A Proactive Approach to Semantically Oriented Service Discovery

by

David Lynch, B.A. (Mod)

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfillment

of the requirements

for the Degree of

Master in Science of Computer Science

University of Dublin, Trinity College

September 2005

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.

David Lynch

September 9, 2005

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

David Lynch

September 9, 2005

Acknowledgments

I would like to thank my dissertation supervisor, Declan O'Sullivan, for all his help and support over the period in which this dissertation was researched and written. I would also like to thank my NDS classmates for help, support and for lightening the mood in serious times. Lastly, I would like to thank my family for their continued support over the course of my school and college years.

DAVID LYNCH

University of Dublin, Trinity College
September 2005

A Proactive Approach to Semantically Oriented Service Discovery

David Lynch

University of Dublin, Trinity College, 2005

Supervisor: Declan O'Sullivan

Web Service based computing has evolved immensely in recent years, supported by standards bodies, academic research and industry alike. One of the noticeable omissions from the web services architecture is that of standards to support automatic discovery, automatic composition and invocation of web services. While semantic web service discovery implementations exist research into autonomous semantically oriented service discovery is far from an efficient, complete solution to the problem.

Current web-service capability matching research and implementation focuses on semantically enhancing the UDDI standard for web-service discovery. While this is definitely a step in the right direction, dependence on the UDDI standard may restrain capability matchers as they mature. A prime example of this is the request/response nature of a UDDI look-up.

This dissertation researches an alternative approach to web service discovery that proactively informs interested participants of the availability of new services that match expressed capability requirements. By evaluating and semantically enhancing a wide area notification system this dissertation develops a scalable publish/subscribe platform for OWL-S service discovery that incorporates efficient content-based routing and an expressive subscription language.

Our evaluation shows that many of the documented optimisations for text-centric content based routing actually hold in routing for more complex OWL-based concepts. We conclude that a publish/subscribe model for semantic service discovery is feasible and can potentially

provide a pro-active discovery environment for human and autonomous agents alike. However, in order to realise this vision, much research in the area of distributed knowledge base consistency, ontology alignment and intelligent sharing of OWL information is needed.

Contents

Acknowledgments **iv**

Abstract **v**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Web Services 1

1.2 Semantically Annotated Web Services 2

1.3 The Publish/Subscribe Model 3

1.3.1 Content Based Routing 3

1.4 Research Objectives 3

1.4.1 A Semantic Alternative to UDDI 4

1.4.2 A Publish/Subscribe Model for Web-Service Discovery 4

1.4.3 Efficient, OWL-S aware Content-Based Routing 4

1.4.4 Ontology Alignment 5

1.5 Dissertation Outline 5

Chapter 2 Background **6**

2.1 The Semantic Web 6

2.1.1 RDF 7

2.1.2 Web Ontology Language - OWL 7

2.1.3 Reasoning with OWL 9

2.2 Web Services 10

2.2.1 UDDI 10

2.2.2 WSDL and SOAP . . . 11
2.3 Semantic Web Service Descriptions - OWL-S . . . 11
2.3.1 An Upper Ontology for Web Services . . . 12
2.3.2 Service Profile . . . 13
2.3.3 Process Model . . . 14
2.3.4 Service Grounding . . . 14
2.4 Publish / Subscribe . . . 15
2.4.1 Content Based Routing . . . 16
2.4.2 Siena . . . 16
2.4.3 Elvin . . . 20

Chapter 3 The State of The Art **22**
3.1 Semantic Service Discovery . . . 22
3.1.1 UDDI and Service Discovery . . . 22
3.1.2 Semantically Enhanced UDDI . . . 23
3.1.3 OWL-S Matchmaker . . . 23
3.1.4 METEOR-S for Service Discovery . . . 24
3.1.5 Ranked Matching . . . 24
3.1.6 OWL-S API . . . 25
3.1.7 Reasoning for OWL-S . . . 25
3.2 Publish/Subscribe . . . 27
3.2.1 Siena . . . 27
3.3 Ontology Alignment . . . 28
3.3.1 Ontology Alignment API . . . 29
3.4 Conclusions . . . 29

Chapter 4 Design **30**
4.1 Overview . . . 30
4.2 Application Description . . . 32
4.2.1 Usage Scenario . . . 33
4.2.2 Activity Diagrams . . . 34
4.3 Enhancements . . . 39
4.3.1 Subscription Language . . . 39

4.3.2 The Subscription Poset 41
4.3.3 Subscription Insertion 44
4.3.4 Subscription and Publication Forwarding 45
4.3.5 Ontology Mappings 46

Chapter 5 Implementation **49**
5.1 Technology Review 49
5.1.1 OWL-S Matcher 49
5.1.2 Siena Java Implementation 51
5.2 Exploratory Implementation 52
5.3 The Enhanced Prototype 55
5.3.1 OWL-S Support Package 56
5.3.2 Communications Layer 58
5.3.3 Matching Integration 59
5.3.4 Subscription (Client) 59
5.3.5 Subscription (Server) 61
5.3.6 Publication (Client) 63
5.3.7 Publication (Server) 63
5.4 Observations 64

Chapter 6 Testing and Evaluation **67**
6.1 Testing 67
6.2 Evaluation Focus 68
6.2.1 Siena Evaluation 68
6.2.2 Further Evaluation 70
6.3 Evaluation Scenario 71
6.4 Test Results 71
6.5 Evaluation Conclusion 73

Chapter 7 Conclusions **74**
7.1 Research Conclusions 74
7.1.1 A Semantic Alternative to UDDI 74
7.1.2 A Publish/Subscribe Model for Service Discovery 74
7.1.3 Efficient, OWL-S aware Content Based Routing 75

7.1.4 Ontology Alignment 75
7.2 Future Work 75
7.2.1 Evaluation 75
7.2.2 Content Based Routing 76
7.2.3 Ontology Integration and Alignment 76
7.3 Final Remarks 77

Bibliography **78**

List of Figures

2.1 An RDF example 7
2.2 An OWL wine ontology excerpt 8
2.3 An application of the abstract OWL Wine class 8
2.4 An Upper Ontology for Web Services [9] 12
2.5 An OWL-S Service Profile 13
2.6 OWL-S to WSDL grounding 15
2.7 A Siena filter. *From [7]* 16
2.8 The Siena Notification/Subscription Covering Relation. *From [7]* 17
2.9 The Subscription Filter Poset. *From [7]* 18
2.10 A Hierarchical Topology. *From [7]* 19
2.11 An OWL-S Process Model 21

4.1 Design Overview 31
4.2 The original and modified Siena Interfaces 32
4.3 Use Case Actors 33
4.4 Plan View 34
4.5 Server Subscription 36
4.6 Server Publication 37
4.7 Client Subscription 38
4.8 A usage scenario 39
4.9 The relation constraint **C** covers constraint **C'** 42
4.10 The Enhanced Poset Structure 43
4.11 Sevice Profile Covering 44
4.12 Insertion Optimisation 44
4.13 A distributed example of subscription forwarding 45

4.14 Two aspects of ontology mappings 47
4.15 Wine to Vino Ontology Mapping 48

5.1 First prototype service notification 54

6.1 Total Network Cost 68
6.2 Cost Per Service 69
6.3 Two Node Test Scenario 71
6.4 Single Siena Node Scenario 72

Chapter 1

Introduction

The research presented here is concerned with the web services paradigm and, in particular, web service discovery. Much has been made of the slow progress in the area of service discovery and overall aim of this research is to contribute a novel body of work to this field. The purpose of this chapter is to; first of all, introduce the web services paradigm and some of the research gaps associated with the field. Following on from this, four research aims and are introduced and motivations behind these aims discussed.

1.1 Web Services

Web Service based computing has evolved immensely in recent years, supported forcefully by standards bodies, such as OASIS [42] and the W3C [16], industry, such as Microsoft [38], IBM [37] and Hewlett-Packard [25], and a plethora of academic research. Web Services, by definition, are self-contained, self-describing applications that can be published, located and invoked remotely and in a dynamic fashion over the Internet [18]. This loosely-coupled remote-service invocation capability has proven a particularly attractive proposition for e-commerce and business integration models. Following this large research push, an interoperability set of standards has been approved and cross-platform, cross-enterprise integration is becoming a reality.

One of the key success factors for the up-take of Web services technology is its focus on implementing cross-organisation communications using already matured and widely implemented protocols, such as HTTP. Three vital standards have been agreed upon to finalise

data-formatting, remote invocation and service description; respectively, eXtensible Markup Language - XML [23], Simple Object Access Protocol - SOAP [11] and Web Services Description Language - WSDL [8]. Complimenting these, Universal Description, Discovery and Integration, or UDDI [13] is an OASIS standard means of locating, publishing and, to a lesser extent, invoking such services.

Popular web services, which provide full WSDL definitions, include the Amazon API [36], The Google Maps API [35] and the excellent E-Bay Web services API [17]. The E-Bay API is a particularly useful example allowing advanced transaction histories, refunds processing and customer tracking all by means of a SOAP/XML based request-response dialogue over standard HTTP.

1.2 Semantically Annotated Web Services

One of the noticeable omissions from the web services architecture described previously is that of standards to support automatic discovery, automatic composition and invocation of web services. In their current form WSDL service descriptions and UDDI searches must be created or conducted with human intervention. A very desirable scenario is one whereby software agents can intelligently reason over required web-service functionality, automatically discover supporting modules and seamlessly integrate them on-the-fly into desired applications.

To realise this scenario, the fundamental absence of semantic information to complement the syntactic provisions of WSDL must be remedied. Indeed, it is the vision of creators of the Semantic Web [3]. Their goals are to better define web semantics in machine-terms so that intelligent agents may feasibly reason over Internet and Web concepts thus enhancing user experience of relevant information and interesting functionality.

Another stack of XML based languages, some not as-yet confirmed as standards, provide a framework for semantically annotating web elements with machine-understandable information. The Resource Description Framework or RDF [10] has paved the way for more complex, application specific semantic annotation standards such as The Ontology Web Language, OWL [15] and An Upper Ontology for Web Services description - OWL-S [9].

OWL-S semantically annotated descriptions is a big step towards enabling automatic invocation of services. The rise of OWL capable reasoners such as pellet [30], JESS [29] and RACER [24] has further accelerated the progress.

1.3 The Publish/Subscribe Model

The Publish/Subscribe model for communication also lies firmly in the scope of loosely-coupled, large-scale distributed systems [21]. The model consists of three basic elements; Subscribers, who express interest in particular information by means of a subscription language,publishers of information, who publish information of interest and an intermediary event notification service connecting the two. Subscribers are notified upon the publication of information which is of interest to them. Filters, which may be applied to specific content, decide if a subscriber should be notified of the publication of a particular object of interest. The asynchronous nature of the model and the general expressiveness of the subscription language are vital to its operability, functionality and scalability. The selective *pushing* of content towards interested subscribers may be viewed as an interesting alternative to explicit one-shot client-server based information retrieval. It is argued here and henceforth that there is potential to integrate such a model into a web services discovery application.

1.3.1 Content Based Routing

Content Based Routing [6] is a closely associated with the publish/subscribe model. Implementations such as Siena [7] and Elvin [34] take a content-based routing approach within their respective publish/subscribe implementations.

Content-Based Routing is distinguished from Address-Based routing by the fact that the information of interest is routed towards the destination in terms of the actual content rather than a simple destination address or a classification of contained information. Routing tables are constructed from end-user specified requirements (e.g. a subscription expressed in some language) and information to be routed navigates towards its destination through application of these requirements to the explicit content of the information published. CBR is a progression of broadcast, flooding and topic-based techniques applied to earlier publish/subscribe systems.

1.4 Research Objectives

Before pursuing any course of research it is necessary to clearly define research goals and discuss the motivations behind such goals.

1.4.1 A Semantic Alternative to UDDI

While the UDDI approach outlines a method for describing how web services function there is a fundamental lack of support for describing, in machine-understandable terms, the web services' capabilities. While work has been done in adding semantic capability to UDDI [39] [1] the focus has been on modifying the UDDI registry to accommodate OWL-S descriptions. Any modification of the UDDI in its current form still inherits poor support for large, loosely integrated wide-area registries of services. The first research objective of this dissertation is to explore the possibility of an alternative to UDDI that integrates semantic support from the design stage together with a loosely-coupled yet co-operative network of server nodes. The primary focus is the service discovery element of the UDDI standard.

1.4.2 A Publish/Subscribe Model for Web-Service Discovery

It is the opinion of the authors that discovery for web services fits naturally into the publish/subscribe paradigm. Currently, there are no standards or implementations that propose to proactively push web service descriptions towards interested parties, such as autonomous software agents. The second research objective of this dissertation is to explore the possibility of a distributed publish/subscribe based service discovery platform. This exploration will focus on enhancing the semantic capability of an existing publish/subscribe system, integrating a semantic web capability matching component and increasing the expressiveness that implementations subscription language to cater for requirements expression in the semantically enhanced environment.

1.4.3 Efficient, OWL-S aware Content-Based Routing

Since semantic capability, in the form of OWL support, will be added to an existing publish/subscribe system, it is a natural progression to explore whether the annotated semantic information can be utilised to assist in content-based routing of semantic service descriptions. It is postulated here that semantic information available from web service descriptions combined with semantics derived from a semantically-enabled subscription language, can help optimise the process of subscription matching and information routing. This research avenue will focus on modifying the structure of an existing subscription storage algorithm

[7] and the optimisations, existing and new, that arise upon reasoning over semantically enhanced subscriptions.

1.4.4 Ontology Alignment

Knowledge-base consistency and accuracy of concept matching becomes an important issue when considering a large scale distributed ontology constructed from different sources. The final research objective of this dissertation is to examine the issues that arise when considering multiple, possibly conflicting, OWL ontologies. In researching possible avenues in this field, the presence of an ontology mapping is assumed. As a result the primary focus is on expression and organisation of ontology mappings across the proposed distributed discovery architecture as well as integration of these mappings into the discovery service knowledge-base.

1.5 Dissertation Outline

Chapter 2 introduces the technologies, standards and tools used throughout research, implementation and evaluation as well as detailed background information on their usage. Chapter 3 explores the current state of the art including examination of academic research into semantic service discovery; publish/subscribe with content- based routing and ontology alignment, concluding with the envisaged contributions of this dissertation. Chapter 4 details a design for a Java implementation which will explore the feasibility of integrating existing tools into a solution realising research objectives. Chapter 5 outlines, in detail, how the design was implemented. Chapter 6 details an evaluation of the implementation in terms of research objectives, feasibility and other selected measures. Finally, Chapter 7 concludes the overall findings of the research conducted and outlines avenues for further work.

Chapter 2

Background

This chapter is intended as a precursor to the main dissertation. In order to fully understand the dissertation research, design and implementation it is felt that an introduction to the main technologies involved is essential. The first three sections of this chapter deal with web services, the web services Stack, The Semantic Web and the OWL and OWL-S languages. Section 2.4 handles the publish/subscribe model and content-based routing with particular emphasis on the approach taken by Siena [7], the chosen basis implementation for this dissertation.

2.1 The Semantic Web

The Semantic Web is intended as an extension of the web as it currently exists. Semantic Web aims to improve upon the meaning, in machine-understandable terms, of information currently available on the world-wide-web [3]. This enables computers, in the form of autonomous software agents, to work with the wealth of world-wide-web information more easily. Moreover, it enhances the human-computer co-operation by bringing the concept of human understanding closer to the machine. The W3C [16] have pushed forward standards to drive the vision of the semantic web. A stack of XML based languages, discussed in detail later, have been standardised. These languages, at various levels, provide mechanisms by which information about the target domain can be captured.

Description logic languages [2] such as OWL and RDF provide a means to capture conceptual information in graph form. Nodes in a graph are used to represent concepts and

arcs the relationships between these concepts. Once captured, intelligence algorithms may be applied and deductions derived autonomously from axioms and assertions provided. An Ontology is an explicit, formal specification of how to represent objects, concepts and relationships between objects and concepts in some target domain. The semantic web languages RDF and OWL are Ontology-based.

2.1.1 RDF

```
<class ID="Parent">
          <subClassOf resource="#Person">
</class>
```

Figure 2.1: An RDF example

RDF, or the Resource Description Framework [10], is an XML based ontology language used for expressing semi-structured meta-data. There is no in-built restriction on semantics but the triple-based syntactic structure of RDF allows applications to effectively extract potentially useful meta-information from a document. A triple consists of a class, property and value. Each class is considered a *thing*. An example listing is shown in figure 2.1. Here it can be deduced that a parent is a type of person. It is desirable, however, to express more sophisticated assertions. RDF has no support for complex data types for properties and semantic constraints on concepts defined as a class.

2.1.2 Web Ontology Language - OWL

The Web Ontology Language or OWL [15] extends the RDF language-schema addressing the shortcomings outlined above. OWL was originally a part of the DARPA project as DAML however having been integrated with OIL and submitted as a standard to the W3C, was renamed to OWL.

An OWL knowledge base is constructed in a similar fashion to an RDF knowledge base. A class hierarchy is defined and properties are assigned to class concepts. The power of OWL emerges when we consider how it improves upon the RDF language. Firstly, OWL expresses complex data-types and value restrictions on those data-types. Secondly, through

```
<owl:Class rdf:ID="Wine">
    <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasMaker" />
  <owl:allValuesFrom rdf:resource="#Winery" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#locatedIn"/>
        <owl:someValuesFrom rdf:resource="&vin;Region"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:label xml:lang="en">wine</rdfs:label>
    <rdfs:label xml:lang="fr">vin</rdfs:label>
  </owl:Class>
```

Figure 2.2: An OWL wine ontology excerpt

```
    <WhiteWine rdf:ID="StGenevieveTexasWhite">
    <locatedIn rdf:resource="#CentralTexasRegion" />
    <hasMaker  rdf:resource="#StGenevieve" />
    <hasSugar  rdf:resource="#Dry" />
    <hasFlavor rdf:resource="#Moderate" />
  </WhiteWine>
```

Figure 2.3: An application of the abstract OWL Wine class

use of OWL keywords complex relationships between classes and types can be defined. The figure 2.2 taken from the WC3 OWL tutorial [1] shows how the concept of wine may be captured in an OWL ontology.

OWL differentiates between the declaration of a concept and an instantiation of that concept. Essentially, OWL individuals are the extensional knowledge of an OWL knowledge base that serve as the application of the intentional knowledge defined by OWL structure keywords. Figure 2.3 illustrates the instantiation of an OWL individual.

[1] http://www.w3.org/TR/owl-features/

OWL provides a means by which equality and difference between semantic concepts can be expressed.
`owl:equivalentClass`, `owl:equivalentProperty` and `owl:sameAs` can each be used to express equivalence between two syntactically differing concepts. For example, the concept of wine the concept of vino can be considered as the same concept defined differently. By asserting the `Vino` class equivalent to `Wine` class using `owl:equivalentClass` the concepts become equivalent in the eyes of any reasoner and deductions are made accordingly. `owl:sameAs` works in the same fashion except it is applied to individuals, not classes. The difference operators `owl:differentFrom`, `owl:AllDistinct` , `owl:distinctMembers`, apply the inverse semantics to individuals and individuals which are declared part of collections.

OWL property characteristics are used to enrich the semantics available in terms of class properties. `owl:objectProperty` and `owl:datatypeProperty` define object types and data-types respectively. Assertions such as `owl:transitiveProperty`, `owl:SymmetricProperty` further enrich the semantics with assertions on related properties of classes. A for a fuller discussion and tutorial the reader is referred to the OWL section of the W3C website[15].

2.1.3 Reasoning with OWL

There are several tools available for reasoning with OWL knowledge-bases. JESS [29] rule engine, RACER [24] and Pellet [30] are three common rule-based tools used for querying knowledge base assertions. All of these tools have in-built support for OWL reasoning. The W3C OWL standard is presented in three flavours and provides different computability guarantees. OWL Lite is a subset of the OWL DL language. Similarly, OWL DL is a subset of OWL Full. The first two of the above omit certain semantic restrictions, such as multiple cardinality, in order to guarantee a certain level of computability. The above mentioned reasoners are capable of reasoning over at least the OWL DL language. In fact, most support OWL-DL plus certain features of OWL-DL full. It is assumed, unless explicitly stated, that any references to OWL will refer to the OWL-DL subset.

2.2 Web Services

The web services paradigm has already been introduced; however it is necessary to discuss the stack of standards and languages that lie at its core. Among these include support for abstract data-types and remote procedure call, message encoding, service description and binding as well as a platform for service advertisement and discovery.

2.2.1 UDDI

The Universal Description, Discovery and Integration protocol is the standard method for publishing and discovering the network based software components of a service oriented architecture [12]. A UDDI registry stores data and metadata about published web services allowing them to be discovered and consumed by applications across an enterprise or across the internet.

The UDDI information model is captured by an XML schema of which the elements `businessService`, `businessEntity`, `bindingTemplate` and `tModel` are used to capture information about the service and the service providers' business model in a structured way. `businessEntity` maintains the company info, `businessService` holds the name of the business entity including a unique business key and service key offering information about the services offered by a company, `bindingTemplate` provides information on how to invoke a service and the `tModel` provides further invocation details.

There are two interesting additions to this data model from UDDI versions 2 and 3. `publisherAssertion` is used to establish relationships among entities in the registry. A subscription is a standing request from a client of a UDDI registry node to notify the subscriber of any changes to any particular element, such as the tModel, of the associated metadata held in the UDDI registry. UDDI entities are classified according to a specific taxonomy. UDDI nodes can recognise multiple taxonomies of entities stored in its database.

There is an important distinction made between a UDDI node and a UDDI registry. A node is a server that supports at least the minimum functionality as defined by the specification and is a member of one UDDI registry. A registry, therefore, is composed of one or more nodes. Registries may also have affiliate registries which share information and a common UDDI namespace for identifying unique records across the network of nodes.

A UDDI registry lookup is a request/response dialog between the registry and an inter-

ested client. The UDDI publish-API is used to add and delete services, tModels, business entities, publisher assertions as well as binding templates for the particular service. Once registered, the potential client may search across the registry in terms of category, business entity or tModel in order to find a specific service.

This request/response dialogue, despite the addition of the subscription API to version 2 and 3 of the UDDI specification, is a look-up service that requires human intervention and cannot be accomplished autonomously by a software agent. Further discussion of this, and other weaknesses of the UDDI registry, is left until chapter 3.

2.2.2 WSDL and SOAP

Web Service Description Language, or WDSL, is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information[8]. The WSDL grammar allows web-services functionality, in terms of message data-types and functions, to be described abstractly. As a result applications have a standard, extensible means of describing functionality of a service regardless of data format or network protocol. In order to bind an abstract definition to a concrete technology it is necessary to extend the WSDL XML grammar. The most widespread WSDL binding is described as the SOAP 1.1 protocol which is often used conjunction with HTTP GET/POST and MIME.

SOAP, or Simple Object Access Protocol, is a lightweight protocol for exchange of information in a decentralized, distributed environment. [11]. Also an XML grammar, SOAP is divided into three core elements. Firstly, SOAP provides an envelope which defines a framework for describing message contents and how they may be processed. Secondly, the standard provides a set of encoding rules for instances of data-types which are defined at the application end-points. Lastly, SOAP provides a convention for remote request/response based procedure call. Again, SOAP may be used in combination with any protocol. Of these SMTP and HTTP are the most common.

2.3 Semantic Web Service Descriptions - OWL-S

While WSDL is capable of describing web services syntactically it is the drive of the semantic web that has encouraged the emergence of a web service description language that

annotates web service descriptions with machine understandable semantics. It is this addition of semantic content for web-service descriptions that is enabling automated service discovery, composition and invocation.

2.3.1 An Upper Ontology for Web Services

Taking advantage of the extensibility of XML, DAML [33] have submitted OWL-S [9], or OWL for web services, to the W3C. Recently accepted as a standard, the goal of OWL-S is to provide an upper ontology for web service descriptions. OWL-S integrates the capability of OWL to describe a concept in human terms and defines four ontologies which are used as a basis for providing structured, human and machine understandable information about a web service. The *service profile* ontology deals with advertising and discovery of services and the *process model* gives a detailed description of a service's operation. A typical OWL-S service description also includes a *grounding*, which is usually expressed as a binding to some WSDL definition. Figure 2.4 illustrates these ontologies and their relationship.

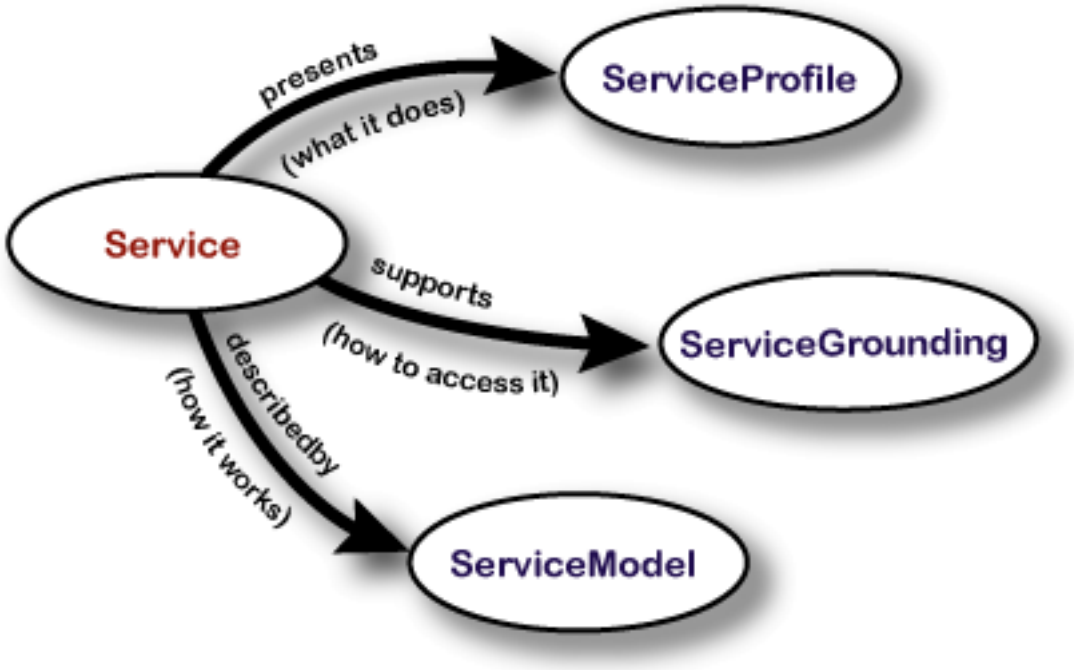

Figure 2.4: An Upper Ontology for Web Services [9]

2.3.2 Service Profile

```
<profile:Profile rdf:ID="BookDetailsFinderProfile">
<service:isPresentedBy rdf:resource="#BookDetailsFinderService"/>
  <profile:serviceName xml:lang="en">
Name
</profile:serviceName>
  <profile:textDescription xml:lang="en">
    Textual Description
</profile:textDescription>
  <profile:hasInput rdf:resource="#BookInfo"/>
  <profile:hasOutput rdf:resource="#BookResearcher"/>
  <profile:hasOutput rdf:resource="#BookPublisher"/>
  <profile:hasOutput rdf:resource="#BookAuthor"/>
</profile:Profile>
```

Figure 2.5: An OWL-S Service Profile

A *Service Profile* provides a way to describe the services offered to an interested or potentially interested client. There can be multiple service profiles for each service, as each service may be capable of performing several functions depending on input and output parameters. There are three major elements of the service profile: provider contact information, functional description and a range of quality factors which aid the intelligent agent, or human, in rating the quality of a service in terms of its requirements.

A profile is defined in terms of *Inputs*, *Outputs*, *Pre-conditions* and *effects* (IOPEs). Each IOPE is an OWL concept. An example functional description in a service profile is shown in figure 2.5. Here the `BookDetailsFinderService` presents a profile which returns to a user researcher, publisher and author information about a book. In addition to the functional information provided above, provider information consists of details about the entity that provides the service, for example, e-mail and telephone of customer support. The UNSPSC categorisation system may be used to categorise the given service via `owl:serviceCategory`. A `categoryName`, taxonomy value and code may be specified to aid this categorisation. Additionally, non-functional service parameters may be specified via `owl:serviceParameter`. Quality factors, confidence and other rating factors are examples of non-functional parameters that may be added to the service profile.

2.3.3 Process Model

The process model gives a detailed perspective on how the service operates. The OWL-S specification defines a process as an entity that transforms a set of outputs into a set of inputs or more generally; causes transition from one world state to another.

A process may have any number of inputs and any number of outputs. Similarly OWL-S supports processes with any number of pre-conditions. A process following on from the previously illustrated service profile is shown in figure 2.11 on page 21.

There are three types of process, atomic, simple and composite. Atomic processes are directly invokable and execute in one single step from the invokers perspective. Each atomic process is associated with a grounding. Simple Processes, in contrast, are not invokable. They are abstract views of some atomic process and may be expanded or realised by a concrete definition.

A composite process is one that may be decomposed into several atomic processes. Their decomposition and sequence of execution may be controlled by the sequence keyword and by an if-then-else control structure. This dissertation deals mainly with atomic processes defined with a set of inputs and outputs that are exposed by possibly multiple service profiles. Service composition, pre-conditions and effects are not considered and the reader is referred to [4] for a more detailed discussion in this area.

2.3.4 Service Grounding

Once the function of a service has been defined it is necessary to ground the service to a concrete realisation of the OWL concepts. Essentially a mapping from an abstract definition to a concrete one, the service grounding explicitly ties each parameter message to a concrete WSDL definition. There is a commonality between the OWL-S concept of grounding and the WSDL concept of binding. Figure 2.6 on page 15 shows an OWL-S to WSDL/SOAP grounding in terms of the operation definition and the input message to a book information service.

```
<grounding:portType>
http://www.winisp.net/cheeso/books/LookyBookServiceSoap
</grounding:portType>

<grounding:operation>
http://www.winisp.net/cheeso/books/DoKeywordSearch
</grounding:operation>
```

Figure 2.6: OWL-S to WSDL grounding

2.4 Publish / Subscribe

In tandem with the rise of service-oriented computing another paradigm, publish/subscribe, is receiving increased academic attention [21]. The fundamental idea behind the publish/-subscribe model is a simple one. *Subscribers* express interest in a particular *events* or *event patterns* and are *notified* subsequently of events *published* by any *publisher* that satisfy a set of *constraints* placed on events by the subscriber.

Subscribers and publishers are decoupled in space i.e. there is no requirement that subscribers and publishers know each other. Instead publishers and subscribers are aware of a common interface through which events are published and notifications are delivered. This interface, or publish/subscribe service provider, must track subscriptions and commonly provide the following functions.

- `Subscribe(Filter f)` : Allow interested parties express interest.
- `Unsubscribe(Filter f)` : Cancel expressed interest
- `Publish(Notification n)` : Publish Events to the Service
- `Advertise(Filter f)` : The Publisher indicates what type of events it may publish in future

Publishers and Subscribers are also decoupled in time and in synchronisation. As a result neither a publisher nor a subscriber need be active at time of publication or notification respectively. Similarly, a publisher is not blocked until notification and a subscriber may

receive notification by means of call-back an any unspecified time after publication has completed.

2.4.1 Content Based Routing

Earlier publish/subscribe implementations used a topic based approach to classify event content. Hierarchies of topics are constructed organising content into a hierarchy according to relationship. By subscribing to one topic, a subscription to a topic below the subscribed one in the hierarchy is implied. However, as cited in [21], despite the introduction of a wildcard matching scheme topic based routing systems were limited in expressiveness.

Content based routing schemes increase expressiveness by introducing routing based on the actual content of an event. A *subscription language* is used as a means of specifying *filters*. These filters describe *constraints* which are applied to content via basic comparison operators.

For example, if we wish to be notified of a situation where a share price reaches a threshold then we could specify `sharePrice = 50` and thus receive notification, including the share price and associated information, when any share price surpasses this threshold.

2.4.2 Siena

In choosing an event notification service to work towards the outlined research goals two freely available implementations were considered. Elvin [34] and Siena were examined. The availability of a Siena implementation and the abundance of associated technical reports and papers as well as its focus on expressiveness in a wide-area distributed environment resulted in Siena being the implementation of choice for this dissertation. A detailed description of Siena follows.

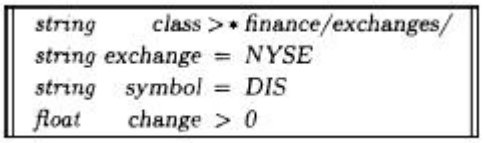

Figure 2.7: A Siena filter. *From [7]*

Notification		Subscription
string what = alarm *time* date = 02:40:03	$\prec_S^N$	*string* what = alarm
string what = alarm *time* date = 02:40:03	$\not\prec_S^N$	*string* what = alarm *integer* level > 3
string what = alarm *integer* level = 10	$\not\prec_S^N$	*string* what = alarm *integer* level > 3 *integer* level < 7
string what = alarm *integer* level = 5	$\prec_S^N$	*string* what = alarm *integer* level > 3 *integer* level < 7

Figure 2.8: The Siena Notification/Subscription Covering Relation. *From [7]*

Any centralised wide area notification service that may realise the requirements of a publish/subscribe implementation can be considered a nave one. Siena has been designed as a wide area event notification service that is both expressive and scalable. Siena takes a content based routing approach to providing the publish/subscribe service.

Primtives

A Siena notification is a set of typed attributes. Each attribute is a triple consisting of a type, name and a value. The type is one of `string, time`, `float` and `integer`. A **filter** is constructed from a set of **constraints** which are applied to content of **notifications**. A constraint is also a triple, consisting of the `attribute name`, a `constraint operator` and a `value`. Where multiple constraints exist they are evaluated as a conjunction. Siena uses a **covering relation** to express a matching between a filter-event pair. Specifically, a filter **covers** an event if that event satisfies each constraint applied to it by the content filter. Figure 2.7 on page 16 shows a complex Siena filter composed of constraints and figure 2.8 shows some examples of holding and non-holding covering relations.

Siena supports Subscriber based semantics and Advertisement based semantics. As the Siena Java implementation is subscriber based a discussion of advertisement based semantics is left to the reader [7].Under both approaches, a notification ***n*** is delivered to an interested party ***X*** if ***X*** has submitted a subscription (as a filter, or conjunction of filters) that covers the notification. Also, a filter ***f*** covers another filter ***f'*** where ***f*** constrains a superset of the notifications covered by ***f'***. This is an important relation when a subscription is considered

as a filter, and is exploited by Siena for efficiency in subscription matching and storage.

Efficency

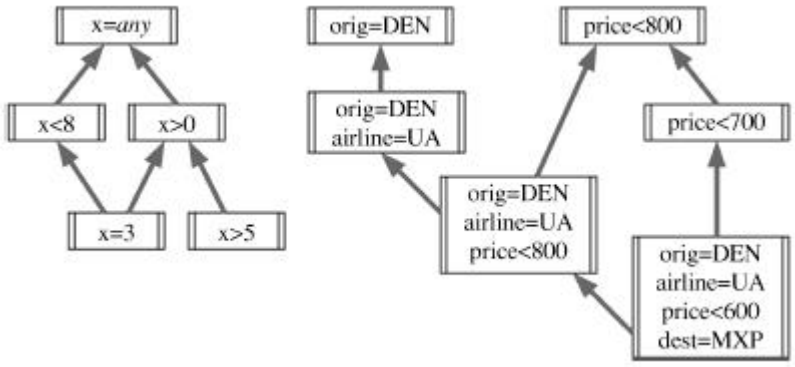

Figure 2.9: The Subscription Filter Poset. *From [7]*

Siena maintains a partially ordered data-structure that keeps track of subscriptions, previous requests and forwarding strategy. Each event server maintains its own copy of this **poset**. An example of this data structure, taken from [7], is shown in figure 2.9 on page 2.9. The covering relation defines arcs to child nodes in this data structure. Each subscription filter is either a root subscription, the case where there is no more general subscription that covers it, or is a subscription with an immediate successor. In both cases a subscription may also have an immediate predecessor. The covering relation, and therefore predecessor and successor relation is a transitive one. A vertically growing subscription poset is indicative of multiple subscriptions of a similar nature and in describing algorithms for the various topologies Siena optimises for this case.

[7] illustrates and evaluates three topologies for the event service. Hierarchical, acyclic peer-to-peer and cyclic peer-to-peer are all specified. However, implementations exist only for the first two and of those the hierarchical one is the least complicated. The hierarchical architecture is where Carizangia et. al begin and will be the subject of study in this dissertation. For a discussion of the peer-to-peer architectures the reader is referred to [7]and [6].

Each node in a hierarchical topology may have any number of incoming connections, other than clients, but only one outgoing connection to its parent node. Conceptually, the

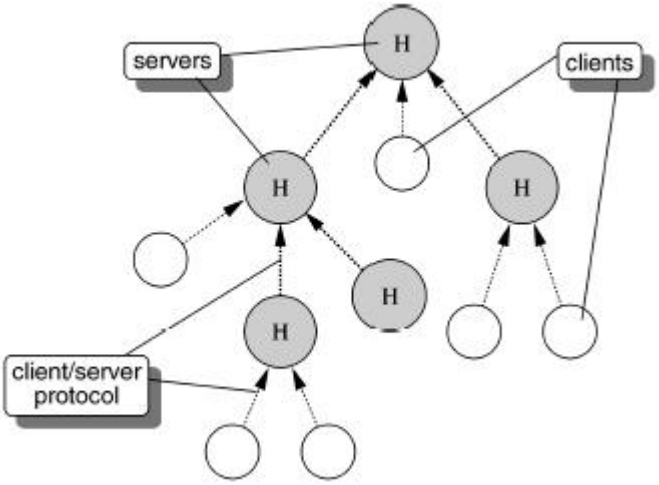

Figure 2.10: A Hierarchical Topology. *From [7]*

nodes have a client server relationship. Thus, a hierarchical node need only propagate information it receives to its parent node in the form of root subscriptions and publications.

The main routing principle behind Siena is to push notifications as close as possible to parties that may be interested in that information. Known as **downstream replication**, this can be achieved both by subscription forwarding and advertisement forwarding. Subscription forwarding is the method used for routing in the Siena hierarchical implementation.

The filters poset is used to assist in pruning the number of subscriptions forwarded. Essentially, root subscriptions are the only ones sent. As such, subscriptions covered by previously forwarded subscriptions are pruned and network traffic is kept to a minimum. In order to ensure consistent notification across the network, Siena employs publication forwarding to master nodes, and leaves further notification beyond that of root subscriptions to the nodes on which the more specific subscriptions reside.

Subscription

The Event server receives a subscription via `Subscribe(X,f)` and walks through its subscription poset starting at each root subscription. If a subscription is found that covers the filter ***f*** and contains X in its subscriber set the search terminates. Otherwise, the search terminates with two possibly null sets, ***sf'*** - sucessors and ***sf"*** - predecessors. If the filter exists in the poset, ***X*** is simply placed in the subscriber set of that particular filter. Finally, should

neither of these apply a new subscription is inserted between ***sf'*** and ***f"*** in the subscription set with ***X*** added to its subscriber set. If ***f'*** is null then the subscription is inserted as a root subscription. In this case the subscription is sent to its master node. A detailed example of subscription forwarding can be found in [7] section 5.5.5.1.

Notification

Upon reception of notifications (i.e. a publication) the event server initialises a queue ***Q*** with all its root subscriptions. If the subscription covers the notification then the immediate predecessors are appended to ***Q***. If there is no covering then the subscription is removed from the queue. Upon termination the subscribers in the subscriber sets of ***Q*** are sent the notification. If the master server was not the source of the notification than a copy of this notification is also sent to the master server. A discussion of unsubscription and unpublication is deemed beyond the scope of this introduction and may be reviewed in [7].

2.4.3 Elvin

Elvin [34] is a content-based routing system similar to the Siena system introduced in the previous section. Elvin is a mature technology and active research is leaning towards standardisation of the protocols as currently implemented as well as adding increased security, quality of service and other performance enhancements. Aside from the presence of a mature, efficient implementation the Elvin subscription language can be considered as a more expressive form of that outlined for Siena. Introducing logical operators and tri-state logic gives the Elvin subscription language a massive power advantage over the Siena subscription language which still relies on conjunction for pattern matching. However, a driving factor behind the Siena is the ability to function as a wide-area event notification service while maintaining certain expressiveness. The closed-source nature of the Elvin implementation is also prohibitive in terms of enhancing content based routing to include more complex datatypes, such as OWL class and is another major factor driving the choice of Siena as our basis implementation.

```
<!-- Process Model description -->
<process:ProcessModel rdf:ID="BookDetailsFinderProcessModel">
  <service:describes rdf:resource="#BookDetailsFinderService"/>
  <process:hasProcess rdf:resource="#BookDetailsFinderProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="BookDetailsFinderProcess">
  <process:hasInput rdf:resource="#BookInfo"/>
  <process:hasOutput rdf:resource="#BookResearcher"/>
  <process:hasOutput rdf:resource="#BookPublisher"/>
  <process:hasOutput rdf:resource="#BookAuthor"/>
</process:AtomicProcess>

<process:Input rdf:ID="BookInfo">
  <process:parameterType rdf:resource="bkont:Book"/>
  <rdfs:label>BookDetails Name</rdfs:label>
</process:Input>

<process:Output rdf:ID="BookResearcher">
<process:parameterType rdf:resource="bkont:Researcher"/>
  <rdfs:label>Book Researcher</rdfs:label>
</process:Output>


<process:Output rdf:ID="BookPublisher">
  <process:parameterType rdf:resource="bkont:Organization"/>
  <rdfs:label>Book Publisher</rdfs:label>
</process:Output>

<process:Output rdf:ID="BookAuthor">
  <process:parameterType rdf:resource="bkont#Person"/>
  <rdfs:label>Book Author</rdfs:label>
</process:Output>
```

Figure 2.11: An OWL-S Process Model

Chapter 3

The State of The Art

The purpose of this chapter is to outline the current state of research in the fields of semantic service discovery, ontologies and publish/subscribe. Each section presents the research conducted for this dissertation and includes an evaluation and general conclusions.

3.1 Semantic Service Discovery

Although heavily supported by languages such as OWL, OWL-S and RDF as well as SOAP and XML research into semantic service discovery is still maturing and as a result a standard means of discovery is still a way off. As a result of this non-convergence research continues in several parallel avenues outlined below.

3.1.1 UDDI and Service Discovery

Not a semantically enhanced standard, the recently agreed UDDI specification version 3 [14]presents some interesting additions. Most notably, in the context of this dissertation, a Subscription API has been added. Not explicitly a publish/subscribe application, at least in the distributed event notification sense, the movement of the standard towards a publish/subscribe concept is an interesting one. The Subscription API provides a means for tracking new, modified and deleted UDDI entries for *businessEntity*, *businessService*, *tModel* and *publisherAssertion*. Subscribers may be notified by email or via SOAP/HTTP by implementing a subscriptionListener API.

Since subscribers must express explicitly the requirements, in terms of the selected category, such as tModel, it is observed here that the UDDI subscription API more closely resembles a topic-based subscription. Aside from the lack of UDDI support for semantic concepts, more expressive textual content based subscriptions would offer more expressiveness of subscription language and hence more accurate notifications in terms of web service discovery. It could however, also be argued that since automatic discovery and invocation is not supported by UDDI web services that expressiveness and accuracy of category may not necessarily be as important as in a semantically enhanced version.

3.1.2 Semantically Enhanced UDDI

There is a very active body of research in semantically enhancing the UDDI registry standard. Since the UDDI standard is plentiful in features and a mature standard, it seems a logical progression to attempt to build on this maturity by adding semantic annotation. In [1] the authors endeavour to provide a structure whereby semantic information may be annotated onto current UDDI elements, such as tModel. Similarly, [40] endeavour to ”import” the semantic web into a UDDI standard implementation. Each of these works aims to introduce concept matching to the UDDI registry by incorporating reasoning and OWL-S support to current implementations. The active research in this area highlights one of UDDI’s main weaknesses, lack of service capability support and emphasises a general consensus amongst the web service academic community that semantic support for capability matching of web services is primary the area forward.

3.1.3 OWL-S Matchmaker

In [32] Paolucci et al outline a methodology and efficient algorithm for semantic service capability matching. The current body of research focuses primarily on comparing inputs and outputs of a service as semantic concepts represented in OWL. By extracting subsumption relationships between input requirements and outputs the authors propose a way of ranking semantic matching results. This ranking can be used in conjunction with other user-defined, or plug-in, constraints to inform of an exact, or potentially useful web-service capability match.

In [41] the same authors propose an efficient way to apply the matching methodologies outlined in [32] to the UDDI Registry. This basic extension adds a *capability port* to the cur-

rent UDDI implementation thus making it semantically aware. An interesting contribution of [41] is an evaluation of ranked matching and a resulting focus on accelerating performance by minimising the amount of matching and, therefore reasoning, that takes place. Any implementation of a semantic matching engine into the publish subscribe model must take this observation into consideration and must endeavour to minimise the frequency of concept matching that takes place.

3.1.4 METEOR-S for Service Discovery

The METOR-S project [20] is a large research effort focusing on the application of semantics to WSDL, in the form of WSDL-S, and semantic support to UDDI. Interestingly in the context of this research, [20] appears to offer significant contribution in the area of distributed, peer-to-peer infrastructures for semantic publication and discovery of services. [20] targets the area of semantic web-service discovery as an important and apparently underdeveloped area in the context of current web-services research. [20] also makes the observation that for a UDDI registry in its current form, web-service discovery across multiple UDDI-registry nodes is inefficient. The research concludes that adding web service description semantics and annotating the UDDI nodes themselves may provide avenues for improving the efficiency of a distributed UDDI registry. This is interesting considering the content based routing avenue of this research. Work done by the METEOR-S project shows that the semantic concepts available in OWL can be used to for other purposes besides explicit matching of service descriptions against requirements.

[20] also addresses the issue of diverging ontologies within such a distributed system. The paper proposes a specialised ontology, the registries ontology, designed to maintain relationships between registries. However, no explicit Ontology alignment takes place. Instead, registered publications are restricted to a set of ontologies explicitly defined by the registry of publication.

3.1.5 Ranked Matching

The work done in [26], focuses on a finer grained approach to matching than presented in [32]. By consideration of the service category and finer-grained user constraints based on concept properties as well as input and output matching the work done by Jaeger et al. proposes a more accurate approach to semantic matching. The matching process is broken into

four distinct phases; input matching, output matching, service category matching and user constraint matching, each of which scores a numerical ranking, also based on the subsumption relation. The semantic matcher then aggregates a ranking in each of these categories and as a result can produce an accurate match with informative matching statistics. A Java prototype has been built and is hosted by the Technischen Universitat at Berlin [1]. Examination of this prototype shows a well designed, apparently efficient implementation based on JESS, RDF/QL and an OWL knowledge base for JESS, all discussed in section 2.1.3. As a freely available and mature implementation of progress in the semantic matching area, integration of this Java implementation into our publish/subscribe implementation would definitely be a useful and worthwhile course of action.

3.1.6 OWL-S API

The OWL-S API [31] [2] provides a Java API for programmatic access to read, write and invoke OWL-S service descriptions. While the focus of this research is in the area of semantic web-service discovery, any implementation of a service-discovery system would involve parsing of an OWL-S service description mark-up as well as integration of the concepts represented in the service into a programmatic model of some form. The OWL-S API has a stable basis for accomplishing this task. Coupled with this the OWL-S API has been built on the Jena framework allowing for relationships to be asserted and reasoned over through a Java API, possibly via the Pellet [30] reasoner. As the most comprehensive Java API for OWL-S that has additional support for reasoning it is felt that the API could be an interesting and valuable component of any implementation developed in the context of this dissertation.

3.1.7 Reasoning for OWL-S

As outlined in section 2.1.3 there is an abundance of technologies available supporting reasoning over OWL concepts and services via programmatic interfaces. A discussion of these tools follows.

[1] http://ivs.tu-berlin.de/Projekte/owlsmatcher/
[2] http://www.mindswap.org/2004/owl-s/api/

Jena

The Jena [27] semantic web framework provides a programmatic environment that includes an RDF/RDFS/OWL inference engine, an OWL API, and RDQL [3] support. The OWL reasoner as included with version 2.2 of the framework is a rule-based implementation that supports fully only the OWL-Lite subset of the Ontology Web Language. The Jena 2.2 documentation actually highlights the weakness of the default reasoner and, through the Jena DIG description logic interface, recommends the use of an external reasoner for completeness and for performance reasons. Our implementation will not rely on Jena reasoning alone but will incorporate a more fully featured reasoner. However, due comprehensive nature of the Jena API which, interestingly, includes an API through which RDF assertions may be made persistent in a MySQL or Oracle database, its contribution cannot be ignored. The Jena framework was the only semantic framework encountered that explicitly supported persistent RDF objects through Java.

Pellet

Pellet [30] is an OWL DL reasoner based on the tableux algorithms developed for expressive description logics. Pellet fully supports the OWL-DL subset of the Ontology Web Language specification. Pellet can deal with both ABox (assertions about individuals), TBox (axioms about class definitions) and RBox (axioms about properties) each of which can be considered as an important feature in the context of OWL-S service matching, particularly when considering the subsumption relation which can be deduced via reasoning over TBox axioms. The Pellet API may be used in conjunction with the OWL-S API , via the Jena DIG interface or as a standalone API. The most recent version, 1.2, was not available at the time of evaluation, however versions 1.1 and 1.0 were freely available for download.

JESS and The OWLJessKB Project

JESS [29] is a general purpose rule engine and scripting environment. Tightly integrated with Java and incorporating the latest in rule-based reasoning algorithms JESS claims to be one of the smallest and fastest rule engines available. OWLJessKB [28] is an OWL Reasoner based on the Jena and JESS APIs. The majority of the semantics are implemented using

[3] A Query Language for RDF

the JESS API. The OWLJessKB supports the TBox reasoning. OWLJessKB has explicit support for subsumption and classification reasoning as well support for transitive properties, subsumption and classification on datatypes defined in the XML schema.

RACER

RACER [24] is another reasoning API for the semantic web. Fully supporting T-Box and A-Box axioms, the RACER API is a core component of several semantic web tools. Currently a commercial endeavour [4] in the context of this research, especially considering the capabilities of freely available tools already discussed, further analysis of the RACER API was not undertaken. However, an interesting observation in [24] is the inclusion of a publish/subscribe interface for registering interest in specific knowledge. Registered IPs are automatically notified of the modification of knowledge based information in terms of permanently-expressed queries. Although not explicitly relevant the similarity between the RACER approach and the goals of this dissertation are obvious.

3.2 Publish/Subscribe

The state of the art in terms of the publish/describe paradigm is captured nicely by Eugster et. al in [21]. This work not only details publish/subscribe implementations, mature and new, but serves as a gentle introduction to the publish/subscribe paradigm. Classification of publish/subscribe systems and related technologies is undertaken. The research conducted for the purposes of this dissertation focused entirely on wide-area publish/subscribe systems that incorporated content-based routing.

3.2.1 Siena

The Siena implementation has already been introduced in 2.4.2. The Siena project is surrounded by a very active body of research. In [7] the foundations for the Siena implementations were outlined. Following on from this, two implementations have arisen. The first, a C++ implementation of the acyclic peer-to-peer network, is maintained for archival

[4] http:/www.racer-systems.com

reasons only and is cited as an obsolete implementation. The Siena Java project is an up-to-date implementation of the hierarchical Siena topology. Full source code is available for both implementations. The Java implementation is currently at version 1.3.2 and is a full implementation of the Siena interface except for the omission of patterns and the correct implementation of the unpublish and unsubscribe functions. On testing the Java interface another, apparently undocumented requirement is for a subscriber to have subscribed before the publication of interested information. This clearly violates time-decoupling as outlined in [21] but whether this is an intentional omission is not clear from the documentation or the surrounding literature.

Siena's forwarding architecture is best detailed in [5]. This work includes various optimisations and enhancements, such as short-circuit filter evaluation and addition of a selectivity table for forwarding as well as other optimisations that have not been implemented in the basic Java implementation. The fast forwarding enhancements, including the new forwarding tables are currently available as an independent C++ module. [5]

In the most recent Siena-related publication [6], Carzangia et al. focus on the presentation of a routing scheme for a cyclical peer-to-peer wide-area-event notification service architecture. The main contribution of this work is to outline algorithms for progressing Siena from the hierachical and acyclical topologies outlined in [7], to the more general environment of a peer-to-peer network and to prove the scalability of the proposed implementation.

3.3 Ontology Alignment

The diverse nature of any wide-area service that integrates ontologies requires that varying representations of the same concept via multiple ontologies is considered. The state of the art in Ontology alignment field is presented in [19]. This work presents the numerous methodologies and tools currently available. A detailed discussion of this work is deemed beyond the scope of this dissertation research. The implementation, for simplicity, will assume the existence of a mapping, or several mappings, relating ontological concepts in a way meaningful in the context of the whole system.

[5] http://serl.cs.colorado.edu/ carzanig/siena/forwarding/index.html

3.3.1 Ontology Alignment API

[22] presents an interesting piece of work in the area of generic ontology alignment. The interesting contributions are two fold. Firstly, an XML format for alignment is presented. This general XML format abstractly defines the relationships between ontologies and the concepts contained within. A relation, such as equality, or subsumption, can be defined between two ontological concepts and an associated confidence factor assigned. The standard defined also supports more complex concept matching, such as that provided by SWRL [6].

The API itself may not be as useful. Despite integration with the OWL API, there is no support for the full level of abstract mappings as expressed in the XML format definition.

3.4 Conclusions

This chapter has shown that the research area of semantic service discovery is a very active one. This is especially the case when considering the addition of capability matching to the service discovery. Work undertaken in the field of content based routing for the publish/subscribe paradigm is currently focused on fast forwarding and improvement of performance for large-scale notification services. There is little evidence of driving progress in the area of subscription language enhancement, or movement towards more complex data-types other than string, integer and date-time. Lastly, work done in the OWL-S service discovery paradigm does not consider the possibility of explicit ontology mapping, instead focusing on standardising ontologies for use in the discovery paradigm.

It can thus be concluded that development of a publish/subscribe model for web-service discovery incorporating expressive complex data-types for subscription, OWL-S enhanced content-based routing and on-the-fly ontology integration is an original concept. It is hoped that this dissertation may make a significant contribution in realising the development and implementation of such a platform.

[6] Semantic Web Rule Language

Chapter 4

Design

This chapter outlines the implementation design. Two phases of implementation took place over the period this dissertation was completed. The first exploratory phase was a simple, centralised text-based system designed as a means by which tools such as the Siena server nodes and semantic reasoning could be tested. Since the main aim was to explore the Siena system, explicit description of the simple design has been amalgamated with the implementation details of chapter 5. The following sections outline the design of an enhanced prototype in terms of our target use case, the details of which are based on the Siena Java implementation discussed in previous chapters.

4.1 Overview

Figure 4.1 illustrates the specific components of our proposed implementation. These components are divided explicitly into three facets; subscription, publication and matching. Within these there are requirements for subscription set structure, a communications layer, ontology integration and ontology alignment.

It has already been established that the implementation will consist of a modified version of hierarchical implementation for Java. The original implementation for Java has no notification support beyond the simple types discussed in [7]. Similarly, no support for OWL-S matching, or OWL programmatic support structure exist. In order to realise the goals of a semantically aware implementation the following enhancements are necessary.

- Integration of an OWL/OWL-S Java support package.

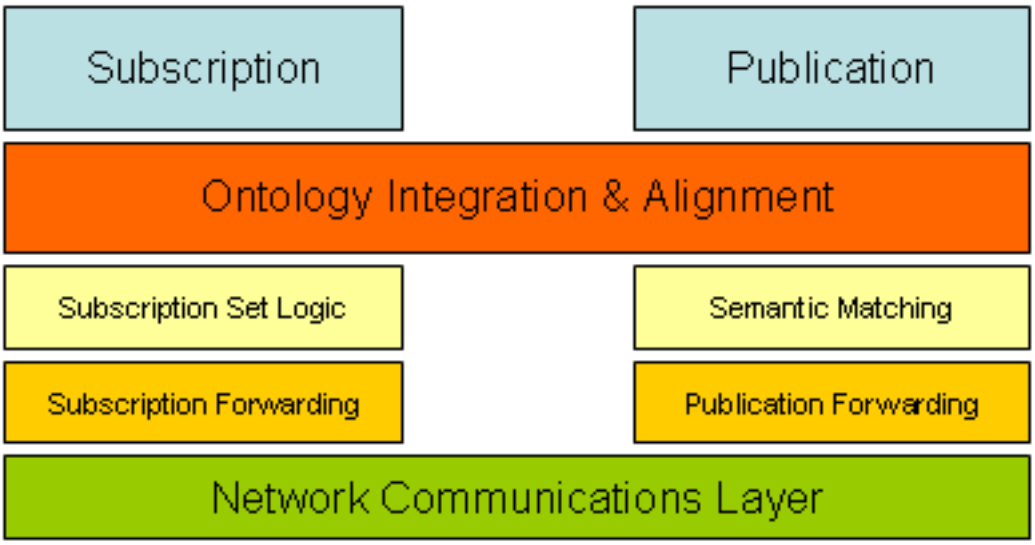

Figure 4.1: Design Overview

- Integration of the JESS/OWLJessKB reasoner API.
- Development of an OWL-S matching component.
- Client and Server Interface Support.
- Modification of the Siena *poset* structure.
- Expansion of the Siena subscription language.

A detailed technical discussion of these components as implemented is undertaken in chapter 5 and therefore the remainder of this chapter proceeds at an abstract level.

Figure 4.2 illustrates the Siena interface. The implementation of this interface is the most basic requirement in fulfilling the goal of any publish/subscribe system. The original Siena interface is capable of receiving and registering a filter and a Java *notifiable* object, which handles the delivery of a notification to the calling subscriber. Similarly, the publish abstract method allows for injection of a notification into the Siena network.

The solution to providing a basic publish/subscribe architecture was modify the current Siena architecture to reflect the requirements of an OWL-S capable publish subscribe system. At the highest level, we must first modify the Siena interface as currently implemented.

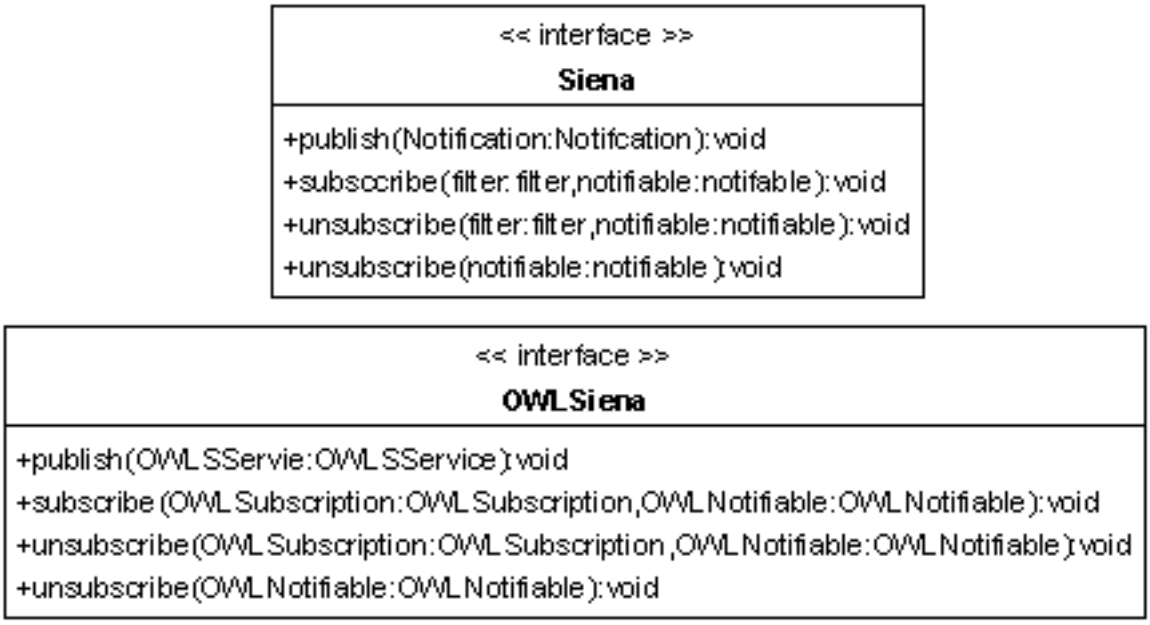

Figure 4.2: The original and modified Siena Interfaces

Firstly, this interface must be changed so that OWL-S services may be directly presented to the Siena node for publication. Secondly, since we are enhancing the subscription language, we must provide classes that encapsulate the required information necessary for correct subscription registration as well as an interface method for the presentation of subscription information to the Siena server.

4.2 Application Description

In order to solidify the vision of a semantically-enhanced publish/subscribe system for web-service discovery a usage scenario was constructed. In this instance a service discovery architecture is presented whereby an OWL-S service may be published and subscriptions constructed.

The aktors [1] publication support ontology provides a comprehensive ontology that captures the publication media domain. This OWL ontology was a component of the MINDSWAP sample services for the OWL-S API.

[1] http://www.aktors.org/ontology/portal.owl

4.2.1 Usage Scenario

There are three main actors involved in this usage scenario; the subscriber the publisher and the discovery node(s). This is illustrated in figure 4.3.

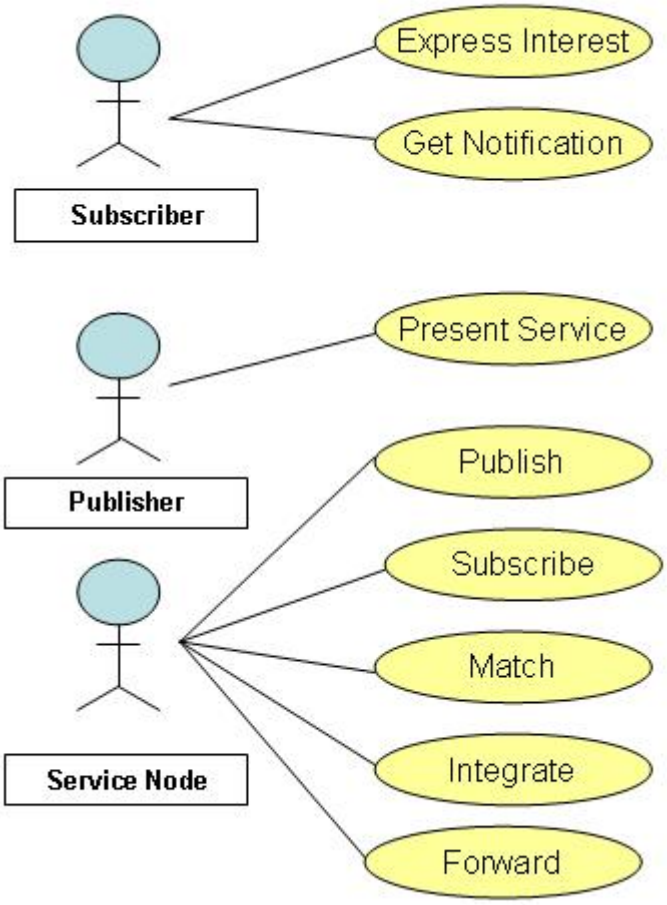

Figure 4.3: Use Case Actors

At the most abstract level a typical enhanced Implementation exchange takes the following form.

1. A Subscriber presents its expressed discovery constraints to the Siena server.
2. The Siena server registers the constraints and the location of the subscriber.
3. The Siena server is presented with an OWL-S Service by a publisher.

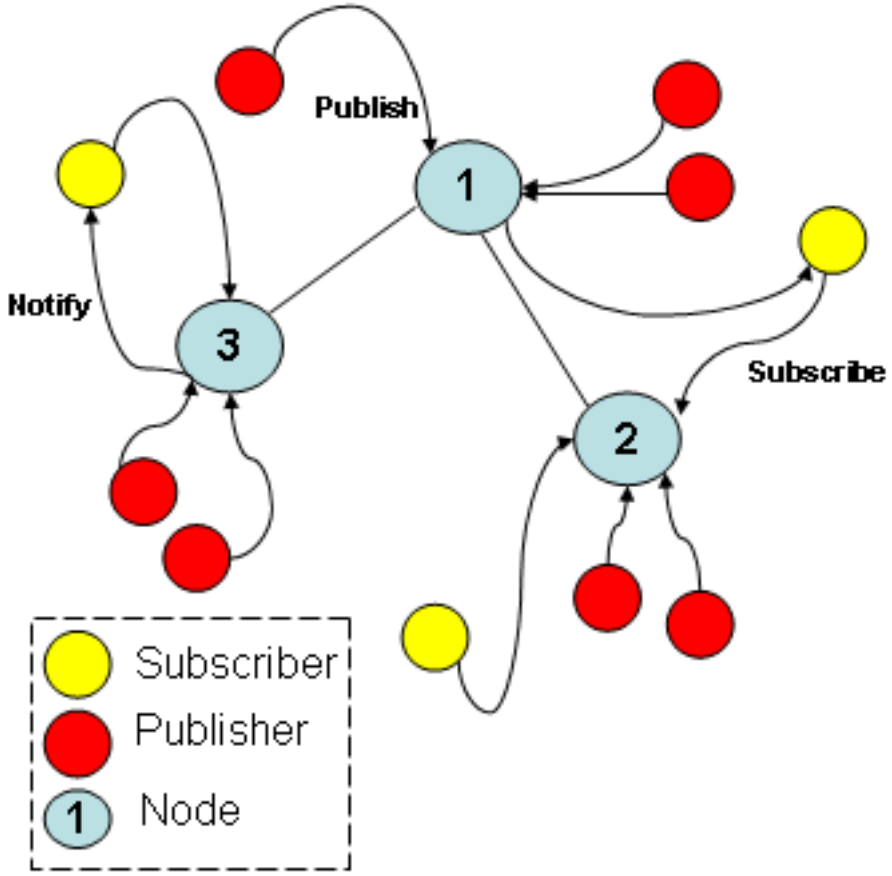

Figure 4.4: Plan View

4. The Siena server parses this service extracting the service profiles.
5. These profiles are subsequently matched against our subscription/subscriber set and a notification list generated.
6. Each subscriber is notified of the publication of a new OWL-S service that matches the content-based requirements previously expressed.

An top-down illustration of this is presented in figure 4.4.

4.2.2 Activity Diagrams

Illustration of the methodology for publication and subscription, both on the client and server side is done by means of activity diagram.

Figure 4.5 shows an activity diagram for subscription from the point of view of the server. Figure 4.6 shows publication on the server side and figure 4.7 shows client side subscription.

Server Subscription

Figure 4.5 starts with the examination of provided ontologies. Should an ontology be unfamiliar to the system the ontology is first integrated into the subscription. The subscription is then examined, if it is a root subscription it is inserted into the *poset* and the subscription is forwarded to the server node. Otherwise the subscription is inserted into the tree and the subscriber is mapped, a record of its ip-address is maintained.

Server Publication

Figure 4.6 shows server publication. Once the profile has been presented and parsed the accompanying ontologies are integrated into the knowledge base. A poset queue is initialised and covering relation applied iteratively. Should a match be found the subscription subscriber set is added to a notify set. Finally each subscriber in the notify set is sent a copy of the published service profile.

Client Subscription

Having initialised a notification handler, the client forks into two threads as illustrated in figure 4.7. In parallel, constraints are specified using the enhanced subscription language, ontologies and mappings used to specify concepts are provided and the subscription sent to the Siena server. Once the service profiles have been received it is up to the client to handle integration and invocation of the service.

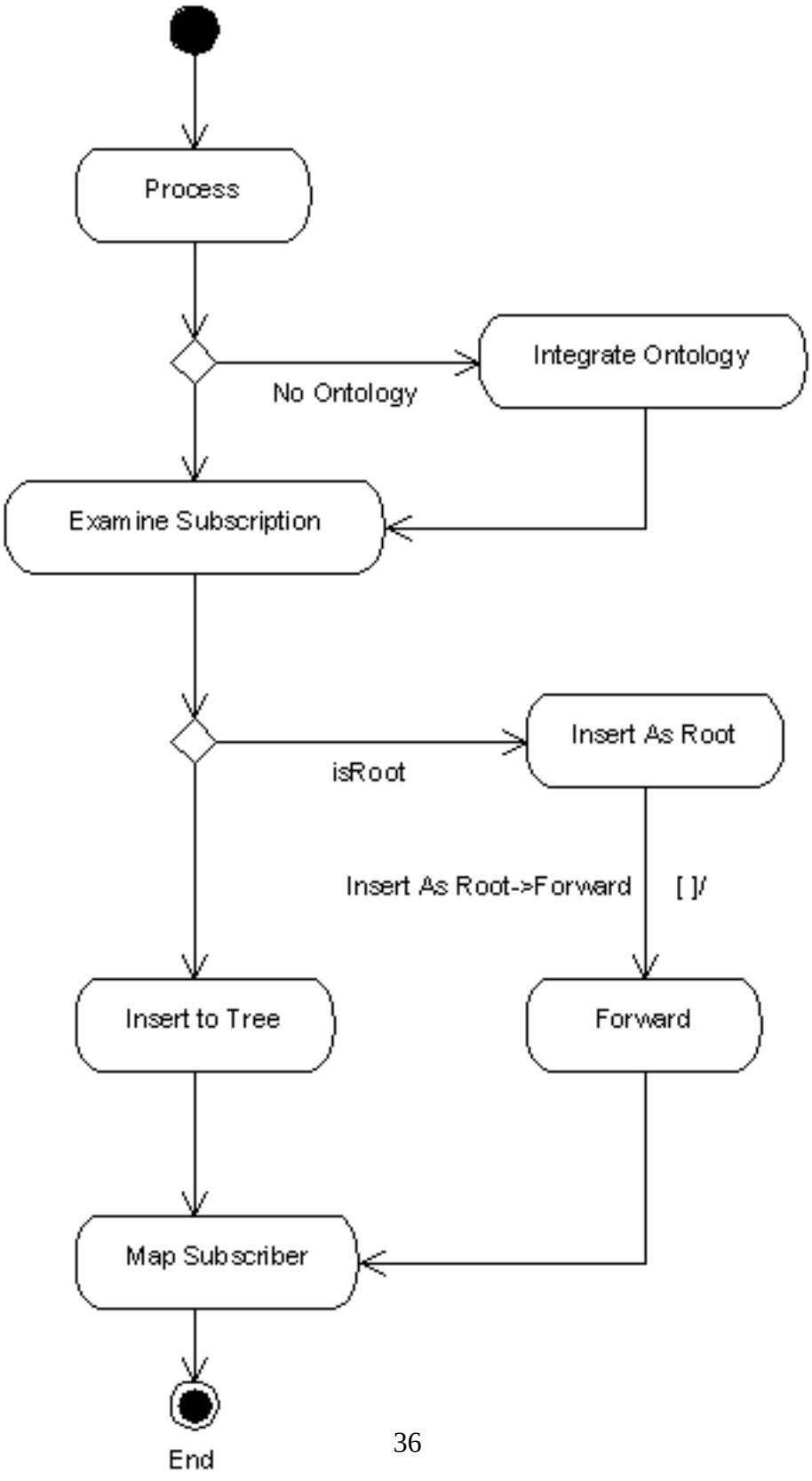

Figure 4.5: Server Subscription

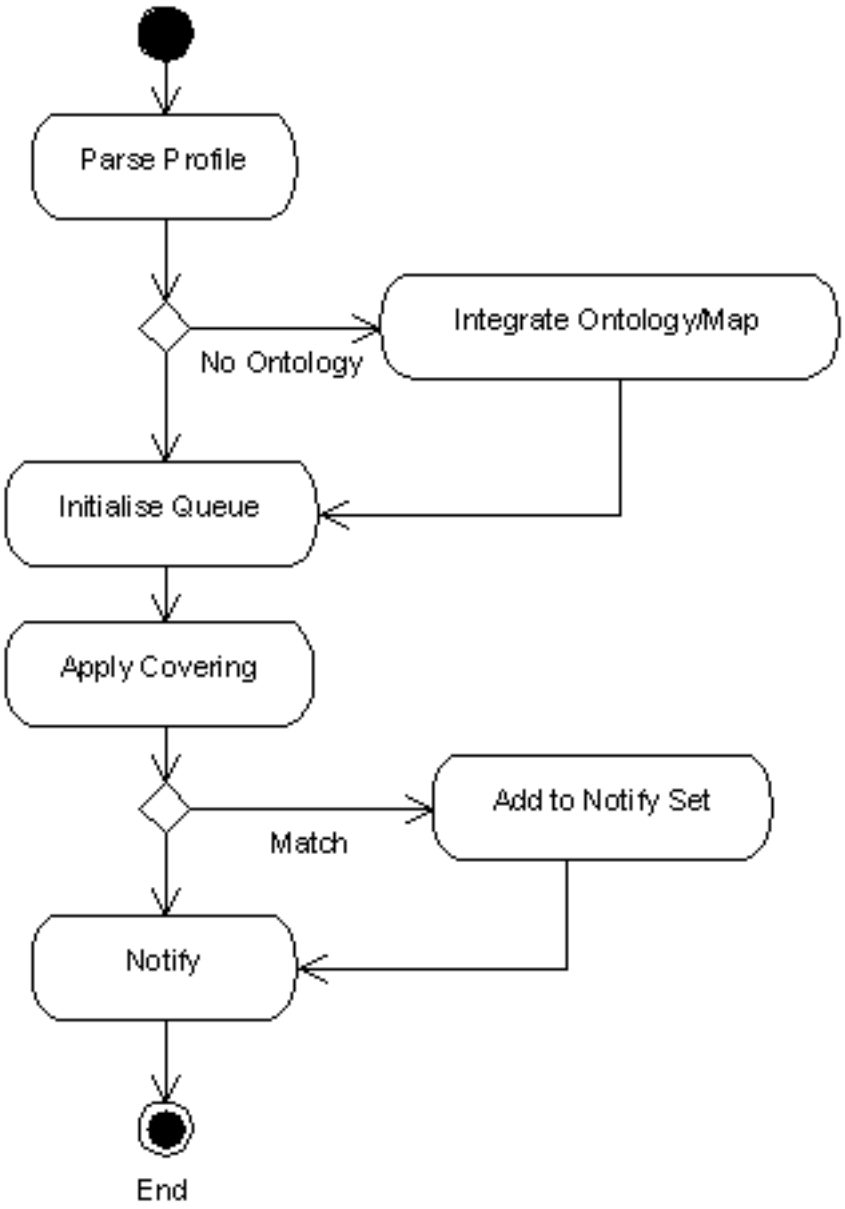

Figure 4.6: Server Publication

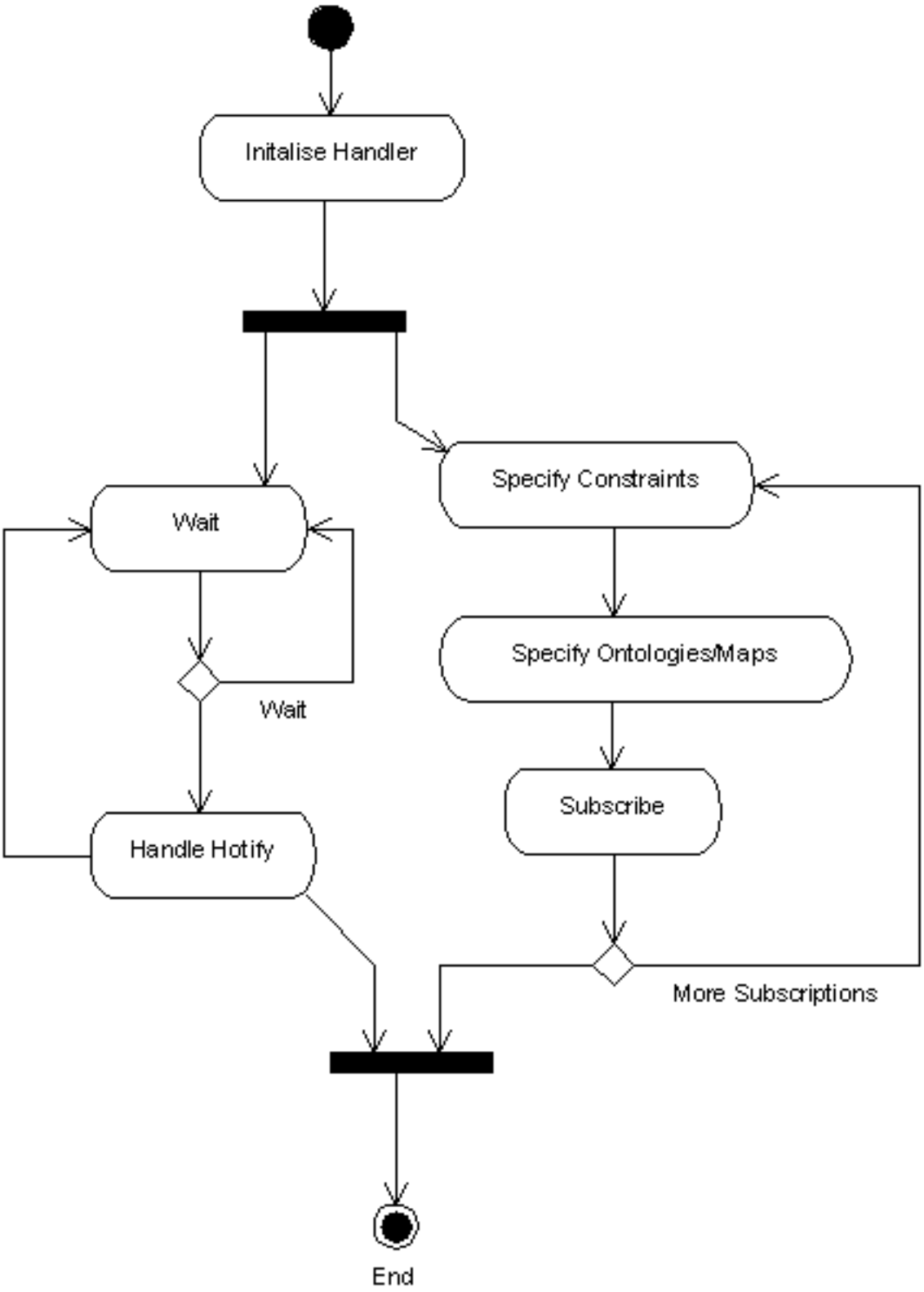

Figure 4.7: Client Subscription

4.3 Enhancements

The following sections describe necessary enhancements to the current Siena implementation.

4.3.1 Subscription Language

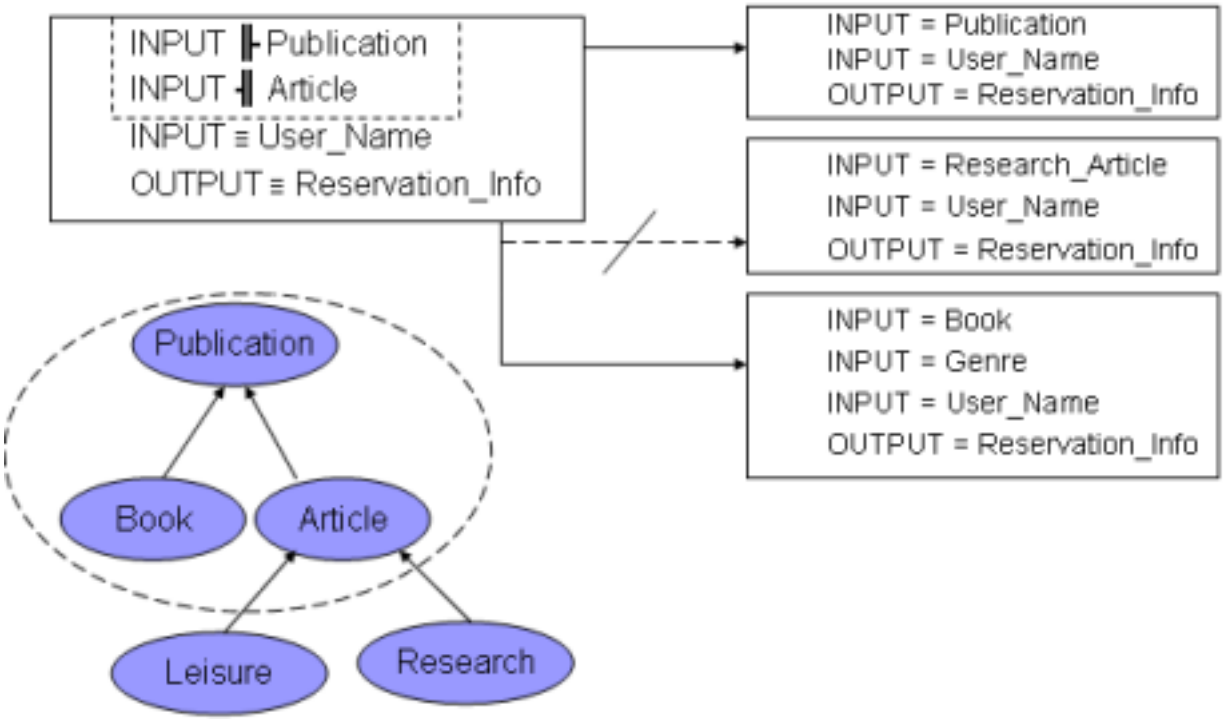

Figure 4.8: A usage scenario

One of the primary goals of the design of this implementation is to enhance the Siena subscription language. The primary addition to the data-types supported by Siena is *owl:Class*. This addition facilitates the subscriber in specifying concepts representing the capabilities of a service in terms of inputs and outputs. Also, the Siena reasoner is now capable of understanding the concepts of explicit interest to the subscriber.

In this form the subscription language is capable of expressing only the exact requirements concepts. Since the matching algorithm is based on the subsumption relation it seems natural to provide the facility for the subscriber to express interest in more general terms via knowledge of this relation. To accommodate this, there are three additions to the Siena

constraint operators, each of which works in terms of the class hierarchy as provided by the ontology referenced by the subscription. The $\vdash$ operator is used to express subsumption. In figure 4.8 *input* $\vdash$ *Publication* can be used to express the constraint that this particular subscription is interested in having the concept Publication as an input. The nature of the subsumption operator also expresses interest in concepts that are *subsumed by* the Publication concept. The $\dashv$ operator expresses an inverse-subsumption constraint. In the case of figure 4.8 the subscriber has expressed an interest in the article concept or concepts that *subsume* this.

There is an or-relation between these constraints represented by the broken-line box. As a result, this constraint is a complex one whereby the subscriber is interested in concepts that lie between publication and article in the class concept hierarchy defined in blue. Since the concept of a book is on the same level of the hierarchy as the article concept, an expression of interest in inputs of type book is implicit. The remainder input and output utilise the $\equiv$ operator. A subscriber may not wish to register interest in concepts related by subsumption and simply request a concept equivalent to that in which interest is expressed.

An enhanced subscription will consist of a conjunction of constraints on inputs and outputs. The enhanced constraint triple takes the form *input or output* , $\equiv$ *or* $\vdash$ *or* $\dashv$, *owl:Class* and an or-relation can be used to form complex subscription requirements for each input and output.

An English expression of the subscription follows :

This subscriber wishes to receive notification of the publication or modification of any service profile with at least two inputs and one output. One of these inputs must, conceptually, be a publication, article or book. This subscriber wishes to receive confirmation of reservation by receipt of reservation information or an equivalent concept and as a result requests that this be an output of the desired service.

The right hand side of figure 4.8 shows three sample service profiles which have been published to the Siena server. The first of these is *covered* by the subscription since we have satisfaction for each constraint placed on the content of the service profile. The second of these profiles fails on the first constraint, since the concept of research article is too specific in terms of the class hierarchy shown. The failure of one constraint results in the failure of the match as a whole.

The third service profile illustrates an interesting application of the covering semantics used in the Siena content based routing system. Each constraint has been satisfied correctly and therefore the subscriber is notified of the existence of a matching subscription. An important observation is that the input concept Genre is also required of the published service profile. The omission of the Genre input parameter may be interpreted as an expression of not caring what other inputs exist on the service. It is assumed in this case that the registered subscriber agent is capable of reasoning over this input requirement in order to provide enough information to invoke the service successfully. However, the omission may also express disinterest in any other inputs or outputs of any time on behalf of the subscriber, perhaps since the subscribing agent is not capable of handling them. Since it is infeasible to express disinterest in every unsupported concept in a large scale system, e.g. by means of a *not* operator and since the inclusion of service inputs and outputs numbers decreases greatly the expressiveness of the notification in this scenario, from henceforth it is assumed that an automatic agent is capable of handling parameters it has not specifically expressed interest in. In this way the covering semantics of the content based routing scheme are preserved more effectively.

Subscription Language Evaluation

The advantages of enabling such an expressive language in a publish/subscribe context are three-fold. Firstly, agents are pro-actively notified of matches that they may explicitly invoke, allowing agents to seamlessly integrate different services and improved publications of the services they may already invoke. Secondly, an agent may be informed of the appearance of potentially useful services that may be invoked given extra-information, or, may provide a less-than-ideal level of service. Similarly, upon the appearance of exact matches these less-ideal matches may be switched out and the more suitable matches switched in. Lastly, the pro-active nature of the notification removes the need for the device to explicitly search for new publications on a regular basis. The agent and end-user can thus automatically assume that it is always aware of the most suitable web-services given the specified constraints.

4.3.2 The Subscription Poset

While remaining at an abstract level it is necessary to discuss enhancements and modifications to the Siena subscription Poset structure and subscription forwarding architecture at the

C Operator	C'Operator	Concept Relation
Subsumption	equivalence	C subsumes C'
Inverse Subsumption	equivalence	C subsumed by C'
Equivalence	Inverse Subsumption	C subsumed by C'
Equivalence	Subsumption	C subsumes C'
Subsumption	Subsumption	C subsumes C'
Inverse Subsumption	Subsumption	C subsumed by C'
Inverse Subsumption	Inverse Subsumption	C subsumes C'
Inverse Subsumption	Subsumption	C subsumed by C'

Figure 4.9: The relation constraint **C** covers constraint **C'**

design stage. Figure 2.9 on page 18 shows the Siena subscription poset. The main consideration behind enabling OWL-S based subscriptions in such a manner is the preservation of the covering relation between filters. In particular, the partial ordering between subscriptions within the structure must be maintained.

In order to accomplish this we must define a covering relation between our enhanced subscriptions. It remains that an OWL-S based subscription **S** covers another subscription **S'** if the notification set generated by **S** is a superset of the notification set generated by **S'** given all publications presented to the Siena server. In less formal terms if **S** is a less specific case of **S'** then **S** covers **S'**. The introduction of the semantic operators, in effect, does not change this relation. In fact, the $\vdash$, $\dashv$ and $\equiv$ operators are analogous to the $\geq$, $\leq$ and = operators in Siena.

The introduction of the or-relation on specific constraints does complicate things slightly. However, if the covering relations are broken down granularly this the relations still hold. The relation constraint **C** is covered by another primitive constraint **C'** is defined exhaustively in terms of the subsumption operators in figure 4.9. The concept relation in this figure is the relation that must hold between concepts expressed between the parameter of each of the constraints. The case where two constraints are equivalent occurs when they have equivalent concepts for the same parameter type of input and output. Although not explicitly true, it is assumed for these purposes that equivalent concepts subsume themselves.

In terms of the subscription set structure there are two more relations we need to examine. It must be possible to determine successors and predecessors in this hierarchy. In particular, a parameter constraint **PC**, i.e. a disjunction of one or more constraints placed upon one parameter, is a **successor** to **PC'** if for every primitive constraint there is a corresponding

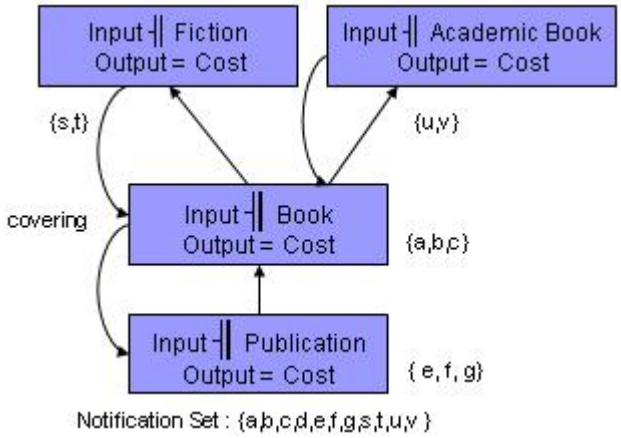

Figure 4.10: The Enhanced Poset Structure

primitive constraint **C'** in **PC'** that is covered by each constraint **C** in **PC**. The predecessor relation is the inverse of this definition. From this it can be concluded that a subscription **S** is a successor of **S'** for each parameter constraint **PC** in **S** there is a parameter constraint **PC'** that is covered by **PC'** in **PC'**.

These verbose assertions are most easily illustrated by an example scenario. Figure 4.10 illustrates an example enhanced poset. We can see here that the input parameter is the factor that determines the covering relation between the subscriptions. The fiction based input subscription covers the book based input subscription which transitively covers the publication based subscription. It can be seen clearly that the fiction subscription is a more general case of the Book and Publication based subscriptions and as any service profile matching the publication subscription would also result in the notification of every subscriber in the subscriber sets of each of the other subscriptions in the diagram.

The final covering relation that must be discussed is that between a service-profile and a subscription. Should a covering relation exist, then the subscriber set for that subscription must be added to the notification set for the matcher. Following on from the relations discussed at the start of this section a parameter **P** part of the OWL-S service **Sc** satisfies a primitive constraint **C** subject to conditions holding between the parameter concept **Sp** in the service profile and the parameter concept in the constraint **Pc**. These holding conditions are shown in figure 4.11.

Concept Relation	Operator(s)
Equivalent Concepts	Subsumption, Inverse Subsumption, Equivalence
Sp subsumes **Pc**	Subsumption
Sp subsumed by **Pc**	Inverse Subsumption

Figure 4.11: Sevice Profile Covering

4.3.3 Subscription Insertion

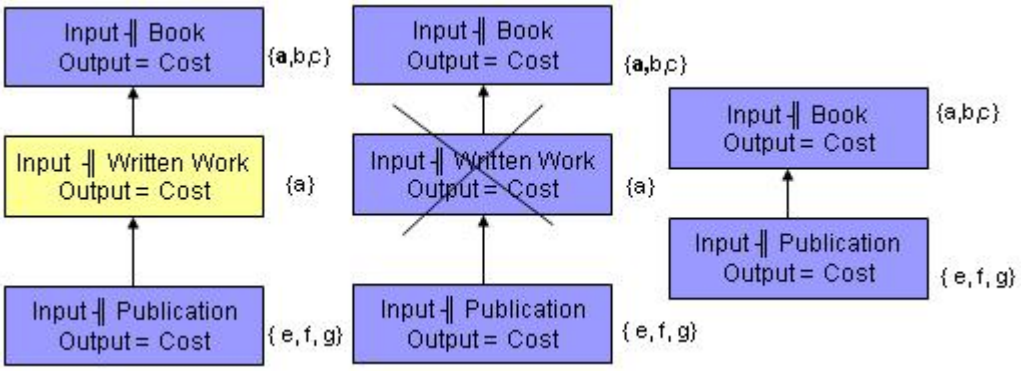

Figure 4.12: Insertion Optimisation

The subscription insertion algorithm remains unchanged. A root subscription is considered such if it has no successors. Given a call to the Siena interface `Subscribe (Subscripton:S, Notifiable:X)` the subscription hierarchy is traversed. Should **X** exist in the subscriber set of an equivalent or covering subscription then nothing is inserted to the subscription poset. Should there exist a subscription equivalent to **S** in the poset then **X** is simply inserted into that subscriptions' subscriber set. Should **S** not be found then the poset is traversed and the set of predecessors and successors is derived. Should both of these be null then **S** is inserted as a root subscription with **X** initialising its subscriber set. Similarly, if the successor set is non-empty and the predecessor set is non-empty then the subscriber set is initialised with X and the new subscription is inserted into the subscription poset preserving the partial ordering relation currently existing on the set. Figure 4.12 illustrates how the partial ordering relation and the explained algorithm gives rise to an important optimisation in the context of the enhanced Siena server. It can be concluded that maintaining a subscriber set per subscription as illustrated, instead of vice versa gives rise to a more efficient method of subscription storage.

4.3.4 Subscription and Publication Forwarding

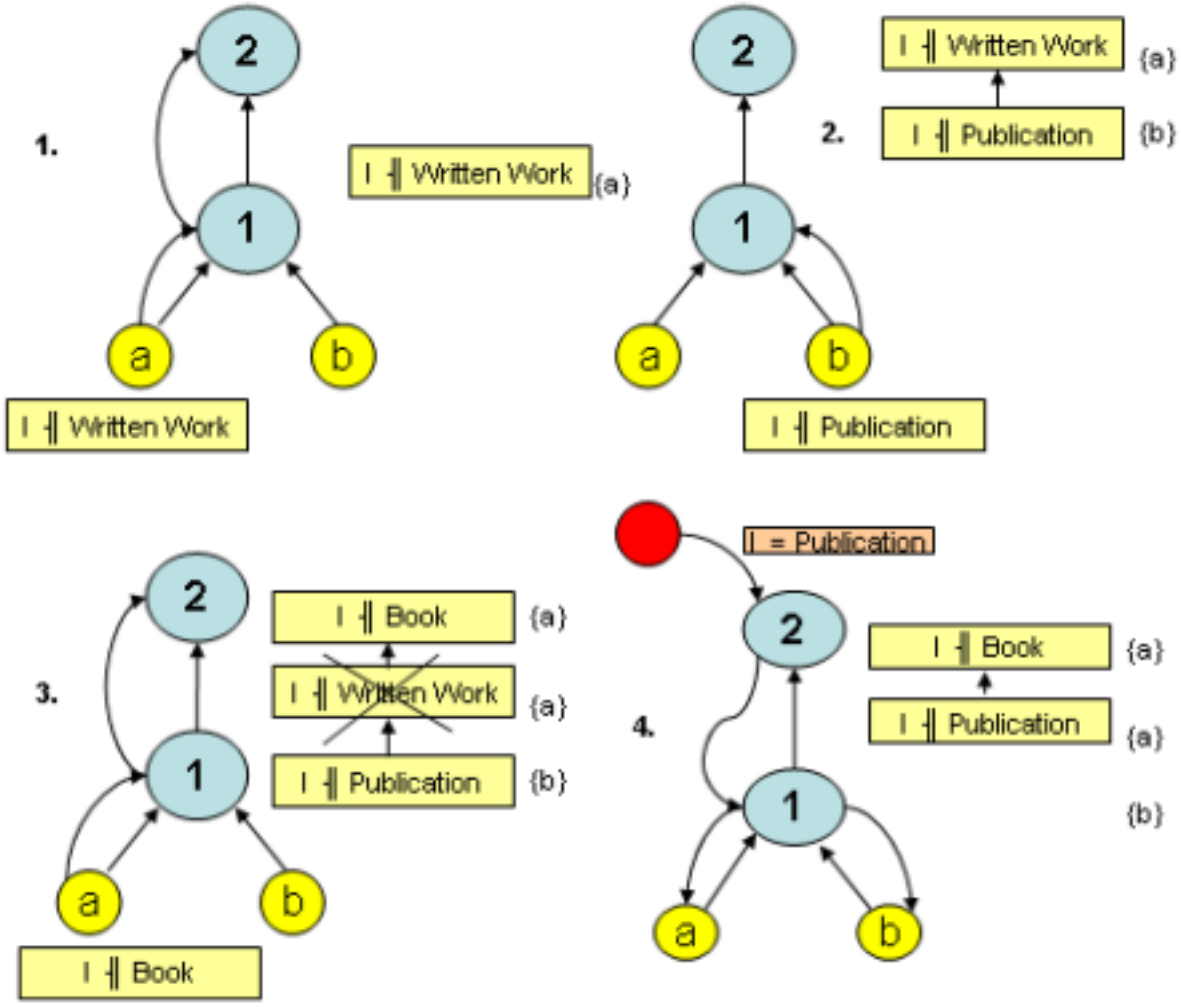

Figure 4.13: A distributed example of subscription forwarding

In the hierarchical implementation subscription forwarding takes place via client node and master node only. In order to maintain the consistency, publications are also forwarded to master nodes. In fact, the relationship between a client Siena node and its master is very similar to that of a subscriber client and the Siena node itself. All root subscriptions are forwarded to master nodes. The reference subscriber in the case where a subscription has been forwarded to a master node is the sending node itself. As a result a root subscription in the master node will cause the client node to be forwarded a copy of the publication. The net

effect of this is that no matter where a publication, or subscription takes place on the network the correct subscriber subset is notified.

Figure 4.13 illustrates subscription forwarding in a sample scenario. At stage 1 subscriber **a** registers interest in the concept of written work or less specific. Since this is a root subscription in the subscriber set for node 1 it is forwarded to node 2 where it handled in the same way as a subscription from a client. In the second illustration client **b** registers interest in the concept of a publication. This is covered by the previous subscription and arranged in the subscription poset accordingly. Note that this subscription is not forwarded to the master node. Illustration 3 sees **a** register a more general subscription and the subsequent covering sees the removal of the redundant subscription for **a**. When the publication takes place to master node 2 the root subscriptions of 1 are also present in 2 therefore the publication is forwarded to node 1 where matching takes place as usual and clients **a** and **b** are notified correctly. Only sending root subscriptions in such a manner keeps network cost low however the trade off between duplicated matching through nodes and distributed storage of publications is a subject for careful evaluation.

4.3.5 Ontology Mappings

Design description up until this point has assumed the presence of universal knowledge. In this design, universal knowledge will never be achieved and as a result partial knowledge about the reasoning domain must be assumed. Before subscription insertion, forwarding and matching takes place it is necessary it assert our Ontology knowledge base. To avoid multiple integrations of the same concepts, an expensive process, it is necessary to keep track of the ontologies previously integrated. This may be done via any efficient data-structure and string comparison of ontology URL.

While on the surface a sufficient approach, this may still lead to duplicated assertions and may result in integration of the same assertion into the knowledge base multiple times. While the string comparison eliminates the most obvious cause of duplication it does not account for two other major factors. Firstly, multiple ontologies may exist that assert the same information in exactly the same fashion. These ontologies may provide the same knowledge to the database but differ in terms of namespace and URI. Secondly, the notion of equivalent concepts defined in differing ontologies is not considered by our niave approach. It is therefore concluded that considering ontology integration and ontology alignment at the design

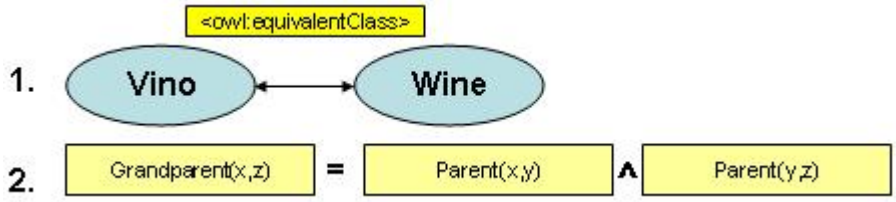

Figure 4.14: Two aspects of ontology mappings

stage may yield some interesting optimisations beneficial to the implementation as a whole.

Automatic construction of mappings between two concepts is deemed beyond the scope of this dissertation so ontology alignment will focus entirely on assertion of relationships defined in existing mappings. At an abstract level we must consider an ontology mapping as a relationship defined between two or more equivalent concepts. The default relation is defined as equivalence between entities. Figure 4.14 part one shows a level 0 mapping. These basic mappings may be captured by the *owl:equivalentClass* relationship as defined in chapter 2. The second, more complex mapping, require a more complex means of representation, perhaps via a SWRL [2] expression.

Both cases of ontology map can be expressed in a file separate to the subscription and OWL-S Service presented to the enhanced Siena server. It is concluded that any XML representation is advantageous because it does not bind itself to one particular method of ontology mapping. Figure 4.15 shows how a level 0 mapping between Vino and Wine can be represented in a generic XML format.

This design proposes that during publication or subscription the actor may be allowed to express level zero mappings in this XML format. Before Ontology integration, this XML mapping can be parsed and equivalence relations concerning know concepts may be established by asserting an equivalence relation in the knowledge base. This may thus expands the reasoners knowledge assertions in a way that may increase accuracy of matching across the notification service as a whole. For example, if Vino and Wine exist as unrelated entities in the knowledge base and the concepts are equivalent in real terms then any expression of interest in the concept of Vino is completely independent of that of the concept of Wine. Should there be an ontology-map defined assertion between these two concepts the subscriber registering interest in Wine will also receive notification of appearance of suitable services that

[2] Semantic Web Rule Language

```
<Alignment>
<xml>yes</xml>
<level>0</level>
<type>**</type>
<onto1>http://www.wine.org/wine.owl</onto1>
<onto2>http://www.vino.org/vino.owl</onto2>
<map>
<Cell>
<entity1 rdf:resource='http://www.wine.org/wine#Wine'/>
<entity2 rdf:resource='http://www.vino.org/vino#Vino'/>
<measure rdf:datatype='&xsd;float'>1</measure>
<relation>=</relation>
</Cell>
</map>
</Alignment>
```

Figure 4.15: Wine to Vino Ontology Mapping

deal with both equivalent concepts.

Chapter 5

Implementation

This chapter details how the design ideas and principles outlined in the previous chapter were implemented. The first section reviews in detail the supporting tools used for implementation purposes. The remaining sections introduce the simple prototype and then enhanced prototype in detail.

5.1 Technology Review

Of the tools examined in the state of the art chapter several were used for implementation. In the first, exploratory phase, of the project the Siena server was used in conjunction with the OWL-S API, The Pellet reasoner and the Jena Framework. The OWL-S API provided a means for parsing services into a Java class structure for easy manipulation. Jena and the pellet reasoner were used for Ontology assertion, integration and manipulation.

As the project progressed to a second prototype the services parsing and integration was merged into the OWL-S matcher Java implementation. This more efficient implementation uses SAX for XML parsing and JESS with the OWLSJessKB knowledge base scripting engine for OWL concept reasoning. Significant modification of the Siena source code was necessary to enable the integration of the matcher and other tools.

5.1.1 OWL-S Matcher

The OWL-S Matcher consists of four Java packages of which two are relevant to this implementation. The first of these is `de.tuberlin.ivs.owl.service`. This package

provides the Java support structure allowing for the parsing of an OWL-S service. The classes `Service` and `Profile` have been modified to include a linked list of ontology URLs which maintain a list of the ontologies that are used by the service and also by the profile.

The class `OWLSParser` provides the SAX based service parsing functionality and creates the instances of `Service` and `Profile`. The package `de.tuberlin.ivs.matching` contains support classes for the matching algorithm implemented in the matcher application. Of these the most useful is the class `Reasoner`. The OWLJessKB is the object through which JESS script initialises assertions.

```
\texttt{kb.executeCommand( "(defquery query-equivalentClass
(declare (variables ?y)) (triple(predicate\"
http://www.w3.org/2002/07/owl#equivalentClass\")
(subject ?x)(object ?y)))" );}
```

The text above shows an example of an equivalent class query set up to interact with the knowledge base. The `loadOwlFile(URL path)` is the method by which any OWL file may be integrated to the knowledge-base. The listing below illustrates the most important method of the reasoner in the context of this implementation.

```
public int conceptMatch(String conceptA, String conceptB) {
if (conceptA.equals(conceptB) || sameClass(conceptA,conceptB)) {
      return EQUIVALENT;
   } else if (subsumes(conceptA,conceptB))  {
      return SUBSUMES;
   } else if (subsumes(conceptB,conceptA))  {
      return SUBSUMES_INVERT;
   } else {
   return FAIL;
   }
}
```

This method captures the concept-matching functionality desired in our Siena implementation, returning equivalent, subsumes, subsumes invert or match failure when given two fully-qualified OWL concepts. The method `subsume(String concepta, String`

conceptb) queries the knowledge base and the underlying JESS reasoner actually proves the assertion.

5.1.2 Siena Java Implementation

Two main packages come bundled with the Siena Java source code. The first of these is the underlying communications layer packaged in `siena.comm`. This package contains the classes `PacketReceiver` and `PacketSender` which are abstract. Concrete implementations of these classes result in Siena supporting TCP communications via `TCPPacketSender`, UDP Communications via `UDPPacketSender` and SSL based communications via `SSLPacketSender`.

The unit of Siena communications is packaged in `Siena` is based on the `SENPPacket` class. Each SENPPacket is capable of transmitting instances of the `Filter` and `Notification` class. The SENPPacket is encoded manually by the implementation i.e. given a set of filters or notifications code to generate a packet stream is provided. This is probably a legacy from an older version of the code as implementing the Java `Serializable` interface for the SENP packet and related instance variables would have the same effect.

The `HierarchicalDispatcher` class in the `Siena` package is the core of the implementation logic. `HierarchicalDispatcher` includes the subscriber poset, a reference to master node, an instantiation of the communications layer objects and a hash-map of its subscribers interfaces amongst its member variables. By default five threads listen for Siena packets to be broadcast from subscribers, other nodes or publishers. Various codes are used to distinguish the function of the packet. Among these `SUB`,`PUB`,`UNS` and `CNF` indicate subscription, publication, unsubscription and configuration respectively.

Subscription

Upon the receipt of a `SENP.SUB` type SENPPacket the `subscribe` method is called. The first goal of the subscribe method is to map the subscribers notifiable object associating it with a reference point in a hash-map. Should a contact for the subscriber not be present in this map then a new class `Interface` is created with the subscriber identification and a `PacketSender`. This is again inserted into an instance of class `RemoteSubscriber`. Having mapped the subscriber a method invocation on the `Poset` inserts the subscriber into the subscription poset and an instance of class `Subscription` is returned. Should this be

a root subscription it is then forwarded through the master node interface. The subscription process is now complete.

The `Poset` class features an exact implementation of the poset graph shown in earlier chapters. Methods to connect, disconnect, insert, and remove subscriptions are all present. For a more detailed view of this, the reader is referred to the `poset.java` class on the accompanying CD.

Publication, Matching and Notification

As outlined, all publications are immediately forwarded to the parent node of the current node, should it exist. Upon publication, the notification is extracted from the siena packet and `matchingSubscribers(Notification n)` method invoked on the subscriber set. This method conducts a breadth first search through the poset starting at the root subscriptions as per outlined in chapter 2. The resulting `SubscriberIterator` contains the all the relevant instances of
`RemoteSubscriber`. This iterator is traversed and the `notify(Notification n)` method invoked resulting in a notification encoded and sent to listening notification handlers via the chosen concretisation of the `PacketSender` class. A `ThinClient` utility class is provided for clients that do not wish to join the network as a node. `ThinClient` provides a listener interface and tracks objects on the same JVM via the `LocalSubscriber` class in much the same way the Hierarchical dispatcher tracks remote subscribers. However, the `notify(Notificaton n)` method may be invoked locally and not over the network in this instance.

5.2 Exploratory Implementation

The exploratory implementation was based on edge-based matching on standard Siena notifications. No explicit matching takes place on the Siena server. Instead, during publication, concepts are parsed and extracted at the edge of the network. OWL concepts are extracted and sub-classes and super-classes of these concepts are then registered with the Siena node. The package `ie.cs.tcd.owlsiena` consists of classes which provide a wrapper around the Siena server.

Publication

The `ServicePublisher` class provides a wrapper to the Siena notification server. Its constructor is provided with a reference to the Siena server and a URI to the OWL-S service that is to be parsed and published. A `readService(URL)` method parses an OWL-S service using the stream based OWL-S reader imported from the OWL-S API package. From this service, a list of inputs, outputs, preconditions and effects are extracted and placed in a the holder class `IOPERecord`. The class `NotificationGenerator` contains an instance of the Jena `OntModel` class and this is where our ontology model will be initialised for publication. Once this has been initialised reasoning then takes place on the inputs and outputs presented to the ontology model. A `getFilters()` method then generates filters based on parsed service inputs, outputs their sub-classes and super classes. Figure 5.1 shows how the input Book from our publication ontology is extracted as a text-based Siena notification. Extracting subclasses and super classes of inputs and outputs was the first step towards incorporating generalised concept matching, whereby exact inputs/output concepts as well as input/output concepts that may have a subsumption, or inverse subsumption relation are considered. Figure 5.1 shows the notification generated by the class structure for a service that has an input book and an output author. This representation is constructed from Siena string data-types as follows :

- **hasExactInput/hasExactOutput** - The specific parameter concepts.
- **hasSubInput/hasSubOutput** - The subclasses of the parameter concepts as expressed in the input OWL-S service ontology
- **hasSuperInput/hasSuperOutput** - The super classes of the parameter concepts as expressed in the input OWL-S service ontology
- **ServiceURI** - From where the Web-Service description may be downloaded.

The code listing below illustrates how these notifications are extracted through Java.

```
/* New service publisher Siena reference and URI */
ServicePublisher sp = new ServicePublisher(sienaServer, \
new URI("http://localhost:8080/owl/WineFinder.owl"));
Service service = sp.getService();
```

```
/* publish the service */
sp.publishService();
```

Figure 5.1: First prototype service notification

Subscription

Subscription in the first prototype involves explicit instantiation of the `IOPERecord` class. Explicit constraints are specified by providing the fully qualified URI of the OWL concept input desired. The code listing below illustrates how this can be done in the first prototype implementation.

```
IOPERecord record = new IOPERecord();
record.addInput("http://www.aktors.org/portal.owl#Book");
record.addOutput("http://www.aktors.org/portal.owl#Author ");
OWLSMatcher handler = new OWLSMatcher(record);
/** generate our subscription */
Subscription sub = new Subscription(handler,record);
/** new subscriber service */
ServiceSubscriber subscriber = new \
ServiceSubscriber(sienaServer,sub);
/** subscribe to the service */
subscriber.registerSubscriptionFilters();
```

Evaluation

The net effect of a publish/subscribe interaction in this prototype is that subscribers receive notification of the publication and modification of web-service notifications that are in any way related to their expressed constraints in the subscription hierarchy. Upon receipt of notification it is then up to the subscribing agent to implement a matching algorithm to determine the quality of the match. While this does somewhat fulfil the goal of a publish/subscribe system for service discovery in the context of the research aims the implementation is lacking. Firstly, no attempt is made to utilise OWL-S information to enhance content based routing. Secondly, the expressiveness of subscription language has not been enhanced. On the contrary, limitation to the types explained in previous chapters actually restricts the expressiveness of the Siena subscription language. Thirdly, there is no support for ontology integration and duplication of information and multiple unnecessary parse requests exist across the network.

To conclude, this exploratory implementation was a useful exercise in analysing some of the reasoning and event notification tools available but, especially given the design specifications in the previous chapter falls short of the research goals. In light of this, the first prototype was shelved and a second, enhanced prototype constructed the details of which are outlined in the following sections.

5.3 The Enhanced Prototype

The second prototype is much more of an attempt at realising the goals outlined in the first chapters of the dissertation. An OWL-S subscription support Java package was developed intended as an extension to the `Subscriber`, `Subscription`, `Filter` and `Notification` classes introduced in the previous section. The next step was to re-write the communications layer to support TCP communication via the `Serializable` Java interface. OWL-S matching was then integrated to the now modified Siena server. Having integrated the OWL-S matching capability, subscription and publication handlers were then modified to handle OWL-S based requirements.

5.3.1 OWL-S Support Package

The package `ie.cs.tcd.owls.discovery.siena` contains the class `RemoteOWLSubscriber`. This essentially captures the same functionality of `RemoteSubscriber` modified for correct function with the remaining support packages. By writing `RemoteOWLSubscriber` to extend the abstract Siena `Subscriber` we ensure neat compatibility with the Siena subscriber mapping functionality. As a result the subscriber tracking component of the hierarchical server can remain largely unchanged.

`ie.cs.tcd.owls.discovery.structure` contains five classes that enable subscription language enhancement. At the top level `OWLSubscription` replaces the Siena `Subscription` class. Each OWLSubscription has a linked list of ontologyURLs, input and output constraints and a reference to an instance of class `Reasoner` which will be used to extract subsumption relations between concepts.

At the very lowest level `OWLConstraint` captures the *(parameter concept, operator, value)* constraint triple. An operator of type String, an instance of class `Parameter` and a reference to the reasoner is also kept. Four primary methods capture the functionality of the OWLConstraint class and allow ordering relations to be defined between constraints as illustrated in chapter 4. `satisfiesInput(Input i)` and `satisfiesOutputOutput o` proved the capability to execute concept comparisons between inputted parameters and the constraint parameters. The following code listing illustrates this.

```
public boolean satisfiesInput(Input p) {
if (this.operator.equals(OP.EQU)|| \
operator.equals(OP.MORESPEC)|| \
operator.equals(OP.LESSPEC)) {
if  (reasoner.conceptMatch(p.getRestrictedTo(), \
param.getRestrictedTo())==Reasoner.EQUIVALENT) {
            return true;
         }
         else {
            return false;
         }
      }
```

```
   /** more specific **/
   else if (this.operator.equals(OP.MORESPEC)) {
      if (reasoner.conceptMatch(p.getRestrictedTo(),
param.getRestrictedTo())==Reasoner.SUBSUMES) {
            return true;
         }
         else {
            return false;
         }
      /**   less specific **/
      } else if (this.operator.equals(OP.LESSPEC)) {
if(reasoner.conceptMatch(p.getRestrictedTo(),
param.getRestrictedTo())==Reasoner.SUBSUMES_INVERT) {
return true;
}
else {
return false;
}
}
return false;
}
```

Three methods deal with comparing individual constraints against each other. Firstly, `boolean equivalent(OWLConstraint p)` compares two OWLConstraints for equivalency in terms of concept and operator. The `isSucessor(OWLConstraint c` and `isPredecessor(OWLConstraint)` implement the functionality necessary for the relations to hold. Again the reasoner subsumption relation is used. The following code piece gives a snap shot of this functionality.

```
if    (this.operator.equals(OP.MORESPEC))
 && (p.operator.equals(OP.EQU))) {
    if(reasoner.conceptMatch(source,target)
==Reasoner.SUBSUMES)
```

```
        return true;
      else
        return false;
}
```

Since we have now established an ordering relation at the lowest level of granularity it is now necessary to abstract one level and consider multiple constraints on the same input or output. The classes `OWLInputConstraint` and `OWLOutputConstraint` classes each support the `isSucessor(OWLInputConstriaint)` type relations by applying an conjunction to each `OWLConstraint` it has in its constraint linked list. The requirement for matching is that every `Constraint` must hold on every `OWLInputConstraint.` and `OWLOutputConstraint` to which it is applied. This is illustrated in the following code sippet.

```
public boolean matches(Input p) {
   Iterator i = constraints.iterator();
   while (i.hasNext()) {
   OWLConstraint c = (OWLConstraint)i.next();
   if (!c.satisfiesInput(p)) {
      return false;
   }
   }
   return true;
}
```

`ServiceNotificationHandler` is the last of the support classes. The purpose of this class is a simple one. This class implements the Siena `Notifiable` interface and runs as a simple listening thread until it gets woken up by the `ThinClient` class. It is intended that code for the handling of services, the appearance of which has just been notified, takes place here.

5.3.2 Communications Layer

The modified Siena interface should be capable of accepting `OWLSubscription` over a chosen transport protocol. As previously pointed out, the Siena communications layer relies

on manual encoding of packets for transport over the network. In order to simplify somewhat the transportation process this encoding of Siena packets has been removed. Taking advantage of the Java package support structure that exists, we may simply implement the `Serializable` interface and allow Java to take care of marshalling between the client and the Siena server node.

To facilitate ease of integration we also only support communications that take place over the TCP protocol via Java Object streams. This is an important consideration when conducting an evaluation on a wide scale system as it must be established whether the TCP overhead outweighs the less-strict guarantees of the UDP protocol. An examination of Java marshalling performance may also be a desirable endeavour. The `SENPPacket` class has been modified and is now capable of the transmission of `OWLSubscription` classes over a TCP stream.

5.3.3 Matching Integration

The `HierarchicalDispatcher` must be extended to support reasoning and an OWL knowledge-base for concept tracking. The modular way in which the OWL-S matcher has been developed allows us to include an instance of `Reasoner` as a member variable directly with only slight modification to the private java modifiers in the concept matching code. Each subscription in the subscription poset, and subsequent constraints will use a reference to this reasoner to conduct its' concept comparison and matching. This all takes place in line with the concept matching code outlined earlier. This extreme ease of integration was one of the driving factors behind the change from Pellet/Jena based reasoning to JESS/OWLJessKB based reasoning.

5.3.4 Subscription (Client)

Step 1 - Instantiate Parameter Classes

Importing the `de.tuberlin.ivs.service` classes the subscriber must first instantiate the parameter concepts of interest.

```
Output o = new Output();
Input i = new Input();
i.setPropertyName("Input");
```

```
i.setRestrictedTo("http://www.aktors.org/ontology/portal#Book ");
o.setPropertyName("Output");
o.setRestrictedTo("http://www.aktors.org/ontology/portal#Person ");
```

Step 2 - Initialise Notification Handler

An instance of the notification handler class or equivalent must be created and the listener thread started.

```
ServiceNotificationHandler sH1
\= new ServiceNotificationHandler("Subscriber 1 - BookFinder Exact");
sH1.start();
```

Step 3 - Create Subscription and Constraints

Next the subscriber must create a new subscription. Instances of `OWLInputConstraint` and `OWLOutputConstraint` are created and one or more instances of `OWLConstraint` are added to each. In this instance we are expressing interest in all published services with at least two outputs, one input equivalent to `Book` and one output equivalent to `Author`. The subscriber must also provide the ontology where the concepts used are defined.

```
OWLSubscription os1 = new OWLSubscription();
OWLInputConstraint ics = new OWLInputConstraint();
ics.addConstraint(new OWLConstraint(OP.EQU,i,true));
OWLOutputConstraint ocs = new OWLOutputConstraint();
ocs.addConstraint(new OWLConstraint(OP.EQU,o,false));
os1.addInputPredicate(ics);
os1.addOutputPredicate(ocs);
os1.addOntologyURL("http://www.aktors.org/ontology/support.owl")
```

Step 4- Register Subscription

An instance of `ThinClient` is then created, feeding the transport protocol, server address and port number to the constructor. Subscription takes place as follows.

```
ThinClient t = new ThinClient("tcp:localhost:1224");
t.subscribe(os1,sH1);
```

5.3.5 Subscription (Server)

Upon the receipt of a new `SENPPacket` the `subscribe(SENPPacket)` method is invoked and subscriber mapping takes place as per the original Siena implementation. A `RemoteOWLSubscriber` instance is then created and passed to a new subscribe method, `RemoteOWLSubscriber s, SENPPacket req)`.

Step 1 - Ontology Integration

```
integrateOntology(req.sub.getOntologyURLs());
```

The `integrateOntology(LinkedList URLs)` method is invoked and passed the linked list of URLs to the ontologies that are referenced by the subscriber. Due to time constraints, the string-compare method outlined in previous chapters has been implemented only. Provided the ontology has not been previously integrated a direct call to the reasoners `loadOWLFile(URI)` method is placed an the knowledge asserted in the knowlegebase.

Step 2 - Subscription Insertion

```
Subscription sub = subscriptions.insertsubscription(req.sub,s);
```

The subscription is now passed to the modified subscriber poset for examination. The modified predecessor set firstly finds the predecessors and successors in the proposed subscription insertion. The following method is then invoked and the subscription inserted into the correct poset location preserving the partial ordering relation. This is done iteratively across possibly null set of predecessors and subscribers.

```
private void insert(Subscription new_sub,
         Collection pre, Collection post) {
//
// inserts new_sub into the poset between pre and post.  The
// connections are rearranged in order to maintain the
// properties of the poset
```

```
//

Subscription x; // x always represents something in the preset
Subscription y; // y has to do with the postset
Iterator xi, yi;

   if (pre.isEmpty()) {
       roots.add(new_sub); // root subscription!
       roots.removeAll(post);
   } else {
       xi = pre.iterator();      /
       while(xi.hasNext()) {
      x = (Subscription)xi.next();
      yi = post.iterator();
      while(yi.hasNext())
          disconnect(x, (Subscription)yi.next());
      connect(x, new_sub);
       }
   }
   yi = post.iterator();
   while(yi.hasNext()) {
       y = (Subscription)yi.next();
       connect(new_sub, y);
   }
   ++mods_since_save;
    }
```

Step 3 - Distribution

The subscription is returned to the handling method and then tested. As per design, if the subscription is a root subscription and our master interface exists, then the subscription is forwarded through the master interface to the parent node.

5.3.6 Publication (Client)

Client side publication also requires an instance of `ThinClient`. Firstly the web-service must be parsed and service profiles extracted.

```
Profile pf1;
final String publishURL1 = \
"http://localhost:8080/owl/BookFinder.owl";
OwlsParser op = new OwlsParser(System.out);
Service s = op.parse \
(new URL(publishURL1),new Reasoner(null));
s.setURL(publishURL1);
Vector v = s.getProfiles();
Iterator it = v.iterator();
while(it.hasNext()) {
   pf1 = (Profile)it.next();
   pf1.setServiceURL(publishURL1);
   t.publish(pf1);
}
```

We must then loop through the list of service profiles and individually publish them to the Siena server via the thin-client.

5.3.7 Publication (Server)

All publications are first forwarded to the master server as per the Siena specification. The OWL-S service and related ontologies are integrated into the knowledge base in a similar fashion as with subscriptions. The service profile is then passed to the Subscriptions poset instance and the set of subscribers we wish to notify is returned. The `(matchinSubscribers(Profile p)` method is responsible for this matching.

First, the covering roots are added

```
while(i.hasNext()) {
sub = (OWLSubscription)i.next();
if (sub.coversProfile(p)) {
 to_visit.addLast(sub);
```

```
    result.addAll(sub.subscribers);
     }
}
```

then we iterate through post sets

```
while((li = to_visit.listIterator()).hasNext()) {
     sub = (OWLSubscription)li.next();
     li.remove();
     i = sub.postset.iterator();
     while(i.hasNext()) {
    OWLSubscription y = (OWLSubscription)i.next();
    if (visited.add(y) && y.coversProfile(p)) {
        to_visit.addLast(y);
        result.addAll(y.subscribers);
    }
     }
}
return result;
}
```

Once the subscription subset has been defined we send a copy of the published service profile to each subscriber node. All our child nodes with root subscriptions that cover the profile also receive a copy of this published service provided they are not the original senders of the service.

5.4 Observations

Due to the nature of integration and the development of glue code across two differing implementations, in software engineering terms, this implementation is inconsistent in its application of the principles of object-oriented design. As a result there is potential to increase efficiency and thus performance. However, having said this, the matching capability implementation is a modular one, and the currently integrated matching capability may be replaced by any concept matching engine that implements a `conceptMatch(String a, String b)` method combined with support for a `loadOWLFile(URL)` method.

The first suggestion for improvement applies at the matching stage of the enhanced implementation. The code in the previous section illustrates how subscriptions are matched against profiles in a breadth first manner. This approach is also explicitly outlined in [7]. However, it may be more beneficial to conduct this matching in a depth first manner. Application of the covering semantics and the subscription language and reasoning can be considered an expensive process. The original subscription language requires execution of a significant amount of code to conduct each match evaluation. While on the surface the enhanced version requires less explicit code to execute, any request of the JESS rule engine to perform a concept match should also be considered an expensive call. There is scope in this implementation to reduce the number of explicitly requests to the JESS engine. While time constraints were prohibitive in the integration of correctly working code for delivery, the pseudo-code solution below illustrates the application of our breadth first principle.

```
for each root in sub_set
   if root.covers(profile) then
   find the last predecessor(s) to root
      for each predecessor p
      for each successor s of p
      find first s or p covering profile
      when found
         add all unique predecessors of s
      next
      next
next
```

The idea behind the above optimisation is that according to the ordering restrictions on the poset, if we find the deepest match in our tree then it follows immediately, without matching, that all the successors, that is more general subscriptions, will cover the profile matched. This optimisation is of benefit particularly in nodes where similar subscriptions are plentiful and subscribers are numerous i.e. the case where the poset tree is a deep one. In the opposite case, the depth first approach becomes closer to a breadth first one and the performance penalty is not hugely significant.

The second observation concerns the client and server side publication algorithms. Cur-

rently, a service is parsed on the client side and individual profiles sent to the Siena server. In the case where related ontologies have not been integrated into our knowledge base, the service must again be parsed and the ontologies integrated. There is a clear overlap here and a definite duplication of activity. The proposed solution to this is very much a trade-off. If we provide a publication interface that allows the OWL-S service URL alone to be sent to the server, then all the parsing may be done on the server side. In the case where there are multiple profiles we may also wish to specify which of these profiles to expose to the Siena server. The trade off comes when we consider scalability. Service parsing, for XML is an expensive process and must be kept minimal. A busy Siena server may wish to alleviate the necessity to parse by encouraging parsing on the client side and subsequent publication via the original interface. Again, time constraints have prohibited the integration of such a solution into the implementation.

Chapter 6

Testing and Evaluation

This chapter discusses the testing and evaluation that has been conducted on the enhanced Siena prototype. The first section explores the Siena evaluation in [7] and its omissions concluding with desired evaluation focus. The remaining sections focus on test scenario, evaluation and results.

6.1 Testing

The implementation required testing on several fronts. Firstly, it was necessary to confirm the correct operation of the communications layer. A simple set of tests to confirm server-to-server communication and client-server communication showed that the communications layer modifications were correctly implemented for TCP communications only. Only slight modifications are needed to implement the UDP protocol.

The second set of tests confirmed the functionality of the subscribe method in terms of subscriber tracking and subscription insertion. All tests verified that subscriber tracking functionality had been correctly implemented. The functionality of the subscription ordering relation support functions was also confirmed.

Tests conducted to verify matching functionality confirmed the correct operation of the reasoner and correct implementation with regard to each of the enhanced subscription language operators.

Lastly, it was necessary to confirm the function of the subscription and publication forwarding mechanism. After a slight modification to the `Notifiable` interface, correct

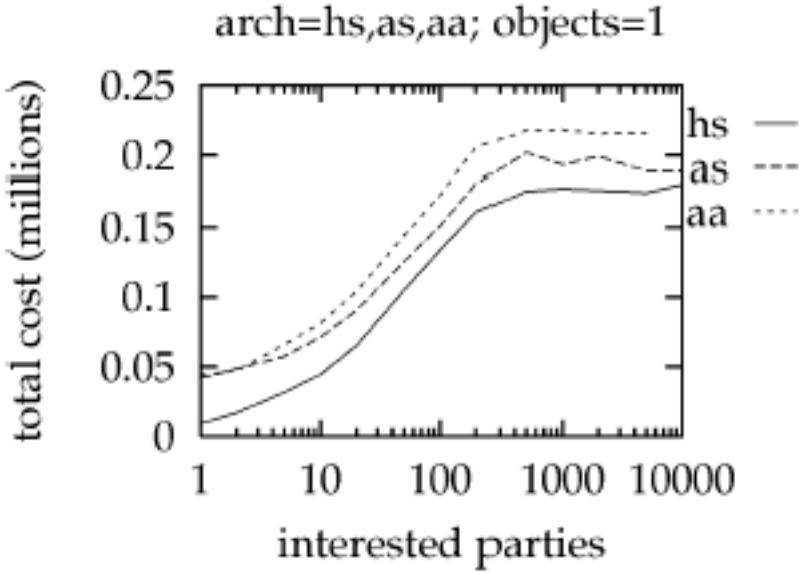

Figure 6.1: Total Network Cost

function of the forwarding mechanisms were confirmed.

All tests conclude that the implementation correctly provides a wide-area OWL-S based publish/subscribe model for service discovery including an enhanced subscription language for specification of requirements. Ordering relations and subscription structure optimisations remain intact, however the enhanced prototype can no longer match and deliver simple, text based Siena notifications.

6.2 Evaluation Focus

6.2.1 Siena Evaluation

The fundamental measurement behind the evaluation conducted in the original Siena server was the *Total Cost* in providing the service. The total cost is calculated by summing the costs of all site-to-site message traffic. In terms of total cost the hierarchical architecture has a threshold where total cost becomes approximately constant. This occurs at about the 200 interested parties level as illustrated in figure 6.1. This is called saturation point and represents the point where interested parties and objects of interest are very likely to be at

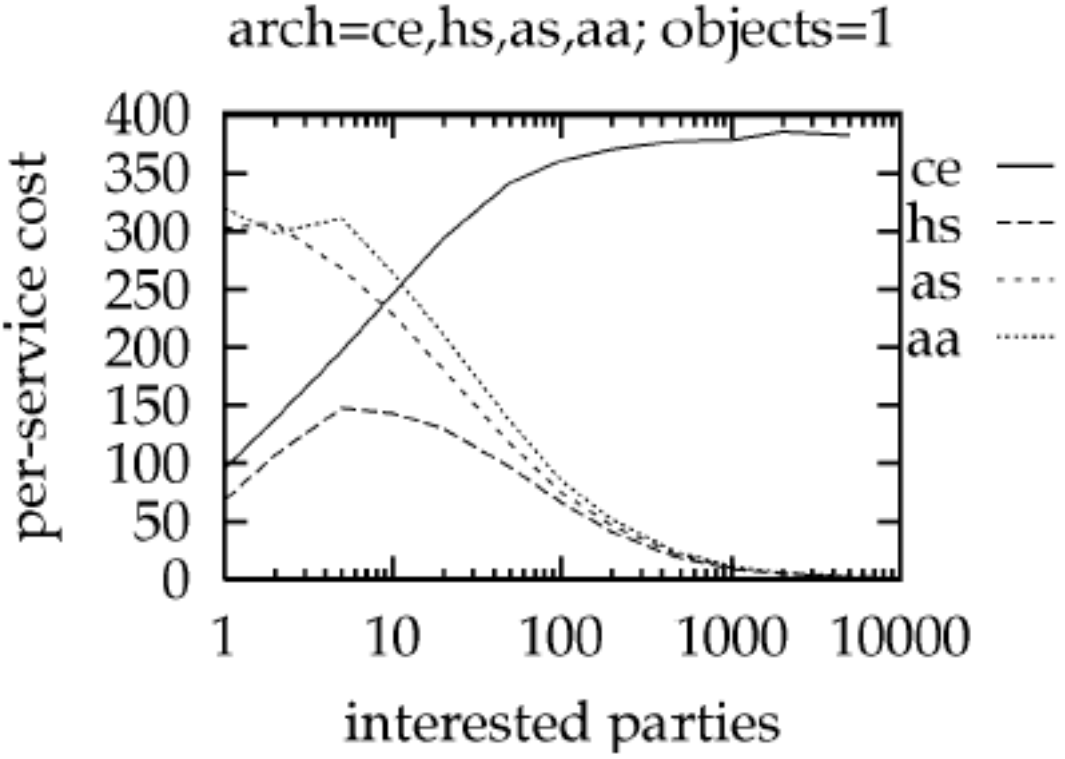

Figure 6.2: Cost Per Service

the same site, or close by. While this metric is a very effective way of illustrating how a structured content-based routing approach may improve on a general-flooding approach in network terms, it is questionable whether total cost of providing the service should be a network traffic measurement alone. It is the opinion of the authors that the matching algorithm and its complexity are vital factors when considering total cost. Any assumption of infinite computing power over such matches (which appears to be implicitly the case here), especially in the case where both high network loads and thus high CPU utilisation loads exist, will most definitely skew the results. It is expected that this characteristic curve as it is pictured will remain generally the same. However, transporting parsed service profiles and accessing sites across the internet not only adds to network costs and processing time costs, but may also introduce a high variance in terms of service processing frequency making it difficult to conduct a total-cost evaluation in the same circumstances.

We continue to the *Cost Per Service Request* metric which, according to Carzangia et al [7], measures how effectively the service amortises cost of satisfying new requests over

the cost of satisfying previous requests. In real terms this metric is tracking how effectively the poset data-structure and the subscription distribution mechanism work with the optimisations outlined in previous chapters. Figure 6.2 shows that this amortization is definitely an effective one in the hierarchical architecture, particularly when considering large amounts of subscriptions and large amount of interested parties, a scenario expected of a wide-are web-service discovery platform. It illustrates that as we approach 1000 subscriptions the per service request cost has decreased dramatically. Since hierarchical architecture distribution and the partial order relation mechanisms have been preserved in making the enhanced prototype OWL-S aware, it is expected, aside from network cost increase, that this characteristic curve will remain static assuming identical evaluation scenarios.

The *Per Subscription* and *Per Notification* costs measure the total network cost per subscription and notification process within the Siena server. Both of these metrics have similar characteristic curves to figure 6.2. It is expected that both of these metrics, per unit, will cost, in general, more in the enhanced prototype. This increase in cost is due to ontology integration and service sourcing that must take place.

6.2.2 Further Evaluation

Since, due to time and other constraints, replication of the Siena evaluation scenario was not possible, we defer a complete evaluation in terms of the above metrics to further work. In addition, the authors would like to see an evaluation of the above metrics in a high-network/high-CPU load environment outlining the effect on characteristic curves, if any. Another very interesting metric in the context of this dissertation is the variance of network cost per subscription and per notification given a statistical analysis of the frequency of parsing and integration of the same OWL-S service and OWL ontologies.

Time constraints have only allowed a smaller scale evaluation of the enhanced Siena prototype. The primary focus of the evaluation metrics is on match time and ontology integration time, metrics either totally ignored or not relevant in the context of the original Siena evaluation.

6.3 Evaluation Scenario

Figure 6.4 and 6.3 illustrate our two evaluation scenarios. Figure 6.4 shows a single node Siena set-up and figure 6.3 shows the distributed model. The effect of publishing to both the centralised and the hierarchical model should be exactly the same.

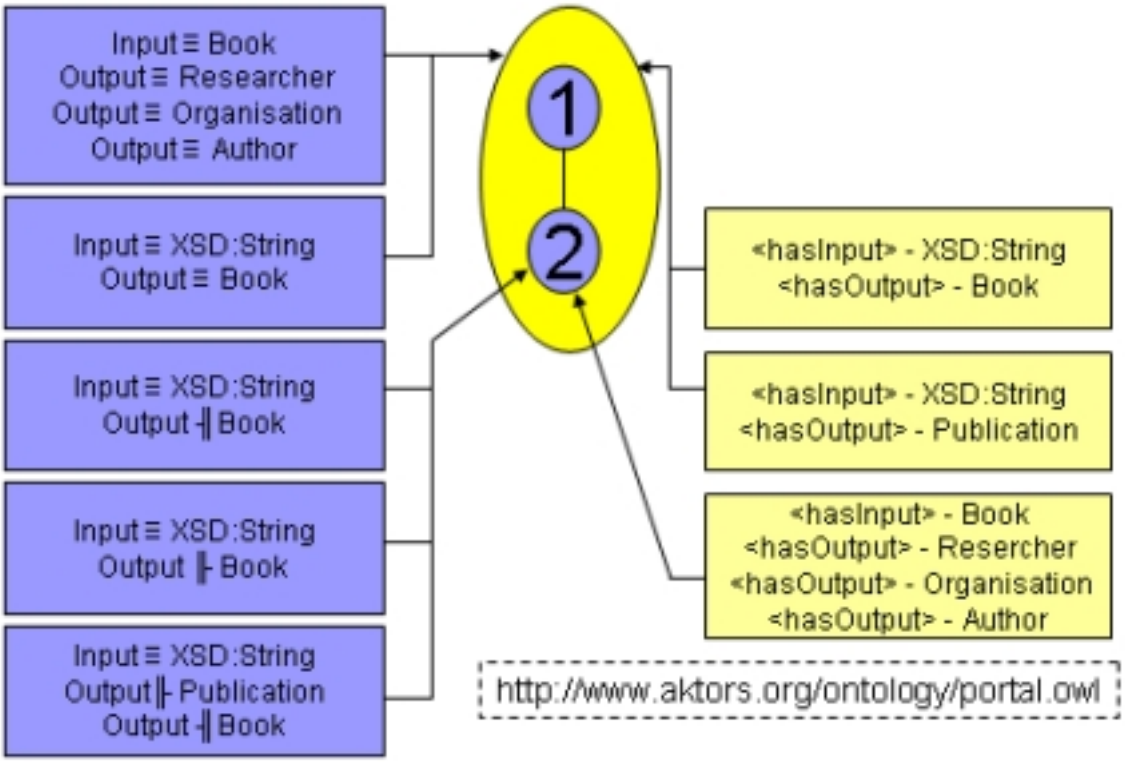

Figure 6.3: Two Node Test Scenario

These evaluation tests correspond to the files `sienatest.java`, `sienatesta.java`, `sienatestb.java` and `StartServer.java` on the accompanying CD. Each node was hosted on a Dell D400 notebook with JDK 1.5, an Intel M 1.2Ghz processor and 256MB of RAM with minimum resident programs.

6.4 Test Results

The average matching time for the most complex subscription over 10 separate runs was 23ms, with the lowest 10ms (least complex) and the longest 28ms. This match time includes

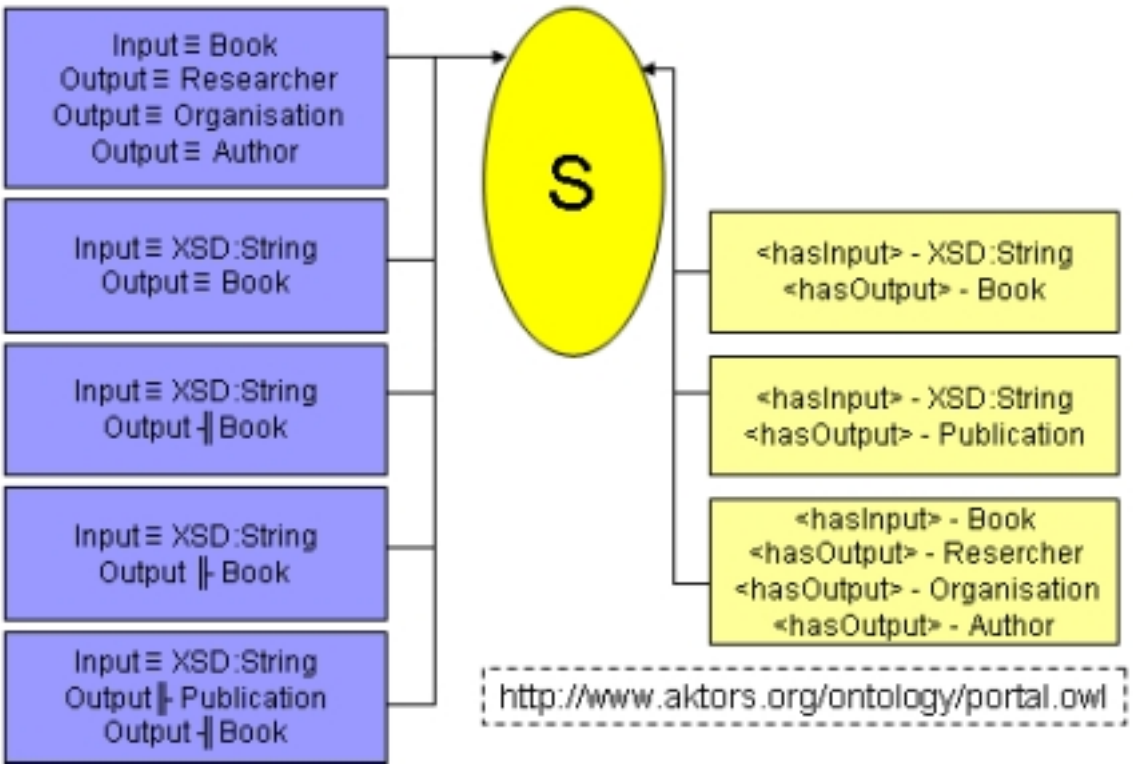

Figure 6.4: Single Siena Node Scenario

the period of traversal through the poset structure and, possibly multiple, calls to the reasoner for the capability match. Although a loose comparison, a covering match on a 5 subscription original Siena implementation (text-based) takes on average 8ms. This comparison can only be considered indicative of the more complex nature of a capability match and a full, larger scale study is needed to confirm if or how this matching time scales linearly. Initial indications, i.e. by inspection, indicate that capability matching does not hugely increase the amount of processing required to match across the poset structure. This can be considered an encouraging result.

The most interesting performance bottleneck occurs when we consider ontology integration time. Integration of OWL ontology, and parsing of an OWL-S service seems a very expensive process. In our sample implementation integration of a standard OWL ontology, http://www.aktors.org/portal.owl, takes on average 1.2 seconds and requires a 98kb download. On the surface of this measurement it seems that ontology integration is definitely a bottleneck when it comes to analysis of matching performance in this system. When the

system is scaled to thousands of interested subscribers over thousands of services this inefficiency has the potential to overload servers.

Another interesting observation is the time it takes to parse and load an OWL-S ontology. On average, our simple book finder service, devoid of pre-conditions, effects, complex assertions and conditional executions, takes 12.5s to parse and load into the knowledge base from a server running on localhost. It is speculated that one of the reasons why this takes so long is the insistence of the OWL-S reasoner in loading, parsing and integrating all ontologies that are imported and referenced by the service itelf. In this case, the aktors ontology was loaded and parsed despite already being asserted in the knowledge base. A similar scenario occurs when we consider the OWL-S specification and the XSD specification. This metric would indicate that the system as a whole would benefit from a finer level of OWL parsing and loading granularity.

6.5 Evaluation Conclusion

In conclusion, the tests confirm the correct function of the OWL-S enhanced wide area notification service in terms of OWL-S matching, enhanced Subscription Language and subscription and publication distribution.

However, performance evaluation conducted was not on a large enough scale to draw valid statistical conclusions regarding increases in network load and network cost as well as CPU cost, however initial tests indicate that parsing and integration of Web-Services and associated ontologies is a definite issue to be considered in future implementations. The final chapter outlines some avenues for future work concerning this finding.

Chapter 7

Conclusions

7.1 Research Conclusions

There were four main aspects of research to this dissertation each of which has been outlined in chapter one. Here, a review of the research and implementation is conducted and research conclusions are drawn.

7.1.1 A Semantic Alternative to UDDI

This dissertation has outlined the need for a semantic alternative to the UDDI model and has shown, through discussion of active research in the area that there is, in fact, a large push to semantically enhancing the Web-Services paradigm as a whole. We have shown that more tightly integrated, distributed models for service discovery are possible and are feasible. We conclude here that active research into this domain, and particularly into a more general wide-area UDDI model, or similar alternative, should continue.

7.1.2 A Publish/Subscribe Model for Service Discovery

By developing a service-discovery platform that uses the publish/subscribe model we have shown a pro-active approach to service discovery that unites research in the service discovery and publish/subscribe domains. We have shown by evaluation that the model holds strong in the presence of OWL-S based capability matching at the core of the event notification system. It is concluded here that in order to solidify this model for discovery, further evaluation of

implementation and further research into application of the more realistic cyclical peer-to-peer architecture must take place.

7.1.3 Efficient, OWL-S aware Content Based Routing

Through semantic-enhancement of the Siena subscription language it has been shown that OWL data-types and OWL concept reasoning can be used to implement content-based routing and, more importantly, this content based routing maintains much of the characteristics of content based routing on simpler data-types notifications. The Siena subscription-storage structure and subscription distribution algorithm can be used to make our publish/-subscribe UDDI alternative scalable while maintaining an expressive way of specifying general content-requirements based on OWL concepts.

7.1.4 Ontology Alignment

At various points throughout this dissertation it has been shown that ontology alignment and ontology integration are vital factors in the realisation of any wide-area publish/subscribe system for semantic service discovery. We have suggested integration of an XML format for ontology concept mapping and outlined the need to minimalise the frequency of ontology integration. It has been shown that even in a small ontology based system such as the enhanced Siena implementation, re-assertion of knowledge-base axioms and relationships must be kept at a minimum through a finer granularity of control over knowledge-base assertions.

7.2 Future Work

While some of the questions posed by the research objective have been answered, as work on this dissertation has progressed further avenues for research and evaluation have emerged.

7.2.1 Evaluation

Time and technology constraints have been prohibitive in terms of conducting a full evaluation of the implementation developed. The primary goal of a further evaluation must be an analysis of the performance of a scaled version of the enhanced system. This evaluation may be conducted in a similar fashion to that conducted in [7]. Another interesting metric in such

an evaluation would be both the network and CPU costs of providing a multi-node wide-area service discovery system under heavy load and network congestion conditions. As outlined in chapter 6, a fuller statistical analysis of ontology integration, in terms of frequency and integration cost is a necessity.

7.2.2 Content Based Routing

Chapter 3 has outlined active research in the area of content based routing. Of particular interest are the fast-forwarding content-based routing algorithms designed for a peer-to-peer implementation of the hierarchical Siena server evaluated. Fast-forwarding algorithms are especially relevant in these more expensive routing topologies. An interesting course of further research would be to consider the OWL enhancements in the context of the routing enhancements of the peer-to-peer architecture that are not relevant to the hierarchical architecture.

One example of this is the application of OWL based subscriptions to the short-circuit-filter evaluation method outlined in [5]. This forwarding enhancement aims to short-cut the amount of filter evaluations and thus matching that must be conducted in terms of the filters poset. Since we have already established that concept matching via the JESS reasoner is an expensive process, research into the application of this forwarding mechanism to the OWL based system could be one solution for increasing the performance of the service-discovery system in general.

7.2.3 Ontology Integration and Alignment

The first obvious course of further work in this area is the actual implementation of the ontology alignment methodology outlined in the design chapter of this dissertation.

Aside from this, there are potential solutions to the ontology integration problems discovered in evaluation of the implementation. Since actual XML parsing of an OWL ontology is an expensive part of an ontology integration, work on minimising XML parsing across the system would be beneficial to the performance of the system as a whole. One possible solution to this would be to ensure once-only semantics for XML parsing on any one OWL-S service or OWL ontology. One method of doing this involves two steps. Firstly, an efficient marshalling algorithm must be devised whereby an asserted OWL knowledge base can be transmitted across the network to other nodes. This has an immediate effect in reducing

XML parsing frequency provided, as expected, the cost of marshalling outweighs the cost of XML parsing and integration conducted across service-discovery system.

Having implemented efficient transmission of a populated knowledge-base it is suggested here that an intelligent means of sharing ontology information be developed. In an ideal scenario, knowledge bases of information, and ontology mappings, should be pushed towards nodes in the network with conceptually similar information. This has two primary effects. By the principle of locality of reference, the more complete knowledge-bases in specific areas of the network become better able to provide accurate concept matches to their subscribers. Further to this, automatic routing of knowledge and maps provides the subscriber with a larger base of information from which to construct ontologies and through which to map concepts. It can be postulated here that, over time, this will have the net effect of reducing the frequency of knowledge based information, reducing cost per subscription and cost per publication metrics in a large-scale set-up. In a similar vein, any caching mechanism based on

7.3 Final Remarks

The semantic web service discovery is very much at the bleeding-edge of semantic web service research. As popularity of the web services paradigm increases we believe a standard for web service discovery for semantic web-service descriptions should be presented.

Our research and evaluation has shown that the publish/subscribe model for OWL-S service discovery opens up avenues for wide area service discovery platforms, optimised routing algorithms, expressive subscription languages and pro-active discovery and integration. We conclude, overall, that any push towards a standard for semantic web service discovery must include further investigation into the models that the publish/subscribe paradigm provide. We believe that work around this model can provide significant contributions to the development of a single standard for semantic web service discovery.

Bibliography

[1] Rama Akiragiu, Richard Goodwin, Prashant Doshi, and Sasha Roeder. A method for semantically enhancing the service capabilities of uddi. *ACM*, 2003. http://dali.ai.uic.edu/pdoshi/research.

[2] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. The descripton logic handbook - theory, implementation and applications. January 2003.

[3] Tim Berners-Lee and Hendler J. Lassila. The semantic web. *Scientific American*, 5/02, May 2001.

[4] Colm Brady. Runtime discovery, composition and invocation of web-services using semantic descriptions. Technical report, Univeristy of Dublin,Trinity College, 2004. http://www.cs.tcd.ie/techreports.

[5] Antonio Carizangia and Alexander L. Wolf. Forwarding in a content-based network. *ACM*, 2003.

[6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based address and routing: A general model and its application. Technical report, University of Colorado, Boulder CO, University of California, Irvine,CA, January 2000.

[7] Antonion Carzaniga, David S. Rosenblum, and Alexander L. Wolf. SIENA : The Design and Evaluation of a Wide-Area Event Notification Service. *ACM*, 2001.

[8] Erik Christenson, Francio Garber, Gre Meredith, and Sanjeeva Weerawaran. WSDL : Web Services Description Language. http://www.w3c.org/TR/WSDL.

[9] DAML Technical Committee. OWL-S : Web Ontology Language for Web Services. http://www.daml.org/services/owl-s/.

[10] RDF Technical Committee. The resource description framework. http://www.w3.org/RDF/.

[11] SOAP Committee. SOAP : Simple Object Access Protocol. http://www.w3c.org/TR/SOAP.

[12] The OASIS Committee. Introduction to UDDI: Important features and concepts. October 2004.

[13] UDDI Technical Committee. UDDI API specification v2.0.4. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf.

[14] UDDI Technical Committee. UDDI API specification v3.0.2. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf.

[15] OWL Technical Committeee. OWL web ontology language. http://www.w3.org/2004/OWL/.

[16] The World Wide Web Consortium. http://www.w3c.org.

[17] The E-bay Web Services API. http://developer.ebay.com/common/api.

[18] David Booth et. al. Web services architecture technical report, 2004. http://www.w3c.org/TR/ws-arch/.

[19] Jerome Euznat et al. The state of the art on ontology alignment. Research conducted for INRA - http://www.inrialpes.fr.

[20] Kunal Verma et al. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. Large Scale Distributed Information Systems (LSDIS) Lab Department of Computer Science, University of Georgia Athens.

[21] Patrick TH. Eugster, Pascal A. Felber, and Anne-Marie Kenmarrec Rachid Guerrout. The many faces of publish/subscribe. *ACM*, 2003.

[22] Jerome Euznat. The ontology alignment api, June 2005. Research conducted for INRA - http://www.inrialpes.fr.

[23] eXtensible Markup Language. http://www.w3.org/XML/.

[24] RACER An Inference Engine for the Semantic Web. http://www.franz.com/products/racer/.

[25] HP Web Services Management Framework. http://devresource.hp.com/drc/specifications/wsmf/ind

[26] Michael C. Jaeger, Gregor Rojec-Goldmann, Christoph Gero Muhl, , and Kurt Geihs. Ranked matching for service descriptions using owl-s. Technical report, TU Berlin, Institute of Telekommunication Systems.

[27] JENA. A semantic framework for java. http://jena.sourceforge.net/.

[28] Joe Kopena. OWLJessKB : OWL Reasoning for JESS. http://edge.cs.drexel.edu/assemblies/software/owljesskb/.

[29] Sandia National Laboratories. JESS - The Rule Engine for Java. http://herzberg.ca.sandia.gov/jess/.

[30] MINDSWAP. Pellet : An OWL Reasoner. http://www.mindswap.org/pellet.

[31] Mindswap. The OWL-S API. http://www.mindswap.org/2004/owl-s/api/.

[32] Massimo Paolucci, Takahiro Kawamura, Rerry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. *ISWC2002*, pages 333–347, 2002.

[33] The DARPA Agent Markup Language Programme. http://www.daml.org.

[34] Bill Segall, David Arnold, Julian Boot, Micheal Henderson, and Ted Phelps. Content based routing with elvin4. University of Queensland, St Lucia, Austrailia.

[35] The Google-Maps Web Service. http://www.google.com/apis/maps.

[36] AWS Amazon Web Services. http://www.amazon.com.

[37] IBM Web Services. http://www-130.ibm.com/developerworks/webservices/.

[38] Microsoft Web Services. http://mdsn.microsoft.com/webservices/.

[39] Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. Adding owl-s to uddi, implementation and throughput. Technical report, Robotics Institute, Carnegie Mellon University, USA.

[40] Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. An Efficent Algorithm for OWL-S based Semantic Search in UDDI. Robotics Institue, Carnegie-Mellon University, USA.

[41] Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. An Efficent Algorithm for OWL-S based Semantic Search in UDDI. Robotics Institue, Carnegie-Mellon University, USA.

[42] The OASIS Organisation. http://www.oasis.org.