numbers of Communities

Message-Centric Algorithm

The minimum composition time for the message-centric algorithm is similar to the minimum time for the operation-centric and community-centric algorithms. In what follows, we compute the maximum composition time. The message-centric algorithm prunes the space of operations to be checked at two levels: community and automaton levels. We consider only operations that belong to a given community. Additionally, the first message composability comparison is done only with *In/Out* operations. Finally, composability checking is done only with operations that belong to a final state of the input automaton. Let us assume that the mode of an operation follows a uniform distribution. Assume also that operations are uniformly distributed over the final states of the automaton. The number of operations to be checked for composability equals $\frac{N_{opc}}{2\times A_F}$. Since $N_{opc} = \frac{N_{op}}{N_C}$, the number of operations equals $\frac{N_{op}}{2\times N_C\times A_F}$.

Let us now compute the number of comparisons for checking message composability. The first comparison is done through the input automaton. The comparison time is hence equal to P_M, that is the size of the message to be verified by the automaton for acceptance. The second comparison is done as in operation-centric algorithm. It is thereby equal to $\frac{P_M\times(P_M+1)}{2}$. The maximum time for checking static semantic composability is:

$$T_{SS}^{max} = \frac{N_{op}}{2 \times N_C \times A_F}(t_{SS} + P_M + \frac{P_M \times (P_M + 1)}{2})$$

The following formulas specify the total execution time for the message-centric algorithm:

$$T_{ST} = \frac{1}{2} \times t_{ST} \times C_{SR} \times (\frac{N_{op}}{N_C \times A_F} + P_{max})$$
$$T_{SS} = \frac{N_{op}}{2 \times N_C \times A_F} \times C_{SR} \times (t_{SS} + P_M \times (P_M + 1))$$

$$+\frac{1}{2} \times P_{max} \times C_{SR} \times (t_{SS} + 2 \times P_M)$$

$$T_{DS} = \frac{9 \times N_{op}}{2 \times N_C \times A_F} \times C_{SR} \times (B_{op}^2 \times TC \times I_R) + \frac{9}{2} \times P_{max} \times C_{SR} \times B_{op} \times TC$$

$$T = \frac{N_{op}}{2 \times N_C \times A_F} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + P_M \times (P_M + 1)$$
$$+9 \times B_{op}^2 \times TC \times I_R)$$
$$+\frac{1}{2} \times P_{max} \times C_{SR} \times (U_t + O_t + t_{ST} + t_{SS} + 2 \times P_M + 9 \times B_{op} \times TC)$$

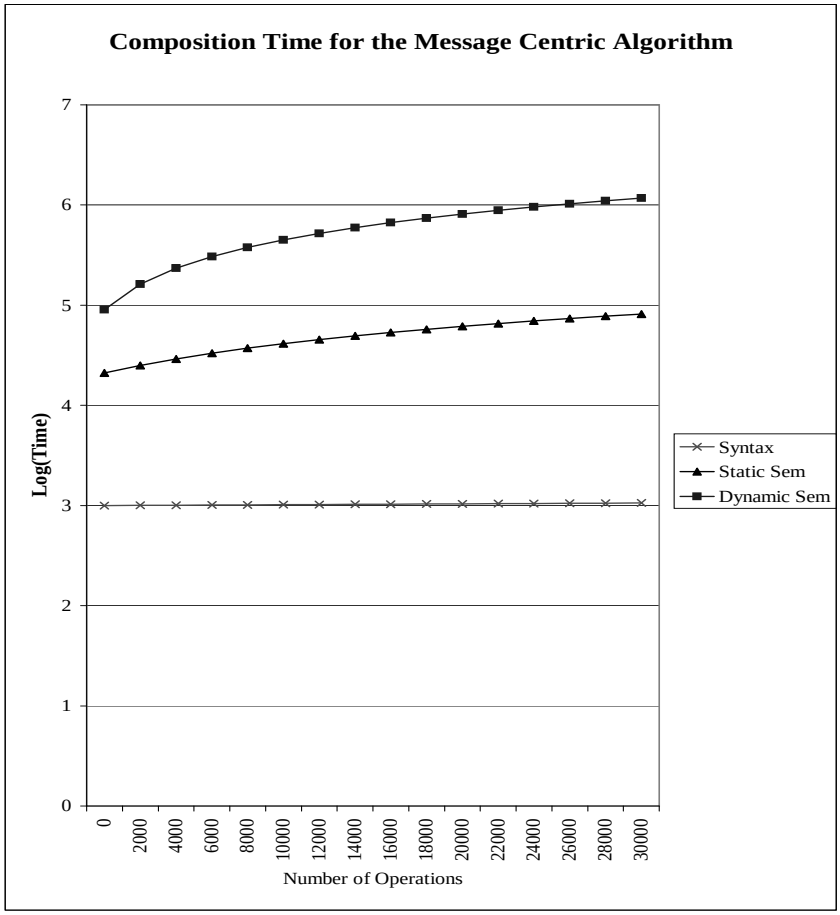

Figure 6.13: Composition Time for the Message Centric Algorithm

We compare in Figure 6.13 the times for syntactic (T_{ST}), static semantic (T_{SS}),

and dynamic semantic composition (T_{DS}). As for the operation-centric and community centric algorithm, the algorithm spends most of the time in checking static and dynamic semantic composability rules.

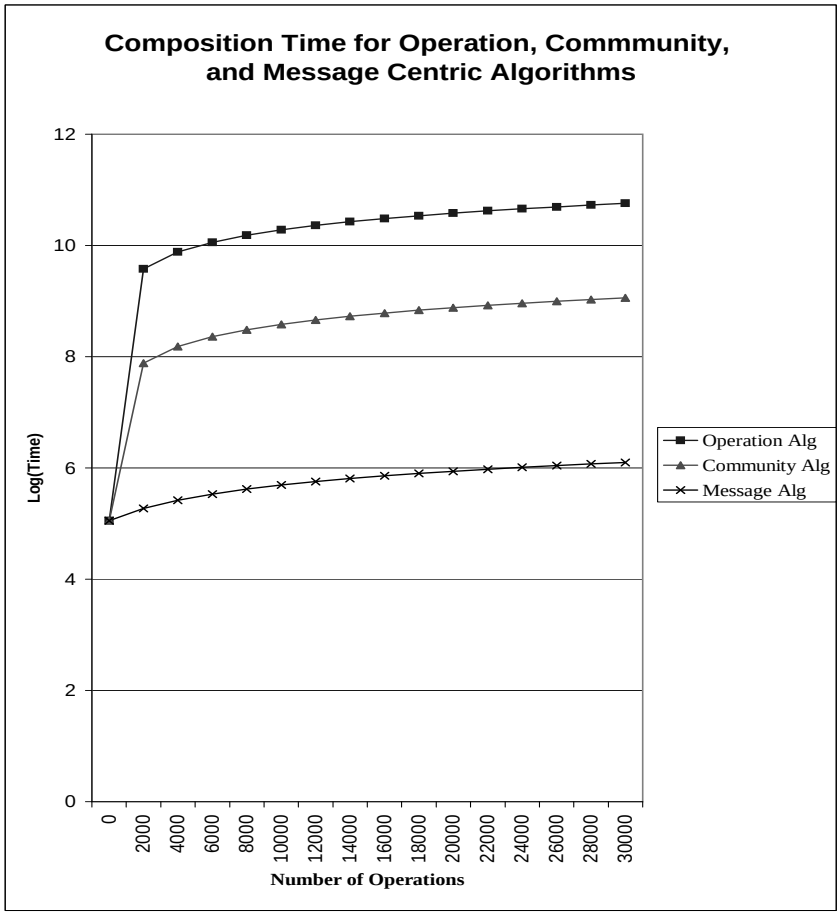

Figure 6.14: Composition Time for Operation, Community, and Message Centric Algorithms

Our main motivation behind designing different algorithms for vertical composition was to decrease the composition time. Figure 6.14 illustrates that the composition time for the operation-centric algorithm is longer than the community-centric algorithm's time. The latter time is longer than the time for composing services using the message-centric algorithm. Indeed, the first algorithm compares sub-requests

with all operations in the registry. The second algorithm focuses on a subset of operations that belong to the same community. The third algorithm reduces the size of the service space further by comparing sub-requests with the operations that belong to final states of the input automaton.

The design of the community-centric and message-centric algorithm was mainly driven by the following design approach: reduce the composition time dedicated to message composability. This is because significant time is spent on static semantic composability as mentioned previously. Figure 6.15 validates our approach. It shows that the message-centric algorithm substantially reduces the static semantic composition time. It also states that the community-centric algorithm outperforms the operation-centric algorithm in terms of static semantic composition time.

6.3.2 Experiments

The purpose of our experiment is to compare the execution times of the different composition algorithms. The simulation results are compared against the analytical results. We run our experiments on a *Sun Enterprise Ultra 10* server with a 440-MHz *UltraSPARC-IIi* processor, 1-GB of RAM, and under *Solaris* operating system. Although the algorithm is implemented in *WebDG* prototype, we use WSBF testbed to run the experiments. This allows the generation of a large number of services which would be difficult to achieve in the current *WebDG* version.

N_{op}	1000 - 15000
N_C	50 - 500
$Pmax$	20
C_{CR}	10
P_M	50 - 100
B_{op}	10 - 20
TC	10
A_F	1000 - 3000
Confidence level	0.98
Confidence accuracy	0.02

Table 6.3: Simulation settings

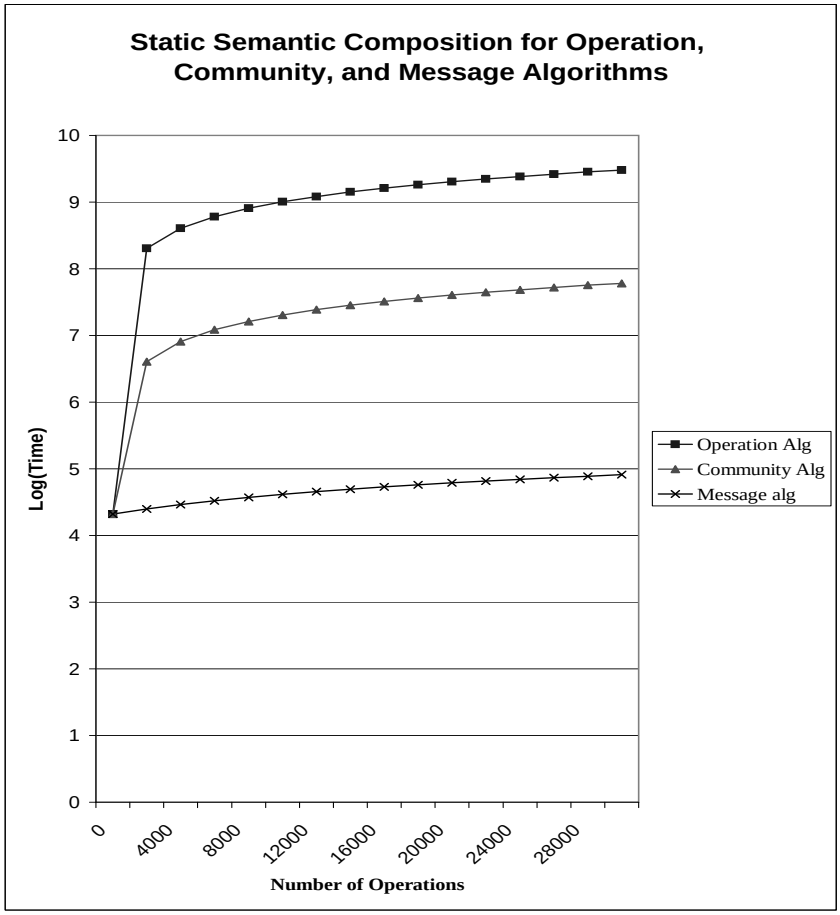

Figure 6.15: Static Semantic Composition for Operation, Community, and Message Centric Algorithms

Table 6.3 shows the common settings for all simulation experiments. The *Confidence level* and *confidence accuracy* shown in Table 6.3 are used to control the accuracy of the simulation results. Given N sample results Y_1, Y_2, ..., Y_N, the *confidence accuracy* is defined as H/Y, where H is the *confidence interval half-width* and Y is the sample mean of the results ($Y = (Y_1 + Y_2 + ... + Y_N)/N$). The *confidence level* is defined as the probability that the absolute value of the difference between the Y and μ (the true mean of the sample results) is equal to or less than H. H is defined by $H = t_{\alpha/2;N-1} \times \sigma/\sqrt{N}$ where σ^2 is the sample variance given by

$\sigma^2 = \Sigma_i(Y_i - Y)^2/(N-1)$ (thus σ is the standard deviation), and t is the standard t distribution. Users can specify the values of confidence level and accuracy before starting simulation. The simulation is not complete until the expected confidence level and accuracy are achieved.

Composition requests are simulated through a random request generator. The generation of requests follows the exponential distribution. During each simulation round, there are 50 requests generated. Each composition request contains 10 sub-requests. At the end of the round, the result is checked against the confidence level and confidence accuracy. The simulation continues if the confidence conditions are not satisfied.

We run several sets of experiments for the different algorithms. In the first set (Table 6.4), we compare the syntactic, static semantic, and dynamic semantic composition times for the operation-centric algorithm. We vary the number of operations from 1000 to 15000 with an iteration range of 1000. Figure 6.16 depicts the experimental results obtained for the operation-centric algorithm. As illustrated in the chart, the behavior of this algorithm is similar to the one obtained in the analytical study.

	Composability	N_{op}
Experiment 1	Syntactic	1000 - 15000
Experiment 2	Static Semantic	1000 - 15000
Experiment 3	Dynamic Semantic	1000 - 15000

Table 6.4: Experiments - Syntactic and Semantic Composability for the Operation Centric Algorithm

In the second set of experiments (Table 6.5), we compare the syntactic, static semantic, and dynamic semantic composition times for the community-centric algorithm. Figure 6.17 depicts the experimental results obtained for the community-centric algorithm. As illustrated in the chart, the behavior of this algorithm is similar to the one obtained in the analytical study.

In the third set of experiments (Table 6.6), we compare the syntactic, static se-

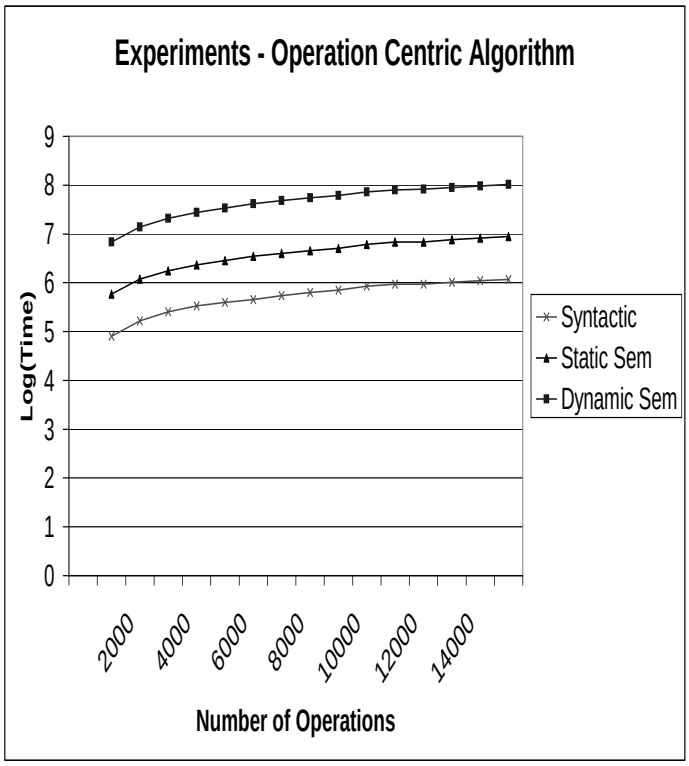

Figure 6.16: Experiments - Syntactic and Semantic Composability for the Operation Centric Algorithm

	Composability	N_{op}
Experiment 1	Syntactic	1000 - 15000
Experiment 2	Static Semantic	1000 - 15000
Experiment 3	Dynamic Semantic	1000 - 15000

Table 6.5: Experiments - Syntactic and Semantic Composability for the Community Centric Algorithm

mantic, and dynamic semantic composition times for the message-centric algorithm. Figure 6.18 depicts the experimental results obtained for the message-centric algorithm. As illustrated in the chart, the behavior of this algorithm is similar to the one obtained in the analytical study.

In the fourth set of experiments (Table 6.7), we assess the impact of the com-

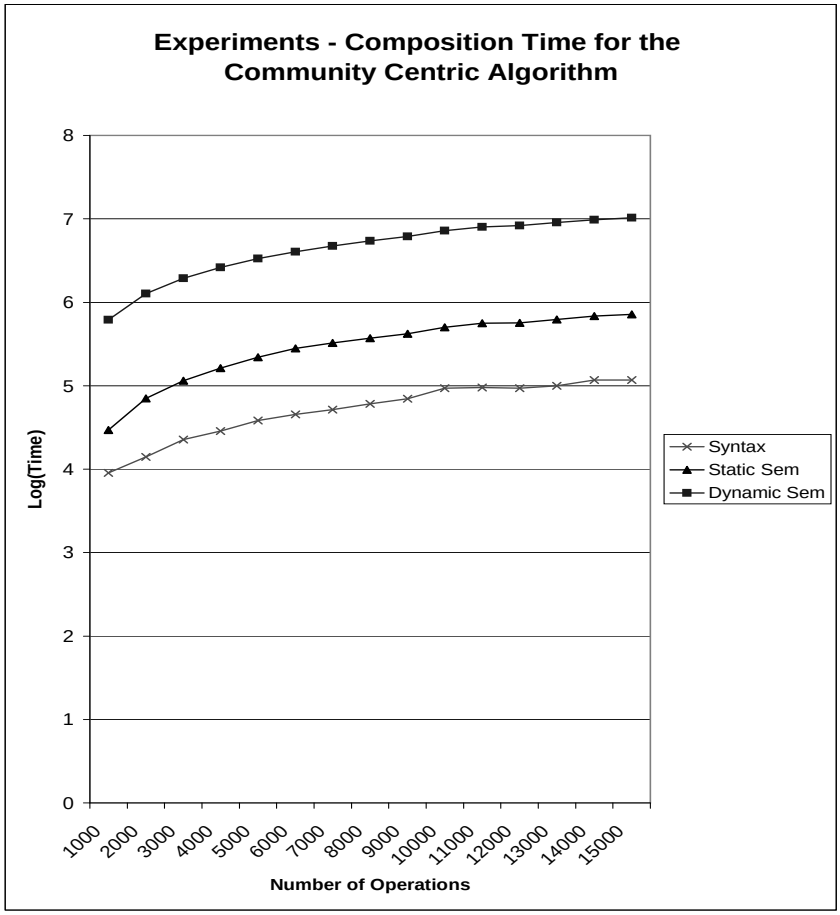

Figure 6.17: Experiments - Syntactic and Semantic Composability for the Community Centric Algorithm

	Composability	N_{op}
Experiment 1	Syntactic	1000 - 15000
Experiment 2	Static Semantic	1000 - 15000
Experiment 3	Dynamic Semantic	1000 - 15000

Table 6.6: Experiments - Syntactic and Semantic Composability for the Message Centric Algorithm

munity number N_C on the performance of the community-centric algorithm. We consider two cases where this number is equal to 10 and 50. Figure 6.19 depicts the

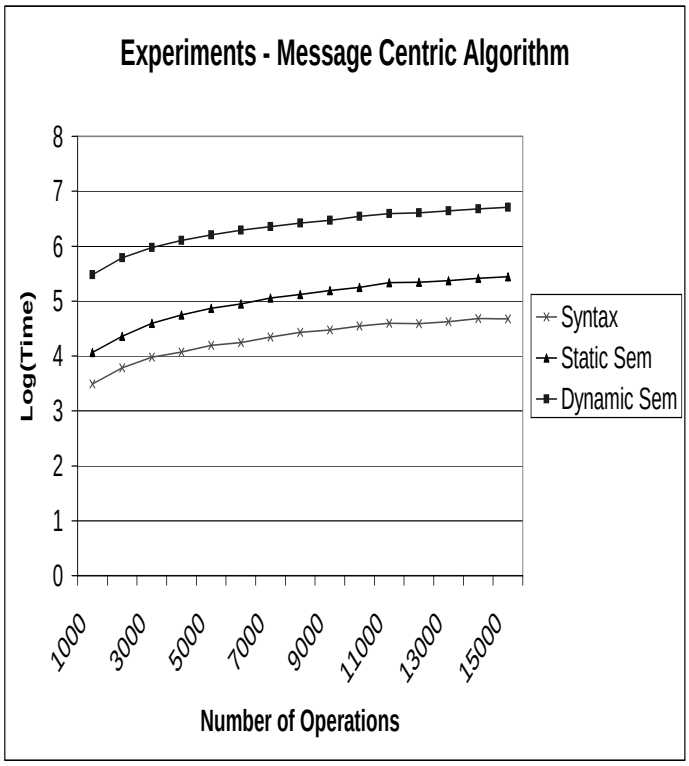

Figure 6.18: Experiments - Syntactic and Semantic Composability for the Message Centric Algorithm

experimental results obtained for the community-centric algorithm for two different numbers of communities. As illustrated in the chart, the behavior of this algorithm is similar to the one obtained in the analytical study.

	N_{op}	N_C
Experiment 1	1000 - 15000	10
Experiment 2	1000 - 15000	50

Table 6.7: Experiments - Community Centric Algorithm for Various Numbers of Communities

In the fifth set of experiments (Table 6.8), we compare the total composition time for the three algorithms: operation-centric, community-centric, and message centric algorithms. Figure 6.20 depicts the experimental results obtained for the operation,

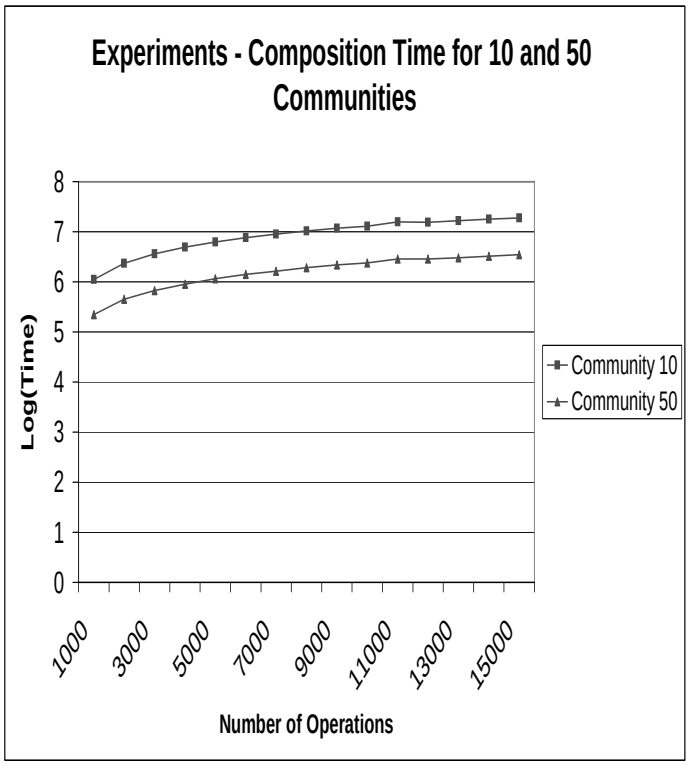

Figure 6.19: Experiments - Community Centric Algorithm for Various Numbers of Communities

community, and message centric algorithms in terms of their total composition time. As illustrated in the chart, the behavior of these algorithms is similar to the one obtained in the analytical study.

	Algorithm	N_{op}
Experiment 1	Operation-Centric	1000 - 15000
Experiment 2	Community-Centric	1000 - 15000
Experiment 3	Message-Centric	1000 - 15000

Table 6.8: Experiments - Total Composition Times for the Different Algorithms

In the last set of experiments (Table 6.9), we focus on the static semantic composition time. We compare this time for the three algorithms: operation-centric, community-centric, and message centric algorithms. Figure 6.21 depicts the exper-

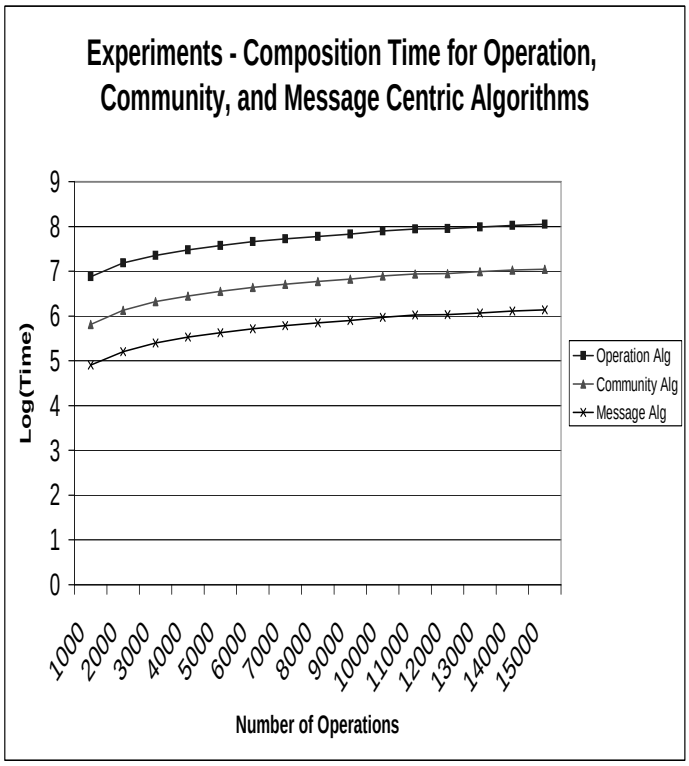

Figure 6.20: Experiments - Total Composition Times for the Different Algorithms

imental results obtained for the operation, community, and message centric algorithms in terms of their static composition time. As illustrated in the chart, the behavior of these algorithms is similar to the one obtained in the analytical study.

	Algorithm	N_{op}
Experiment 1	Operation-Centric	1000 - 15000
Experiment 2	Community-Centric	1000 - 15000
Experiment 3	Message-Centric	1000 - 15000

Table 6.9: Experiments - Static Semantic Composition Times for the Different Algorithms

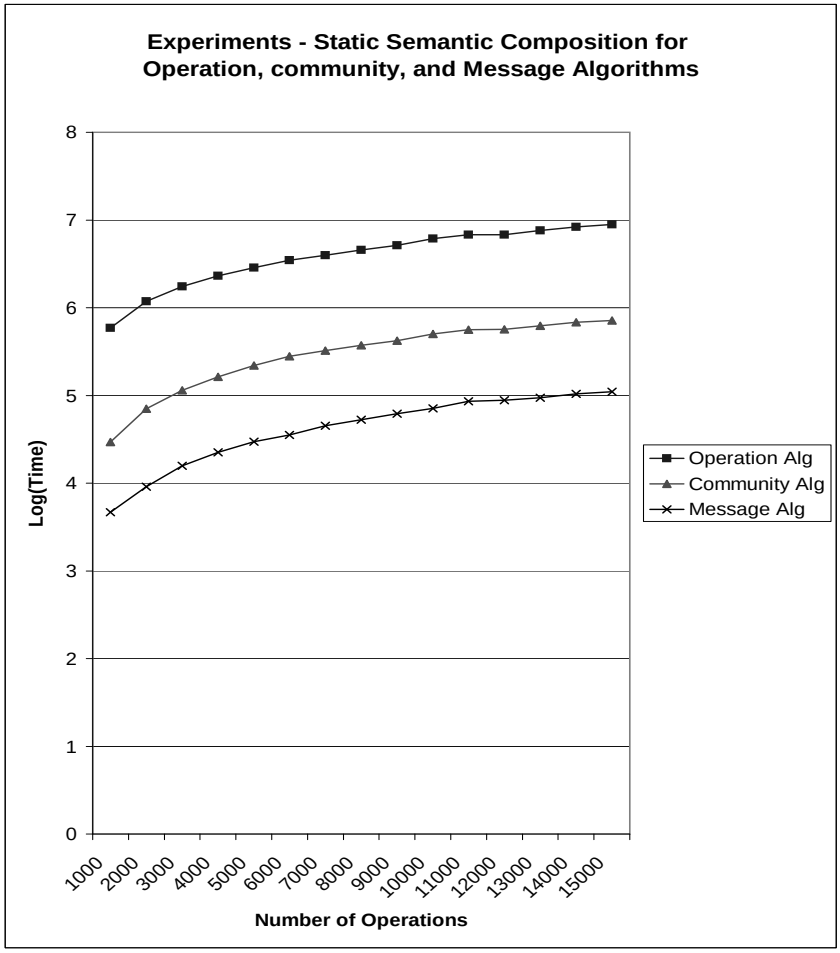

Figure 6.21: Experiments - Static Semantic Composition Times for the Different Algorithms

6.3.3 Summary

The results obtained in the analytical and experimental study are comparable. These can be summarized as follows. First, most of the composition time (for the three algorithms) is spent on checking static and dynamic composability. Indeed, B-composability (dynamic semantics) compares all business logic rule of each sub-request with the business logic rules of target operations. Similarly, static semantics compares all message parameters of each sub-request with message parameters of

target operations. This is in contrast with syntactic composability which performs three comparisons for each pair (sub-request,target): binding, mode, and number of parameters. Second, the number of communities has an impact on the composition time. The bigger is the number of communities, the smaller is the composition time. Third, the composition time for the operation-centric algorithm is longer than the community-centric algorithm's time. The latter time is longer than the time for composing services using the message-centric algorithm. Indeed, the first algorithm compares sub-requests with all operations in the registry. The second algorithm focuses on a subset of operations that belong to the same community. The third algorithm reduces the size of the service space further by comparing sub-requests with the operations that belong to final states of the input automaton. Fourth, the message-centric algorithm substantially reduces the static semantic composition time. Additionally, the community-centric algorithm outperforms the operation-centric algorithm in terms of static semantic composition time.

Chapter 7

Related Work

There is a large body of research in the field of Web service composition, the result of which appeared in the related literature. In this Chapter, we overview major techniques, prototypes, standards, and platforms for Web service composition that are most closely related to our research. In Section 7.1, we describe the major efforts for enabling the automatic composition of Web services. In Section 7.2, we overview Web service composition research prototypes. In Section 7.3, we discuss ongoing standardization efforts for Web service composition In Section 7.3, we describe current commercial platforms for Web service composition.

7.1 Automatic Composition of Web Services

In this section, we discuss related work dealing with the different issues for the automatic composition of Web services. These include the ontological support of semantic Web services, checking the composability of Web services, and generating composite service descriptions.

Ontological Support for Semantic Web Services – Significant research is being devoted to the standardization of service ontologies in *DAML-S* (DARPA Agent Markup Language for Web Services) [74]. *DAML-S* provides the ability to organize

Web services into ontologies. DAML-S divides service descriptions into the service profile, model, and grounding. The service *profile* provides a high level description of a Web service. It expresses required input of the service and the output the service will provide to the requester. The service *model* defines the operations and their execution flow in the Web service. Service *grounding* provides a mapping between DAML-S and the WSDL standard and describes how the service may actually be invoked. The service profile in DAML-S provides the means for Web service discovery. It is divided into a description of the service, its functionalities, and its functional attributes. The *description* provides human understandable information about the Web service. For example, a description includes the name of the service and its textual description. A *functionality* in DAML-S describes properties like input, output, precondition, effect, etc.

Several features distinguish our ontological approach from *DAML-S*. First, *DAML-S* proposes an ontology for Web services *not* for community of Web services. Our approach provides means for the semantic description of Web services and their ontological organization into communities. Second, the static semantics in *DAML-S* mostly focuses on describing operations' features. We define a broader view of static semantics by describing semantics both at the operation and message levels. Third, *DAML-S* gives little support for the dynamic semantics of Web services. It does not allow the specification of pre-operations and post-operation which are particularly important for enabling the automatic generation of business processes. Additionally, the notion of behavior and business logic is not explicitly defined. Fourth, *DAML-S* providers define their service operations from scratch. In our approach, providers inherit the functionalities of a community simply by registering their services with it. They may also personalize that community to best fit their capabilities.

Composability of Web Services – Several techniques have recently been proposed in the literature to deal with service matching and composability. [104] proposes a solution based on DAML-S for semantic matching between service advertisements

and capabilities. The matching technique is limited to comparing inputs and outputs of the advertisement with inputs and outputs of the request. *LARKS* defines five rules for service matchmaking: context matching, profile comparison, similarity matching, signature matching, and constraint matching [120]. Those rules mostly compare service text descriptions, signatures (inputs and outputs), and logical constraints about inputs and outputs. [72] describes the design of a service matchmaker that uses DAML-S based ontology. It uses techniques from knowledge representation to match service capabilities. In particular, it defines a Description Logic (DL) reasoner; advertisements and requests are represented in DL notations.

A composability rule that compares service categories is presented in [7]. However, features operation modes, message data types, static semantics, dynamic semantics, and soundness are not considered. [75] presents a model for checking composability. Unlike in our approach, this model is limited to checking syntactic features (input and output events of component services). In addition, it only checks composability on a *a posteriori* basis to replace a component service by another. [57] defines composability rules to compare service categories and messages. It does not seem to consider features related the static semantics, dynamic semantics, quality of operation, and composition soundness. *Process templates* and *reference processes* are defined in [25] and [113] respectively. However, these notions are different from the notion of stored templates. Indeed, process templates and reference processes are used as *a priori* "canvas" when defining composite services. In contrast, stored templates are used *a posteriori* to check *a posteriori* the soundness of composite services.

Generation of Composite Services – Automatic service composition has been the focus of several recent projects. *DAML-S* defines a semantic markup for Web services based on the use of ontologies [74] . *DAML-S* introduces the notions of prerequisites (called Preconditions) and consequences (called Effects) of Web services to cater for automatic composition. It is not clear, however, how composite services are generated using DAML-S specifications. DAML-S does not define the notion of service composability. Additionally, it does not consider semantic properties such

as purpose, parameter's unit and business role. An architecture for service composition in pervasive computing environments is presented in [29]. Service descriptions are provided in *DAML-S*. They also include platform specific information such as processor type, speed, and memory availability. The composition manager uses a semantic service discovery mechanism to select component services. This mechanism uses a *Jini* based semantic discovery framework. The matching mechanism mostly focuses on comparing service attributes. In contrast, the matchmaking algorithm proposed in this dissertation is based on a set of composability rules that compare the structure of messages, their business function, the semantics and data types of their parameters, qualitative properties, and the soundness of composite services.

WSMF (Web Service Modeling Framework) combines the concepts of Web services and ontologies to cater for semantic Web enabled services [21]. *WSMF* is still in its early stage. The techniques for the semantic description and composition of Web services are still ongoing. Furthermore, *WSMF* does not address the issue of service composability. An approach for ontology-based composition of Web services is proposed in [22]. It uses DAML-S for describing Web services. Semantic matching of Web service capabilities is limited to comparing QoS (Quality of Service) dimensions and input/output parameters of Web services. In our approach, we compare additional features such as category, purpose, parameters' data types, units, and business roles. Additionally, [22] mostly focuses on the automatic selection of Web services. Only few details are given about the way composite services are generated. *Ninja* [55] introduces a technique called Automatic Path creation (APC) to cater for automatic service composition. When an APC receives requests for composite service execution, the APC creates a path that includes a sequence of operators that perform computation on data and connectors that provide data transport between operators. *Ninja* mostly focuses on fault tolerance by replicating services on multiple workstations. It uses a limited operator functional classification (four categories) to automate the selection of operators. It is also mainly based on input-output matching of services. *SWORD* [107] uses a rule-base expert system to automatically determine whether a desired composite service can be achieved using

existing services. SWORD does not seem to focus on service composability and semantic description of Web services.

7.2 Research Prototypes

In this section, we overview research prototypes that support Web service composition. Note that the list of prototypes described in this section is not exhaustive. We focus on a representative set of such prototypes.

CMI (*Collaboration Management Infrastructure*) [112, 53] – CMI provides an architecture for inter-enterprise workflows. The main components of CMI engine includes the *CORE*, *coordination* and the *awareness engines*. The *CORE engine* provides basic primitives used by the coordination and awareness engines. These primitives include constructs for defining resources, roles, and generic state machines. CMI's *coordination model* extends the traditional workflow coordination primitives with advanced primitives such as *placeholder*. The concept of placeholder enables the dynamic establishment of trading relationships. A placeholder activity is replaced at runtime with a concrete activity having the same input and output as those defined as part of the placeholder. A selection policy is specified to indicate the activity that should be executed. If multiple providers offer implementations for an activity interface, the selection policy may use a broker to choose the implementation that offers the "best" quality of service. CMI's *awareness model* captures information that is closely related to a specific role and situation of a process participant. Awareness information is specified by process/awareness designers using *awareness specifications*.

eFlow [26] – *eFlow* is a platform that supports the specification, enactment, and management of composite services. A composite service is described as a process schema that combines basic or composite services. A composite service is modeled by a graph that defines the order of execution among the nodes in the process. It may include *service*, *decision*, and *event* nodes. Service nodes represent the

invocation of a basic or composite service. The definition of a service node contains a *search recipe* represented in a query language. When a service node is invoked, a search recipe is executed to select a reference to a specific service. Decision nodes specify the alternatives and rules controlling the execution flow. Event nodes enable service processes to send and receive several types of events. A *service process instance* is an enactment of a process schema. To support heterogeneity of services, eFlow provides adapters for services that support various B2B interaction protocols such as OBI and RosettaNet.

WISE (*Workflow based Internet SErvices*) [111, 70] – WISE aims at providing an infrastructure for the support of cross-organizational business processes in *virtual enterprises*. WISE architecture is organized into four components: process *definition*, *enactment*, *monitoring* and *coordination*. The *process definition* component allows *Virtual Business Process* (VBPs) to be defined using as building blocks the entries of a catalog where companies within a trading community (TC) can post their services. The *process enactment* component compiles the description of the VBP into a representation suitable for enactment and controls the execution of the process by invoking the corresponding services of the TC. The *process monitoring* component keeps track of the progress made in the execution of the VBP. The information produced by this tool is used to create an awareness model used for load balancing, routing, quality of service, and analysis purposes. The *process coordination* component supports multimedia conferencing and cooperative browsing of relevant information between all participants in the TC.

CrossFlow [73] – The main contribution of CrossFlow is in using the concept of *contracts* as a basic tool for cooperation. Businesses specify their interactions through *contracts* (e.g., purchase and employment contracts). When a provider wants to advertise a service, it uses its contract manager to send a contract template to a *trader* or *matchmaking engine*. When a consumer wants to outsource a service, it uses a contract template to search for relevant providers via the trader.

If a matching is found between consumer's requirements and provider's offer, an electronic contract is made by filling in the template. Based on the specifications in the contract, a dynamic contract and service enactment infrastructure are set up. The symmetrical infrastructure in provider's and consumer's sides contains proxy gateways that control their interactions. The dynamically created modules can be removed after contract completion.

Mentor-Lite [134] – Mentor-Lite addresses the problem of distributing the execution of workflows. The idea is to partition the overall workflow specification into several sub-workflows, each encompassing all the activities that are to be executed by a given entity within an organization. The basic building block of Mentor-Lite is an interpreter for workflow based on state charts. Two other modules are integrated with the workflow interpreter defining the workflow engine: *communication manager* and *log manager*. The *communication manager* is responsible for sending and receiving synchronization messages between the engines. It uses the *Transaction Processing* (TP) monitor *Tuxedo* for delivering synchronization messages within queued transactions. The *log manager* provides logging and recovery facilities. A separate workflow log is used at each site where a workflow engine is running.

XL (*XML Language*) [49, 50] – XL defines an XML language for the specification of Web services. An XL service specification contains *local declarations*, *declarative clauses*, and *operation specifications*. Two kinds of local variables can be declared in XL. The first kind of variable represents the internal state of the service. The second kind of variable represents the internal state of a particular conversation in which the service is involved (e.g., session ID). Declarative clauses include variables that control the Web service global state. In particular, the *history* and *on change* clauses address the issue of external manageability and adaptability. If the *history* clause is specified, all operation invocations are automatically logged. The *on change* clause uses triggers to detect changes in variables declared in the Web services' local declarations.

SELF-SERV (*compoSing wEb accessibLe inFormation and buSiness sERvices*) [10, 114] – SELF-SERV proposes a process-based language for composing Web services based on *state charts*. It also defines a *peer-to-peer* Web service execution model in which the responsibility of coordinating the execution of a composite service is distributed across several peer components called *coordinators*. The coordinator is a lightweight scheduler which determines when a state within a state chart should be entered and what should be done when the state is entered. It also determines when should a state be exited and what should be done after the state is exited. The knowledge needed by a coordinator to answer these questions at runtime is statically extracted from the state chart describing the composite service operations and represented in the form of routing tables.

In Table 7.2, the aforementioned prototypes are compared using key interaction layers presented in Chapter 2. For example, eFlow uses RMI at the communication layer. At the content layer, eFlow provides adapters to support different interaction protocols such as OBI and RosettaNet. Interoperability at the business layer is enabled through a process description model based on state machines.

The same prototypes are compared in Table 7.2 using key interaction dimensions defined in Chapter 2. For example, eFlow allows loose coupling among B2B participants. In terms of autonomy, trading partners do not need to reveal how their services are implemented. Heterogeneous interaction protocols are supported through adapters. External manageability and adaptability are possible via event tracking and process templates respectively. Security, however is not addressed. Scalability is accommodated using distributed service enactment engines.

	Communication Layer	*Content Layer*	*Business Process Layer*
CMI	Transport protocols (e.g., HTTP, CORBA) must *a priori* be agreed upon	Message format (e.g., XML, EDI) must *a priori* be agreed upon	State machine based model for process description
WISE	Coordination and communication module	Not Addressed	Virtual business processes
Cross Flow	Java RMI	Contract in XML	Contracts
Mentor-Lite	Transaction Processing (TP) monitor (Tuxedo)	Not Addressed	Business processes expressed as state and activity charts
eFlow	Java RMI	Provides adapters to support different protocols such as OBI and RosettaNet	State machine based model for process description
XL	SOAP	XML Schema	Little or no statements for inter-service business processes
SELF-SERV	SOAP	Not Addressed	State charts

Table 7.1: Prototypes vs. Interaction Layers

7.3 Standardization Efforts

Efforts are underway to define standards for composing Web services [3]. These include WSFL (*Web Services Flow Language*) [62], XLANG [87], and BPEL4WS (*Business Process Execution Language for Web Services*) [9], and *BPEL4WS* [9].

WSFL [62] – WSFL introduces the notions of *flow* and *global model* for defining composite services. The *flow model* specifies the execution sequence between component services. It is represented by a directed graph. Each node of the graph, called *activity*, models a single step of the overall business goal to be achieved through composition. Activities are bound to services through a *locator* element. This binding can be either *static* or *dynamic*. In a *static* binding, the service is directly specified in the locator. In a *dynamic* binding, the locator may, for example, contain a UDDI query that returns a list of candidate services; a service is then selected through a given *selection policy* (e.g., the first service in the list). Two types of edges are used to connect activities: *control links* and *data links*. *Control links* prescribe the order in which activities have to be performed. *Data*

	Coupling	*Autonomy*	*Heterogeneity*	*Ext. Manag.*	*Adaptability*	*Security*	*Scalability*
CMI	Tight and long term	External systems only need to reveal the state they are in after they accomplish a task, not how they accomplish the task	Use of object-oriented proxies	State dependent control flow and use of awareness events	Primitives such as *optional* and *inhibitor* can be used for coping with some unforeseen events	Role	Distributed and parallel engines for execution
WISE	Tight and long term	Partners must advertise services in encapsulated objects	Object-based middleware	Process monitoring and analysis module	Execution guarantee	Not Addressed	Distributed architecture
Cross Flow	Loose and transient	Partners must agree on service contract definition	Partners must install service contract run time environment	Quality of Service (QoS) module provides monitoring facilities	Primitives for flexible execution are restricted to those provided by traditional workflows	Not Addressed	Cost of entry: participants must locally install contract run time environment
Mentor-Lite	Tight and long term	Participants do not need to reveal how services are implemented	Application programs are connected to the workflow engine by specific wrappers	Not Addressed	Not Addressed	Not Addressed	Workflows are partitioned into several sub-workflows and distributed
eFlow	Loose and long term	External systems need to describe their services not their implementation	Provides adapters for different protocols and platforms such as OBI, RosettaNet, and e-speak	Event tracking	Provides process templates, service nodes, and service data repositories for reuse	Not Addressed	Distributed service enactment engines
XL	Loose and transient	Participants do not need to reveal how services are implemented	Web services can be written in XL, Java, or other languages	History clauses	Change clauses	Security features of J2EE	Not Addressed
SELF-SERV	Loose and transient	Participants do not need to reveal how services are implemented	Service wrappers	Not Addressed	Not Addressed	Not Addressed	Peer-to-peer execution model

Table 7.2: Prototypes vs. Interaction Dimensions

links represent the flow of information between activities. The *global model* specifies how component services interact. It includes a set of *plug link* elements. A *plug link* connects an operation of the composite service (called *exported* operation) to an operation of a component service. This indicates that the corresponding interaction must take place to completely implement an activity.

XLANG [87] – XLANG provides language constructs for describing behavioral aspects of Web services and combining those services to build multi-party business processes. At the intra-service level, XLANG extends WSDL language by adding a *behavior* element. A *behavior* defines the list of actions that belong to the service and the order in which these actions must be performed. XLANG defines two types of actions: regular WSDL operations and XLANG-specific actions (e.g., timeout operations). At the inter-service level, XLANG defines a *contract* element which provides means for interconnecting several XLANG service descriptions. The execution order of XLANG actions is defined through *control processes* (e.g., *sequence*, *while*). A particular control process named *context* enables the support of *transactions*. The concept of transaction, as used in databases, guarantees that in case of failure, the partial updates of a service execution are rolled back. XLANG adopts a looser notion of transaction based on compensation. The execution of actions in a *context* may fail or be cancelled for a variety of business and technical reasons (e.g., communication failure). In this case a compensation code, explicitly specified by the provider, is executed.

BPEL4WS [9] – BPEL4WS combines the features of both WSFL (support for graph oriented processes) and XLANG (structural constructs for processes) for defining business processes. A business process is composed of several steps called *activities*. BPEL4WS defines a collection of primitive activities such as *invoke* to invoke a Web service operation. These primitive activities can combined into more complex primitives using any of the structure activities provided in BPEL4WS. These include the ability to (1) define an ordered sequence of steps (*sequence*), (2)

have branching using the now common "case-statement" approach (*switch*), (3) define a loop (*while*), (4) execute one of several alternative paths (*pick*), and (5) indicate that a collection of steps should be executed in parallel (*flow*). BPEL4WS provides mechanisms to handle and recover from errors in business processes (*throw* and *catch* constructs). It also adopts the notion of *compensating* actions defined in XLANG. Fault handling and compensating are supported by introducing the notion of a *scope*. A *scope* is the unit of fault handling and/or compensation.

ebXML (*Electronic Business XML*) [96] – *ebXML* aims at defining a set of specifications for enabling B2B interactions among companies of any size. The basic part of the ebXML infrastructure is the *repository*. It stores important information about businesses along with the products and services they offer. At the communication layer, businesses exchange messages through the *messaging service*. One important feature of the ebXML messaging service is that it does not rely on a specific transport protocol. It allows for the use of any common protocol such as SMTP, HTTP, and FTP.

At the content layer, companies interact through *business documents*. A *business document* is a set of information components that are interchanged as part of a business process. Business documents are composed of three types of components: *core components*, *domain components*, and *business information objects*. *Core components*, stored in the *core library*, are information components that are re-usable across industries. *Domain components* and *business information objects* are larger components stored in the *domain library* and *business library* respectively. Core components are provided by the ebXML library while domain component and business information objects are provided by specific industries or businesses.

At the business process layer, ebXML defines a *business process specification schema* available in UML and XML versions. The UML version only defines a UML class diagram. It is not intended for the direct creation of a business process specification but provides a representation of all the elements and relationships required for its creation. The XML version allows the creation of XML documents representing

ebXML-compliant *business process specifications*. ebXML provides a set of common business process specifications that are shared by multiple industries. These specifications, stored in the *business library*, can be used by companies to build customized business processes. Interactions between business processes are represented through *choreographies*. A *choreography* specifies the ordering and transitions between business transactions. To model collaboration in which companies can engage, ebXML defines *collaboration protocol agreements* (CPAs). A CPA is an agreement by two trading partners which specifies in advance the conditions under which the trading partners will collaborate (e.g., terms of shipment and terms of payment).

The ebXML infrastructure enables secure and reliable communications by using emerging security standards (e.g., SSL and S-HTTP). In addition, digital signatures can be applied to individual messages or a group of related messages to guarantee authenticity. With regard to autonomy and adaptability, ebXML appears to offer the same kind of properties as eCO. External manageability can be provided by adding specific activities in shared business processes. The initial goal of the ebXML initiative was to support a fully distributed set of repositories which is an interesting feature for improving scalability. However, to date, only a single repository is specified.

7.4 Deployment Platforms

Major software vendors (*IBM*, *Microsoft*, *Sun Microsystems*, *HP*, *Oracle*, *BEA systems*, etc) are currently working on implementing Web service platforms. The purpose of this section is not to compare commercial products but to overview their main features. Because there are a large number of products, this section does not attempt to cover all of them. Instead, we focus on the major players in this arena. Our coverage is based on user manuals and white papers since there are few or no published technical papers detailing commercial products. Additionally, existing products are at various development stages and operate at different levels of disclosure.

Microsoft .NET [86] – *.NET* embraces the concept of Web services to enable B2B interaction. It consists of three key elements: *.NET Framework and tools*, *.NET Enterprise Servers*, and *.NET Service Building Blocks*. *.NET Framework and tools* provides the standard-based tools for SOAP, WSDL, and UDDI. *.NET Enterprise Servers* provides the core components for building Web services. These include database like *SQL Server 2000*, messaging software like *Exchange 2000 Server*, business process technology like *BizTalk Server 2000*, and *Internet Security and Acceleration Server*. *.NET Service Building Blocks* contains pre-defined Web services created using the *.NET* infrastructure (e.g., *Passport* and *HailStorm*).

SOAP is used as the main transport protocol in the communication layer. Interoperability at the communication layer is also supported by *Microsoft Message Queue* (MSMQ) supplemented with gateways for sending and receiving documents in various formats from trading partners. *Microsoft Host Integration Server* is used to support connection to proprietary systems like IBM mainframes. Heterogeneity at the content layer is addressed by adhering to open standards (XML and WSDL) and the wrapping of applications as *.NET Managed Components*. Building business processes (called *Orchestration*) is done through *BizTalk Server*. Developers use the *Biztalk Orchestration Designer* to create *Biztalk processes*. These are compiled into *XLANG schedules* which are executed by the *Biztalk Scheduler Engine*.

WebSphere [63] – *WebSphere* is a family of IBM products for B2B interactions. The *application server* is the cornerstone of *WebSphere*. It aims at providing database and backend integration as well as security and performance capability (e.g. workload management). The *WebSphere application server Advanced Edition* adds support for J2EE specification. It also extends J2EE with direct access to advanced CORBA services for greater flexibility and improved interoperability. The advanced edition integrates support for key Web service standards such as SOAP, UDDI, and WSDL. Additionally, it provides distributed transaction support for major database systems including *IBM*'s *DB2*, *Oracle*, *Sybase*, and *Informix*. Other

products make up the *WebSphere* platform. These include *WebSphere Business Components*, *WebSphere Commerce*, and *WebSphere MQ Family*. The *WebSphere Business Components* provides pre–built, tested, and "plug and play" components for building new applications or extending existing ones. *WebSphere Commerce* provides mechanisms for building B2B sites including catalog creation and payment processing. *WebSphere MQ Family*, formerly known as as *MQSeries*, is a family of message-oriented middleware products that enable communication between applications running on different hardware platforms.

Sun ONE (*Sun Open Net Environment*) [89] – *Sun ONE* is a platform for Web services developed by *Sun*. Two main product lines make up the *Sun ONE* platform: *Forte tools* and *iPlanet*. *Forte tools* offer *Integrated Development Environment* (IDE) for the Java, C, C++, and Fortran languages. It enables developers to access the plug-ins they need and hence speed the development of Web services. *iPlanet* is the core of *Sun ONE* platform. It includes a stack of products that allow the creation, deployment, and execution of Web services. Examples of such products are the *iPlanet Portal Server*, *iPlanet Application Server*, and *iPlanet Integration Server*. The *iPlanet Portal Server* is the representation layer of *iPlanet*. It delivers services to end-users by aggregating content and providing security, personalization, and knowledge management. The *iPlanet Application Server* enables access to legacy applications and databases. It also provides a J2EE execution environment for Web services. The *iPlanet Integration Server* is a workflow-based engine that enables businesses to define workflows across legacy applications and create services.

Sun ONE uses workflows to ensure interoperability at the business process layer. However, it is not clear how services are composed using the *iPlanet Application Server*. *Sun ONE* supports the emerging Web service standards such as SOAP, WSDL, and UDDI. The *iPlanet Portal Server* enables the integration of any HTML or XML encoded content and heterogeneous applications that run on major operating systems such as *Microsoft Windows*, and *Unix*. Complementary packages provide additional functionality including secure communications. *iPlanet*

addresses scalability by offering built-in services such as load balancing.

Vitria BusinessWare [128] – *Vitria BusinessWare* emphasizes on business process management and automation. It adopts *UML* and *WfMC reference model* for modeling business processes. The exchange of information between trading partners is done using XML. However, *BusinessWare* assumes that those partners will agree upon a common standard XML DTD to describe the documents to be exchanged. *BusinessWare* also requires businesses to agree on the semantics of business processes' activities. *BusinessWare*'s processes are divided into two types: *public* and *private*. A placement of purchase order described in RosettaNet PIPs is an example of public process. The way that different companies deal with an incoming order from a customer is an example of private process. The separation between private and public process allows trading partners to change their private process without affecting the cross-organization public business process.

BusinessWare is composed of four modules: *Business process management*, *Business-to-Business communications*, *Enterprise application integration*, and *Real-time analysis*. The *Business process management* controls and coordinates the flow of information between internal and external process systems. Both private and public processes can be defined using the graphical modeling tool. The *Business-to-Business communications* is responsible for interactions with trading partners using multiple protocols (HTTP-S, FTP, IIOP, SOAP, EDI, fax and email) and data formats (XML, IDL, EDI, RosettaNet). The *Enterprise application integration* provides connectors for major databases, messaging systems, and packaged applications. The *Real-time analysis* enables the gathering and analysis of process information. It allows businesses to identify processing bottlenecks and react to fast-changing business conditions.

Oracle Integration Server [98] – *Oracle Integration Server* is one of the products of *Oracle Application Server* which is based on J2EE and emerging Web service standards. It supports transport protocols such as SOAP, HTTP-S, SMTP, FTP/S,

IIOP, and various messaging systems (JMS, IBM MQSeries, TIBCO/Rendezvous). The *Integrator* has two main components. The first component provides an EJB container for executing the designed business process. The second component consists of the design and management tools which include *Integration Modeler*, *Business Process Monitor*, and *Business Intelligence*. The *Integration Modeler* offers a set of Web-based tools to model business process, map data sources from one form to another, and set up relationships with trading partners. The *Business Process Monitor* provides means for users to monitor, analyze, and drill down on the state of the business process (such as start, stop, resume). The *Business Process Intelligence* uses *Oracle* data warehousing facilities to analyze and gather information about the overall flow of business processes (i.e., the frequency of messages being sent/received).

HP NetAction [60] – The *HP NetAction* software suite includes the *HP NetAction Internet Operating Environment* (IOE), a platform for building B2B applications. The IOE includes the *HP Process Manager* and *HP Web Services Platform*. *HP Process Manager* (formerly called *ChangeEngine*) allows the graphical definition of business processes and provides an environment that automates the execution of those processes. It has a component-based architecture based on J2EE. *HP Process Manager* also provides an audit logger that can be used to read information in XML format from a JMS (*Java Message Service*) queue. It allows the definition of audit nodes within a business process to indicate the points in the process at which audit information should be collected. *HP Web Services Platform* is a standards-based architecture for developing Web services. Key components of the *HP Web Services Platform* include *HP-SOAP 2.0*, *HP Service Composer* (a graphical tool for creating and mapping WSDL interfaces), *HP Registry Composer* (a graphical tool for registering and discovering Web services in UDDI registries). HP announced in July 2002 it was discontinuing its development and support of *NetAction*.

BEA WebLogic Integrator [8] – *BEA WebLogic Integrator* is the cornerstone of

BEA WebLogic E-Business Platform. It is built on top of a J2EE compliant application server and J2EE connector architecture. It supports current Web service standards such as SOAP, UDDI, and WSDL. The *Integrator* is composed of four major modules: *Application server*, *Application integration*, *Business process management*, and *B2B integration*. The *Application server* provides the infrastructure and functionalities for developing and deploying multi-tiers distributed applications as EJB components. The *Application integration* leverages the J2EE connector architecture to simplify integration with existing enterprise applications such as SAP R/3 and PeopleSoft. The *Business process management* provides a design tool and execution engine for business processes. The *B2B integration* manages interactions with external business processes. A separate module called *B2B integration/collaboration* is used to manage different B2B protocols (such as RosettaNet PIPs, BEA's eXtensible Open Collaboration Protocol) and *Quality of Service* (QoS) of the trading partners.

WebMethods [132] – *WebMethods* is composed of three modules: *WebMethods Enterprise Server*, *WebMethods Enterprise Adaptor* and *WebMethods Enterprise Rule Agent*. The *Rule Agent* is used to set up specific business rules that are required for integrating business processes across different enterprises. The *adaptors* connect information sources to *WebMethods Enterprise Server* and provide bi-directional mapping of information between the native format and the server's. Several adaptors are provided to allow the mapping of XML messages to industry-adopted message types (RossettaNet, cXML, OBI, EDI). The hub of the system is *WebMethods Enterprise Server* which acts as the central control and storage point. It uses XML for exchanging messages between trading partners. The server supports multiple transport protocols such as SOAP, HTTP, HTTP-S, RMI-IIOP, SMTP and FTP. It also defines a process-oriented language called *Flow* to visually compose services.

TIBCO ActiveEnterprise [123] – *ActiveEnterprise* uses a set of products to enable

B2B interactions. *TIBCO InConcert* is a tool for defining and managing dynamic workflows. *TIBCO IntegrationManager* defines and manages automated business processes that span multiple applications and transactions. *TIBCO MessageBroker* performs rule-based transformation and mapping of messages between different messaging softwares. *TIBCO Hawk* is a sophisticated tool for administrating and monitoring of system behaviors within *ActiveEnterprise*. *TIBCO Rendezvous* is an advanced messaging system that supports publish/subscribe, request/reply, synchronous/asynchronous, certified and transactional messaging paradigms. *ActiveEnterprise* supports other messaging protocols such as JMS, HTTP/S, COM, CORBA and MQSeries. At the content layer, *ActiveEnterprise* supports various vertical and horizontal industry standards such as cXML, RosettaNet, EDI, and HealthCare standards.

In Table 7.3, commercial Web service platforms are summarized using the following set of parameters: major modules, communication standards, content and business process standards, and key technologies. For example, BEA Weblogic Integrator includes an application server, application integration, business process management, and B2B Integration. The communication standards supported in BEA Weblogic Integrator are SOAP, JMS and IIOP. BEA Weblogic Integrator supports WSDL, XML, RosettaNet-PIP, and BEA-XOCP as content and business process standards. The key technologies that are supported include components (J2EE), XML, workflows, and Web services. Note that all deployment platforms support HTTP as a communication protocol. Additionally, the list of supported standards (communication, content, and B2B protocol) is non-exhaustive as new standards are constantly being added.

	Major Modules	*Communication Standards*	*Content and Business Process Standards*	*Key Technologies*
IBM WebSphere	Application Server, MQSeries, Business Components, WebSphere Commerce	MQSeries, JMS, IIOP, SOAP, HTTP	WSDL, XML, RosettaNet-PIP, cXML, EDI	Components (J2EE), XML, Web Services
Sun ONE	Forte tools and iPlanet	JMS, SOAP, LDAP, WAP, IIOP, HTTP	EDI, XML, WSDL	Components (J2EE), XML, Web services and Workflow
Oracle Integration Server	Integration Modeler, System Monitoring and Administration, Business Process Monitor, Business Intelligence	Oracle Queue, JMS, SOAP, IIOP, MQSeries, TIBCO/rendezvous, HTTP	XML, WSDL, EDI, RosettaNet-PIP, ebXML	Components (J2EE), workflow, XML, data mining, Web services
HP NetAction	HP Opencall, HP Chat, HP NetAction Internet Operating Environment	SOAP, JMS, IIOP, HTTP	XML, WSDL	Components (J2EE), XML, workflow (ChangeEngine), Web services
Microsoft .NET	.NET Framework and Tools, .NET Enterprise Servers, .NET Service Building Blocks	MSMQ, SOAP, Microsoft Host Integration Server, HTTP	XML, WSDL, RosettaNet-PIP, XLANG from BizTalk Server	DCOM, MSMQ, Web services, XML, BizTalk Orchestration Engine
BEA WebLogic Integrator	Application Server, Application Integration, Business Process Management, B2B Integration	SOAP, JMS, IIOP, HTTP	WSDL, XML, RosettaNet-PIP, BEA-XOCP	Components (J2EE), XML, workflow, Web services
WebMethods	Enterprise Server, Enterprise Adaptor, and Enterprise Rule Agent	SOAP, IIOP, JMS, HTTP	WSDL, XML, EDI, RosettaNet-PIP, ebXML, cXML, OBI	Components, workflow, Web services and Agents
Vitria Business Ware	Business Process Management, B2B Communications, Enterprise Application Integration and Real-Time Analysis	SOAP, IIOP, JMS, HTTP	XML, EDI, RosettaNet-PIP, ebXML, xCBL, cXML	Components, XML, workflow, process model, process analysis
TIBCO Active Enterprise	InConcert, IntegrationManager, MessageBroker, Hawk, and Rendezvous	SOAP, JMS, IIOP, MQSeries, HTTP	WSDL, XML, HL7, EDI, RosettaNet-PIP, BizTalk, ebXML, cXML, xCBL	Messaging software, XML, workflow, Web services

Table 7.3: Deployment Platforms

Chapter 8

Conclusions

In this chapter, we summarize the results of our dissertation and discuss future research directions for Web service composition.

8.1 Summary

Web service composition is emerging as *the* technology of choice for enabling inter-enterprise interactions on the Web [5, 77]. However, current techniques for service composition are generally ad-hoc, time consuming, and error prone. In this dissertation, we proposed an approach for the automatic composition of Web services. Based on "abstract" specifications of composition requests, the composition engine automatically generates detailed descriptions of composite services that fulfill those requests. This requires dealing with three major research thrusts: (i) describing the semantics of Web services, (ii) selecting "relevant" participants and checking their composability, and (iii) generating composite service descriptions. We also implemented our approach in *WebDG*, a prototype for accessing e-government Web services. We summarize below our major contributions in this dissertation.

Ontology-based Organization and Description of Web Ser-

vices [76, 78, 81, 11, 4] – Understanding the semantics of Web services is a key requirement for the automatic composition of those services. The first step of our research was thereby to define a framework for the semantic description of Web services. We proposed an *ontology-based framework for organizing Web services and describing their semantics*. Web Services are grouped into communities based on their domain of interest. Service providers identify a community of interest and register their service with it. Each community is an instance of an ontology, called community ontology. The community ontology includes a set of generic operations that can be used "as is" or customized by underlying services.

Multilevel Model for Web Service Composability [83, 79] – Assume now that the composition engine is able to "understand" the meaning and capabilities of each Web service. The issue then is to select "relevant" participants while making sure that those participants "can" actually interact with each other. For that purpose, we proposed a *composability model for semantic Web services*. The model is defined by a set of rules called composability rules. Each rule compares a specific feature of interacting Web services. We organized composability rules into five levels: syntactic, static semantic, dynamic semantic, qualitative, and business process. Each rule specifies the constraints and requirements for checking horizontal, vertical, and hybrid composability. We also introduced the notions of composability degree and τ-composability to cater for partial and total composability.

Automatic Generation of Composite Services [80, 82] – Based on the proposed framework for semantic Web services and multilevel composability model, we designed a *technique for the automatic generation of composite services from high-level specifications of composition requests*. The proposed technique consists of three conceptually separate phases: specification, matchmaking, and generation. We defined a Quality of Composition (QoC) model to assess the quality of the generated composite service. We also proposed two sets of algorithms for enabling one-to-one and one-to-many compositions, respectively. The first set (one-to-one

composition) includes *operation-centric*, *community-centric*, and *message-centric* algorithms. The *operation-centric* algorithm uses operations as a basis for checking composability. It performs an exhaustive search in the service registry. The *community-centric* algorithm uses communities as a basis for selecting service operations. It focuses on communities whose category may be composed with sub-requests' categories. The remaining communities and their imported operations are pruned from the service space. The *message-centric* algorithm checks the composability of messages instead of operations and then derives the operations that could be vertically composed with sub-requests. The one-to-many composition algorithm is based on the notion of flow graph which links together all operations that are horizontally composable.

Implementation and Performance Study [83, 17] – We implemented the proposed techniques in WebDG, a prototype for accessing e-government Web services. We defined an analytical model for studying the performance of the proposed algorithms for composition. We also conducted a set of experiments to evaluate the performance and scalability of these algorithms. We defined a benchmarking framework for Web services. The benchmark enables the generation of a large number of Web services and the simulation of important characteristics of the Web service space such as dynamics and heterogeneity. Its main features include customized generation (i.e., user-controlled generation), use of statistical distribution models (e.g., Poisson for service arrival rate) and extensibility (e.g., adding new service attributes).

8.2 Directions for Future Research

We identify the following directions for future research: dynamic composition of Web services, dependable service composition, the support of mobile services, and grid services.

Dynamic Composition of Web Services – The number of services to be integrated may be large and continuously changing. Web service composition requires flexibility to dynamically adapt to changes that may occur in partners' applications. Participants must be able to respond rapidly to changes where both operational (e.g., server load) and market (e.g., changes in regulations) environments are not easily predictable. Additionally, the competitive nature of the Web makes possible the availability of alternate services that provide "similar" functions. To stay competitive, businesses should team up with the "best" available services at any given time. They need to form *short term* relationships and then disband when it is no longer profitable to stay together. This form of partnership does not assume any *a priori* trading relationship. The support of dynamic composition will facilitate the establishment of *on demand* and *real-time* partnerships. Services will not statically be bound to each other. New partners with relevant features should be dynamically discovered and assembled. Currently, relationships among component services are mostly established at development time. While technologies such as SOAP, WSDL, and UDDI provide capabilities for defining Web services, they clearly are not sufficient to facilitate the establishment of dynamic business relationships. More research effort is needed to enable the creation of dynamic relationships.

Dependable Composition of Web Services – Transaction support is required to provide reliable and dependable execution of composite services. Traditional transaction management techniques [47] are not appropriate in the context of composite services. The participants of a composite service may be heterogeneous and autonomous. They may not be transactional and if they are, their transactional features may not be compatible with each other. In addition, participant services, for different reasons (e.g., quality of services), may not be willing to comply with constraints such as resource locking, until the termination of the composite service execution. New transaction techniques are required in the context of Web services. For instance, it is important to extend the description of services by explicitly describing transactional semantics of Web service operations. An

example is to specify that an operation can be aborted without effect from a requester's perspective. It is also imperative to extend service composition models to specify transactional semantics of an operation or a group of operations. An example is to specify how to handle the unavailability of a participant service. The effective handling of transactional aspects at the composite service level, should be facilitated by exploiting the transactional capabilities of participant services. A few industry standards such as *WS-Coordination* [64], *WS-Transaction* [65], and *Business Transaction Protocol (BTP)* [95] are already emerging for transaction support of composite services.

Support of Mobile Services – In our current work, we focused on Web service composition in "wired" infrastructures with fixed or stationary users. In our future research, we intend to extend our research on Web service composition to wireless environments [137, 126]. Indeed, the past years have witnessed a boom in wireless technologies. Sophisticated wireless devices such as cellular phones and PDAs are now available at affordable prices. Emerging technologies including 3G and 4G (third and fourth generation) are under development to increase the bandwidth of wireless channels. However, most of the proposed Web service concepts cannot or may not be easily applicable to mobile services. This is due to the peculiarities of wireless environments including limited bandwidth, unbalanced client-server communication, limited power supply, and frequent unavailability of wireless networks. For example, using UDDI for discovering Web services requires multiple *costly* round-trips over wireless networks. Invoking Web services using SOAP may increase mobile hosts' power consumption and waiting time. This calls for new techniques to adapt Web services to the wireless world.

Grid Services – Grid computing is another research area we would like to explore in the future. The aim of research on grids is to provide scalable and transparent methods for accessing resources in distributed environments. Grid concepts and technologies were first developed to enable scientific collaborations. Applications

include collaborative visualization of large scientific datasets and computationally demanding data analyses (e.g., in astronomy). Just as the Web began as the technology for scientific collaboration and was recently adopted in various applications such as e-commerce and e-government, a similar trajectory is expected for Grid technologies [45].

One of the challenges is the support of dynamic integration of resources. These resources must be discovered on-the-fly, selected when requests are submitted, and released after the requests are fulfilled. In both grids and Web services, we often need to integrate "resources" across distributed, heterogeneous, and autonomous systems. One possible way to realize grids on the Web is to view grid resources as Web services. In this way, grids can be defined by reusing, assembling, and coordinating existing resources. The challenge is to define techniques for discovering, selecting, creating, and assembling grids using Web service composition techniques.

Bibliography

[1] N. Adam, O. Dogramaci, A. Gangopadhyay, and Y. Yesha. *Electronic Commerce: Technical, Business, and Legal Issues.* Prentice Hall (ISBN: 0139490825), August 1998.

[2] N. R. Adam and Y. Yesha. Strategic Directions in Electronic Commerce and Digital Libraries: Towards a Digital Agora. *ACM Computing Surveys*, 28(4):818–835, December 1996.

[3] S. Aissi, P. Malu, and K. Srinivasan. E-Business Process Modeling: The Next Big Step. *IEEE Computer*, 35(5):55–62, May 2002.

[4] M. S. Akram, B. Medjahed, and A. Bouguettaya. Supporting Dynamic Changes in Web Service Environments. In *Proceedings of the International Conference on Service Oriented Computing*, pages 319–334, Trento, Italy, December 2003.

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture, and Applications.* Springer Verlag (ISBN: 3540440089), June 2003.

[6] ATIS. *Electronic Data Interchange Guideline Consistency Subcommittee (EGCS).* http://www.atis.org/atis/tcif.

[7] K. Baina, K. Benali, and C. Godart. A Process Service Model for Dynamic Enterprise Process Interconnection. In *Proceedings of the International Con-*

ference on Cooperative Information Systems, pages 239–254, Trento, Italy, September 2001.

[8] BEA. *WebLogic Integrator*. http://www.bea.com/products/weblogic/integrator.

[9] BEA, IBM, and Microsoft. *Business Process Execution Language for Web Services (BPEL4WS)*. http://xml.coverpages.org/bpel4ws.html.

[10] B. Benatallah, M. Dumas, M. Sheng, and A. H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proceddings of the IEEE International Conference on Data Engineering*, pages 297–308, San Jose, California, USA, February 2002.

[11] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Composing and Maintaining Web-based Virtual Enterprises. In *Proceedings of the International Workshop on Technologies for E-Services*, pages 155–174, Cairo, Egypt, September 2000.

[12] T. Berners-Lee. *Services and Semantics: Web Architecture*. http://www.w3.org/2001/04/30-tbl, April 2001.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[14] M. Bichler, A. Segev, and J. L. Zhao. Component-based E-Commerce: Assessment of Current Practices and Future Directions. *ACM SIGMOD Record*, 27(4):7–14, December 1998.

[15] A. Bouguettaya, B. Benatallah, and A. K. Elmagarmid. *Interconnecting Heterogeneous Information Systems*. Kluwer Academic Publishers (ISBN 0792382161), July 1998.

[16] A. Bouguettaya, A. Elmagarmid, B. Medjahed, and M. Ouzzani. Ontology-Based Support for Digital Government. In *Proceedings of the International*

Conference on Very Large Databases, pages 633–636, Roma, Italy, September 2001.

[17] A. Bouguettaya, B. Medjahed, A. Rezgui, M. Ouzzani, and Z. Wen. Privacy Preserving Composition of Government Web Services (Demo). In *Proceedings of the NSF Conference for Digital Government Research*, pages 429–432, Los Angeles, California, USA, May 2002.

[18] A. Bouguettaya, M. Ouzzani, B. Medjahed, and J. Cameron. Managing Government Databases. *IEEE Computer*, 34(2):56–64, February 2001.

[19] M. Brodie. The B2B E-commerce Revolution: Convergence, Chaos, and Holistic Computing. In *Information System Engineering: State of the Art and Research Themes. S. Brinkkemper, E. Lindencrona, and Solvberg (editors)*, London, England, June 2000.

[20] C. Bussler. B2B Protocol Standards and their Role in Semantic B2B Integration Engines. *IEEE Data Engineering Bulletin*, 24(1):3–11, March 2001.

[21] C. Bussler, D. Fensel, and A. Maedche. A Conceptual Architecture for Semantic Web Enabled Web Services. *SIGMOD Record*, 31(4):24–29, December 2002.

[22] J. Cardoso and A. Sheth. Semantic e-Workflow Composition. Technical report, LSDIS Lab, Department of Computer Science, University of Georgia, July 2002.

[23] F. Casati, U. Dayal, and M.-C. Shan. E-Business Applications for Supply Chain Automation: Challenges and Solutions. In *Proceddings of the IEEE International Conference on Data Engineering*, pages 71–78, Heidelberg, Germany, April 2001.

[24] F. Casati, D. Georgakopoulos, and M.-C. Shan, editors. *Proceedings of the International Workshop on Technologies for E-Services*, Roma, Italy, September 2001.

[25] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 13–31, Stockholm, Sweden, June 2000.

[26] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a Platform for Developing and Managing Composite e-Services. Technical Report HPL-2000-36, HP Laboratoris, Palo Alto, California, USA, 2000.

[27] F. Casati, M. Sayal, and M.-C. Shan. Developing E-Services for Composing E-Services. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 171–186, Interlaken, Switzerland, June 2001.

[28] F. Casati and M.-C. Shan. Models and Languages for Describing and Discovering E-Services (Tutorial). In *Proceedings of the International ACM SIGMOD Conference on Management of Data*, Santa Barbara, California, USA, May 2001.

[29] D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha. A Reactive Service Composition Architecture for Pervasive Computing Environments. In *Proceedings of the International Personal Wireless Communications Conference*, pages 53–62, Singapore, October 2002.

[30] E. Cobb. The Evolution of Distributed Component Architectures. In *Proceedings of the International Conference on Cooperative Information Systems*, pages 7–21, Trento, Italy, September 2001.

[31] C. Collet, T. Coupaye, and T. Svensen. NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Proceedings of the*

International Conference on Very Large Databases, pages 132–143, Santiago, Chile, September 1994.

[32] CommerceNet. *eCO*. http://eco.commerce.net.

[33] RosettaNet Consortium. *RosettaNet*. http://www.rosettanet.org.

[34] X12 Consortium. *EDI (Electronic Data Interchange) ANSI X12*. http://www.x12.org.

[35] XML/EDI Consortium. *XML/EDI*. http://www.xmledi-group.org.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press (ISBN: 0262032937), September 2001.

[37] F. Curbera, M. Duftler, R. Khalaf, and W. Nagy. Unraveling the Web Services Web. *IEEE Internet Computing*, 6(2):86–93, March 2002.

[38] cXML Consortium. *cXML*. http://www.cxml.org.

[39] U. Dayal, M. Hsu, and R. Ladin. Business Process Coordination: State of the Art, Trends, and Open Issues. In *Proceedings of the International Conference on Very Large Databases*, pages 3–13, Roma, Italy, September 2001.

[40] Y. Ding, D. Fensel, and and B. Omelayenko M. Klein. The Semantic Web: Yet Another Hip? *Data and Knowledge Engineering*, 41(3):205–227, May 2002.

[41] A. Dogac. A Survey of the Current State-of-the-Art in Electronic Commerce and Research Issues in Enabling Technologies. In *Proceddings of the Euro-Med Net Conference, Electronic Commerce Track*, pages 50–53, Nicosia, Cyprus, March 1998.

[42] A. Dogac, editor. *Special Issue on Electronic Commerce*, *ACM SIGMOD Record*, 27(4), December 1998.

[43] A. Dogac, editor. *Special Issue on Electronic Commerce, Distributed and Parallel Databases, an International Journal*, 7(2), April 1999.

[44] A. Dogac and I. Cingil. A Survey and Comparison of Business-to-Business E-Commerce Frameworks. *ACM SIGecom Exchanges*, 2(2):16–27, June 2001.

[45] F. Douglis and I. T. Foster. The Grid Grows Up. *IEEE Internet Computing*, 7(4):24–26, July 2003.

[46] Drala. *Drala Event Broker*. http://www.dralasoft.com.

[47] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann (ISBN:1558602143), December 1992.

[48] D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer Verlag (ISBN: 3540003029), September 2003.

[49] D. Florescu, A. Grunhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proceedings of the International World Wide Web Conference*, pages 65–76, Honolulu, Hawaii, USA, May 2002.

[50] D. Florescu, A. Grunhagen, D. Kossmann, and S. Rost. XL: Platform for Web Services. In *roceedings of the International ACM SIGMOD Conference on Management of Data*, page 625, Madison, Wisconsin, USA, May 2002.

[51] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

[52] D. Georgakopoulos, editor. *Information Technology for Virtual Enterprises*, Proceedings of the International Workshop on Research Issues on Data Engineering, Sidney, Australia, March 1999.

[53] D. Georgakopoulos, H. Schuster, A. Cichocki, and D. Baker. Managing Process and Service Fusion in Virtual Enterprises. *Information Systems*, 24(6):429–456, September 1999.

[54] A. Gomez-Perez and O. Corcho. Ontology Languages for the Semantic Web. *IEEE Intelligent Systems*, 17(1):54–60, January 2002.

[55] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 319–332, San Diego, California, USA, October 2000.

[56] Object Management Group. *Unified Modeling Language Specification (Version 1.3)*. http://www.omg.org/technology/documents/formal/uml.htm, 1999.

[57] J. V. D. Heuvel, J. Yang, and M. P. Papazoglou. Service Representation, Discovery and Composition for E-marketplaces. In *Proceedings of the International Conference on Cooperative Information Systems*, pages 270–284, Trento, Italy, September 2001.

[58] J. Hopkins. Component Primer. *Communications of the ACM*, 43(10), October 2000.

[59] I. Horrocks. DAML+OIL: a Description Logic for the Semantic Web. *IEEE Data Engineering Bulletin*, 25(1):4–9, March 2002.

[60] HP. *NetAction*. http://www.hp.com.

[61] M. N. Huhns. Agents Web Services. *IEEE Internet Computing*, 6(4):93–95, July 2002.

[62] IBM. *Web Services Flow Language (WSFL)*. http://xml.coverpages.org/wsfl.html.

[63] IBM. *WebSphere*. http://www.ibm.com.

[64] IBM. *WS-Coordination*. http://www-106.ibm.com/developerworks/.

[65] IBM. *WS-Transaction*. http://www-106.ibm.com/developerworks/.

[66] IETF. *EDIINT*. http://www.ietf.org.

[67] IETF. *The Internet Engineering Task Force*. http://www.ietf.org.

[68] R. Kalakota and A. B. Whinston. *Frontiers of Electronic Commerce*. Addison Wesley (ISBN: 0201845202), February 2000.

[69] G. Larsen. Component-based Enterprise Frameworks. *Communications of the ACM*, 43(10):24–26, October 2000.

[70] A. Lazcano, H. Schuldt, G. Alonso, and H. J. Schek. WISE: Process based E-Commerce. *IEEE Data Engineering Bulletin*, 24(1):46–51, March 2001.

[71] S. M. Lewandowski. Frameworks for Component-based client/server Computing. *ACM Computing Survey*, 30(1):3–27, March 1998.

[72] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proceedings of the International World Wide Web Conference*, pages 331–339, Budapest, Hungary, May 2003.

[73] H. Ludwig and Y. Hoffner. Contract-based Cross-Organisational Workflows - The CrossFlow Project. In *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration*, pages 1–6, San Francisco, California, USA, February 1999.

[74] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, March 2001.

[75] M. Mecella, B. Pernici, and P. Craca. Compatibility of e -Services in a Cooperative Multi-platform Environment. In *Proceedings of the International Workshop on Technologies for E-Services*, pages 44–57, Roma, Italy, September 2001.

[76] B. Medjahed, B. Benatallah, A. Bouguettaya, and A. Elmagarmid. WebBIS: A Framework for Agile Integration of Web Services. *International Journal of Cooperative Information Systems*, 13(2), June 2004.

[77] B. Medjahed, B. Benatallah, A. Bouguettaya, A. Ngu, and A. Elmagarmid. Business-to-Business Interactions: Issues and Enabling Technologies. *The VLDB Journal*, 12(1):59–85, May 2003.

[78] B. Medjahed and A. Bouguettaya. A Dynamic Foundational Architecture for Semantic Web Services. *Distributed And Parallel Databases, an International Journal*, 2004 (submitted).

[79] B. Medjahed and A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. *IEEE Transactions on Knowledge and Data Engineering*, 2004 (submitted).

[80] B. Medjahed, A. Bouguettaya, and A. Elmagarmid. Composing Web Services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, November 2003.

[81] B. Medjahed, A. Bouguettaya, and M. Ouzzani. Semantic Web Enabled E-Government Services. In *Proceedings of the NSF Conference for Digital Government Research*, pages 250–253, Boston, Massachussets, USA, May 2003.

[82] B. Medjahed, M. Ouzzani, and A. Bouguettaya. Using Web Services in E-Government Applications. In *Proceedings of the NSF Conference for Digital Government Research*, pages 371–376, Los Angeles, California, USA, May 2002.

[83] B. Medjahed, A. Rezgui, A. Bouguettaya, and M. Ouzzani. Infrastructure for E-Government Web Services. *IEEE Internet Computing*, 7(1):58–65, January 2003.

[84] B. Meyer. On To Components. *IEEE Computer*, 32(1):139–140, January 1999.

[85] Microsoft. *Distributed Component Object Model (DCOM).* http://www.microsoft.com.

[86] Microsoft. *.NET.* http://www.microsoft.com/net/.

[87] Microsoft. *Web Services for Business Process Design (XLANG).* http://xml.coverpages.org/xlang.html.

[88] Sun Microsystems. *Java RMI (Remote Method Invocation).* http://java.sun.com/products/jdk/rmi.

[89] Sun Microsystems. *Sun ONE.* http://www.sun.com.

[90] P. Muth, D. Wodtke, J. Weissenfels, A. K. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2):159–184, March 1998.

[91] NASA. *Scientific and Engineering Workstation Procurement (SEWP).* http://www.sewp.nasa.gov.

[92] United Nations. *United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport (UN/EDIFACT).* http://www.unece.org/trade/untdid/welcome.htm.

[93] Netscape. *Secure Socket Layer (SSL) 3.0 Specification.* http://wp.netscape.com/eng/ssl3/.

[94] H. S. Nwana and D. T. Ndumu. An Introduction to Agent Technology. *Lecture Notes in Artificial Intelligence, Number 1198, Springer Verlag*, 1997.

[95] OASIS. *Business Transaction Protocol.* http://www.oasis-open.org/cover.

[96] OASIS and United Nations. *ebXML.* http://www.ebxml.org.

[97] OBI. *OpenBuy.* http://www.openbuy.org.

[98] Oracle. *Integration Server Starter Pack*. http://otn.oracle.com/software.

[99] R. Orfali and D. Harkey. *Client/Server Programming With Java and CORBA*. Wiley Computer Publishing (ISBN: 047124578X), March 1998.

[100] M. Ouzzani, B. Benatallah, and A. Bouguettaya. Ontological Approach for Information Discovery in Internet Databases. *Distributed and Parallel Databases, an International Journal*, 8(3):367–392, July 2000.

[101] M. Ouzzani and A. Bouguettaya. Query Processing and Optimization on the Web. *Distributed and Parallel Databases, an International Journal*, 15(3):187–218, May 2004.

[102] M. Ouzzani, B. Medjahed, and A. Bouguettaya. Optimized Querying of E-Government Services. In *Proceedings of the NSF Conference for Digital Government Research*, pages 363–366, Boston, Massachussets, USA, May 2003.

[103] A. Paepcke, C. K. Chang, H. Garcia-Molina, and T. Winograd. Interoperability for Digital Libraries Worldwide. *Communications of the ACM*, 41(4):33–43, April 1998.

[104] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the International Semantic Web Conference*, pages 318–332, Sardinia, Italy, June 2002.

[105] S. K. Park and K. W. Miller. Random Number Generators: Good Ones Are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[106] C. Petrie and C. Bussler. Service Agents and Virtual Enterprises: A Survey. *IEEE Internet Computing*, 7(4):68–78, July 2003.

[107] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the International World Wide Web Conference*, pages 83–107, Honolulu, Hawaii, USA, May 2002.

[108] S. Ran. A Model for Web Services Discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, March 2003.

[109] E. Roman, S. W. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans*. Wiley Computer Publishing (ISBN: 0471417114), December 2001.

[110] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall (ISBN: 0137903952), December 2002.

[111] C. Schuler, H. Schuldt, G. Alonso, and H.-J. Schek. Workflows over Workflows: Practical Experiences with the Integration of SAP R/3 Business Workflows in WISE. In *Proceedings of the Informatik'99 Workshop "Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications"*, pages 65–72, Paderborn, Germany, October 1999.

[112] H. Schuster, D. Baker, A. Cichocki, D. Georgakopoulos, and M. Rusinkiewicz. The Collaboration Management Infrastructure. In *Proceddings of the IEEE International Conference on Data Engineering*, pages 485–487, San Jose, California, USA, March 2000.

[113] H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and Composing Service-Based and Reference Process-Based Multi-enterprise Processes. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 247–263, Stockholm, Sweden, June 2000.

[114] M. Shen, B. Benatallah, M. Dumas, and E. O.-Y. Mak. SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. In *Proceedings of the International Conference on Very Large Databases*, pages 1051–1054, Hong Kong, China, August 2002.

[115] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[116] S. S. Y. Shim, V. S. Pendyala, M. Sundaram, and J. Z. Gao. Business-to-Business E-Commerce Frameworks. *IEEE Computer*, 33(10):40–47, October 2000.

[117] M. P. Singh. Physics of Service Composition. *IEEE Internet Computing*, 5(3):6, May 2001.

[118] M. Sipser. *Introduction to the Theory of Computation*. Brooks Cole (ISBN 053494728X), December 1996.

[119] B. Spitznagel and Garlan D. A Compositional Formalization of Connector Wrappers. In *Proceedings of the International Conference on Software Engineering*, pages 374–384, Portland, Oregon, USA, May 2003.

[120] K. Sycara, M. Klush, and S. Widoff. Dynamic Service Matchmaking Among Agents in Open Information Environments. *ACM SIGMOD Record*, 28(1):47–53, March 1999.

[121] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley (ISBN: 0201745720), November 2002.

[122] The Stencil Group. *How Web Services Will Beat the "New New Thing" Rap*. http://www.stencilgroup.com/ideas_scope_200106newnew.html.

[123] TIBCO. *ActiveEnterprise*. http://www.tibco.com.

[124] S. Tsur, S. Abiteboul, R. Agrawal, U. Dayal, J. Klein, and G. Weikum. Are Web Services the Next Revolution in e-Commerce? (Panel). In *Proceedings of the International Conference on Very Large Databases*, pages 614–617, Roma, Italy, September 2001.

[125] S. D. Urban, S. W. Dietrich, A. Saxena, and A. Sundermier. Interconnection of Distributed Components: An Overview of Current Middleware Solutions. *Journal of Computer and Information Sciences and Engineering*, 1(1):23–31, March 2001.

[126] U. Varshney and R. J. Vetter, editors. *Special Issue on Mobile Commerce*, MONET 7(3), June 2002.

[127] S. Vinoski. Web Services Interaction Models, Part 1: Current Practice. *IEEE Internet Computing*, 6(3):89–91, May 2002.

[128] Vitria. *BusinessWare*. http://www.vitria.com.

[129] W3C. *Simple Object Access Protocol (SOAP)*. http://www.w3.org/TR/soap.

[130] W3C. *Universal Description, Discovery, and Integration (UDDI)*. http://www.uddi.org.

[131] W3C. *Web Services Description Language (WSDL)*. http://www.w3.org/TR/wsdl.

[132] WebMethods. *WebMethods*. http://www.webmethods.com.

[133] G. Weikum, editor. *Special Issue on Organizing and Discovering the Semantic Web*, IEEE Data Engineering Bulletin, 25(1), March 2002.

[134] J. Weissenfels, M. Gillmann, O. Roth, G. Shegalov, and W. Wonner. The Mentor-Lite Prototype: A Light-Weight Workflow Management System. In *Proceddings of the IEEE International Conference on Data Engineering*, pages 685–686, San Diego, California, USA, February 2000.

[135] M. Wooldrige and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[136] J. Yang and M. P. Papazoglou. Interoperation Support for Electronic Business. *Communications of the ACM*, 43(6):39–47, June 2000.

[137] X. Yang, A. Bouguettaya, B. Medjahed, W. He, and H. Long. Organizing and Accessing Web Services on Air. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 33(6):742–757, November 2003.

[138] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z.Sheng. Quality Driven Web Services Composition. In *Proceedings of the International World Wide Web Conference*, pages 411–421, Budapest, Hungary, May 2003.

Appendix A

Vita

Personal Information

- *Full Name*: Brahim Medjahed
- *E-mail*: brahim@vt.edu
- *Home Page*: http://www.nvc.cs.vt.edu/∼brahim

Research Interests

Databases, Internet Computing, Semantic Web, Web Service, Workflow.

Education

- May 2004: Ph.D. in Computer Science and Applications, Department of Computer Science, Virginia Tech (GPA: 4.0/4.0).
  - Thesis: Semantic Web Enabled Composition of Web Services
  - Advisor: Dr Athman Bouguettaya

- September 1995: M.Sc. in Computer Science, Algiers University of Sciences and Technology, Algeria.
- June 1991: B.Sc. in Computer Science, Algiers University of Sciences and Technology, Algeria.
- June 1986: Baccalaureate (high school diploma), Algiers, Algeria.

Research Experience

- **Research Assistant (May 2000 - December 2003): Department of Computer Science, Virginia Tech**.
  - For my Ph.D. thesis research, I worked on developing an approach for the automatic composition of Semantic Web services. I proposed an ontological framework for organizing and describing Web services on the Semantic Web. Based on this framework, I defined a multilevel composability model to check whether Web services can be combined together. The model checks Web service composability at different levels including syntactic, semantic (static and dynamic), qualitative (Quality of Operation), and business process levels. Finally, I proposed a set of algorithms for checking composability and automatically generating a detailed composite service description from the high-level description of a desired composition. The proposed techniques were implemented in the WebDG prototype, a system for providing customized government Web services (NSF funded project).
- **Visiting Scholar (Summer 2001): Department of Computer Sciences, Purdue University**.
  - I designed and implemented e-services using different e-service platforms including HP NetAction platform, HP e-speak, and UDDI/WSDL/SOAP.

The aim was to compare different e-service platforms. This work was part of a HP-funded project conducted in collaboration between Virginia Tech and Purdue University.

- **Research Assistant (August 1999 - May 2000): Department of Computer Science, Virginia Tech**.
  - My early work in the WebDG Project (funded by NSF) focussed on organizing and querying Web databases. I looked at the use of ontologies to cluster and browse Web databases. Tools used for that purpose include CORBA products (Orbix, OrbixWeb, Visibroker, JavaIDL, etc), RMI, DCOM, EJB and the IIOP standard.

- **Research Assistant (March 1999 - August 1999): School of Information Systems, Queensland University of Technology, Australia**.
  - I worked on the design of a proactive and adaptive transaction model in open, heterogeneous, and distributed environments. This work was funded by an ARC SPIRIT project (Australia).

- **Research Assistant (September 1993 - May 1995): Department of Computer Science, Algiers University of Sciences and Technology, Algeria**.
  - The topic of my M.Sc. research thesis was concurrency control for nested transactions. I designed and implemented a technique based on serialization graphs for checking the serializability of a concurrent execution of nested transactions. This work was part of a project funded by the Algerian Department of Education.

Teaching Experience

- **Teaching Assistant (March 1999 - August 1999): School of Information Systems, Queensland University of Technology, Australia**.
  - *Course: Introduction to Databases.* I conducted weekly Lab sessions, graded projects and exams, and held office hours.
- **Lecturer (September 1995 - February 1999): Algiers University of Sciences and Technology, Algeria**.
  - *Courses: Algorithms - Compilers - Computer Architecture.* I prepared and gave weekly lectures, conducted weekly Lab sessions, designed and graded homeworks, projects, and exams.
- **Associate Lecturer (September 1995 - December 1998): Educaform (Private Undergraduate Institute), Algeria**.
  - *Courses: Introduction to Computer Science - Data Structures - Computer Architecture.* I prepared and gave weekly lectures, conducted weekly Lab sessions, designed and graded homeworks, projects, and exams.
- **Teaching Assistant (September 1992 - June 1994): National Institute of Computer Science, Algiers, Algeria**.
  - *Course: Compilers.* I conducted weekly Lab sessions, graded projects and exams, designed and graded homeworks, and held office hours.
- **Teaching Assistant (September 1993 - June 1994): Algiers University of Sciences and Technology, Algeria**.
  - *Course: Introduction to the Theory of Computation.* I conducted weekly Lab sessions, designed and graded homeworks, graded exams, and held office hours.

Publication

Journals

1. B. Medjahed, B. Benatallah, A. Bouguettaya, and A. Elmagarmid. *WebBIS: A Framework for Agile Integration of Web Services.* International Journal of Cooperative Information Systems, 13(2), June 2004 (to appear).

2. B. Medjahed, A. Bouguettaya, and A. Elmagarmid. *Composing Web Services on the Semantic Web.* The VLDB Journal, Special Issue on the Semantic Web, 12(4), November 2003.

3. X. Yang, A. Bouguettaya, B. Medjahed, H. Long, W. He. *Organizing and Accessing Web Services on Air.* IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, 33(6), November 2003.

4. B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. Elmagarmid. *Business-to-Business Interactions: Issues and Enabling Technologies.* The VLDB Journal , 12(1), May 2003.

5. B. Medjahed, A. Rezgui, A. Bouguettaya, and M. Ouzzani. *Infrastructure for E-Government Web Services.* IEEE Internet Computing, 7(1), January/February 2003.

6. A. Bouguettaya, M. Ouzzani, B. Medjahed, and J. Cameron. *Managing Government Databases.* IEEE Computer, 34(2), February 2001.

7. B. Medjahed, A. Bouguettaya. *Customized Delivery of Pervasive Web Services.* IEEE Computer (under review).

8. B. Medjahed and A. Bouguettaya. *A Multilevel Composability Model for Semantic Web Services.* IEEE Transactions on Knowledge and Data Engineering (under review).

9. B. Medjahed and A. Bouguettaya. *A Dynamic Foundational Architecture for Semantic Web Services*. Distributed And Parallel Databases (under review).

Book Chapters

10. A. Bouguettaya, A. Rezgui, B. Medjahed, M. Ouzzani. *Internet Computing Support for Digital Government*. In The Practical Handbook of Internet Computing, CRC Press (to appear).

11. A. Bouguettaya, B. Benatallah, B. Medjahed, M. Ouzzani, L. Hendra. *Adaptive Web-based Communities*. In Information Modeling for Internet Application, (P. Van Bommel, editor), ISBN 1- 59140- 050- 3, Idea Group Publishing, October 2002.

12. A. Bouguettaya, B. Medjahed, M. Ouzzani, Y. Meng. *Ubiquitous Access to Web Databases*. In Web Powered Databases (D. Taniar and W. Rahayo, editors), ISBN 1-59140-035-X, Idea Group Publishing, August 2002.

13. A. Bouguettaya, M. Ouzzani, B. Medjahed, and A. Elmagarmid. *Supporting Data and Services Access in Digital Government Environments*. In Advances in Digital Government Technology, Human Factors, and Policy (W. J. McIver and A. K. Elmagarmid, editors), ISBN 0-306-47374-7, Kluwer Academic Publishers, May 2002.

Conferences

14. M. S. Akram, B. Medjahed, and A. Bouguettaya. *Supporting Dynamic Changes in Web Service Environments*. 1st International Conference on Service Oriented Computing, SOC 2001, Trento, Italy, December 2003.

15. B. Medjahed, A. Bouguettaya, and M. Ouzzani. *Semantic Web Enabled E-Government Services*. The dg.o 2003 NSF Conference for Digital Government Research, Boston, USA, May 2003.

16. M. Ouzzani, A. Bouguettaya, and B. Medjahed. *Optimized Querying of E-Government Services*. The dg.o 2003 NSF Conference for Digital Government Research, Boston, USA, May 2003.

17. A. Rezgui, M. Ouzzani, A. Bouguettaya, and B. Medjahed. *Preserving Privacy in Web Services*. The 4th International ACM Workshop on Web Information and Data Management, WIDM 2002, Virginia, USA, November 2002.

18. B. Medjahed, M. Ouzzani, and A. Bouguettaya. *Using Web Services in E-Government Applications*. The dg.o 2002 NSF Conference for Digital Government Research, Los Angeles, USA, May 2002.

19. A. Bouguettaya, B. Medjahed, A. Rezgui, M. Ouzzani, and Z. Wen. *Privacy Preserving Composition of Government Web Services* (Demo Paper). The dg.o 2002 NSF Conference for Digital Government Research, Los Angeles, USA, May 2002.

20. A. Bouguettaya, A. Elmagarmid, B. Medjahed, and M. Ouzzani. *Ontology-based Support for Digital Government*. The 27th Conference on Very Large Databases, VLDB 2001, Roma, Italy, September 2001.

21. A. Bouguettaya, A. Elmagarmid, B. Medjahed, and M. Ouzzani. A *Web-based Architecture for Government Databases and Services*. The dg.o 2001 NSF Conference for Digital Government Research, Los Angeles, USA, May 2001.

22. B. Medjahed. *Discovering and Integrating Web-based E-Services*. ACM International Student Research Workshop, Charlotte, North Carolina, USA, February 2001.

23. B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. *Composing and Maintaining Web-based Virtual Enterprises*. First VLDB Workshop on Technologies for E-Services, TES 2000, Cairo, Egypt, September 2000

24. H. Seba, B. Medjahed, and M. Atroun. *Compound Conflict Serializability: A Criterion for Strong Correctness in Multidatabases*, BDA'98 (French Database Conference), Hamamet, Tunisia, October 1998 (in French).

25. H. Seba, B. Medjahed, and M. Atroun. *Preservation of Strong Correctness in Multidatabases*, 2nd National conference on Computer Science, Biskra, Algeria, November 1997 (in French).

26. B. Medjahed. *Serialization Graph Construction in Object Oriented Database Systems*, 3rd International Symposium on Programming and Systems (ISPS'97), Algiers, Algeria, April 1997 (in French).

27. B. Medjahed and N. Belkhodja. *On the Serializability for Nested Transactions*, 3rd IASTED International Conference on Computer Applications in Industry, Cairo, Egypt, December, 1994.

28. B. Medjahed and N. Belkhodja. *A Serializability Approach for Nested Transactions*, 6th Conf. on Parallel and Distributed Systems of French-speaking Countries (RenPar'6), Lyon, France, June 1994 (in French).

29. B. Medjahed, N. Belkhodja, and M. Atroun. *Multilevel and Nested Transactions*, 7th International Conference on Computer Systems (JISI'7), Tunis, Tunisia, May 1994 (in French).

Theses

31. B. Medjahed. M.Sc. Thesis, *Concurrency Control for Nested Transactions*, Algiers University of Sciences and Technology, Algeria, September 1995 (in French).

32. B. Medjahed and R. Hamadi. B.Sc. Project, *Implementation and Comparison of Multiversion Concurrency Control*, Algiers University of Sciences and Technology, Algeria, June 1991 (in French).

Talks and Presentations

1. *Automatic Composition of Web Services.* Invited talk, Rutgers University, July 2003.

2. *Semantic Web Enabled E-Government Services.* Paper presentation, dg.o 2003 NSF Conference for Digital Government Research, May 2003.

3. *Semantic Web Support for Digital Government.* Birds of a feather session, dg.o 2003 NSF Conference for Digital Government Research, May 2003.

4. *Optimized Querying of E-Government Services.* Poster presentation, dg.o 2003 NSF Conference for Digital Government Research, May 2003.

5. *Declarative and Dynamic Composition of Web Services.* Invited talk, Mitretek, June 2002.

6. *WebDG: Integrated Access to E-Government Databases and Services.* Demo Presentation, Virginia Department for the Aging, October 2002.

7. *WebDG: An E-Government Infrastructure for Uniform Privacy Preserving Access to Autonomous Web Databases and Services.* Paper presentation, AOL Common Ground Research Day Workshop, November 2002.

8. *Uniform and Secure Access to Web Databases and Services.* Paper presentation, AOL Common Ground Research Day Workshop, November 2001.

9. *Using Web Services in E-Government Applications.* Paper presentation, dg.o 2002 NSF Conference for Digital Government Research, May 2002.

10. *Privacy Preserving Composition of Government Web Services.* Demo presentation, dg.o 2002 NSF Conference for Digital Government Research, May 2002.

11. *Web-based Architecture for Government Databases and Services.* Demo presentation, dg.o 2001 NSF Conference for Digital Government Research, May 2001.

12. *Managing Government Databases.* Demo Presentation, Indiana Family and Social Services Administration, August 2001.

13. *Ontology-based Support for Digital Government.* Paper presentation, VLDB 2001 Conference, September 2001.

14. *Discovering and Integrating Web-based E-Services.* Poster presentation, ACM International Student Research Workshop, February 2001.

Grants

- I participated in the write-up of two research grant proposals successfully funded by NIH and Sun Microsystems respectively. I also helped in the write-up of an NSF research grant proposal (under review).

- I attended two workshops at Virginia Tech on grant writing. The titles of the workshop were "Finding Funds" and "Writing Successful Grants".

Awards

- April 2003: Virginia Tech's Award to present a paper at dg.o.2003 Conference

- November 2002: Outstanding reviewer award, IEEE Internet Computing.

- December 2001: Computing Research Association's (CRA) award to attend the CRA Academic Careers Workshop.

- September 2001: Virginia Tech's award to present a paper at VLDB 2001 Conference
- February 2001: ACM award to present a paper at the ACM International Student Research Workshop, Charlotte, North Carolina, USA, February 2001.
- March 1999: Recipient of a three-year Scholarship for Ph.D. Studies, School of Information Systems, Queensland University of Technology, Brisbane, Australia.

Honors

- Invited to appear in the "Who's Who In America" (October 2004).
- Student Member of the IEEE, IEEE Computer Society, and ACM since 1999.
- B.Sc. and M.Sc. in Computer Science with First Honor.

Professional Activities

Program Committee Membership

- International Workshop on Ubiquitous Computing (IWUC 2004), April 13-14 2004, Porto, Portugal.
- First IEEE International Workshop on Electronic Contracting (WEC'04), July 6-9, San Diego, California.

Workshop and Seminar Organization

- I Helped in the organization of the IEEE RIDE (Research Issues on Data Engineering) 2004 Workshop. I was responsible of maintaining the workshop Web site, managing submissions, and handling reviews.

- Student volunteer in the 32nd ACM Technical Symposium on Computer Science Education.
- Fall 2002 - Spring 2003: Coordinator of ECEG (E-Commerce & E-Government Research Lab) Seminars.

Technical Refereeing

- **Journals:**
  - ACM Computing Survey, 2001. IEEE Transactions on Knowledge and Data Engineering, 2001. IEEE Internet Computing, 2002. Distributed and Parallel Databases Journal, 1999, 2000, 2001, 2002, and 2003. Encyclopedia of Biomedical Engineering, 2003. Knowledge and Information Systems, 2000. International Journal of Information Technology, 1999.
- **Conferences:**
  - International Conference on Data Engineering (ICDE), 2002. International Conference on Extending Database Technology (EDBT), 2004. International Conference on Web Services (ICWS), 2003. International Conference on Electronic Commerce (CEC), 2003. International Conference on Information and Knowledge Management (CIKM), 2001. International Conference on Tools with Artificial Intelligence (ICTAI), 1999, 2000, 2003. International Conference on Cooperative Information Systems (CoopIS), 1999. Distributed Objects and Applications Conference (DOA), 1999. VLDB Workshop on Technologies for E-Services (TES), 2003. Workshop on Research Issues on Data Engineering (RIDE), 2002. International Workshop on Mobility in Databases and Distributed Systems (MDDS), 2000.

Students Co-Advising

- M.Sc. Independent Study Projects, Department of Computer Science, Virginia Tech.

- Yu Wang. *Implementation of Distributed Ontologies.*
- Edward Gemperline. *Deploying E-Services Using E-Speak and DCOM.*
- Zhaohong Li. *Deploying E-Services Using E-Speak and EJB.*
- Rongfang Ma. *A Uniform Interface for Querying Government Databases and Invoking Services.*
- Xinyuan Zhang. *Benchmarking Web Services.*
- Hongmei Hao. *Accessing E-Government Web Services.*
- Hao Long. *Implementing Web Services for Government Applications.*