Decentralized Discovery and Execution for Composite Semantic Web Services

by

Dominik Roblek

A Dissertation submitted to the University of Dublin,

in partial fulfillment of the requirments for the degree of

Master of Science in Computer Science

2006

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other university, and that unless otherwise stated, is my own work.

Dominik Roblek

September 11, 2006

Permission to Lend and/or Copy

To my wife Maryna

for her love and encouragement.

Acknowledgments

Many thanks are due to my supervisor Dr. Dave Lewis for ideas, help, advice, reviews, patience and guidance throughout this project.

Special thanks to Dr. John Keeney for ideas, help, advice and for reviewing my work.

I would like to thank the friendly and open people of Ireland for having made this year unforgettable.

And to the NDS class of '06 – it's been a great fun.

Dominik Roblek

University of Dublin, Trinity College
September 2006

Decentralized Discovery and Execution for Composite Semantic Web Services

Dominik Roblek
University of Dublin, Trinity College, 2006

Supervisor: Dave Lewis

Current mechanisms for discovery of Web services and execution of compositions of Web services are centralized. Services are advertised by means of central service registries. Conventional models of composite service execution employ a central coordinator. This represents a performance bottle neck and a single point of failure. Besides, all process data are exposed to a central entity, which is not appropriate for processes involving sensitive data. Bindings between component services are determined at the composite service definition time.

The dissertation addresses these issues by developing a novel decentralized peer-to-peer discovery and execution model for composite semantic Web services. Semantic description of services enables inexact matching of service consumer requirements and service provider capabilities. The service advertisement and discovery is performed by a loosely coupled knowledge-based network. Bindings between component services are dynamically determined at the composite service execution time. A composite service execution is decentralized and orchestrated by the providers of the participating component services.

In order to enable effective semantic service discovery the dissertation extends the knowledge-based network subscription language with a bag type and a novel family of bag operators, termed composite bag operators, that offer a much higher expressiveness for semantic information filtering than current operators.

The proposed decentralized discovery and execution model is robust and suitable for dynamic distributed environments with a high service churn rate.

Contents

List of Tables

List of Figures

List of Listings

Notation

Notation for Ontology Resources

The following notation is used in this document for ontological resources that appear in the text:

- Classes and properties from OWL ontology are prefixed with **owl:**.
- Classes and properties from OWL-S ontologies are prefixed with **owls:**, regardless whether they are actually defined in Service.owl, Profile.owl, Process.owl or Grounding.owl.
- Classes and properties from ontologies developed in scope of this project are prefixed with **dowls:**, regardless of the actual name of their respective ontology.
- Other ontological resources are written without prefix.

Ontological resources that appear in the listings are written as they appear in their respective source files.

Notation for Listings

In listings three consecutive dots (. . .) denote omitted content.

Chapter 1

Introduction

1.1 Motivation

Most of today's World Wide Web is designed for human consumption. It lacks machine-processable information that would allow automated interoperation. Such interoperation is realized through hand-coded configuration code to locate, orchestrate and invoke Web services. The Web is not organized in a form that would allow machines to find, share, and combine information effectively (Antoniou and van Harmelen, 2004).

Semantic Web is an extension of the current Web that aims to improve the current state by providing machine-processable information. The primary concept of Semantic Web are ontologies, which are controlled vocabularies to describe objects and relations between them in a formal way (Antoniou and van Harmelen, 2004). While ontological representation of information has been thoroughly studied, little has been done to exploit it in disseminating semantically enriched information through the communication networks.

The Web, once solely a repository for information, is evolving into a provider of Web services, such as e-commerce and business-to-business services (McIlraith et al., 2001) and a variety of personal and business productivity applications. By allowing applications to be encapsulated in a reusable standardized format, Web services have enabled businesses to share functionality with arbitrary numbers of partners, without having to pre-negotiate communication mechanisms or syntax representations (Cabral et al., 2004). Although Web services have reached a certain degree of maturity, many limitations of the existing approaches still exists.

A typical Web service architecture consists of three entities: (i) service providers that create and publish Web services, (ii) service brokers that maintain a registry of published services and support their discovery, and (iii) service consumers that search the service brokers registries. Registries of published services have traditionally had a centralized architecture consisting of multiple repositories that synchronize periodically (Schmidt and Parashar, 2004). Such architecture is not suitable for dynamic decentralized peer-to-peer environments with high peer churn rate, where each participant can dynamically adapt any, or more of these roles. In order to enable peer-to-peer interoperability it is fundamental to make services computer interpretable. This is provided by semantic Web services that augment Web services with ontological descriptions of their capabilities, thus facilitating automated discovery, orchestration, dynamic binding, and invocation (Cabral et al., 2004).

Organizations can combine Web services into workflows to provide new added value. Workflows can be exposed as Web services themselves. Such Web services are called *composite Web services*. Current mechanisms for executing compositions of Web services are centralized, relying on a central workflow engine to manage the flow of control and data between web service invocations. The disadvantages of this approach are:

- Centralized process control clearly represents a performance bottle neck and a single point of failure.
- Centralized process control is suitable for intra-enterprise process management. However, when multiple parties belonging to different enterprises participate in the process, they are reluctant to give away too much control and to share all the process data. Rather, they need to collaborate with business partners directly (Chen and Hsu, 2001).

Bringing e-commerce to its full potential requires a decentralized peer-to-peer approach, where anybody is able to trade and negotiate with everybody else (Fensel and Bussler, 2002).

Compositions of Web services typically bind participating services in the design time. Such workflow is not robust, since unavailability of any component service renders the whole composition unavailable. Static compositions are also not capable of adapting to the changing conditions in the environment.

1.2 Research Objectives

In the previous section the following issues were identified:

1. Existing network communication patterns have poor support for effective dissemination of ontologically enriched data.
2. Web services are advertised and discovered through central repositories. However as the number of Web services grows and become more dynamic, such architecture quickly becomes inadequate (Schmidt and Parashar, 2004).
3. Centralized Web service compositions are not suited for dynamic peer-to-peer environments. A central process scheduler represents a performance bottle neck and a single point of failure. Besides, all process data are exposed to the central scheduler, which makes this approach inappropriate for inter-enterprise process management involving sensitive data (Chen and Hsu, 2001).
4. Static compositions of Web services are not able to adapt changing conditions in

the environment.

The objective of this research is to provide solutions for certain aspects of the above problems. This work aims to investigate decentralized semantic Web service discovery and associated decentralized execution of composite semantic Web services. The design should use semantic descriptions of services to achieve effective discovery and loose coupling. Workflow bindings should be based on inexact matching. The associations between participating parties should be provisional to accommodate dynamic conditions in the environment, like high churn rate. The solution design should not rely on central entities, because they represent a single point of failure and hinder scalability. In order to achieve this we need to investigate the folllowing:

1. Content-based networking is a communication pattern whereby the flow of messages from senders to receivers is driven by the content of the messages. Traditionally message content is structured as a set of attribute/value pairs and a selection predicate supports simple constraints over the values of individual attributes (Carzaniga et al., 2004). Keeney et al. (2006b) have extended content-based networking with support for simple ontological constraints that are able to compare two ontological concepts with *subsumes*, *subsumed by* and *equivalent*. This research will try to bring the support for semantically enriched messages a step further by providing support for collections of ontological concepts.
2. The content-based networking middleware extended in such manner will be considered for driving decentralized semantic service advertisement and discovery in a peer-to-peer topology. It requires understanding the capabilities of advertised services and how the capabilities match the service consumer requirements. The research will draw on the ideas of the proactive service discovery model utilizing content-based networking developed by Lynch (2005).
3. This project will involve implementing a decentralized workflow execution model using ontology-based descriptions of Web services so that the workflow bindings between them can be made dynamically based on inexact matches, thus promoting much more flexible and robust workflows. The responsibility of coordinating the workflow will be distributed across several peer software components. The workflow model will exploit the capabilities of the semantic service discovery model driven by content-based networking for runtime service discovery and selection. In addition, the research will investigate how far a content-based routing approach can support decentralized workflow execution. Decentralization and semantic interoperability will provide looser binding between Web services in a composition, supporting runtime reconfiguration and thus robustness.

1.3 Research Approach

First of all the current state of the art will be investigated. This will include exploration of the ideas, propositions, designs and solutions in this area up to date.

The rest of the research will be carried out in three phases with each next phase building upon the previous. The phases will correspond to the three research objective issues presented in the previous section. During each phase first a theory will be developed simultaneously with a design of a prototype. Then the prototype will be implemented and used to evaluate the findings of the research. Such approach will minimize the risk of losing the way and running out of time.

This report itself will not be divided by the research phases, but will present all three phases as an integral work.

1.4 Dissertation Roadmap

The remainder of this report is organized as follows:

Chapter 2 - State of the Art provides essential background information and analyses the current state of the art in the Semantic Web, Web Services and Content-Based Networking domain.

Chapter 3 - Design describes the design of the bag extension for content based-networking, and the desing of the decentralized discovery and execution model for composite semantic Web Services.

Chapter 4 - Implementation described the implementation of the prototype.

Chapter 5 - Evaluation provides the evaluation of the design.

Chapter 6 - Conclusion presents the contributions of this research and further work.

Chapter 2

State of the Art

2.1 Service Oriented Architecture

2.1.1 Concepts

Service-Oriented Architecture (SOA) is a style of software architecture that utilizes loosely coupled interoperable services to support the requirements of software users. Business applications and resources are delivered as services. SOA is often based on Web services, since they have broad support in the industry. However, SOA can be implemented using any service-based technology.

The OASIS SOA Reference Model group (OASIS, 2006) defines SOA as:

> Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

In SOA a service is a mechanism to enable the service consumers to access to one or more capabilities of the service provider. It is independent as much as possible from specific platforms and computing paradigms. The access to the service is provided through a prescribed interface. It is assumed that service interface is cleanly separated from its internal implementation. The service could carry out its described functionality through one or more processes that themselves could invoke other available services. The consequence of invoking a service is a realization of one or more real world effects. These effects may include (i) information returned from the provider to the consumer, and (ii) change to the shared state of defined entities (OASIS, 2006). The choreography and orchestration can combine invocations of many services into a new process.

2.1.2 Enterprise Service Bus

Enterprise Service Bus (ESB) is an implementation of Service-Oriented Architecture. It provides foundational services via an event-driven messaging engine (the bus). The primary aim of ESB is to enable loosely coupled integration of various enterprise services and applications.

The notion of ESB is vague and usually includes the following capabilities:

- It has standard-based adapters to connect to third party systems.
- Messaging engine support diverse interaction models (synchronous, asynchronous, point-to-point, publish-subscribe) and communication protocols.
- It has the ability to transform messages from one format to another, and to split or combine multiple messages.
- It is aware of connected applications and services and uses content-based routing facilities to make informed decisions about how to communicate with them (Papazoglou and van den Heuvel, 2005).
- It includes support for service orchestration and choreography.
- It implements a security model to authenticate and authorize use of the ESB.
- It provides monitoring the course of events.

The majority of ESB implementations strongly rely on XML and Web services technologies. ESB operations are usually governed by rule-based policies, which are often delivered in non-centralized fashion.

There are many open source ESB implementation available, for example Apache ServiceMix (Apache Software Foundation, 2006b), Codehaus Mule (Codehaus, 2006), ObjectWeb Celtix (ObjectWeb Consortium, 2006), and commercial ESB implementations available.

2.1.3 Java Business Integration

2.2 Web Services

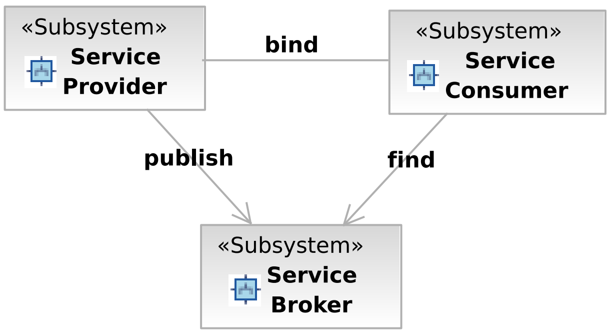
A Web service is an application that is accessible to other applications over the Internet and uses a standardized XML to encode communications. It is defined by W3C (2004d) as

> a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

A typical Web services architecture consists of three entities (Roy and Ramanujan, 2001) presented on Figure 2.1:

- **Service Providers** create Web services and publish them to the outside world by registering the services with service brokers.
- **Service Brokers** maintain a registry of published services.
- **Service Consumers** find required services by searching the service brokers registry and then bind their applications to the service provider to use particular services.

Figure 2.1: Web services architecture


«Subsystem» Service Provider
bind
«Subsystem» Service Consumer
publish
find
«Subsystem» Service Broker

Other models, for example based on peer-to-peer topology, exist and will be discussed later in this work.

2.2.1 Core Specifications

The specifications that define Web services are deliberately modular, to allow addition of new supplemental specifications when necessary. The core specifications are the following:

SOAP (W3C, 2003) is a protocol for exchanging messages in common XML format over the Web. Examples of valid application layers for SOAP are HTTP, SMTP and XMPP. HTTP is by far most widely used. As a communication protocol SOAP is stateless and one-way (Alonso et al., 2004, pg. 156). Because SOAP messages are encoded in human readable XML, SOAP has high overhead and so relatively poor performance compared to binary protocols such as CORBA IIOP or Java RMI.

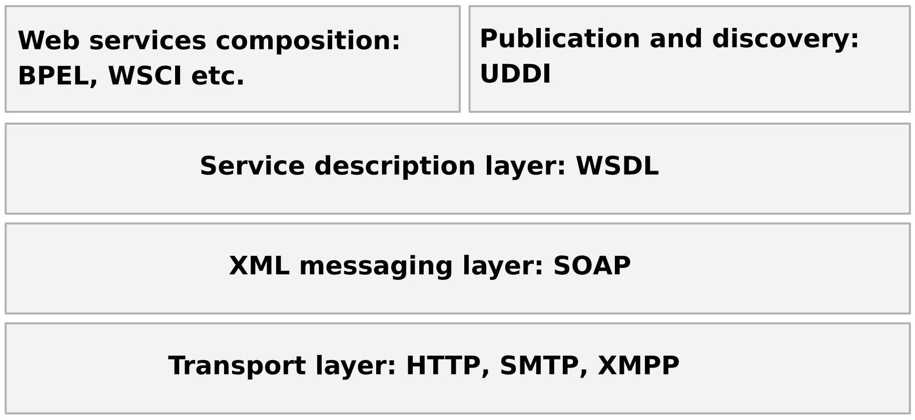
Web Services Description Language (WSDL) (W3C, 2001) is an XML language used to describe Web Services. In particular it describes service interfaces and their bindings to specific protocols, for example (HTTP, SMTP, XMPP).

Universal Description, Discovery, and Integration (UDDI) (OASIS, 2004) is an XML-based registry used to register and locate Web services. UDDI is itself implemented as a Web service. A UDDI registry consits of three components:

- **White Pages** are listing of organizations, their contact information (for example, e-mail address, telephone number), and of the services these organizations provide (Alonso et al., 2004, pg. 175).

- **Yellow Pages** are listings of Web services categorized by some industrial taxonomy.
- **Green Pages** provide technical information about how to invoke the Web service.

Figure 2.2: Web services protocol stack


Web services composition: BPEL, WSCI etc.
Publication and discovery: UDDI
Service description layer: WSDL
XML messaging layer: SOAP
Transport layer: HTTP, SMTP, XMPP

2.2.2 UDDI Deficiencies

Pokraev et al. (2003) pointed out the following UDDI flaws:

- Lack of semantic description mechanisms in WSDL and UDDI to understand the queries and reason about the knowledge.
- Lack of mechanisms for inexact match. For example, a request for a ”accommodation services” cannot discover ”hotel services”.
- Lack of ontology support. Service providers and service consumers have different knowledge about a service. Service descriptions and service requests have to be understood and agreed upon between the two parties by means external to the registry. A common ontology is a must in order to facilitate an effective discovery process.

2.2.3 Web service composition

If the implementation of a Web service's process involves the invocation of other Web services this is termed a composite Web service. A composite Web service is composed of other elementary or composite services. This approach allows the definition of increasingly complex services by progressively aggregating service components at higher levels of abstraction. A client invoking a composite service can itself be a Web service (Dustdar and Schreiner, 2005).

Service composition is closely related to workflows (Piccinelli and Williams, 2003), since a composite service actually organizes other services into a workflow. There are two typical system architectures for workflow management: the centralized client-server based architecture and the decentralized peer-to-peer based architecture. Grid workflows are also becoming an emerging area since they address computationally intensive e-science and e-business processes (Shen et al., 2006).

Web service composition is an implementation technology. It is supported by middleware that typically includes the following (Alonso et al., 2004, pg. 249-250, 256):

- **Composition model and language** specifies the services to be combined, the order in which these services are to be invoked, and the way in which parameters are passed. In addition it might specify pre- and post-conditions of individual steps. The specification of composite service in a composition language is called *process schema* or simply *process*.
- **Development environment** assists the developer to specify *composition schema*.
- **Run-time environment** executes the composite service by executing steps described by composition schema.

Alonso et al. (2004, pg. 256) describe six different dimensions of service composition models:

- **Component model** defines the nature of the components to be composed in terms of assumptions that the model makes on such components.
- **Orchestration model** defines abstractions and languages used to define the order in which components are to be invoked. Orchestration models use process-modelling languages, like activity diagrams, Petri-nets, π-calculus, state charts, and rule-based orchestration.
- **Service selection model** defines how a Web service is selected as a component, either statically at design-time or dynamically during run-time.
- **Data and data access model** defines how data are defined and exchanged between components.
- **Transactions** define how transaction semantics is associated with the composition model.
- **Exception handling** defines how exceptional situations that can arise during execution of the composite model should be handled.

2.2.3.1 Business Process Execution Language (BPEL)

There exist a number of Web service composition languages that vary in realization and functionality. BPEL (IBM et al., 2005) seems to have the strongest industrial support. BPEL is a Web service composition language layered on top of WSDL. In BPEL, the composition result is called a process, participating services are called partners, and message exchange or intermediate result transformation is called an activity. A process consists of a set of activities (Milanovic and Malek, 2004).

The basic concepts of BPEL are (Milanovic and Malek, 2004):

- process initiation: `<process>`

- definition of services participating in composition: `<partnerLink>`
- synchronous and asynchronous calls: `<invoke>`, `<invoke>...<receive>`
- intermediate variables and results manipulation: `<variable>`, `<assign>`, `<copy>`
- error handling: `<scope>`, `<faultHandlers>`
- sequential and parallel execution: `<sequence>`, `<flow>`
- logic control: `<switch>`

BPEL supports static composition where component services are selected in advance. This is not flexible since any changes to the component services require changes to the overall composition. The BPEL process definition can be execute by a BPEL orchestration engine, such as Actvie BPEL (Apache Software Foundation, 2006a).

There are several BPEL implementations available, for both J2EE and .NET platforms. Some other composition languages for Semantic Web services will be discussed later in this chapter.

2.2.4 Decentralized Execution of composite Web services

Standard BPEL orchestration engines assume that the execution of composite Web service is controlled by a single node that acts as the central scheduler. This approach has many drawbacks since it generates a large amount of round-trip messages between participating component service providers and central scheduler. The central scheduler also represents a bottleneck and a central point of failure (Benatallah et al., 2003). For these reasons alternative approaches based on a decentralized peer-to-peer execution of a composite process have been suggested.

2.2.4.1 Self-Serv

Self-Serv is a middleware infrastructure for decentralized execution of Web services in a peer-to-peer topology developed by Benatallah et al. (2003).

Self-Serv supports declarative Web Service composition based on *state charts* that encode a composition model. A Self-Serv state chart is comprised of states that are either basic or compound, and transitions between states. A basic state coresponds to the invocation of an elementary or composite Web service and a compound states corresponds to a subordinate state chart.

In Self-Serv a *service container* is a component that aggregates a set of Web services. The set membership is not fixed and can change dynamically at run-time. The service container is a Web service itself. It facilitates dynamic late binding, since instead of invoking a member Web service directly, the service consumer can send the invocation request to the service container, and let the service container decide which element Web service will perform the requested operation. A Web service can be a member of a container in the following modes:

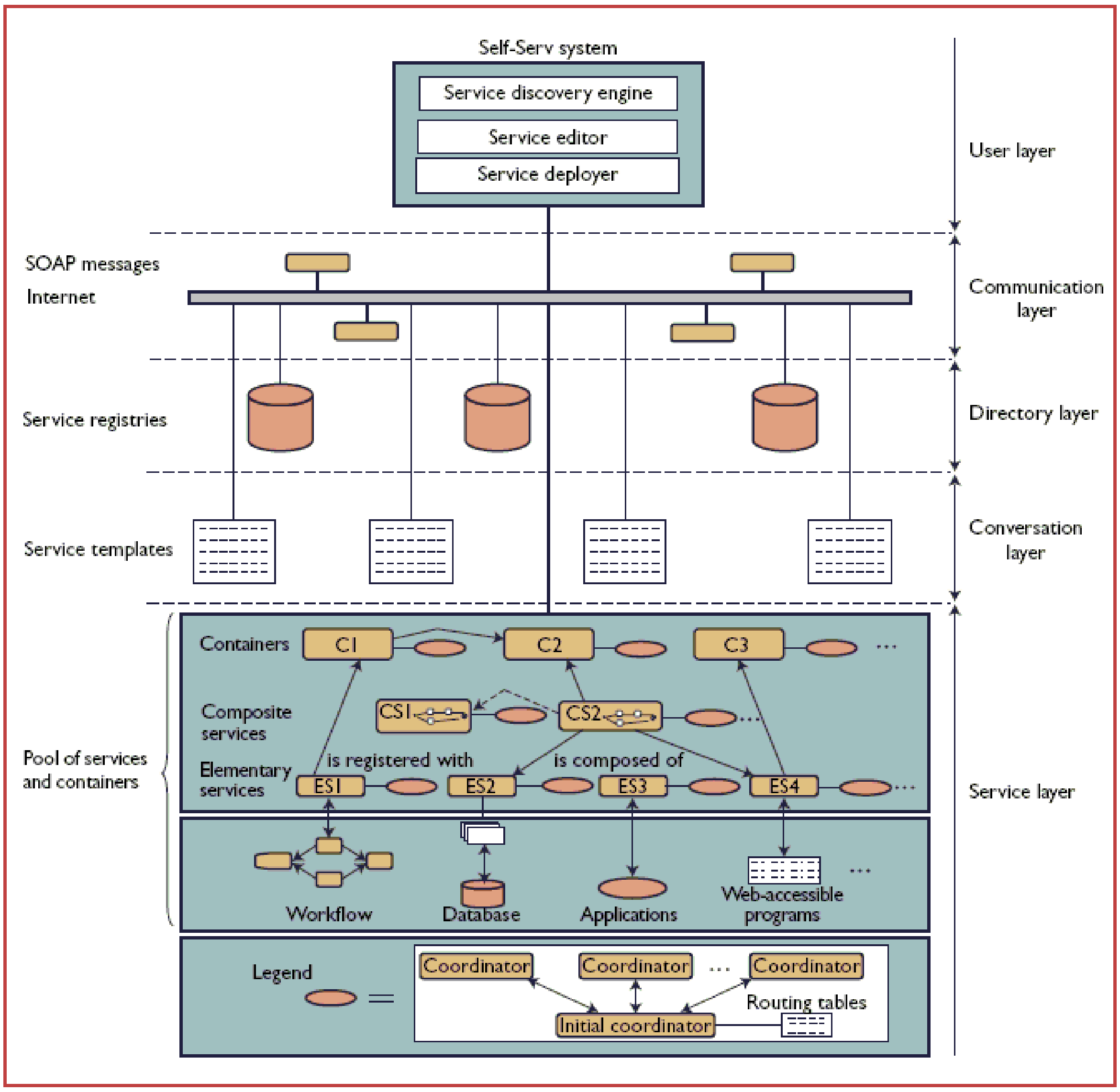
- Services in **explicit mode** are registered with the container statically.

- Services registered in **query mode** are specified in form of a queries to service registries, such as a UDDI registry.
- In **registration mode** services need to register with the container themselves to become members.

For each state of the composite service Self-Serv generates a state coordinator. The state coordinator is hosted by the provider of the service associated with the state. During decentralized composite service execution the coordinator is responsible for receiving notifications of completion from each state coordinator, executing its state, and sending notifications of completion to the coordinators of the states that might need to be entered next.

Each coordinator has a routing table, which specifies the pre-conditions that must be satisfied prior to the invocation of the associated service, and post-processing actions that must be performed after the execution of the associated service is complete.

Figure 2.3: Self-Serv layered architecture (Benatallah et al., 2003)



Self-Serv has a layered architecture as presented in Figure 2.3.

Service layer consists of composite services, service containers and state coordinators.

Conversation layer provides support for standardized interactions among services. It consists of a set of predefined service templates for various industrial standards (for example, Electronic Data Interchange, RosettaNet).

Directory layer stores metadata about services and container. The implementation described in (Benatallah et al., 2003) uses a centralized UDDI for this purpose.

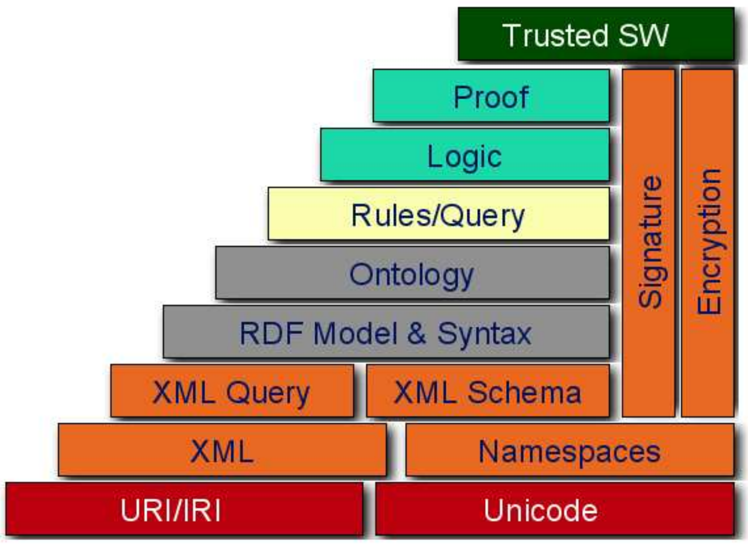
Communication layer uses SOAP to facilitate message exchange among participating nodes.

User layer provides three components to assits the user. The *service discovery engine* assits the user to locate the services, which can be imported into service containers, or used as components in composite services. The *service builder* is used to create and configure composite services and service container. The *service deployer* generates routing tables from the state charts and uploads them to the hosts of the corresponding component services.

2.3 Semantic Web

Most of today's World Wide Web is designed for human users. There is little support for automated access to the resources on the Web. Web content is only rarely machine-accessible. Semantic Web is an initiative that aims to use machine-processable Web information to improve the current state of World Wide Web. Key technologies of Semantic Web are explicit metadata (XML), ontologies, logic and inferencing, and intelligent agents. The development of Semantic Web consists of layers, where each new layer is build upon another, as shown on Figure 2.4 (Antoniou and van Harmelen, 2004).

Figure 2.4: Semantic Web layers



2.3.1 Ontologies

In computer science, an ontology is a data model that represents the knowledge about the domain and is used to reason about the objects in that domain and the relations between them. Ontologies generally describe:

- **Classes**: families of objects (for example, city or country)
- **Individuals**: the basic objects (for example, Dublin or Ireland)
- **Attributes**: properties, features, characteristics, or parameters of objects (for example Ireland has 4 million residents)
- **Relations**: the relationships between objects (for example, Dublin is the capital of Ireland)

2.3.1.1 RDF and RDF Schema

RDF (W3C, 2004b) is a graph-based data model that provides foundation for representing and processing metadata (Antoniou and van Harmelen, 2004). An RDF graph is a set of statements. A statement is a triple comprised of a subject, a predicate and an object. The subject of an RDF statement is a resource, either anonymous, or named by a Uniform Resource Identifier (URI). The predicate is a resource as well, representing a relationship. The object is a resource or a string literal.

RDF Schema (W3C, 2004c) builds on top of RDF and provides a mechanism for describing specific domains. It is a primitive language that offers a few basic modelling primitives with fixed meaning.

Some of the key primitives of RDF and RDFS are:

- **rdfs:Resource** is the class of all resources.
- **rdfs:Class** is the class of all classes.
- **rdf:Property** is the class of all properties.
- **rdf:type** relates resource to its class.
- **rdfs:subClassOf** allows to declare hierarchies of classes.
- **rdfs:domain** of a rdf:Property declares the class of the subject in a triple using this property as predicate.
- **rdfs:range** of a rdf:Property declares the class or datatype of the object in a triple using this property as predicate.

There exist many query languages for getting information from RDF graphs. One of these is SPARQL, which is able to (W3C, 2006):

- extract information in the form of URIs, blank nodes, plain and typed literals,
- extract RDF subgraphs, and
- construct new RDF graphs based on information in the queried graphs.

2.3.1.2 OWL

RDF and RDF schema allow a limited representation of ontological knowledge. Their primary concern is the organization of concepts in typed hierarchies. However many more sophisticated ontological constructs are missing. For example, using RDF and RDFS it is not possible to express that some classes are disjoint (for example, males and females), or to create classes which are unions, intersections or complements of other classes. It is also not possible to express cardinality restrictions. These facilities are provided by Web Ontology Language (OWL) (W3C, 2004a), which is an ontology language on top of RDF and RDFS (Antoniou and van Harmelen, 2004).

OWL comes in three flavors: OWL Lite, OWL DL, and OWL Full. These flavors incorporate different features, with OWL Lite being the simplest and OWL Full the most expressive. OWL Lite and OWL DL are constructed in such a way that every statement can be decided in finite time; OWL Full can contain infinite loops.

2.4 Semantic Web Services

Semantic Web Services describe the various aspects of a Web Service using explicit, machine-understandable semantics, enabling the automatic location, combination and use of Web Services. These potential benefits have led to number of initiatives both in industry and academia (Lara et al., 2005). The major specification models to semantically describe services and support their automated discovery, execution, composition and seamless interoperation, are OWL-S (Martin et al., 2004b), WSMO (de Bruijn et al., 2005) and WSDL-S (Akkiraju et al., 2005).

2.4.1 OWL-S

OWL-S (Martin et al., 2004b) defines an OWL ontology for Web services with four major elements (see Figure 2.5):

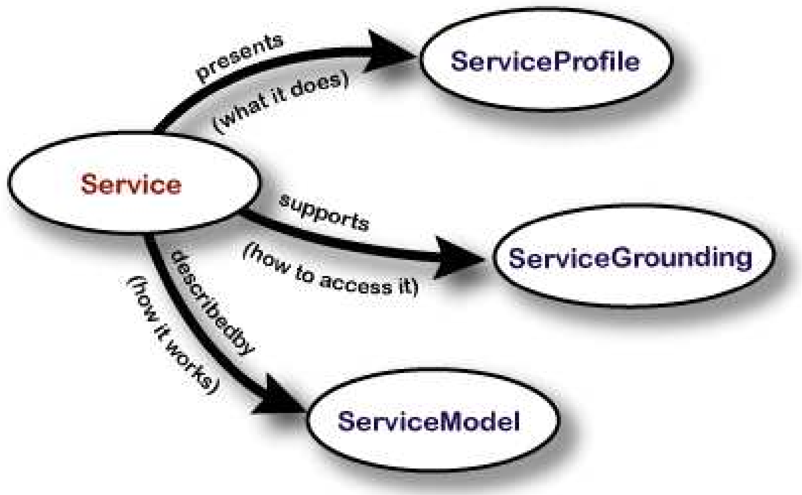
Service serves as an organizational point of reference for declaring Web Services. Every service is declared as an instance of the Service class.

Service Profile describes what the service does in a way that is suitable for service advertisement, discovery and selection.

Service Model tells a client how to use the service. It details the semantic content of requests, the conditions under which particular outcomes will occur, and the exact order of steps that must be performed. For composite services this description may be used by a service consumer to compose and coordinate multiple services to perform a specific task.

Service Grounding describes how a service consumer can invoke the service. Typically the grounding specifies the communication protocols, message formats, and addresses used in contacting the service.

Figure 2.5: Top level of the OWL-S ontology (Martin et al., 2004b)



2.4.1.1 Service Profile

The OWL-S Service Profile presented in the Figure 2.6 provides high level descriptions of a service as a transformation from one state to another. It provides a view of the Web service as a process which requires inputs, and some precondition to be valid, and it results in outputs and some effects to become true. OWL-S provides a schema by which Service Profiles can be subclassed to describe a specific class of capabilities Martin et al. (2004c).

2.4.1.2 Service Model

The OWL-S Service Model presented on Figure 2.7 describes how a service can be used. It tells a client how to use the service, by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and in case of the composite service, the step by step processes. OWL-S defines a Process, which is a subclass of Service Model. OWL-S defines three types of processes (Martin et al., 2004b):

Atomic Process corresponds to the action a service can perform in a single interaction.

Composite Process corresponds to the action that requires multi-step actions.

Simple Process offers an abstraction mechanism to provide multiple views of the same process.

The OWL-S Composite Processes are decomposable into other processes. Their decomposition can be specified by using control constructs such as *Sequence* and *Repeat-While*. A composite process is not a behavior a service will do, but a behavior the client can

Figure 2.6: OWL-S profile model (Martin et al., 2004b)

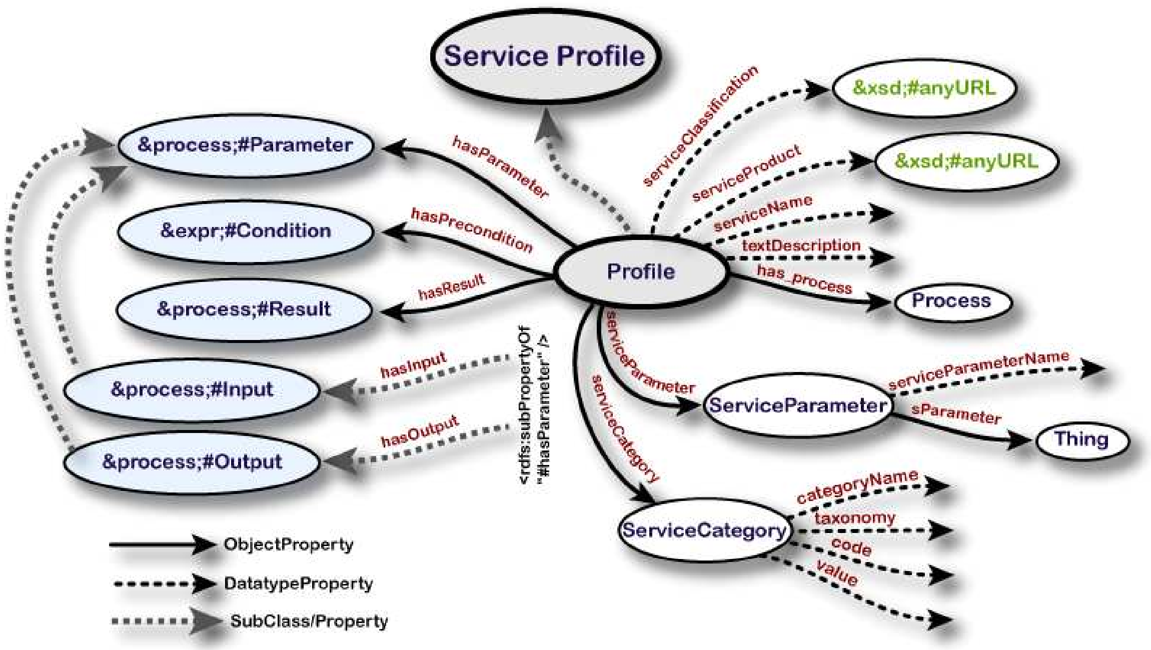
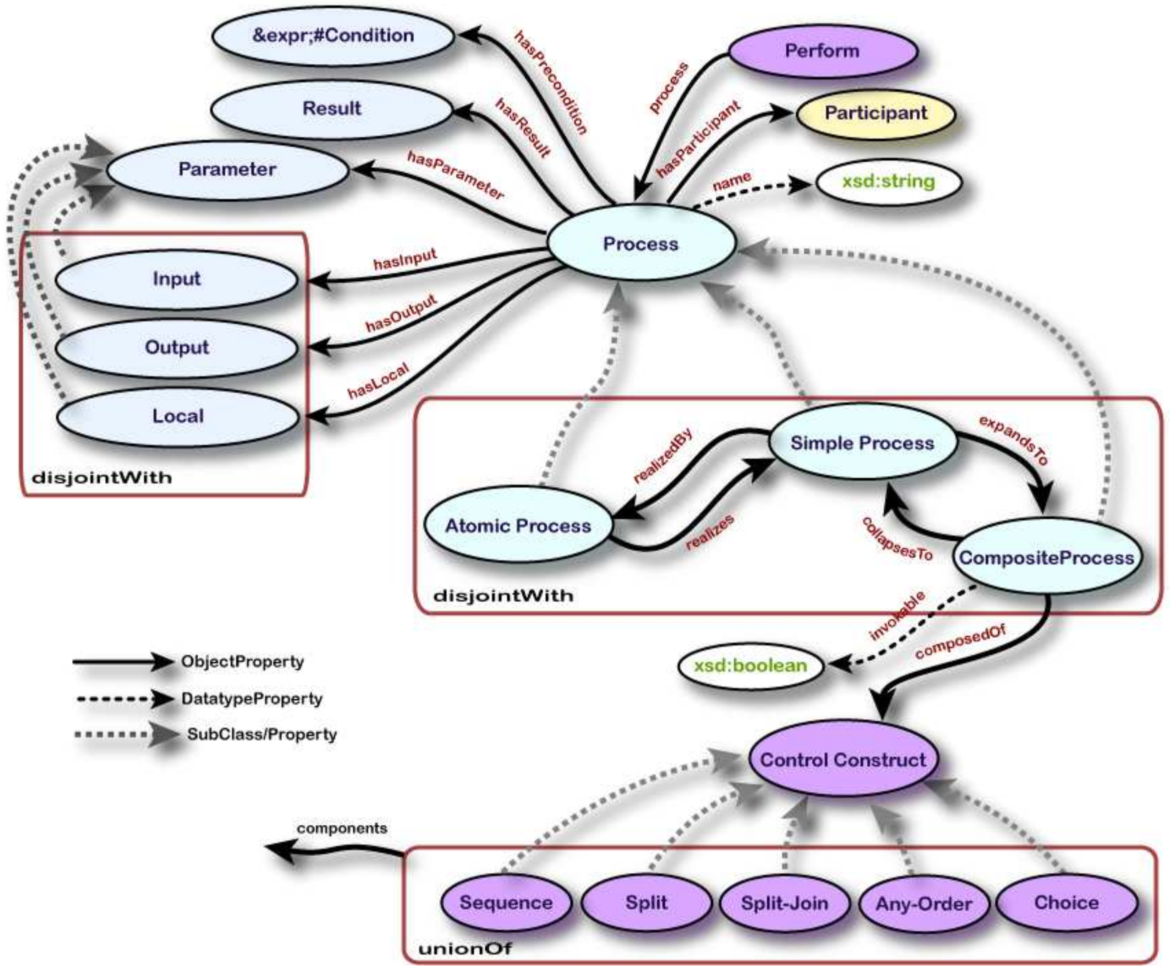


Figure 2.7: OWL-S process model (Martin et al., 2004b)



perform by interacting with the service. The OWL-S Composite Process supports the following control constructs (Martin et al., 2004b):

Sequence A list of control constructs to be done in order.

Split A list of control constructs to be done in parallel. Split completes as soon as all of its component processes have been scheduled for execution.

Split+Join A list of control constructs to be done in parallel. Split+Join completes when all of its components processes have completed.

Choice picks the execution of a single control construct from a given bag of control constructs.

Any-Order allows the process components to be executed in some unspecified order, but not concurrently.

If-Then-Else tests for the condition, executes the first control constructs if the condition is true, or the second otherwise.

Iterate repeatedly executes the same control construct.

Repeat-While tests for the condition, exits if it is false and does the operation if the condition is true, then loops.

Repeat-Until Repeat-Until does the operation, tests for the condition, exits if it is true, and otherwise loops.

2.4.1.3 Service Grounding

The OWL-S Service Grounding describes how to interact with the service. It specifies communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service. In addition, the grounding must specify, for each semantic type of input or output specified in the Service Model, a way of exchanging parameters of that type with the service (Martin et al., 2004b).

OWL-S is a mature technology with a decent software support. For example, the OWL-S Java API library provides programmatic access to read, execute and write OWL-S service descriptions (Sirin and Parsia, 2004).

2.4.2 WSMO

Web Service Modeling Ontology (WSMO) (de Bruijn et al., 2005) is based on the concepts from the Web Service Modeling Framework (WSMF) (Fensel and Bussler, 2002).

It defines four major components (Lara et al., 2005):

Ontologies provide the terminology and formal semantics for describing the other elements in WSMO. They are used to specify conceptual real-world semantics defined and agreed upon by communities of users.

Goals provide the means to specify the service consumer objectives when consulting a Web Service, describing at a high-level a concrete task to be achieved. They consist of non-functional properties , imported ontologies, mediators used, postconditions and effects. Postconditions and effects describe the state of the information space and the environment, desired by the service consumer.

Web Services provide a semantic description of Web services, including their functional and non-functional properties. The main elements of a Web service descriptions are the capability and the interfaces:

- **Capability** defines the functional aspects of the provided service. It is comprised of preconditions, assumptions, postconditions and effects. Capabilities are defined separately from the requester goals, clearly distinguishing between the requester and provider points of view. The preconditions describe the valid state of the information space prior to the service execution. Postconditions describe the guaranteed state of the information space after the service execution. Analogously assumptions describe the valid state of the environment prior to the service execution, and effects describe the guaranteed state of the environment after the service execution.
- **Interfaces** provide details about Web service's operation in terms of its choreography and its orchestration.

Mediators represent connectors that resolve heterogeneity problems in order to enable interoperability between heterogeneous parties.

WSMO provides expressive constructs that do not have a corresponding counterparts in OWL-S, for example, Goal construct used to describe service consumer objectives, or mediators. In spite of all that the design presented here is based on OWL-S, since WSMO does not have the adequate software support at the moment.

2.4.3 Jini

The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Jini moves data and executables via a Java object over a network. Its protocols are independent of the underlying networking technology (Sun Microsystems, 2006a).

2.4.3.1 Service Advertisement

A *djinn* is the group of devices, resources, and users that are joined by the Jini technology infrastructure. The Jini *lookup service* is a fundamental part of the Jini architecture that provides a central registry of services available within the djinn. It us used by the clients to find services within the djinn. The lookup service maintains a flat collection of service items (Sun Microsystems, 2006b).

Service advertisement functionality is provided by discovery and join protocols.

Entities that wish to start participating in a distributed system of Jini enabled services and devices must first obtain references to one or more Jini lookup services. The protocols that govern the acquisition of these references are known as the discovery protocols (Sun Microsystems, 2006b).

The discovery process involves three related protocols (Sun Microsystems, 2006b):

- **Multicast request protocol** is used to discover nearby lookup services. This protocol is employed by entities that are starting up and need to locate a djinn in proximity.
- **Multicast announcement protocol** is used to announce the existence of a lookup service on a local network. When a new lookup service is started, or when an existing service recovers after the failure, it might need to announce its availability to potential clients.
- **Unicast discovery protocol** is used to establish communications over a local area or wide area network with a lookup service whose address is known in advance. This protocol is used to establish connections with remote lookup services, or to deal with specific lookup services over a long period of time.

Once the lookup service has been located, a number of steps described by the join protocol must be taken for entities to start communicating usefully with these services (Sun Microsystems, 2006b).

2.4.4 JXTA

The JXTA (Sun Microsystems, 2006c) is a framework providing set of open protocols that support ad-hoc, pervasive, peer-to-peer computing. These protocols enable peers to collaborate irrespective of the network topology and their position in the network. The JXTA protocols are language independent, platform independent, and secure.

While Jini is more concerned with services located on a particular network, JXTA is used to communicate with a software services that are not location specific.

JXTA framework is comprised of three layers similar to the standard structure of operating systems:

1. **JXTA Core** provides the core functionality.
2. **JXTA Services** are the second layer built on top of the JXTA Core layer. Services provide the access to the JXTA protocols.
3. **JXTA Applications** form the third layer. The applications use JXTA Services layer to access the JXTA network.

2.4.4.1 Peers

JXTA peers create a virtual overlay network that hides the topology of underlying network. Peers can interact directly even when some of them are behind firewalls and

NATs or use different network transports. A peer is uniquely identified by a virtual address that allows him to change the location while keeping the same identity in the JXTA network (Traversat et al., 2003).

There are two main types of peers: edge-peers and super-peers. The super-peers are further divided into rendezvous and relay-peers. Edge-peers usually resides on the border of the Internet or behind the firewalls, and have a low bandwith connection to the network. Rendezvous-peers have agreed to index other peer advertisements to facilitate the discovery of resources in a peergroup. Relay-peers allow the peers behind firewalls or NAT systems to access the JXTA network (Traversat et al., 2002, 2003).

The peers exchange messages through pipes. Pipes are virtual communication channels used by JXTA to exchange messages. Pipes are asynchronous, unreliable and unidirectional (Traversat et al., 2003).

The peers self-organize into peergroups that represents a dynamic set of peers that share a common set of interests (Traversat et al., 2003).

2.4.4.2 Advertisements

All JXTA resources are represented by advertisements encoded as XML documents. JXTA standardizes advertisements for the core JXTA resources (for example, peer, peergroup, pipe, service, rendezvous). These advertisements can be subtyped to provide advertisements for other application specific resources. There are no limitations and advertisements can describe virtually anything (for example, Java objects, Web services) (Traversat et al., 2003).

The JXTA network acts as an always-available, network-wide, dynamic, distributed virtual hashtable that contains the index of all published advertisements. A peer can query the hash table at any time by supplying a set of attributes - the keys in the hash table. The query is resolved by the rendezvous network by hashing the key to the rendezvous peer containing the requested advertisement (Li, 2003).

JXTA uses the so called loosely-consistent DHT walker approach to search for advertisements in the JXTA rendezvous network. The loosely-consistent DHT walker uses a hybrid approach that combines the use of a DHT to index and locate advertisements, with a limited range walker to resolve inconsistency of the DHT within the dynamic rendezvous network. This approach is very robust, since it does not require maintaining consistency across the network, or a stable rendezvous peer infrastructure. It is is well adapted to ad hoc peer-to-peer network with high peer churn rate (Traversat et al., 2002).

2.4.4.3 Semantic Search

Zhou et al. (2003) pointed out that a DHT requires a unique hash techniques that transform the search criteria into a unique key set. Such approach is not suitable for semantic search mechanisms, since a typical semantic search consists of an arbitrary combination of concepts and the relationships between them. A semantic search technique should be able to search on a set of related entities rather than a single hash expression (Zhou et al., 2003).

2.5 Content Based Networking

An event notification middleware is a loosely coupled system, that provides notification delivery and notification selection services (Carzaniga et al., 2001). The clients can publish notifications and subscribe for notifications. The event notification middleware filter events a few well-known attributes of a notifications are available for selection to subscriptions (Carzaniga and Wolf, 2002). Event notification service is a standard component of current commercial messaging middlewares.

Content-based networking (CBN) is an event notification middleware that filters events based on matching client subscriptions to the full message type rather than message attributes. It facilitates still looser coupling between producers and consumers (Keeney et al., 2006b). In content-based networking, receivers subscribe to notifications that are of interest to them without regard to any specific source, while senders simply publish notifications without addressing it to any specific destination (Carzaniga and Wolf, 2002). Several CBN solutions and prototypes exist, for example Siena (Carzaniga et al., 2001), Elvin (Segall et al., 2000) and Hermes (Pietzuch and Bacon, 2002).

A limited support for CBN is provided by some Enterprise Service Bus products. Some products reviewed, for example ServiceMix (Apache Software Foundation, 2006b) and Mule (Codehaus, 2006), do support content-based routing, however routing rules must be statically encoded into their policies. They do not support dynamic subscriptions, which are one of the primary benefits of more advanced CBN.

2.5.1 Siena

Siena is an implementation of CBN middleware. A Siena notification is a set of typed attributes. Each attribute is comprised of a name, a type and a value. Original version of Siena supports only basic scalar types. A Siena subscription is a conjunction of filtering constraints. A constraint is comprised of the attribute name, an operator, and a value. A subscription covers a notification, if the even match to all filtering constraints of a filter. A notification is delivered to a client, if client has has submitted a subscription filter that covers that notification. Siena also discovers coverings between filters to optimize routing of the notifications. A filter covers another filter, if all notifications selected by the latter are also selected by the former (Keeney et al., 2006b).

Carzaniga et al. (2001) defined three based types of Siena topology: hierarchical client/server, acyclic peer-to-peer, and general peer-to-peer. All topologies provide the same functionality, however they differ in non-functional features, like time complexity, scalability and fault tolerance.

2.5.2 Ontologically Extended Siena

Even more flexible event-based system is a CBN based on messages containing semantic markup and queries. Such a semantic-based content-based networking is termed a *knowledge-based networking* (Keeney et al., 2006b). A partial knowledge-based networking implementation based upon Siena was conceived by Lynch et al. (2006) and developed by Keeney et al. (2006b). They extended Siena with support for three new ontological operators for strings containing ontological URIs: *subsumption*, *reverse subsumption*, and *equivalence*.

Chapter 3

Design

This chapter provides an overview of the main concepts and salient features of the design developed in scope of this research.

3.1 Requirements

The aim of this project is the design and implementation of a decentralized peer-to-peer workflow model using ontology-based descriptions of Web services. We consider two main issues, namely (i) the benefits of content-based networking to provide decentralized service discovery and loose coupling of decentralized data and control flow binding, and (ii) the usage of ontology based-descriptions of Web services.

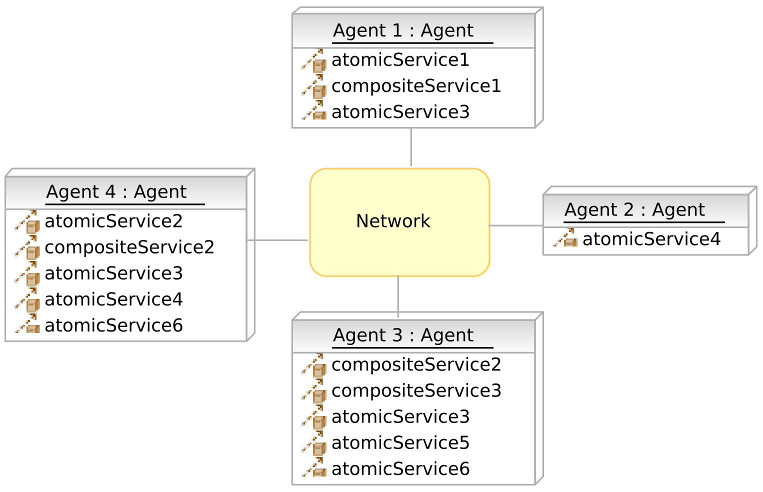
The ultimate goal is to produce loosely coupled distributed workflow system, where connections between participating services are made and remade dynamically based on inexact matches, thus promoting flexibility and robustness.

The main functional requirements for the system are the following:

- The system should be comprised of distributed software components (hereinafter referred to as agents) cooperating in a decentralized peer-to-peer fashion. The agents should be loosely coupled.
- Each agent should serve as the service container and is able to host atomic and composite semantic Web services.
- Agents are should be able to advertise hosted semantic services and discover services advertised by other agents. The discovery algorithm should work well in a dynamic environment, where services continually appear and disappear and where agents do not know other agents.
- Agents should be able to autonomously perform execution of hosted atomic Web services.

- Agents should be able to conjointly perform decentralized execution of composite Web services. The bindings between participating services are made and remade dynamically based on inexact semantic matches.

Figure 3.1: Agents with deployed services



The Figure 3.1 shows a set of agents with deployed services.

3.2 Outline

As mentioned, is the objective of this work is to support decentralized execution of a workflow described in terms of semantic Web services. Additionally it should support decentralized advertisement and discovery of available services. OWL-S (Martin et al., 2004b), respectively its CompositeProcess, will be used as a formal language for encoding the workflow process. While there are several other languages available, OWL-S is currently the most mature. While WSMO might be a better choice it has currently inadequate software support, which is needed for the implementation of the prototype.

Advertisement and discovery information must be efficiently broadcasted between peers. The content-based networking middleware has been chosen for this purpose. The motivation for this is given in § 3.3 and the design of the powerful extension of content-based networking operator set is explained in § 3.4. The design is based on Siena developed by Carzaniga et al. (2001) with ontological extension conceived by Lynch et al. (2006) and developed by Keeney et al. (2006b). It is currently the only available functioning content-based networking middleware middleware with support for ontological operators. As an alternative, enterprise service buses supporting content-based routing have also been considered, but the products currently available (for example, Apache ServiceMix and Codehaus Mule) do not have support for dynamic subscriptions, which is one of the primary requirements of this design.

An OWL-S CompositeProcess statically binds to the participating Web services. This is not suitable for our scenario, where bindings between participating services will be made dynamically and short-lived. Hence it is necessary to describe participating services in abstract terms of required capabilities, without assuming any concrete service implementation. For this purpose the AbstractProcess construct is introduced in § 3.5.1. We also need an effective mechanism for flexible, but accurate description of advertised capabilities, and discovery goals. As a solution to this requirement the design of discovery annotations is presented in § 3.5.2. Abstract processes and discovery annotations are integrated into a flexible peer-to-peer advertisement and discovery mechanism presented in § 3.5.3.

The decentralized execution of a composite service is orchestrated by several cooperating peer agents. Each agent is actually a service container that can host any number of services. The decentralized orchestration has two complementary parts, namely control-flow that defines the sequencing of activities, and data-flow that defines how information flows between activities. The design of the distributed control and data-flow is presented in § 3.6.

These concepts are combined in § 3.7 to define the architecture of the agent, which is the essential component of our decentralized composite service execution model. The agent can host, advertise and execute semantic Web services and participate in decentralized execution of composite semantic Web services.

Finally a realization of decentralized execution is explained in § 3.8.

3.3 Applicability of Content-Based Networking

This section investigates the possible usages of the content-based networking in a distributed workflow model. The distributed peer-to-peer workflow model presented in this research is comprised of (i) the service discovery model and (ii) the workflow execution model.

We will characterize the interaction patterns that occur in those two models with regard to the classification suggested by Buchmann et al. (2004). The study characterizes interaction patterns by two dimensions based upon: (i) who is the initiator of the interaction, (ii) whether the initiator has the knowledge of the counterpart (see Table 3.1).

Table 3.1: Categorization of the interactions patterns

		Initiator of Interaction	
		Consumer	Producer
Knowledge of Counterpart	Yes	Request/Reply	Messaging
	No	Anonymous Request/Reply	Event-based

3.3.1 Distributed Service Discovery

The distributed service discovery model is comprised of peers, discovering suitable services hosted by other peers. In that sort of interaction, the ultimate counterpart of any peer is another peer hosting a matching service. During the service discovery the service requester has no direct knowledge of the peers hosting the services to be discovered. According to Buchmann et al. (2004) interactions of this type belong to either *Anonymous Request/Reply* or *Event-based* category. The service requester can either query for matching services, which fits into the first category, or receive notifications for matching services, which fits into the second category.

The event-based systems are particularly well suited for accommodating communities of cooperating distributed parties that establish communication relationships dynamically over time in an unpredictable fashion (Meier and Cahill, 2005). The traditional event based systems match client subscriptions to the message type. A special type of an event-based system is the content-based network. Content-based networking facilitates still looser coupling between producer and consumer by matching client subscriptions to message attributes rather than the full message type. Indeed, already Carzaniga and Wolf (2002) envisioned the usage of content-based networking for the service discovery. Even more flexible event-based system is a content-based networking based on messages containing semantic markup and queries (Keeney et al., 2006b).

For these reasons the design presented here makes use of CBN with ontological extension for service advertisement and discovery. The CBN with ontological extension provides an efficient mechanism for delivering knowledge, in this case about available services, from its source to consumers, who register specific interests. It offers the potential for scaling delivery to Internet dimensions (Lewis et al., 2005).

There has been some research done in this area already. For example, a semantic service discovery system based on content-based networking has been developed by Lynch (2005) and forms part of the motivation for this work.

3.3.2 Distributed Workflow Execution

In the distributed workflow execution model there are two types of interactions between peers: control-flow and data-flow. The control-flow defines the sequencing of participating services. The data-flow defines how information flows between services. In both cases the initiator must know the identity of the counterpart. This is especially true in more complex workflows, where for example:

- The same peer hosting a particular activity has to be revisited several times during the distributed workflow execution, for example while looping through the sequence.
- The distributed workflow execution splits into some parallel sub-flows, which later join together. The peers hosting final activities of all these parallel sub-flows must

have an identical understanding which peer will execute the activity that occurs after the join.

This makes the content-based networking unsuitable for driving control or data-flow, since in the content-based networking interacting parties do not directly know each other, but are bound by routing brokers based on the content of the messages. An appropriate solution would be either *remote procedure call* or *messaging* middleware. More thorough investigation of this issue is out of the scope of this research. Hence a very simple solution based on standard non-semantic Web services is used in the implementation of the prototype.

3.4 Bag Extension for Content-Based Networking

In scope of this research a novel service discovery model based on ontologically extended Siena is presented as will be seen in § 3.5. As distinguished from Lynch (2005), the approach presented here is very general and not tied to discovery of semantic Web services. It is based on a new powerful family of matching operators, which can be applied to discovery of diverse types of ontologically described resources. Actually the discovery of semantic Web services serves merely as one example of its possible usages.

In the following sections, we will first look at how Siena has been extended with support for bags and bag operators. In the subsequent sections we will see how this extension has been exploited to build a robust distributed peer-to-peer service discovery system particularly suited to dynamic environments.

The design presented here extends Siena (Carzaniga et al., 2001) with a bag type. It also extends the Siena subscription language with the bag operators.

3.4.1 Bag Algebra

This section presents some formal definitions of the bag algebra which are needed to understand the Siena bag extension. It also defines and describes the composite bag relations, which are a novel contribution of this research to the field of bag algebra.

3.4.1.1 Bag

According to Weisstein (2002), a bag (also called multiset) is a set-like object in which order is ignored, but multiplicity is explicitly significant. Therefore, bags $[1, 2, 3]$ and $[2, 1, 3]$ are equivalent, but $[1, 1, 2, 3]$ and $[1, 2, 3]$ differ. Bag differs from a set in that each member has a multiplicity, which is a natural number indicating how many times it is a member.

For the purpose of the subsequent definitions we will use a more formal definition of a bag of Baeten and Basten (2001):

Definition 3.4.1. Suppose there exists some set A. A bag over set A is a function from A to the natural numbers[1] $\mathbb{N}$ such that only a finite number of elements from A is assigned a non-zero function value. For some bag P over set A and $a \in A$, $P(a)$ denotes the number of occurrences of a in P, often called the cardinality of a in P. A bag is a set if the cardinality of every element is one.

The function P is a set of ordered pairs $\{(a, P(a)) : a \in A\}$. For example, the bag written as $[a, a, b, a]$ is defined as $\{(a, 3), (b, 1)\}$, and the bag $[a, b]$ is defined as $\{(a, 1), (b, 1)\}$.

3.4.1.2 Finite Sequence

While finite sequences do not directly appear in the implementation of the Siena extension, they are needed in the subsequent sections for the formal definition of the composite bag relation.

Definition 3.4.2. Suppose there exists some set A. A finite sequence of length $n \in \mathbb{N}$ over set A is a function from set $M_n := \{m \in \mathbb{N} : m < n\}$ to A. For some finite sequence X over set A and $i \in M_n$, $X(i)$ is the i-th element of the sequence, often denoted as x_i.

The function X is a set of ordered pairs $\{(i, S(i)) : i \in M_n\}$. For example, the sequence written as $\langle a, a, b, a\rangle$ is defined as $\{(0, a), (1, a), (2, b), (3, a)\}$.

Definition 3.4.3. Suppose there exist some sets A and B, some function $f : A \to B$, and a set $C \subset B$. The inverse image of the set C under f is the subset of A defined by $f^{-1}[C] := \{a \in A : f(a) \in C\}$.

Definition 3.4.4. For any sequence X over set A, $\tau(X)$ denotes a bag of all elements from X, defined as $\tau(X) : A \to \mathbb{N}$, so that for all $a \in A$, $\tau(X)(a) := |X^{-1}[\{a\}]|$. That is, for any $a \in A$ the cardinality of inverse image of the singleton $\{a\}$ under the sequence X equals to the cardinality of the element a in the bag $\tau(X)$.

For example, $\tau(\langle a, a, b, a\rangle) = [a, a, a, b]$.

3.4.1.3 Simple Bag Relations

This section provides definitions of the three well-known binary bag relations: equal, subbag, and superbag. We will call these bag relations simple bag relations to distinguish them from composite bag relations defined in the next section.

Definition 3.4.5. Suppose there exist some set A and bags P and Q over A. P is equal to Q, denoted $P = Q$, if and only if for all $a \in A$, $P(a) = Q(a)$. P is subbag of Q, denoted $P \sqsubseteq Q$, if and only if for all $a \in A$, $P(a) \leqslant Q(a)$. P is superbag of Q, denoted $P \sqsupseteq Q$, if and only if $Q \sqsubseteq P$.

[1] It is assumed that natural numbers are non-negative integers $(0, 1, 2, 3, 4, \ldots)$, that is, $0 \in \mathbb{N}$.

[2] For some collection S, $|S|$ denotes the cardinality, or the number of elements, of the collection S.

Example 3.4.1. Here are some examples of true statements using the subbag relation[3]:

$$\varnothing \subseteq \varnothing$$
$$[a] \subseteq [a]$$
$$\varnothing \subseteq [a, b, c, d, e, f]$$
$$[b, c, d] \subseteq [a, b, c, d, e]$$
$$[e, a, e] \subseteq [a, b, c, d, e, e, e]$$

The following statements, however, are not true:

$$[a] \subseteq \varnothing$$
$$[a, a] \subseteq [a]$$
$$[k] \subseteq [a, b, c, d, e, f]$$
$$[b, c, d, c] \subseteq [a, b, c, d, e]$$
$$[a, a, e, e] \subseteq [a, b, c, d, e, e]$$

Corollary. *All three simple bag relations, namely equal, subbag, and superbag, are transitive and reflexive.*

The transitivity of simple bag relations follows from the transitivity of numerical *equal* and *less then* relations that were used in the definition of the simple bag relations.

3.4.1.4 Composite Bag Relations

This section defines the composite binary bag relation. The composite bag relation is a binary relation over bags composed of (i) another binary bag relation over bags and of (ii) a sub-relation over the bag elements. The motivation for it is explained in § 3.4.4 and § 3.5. The composite bag relation is one of the primary contributions of this research.

Definition 3.4.6. Suppose there exist some set A and some bags P and Q over A. Suppose Φ is a simple binary relation over bags, and λ is an arbitrary binary relation over the elements of A. **Bag P is Φ-related to bag Q with regard to sub-relation** λ, written as $P \, \Phi_\lambda \, Q$, if and only if there exist finite sequences X and Y over set A, so that all of the following statements are true:

1. P is Φ-related to $\tau(X)$ (P is Φ-related to the bag of all elements from sequence X),
2. $\tau(Y)$ is Φ-related to Q (the bag of all elements from sequence Y is Φ-related to Q),
3. $|X| = |Y|$ (sequences X and Y have the same number of elements),

[3] The symbol $\varnothing$ denotes an empty set.

4. for all $i \in \mathbb{N}$, $i < |X|$, $X(i)$ is λ-related to $Y(i)$.

We call relation Φ primary relation of composite bag relation, and relation λ sub-relation of composite bag relation.

Remark. The above definition could be without any changes generalized so that any transitive binary bag relation would be used in place of the simple binary bag relation as the primary relation. But this generalization is not needed in the scope of this research.

In the continuation some examples of composite bag relations over bags of integers are presented. Let's define a few example bags first:

$$\begin{aligned} P &:= [0, 1, 5, 7] \\ Q &:= [1, 2, 8, 8] \\ R &:= [0, 2, 8, 10] \\ S &:= [9, 9, 9, 9] \\ T &:= [9, 9, 9, 9, 9] \\ U &:= [0, 0, 0, 1, 9, 9, 9, 9] \end{aligned}$$

Example 3.4.2. Let's take a composite bag relation $=_<$, which is composed of a primary bag relation *equal* and a sub-relation *less than*.

The following statement is true:

$$P =_< Q$$

The statement is true because there exist sequences $X = \langle 0, 1, 5, 7\rangle$ and $Y = \langle 1, 2, 8, 8\rangle$, so that $P = \tau(X)$ and $\tau(Y) = Q$, and for $i \in \{0, 1, 2, 3\}$, $X(i) < Y(i)$. Indeed: $0 < 1$, $1 < 2$, $5 < 8$, and $7 < 8$.

The following statements, however, are both false:

$$\begin{aligned} P &=_< R \\ Q &=_< P \end{aligned}$$

The statements are false because sequences which would satisfy the requirements from the definition of the composite bag operator do not exist.

Example 3.4.3. Let's take a composite bag relation $\subseteq_\leqslant$, which is composed of a primary bag relation *subbag* and a sub-relation *less than or equal to.*

The following statement is true:

$$P \subseteq_\leqslant Q$$

The statement is true because there exist sequences $X = \langle 0, 1, 5, 7\rangle$ and $Y = \langle 1, 2, 8, 8\rangle$, so that $P \subseteq \tau(X)$ and $\tau(Y) \subseteq Q$, and for $i \in \{0, 1, 2, 3\}$, $X(i) \leqslant Y(i)$.

The following statement is also true:

$$P \subseteq_{\leqslant} U$$

The statement is true because there exist sequences $X = \langle 0, 1, 5, 7\rangle$ and $Y = \langle 0, 1, 9, 9\rangle$, so that $P \subseteq \tau(X)$ and $\tau(Y) \subseteq U$, and for $i \in \{0, 1, 2, 3\}$, $X(i) \leqslant Y(i)$.

The following statement is also true:

$$Q \subseteq_{\leqslant} U$$

The statement is true because there exist sequences $X = \langle 0, 2, 8, 8\rangle$ and $Y = \langle 0, 9, 9, 9\rangle$, so that $Q \subseteq \tau(X)$ and $\tau(Y) \subseteq U$, and for $i \in \{0, 1, 2, 3\}$, $X(i) \leqslant Y(i)$.

The following statement is also true:

$$S \subseteq_{\leqslant} U$$

The statement is true because there exist sequences $X = \langle 9, 9, 9, 9\rangle$ and $Y = \langle 9, 9, 9, 9\rangle$, so that $S \subseteq \tau(X)$ and $\tau(Y) \subseteq U$, and for $i \in \{0, 1, 2, 3\}$, $X(i) \leqslant Y(i)$.

The following statements, however, are all false:

$$Q \subseteq_{\leqslant} R$$
$$R \subseteq_{\leqslant} U$$
$$T \subseteq_{\leqslant} U$$

The statements are false because sequences which would satisfy the requirements from the definition of the composite bag operator do not exist.

Example 3.4.4. Let's take a composite bag relation $\supseteq_{\leqslant}$, which is composed of a primary bag relation *superbag* and a sub-relation *less than or equal to.*

The following statement is true:

$$U \supseteq_{\leqslant} Q$$

The statement is true because there exist sequences $X = \langle 0, 0, 0, 1\rangle$ and $Y = \langle 1, 2, 8, 8\rangle$, so that $U \supseteq \tau(X)$ and $\tau(Y) \supseteq Q$, and for $i \in \{0, 2, 3, 4\}$, $X(i) \leqslant Y(i)$.

Remark. Note that the following statements are not mutually exclusive:

$$Q \subseteq_{\leqslant} U$$
$$Q \subseteq_{\geqslant} U$$

Remark. The definition of the composite bag relation is recursive. If a composite bag relation operates on bags of bags, then its sub-relation can be itself a composite bag relation.

Example 3.4.5. Let's take a composite bag relation $\subseteq_{(\subseteq_{<})}$, which is composed of a primary bag relation *subbag* and a sub-relation $\subseteq_<$, which is itself a composite bag relation composed of a primary bag relation *subbag* and a sub-relation *less than.*

Let's define two bags of bags of integers:

$$V = [\varnothing, [0,0], [1,2,3,4]]$$
$$W = [[8], [0,0], [1,1,1], [1,2,3,4,5]]$$

Then the following statement is true:

$$V \subseteq_{(\subseteq_<)} W$$

The statement is true because there exist sequences $X = \langle \varnothing, [0,0], [1,2,3,4] \rangle$ and $Y = \langle [8], [1,1,1], [1,2,3,4,5] \rangle$, so that $V \subseteq \tau(X)$ and $\tau(Y) \subseteq W$, and for $i \in \{0,1,2\}$, $X(i) \subseteq_< Y(i)$.

The following statement, however, is false:

$$V \subseteq_{(\supseteq_<)} W$$

Theorem 3.4.1. *Let Φ be a simple binary relation over bags, and λ a binary relation over bag elements. If λ is transitive, then Φ_λ is also transitive. If λ is reflexive, then Φ_λ is also reflexive.*

The proof of the above theorem is simple, however, it is out of the scope of this work.

Corollary. *As an interesting observation, a composite bag relation $\subseteq_=$ is equivalent to a simple bag relation $\subseteq$. More generally, any bag relation Φ is equivalent to $\Phi_=$, denoted as $\Phi \equiv \Phi_=$.*

Theorem 3.4.2. *Another interesting observation is that if $A \subseteq_> B$, then $B \supseteq_< A$. More generally, if Φ is a simple binary relation over bags, λ is a binary relation over bag elements, Φ^{-1} is the inverse relation of Φ, and λ^{-1} is the inverse relation of λ, then $P \, \Phi_\lambda \, Q$ exactly when $Q \, \Phi^{-1}_{\lambda^{-1}} \, P$.*

In the past section bag algebra has been described to the extent needed by this project. In the following sections we will look at how Siena has been extended with support for bags.

3.4.2 Extending Siena with Bags

This section describes the design of the bag support for the content-based event notification service, which is one of the major contributions of this research. The design is based upon the hierarchical version of Siena for Java (Carzaniga et al., 2001) with the ontological extension (Keeney et al., 2005). The bag type and the bag operators greatly extend the expresiveness of the Siena subscription mechanism, especially when combined with the ontological operators. Examples of its usage are provided in § 3.4.4 and § 3.5.

3.4.2.1 Siena Bag Type

An event notification in Siena is a set of typed attributes. Each typed attribute is comprised of a type, a name and a value. The current version of Siena supports the following types: *string*, *long*, *integer*, *double* and *boolean*. This project extends Siena with the *bag* type. The semantics of the bag type corresponds to the bag definition from § 3.4.1.1.

A bag value can contain any valid Siena values, including other bag values. A Siena bag is not allowed to contain itself, either directly or indirectly via other bags. Elements of a bag do not need to be of a uniform type. Bags in Siena are first order members of the Siena type set. They can appear in notifications, as well as in subscription filters, like any other Siena type. Siena advertisements, which are part of the theoretical Siena model, are not supported in the hierarchical version of Siena, but should they be, bag type should work seamlessly with them.

Example 3.4.6. Here are some examples of valid Siena bags:

$$\begin{gathered}\varnothing \\ [3, 345, 27, 35, 3476, 0, 27, 27] \\ [\text{"Ljubljana"}, \text{"Vienna"}, \text{"Amsterdam"}, \text{"Dublin"}] \\ [\text{"Ljubljana"}, 2, \text{"Ljubljana"}, 3.14159]\end{gathered}$$

Example 3.4.7. The Figure 3.2 shows an example of extended Siena notification with a bag attribute.

Figure 3.2: Siena notification comprising a bag attribute

Attribute Name	*Type*	*Value*
serviceUri	string	"http://kdeg/BookHotel.owl#BookHotelService""
inputStereotypes	bag	"PartyName" "City" "BeginningDate" "EndDate" "HotelCategory"
validity	integer	300
agentUrl	string	"http://kyi:9080/dowls/"

Remark. Since a set is a bag where all elements have cardinality one, this extension also implicitly supports sets.

3.4.2.2 Siena Bag Operators

An subscription filter in Siena is used to subscribe to event notifications by specifying a set of attributes and constraints on the values of those attributes. Each constraint is comprised of a type, a name, a binary predicate operator, and a value for an attribute. The original version of the Java Siena CBN supports common equality and ordering relations for its types (Carzaniga et al., 2001). Keeney et al. (2006b) have extended Siena with support for three new ontological operators for strings containing ontological URIs: *subsumption*, *reverse subsumption*, and *equivalence*.

This work extends the set of supported operators further with simple and composite binary bags operators. The semantics of the simple bag operator corresponds to the simple binary bag relation defined in § 3.4.1.3, and the semantics of the composite bag operator corresponds to the composite binary bag relation defined in § 3.4.1.4. The primary bag operator and the sub-operator correspond to the primary bag relation and the sub-relation respectively.

Remark. Although there is no *bag membership* operator in Siena, it is very easy to simulate the *bag membership* operator with the *subbag* wrapping a in a singleton bag. Let A be an arbitrary set, P a bag over A, and $a \in A$. $a \in P$ if and only if $\{a\} \subseteq P$.

Remark. Although elements of a bag do not need to be of a uniform type, this is not particularly useful when used in combination with a composite bag operator. The reason is that its sub-operator will normally yield to false when comparing values of different types (for example, "$3 <$ **true**" is false).

3.4.2.3 Covering of Bag Operators

Siena optimizes its routing tables by aggregating event filters. The aggregation rules are derived from coverings between event filters. If the filter A always selects all events, which are selected by the filter B, then A covers B. In order to preserve correctness and efficiency of Siena, routing covering relationships between new Siena bag operators must be properly implemented.

Consider two filtering constraints A and B, such that A is given as $x\ \Phi\ P$, and B is given as $x\ \Phi\ Q$, where Φ is one of the $=$ (equal), $\subseteq$ (subbag), or $\supseteq$ (superbag) operator. P and Q are the bag values and the variable x is the attribute to be compared with P respectively Q. Covering relationships between simple bag operators are straightforward and presented in the Table 3.2.

Composite bag operators have much more complex semantics and for this reason covering relationships between them are harder to discover and more computationally intensive to evaluate. For this reason the following three observations have been taken into account during their construction:

1. It is important to note that the covering relationship is purely an optimization and that Siena would work correctly without it, albeit not as efficiently as with it

Table 3.2: Covering relationships between the simple bag operators

A	B	A covers B exactly when
$x = P$	$x = Q$	$Q = P$
$x \supseteq P$	$x = Q$	$Q \supseteq P$
$x \subseteq P$	$x = Q$	$Q \subseteq P$
$x = P$	$x \supseteq Q$	never
$x \supseteq P$	$x \supseteq Q$	$Q \supseteq P$
$x \subseteq P$	$x \supseteq Q$	never
$x = P$	$x \subseteq Q$	never
$x \supseteq P$	$x \subseteq Q$	never
$x \subseteq P$	$x \subseteq Q$	$Q \subseteq P$

(Heimbigner, 2003). Hence, the evaluation of covering relationship as *false*, when it is actually *true*, can affect the routing efficiency, but can not affect the routing accuracy. In other words, false negatives reduce routing efficiency, but do not affect routing accuracy.

2. Evaluation of covering relationship as *true*, when it is actually *false*, can affect routing accuracy. In other words, false positives do affect routing accuracy.
3. Absence of false negatives in the covering relationship algorithm would not necessarily result in the optimal routing performance. In a dynamic environment, where subscription filters have a short lifespan, more precise covering relationship algorithm could cause a high computational burden and result in bad overall routing performance.

Considering this, and with a goal to keep things simple, the covering relationships between composite bag operators, which allow for some false negatives, have been defined. They are presented in the Table 3.3. The correctness of these relationships is based upon transitivity rule for composite bag operators given in the Theorem 3.4.1. All symbols used in the table have the same meaning as in the simple bag operators case, with an addition of λ and ξ that represent sub-operators of composite bag operators.

3.4.3 Modifications of Ontologically Extended Siena

The composite bag operator makes the CBN very powerful, especially when combined with the ontological operators, as demonstrated in § 3.4.4 and § 3.5.

Keeney et al. (2006b) have extended original Siena with support for three new ontological operators: *subsumes*[4] ($\Vdash$), *subsumed by*[5] ($\dashv\vert$), and *equivalent* ($\equiv$). In order to use the ontologically extended Siena effectively in this project, two minor modifications

[4] An alternative name for operator *subsumes* is *less specific*.

[5] An alternative name for operator *subsumed by* is *more specific*.

Table 3.3: Covering relationships between the composite bag operators

A	B	A covers B when
$x =_\lambda P$	$x =_\xi Q$	$\lambda \equiv \xi \wedge \lambda$ is transitive $\wedge\ Q =_\lambda P$
$x \supseteq_\lambda P$	$x =_\xi Q$	$\lambda \equiv \xi \wedge \lambda$ is transitive $\wedge\ Q \supseteq_\lambda P$
$x \subseteq_\lambda P$	$x =_\xi Q$	$\lambda \equiv \xi \wedge \lambda$ is transitive $\wedge\ Q \subseteq_\lambda P$
$x =_\lambda P$	$x \supseteq_\xi Q$	never
$x \supseteq_\lambda P$	$x \supseteq_\xi Q$	$\lambda \equiv \xi \wedge \lambda$ is transitive $\wedge\ Q \supseteq_\lambda P$
$x \subseteq_\lambda P$	$x \supseteq_\xi Q$	never
$x =_\lambda P$	$x \subseteq_\xi Q$	never
$x \supseteq_\lambda P$	$x \subseteq_\xi Q$	never
$x \subseteq_\lambda P$	$x \subseteq_\xi Q$	$\lambda \equiv \xi \wedge \lambda$ is transitive $\wedge\ Q \subseteq_\lambda P$

must be applied to it.

The paper defines that the *subsumes* and *subsumed by* relationships between two *equivalent* ontological concepts do not hold. For the purposes of this project *subsumes* and *subsumed by* will be considered true for *equivalent* concepts. More formally, for any ontological resource A and any ontological concepts B, if $A \equiv B$, then $A \Vdash B$ and $A \dashv\!| B$.

The paper also states that filtering constraint $x \equiv A$ never covers filtering constraint $x \equiv B$. For the purposes of this project the covering relationships between filtering constraints are changed so that $x \equiv A$ covers $x \equiv B$ precisely when $A \equiv B$. In other words, two filtering constraints with *ontological equivalence* operators cover each other if and only if their respective values are *ontologically equivalent.*

It is important to note that the covering relationships of other Siena operators are not affected. Their descriptions are available in (Carzaniga et al., 2001) and (Rutherford, 2004), and remain completely unchanged.

3.4.4 Semantic Resource Discovery

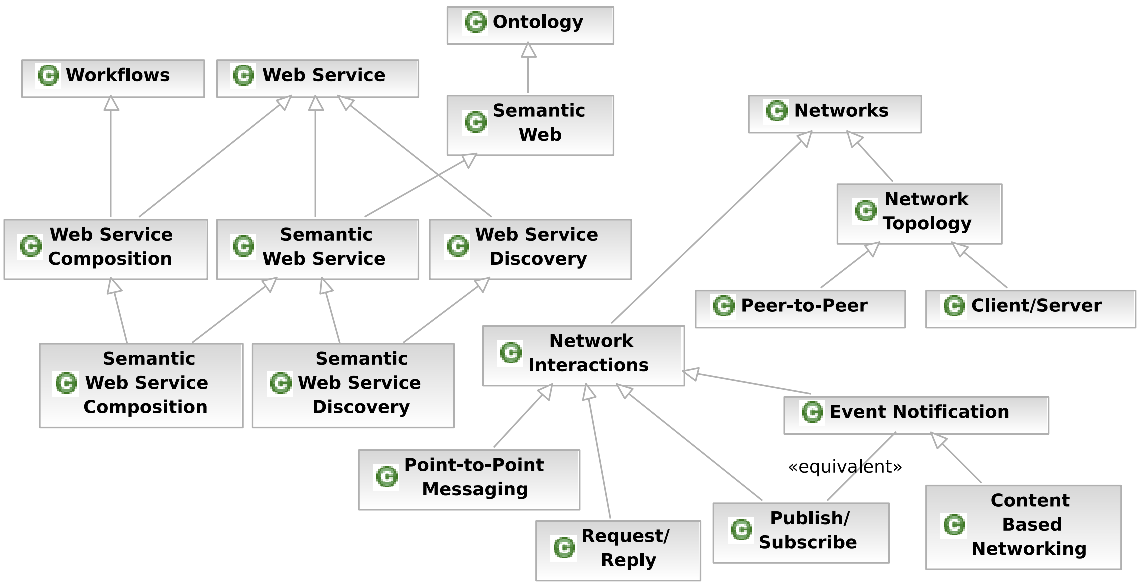
This section presents the applicability of bags and composite bags operators for discovery of ontologically described resources. § 3.5 specializes this general approach to semantic service discovery.

The filtering constraint in the ontologically extended Siena is capable of comparing a single ontological concept[6] from the constraint with a single ontological concept from the notification attribute. However, it does not provide any means for handling more than one concept within a single attribute. Let's illustrate this point using an example

[6] Ontological concept is either OWL class, OWL property, or OWL individual.

from the Figure 3.3. The figure shows an example of Keywords Taxonomy for tagging academic papers.

Figure 3.3: Keywords taxonomy



Example 3.4.8. Suppose that each academic paper is tagged with exactly one keyword. Suppose that notifications for new academic papers are published via ontologically extended Siena. Each notification contains an attribute *keyword* with the value of the keyword. Suppose also that there are Siena clients, which would like to subscribe for notifications about all academic papers that are tagged with the particular keyword, or any equivalent or more specific keyword. For example, if client would like to receive notifications about all academic papers, which are tagged with the keyword *Semantic Web Service*, or any equivalent or more specific keyword, the appropriate constraint could be expressed in extended Siena with the following statement:

$$\text{keyword} \dashv\mid \text{"Semantic Web Service"}$$

Example 3.4.9. Let us now consider a more realistic situation, where each academic paper is tagged with several keywords. Suppose that in this case Siena clients subscribe for academic papers that have at least some (one or more) required keywords, also allowing for equivalent or more specific keywords. It is not possible to express this constraint in a single subscription in ontologically extended Siena. However it is possible to express the constraint in ontologically extended Siena with the bag extension.

Each notification must contain an attribute *keywords* with the bag containing all the keywords of advertised academic paper. The Figures 3.4 and 3.5 show two examples.

Suppose there is a Siena client, which would like to subscribe for notifications about all academic papers, which keywords match *Ontology* and *Workflow*, also allowing for equivalent or more specific keywords. The following filtering constraint in ontologically

Figure 3.4: Example of the notification for Academic Paper 1

Attribute Name	*Type*	*Value*
title	string	”Peer-to-peer Semantic Workflow”
authors	string	”Micka Kovaceva and Tone Balone and Ziva Zver”
keywords	bag	”Semantic Web” ”Workflow” ”Peer-to-Peer”

Figure 3.5: Example of the notification for Academic Paper 2

Attribute Name	*Type*	*Value*
title	string	”Semantic Discovery of Composite Service Components”
authors	string	”Janez Kranjski and Micka Kovaceva”
keywords	bag	”Semantic Web Service Composition” ”Peer-to-Peer” ”Event Notification”

extended Siena with bag extension can be used to match the appropriate notifications:

$$(\text{keywords} \sqsupseteq_{\dashv\|} [\text{"Ontology"}]) \wedge (\text{keywords} \sqsupseteq_{\dashv\|} [\text{"Workflow"}])$$

The constraint requires that the notification contains at least one keyword, which is *subsumed by Ontology*, and at least one keyword, which is *subsumed by Workflow*. It allows for the same keyword to match both conditions. This constraint would correctly select both notifications from the Figures 3.4 and 3.5.

Example 3.4.10. Now let us look at the following constraint:

$$\text{keywords} \sqsupseteq_{\dashv\|} [\text{"Ontology"}, \text{"Peer-to-Peer"}]$$

Though the constraint might seem correct at a hasty glance, it would only match the notification from the Figure 3.4. The reason is that this constraint requires that notification contains at least one keyword, which is *subsumed by Ontology*, and at least *an other* keyword, which is *subsumed by Workflow*. This form of bag constraint is not useful in the example at hand, but it provides exactly what we will need for the service discovery in § 3.5.

3.5 Semantic Service Discovery

The general semantic resource discovery concept outlined in the previous section can be applied to semantic service discovery. Before we do so, constructs for describing the service provider’s capabilities and service consumer’s requirements must be available.

Hence we will first introduce two new OWL-S concepts, which are abstract processes and discovery annotations, and then use them to enable semantic service discovery driven by CBN middleware.

3.5.1 Abstraction of OWL-S Process Model

This section introduces abstract semantic Web service processes. The OWL-S overview (Martin et al., 2004b) describes the process as a detailed perspective on how to interact with a service. The design proposed here is based upon the template-based composition of semantic Web services presented by Sirin et al. (2005).

3.5.1.1 OWL-S Processes

OWL-S defines three types of processes: owls:AtomicProcess, owls:SimpleProcess and owls:CompositeProcess. While owls:AtomicProcess and owls:CompositeProcess are both executable, owls:SimpleProcess is used as an element of abstraction. According to the OWL-S specification owls:SimpleProcess may be used either to provide a specialized way of using some owls:AtomicProcess, or as a simplified representation of some owls:CompositeProcess. In the former case, owls:SimpleProcess is realized by the owls:AtomicProcess; in the latter case, owls:SimpleProcess expands to the owls:CompositeProcess. To sum up, owls:SimpleProcess is an abstraction of a specific executable process.

Composite processes statically bind to the participating processes. That is not suitable for our scenario, where bindings between participating processes will be made dynamically and are short-lived. Hence we must be able to describe participating processes in abstract terms of required capabilities, without assuming any concrete process implementation.

Pure abstract description should describe the process solely in terms of inputs, outputs, preconditions and effects. It should not be connected to any specific profile (and consequently it would not be grounded), and it should have no links to executable processes. Sirin et al. (2005) pointed out that there is no concept in OWL-S that would provide pure abstract descriptions of processes. The authors proposed the new OWL-S process type named AbstractProcess, however they dis not formally define it.

3.5.1.2 Abstract Process

Definition 3.5.1. For the purposes of this project we define a new OWL class called dowls:AbstractProcess. dowls:AbstractProcess is a subclass of the owls:SimpleProcess with additional restrictions that prevent assigning values to those inherited properties that should not occur in a pure abstract class. dowls:AbstractProcess instances cannot have assigned values for properties owls:describes, owls:hasParticipant, owls:realizedBy and owls:expandsTo. That leaves the dowls:AbstractProcess only with the properties to specify inputs, outputs, preconditions and effects. Therefore the dowls:AbstractProcess meets the requirements asserted above. The OWL definition is shown in the Listing 3.1.

Listing 3.1: OWL definition of dowls:AbstractProcess

```
<owl:Class rdf:ID="AbstractProcess">
  <rdfs:subClassOf rdf:resource="&process;#SimpleProcess"/>
</owl:Class>

<owl:Class rdf:about="#AbstractProcess">
    <rdfs:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
            <owl:onProperty rdf:resource="&service;#describes" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">0</owl:maxCardinality>
        </owl:Restriction>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&process;#hasParticipant" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">0</owl:maxCardinality>
        </owl:Restriction>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&process;#realizedBy" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">0</owl:maxCardinality>
        </owl:Restriction>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&process;#expandsTo" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">0</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:intersectionOf>
</owl:Class>
```

Example 3.5.1. The Listing 3.2 shows an instance of dowls:AbstractProcess with one input, one output, no preconditions and no effects.

Listing 3.2: Example of abstract process instance

```
<dowls:AbstractProcess rdf:ID="AbstractGetBookPriceProcess">
    <process:hasInput>
        <process:Input rdf:ID="bookName">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
        </process:Input>
    </process:hasInput>
    <process:hasOutput>
        <process:Output rdf:ID="bookPrice">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#integer
            </process:parameterType>
        </process:Output>
    </process:hasOutput>
</dowls:AbstractProcess>
```

Remark. The given definition of an abstract process is pragmatic and not very elegant. A better alternative would be to define abstract process either as a class, equivalent to the owls:Process class, or as a class on the same hierarchical level as the owls:AtomicProcess, owls:CompositeProcess and owls:SimpleProcess classes are defined. However, this formally correct solution would break the behavior of existing software tools for OWL-S, since some tools are only able to handle standard OWL processes. Our solution actually effectively tricks the tools by allowing them to treat the dowls:AbstractProcess as the owls:SimpleProcess.[7]

3.5.1.3 Composite Service Template

This section defines composite service templates. The conception is based upon the work of Sirin et al. (2005).

Composite processes organize other processes into a workflow process. Invocations of participating processes are indicated as instances of owls:Perform constructs. dowls:AbstractProcess can be used with the owls:Perform construct as an ordinary owls:Process. Having dowls:AbstractProcess it is now possible to construct a composite process without any reference to executable processes. We call such composite processes *composite process template*, and a service described by such a process *composite service template*. A composite process template is a generalized composite process where steps are defined as abstract processes. This enables dynamic selection of participating processes. The decision regarding which executable process will handle a given invocation can be postponed until composite service template execution.

Abstract processes represent the *component model dimension* (see § 2.2.3) of service composition model. An abstract process describes the contract that must be fulfilled by an executable service in order to be able to participate in workflow execution in lieu of the abstract process. Therefore we need a facility to describe requirements for suitable executable services. The next section will present discovery annotations that will characterize dowls:AbstractProcess for the purposes of discovery of executable services that are able to fulfill its contract.

3.5.2 OWL-S Discovery Annotations

This section outlines the design of discovery annotations. Discovery annotations are additional information elements in an OWL-S document that describe its characteristics for purposes such as advertisement, discovery, and selection.

3.5.2.1 OWL-S Service Profile

In OWL-S a owls:ServiceProfile is used to characterize a service for advertisement purposes. OWL-S does not prescribe or limit the ways in which profiles may be used, but rather, seeks to provide a basis for their construction that is flexible enough to accommodate many different contexts and methods of use (Martin et al., 2004a).

Several approaches about how to use the service profile have been suggested so far:

[7] For example, OWL-S Java API (Sirin and Parsia, 2004) recognizes dowls:AbstractProcess instances as instances of the owls:SimpleProcess.

1. OWL-S itself provides one possible representation of a service profile through the class owls:Profile (Martin et al., 2004b). The owls:Profile presents what the service does by specifying the input and output types, preconditions, and effects. It does not provide specific classes for modeling these characteristics, but uses the schema offered by the Process.owl ontology.

2. Martin et al. (2004a) suggest to construct a hierarchy of subclasses of the owls:Profile that would provide a means of service categorization. This approach has the following drawbacks: (i) The hierarchy of OWL-S profile classes is proprietary. It is specific to OWL-S and hence not simply reusable outside of OWL-S. (ii) Services can not be categorized by a third party taxonomy that is not rooted in the owls:ServiceProfile, at least not without special adaptation to the profile hierarchy.

3. A different approach mentioned by Sirin et al. (2004) bases the matching on the inheritance relationship of parameter types. For example, an input of type *City* of a provided service can be said to match an input of type *Place* of a requested service. Note that *City* and *Place* are data types. This proposition requires the construction of a data type hierarchy for the purposes of semantic selection. The main drawback of this approach is that it does not separate different concerns, namely the data type and the semantic meaning.

4. Another possible approach would be to construct a hierarchy of subclasses of owls:Input and owls:Output classes. This approach suffers from the same drawbacks as approach 2.

3.5.2.2 Stereotypes

In this section we provide an alternative approach very similar to that of Akkiraju et al. (2005). However, our design builds upon existing owls:Profile and dowls:AbstractProcess classes. Both classes describe the functional aspects of the provided, respectively required, services. But both fall short in describing semantic features needed for dynamic discovery. Our approach defines a new property dowls:hasStereotype, which is used to annotate owl:Profile, dowls:AbstractProcess, and their inputs and outputs, with concepts from some hierarchical taxonomy. The idea is based on the presumption that these semantic annotations characterize the service sufficiently well to enable automation of the process of service discovery.

Definition 3.5.2. *dowls:hasStereotype* is an OWL property that classifies the annotated OWL individual for the purposes of advertisements, discovery, or selection. The domain of *dowls:hasStereotype* property is the union of owls:Profile, dowls:AbstractProcess, and dowls:Parameter classes. The range of *dowls:hasStereotype* property is any URI. In other words, property *dowls:hasStereotype* is applicable to profiles, abstract processes, and their inputs and outputs, and its value must be an URI. The OWL definition is shown in the Listing 3.3.

Remark. The reason for specifying the range of stereotype as an URI rather then owl:Thing[8] is to enable the usage of third party business taxonomies, eventually being outside OWL-S and possibly outside OWL. In the latter case the discovery system would require some specialized reasoner for that taxonomy.

Listing 3.3: OWL definition of dowls:stereotype

```
<owl:DatatypeProperty rdf:ID="hasStereotype">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="&profile;#Profile"/>
          <owl:Class rdf:about="&dowls;#AbstractProcess"/>
          <owl:Class rdf:about="&process;#Parameter"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
    <rdfs:range rdf:resource="&xsd;anyURI" />
</owl:DatatypeProperty>

<owl:Class rdf:about="&profile;#Profile">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasStereotype" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&dowls;#AbstractProcess">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasStereotype" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&process;#Parameter">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasStereotype" />
            <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Remark. The solution presented here is similar to the one provided by *owls:serviceCategory*, *owls:serviceClassification* and *owls:serviceProduct* properties of the owls:Profile. However, our solution extends this approach further to abstract processes and parameters. It is not a fully developed solution and it serves basically as an illustration of the idea that could be easily extended further to preconditions and effects. Moreover, the annotation itself could be much further elaborated from being just a single property. For example, a more elaborated annotation of parameters in abstract processes could designate whether the parameter is mandatory or optional.

[8] Every OWL class is a subclass of owl:Thing.

Remark. These discovery annotations are unintrusive, because they can be applied to semantic OWL-S services defined elsewhere. That is because OWL (as well as RDF) allows to add additional properties to resources defined elsewhere.

To take advantage of stereotypes, they must be understood correctly by the discovery system. Here we present some simple examples that illustrate correct interpretation. The formal definition of matching rules is provided in the next section.

Example 3.5.2. Suppose there exists some input parameter to an OWL-S accommodation booking service named *numberOfStars* of type *integer* to express the quality of the accommodation. Suppose there exists a taxonomy, which categorizes tourism concepts into a hierarchy, and which contains concept *HotelCategory*. OWL-S provides no means to express that a service actually expects a hotel category as the value of parameter *numberOfStars*. The new property *dowls:hasStereotype* enables the formal expression of this requirement by assigning the concept *HotelCategory* to the parameter *numberOfStars*. That is, the stereotype of input parameter *numberOfStars* is *HotelCategory*.

Listing 3.4: Example of parameter stereotype instance

```
<process:hasInput>
    <process:Input rdf:ID="numberOfStars">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
            &xsd;#integer
        </process:parameterType>
        <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
            &dowls;#HotelCategory
        </dowls:hasStereotype>
    </process:Input>
</process:hasInput>
```

Example 3.5.3. Suppose there exists some OWL-S abstract process *BookAccommodationAbstractProcess* to specify the requirements for a service that provides accommodation booking. Suppose there exists a taxonomy, which categorizes accommodation services into a hierarchy, and contains the concept *HotelsAndMotelsAndInns*.[9] By assigning the stereotype *HotelsAndMotelsAndInns* to the abstract process *BookAccommodationAbstractProcess*, we formally conveyed that only those services, which provide bookings for hotels, motels and inns, can fulfill the role of the *BookAccommodationAbstractProcess*. That also means that those services that provide camping bookings would not do.

Listing 3.5: Example of abstract process stereotype instance

```
<dowls:AbstractProcess rdf:ID="AbstractBookAccommodationProcess">

    <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
        &dowls;#Hotels_and_motels_and_inns
    </dowls:hasStereotype>

    <process:hasInput>
...
```

[9] Concepts *HotelsAndMotelsAndInns* and *Hotels* are taken from the United Nations Standard Products and Services Code taxonomy (UNSPSC, 2006).

Example 3.5.4. Suppose there exists some OWL-S profile *BookHotelProfile* for a service that provides hotel booking. Suppose there exists a taxonomy, which categorizes accommodation services into a hierarchy, and contains the concept *Hotels*, which is a subclass of the concept *HotelsAndMotelsAndInns*.[9] By assigning the stereotype *Hotels* to the profile *BookHotelProfile* we formally conveyed that the service advertised by profile *BookHotelProfile* offers hotel bookings. Consequently such a service could perform the role of the *BookAccommodationAbstractProcess* from the Example 3.5.3, since concept *Hotels* is a subclass of concept *HotelsAndMotelsAndInns*.

Listing 3.6: Example of OWL-S profile stereotype instance

```
<profile:Profile rdf:ID="BookHotelProfile">
    <service:presentedBy rdf:resource="#BookHotelService" />

    <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
        &dowls;#Hotels
    </dowls:hasStereotype>

    <profile:hasInput rdf:resource="#name" />
...
```

Example 3.5.5. An example of an instance of a dowls:AbstractProcess with discovery annotations encoded in OWL is presented in the Appendix A. The example uses a subset of UNSPSC taxonomy encoded in OWL (Klein, 2002) for annotation of abstract process, and a proprietary OWL taxonomy for annotation of parameters. The values of the stereotypes are URI identifiers of the classes from these taxonomies.

3.5.3 Semantic Service Discovery based on Siena with Bag Extension

This section presents the design of a semantic service discovery system based on semantic CBN with bag extension introduced in § 3.4. The design builds upon abstract processes from § 3.5.1 and stereotypes from § 3.5.2. First we will look at the agents, which are the primary actors in our decentralized workflow model, then we will define service matching rules and in the subsequent sections we will look at the adoption of these matching rules to content-based network.

3.5.3.1 The Role of an Agent

The decentralized workflow model presented by this project is comprised of peer-to-peer agents connected to a CBN middleware. Each agent is actually a service container that can host any number of services. The hosted service can be one of (i) an atomic service, (ii) a composite service, or (iii) a composite service template.

By the term *host* we assume that the OWL-S description of the semantic Web service is registered with the agent via service deployment specification described in § 3.7.

Both atomic services and composite services are executable. That is not the case for composite service templates which are comprised of abstract processes. An abstract process is executable only if the agent is able to discover matching executable services for all comprising abstract processes. Matching executable services can be either hosted locally on the same agent or remotely on some other peer agents. The duty of the discovery system presented here is to enable the discovery of matching executable services.

Each agent in this model is able to interact with the CBN middleware in the following ways:

- The agent hosting an executable service can *multicast service notifications* about that service. The agent performing that role is called the *service provider*.
- The agent hosting a composite service template can *subscribe for service notifications* for the services that can be invoked in place of the template's abstract processes. The agent performing that role is called the *service consumer*. The service consumer needs services offered by the service providers.

The same agent can simultaneously have both roles, the one of the service provider and the one of the service consumer. The network topology is truly peer-to-peer, since any number of agents can adopt either one or both roles at any time. All agents are equal in their capabilities and responsibilities.

The primary role of the CBN in this service discovery model is the effective routing of service notifications from service providers to service consumers to support their dynamic discovery.

3.5.3.2 Stereotype Matching Rules

In § 3.5.2 discovery annotations were presented. Semantic services can be annotated with stereotypes that characterize them for purposes of advertisement, discovery, and selection. Stereotypes must be correctly interpreted by the discovery system. It should be reminded that ab abstract process is a specification of a required process, and profile is a presentation of some available service. If a service can fulfill the requirements of an abstract process, then we say that the service and the abstract process match.

Suppose a service is presented by a profile and an abstract process. Matching rules follow:

Input parameters match if the stereotype of the profile's input *subsumes* the stereotype of the abstract process's input. That is because the semantically more specific input of the abstract process can be assigned to the less specific input of the service.

Output parameters match if the stereotype of the abstract process's output *subsumes* the stereotype of the profile's output. That is because the semantically more specific output of the service can be assigned to the less specific output of the abstract process.

Service and abstract process match if (i) the abstract process stereotype *subsumes* the service profile stereotype, and (ii) any profile's input has its own matching counterpart among the abstract process's inputs, and (iii) any abstract process's output has its own matching counterpart among the profile's outputs.

These matching rules rely upon the availability of a common ontology for profile and abstract process stereotypes, and a common ontology for input and output stereotypes.

The most important rule is the last one that determines the matching of the service and the abstract process. It can be formally expressed using ontological and composite bag relations.

Definition 3.5.3. A service, presented by a profile, and an abstract process match if and only if all of the following is true:

1. The profile stereotype is *subsumed by* the abstract process stereotype:

$$\text{profile stereotype} \dashV \text{abstract process stereotype}$$

2. The bag of profile's input stereotypes is a *subbag* of the bag of abstract process's input stereotypes *with regard to subsumes*:

$$\text{bag of profile's input stereotypes} \subseteq_{\Vdash} \text{bag of abstract process's input stereotypes}$$

3. The bag of profile's output stereotypes is a *superbag* of the bag of abstract process's output stereotypes *with regard to subsumed by*:

$$\text{bag of profile's output stereotypes} \supseteq_{\dashV} \text{bag of abstract process's output stereotypes}$$

Remark. The matching rules do not require that for each abstract process's input there exists a service's input. A service can match the abstract process even if it has less inputs than the abstract process, as long as (i) the remaining inputs match, (ii) its profile stereotype is subsumed by the abstract process stereotype, and (iii) it provides an output for each of the abstract process outputs. In extreme case the service could have no inputs and it would still match the abstract process as long as the above conditions are true.

Remark. The model proposed here is simplistic and serves as an illustration of the concept. The matching rules could be actually much more elaborated. They could consider preconditions and effects in the same way they consider inputs and outputs. Another very useful feature would be to annotate inputs, outputs, preconditions and effects as mandatory or optional. It would be very easy to express these rules using composite bag relations, but due to time limitations of this research that is not included in this work.

3.5.3.3 Service Notifications and Subscriptions Filters

The previous section defined the matching rules using ontological and composite bag relations. Since extended Siena supports all these relations, it is possible to (i) encode service notifications as Siena notifications, (ii) encode service matching rules as Siena subscription filters, and (iii) use Siena to route service notifications from service providers to service consumers, thereby performing the discovery filtering in the network.

The format of a service notification is described in the Figure 3.6. A service notification comprises (i) a unique identification of the service, (ii) a unique identification of the hosting agent, (iii) service stereotypes needed for matching, and (iv) validity of the advertisement.

Figure 3.6: Service notification format

Attribute Name	*Type*	*Value Description*
serviceUri	string	URI of service
agentUrl	string	URL of hosting agent[10]
serviceStereotype	string	Stereotype of profile
inputStereotypes	bag	Stereotypes of all profile's inputs
outputStereotypes	bag	Stereotypes of all profile's outputs
validity	integer	Validity of this advertisement in seconds[11]

The format of the service subscription filter is described in the Figure 3.7.

Figure 3.7: Service subscription filter format

Attribute Name	*Type*	*Operator*	*Value Description*
serviceStereotype	string	$\dashv\vert$	Stereotype of abstract process
inputStereotypes	bag	$\subseteq_{\Vdash}$	Stereotypes of all abstract process's inputs
outputStereotypes	bag	$\supseteq_{\dashv\vert}$	Stereotypes of all abstract process's outputs

The ontologies used to describe stereotype values must be made available to extended Siena.

Remark. The discovery system presented here only matches parameters' stereotypes, but not their data types. It is the duty of the annotation creator to assure that if stereotypes match, then data types are compatible.

For an example of the semantic Web service advertisement and discovery system based on this design refer to § 5.2.2.

3.6 Control and Data-Flow in Decentralized Execution

Once the agent hosting a composite service template discovered the services that match its comprising abstract processes, the agent can trigger the decentralized execution of

[10] Each agent has a unique URL.

[11] The validity of the advertisement can stretch from several tens of seconds to several hours or even days, depending on the dynamism of the environment.

the composite service template.

This section describes the design of the decentralized composite service execution model, whereby the responsibility of coordinating a composite service is distributed across several agents. There are two complementary parts of the composite process workflow: the *control-flow* and the *data-flow*. The control-flow defines sequencing of activities in the process. The data-flow defines how information flows between activities (Kalogeras et al., 2006). The description of these two flows describes the workflow.

The design presented here is based upon the algorithm described by Benatallah et al. (2002). Only a short summary of the algorithm is provided in this report. The detailed description, with the exception of data-flow issues, is available in (Benatallah et al., 2001, 2002, 2003). The data-flow issues are not covered by Benatallah et al. (2002) and for this reason their original algorithm has been extended with data-flow design by this research.

Benatallah et al. showed that it is possible to represent a composite service as a state chart, which is a set of *states* and *transitions* between states. Hence it is possible to decompose owls:CompositeProcess, that is to be executed in a decentralized fashion, into *states* and *transitions* between states. Each state, except of the initial and final state, corresponds to an owls:Perform construct and is labeled with the abstract process that is associated with that owls:Perform. Transitions are labeled by *event-condition-action* rules. When a transition fires, its target state is entered providing the condition is true. The event, condition, and action parts of the transition are all optional (Benatallah et al., 2002).

3.6.1 State Coordinators

In a decentralized execution model the constituent states are not executed locally, but by distributed peer-to-peer agents. Therefore, when the request for the invocation of a composite service is sent to an agent that hosts the composite service, a lightweight scheduler, named *state coordinator*, is created by that agent for each constituent state. The state coordinators are then distributed to and provisioned at the peer agents that are going to participate in a decentralized execution of the composite service. There are three types of state coordinators:

Executable state coordinator corresponds to an executable state. It is associated with a service that can fulfill the abstract process associated with the state. The executable state coordinator must be provisioned at the participating agent that hosts that service.

Initial state coordinator corresponds to the initial state. It is provisioned locally at the agent that hosts the composite service.

Final state coordinator[12] corresponds to the final state. It is also provisioned locally at the agent that hosts the composite service.

A state coordinator is a lightweight scheduler responsible for (Benatallah et al., 2003):

- receiving notifications of completion from other state coordinators and determining when to enter the state from these notifications,
- invoking the associated service, once all preconditions for entering the state are met, and waiting for a reply; and
- notifying the coordinators of the states that might need to be entered next that the associated service execution is complete.

The behavior of the coordinator is determined by its *routing table* which specifies the control and data-flow during decentralized execution.

A routing table is comprised of three sets:

- a set of *pre-conditions* such that the state is entered when one of these preconditions is met,
- a set of *post-processing* actions indicating which coordinators need to be notified when a state is exited, and

Control-flow

- a set of *data bindings* describing how output values of the current state are bound to the input values of the succeeding states.

Data-flow

The sets of pre-conditions, post-processing actions and data bindings are defined in a way to ensure minimal communication overhead. When a state is exited, only those states that potentially need to be entered next are notified, and data parameters are only sent to those states that potentially need them.

3.6.2 Pre-Conditions

Pre-conditions of the state describe, (i) what are the source states of the transitions leading to a given state, and (ii) what are the conditions that need to be satisfied for this transition to be taken (Benatallah et al., 2001). A pre-condition has the form $E[C]$, where:

- E is a logical conjunction of *ready* events. Each *ready* event indicates that the current state has received notification of completion from some proceeding state.
- C is a logical conjunction of conditions appearing in the state chart's transitions.

The initial state is a special case that has no pre-conditions.

3.6.3 Post-Processing Actions

Post-processing actions of the state describe, which states may need to be entered next. A post-processing actions has the form $[C]/A$, where:

- C is a logical conjunction of conditions appearing in the state chart's transitions.

[12] The original design of Benatallah et al. does not differentiate between the *initial* and *final coordinator*, however functions of both are performed by the *initial coordinator*.

- A is a *notify* action. *Notify* action describes which succeeding state needs to be notified of completion of the current state.

The final state is a special case that has no post-processing actions.

Remark. Conditions of pre-conditions and post-processing actions are not supported by the prototype developed in scope of this research due to time limitations.

3.6.4 Data Bindings

The data-flow design presented here uses *explicit data-flow model* described by Sadiq et al. (2004). *Explicit data-flow model* is carried out through the explicit data-flow messages, as distinguished from the *implicit data-flow model* which makes use of control-flow to pass data from one state to another.

Data bindings of the state describe which states may need the parameters provided by the current state and how these parameters are bound. A data binding has the form $S[B]$, where:

- S is the destination state, whose input values need to be bound to some output values of the current state.
- B is a a set of *parameter bindings*. Each *parameter binding* is described by the name of an output parameter of the current state, and the name of an input parameter of the destination state.

Again, initial and final states are special cases, because they do not perform any action, and hence they do not need any inputs and do not produce any outputs by themselves. Notwithstanding their input and output parameters need to be defined to be able to participate in the data-flow. Therefore parameters of the initial and final state are defined as follows:

- Input and output parameters of the *initial state* are both equal to the *input parameters of the composite process.*
- Input and output parameters of the *final state* are both equal to the *output parameters of the composite process.*

Before decentralized execution the inputs of the composite service are provided to the initial state and after completion the outputs of the composite process are collected from the final state.

3.7 Agent Architecture

In the previous sections of this chapter the fundamental concepts of this design have been presented. Here all these concepts are put together to provide the architecture for the essential component of our decentralized service execution model, called *agent.* The agent is an autonomous software component which is able to host, advertise and execute atomic and composite semantic Web services. It is also able to discover services advertised by other agents and collaborate with other agents in decentralized execution of composite services. The architecture of agent is partly based on the Self-Serv environment for Web services composition described in § 2.2.4.1.

The agent itself is implemented as a standard non-semantic Web service and uniquely identified by the URL address of that Web service.

The agent can host all known types of OWL-S services, including composite service templates.[13] By the term *host* we assume that the OWL-S description of the semantic Web service is registered with the agent via a service deployment specification, which is a file with the logical format presented in the Figure 3.8.

Figure 3.8: Format of the OWL-S service deployment specification

Property	*Description*	*Example Value*
owlUrl	URL of OWL-S service description.	http://kyi/BB.owl
advertisable	Whether the agent should advertise the service or not.	true
advertisementPeriod	Time in seconds between successive service notifications.	300
advertisementValidity	The validity of the service notification in seconds.	150
decentralized	Whether the service is to be executed locally or in decentralized fashion.	true

The agent serves as a semantic Web service container. It is comprised of the following software components (see Figure 3.9):

Service Deployer component is responsible for the deployment and undeployment of semantic Web services. At deployment it registers the newly deployed service with the Service Advertisement Engine. In case of the composite service the deployer is also able to decentralize the composite process and register its constituent abstract processes with Service Discovery Engine.

Composite Service Compiler component compiles composite Web services into a form suitable for decentralized execution. It identifies constituent abstract processes and generates abstract routing tables for control and data-flow.

Deployed Service Repository component stores metadata about Web services hosted by the agent.

Service Advertisement Engine component advertises the availability of deployed Web services to other agents via CBN middleware.

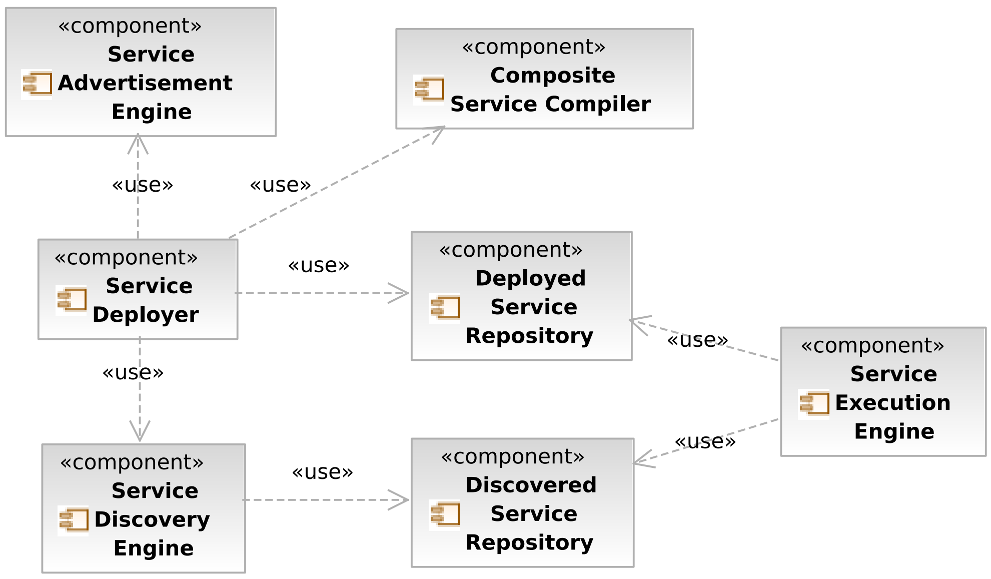
[13] Composite service template is not a part of OWL-S specification. It is explained in § 3.5.1.3.

Service Discovery Engine component listens for advertisements for services hosted by other agents via CBN middleware.

Discovered Service Repository component stores metadata about Web services hosted by other agents, which have been discovered by the Service Discovery Engine.

Service Execution Engine component is able to autonomously execute hosted atomic Web services, initiate execution of hosted composite Web services, and participate in execution of composite Web services hosted by other agents.

Figure 3.9: Agent component diagram


«component» Service Advertisement Engine
«component» Composite Service Compiler
«use»
«use»
«component» Service Deployer
«use»
«component» Deployed Service Repository
«use»
«component» Service Execution Engine
«use»
«use»
«component» Service Discovery Engine
«use»
«component» Discovered Service Repository

In the following sections the architecture of each of these software components is presented.

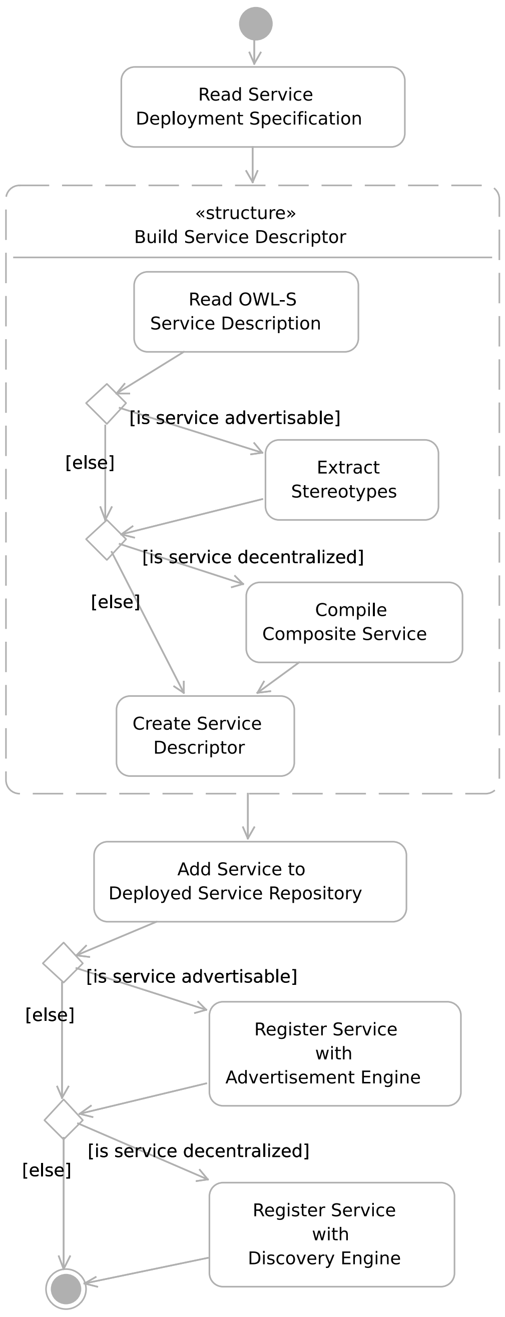
3.7.1 Service Deployer

The Service Deployer component is responsible for the deployment and undeployment of semantic Web services. It monitors a preconfigured deployment folder[14] for service deployment specification files.

When Service Deployer detects a new deployment specification it executes the service deployment activities presented in the Figure 3.10. Conversely, when service deployer detects the removal of a deployment specification it executes undeployment activities presented in the Figure 3.11. In the case when a service deployment specification is modified while the service is being deployed, the service deployer redeploys the service by undeploying and deploying it again. Moreover, all previously deployed services must be deployed at the agent startup and undeployed at the agent shutdown.

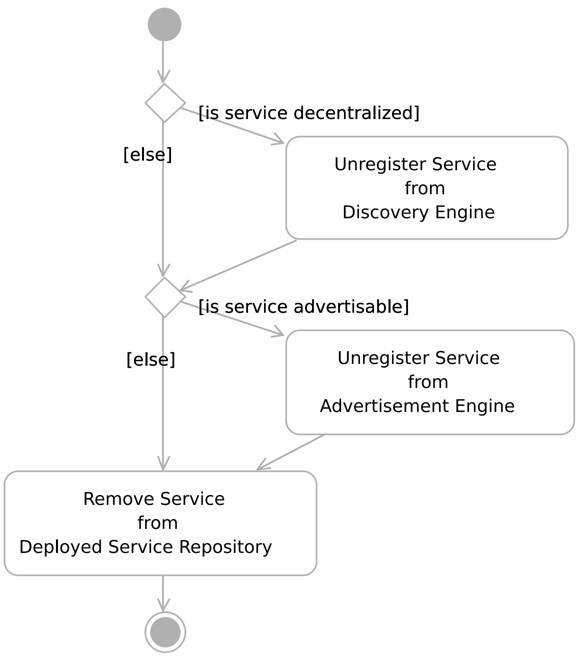
[14] Term *folder* denotes *file system directory*.

Figure 3.10: Service deployment activities



Read Service Deployment Specification
«structure» Build Service Descriptor
Read OWL-S Service Description
[is service advertisable]
[else]
Extract Stereotypes
[is service decentralized]
[else]
Compile Composite Service
Create Service Descriptor
Add Service to Deployed Service Repository
[is service advertisable]
[else]
Register Service with Advertisement Engine
[is service decentralized]
[else]
Register Service with Discovery Engine

Figure 3.11: Service undeployment activities



[is service decentralized]
[else]
Unregister Service from Discovery Engine
[is service advertisable]
[else]
Unregister Service from Advertisement Engine
Remove Service from Deployed Service Repository

3.7.2 Composite Service Compiler

The Composite Service Compiler component compiles decentralized composite services into the form suitable for decentralized execution. It analyzes the composite process and decomposes it into constituent states as described in (Benatallah et al., 2001, 2002, 2003) and § 3.6. The compiler does not produce concrete states, but abstract states. While concrete states are associated with concrete executable services, abstract states are associated with constituent abstract processes. At the composite service execution time these abstract states are used by Service Execution Engine as templates to produce concrete states, which are associated with concrete discovered services, and have concrete routing tables.

3.7.3 Deployed Service Repository

The Deployed Service Repository component maintains the in-memory repository of services hosted by the local agent. For every hosted service it stores the following information:

- parsed OWL-S description of the service;
- service deployment specification;
- if service is advertisable: its profile stereotype, input stereotypes, and output stereotypes;
- if service is decentralized: its routing tables;

3.7.4 Service Advertisement Engine

Service Advertisement Engine component is responsible for multicasting Siena service notifications for hosted services that are labeled as *advertisable*. Advertisement engine is multicasting notifications periodically as specified by the service deployment specification. The format of service notification is described in § 3.5.3.3.

Decentralized composite services labeled as *advertisable* are advertised only if they are executable, in other words, if the agent knows at least one executable service for each of its constituent abstract processes. There would be no sense in advertising services that cannot be executed.

3.7.5 Service Discovery Engine

The agent can host composite services that are labeled as *decentralized*. These services are not executed locally, but in a decentralized fashion. Each such composite service is decomposed into abstract processes by the composite service compiler at deploy-time. For each such abstract process the agent has to discover at least one matching executable service that can fulfill the role of the abstract processes. The duty of the Service Discovery Engine component is to discover these matching executable services.

Suppose there exists a decentralized composite service whose composite process is comprised of abstract processes. Suppose the service is being hosted by an agent. The responsibilities of the service discovery engine are the following:

At service deploy-time the engine creates Siena service subscription for each of the service's constituent abstract processes. The constituent abstract processes are identified by Composite Service Compiler component.

During service hosting-time the engine is receiving Siena service notifications that match its subscriptions. The received service notifications are forwarded to the Discovered Service Repository component described in the next section.

At service undeploy-time the engine unsubscribes from Siena service notifications. It also informs the Discovered Service Repository component about it.

3.7.6 Discovered Service Repository

The Discovered Service Repository maintains an in-memory repository of discovered services. It is continually receiving notifications for discovered services from Service Discovery Engine component. Each service notification contains the validity of notification in seconds. The discovered service is kept in repository until the notification validity expires, but each new notification for the same discovered service resets its expiration timer.

The repository also keeps the evidence about the number of composite services that need the discovered service to fulfill the roles of their abstract processes. If all composite services that are interested in particular discovered service get undeployed, then the repository immediately evicts the discovered service, since it is no longer needed.

Example 3.7.1. Suppose a service provider hosts an advertisable service, which is advertised every 1 hour, and the validity of a notification is 2 hours. Suppose this service is discovered by a service consumer and for this reason registered in its own discovered service repository. The discovered service is kept in the repository as long as the service provider is alive. Because every hour a new notification arrives that resets its expiration timer, the discovered service never expires. But once the service provider dies, or the provided service gets undeployed, the discovered service gets evicted from the repository in maximum 2 hours.

Remark. Note that this design allows for unavailable services to be kept in a discovered service repository for a certain period of time until their validity expires. That does not influence the robustness of a service execution engine. During the provisioning of a decentralized execution the execution engine detects the unavailability of the service. As a consequence the execution engine immediately forces eviction of the service from the discovered service repository and selects an alternative suitable discovered service if one exists.

3.7.7 Service Execution Engine

The Service Execution Engine component is able to autonomously execute hosted non-decentralized semantic Web services, or participate in peer-to-peer execution of decentralized semantic Web services.

Remark. By *autonomous execution* we do not assume that service's business logic is actually executed by the agent, but merely that the semantic Web service is being invoked by the agent. Which service implementation is actually being executed depends on the grounding of the semantic Web service and is of no importance for this work.

The capabilities of the Service Execution Engine depend on the type of the hosted Web service:

- An *atomic service* can be executed locally by the agent using OWL-S Java API (Sirin and Parsia, 2004). An atomic service is not allowed to be labeled as *decentralized* or else an error occurs.
- A *composite service* can be optionally labeled as *decentralized*:
  - If it is not labeled as *decentralized*, then the service is locally executable in the same way as the atomic service using OWL-S Java API. This case will not be specifically considered, since its treatment is completely identical to the treatment of the atomic service.
  - Alternatively the composite service can be labeled as *decentralized*. Such a service is treated in the same way as the composite service template described below, and its grounding is ignored.
- A *composite service template* must be labeled as *decentralized* or else an error occurs. At deploy time the agent compiles the composite service into the form appropriate for decentralized execution. The composite service template is not immediately executable when deployed to an agent. It becomes executable only after the agent discovers appropriate service providers for all comprising abstract processes. If an executable composite service template is invoked, then its hosting agent does not control the whole execution, but merely selects and binds the participating service providers, initiates the start of the execution, waits for completion and returns the result. The execution itself is performed in a peer-to-peer fashion between participating service providers.

The agent can also participate in the execution of a larger composite process.

3.7.7.1 Structure

The essential sub-components or Service Execution Engine component are the following:

Service Wrappers are software components that act as the semantic Web service's entry points for the execution engine itself.

State Coordinators are software components that act as lightweight schedulers, which coordinate control and data-flow during a decentralized execution of a composite process.

Agent Façade is a standard non-semantic Web service that acts as the engine's entry point for clients of the system and peer agents.

3.7.7.2 Service Wrappers

Service wrappers are software components of the agent that act as the semantic Web service's entry points for the execution engine itself. There are two types of service wrappers:

- The *local service wrapper* is used to execute the service autonomously by the hosting agent. It can execute atomic and composite services not labeled as decentralized.
- The *decentralized service wrapper* is used to initiate decentralized execution of composite services and composite service templates labeled as decentralized. The components of the composite process are treated as abstract processes by the decentralized service wrapper, although they are not really abstract, thus making composite services equal to composite service templates. For this reason we will not distinguish between composite services and composite service templates when talking about decentralized execution. They will be both treated equally as composite service templates.

Service wrappers are created and destroyed dynamically. A service wrapper is created when an invocation request appears and is destroyed after the completion of the invocation.

3.7.7.3 State Coordinators

State coordinators are software components that act as lightweight schedulers, which coordinate control and data-flow in decentralized execution of a composite process. In the decentralized execution model the responsibility of coordinating the execution of a composite service is distributed across several peer-to-peer agents. Each agent hosts one or more state coordinators that interact in a peer-to-peer fashion in order to ensure that each instance of a composite service is executed in accordance with its control and data-flow specifications. The behavior of a state coordinator is determined by its routing table as described in § 3.6.

3.7.7.4 Agent Façade

The *agent façade* is a standard non-semantic Web service that acts as the execution engine's entry point for the clients of the system and for the peer agents. It enables clients to invoke the hosted Web service and agents to communicate during decentralized execution of the composite service. The façade provides the operations:

invokeService is used by a client to invoke the service hosted by the agent. The service is executed either locally or in decentralized fashion depending on the deployment specification of the service. After the completion the results are returned to the client.

getServiceDescriptor is used by peer agents to get a detailed technical description of the discovered service during the provisioning of a decentralized execution. This operation is requisite because service discovery is only able to provide the capabilities of the service (for example, profile stereotype and parameter stereotypes), and not technical details needed for the provisioning of the decentralized execution (for example, parameter names). The rationale behind is to keep the advertisement network traffic as low as possible.

provisionStateCoordinator provisions state coordinator at the agent. The provisioning of the state coordinator is performed by some peer agent during the provisioning of a decentralized execution.

assignStateCoordinatorInputs is used to realize the data-flow during a decentralized execution. A remote peer agent can use this operation to assign outputs of some state coordinator hosted by his execution engine to the inputs of some state coordinator hosted by the local execution engine.

notifyStateCoordinator is used to realize the control-flow during a decentralized execution. A remote peer agent can use this operation to notify some state coordinator hosted by the local execution engine about the completion of some state coordinator hosted by his execution engine.

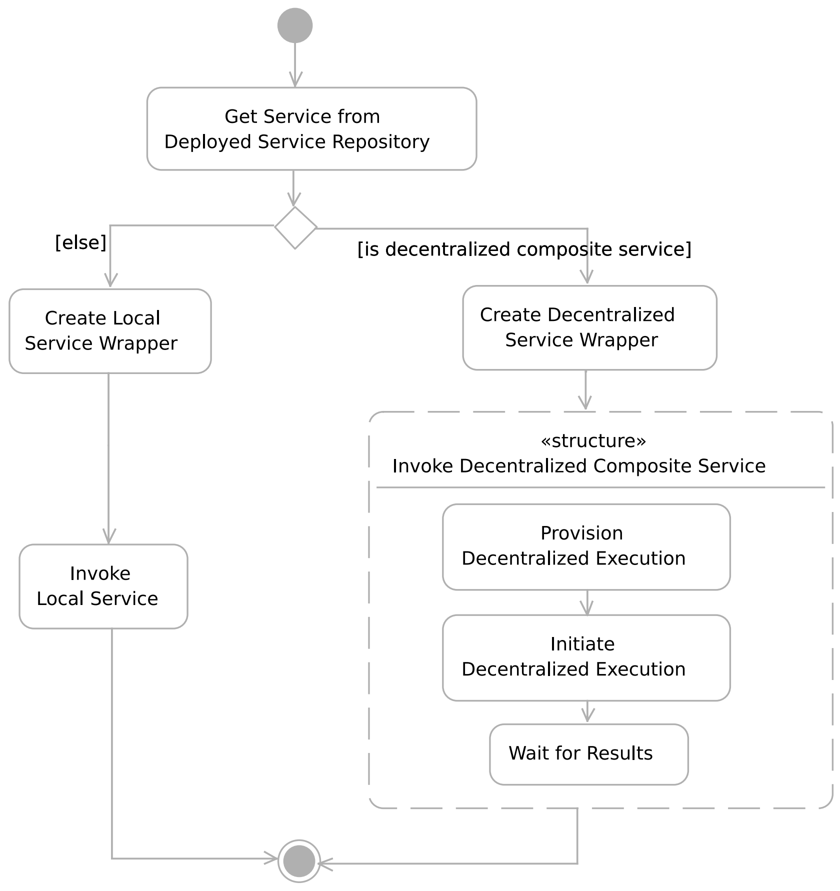
3.8 Service Execution

The principal capability of the agents described in § 3.7 is the ability to orchestrate decentralized execution of composite semantic Web services.

A client that would like to invoke a Web service must first find an agent that hosts that service. The URL address of the appropriate agent can be either statically configured at the client, or alternatively the client can use the same service discovery mechanism that is used by the agents themselves, to discover the service that suits its needs. Once the client knows the URL address of the hosting agent, it can send the service invocation request to the agent's façade.

When the agent receives the request, it instantiates either local or decentralized service wrapper as shown on Figure 3.12.

Figure 3.12: Execute service activities



3.8.1 Local Service Execution

The local service wrapper is responsible for the invocation of non-decentralized services. The execution of such services is straightforward and is carried out by OWL-S Java API library (Sirin and Parsia, 2004).

3.8.2 Decentralized Service Execution

The decentralized service wrapper is responsible for the invocation of decentralized composite services. The invocation of a decentralized service is comprised of two main phases, namely the *provisioning phase* and the *execution phase*. The sequence diagram illustrating the main steps is presented in Figure 3.13.

3.8.2.1 Provisioning Phase

In the provisioning phase the decentralized service wrapper selects and provisions the participating peer agents. The main steps performed during the provisioning phase are:

1. The service wrapper first selects discovered services that can fulfill the roles of constituent abstract processes. Simultaneously it also determines mappings between abstract process's parameters and discovered service's parameters. The parameter mapping algorithm must take into account stereotypes of the abstract process's parameters and stereotypes of the discovered service's parameters, and find a mapping that satisfies parameter matching rules described in § 3.5.3.2. Such mapping must exists or the service would not have been discovered.
2. Once the services that can fulfill the roles of constituent abstract processes are discovered, the service wrapper creates the concrete initial and final state, and concrete executable states with concrete routing tables.
3. Afterwards the service wrapper provisions state coordinators. The coordinators for initial and final state are provisioned locally. The coordinators for executable states are provisioned at the peer agents that host the services associated with the executable states.

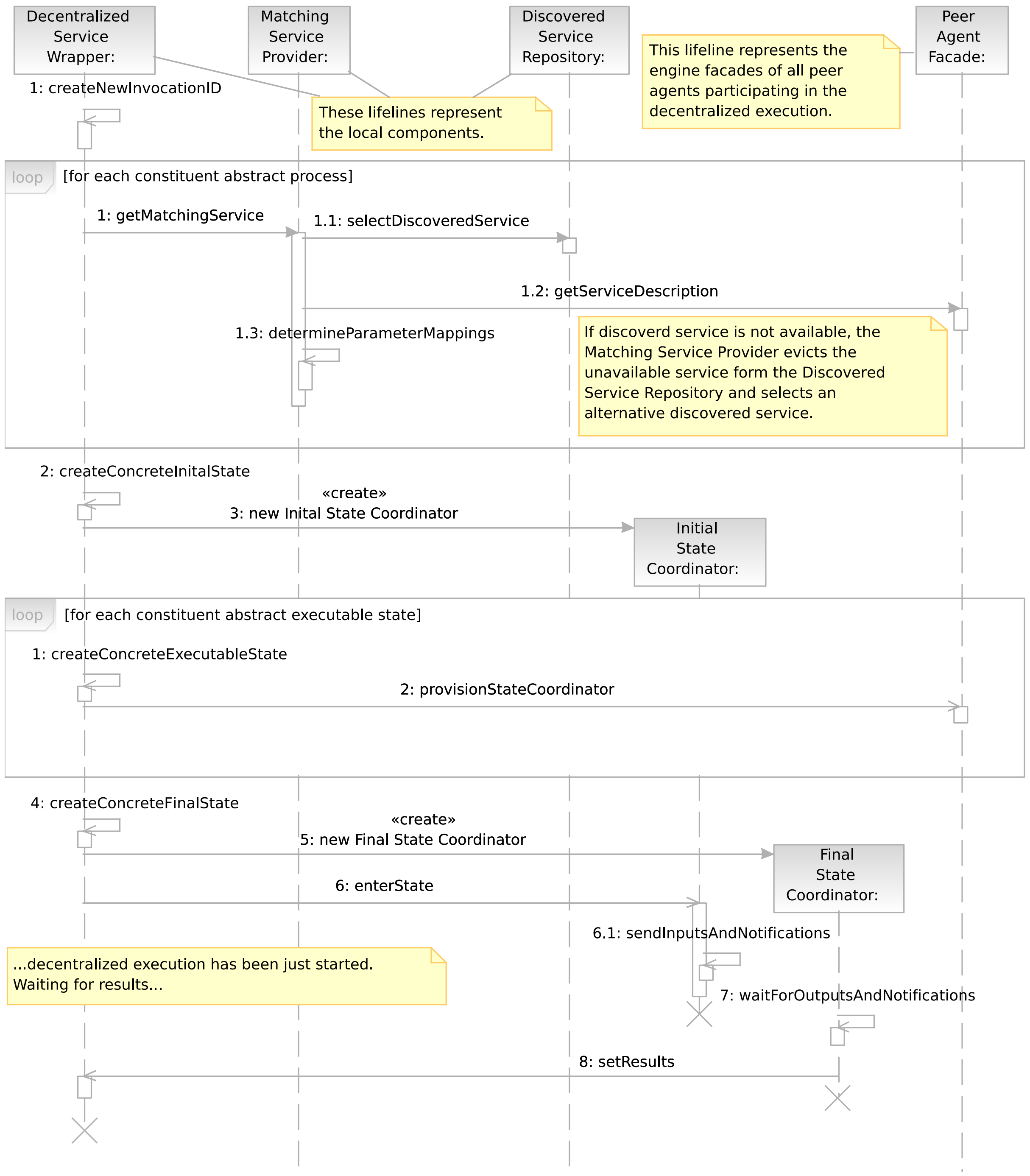
3.8.2.2 Execution Phase

After the provisioning phase the decentralized service wrapper initiates the decentralized execution. The execution itself is orchestrated by the state coordinators created during the provisioning phase. The decentralized service wrapper waits for the completion of the decentralized execution and returns the results back to the client.

The course of events unfolding during decentralized execution phase can be best explained by an example presented in § 5.2.3.

Remark. The design of decentralized composite service execution presented here could be improved in many ways. Some possible optimizations are:

Figure 3.13: Invocation of a decentralized composite service


Decentralized Service Wrapper:
Matching Service Provider:
Discovered Service Repository:
This lifeline represents the engine facades of all peer agents participating in the decentralized execution.
Peer Agent Facade:
1: createNewInvocationID
These lifelines represent the local components.
loop
[for each constituent abstract process]
1: getMatchingService
1.1: selectDiscoveredService
1.2: getServiceDescription
1.3: determineParameterMappings
If discoverd service is not available, the Matching Service Provider evicts the unavailable service form the Discovered Service Repository and selects an alternative discovered service.
2: createConcreteInitalState
«create»
3: new Inital State Coordinator
Initial State Coordinator:
loop
[for each constituent abstract executable state]
1: createConcreteExecutableState
2: provisionStateCoordinator
4: createConcreteFinalState
«create»
5: new Final State Coordinator
Final State Coordinator:
6: enterState
6.1: sendInputsAndNotifications
...decentralized execution has been just started. Waiting for results...
7: waitForOutputsAndNotifications
8: setResults

- In the existing design state coordinators are provisioned prior to every decentralized execution and destroyed after the completion. In situations, where successive executions of the same decentralized composite service are likely to occur, the state coordinators could be partially cached at hosting agents to reduce the provisioning overhead.
- The decentralized execution model presented here lacks any monitoring and error handling capabilities. The implementation of these would be crucial for real-world scenarios.

Chapter 4

Implementation

Two main software artifacts have been developed in scope of this research:

Bag extension for Siena provides support for bags, simple bag operators, and composite bag operators for Siena.

Dowls[1] **Agent prototype** implements the functionality of the peer-to-peer agent which is able to advertise and execute OWL-S services, and, in collaboration with other agents, orchestrate decentralized execution of composite OWL-S services.

4.1 Bag Extension for Siena

4.1.1 Technologies Used

Bag extension for Siena is based upon ontologically extended Siena, which is itself extended from a Java version of Siena (Carzaniga et al., 2001). It is developed in Java and requires Java 2, Standard Edition 5.0 runtime.

In addition the following open source Java libraries have been used:

- Jakarta Commons Collections 3.2 provides Java Bag interface for collections.
- Jakarta Commons Lang 2.1 provides helper utilities for the java.lang API.
- Jena 2.3 is a Java framework for building Semantic Web applications.
- Pellet 1.3 is a Java based OWL DL reasoner.

The following tools have been used to assist the development:

- Eclipse Platform 3.2 is an integrated development environment for Java.
- Apache Maven 2.0.4 is a build tool.
- Subversion 1.3.1 is a source code control tool.

[1] *Dowls* is a codename of the prototype. It is an acronym for *Distributed OWL-S*.

4.1.2 Ontologies

The extended Siena needs access to OWL ontologies, used to describe stereotype values in service notifications and subscription filters. This is achieved by means of a configuration file, where URLs of required ontologies are listed.

The extended Siena must also use an ontological reasoner, which is equivalent to the one used by the Dowls Agent. In our implementation they both use Pellet 1.3.

4.1.3 Implementation of the Composite Bag Operator

One of the crucial tasks in the development of Siena bag extension was implementation of the algorithm for comparison of bags by the composite bag operator. The algorithm presented here is a very simple brute force algorithm that clearly illustrates the performance of the composite bag operator. It simply checks all possible arrangements of one bag with another until a matching arrangement is found, or else it returns *false*. The efficiency of the algorithm and possible optimizations are discussed in more detail in § 5.1.3.

The Java source code of the algorithm with descriptive comments is presented in Listing 4.1.

Extended Siena has a method `areValuesRelated`, which is used to compare any pair of valid Siena values against an operator. If the values happen to be bags, then the method delegates the comparison evaluation to the method `areBagsRelated`. Method `areBagsRelated` takes three parameters: composite bag operator, first bag, and second bag. The primary operator of the composite operator can be either *superbag*, *subbag* or *equal*. The method translates the comparison to the *subbag* comparison and delegates its evaluation to the method `areBagsRelatedWithSubbag`.

Listing 4.1: Algorithm for comparison of bags with a composite bag operator

```
/**
 * Compares two bags with the composite bag operator.
 *
 * @param primaryOperator Primary bag operator
 * @param subOperator Sub-operator applicable to bag elements; subOperator can be a composite
 *        operator itself
 * @param xBag First bag; the order of elements is ignored
 * @param yBag Second bag; the order of elements is ignored
 * @return true if and only if bags are related; false otherwise
 */
private boolean areBagsRelated(CompositeOperator compositeOperator,
    List<AttributeValue> xBag, List<AttributeValue> yBag) {
  switch (compositeOperator.getPrimaryOperator()) {
  case ExtOp.EQBAG: // evaluate: xBag is equal to yBag
    if (xBag.size() != yBag.size())
      return false; // The bags can not be equal, if they are not of the same size.
    // The bags are of the same size! Now it is enough to check, if xBag is a subbag
    // of yBag with regard to the subOperator.
    return areBagsRelatedWithSubbag(compositeOperator.getSubOperator(), xBag, yBag,
        false);
  case ExtOp.SUPERBAG: // evaluate: xBag is superbag of yBag
    if (xBag.size() < yBag.size())
      return false; // xBag cannot be superbag of yBag, if it has less elements.
    // xBag does not have less elements than yBag! Now we will check, if yBag is subbag
    // of xBag with regard to the subOperator. Since xBag and yBag are exchanged, the
    // reverseComparison parameter must be true.
    return areBagsRelatedWithSubbag(compositeOperator.getSubOperator(), yBag, xBag,
        true);
  case ExtOp.SUBBAG: // evaluate: xBag is subbag of yBag
    if (xBag.size() > yBag.size())
      return false; // xBag cannot be subbag of yBag, if it has more elements.
    // xBag does not have more elements than yBag! Now we will check, if xBag is subbag
    // of yBag with regard to the subOperator.
    return areBagsRelatedWithSubbag(compositeOperator.getSubOperator(), xBag, yBag,
        false);
  }
}

/**
 * Checks if the first bag is a subbag of the second bag with regard to the given suboperator.
 *
 * @param subOperator Sub-operator applicable to bag elements
 * @param xBag First bag; the order of elements is ignored
 * @param yBag Second bag; the order of elements is ignored
 * @param reverseComparison If true, the elements should be exchanged before the comparison
 * @return true if and only if the first bag is a subbag of the second bag with regard to the
 *         subOperator; false otherwise
 */
private boolean areBagsRelatedWithSubbag(CompositeOperator subOperator,
    List<AttributeValue> xBag, List<AttributeValue> yBag, boolean reverseComparison) {
  // Did we find related elements from yBag for all elements from xBag?
  if (xBag.isEmpty())
    return true;

  // Take the head element of xBag.
  AttributeValue xElement = (AttributeValue) xBag.get(0);
  // Create a new subbag, equal to xBag with removed head element.
  List<AttributeValue> xSubBag = xBag.subList(1, xBag.size());
  // Try to find an element from yBag, that match to taken element from xBag.
  for (int i = 0; i < yBag.size(); i++) {
    // Take the i-th element from yBag.
    AttributeValue yElement = yBag.get(i);
    // Do the taken elements match?
    if (areElementsRelated(subOperator, xElement, yElement, reverseComparison)) {
      // Elements match! Create a new subbag, equal to yBag with removed i-th element.
      List<AttributeValue> ySubBag = ListUtils.subList(yBag, i);
      // Recursively check new subbags.
      if (areBagsRelatedWithSubbag(subOperator, xSubBag, ySubBag, reverseComparison)) {
        // Subbags match as well! Consequently xBag and yBag match!
        return true;
      }
    }
  }

  return false; // We could not find matching element from yBag.
}

/**
 * Checks if two elements match with regard to the given suboperator.
 *
 * @param subOperator Sub-operator applicable to given elements
 * @param xElement The first element
 * @param yElement The second element
 * @param reverseComparison If true, the elements should be exchanged before the comparison
 * @return true if and only if the first element match to the second element with regard to
 *         the subOperator; false otherwise
 */
private boolean areElementsRelated(CompositeOperator subOperator,
    AttributeValue xElement, AttributeValue yElement, boolean reverseComparison) {
  // Recusively call method that checks whether arbitrary two values are related.
  if (reverseComparison) {
    // Exchange elements before comparison.
    return areValuesRelated(subOperator.getPrimaryOperator(), subOperator
        .getSubOperator(), yElement, xElement);
  } else {
    return areValuesRelated(subOperator.getPrimaryOperator(), subOperator
        .getSubOperator(), xElement, yElement);
  }
}
```

4.2 Dowls Agent Prototype

4.2.1 Technologies Used

Dowls Agent is realized as a standard non-semantic Web service. It is implemented as Java Web Application and can run in any servlet container that is compatible with Java 2, Standard Edition 5.0, and supports Servlet 2.4 specification. Examples of such container are Apache Tomcat and JBoss application server.

Additionally the following open source Java libraries have been used:

- Apache Axis 1.4 is a Java based Web service framework.
- Apache Log4j 1.2.13 is a Java based logging utility.
- Jakarta Commons Collections 3.2 provides Java bag interface for collections.
- Jakarta Commons Lang 2.1 provides helper utilities for java.lang API.
- Jena 2.3 is a Java framework for building Semantic Web applications.
- OWL-S API (compiled from the latest source code on 2nd June 2006 and upgraded to Apache Axis 1.4) provides a Java API for programmatic access to OWL-S service descriptions.
- Pellet 1.3 is a Java based OWL DL reasoner.
- Spring Framework 2.0 RC2 is a layered Java/J2EE application framework.

The following tools have been used to assist the development:

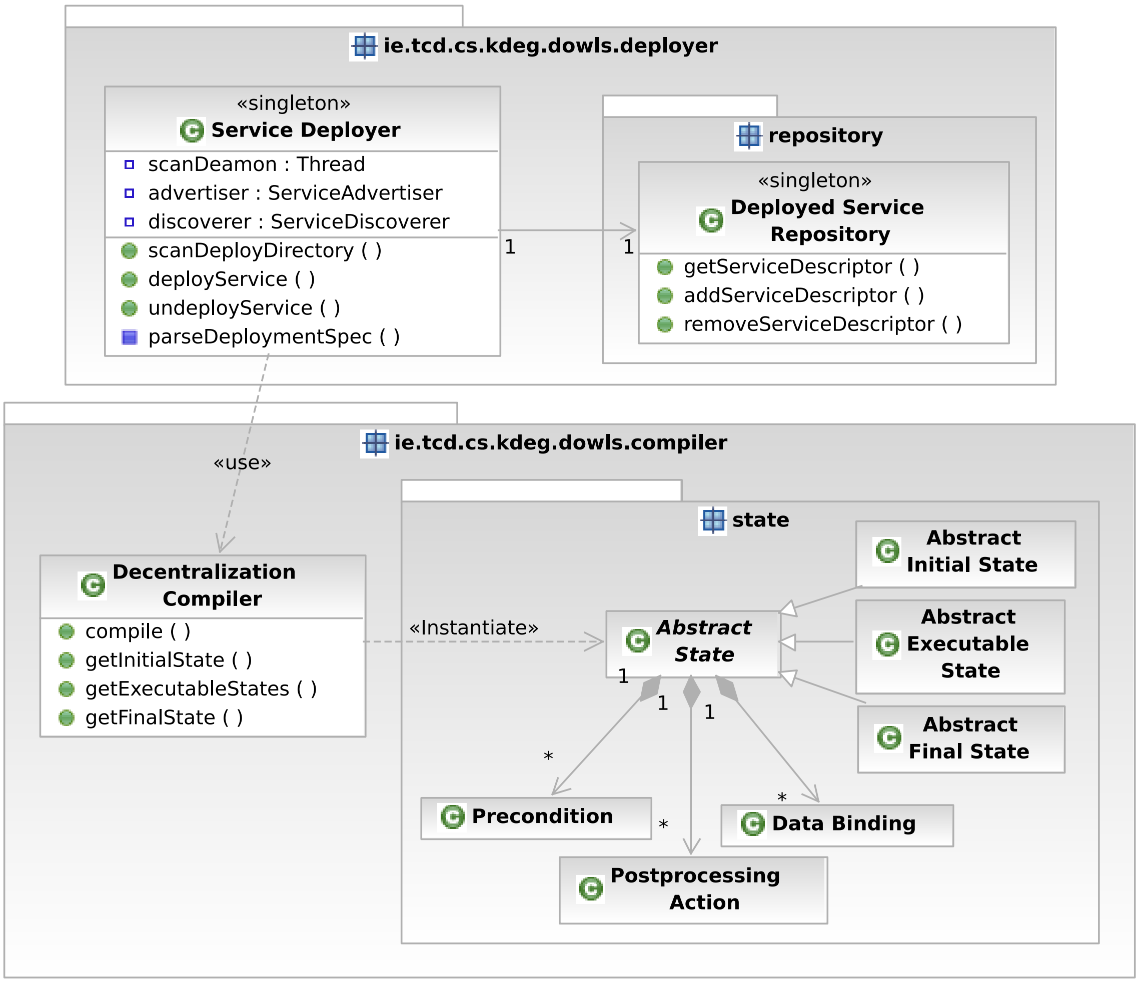
- Eclipse Platform 3.2 is an integrated development environment for Java.
- Apache Maven 2.0.4 is a build tool.
- Subversion 1.3.1 is a source code control tool.

4.2.2 Main Packages and Classes

This section presents the class diagrams of the agent's components. Many details, such as some classes, methods, method parameters, associations and attributes, are intentionally omitted to increase the clarity of the diagrams.

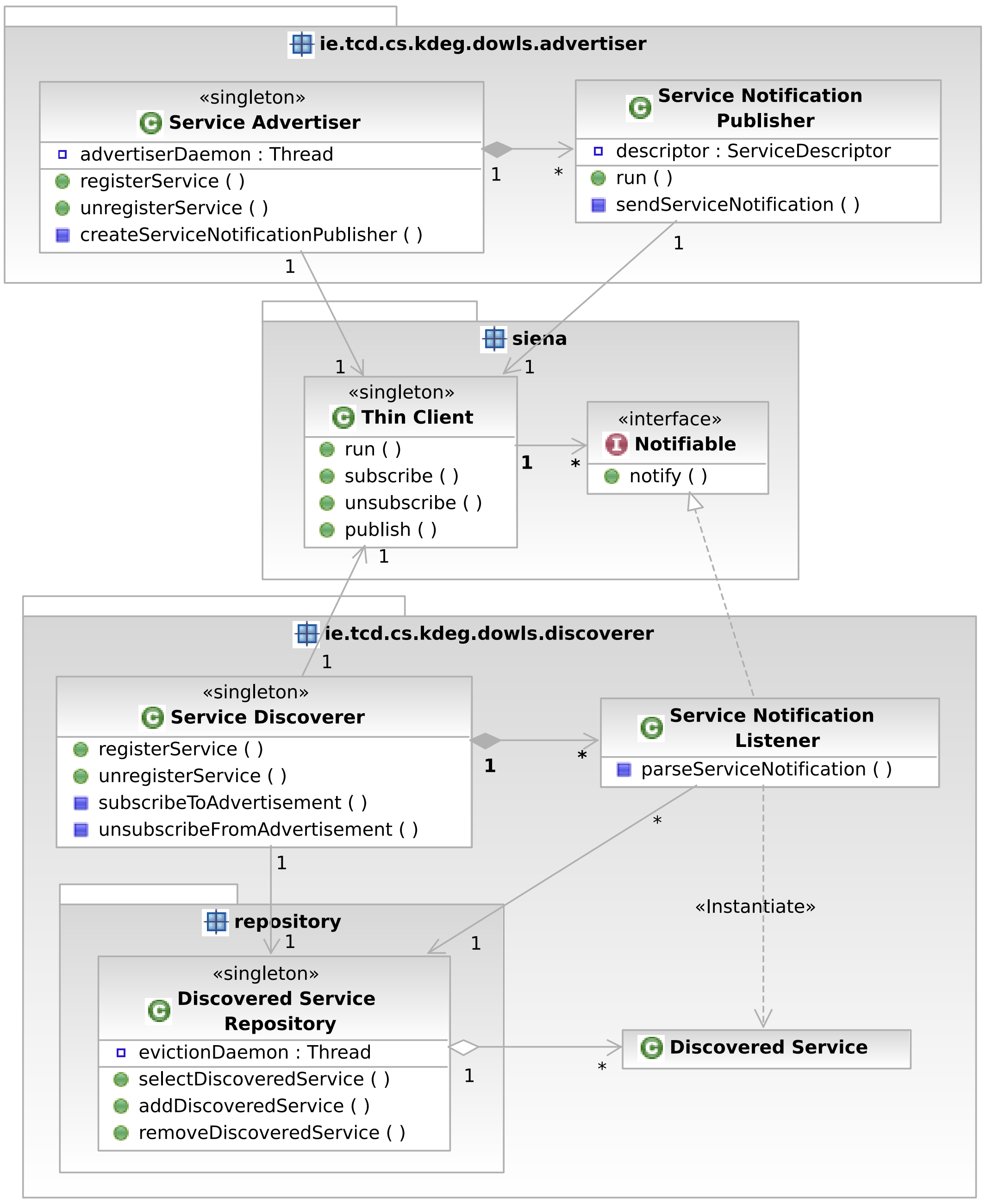
The Figure 4.1 shows the class diagram of the Service Deployer, Composite Service Compiler and Deployed Service Repository components.

Figure 4.1: Service Deployer, Composite Service Compiler and Deployed Service Repository class diagram



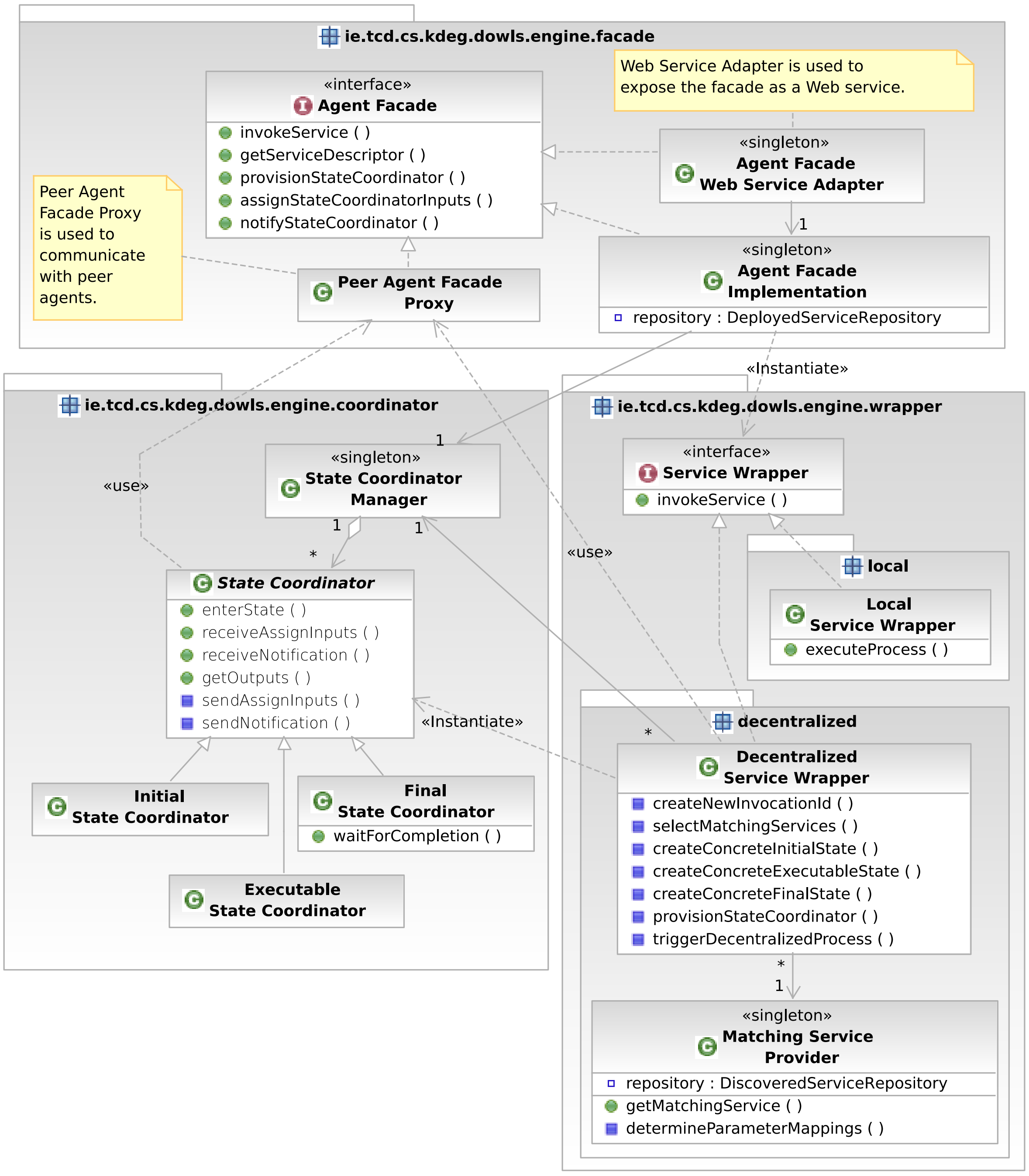
In the same manner the Figure 4.2 shows the class diagram of the Service Advertisement Engine, Service Discovery Engine and Discovered Service Repository components.

Figure 4.2: Service Advertisement Engine, Service Discovery Engine and Discovered Service Repository class diagram


ie.tcd.cs.kdeg.dowls.advertiser
«singleton»
Service Advertiser
advertiserDaemon : Thread
registerService ( )
unregisterService ( )
createServiceNotificationPublisher ( )
Service Notification Publisher
descriptor : ServiceDescriptor
run ( )
sendServiceNotification ( )
1
*
1
1
siena
«singleton»
Thin Client
run ( )
subscribe ( )
unsubscribe ( )
publish ( )
«interface»
Notifiable
notify ( )
1
*
ie.tcd.cs.kdeg.dowls.discoverer
«singleton»
Service Discoverer
registerService ( )
unregisterService ( )
subscribeToAdvertisement ( )
unsubscribeFromAdvertisement ( )
Service Notification Listener
parseServiceNotification ( )
«Instantiate»
repository
«singleton»
Discovered Service Repository
evictionDaemon : Thread
selectDiscoveredService ( )
addDiscoveredService ( )
removeDiscoveredService ( )
Discovered Service

Finally the Figure 4.3 shows the class diagram of the Service Execution Engine component.

Figure 4.3: Service Execution Engine class diagram



Remark. The parameter mapping algorithm, which is implemented in the MatchingServiceProvider class, is using ontological reasoning to reveal relationships between stereotypes. The same principles are employed by extended Siena for routing service notifications. For this reason ontological reasoners must be functionally equivalent on both places.

Chapter 5

Evaluation

5.1 Bag Extension for Content-Based Networking

The evaluation of the bag extension for Siena requires verification of the correct behavior of the system and evaluation of the system performance.

The correctness of the system behavior has been empirically confirmed by applying human performed functional test cases. Due to the lack of time the functional tests have not been formalized and automated.

The performance evaluation should investigate the following:

- performance of notifications containing bags,
- routing performance of subscription filters utilizing simple bag operators,
- routing performance of subscription filters utilizing composite bag operators.

5.1.1 Performance of Notifications Containing Bags

Performance of notification containing bags has not been specifically addressed, since a bag of the size n is roughly equivalent to n attributes with the scalar values equal to the bag's elements. Actually the size of the equivalent bag attribute is even smaller since it contains only one attribute name compared to n attribute names of the equivalent scalar attributes. Therefore the performance of a notification containing a bag attribute is expected to be at least as good, and probably better, than the performance of a notification containing equivalent set of scalar attributes.

5.1.2 Routing Performance of Simple Bag Operators

Routing performance of subscription filters utilizing simple bag operators was also not specifically addressed, since the algorithm for comparing bags with simple bag operators is straightforward. Suppose there exist some bags B and C. Without loss of generality we can assume that $|B| \leqslant |C|$. The simple bag operator comparison has the best time complexity $O(|B|)$ and the worst time complexity $O(|C|^2)$. If bags are pre-ordered, assuming bag elements can be totally ordered, it is possible to use even better algorithm that has the worst time complexity of $O(|C|)$.

5.1.3 Routing Performance of Composite Bag Operators

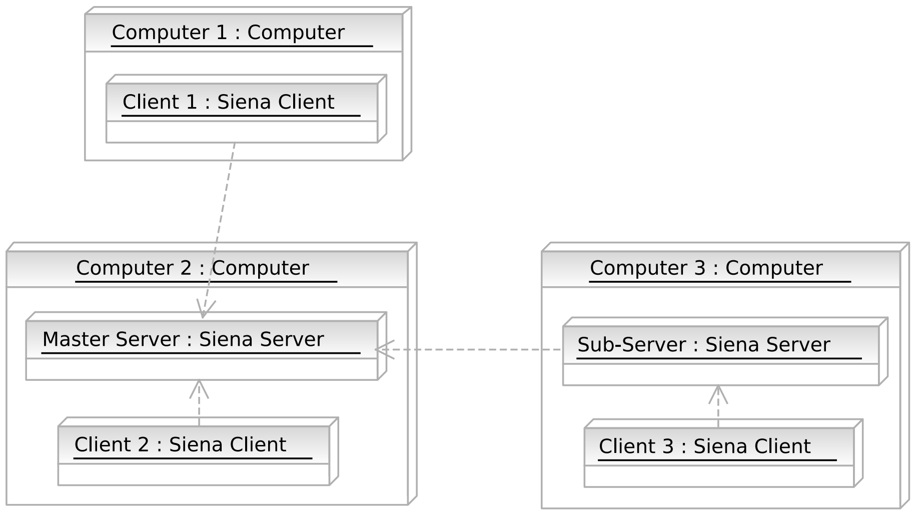
The most interesting case is the routing performance of subscription filters utilizing composite bag operators, since composite bag operators offer the biggest expressiveness. For this reason their performance was evaluated empirically. Measurements were repeated once with the *original* version of the algorithm, and once with an *optimized* version of the algorithm.

The optimized version of the algorithm pre-orders the elements from the first set by the number of matching counterparts in the second set, and starts the algorithm with the element that has least matching counterparts.

5.1.3.1 Test Setup

The test setup is made up of two Siena servers and three Siena clients. The setup deployment is presented in the Figure 5.1.

Figure 5.1: Composite bag operator performance test setup



The roles of the test components are the following:

Servers

- **Master Server** is a Siena master server.
- **Sub-Server** is a Siena server connected to the Master Server.

Clients

- **Client1** publishes notifications with bags. Each notification contains an attribute with name x containing a randomly generated bag. The randomly

[1] For some bag B, $|B|$ denotes the cardinality, or the number of elements, of the bag B.

generated bag is of random size between 1 and $2n$. Bag elements are randomly generated integers from set $\{0, \ldots, 9\}$. In addition, the Client1 is also able to publish special markup notifications with an attribute named x containing a bag of size n, which elements are all integers of value 10. Markup notifications are used for time measurement and match any subscription filter of Client2 and Client3.

- **Client2** subscribes for notifications using a filter utilizing a composite bag operator. A filter contains a filtering constraint of the format $x \supseteq_{\geqslant} B$, where B is a randomly generated bag of size n. Elements of the bag are randomly generated integer values from set $\{0, \ldots, 9\}$.
- **Client3** subscribes for notifications using a filter utilizing a composite bag operator as well. The format of the filter is equal to the format of the filter described above.

During a single measurement Client1 first publishes a markup notification, then it publishes 250 randomly generated notifications, and finally another markup notification. Client2 and Client3 both measure the elapsed time between the first and second markup notification. The first markup notification was always the first delivered notification and the last markup notification was always the last, or the last but one delivered notification. The latter out of order delivery is considered insignificant, since it affects only one notification.

Single measurements are combined into measurement sets. Each measurement set is made up of 20 repetitions of a single measurement. The final results of the measurement set are minimum, median and maximum elapsed time needed to transfer 250 random notifications.

Remark. The performance of the composite bag operator could be much better evaluated by using a benchmark for CBN platforms proposed by Keeney et al. (2006a). This benchmark is particularly suitable for assessing the ability of CBN platforms to deal with semantically enriched data. This work still remains to be done in the future.

5.1.3.2 Measurements

Measurement sets with the original algorithm were performed for different values of parameter $n \in \{3, \ldots, 9\}$ that affects the bag size. The results are presented in the Table 5.1 and in the Figure 5.2 (note that elapsed time is presented on a logarithmic scale).

Likewise measurement sets with the optimized algorithm were performed for different values of parameter $n \in \{9, \ldots, 15\}$. Note that it was possible to use bigger bags in this case. The results are presented in the Table 5.2 and in the Figure 5.3 (note that elapsed time is presented on a logarithmic scale).

Table 5.1: Original composite bag operator algorithm performance results

	Elapsed time (milliseconds)		
n	*Min*	*Median*	*Max*
3	450	550	640
4	440	550	690
5	450	680	830
6	550	720	890
7	530	670	1,050
8	1,820	4,540	13,590
9	780	3,750	415,050

Figure 5.2: Original composite bag operator algorithm performance results

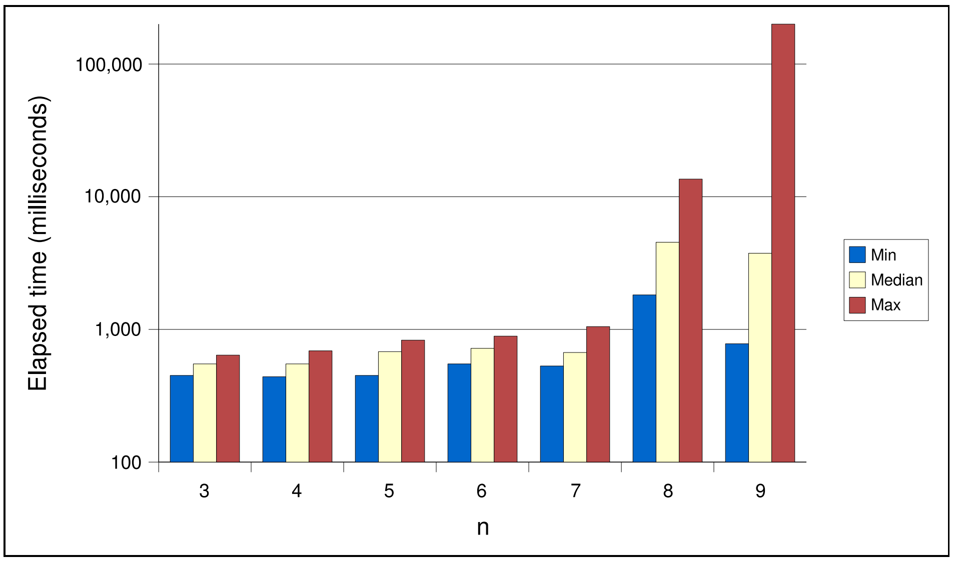
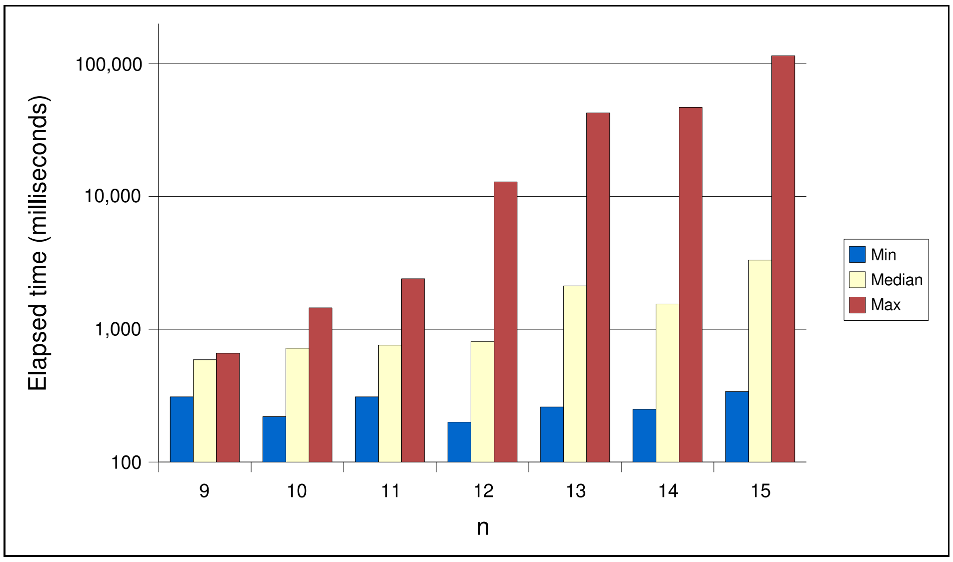


Table 5.2: Optimized composite bag operator algorithm performance results

	Elapsed time (milliseconds)		
n	*Min*	*Median*	*Max*
9	310	590	660
10	220	720	1,450
11	310	760	2,400
12	200	810	12,880
13	260	2,120	42,580
14	250	1,550	46,890
15	340	3,320	114,750

Figure 5.3: Optimized composite bag operator algorithm performance results



5.1.3.3 Interpretation of Results

The measured results with the original algorithm are in line with the characteristics of the brute force algorithm used to compare bags with the composite bag operator presented in § 4.1.3.

In the most optimistic case the algorithm finds matching elements immediately. In this case the time complexity of the algorithm is $O(|B|)$ for same bags B and C where $|B| \leqslant |C|$.

In the most pessimistic case the algorithm must fully match all possible arrangements of the elements of the smaller bag with the elements of the bigger set. In this case the time complexity is:

$$O\left(\frac{|C|!}{(|C|-|B|)!}\right) < O\left(|C|^{|B|}\right)$$

The measurement results reflect the time complexity of the algorithm. Minimum measured times are very short and correspond to the optimistic case. Maximum measured times grow very fast and correspond to the pessimistic case.

The simple optimization applied improved the performance of the algorithm significantly. While empirically the optimized algorithm manifested much better performance, theoretically its worst time complexity is still the same as with the original algorithm. The optimization only makes the worst case extremely unlikely to occur in this measurement scenario. For this reason the maximum elapsed times are much better with the optimized algorithm.

Remark. We can see from the formulas above that the time complexity depends above all on the size of the smaller bag, because the size of the smaller bag appears in the exponent of the time complexity. This fact was also empirically confirmed although the corresponding measurement scenario is not presented in this report.

Remark. The bags used in this evaluation have integer elements, which are compared with the sub-operator *greather then or equal to.* Alternatively, when using extended Siena, bags over ontological concepts could be used together with an ontological sub-operator. This would increase all measured delays approximately by a constant factor, but it should not affect the ratios between the measured values.

5.1.3.4 Discussion

It is evident that the algorithm for composite bag operators implemented in the scope of this research is truly useful only for bag comparisons where at least one of the bags is small (for example, of cardinality less then or equal to 12 for the optimized version of the algorithm).

It is possible to develop much more effective comparison algorithms for certain specialized composite bag operators. For example, it was suggested to exploit the ordering of the elements to improve the algorithm.

For composite bag operators over integer bags, where the sub-operator is one of the $<$, $\leqslant$, $>$ and $\geqslant$, it is possible to develop a very effective comparison algorithm if bags are pre-ordered. This is because the set of all integers is a totally ordered set[2] with regard to $\leqslant$ or $\geqslant$. Such an algorithm would have a very low time complexity of $O(max(|B|, |C|))$.

Unfortunately not all sets are totally ordered. For example, ontological concepts form only a partially ordered set[3] with regard to *subsumes.* An effective algorithm for composite bag operators over partially ordered sets could be a subject of the future research (see § 6.2.2.2).

Alternatively, there might exist some constraints that narrow the set of all possible bags that are to be published in notifications or subscription filters, to some subset, for which good performance of the composite bag operator could be guaranteed.

5.2 Decentralized Service Discovery and Execution

This section addresses evaluation of the decentralized semantic Web service discovery and execution model. Such evaluation requires verification of the following properties:

- functional correctness,
- robustness, especially in a dynamic environment,
- performance, and
- scalability.

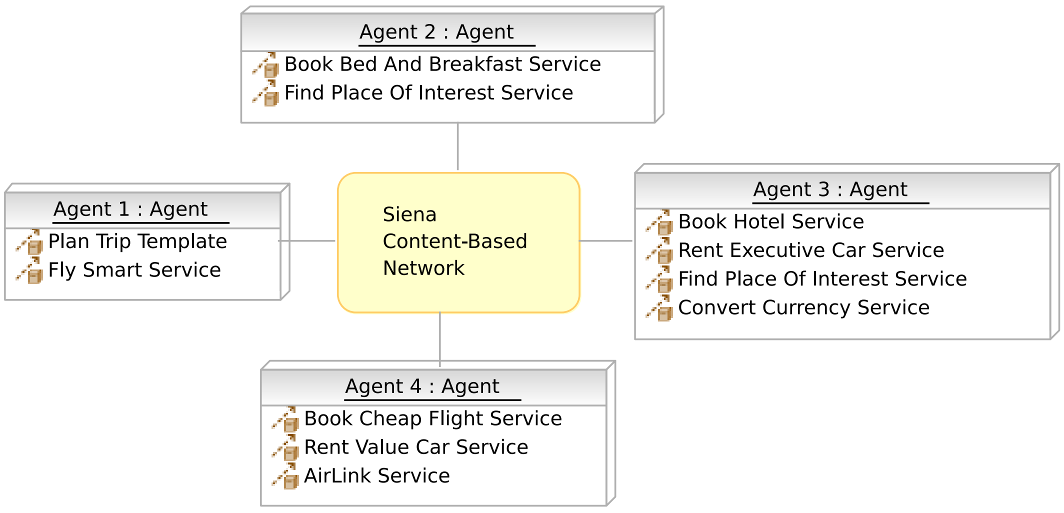
Due to time limitations this evaluation is limited to functional correctness and basic robustness and performance evaluation. The rest remains to be done in some future work.

[2] The definition of the totally ordered set is available at (Weisstein, 2000).
[3] The definition of the partially ordered set is available at (Weisstein, 2004).

5.2.1 Test Example

For the purposes of evaluation a flight and accommodation booking service based upon the similar example presented by Benatallah et al. (2002) has been defined and set up. The example, referred to as the *Plan Trip*, is comprised of the OWL-S composite service template PlanTripTemplate and a number of OWL-S atomic services. All atomic services have a grounding provided by dummy standard non-semantic Web services. The Figure 5.4 depicts an example deployment schema with four agents connected to the Siena CBN.

Figure 5.4: An example of the Plan Trip deployment schema



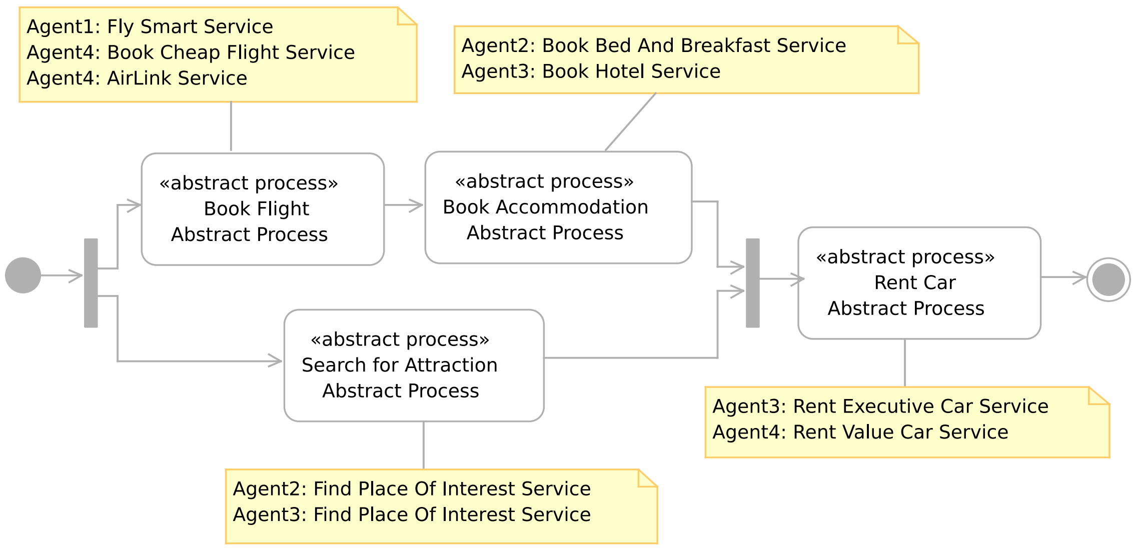
Agent1 hosts the composite service template PlanTripTemplate. The PlanTripTemplate organizes the following abstract processes into a workflow: BookFlightAbstractProcess, BookAccommodationAbstractProcess, SearchForAttractionAbstractProcess and RentCarAbstractProcess.

The PlanTripTemplate's composite process and the available services that match constituent abstract processes are depicted in Figure 5.5.

5.2.2 Service Discovery

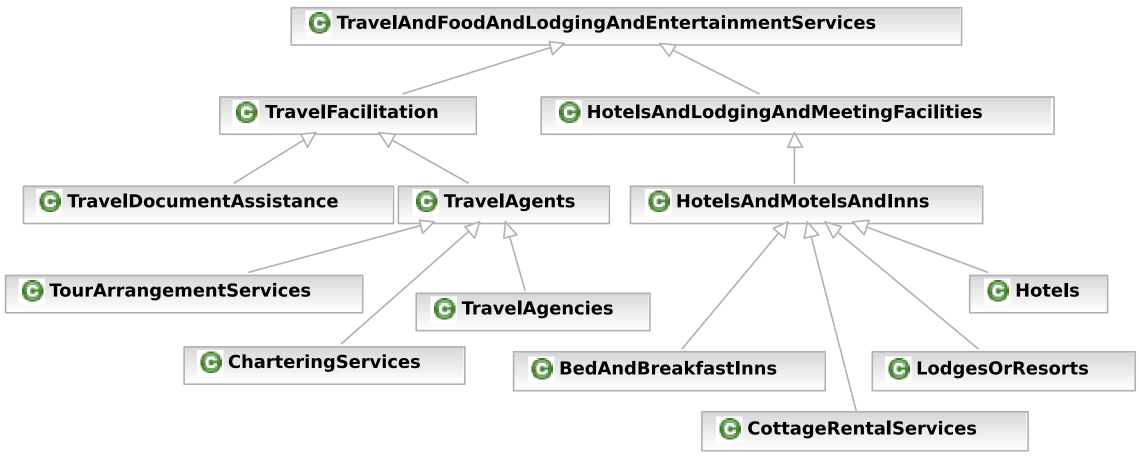
The objective of Agent1 is to discover matching executable services for all of the abstract processes of PlanTripTemplate's composite process. The discovery process is initiated by the submitting the service subscriptions to the Siena CBN. That happens immediately at agent startup or when a service is deployed or redeployed. When at least one executable service is discovered for each abstract process, the agent is able to accept invocation requests for the PlanTripTemplate. During the invocation the agent does not coordinate the execution of the composite process, but it only selects and binds appropriate service providers, initiates the start of the execution and waits for the completion. The process itself is executed in a decentralized peer-to-peer fashion described in the subsequent section.

Figure 5.5: PlanTripTemplate's composite process and available matching services


Agent1: Fly Smart Service
Agent4: Book Cheap Flight Service
Agent4: AirLink Service
Agent2: Book Bed And Breakfast Service
Agent3: Book Hotel Service
«abstract process» Book Flight Abstract Process
«abstract process» Book Accommodation Abstract Process
«abstract process» Rent Car Abstract Process
«abstract process» Search for Attraction Abstract Process
Agent3: Rent Executive Car Service
Agent4: Rent Value Car Service
Agent2: Find Place Of Interest Service
Agent3: Find Place Of Interest Service

Let's take a closer look at the subscriptions and notifications related to the abstract process BookAccommodationAbstractProcess. The taxonomy used to annotate service profiles and abstract processes is presented in the Figure 5.6, and the taxonomy used to annotate the parameters is presented in the Figure 5.7. The former is a small subset of the United Nations Standard Products and Services Code taxonomy (UNSPSC, 2006).

Figure 5.6: Service stereotypes taxonomy



The summary of the BookAccommodationAbstractProcess is presented in the Figure 5.8 and the listing of its OWL-S source is available in the Appendix A.

The Figure 5.9 shows the Siena service subscription filter for the BookAccommodation-AbstractProcess.

Now let's consider the BookBedAndBreakfastService deployed on the Agent2. The service is presented by its profile BookBedAndBreakfastProfile. The Figure 5.10 shows the summary of the BookBedAndBreakfastProfile description. The entire listing of its OWL-S source is available in the Appendix B. The Figure 5.11 shows a Siena service notification for the BookBedAndBreakfastService.

Figure 5.7: Parameter stereotypes taxonomy

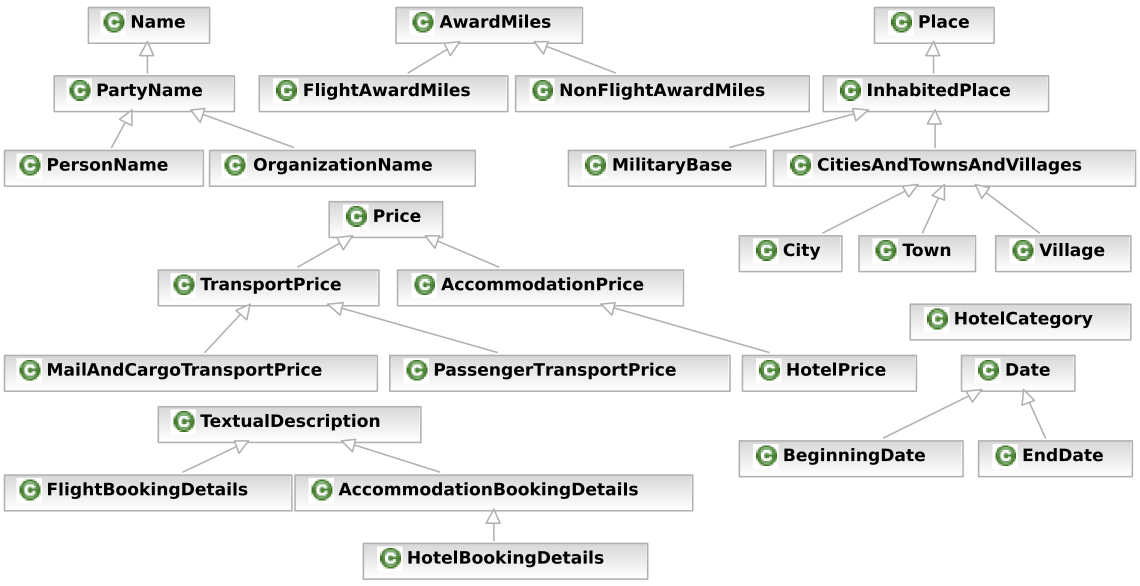


Figure 5.8: Summary of BookAccommodationAbstractProcess

	Name	*Data Type*	*Stereotype*
Abstract Process			"HotelsAndMotelsAndInns"
Inputs	name city arrivalDate departureDate preferedStars	string string date date integer	"PersonName" "City" "BeginningDate" "EndDate" "HotelCategory"
Outputs	accommodationPrice bookingDetails	double string	"Price" "TextualDescription"

Figure 5.9: Service subscription filter for BookAccommodationAbstractProcess

Attribute Name	*Type*	*Operator*	*Value*
serviceStereotype	string	$\dashv\!\vdash$	"HotelsAndMotelsAndInns"
inputStereotypes	bag	$\subseteq_{\Vdash}$	"PersonName" "City" "BeginningDate" "EndDate" "HotelCategory"
outputStereotypes	bag	$\supseteq_{\dashv\vert}$	"Price" "TextualDescription"

When comparing the BookBedAndBreakfastService notification against the BookAccommodationAbstractProcess subscription filter it can be seen that the filter selects the notification, because all its constraints are true.

The names of parameters, data types, stereotypes and the number of parameters in the BookAccommodationAbstractProcess and the BookBedAndBreakfastProfile are deliberately not the same to demonstrate the power and flexibility of the proposed discovery model. By looking at the definition of the matching rules in § 3.5.3.2 it can be seen that inputs match in spite of the fact that the BookAccommodationAbstractProcess can provide five inputs while the BookBedAndBreakfastService has only four.

Figure 5.10: Summary of BookBedAndBreakfastProfile

	Name	*Data Type*	*Stereotype*
Profile			"BedAndBreakfastInns"
Inputs	personName city fromDate toDate	string string date date	"PersonName" "CitiesAndTownsAndVillages" "BeginningDate" "EndDate"
Outputs	price bookingDetails	integer string	"AccommodationPrice" "AccommodationBookingDetails"

Figure 5.11: Service notification for BookBedAndBreakfastService

Attribute Name	*Type*	*Value*
serviceUri	string	"http://kdeg/BookBB.owl#BookBBService"
agentUrl	string	"http://kyi-2/dowls/"
serviceStereotype	string	"BedAndBreakfastInns"
inputStereotypes	bag	"PersonName" "CitiesAndTownsAndVillages" "BeginningDate" "EndDate"
outputStereotypes	bag	"AccommodationPrice" "AccommodationBookingDetails"
validity	integer	300

Now let's look at the BookHotelService deployed on Agent3, which is presented by its profile BookHotelProfile. Again the Figure 5.12 shows the summary of the BookHotelProfile description, and a corresponding Siena service notification is shown in the Figure 5.13. Also in this case the BookAccommodationAbstractProcess subscription filter selects the BookHotelService notifications, because all its constraints are true. Also in this case the names of parameters, stereotypes and the number of parameters are different from those in the BookAccommodationAbstractProcess and the BookBedAndBreakfastProfile. For example, the BookHotelService is able to provide more outputs than required by the BookAccommodationAbstractProcess.

Figure 5.12: Summary of BookHotelProfile

	Name	*Data Type*	*Stereotype*
Profile			"Hotels"
Inputs	name city arrivalDate departureDate numberOfStars	string string date date integer	"PartyName" "City" "BeginningDate" "EndDate" "HotelCategory"
Outputs	hotelPrice reservationDetails awardMiles	integer string integer	"HotelPrice" "HotelBookingDetails" "NonFlightAwardMiles"

Figure 5.13: Service notification for BookHotelService

Attribute Name	*Type*	*Value*
serviceUri	string	"http://kdeg/BookHotel.owl#BookHotel"
agentUrl	string	"http://kyi-4/dowls/"
serviceStereotype	string	"Hotels"
inputStereotypes	bag	"PartyName" "City" "BeginningDate" "EndDate" "HotelCategory"
outputStereotypes	bag	"HotelPrice" "HotelBookingDetails" "NonFlightAwardMiles"
validity	integer	300

In the presented scenario the Agent1, which hosts the PlanTripTemplate, discovers two executable services matching the BookAccommodationAbstractProcess. Analogically it discovers executable services for other abstract processes.

Once there is at least one executable service for each of the abstract processes discovered, the Agent1 is able to start advertising the PlanTripTemplate as an executable service, and also to start decentralized execution of the PlanTripTemplate provided the invocation request arrives.

5.2.3 Decentralized Execution

When the agent that hosts the PlanTripTemplate receives an invocation request from a client, it creates a new instance of decentralized service wrapper. The wrapper first selects discovered services that can fulfill the requirements of the abstract processes. Let's presume that the following services are selected: BookCheapFlightService, BookHotelService, FindPlaceOfInterestService and RentValueCarService.

Remark. If there is more than one executable service candidate for an abstract process, then the agent can apply a scoring function to select one. The scoring function could be based on either semantic similarity, or some other features, like reliability or cost of the service. More detailed investigation of the scoring function is out of the scope of this research and a random function is used to select the service in the prototype.

To understand the distributed data-flow during decentralized execution the input and output parameters of the participating services must be presented. The input and output parameters of the PlanTripTemplate are shown in the Figure 5.14 and the input and output parameters of the selected services are shown in the Figure 5.15.

Figure 5.14: PlanTripTemplate's input and output parameters

Service	*Inputs*	*Outputs*
PlanTripTemplate	name fromCity toCity minDepartureDate maxDepartureDate minReturnDate maxReturnDate hotelStars	departureDate returnDate flightPrice flightDetails accommodationPrice accommodationDetails attraction carRentalPrice carRentalDetails

After the wrapper selects the discovered services that can fulfill the requirements of the constituent abstract processes, it creates a state coordinator for every executable state of the composite process, and provisions it at the peer agent that hosts the associated discovered service. It also provisions the initial and final state coordinator at its own agent. Finally the wrapper initiates the decentralized execution by sending a message to the initial state coordinator. The course of events unfolding during decentralized execution is shown in the Figure 5.16. After the completion of execution the wrapper collects the results from the final state coordinator and returns them to the client.

By looking at the messages sent during the decentralized execution it can be seen that the data-flow messages always precede the control-flow messages. This is necessary because all input data of a state must be available at the moment when a control-flow notification triggers the execution of the associated service.

5.2.4 Experiments

A series of experiments were conducted to investigate the system's functional correctness, robustness and performance.

Figure 5.15: Input and output parameters of the selected services

Service	*Inputs*	*Outputs*
BookCheapFlightService	name fromCity toCity minDepartureDate maxDepartureDate minReturnDate maxReturnDate	departureDate returnDate price flightDetails
BookHotelService	name city arrivalDate departureDate numberOfStars	hotelPrice reservationDetails awardMiles
FindPlaceOfInterestService	place	attraction
RentValueCarService	name city firstDay lastDay	amount details

5.2.4.1 Functional Correctness

The proper behavior of the system, as described above, has been empirically confirmed by observing (i) the inputs and outputs of the PlanTripTemplate executions and (ii) the logging outputs of the participating agents. Since a random scoring function was used to select the matching services, if more than one matching service was available, it was possible to observe different selections of the participating services in consecutive executions of the PlanTripTemplate.

5.2.4.2 Robustness

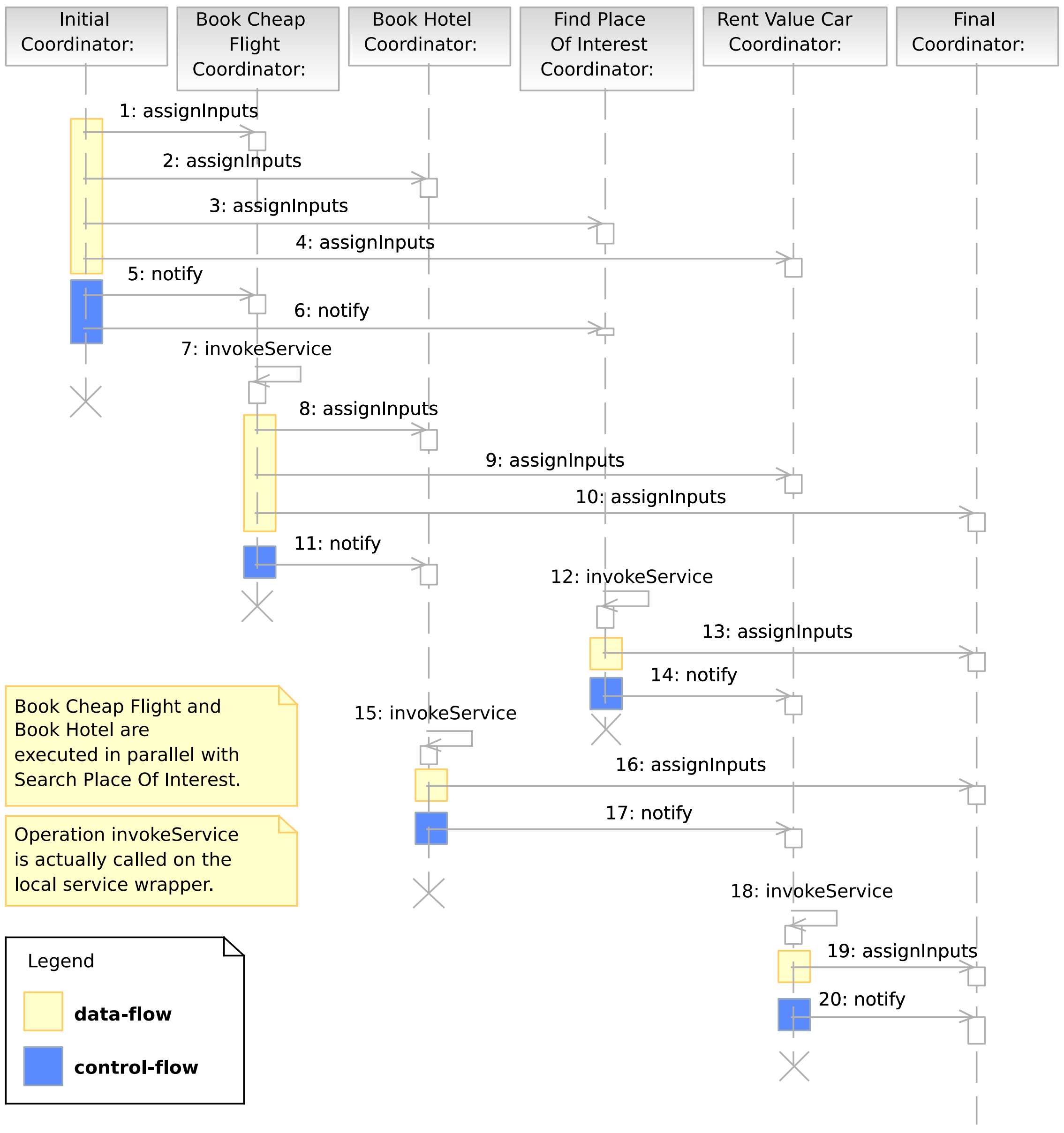
In addition, the robustness of the system has been tested by performing the following actions at run-time:

- undeploying one or more services at a random time,
- deploying one or more services to random agents at a random time,
- shutting down or killing an agent at a random time,
- starting up an agent at a random time.

While performing these actions the system manifested the following behavior:

- The system can adapt to and remain stable after any combination of the actions described above.
- In case, when the agent hosting the PlanTripTemplate is restarted, it may need some time to discover all suitable services. This is because services are advertised

Figure 5.16: Decentralized execution of PlanTripTemplate



periodically. The maximum time needed to discover an available service is limited by the duration of its advertisement period. This phenomenon is termed *warm-up time*.

- Except of this initial warm-up time, the system is able to perform decentralized execution of the PlanTripTemplate as long as there exists at least one live matching service for each constituent abstract process.
- In case, when the agent hosting atomic services is (re)started, its services are available to the agent hosting the PlanTripTemplate immediately. This is because the agent starts publishing advertisements for the deployed advertisable executable services immediately at the startup.
- If the agent participating in a decentralized execution becomes unavailable during this execution, an error or a timeout can occur. This depends upon whether the participating agent already completed its job in decentralized execution or not at the moment of becoming unavailable. In the former case the unavailability of the agent does not affect the decentralized execution and in the latter case it causes an error or a timeout.

5.2.4.3 Performance

Due to time limitations the performance of the system was assessed only briefly by comparing single decentralized executions of the PlanTripTemplate to its equivalent centralized executions implemented by the OWL-S composite service PlanTripService. All agents were running locally on a single computer. For each scenario the average elapsed time of 20 consecutive executions was calculated. The results are shown in the Table 5.3.

Table 5.3: Composite service execution performance results

	Average elapsed time (milliseconds)
Centralized execution	1,620
Decentralized execution	1,040

The results are surprising, since decentralized execution was expected to be slower in this simplified scenario, because of the fixed decentralized orchestration overhead. Decentralized execution was expected to perform better when scaling up to complex workflow, and to many parallel executions in a highly distributed system, since it reduces message communication and balances the load among peer agents.

A possible explanation for these results could be that in the design presented here the decentralized composite service is compiled into decentralized abstract states already at the composite service deployment time. The OWL-S Java API library, which was used for the execution of the centralized composite service, might still need to do some compiling work after the invocation of the composite service, which can cause its bad performance.

A better assessment of the system's performance involving a real distributed scenario with parallel invocations of composite services remains to be done by some future work.

5.2.4.4 *Discussion*

An experimental evaluation of the system demonstrated correctness of the system, its robustness and the ability of adapt to changing availability of peer agents and their hosted services.

The strong point of the discovery design presented here is that the service consumer agent has a rather good knowledge of the discovered services at any moment. When a request for decentralized execution arrives the agent does not need to query for matching services, but can proceed with the execution immediately. The drawbacks related to it are that (i) the service advertisement generates some constant network traffic and that (ii) the discovered services might become stale. The weak point of the design is exposed when the agent participating in the decentralized execution becomes unavailable during that execution.

As a consequence it can be concluded that this design is suitable for dynamic environments with a high service churn rate, as long as the ratio between service availability time and service execution time remains high.

Remark. An alternative advertisement and discovery design, with the trade-offs different from those of this design, is briefly outlined in § 6.2.4.2. It does not suffer from the warm-up time phenomenon and some of the drawbacks described above.

Chapter 6

Conclusion

This chapter presents the contributions of this research and gives some suggestions for further work. The main ideas presented in this work are general and not restrained to particular technology choices.

6.1 Contributions of the Research

6.1.1 Composite Bag Relations

One of the primary contributions of this research is the definition of the *composite bag relation*. The composite bag relation extends bag algebra with a new concept, which is particularly useful when combined with ontological relations (for example *subsumes*, *subsumed by*, *equivalent*). Compared to simple bag relations (which are $\supseteq$, $\subseteq$, $=$), composite bag relations make possible looser comparisons of bags. They allow for "inexact" matches that would not be possible should we use only simple bag relations.

Since a *set* is a special case of a *bag*, the composite bag operator is applicable to sets as well.

6.1.2 Bag Extension for Content-Based Networking

The second major contribution of this research is the design of a *bag extension* for content-based networking. The bag extension supports:

(i) bag attribute values,
(ii) simple bag operators (which are $\supseteq$, $\subseteq$, $=$), and
(iii) composite bag operators (which are an implementation of the theory of composite bag relations).

For effective routing the covering relationships for subscription aggregation between bag operators have been identified as well.

The combination of the composite bag operator and ontological operators in the CBN offers a much higher expressiveness for subscribing and filtering than current CBNs. This makes the CBN particularly suitable for disseminating ontologically described information.

6.1.3 OWL-S Extended with Stereotypes

Another minor contribution of this work is the definition of the *stereotypes* that are used to annotate OWL-S profiles, abstract processes, inputs and outputs for the purposes of semantic advertisement, discovery and selection. Stereotype is an OWL property that semantically categorizes the annotated resource according to some hierarchical taxonomy. The conception is similar to some existing properties of OWL-S profile. What makes stereotypes different is that they are also applied to abstract processes, inputs and outputs. Semantically annotated OWL-S profile presents service capabilities. Semantically annotated abstract process presents service consumer requirements.

6.1.4 Semantic Service Discovery Utilizing Composite Bag Operators

An important contribution of this research is the design of dynamic, scalable semantic service advertisement and discovery model based on the semantic CBN with bag support. The presented model is in a way similar to some exiting models, for example to the one described by Lynch (2005). The difference is that the model presented here effectively utilizes composite bag operators in CBN filtering constraints, and uses stereotypes to describe the provided and required services. This combination enables very expressive semantic service matching that fully exploits covering relationships between Siena operators, which makes it scalable to high dimensions. The conceptions presented are not specific to semantic services and can be applied to advertisement and discovery of any semantically described resources.

6.1.5 Data-Flow in Decentralized Composite Service Execution

Another contribution of this research is the design of the *distributed data-flow routing* for decentralized executions of composite services, which is not covered by Benatallah et al. (2001, 2002, 2003). Decentralized data-flow routing is described by structures called *data bindings*, which together with preconditions and post-processing actions form distributed routing tables for decentralized workflow executions.

6.1.6 Decentralized Discovery and Execution for Composite Services

Last but not least, an autonomous multi-agent design for decentralized discovery and execution of OWL-S composite services has been developed. While similar architectures for decentralized execution of composite services have been presented before, this is presumably the first one that supports composite semantic Web services. The design integrates all the above presented contributions.

It is based upon loosely coupled peer-to-peer system of software agents. The agent is an autonomous software component, which is able to host, advertise and execute atomic and composite semantic Web services. It is also able to discover services advertised by other agents and collaborate with other agents in decentralized execution of composite services. The service bindings are made and remade dynamically, so that each execution of the same composite service might involve other participating component services.

The presented design is robust and able to adapt to changing availability of peer agents and their hosted services. The decentralized execution is expected to scale better then the centralized counterpart since it reduces message communication and balances the load among peer agents. The architecture of the design is suitable for dynamic distributed environments with a high service churn rate, as long as the ratio between service availability time and service execution time remains high.

6.2 Further Work

6.2.1 Composite Bag Relation

A future research should investigate the applicability of the composite bag relation in ontology query languages. For example, SPARQL (W3C, 2006) could be extended with a composite bag query operator.

6.2.2 Bag Extension for Content-Based Networking

6.2.2.1 Benchmarking Bag Extension

A future work should thoroughly evaluate the performance of CBN bag extension. Keeney et al. (2006a) have developed a benchmark for knowledge based context delivery platforms, that is particularly suitable for assessing the ability of CBN platforms to deal with semantically enriched data.

6.2.2.2 Algorithm for Composite Bag Operator

In order to improve the efficiency of the composite bag operator a design of an effective composite bag operator matching algorithm should be investigated, perhaps by narrowing the applicability of the algorithm to certain specialized composite bag operators. For example, it is possible to develop a very effective algorithm for bag operators over totally ordered sets, if the sub-operator of the composite bag operator respects the total order (see § 5.1.3.4).

Unfortunately, not all sets are totally ordered. For example, ontological concepts form only a partially ordered set with regard to *subsumes*. A future research could try to develop an effective algorithm for composite bag operators over partially ordered sets.

6.2.3 Toward the Knowledge-Based Networking

The ontologically extended Siena with bag operators could be further extended with richer ontology-based operators, for example, with arbitrary OWL relationships. Furthermore, a research is needed for embedding the semantic interoperability into the knowledge-based networks.

6.2.3.1 Potential Applications of the Knowledge-Based Networking

This research has only just begun to explore application for the expressiveness of the knowledge-based networking. Potential applications are:

- discovery and change notification of policies between federated communication service providers,
- sensor readings in a multi-domain heterogeneous ubiquitous computing application,
- RSS with semantic markup in Web 2.0/Semantic Web,
- notifications from heterogeneous network elements in OSS,
- semantic routing of multimedia (MPEG) stream with semantic meta-data.

6.2.3.2 Extensible Control Plane for Knowledge-Based Networking

The knowledge-based network implementation will be used in the new Science Foundation Ireland project call MECON (Managed Extensible Control Plane for Knowledge-Based Networking), evaluating its performance with different reasoners in OPNET simulations and for applications running on PlanetLab. This has the aim of identifying optimal semantic clustering strategies to enable scalability of a knowledge-based networking by restricting ontology load and forwarding table sizes at individual nodes.

6.2.4 Semantic Service Discovery

6.2.4.1 Extending OWL-S Discovery Annotations

The OWL-S discovery annotations design presented in this work is not a fully developed solution, but merely an illustration of the idea. This work has defined a property called dowls:hasStereotype applicable to OWL-S profiles, abstract processes, and their inputs and outputs. dowls:hasStereotype could be extended further to preconditions and effects. In addition, the discovery annotation itself could be elaborated much further from being just a single property. For example, a more elaborated discovery annotation of parameters in abstract processes could designate whether the parameter is mandatory or optional.

6.2.4.2 Alternative Semantic Service Discovery Design

The proposed architecture for decentralized semantic Web service discovery maintains a local repository of discovered services at each service consumer agent. The agent that hosts a decentralized composite service template decomposes the service into constituent abstract processes and subscribes for matching executable services that can fulfill the requirements of the abstract processes. The agents that host advertisable executable services periodically multicast service advertisement notifications. The CBN effectively routes those service notifications to the agents that expressed interest in them.

This approach has some benefits and some drawbacks. The benefit of this approach is that the service consumer agent has a rather good knowledge of the discovered services at any moment. When a request for decentralized execution arrives the agent does not need to query for matching services, but can proceed with the execution immediately. The drawbacks of this approach are that (i) the service advertisement generates some constant network traffic and that (ii) the discovered services might become stale.

An alternative semantic service discovery design without these deficiencies, but with some others, that could be a subject of some future research, is briefly outlined here. The alternative approach could be applied to the service discovery as follows:

1. Queries for required services are wrapped into CBN notifications.
2. The service providers that host advertisable executable services subscribe for matching queries.
3. When the agent that hosts a decentralized composite service template needs to invoke the service, it multicasts a query notification for each of the service's constituent abstract processes.
4. The CBN effectively routes the queries toward the service providers that are able to answer the query.
5. Once the service provider receives the query it responds to the service consumer with the information about the matching service.

This alternative design has several benefits, but also some drawbacks. The benefits are:

- Agents do not need to maintain the local repository of discovered services.
- The overall network load is lower than in the original design, providing that there are not too many active service consumers. In the original design notifications are published periodically, while in the alternative design query notifications are published only when needed.
- In contrast to the original design the discovered services are much less likely to be stale at the moment of a decentralized composite service invocation.

The drawbacks of the alternative design are:

- The provisioning phase of a decentralized execution is much longer, since it includes multicasting the queries and waiting for the respones.
- The service consumer can also not know, how long it should wait for the responses. Even if it receives some respones in a very short time there is always a chance that a better response is yet to arrive.

Remark. It would be very easy to developed this design using semantic CBN with the bag extension. Actually the format of a query notification could be similar to the format of a service notification from the original approach, by using inversions of the operators from the original approach.

6.2.5 Decentralized Execution of Compose Semantic Web Service

6.2.5.1 Preconditions and Effects

The prototype developed in scope of this research does not implement conditions, preconditions and effects. These are fundamental constructs of service compositions and hence they should be addressed by a future work. In addition, a research is needed to investigate how the external context gathered via the knowledge-based network could be used in precondition evaluation during service execution.

6.2.5.2 Monitoring, Error Handling and Transactions

This work does not investigate monitoring of the decentralized enactment of a composite semantic Web service. It also does not specifically deals with errors and other exceptional situations. Since monitoring and error handling are both crucial for any business process executions, a future work should research these issues and investigate the applicability of the knowledge-based network.

The error, that occurs during execution, might cause the involved resources and the environment to be left in an inconsistent state after the completion of the execution. This could be avoided by providing support for transactions. Hence a future research should investigate support for transactions in decentralized executions of composite services.

References

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.-T., Sheth, A., and Verma, K. (2005). Web Service Semantics – WSDL-S 1.0, W3C Member Submission. World Wide Web Consortium (W3C) – Web Resource. Retrieved August 19, 2006, from http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services: Concepts, Architecture and Applications*. Springer Verlag.

Antoniou, G. and van Harmelen, F. (2004). *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA.

Apache Software Foundation (2006a). Agila Wiki. Apache Software Foundation – Web Resource. Retrieved August 22, 2006, from http://wiki.apache.org/agila/.

Apache Software Foundation (2006b). Apache ServiceMix. Apache Software Foundation – Web Resource. Retrieved August 20, 2006, from http://servicemix.org/.

Baeten, J. and Basten, T. (2001). Handbook of Process Algebra. *Handbook of Process Algebra. Elsevier, North-Holland*.

Benatallah, B., Dumas, M., Fauvet, M.-C., and Paik, H.-Y. (2001). Self-Coordinated and Self-Traced Composite Services with Dynamic Provider Selection. Technical report, School of Computer Science & Engineering, University of New South Wales.

Benatallah, B., Sheng, Q. Z., and Dumas, M. (2003). The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 07(1):40–48.

Benatallah, B., Sheng, Q. Z., Ngu, A. H., and Dumas, M. (2002). Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. *icde*, 00:0297.

Buchmann, A. P., Bornhövd, C., Cilia, M., Fiege, L., Gärtner, F. C., Liebig, C., Meixner, M., and Mühl, G. (2004). *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*, chapter DREAM: Distributed Reliable Event-Based Application Management, pages 319–352. Springer.

Cabral, L., Domingue, J., Motta, E., Payne, T. R., and Hakimpour, F. (2004). Approaches to Semantic Web Services: an Overview and Comparisons. In *European Semantic Web Symposium*, pages 225–239.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383.

Carzaniga, A., Rutherford, M., and Wolf, A. (2004). A Routing Scheme for Content-based Networking. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 2:918–928.

Carzaniga, A. and Wolf, A. L. (2002). Content-Based Networking: A New Communication Infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, London, UK. Springer-Verlag.

Chen, Q. and Hsu, M. (2001). Inter-Enterprise Collaborative Business Process Management. In *Proceedings of the 17th International Conference on Data Engineering*, pages 253–260, Washington, DC, USA. IEEE Computer Society.

Codehaus (2006). Mule. Codehaus – Web Resource. Retrieved August 20, 2006, from http://mule.codehaus.org/.

de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., König-Ries, B., Kopecký, J., Lara, R., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., and Stollberg, M. (2005). Web Service Modeling Ontology (WSMO), W3C Member Submission. W3C – Web Resource. Retrieved August 19, 2006, from http://www.w3.org/Submission/WSMO/.

Dustdar, S. and Schreiner, W. (2005). A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30.

Fensel, D. and Bussler, C. (2002). The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137.

Heimbigner, D. (2003). Extending the Siena Publish/Subscribe System. Technical Report CU-CS-946-2003, University of Colorado at Boulder, Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309-0430, USA.

IBM et al. (2005). Business Process Execution Language for Web Services Version 1.1. Technical report, IBM.

Kalogeras, A., Gialelis, J., Alexakos, C., Georgoudakis, M., and Koubias, S. (2006). Vertical Integration of Enterprise Industrial Systems Utilizing Web Services. *Industrial Informatics, IEEE Transactions on*, 2(2):120–128.

Keeney, J., Carey, K., Lewis, D., OSullivan, D., and Wade, V. (2005). Ontology-based Semantics for Composable Autonomic Elements. In *Proceedings of the Workshop of AI in Autonomic Communications at the Nineteenth International Joint Conference on Artificial Intelligence*, Edinburgh, Scotland.

Keeney, J., Lewis, D., and OSullivan, D. (2006a). Benchmarking Knowledge-based Context Delivery Systems. In *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 06)*, Silicon Valley, USA.

Keeney, J., Lynch, D., Lewis, D., and O'Sullivan, D. (2006b). On the Role of Ontological Semantics in Routing Contextual Knowledge in Highly Distributed Autonomic Systems. Technical Report TCD-CS-2006-15, University of Dublin, Trinity College.

Klein, M. (2002). DAML+OIL and RDF Schema Representation of UNSPSC. Department of Computer Science, Vrije Universiteit, Amsterdam – Web Resource. Retrieved August 21, 2006, from http://www.cs.vu.nl/ mcaklein/unspsc/.

Lara, R., Polleres, A., Lausen, H., Roman, D., de Bruijn, J., and Fensel, D. (2005). Conceptual Comparison between WSMO and OWL-S, WSMO Final Draft. Technical Report D4.1 v0.1, Digital Enterprise Research Institute (DERI).

Lewis, D., O'Sullivan, D., Power, R., and Keeney, J. (2005). Semantic Interoperability for an Autonomic Knowledge Delivery Service. In *Proceedings of the Second IFIP TC6 International Workshop on Autonomic Communication (WAC 2005)*, pages 129–140, Vouliagmeni, Athens, Greece.

Li, S. (2003). JXTA 2: A High-Performance, Massively Scalable P2P Network. IBM developerWorks – Web Resource. Retrieved September 6, 2006, from http://www-128.ibm.com/developerworks/java/library/j-jxta2/.

Lynch, D. (2005). A Proactive Approach to Semantically Oriented Service Discovery. Master's thesis, University of Dublin, Trinity College.

Lynch, D., Keeney, J., Lewis, D., and O'Sullivan, D. (2006). A Proactive Approach to Semantically Oriented Service Discovery. In *Proceedings of the Second Workshop on Innovations in Web Infrastructure (IWI 2006).* Co-located with the 15th International World-Wide Web Conference, Edinburgh, Scotland. May 2006.

Martin, D., Burstein, M., Denker, G., Hobbs, J., Kagal, L., Lassila, O., McDermott, D., McIlraith, S., Paolucci, M., Parsia, B., Payne, T., Sabou, M., Sirin, E., Solanki, M., Srinivasan, N., and Sycara, K. (2004a). Profile-based Class Hierarchies, Explanatory Remarks for ProfileHierarchy.owl, OWL-S 1.1. OWL-S Coalition – Web Resource. Retrieved August 18, 2006, from http://www.daml.org/services/owl-s/1.2/ProfileHierarchy.html.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K. (2004b). OWL-S: Semantic Markup for Web Services, W3C Member Submission. World Wide Web Consortium (W3C) – Web Resource. Retrieved July 19, 2006, from http://www.w3.org/Submission/OWL-S/.

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., and Sycara, K. (2004c). Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, San Diego, California, USA.

McIlraith, S., Son, T., and Zeng, H. (2001). Semantic Web Services. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 16(2):46–53.

Meier, R. and Cahill, V. (2005). Taxonomy of Distributed Event-based Programming Systems. *The Computer Journal*, 48(5):602–626.

Milanovic, N. and Malek, M. (2004). Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59.

OASIS (2004). UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. Technical report, OASIS.

OASIS (2006). Reference Model for Service Oriented Architecture 1.0, Committee Specification 1. Technical Report soa-rm-cs, Organization for the Advancement of Structured Information Standards (OASIS).

ObjectWeb Consortium (2006). Celtix. ObjectWeb Consortium – Web Resource. Retrieved August 20, 2006, from http://celtix.objectweb.org/.

O'Sullivan, D. and Lewis, D. (2003). Semantically Driven Service Interoperability for Pervasive Computing. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 17–24, New York, NY, USA. ACM Press.

Papazoglou, M. P. and van den Heuvel, W.-J. (2005). Service Oriented Architectures: Approaches, Technologies and Research Issues. To appear in VLDB Journal.

Piccinelli, G. and Williams, S. L. (2003). Workflow: A Language for Composing Web Services. In *Business Process Management*, pages 13–24.

Pietzuch, P. and Bacon, J. (2002). Hermes: A Distributed Event-based Middleware Architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618.

Pokraev, S., Koolwaaij, J., and Wibbels, M. (2003). Extending UDDI with Context-Aware Features Based on Semantic Service Descriptions. In *Proceedings of the International Conference on Web Services, ICWS '03*, pages 184–190.

Roy, J. and Ramanujan, A. (2001). Understanding Web Services. *IT Professional*, 3(6):69–73.

Rutherford, M. J. (2004). Siena Simplification Library Documentation 1.1.4. University of Colorado – Web Resource. Retrieved August 13, 2006, from http://serl.cs.colorado.edu/ carzanig/siena/forwarding/ssimp/namespacesiena.html.

Sadiq, S., Orlowska, M., Sadiq, W., and Foulger, C. (2004). Data Flow and Validation in Workflow Modelling. In *ADC '04: Proceedings of the fifteenth Australasian database conference*, pages 207–214, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Schmidt, C. and Parashar, M. (2004). A peer-to-peer approach to web service discovery. *World Wide Web*, 7(2):211–229.

Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. (2000). Content Based Routing with Elvin4. *Proceedings AUUG2K, Canberra, Australia.*

Shen, J., Yan, J., and Yang, Y. (2006). SwinDeW-S: Extending P2P Workflow Systems for Adaptive Composite Web Services. *aswec*, 0:61–69.

Sirin, E. and Parsia, B. (2004). The OWL-S Java API. In *Third International Semantic Web Conference (ISWC2004) Poster*, Hiroshima, Japan.

Sirin, E., Parsia, B., and Hendler, J. (2004). Filtering and Selecting Semantic Web Services with Interactive Composition Techniques. *IEEE Intelligent Systems*, 19(4):42–49.

Sirin, E., Parsia, B., and Hendler, J. (2005). Template-based Composition of Semantic Web Services. In *AAAI fall symposium on agents and the semantic web*, Virginia, USA.

Sun Microsystems (2006a). Jini Network Technology. Sun Microsystems, Inc. – Web Resource. Retrieved September 7, 2006, from http://www.sun.com/software/jini/.

Sun Microsystems (2006b). Jini Specifications Archive - v2.1. Sun Microsystems, Inc. – Web Resource. Retrieved September 7, 2006, from http://java.sun.com/products/jini/2_1index.html.

Sun Microsystems (2006c). JXTA. Sun Microsystems, Inc. – Web Resource. Retrieved September 6, 2006, from http://www.jxta.org/.

Traversat, B., Abdelaziz, M., and Pouyoul, E. (2002). Project JXTA: A Loosely-Consistent DHT Rendezvous Walker. Technical report, Sun Microsystems, Inc.

Traversat, B., Arora, A., Abdelaziz, M., Duigou, M., Haywood, C., Hugly, J.-C., Pouyoul, E., and Yeager, B. (2003). Project JXTA 2.0 Super-Peer Virtual Network. Technical report, Sun Microsystems, Inc.

UNSPSC (2006). The United Nations Standard Products and Services Code Homepage. The United Nations Standard Products and Services Code (UNSPSC) – Web Resource. Retrieved August 19, 2006, from http://www.unspsc.org/.

W3C (2001). Web Services Description Language (WSDL) 1.1, W3C Note. Technical report, W3C.

W3C (2003). SOAP Version 1.2 Part 0: Primer, W3C Recommendation. Technical report, W3C.

W3C (2004a). OWL Web Ontology Language Overview, W3C Recommendation. W3C – Web Resource. Retrieved August 19, 2006, from http://www.w3.org/TR/owl-features/.

W3C (2004b). RDF Primer, W3C Recommendation. Technical report, W3C.

W3C (2004c). RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation. Technical report, W3C.

W3C (2004d). Web Services Architecture Requirements, W3C Working Group Note. Technical report, W3C.

W3C (2006). SPARQL Query Language for RDF, W3C Candidate Recommendation. W3C – Web Resource. Retrieved August 19, 2006, from http://www.w3.org/TR/rdf-sparql-query/.

Weisstein, E. W. (2000). Totally Ordered Set. MathWorld – A Wolfram Web Resource. Retrieved September 8, 2006, from http://mathworld.wolfram.com/TotallyOrderedSet.html.

Weisstein, E. W. (2002). Multiset. MathWorld – A Wolfram Web Resource. Retrieved July 19, 2006, from http://mathworld.wolfram.com/Multiset.html.

Weisstein, E. W. (2004). Partially Ordered Set. MathWorld – A Wolfram Web Resource. Retrieved September 8, 2006, from http://mathworld.wolfram.com/PartiallyOrderedSet.html.

Zhou, J., Dialani, V., Roure, D. D., and Hall, W. (2003). A Semantic Search Algorithm for Peer-to-Peer Open Hypermedia Systems. In *Proceedings of the 1st Workshop on Semantics in Peer-to-Peer and Grid Computing at the 12th International World Wide Web Conference*, Budapest, Hungary.

Appendix A

Listing A.1: OWL-S description of BookAccommodationAbstractProcess

```
<!DOCTYPE rdf:RDF [
    <!ENTITY abstractBookAccommodation "http://kdeg/AbstractBookAccommodation.owl" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns" >
    <!ENTITY owl "http://www.w3.org/2002/07/owl" >
    <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl" >
    <!ENTITY dowls "http://kyi/~dominik/dowls-demo/ontologies/Dowls.owl" > ]>

<rdf:RDF
    xmlns="&abstractBookAccommodation;#"
    xml:base="&abstractBookAccommodation;#"
    xmlns:abstractBookAccommodation="&abstractBookAccommodation;#"
    xmlns:xsd="&xsd;#"
    xmlns:rdf="&rdf;#"
    xmlns:owl="&owl;#"
    xmlns:process="&process;#"
    xmlns:dowls="&dowls;#">

<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="&process;" />
    <owl:imports rdf:resource="&dowls;" />
</owl:Ontology>

<dowls:AbstractProcess rdf:ID="BookAccommodationAbstractProcess">
    <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
        &dowls;#HotelsAndMotelsAndInns
    </dowls:hasStereotype>

    <process:hasInput>
        <process:Input rdf:ID="name">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#PersonName
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="city">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#City
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="arrivalDate">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#dateTime
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#BeginningDate
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="departureDate">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#EndDate
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#EndDate
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="preferedStars">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#integer
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#HotelCategory
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>

    <process:hasOutput>
        <process:Output rdf:ID="accommodationPrice">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#double
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#Price
            </dowls:hasStereotype>
        </process:Output>
    </process:hasOutput>
    <process:hasOutput>
        <process:Output rdf:ID="bookingDetails">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#TextualDescription
            </dowls:hasStereotype>
        </process:Output>
    </process:hasOutput>
</dowls:AbstractProcess>

</rdf:RDF>
```

Appendix B

Listing B.1: OWL-S description of BookBedAndBreakfastService

```
<!DOCTYPE rdf:RDF [
    <!ENTITY bookBedAndBreakfast "http://kdeg/BookBedAndBreakfast.owl" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns" >
    <!ENTITY owl "http://www.w3.org/2002/07/owl" >
    <!ENTITY service "http://www.daml.org/services/owl-s/1.1/Service.owl" >
    <!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl" >
    <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl" >
    <!ENTITY grounding "http://www.daml.org/services/owl-s/1.1/Grounding.owl" >
    <!ENTITY dowls "http://kyi/~dominik/dowls-demo/ontologies/Dowls.owl" >
    <!ENTITY wsdl "http://kyi:8080/dowls-demo-ws/services/touristAgencyService?wsdl" >
    <!ENTITY touristAgency "http://demo.dowls.kdeg.cs.tcd.ie/services/touristAgency" > ]>

<rdf:RDF
    xmlns="&bookBedAndBreakfast;#"
    xml:base="&bookBedAndBreakfast;#"
    xmlns:bookBedAndBreakfast="&bookBedAndBreakfast;#"
    xmlns:xsd="&xsd;#"
    xmlns:rdf="&rdf;#"
    xmlns:owl="&owl;#"
    xmlns:service="&service;#"
    xmlns:profile="&profile;#"
    xmlns:process="&process;#"
    xmlns:grounding="&grounding;#"
    xmlns:dowls="&dowls;#"
    xmlns:wsdl="&wsdl;#"
    xmlns:touristAgency="&touristAgency;#">

<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="&service;" />
    <owl:imports rdf:resource="&profile;" />
    <owl:imports rdf:resource="&process;" />
    <owl:imports rdf:resource="&grounding;" />
    <owl:imports rdf:resource="&dowls;" />
</owl:Ontology>

<service:Service rdf:ID="BookBedAndBreakfastService">
    <service:presents rdf:resource="#BookBedAndBreakfastProfile" />
    <service:describedBy rdf:resource="#BookBedAndBreakfastProcess" />
    <service:supports rdf:resource="#BookBedAndBreakfastGrounding" />
</service:Service>

<profile:Profile rdf:ID="BookBedAndBreakfastProfile">
    <service:presentedBy rdf:resource="#BookBedAndBreakfastService" />
    <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
        &dowls;#BedAndBreakfastInns
    </dowls:hasStereotype>

    <profile:hasInput rdf:resource="#personName" />
    <profile:hasInput rdf:resource="#city" />
    <profile:hasInput rdf:resource="#fromDate" />
    <profile:hasInput rdf:resource="#toDate" />

    <profile:hasOutput rdf:resource="#price" />
    <profile:hasOutput rdf:resource="#bookingDetails" />
</profile:Profile>

<process:AtomicProcess rdf:ID="BookBedAndBreakfastProcess">
    <service:describes rdf:resource="#BookBedAndBreakfastService" />

    <process:hasInput>
        <process:Input rdf:ID="personName">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#PersonName
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="city">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#CitiesAndTownsAndVillages
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="fromDate">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#dateTime
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#BeginningDate
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="toDate">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#dateTime
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#EndDate
            </dowls:hasStereotype>
        </process:Input>
    </process:hasInput>

    <process:hasOutput>
        <process:Output rdf:ID="price">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#integer
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#AccommodationPrice
            </dowls:hasStereotype>
        </process:Output>
    </process:hasOutput>
    <process:hasOutput>
        <process:Output rdf:ID="bookingDetails">
            <process:parameterType rdf:datatype="&xsd;#anyURI">
                &xsd;#string
            </process:parameterType>
            <dowls:hasStereotype rdf:datatype="&xsd;#anyURI">
                &dowls;#AccommodationBookingDetails
            </dowls:hasStereotype>
        </process:Output>
    </process:hasOutput>
</process:AtomicProcess>

<grounding:WsdlGrounding rdf:ID="BookBedAndBreakfastGrounding">
    <service:supportedBy rdf:resource="#BookBedAndBreakfastService" />
    <grounding:hasAtomicProcessGrounding rdf:resource="#BookBedAndBreakfastAtomicProcessGrounding" />
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="BookBedAndBreakfastAtomicProcessGrounding">
...
</grounding:WsdlAtomicProcessGrounding>

</rdf:RDF>
```