RUN-TIME DISCOVERY, SELECTION, COMPOSITION & INVOCATION OF WEB SERVICES USING SEMANTIC DESCRIPTIONS

By

Colm Brady

A dissertation submitted to the University of Dublin, Trinity College in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science.

Trinity College Dublin

September 13, 2004

DECLARATION

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Date: September 13, 2004

PERMISSION TO LEND AND/OR COPY

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Date: September 13, 2004

ACKNOWLEDGMENTS

The author wishes to thank the following people for contributing to this research:

Declan O'Sullivan and Dave Lewis for their guidance and tireless work on my behalf. It would not have been possible for me to complete this work with out their support. Thank You.

Owen Conlon and Ian O' Keeffe for their assistance while I evaluated APeLS.

Andrew Jackson who reviewed this dissertation and provided invaluable critique.

Fellow NDS class mates for their help and opinion through out the year.

And last but most importantly, to my family for their love, support and tolerance during all my education years.

TRINITY COLLEGE DUBLIN

ABSTRACT

RUN-TIME DISCOVERY, SELECTION, COMPOSITION & INVOCATION OF WEB SERVICES USING SEMANTIC DESCRIPTIONS

By Colm Brady

Web Service computing is enabled by using an architecture that provides interoperability between disparate and diverse applications. One goal of Web Services is to facilitate inter-organisational distributed computing using traditional protocols such as the Hyper Text Transfer Protocol (HTTP) and the Simple Mail Transfer Protocol (SMTP). However, these web service technologies do not provide standards for dynamically discovering, selecting candidates, composing and invoking Web Services based on their capabilities. In effect, a human must interpret the functionality or applicability of a web service and write some software capable of using the service or configure a generic client to invoke the service.

Industry is currently proposing several standards which allow for software Agents to automate the process of composing Web Services, by annotating Web Services capabilities in machine readable form and using reasoners to reason over this information.

OWL-S is an emerging XML based mark-up language that can be used to describe the non-functional and functional attributes of a Web Service. OWL-S provides a well defined framework for expressing the capability of Web Services, in a platform and technology neutral representation.

This Dissertation is concerned specifically with researching the OWL-S specification for semantic mark-up of Web Services. Of specific interest are methods and techniques to use OWL-S descriptions for semantic discovery incorporating non-functional service attributes, candidate selection, service composition and semantic service invocation.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

LIST OF TABLES

Number *Page*

Chapter 1

INTRODUCTION

1 Introduction

1.1 Web Service Computing

Web Service computing is enabled by using an architecture that provides interoperability between disparate and diverse applications. Web Services are self-contained, self-describing modular applications that can be published, located and invoked in a dynamic fashion over the Internet [5]. Web Services are enabled by using a set of industry standard, platform neutral specifications such as Universal Description, Discovery, and Integration (UDDI) [1], Simple Object Access Protocol (SOAP) [3] and Web Service Description Language (WSDL) [4].

One goal of Web Services is to facilitate inter-organisational distributed computing using traditional protocols such as the Hyper Text Transfer Protocol (HTTP) [27] and the Simple Mail Transfer Protocol (SMTP) [28] [5]. However, these web service technologies do not provide standards for dynamically discovering, selecting candidates, composing and invoking Web Services based on their capabilities. In effect, a human must interpret the functionality or applicability of a web service and write some software capable of using the service or configure a generic client to invoke the service.

This scenario is one factor in limiting interoperable and global scale distributed applications. Industry is currently proposing several standards which allow for software Agents to automate the process of composing Web Services, by annotating Web Services capabilities in machine readable form and using reasoners to reason over this information.

1.2 The Semantic Web

The Semantic Web is a grand and ambitious vision for how World Wide Web content should be structured, stored, managed and used. The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation. [8]. These capabilities for information expression will usher in significant new functionality as machines become much better able to process and "understand" the data that they merely display at present [8].

1.3 Semantic Web Services

OWL-S [6] is an emerging XML [2] based mark-up language that can be used to describe the non-functional and functional attributes of a Web Service. OWL-S provides a well defined framework for expressing the capability of Web Services, in a platform and technology neutral representation.

This Dissertation is concerned specifically with researching the OWL-S specification for semantic mark-up of Web Services. Of specific interest are methods and techniques to use OWL-S descriptions for semantic discovery incorporating non-functional service attributes, candidate selection, service composition and semantic service invocation.

1.3.1 E-Commerce Enrichment

It is also a goal to provide knowledge and feedback relating to development of an E-Commerce semantic application, as there are still limited examples of semantically enabled Agent applications available on the Internet. There is a requirement to promote practical applications that casual users can benefit from and develop these applications immediately [11].

These requirements will be factored in to an experimental framework which will examine methods for achieving this semantic functionality. The framework will be applied to an E-Commerce portal application to showcase possible use cases for the Semantic Web.

1.4 Research Objectives

From this research, we wanted to address techniques and methods to realise an enriched E-Commerce environment. To achieve this goal we needed to investigate the following research questions:

- How to enable semantic discovery using UDDI.
- How to undertake candidate selection using Semantic Reasoning.
- How to enable Service Composition.
- How to automate Semantic Service Invocation

Before we can discuss the research questions, we must firstly indicate the differences that exist between functional and non-functional Web Service attributes.

Functional is this context means, attributes that are directly bound to the operation and invocation of a Web Service. For example, an input parameter is a functional attribute, because the service needs this value in order to function correctly.

Non-functional in this context of Web Service attributes means, characteristics of a Web Services that are not related to its invocation and usage, but are related conceptually. An example of a non-functional

attribute is a geographical location for a specific service. A user may have a requirement to invoke a service that is located in Ireland, for example.

1.4.1 Enabling Semantic Discovery using UDDI

Current Web Service models facilitate automated discovery of services. This mechanism is purely syntactic and does not enable capability based discovery. On delivery of query results, a human needs to review the discovered services and evaluate the services suitability. This is a limiting factor in automating the discovery process.

Discovery in a semantic context means finding the location of Web Services on the basis of the capabilities that they provide [9]. Essentially, the discovery process needs to be able to find semantically annotated Web Services that will satisfy known user requirements. These requirements will most likely be expressed as semantic concepts.

It is an objective to formulate a method to enable semantic capability based discovery of Web Services using an industry standard called UDDI[1]. We believe that Semantic Web Services will need to leverage current Web Service standards in order for their adoption to be widespread.

1.4.2 Candidate Selection using Semantic Reasoning

A critical element to the discovery process is the ability to evaluate, rank and prioritise the discovered services based on user requirements. Without accurate candidate selection, automation of the discovery process is highly restricted. A method to select services based in non-functional service attributes is of specific interest for this research.

The candidate selection process involves identifying Web Services in some priority ordering that best satisfy a set of user preferences. This process is typically done at discovery and/or composition time. A candidate service is classed as any service that matches any of the capabilities sought. A candidate that meets all the users' requirements will be ranked as the most suited candidate.

It is an objective to examine methods for providing semantic inference functionality. We believe that performance and speed are critical factors in evaluating inference techniques.

1.4.3 Enabling Service Composition

It is conceivable that users will require multiple services to achieve their goals. There are several emerging models for Service Composition. To date, these models have failed to address dynamic

[1] Section 2.2.1

composition of services at run-time. Dynamic service composition at run-time is important because the number of compositions possible is greatly increased over and above the current static models.

Composing existing services to obtain new functionality will prove to be essential for both business-to-business and business-to-consumer applications [10]. The service composition process selects two or more Web Services and amalgamates them to provide new functionality. For example, a typical composition scenario is composing an air reservation service with a hotel reservation service, to produce a new outcome, not possible by using the services individually. In this case, an airline booking and a hotel booking.

It is also a research objective to enable the invocation of a composition of services. This requires using candidate service selection methods to formulate an execution plan that achieves a known semantic encoded plan.

1.4.4 Automating Semantic Service Invocation

The final piece of the problem is that currently, it is impossible to dynamically invoke a series of Web Services at run-time due to a lack of encoded semantic information about the functional attributes of a service. Without this encoded knowledge the semantics of inputs and outputs can not be determined.

Semantic service invocation involves executing concrete Web Services dynamically, using the encoded semantic metadata as an instruction set. This process involves selecting an appropriate operation to achieve a known outcome and also dynamic reconciliation of input parameters and interpretation of message return types.

Our final objective is to provide a service execution engine that is capable of automating the invocation of Web Services using semantic descriptions.

1.5 Dissertation Road Map

Following this chapter, Chapter Two provides the background information relating to Web Services and Semantic Web technologies. Chapter Three analyses the current state of the art in the discovery, candidate selection, service composition and semantic invocation of Web Services domain. Recent research that is related to our work is reviewed and analysed in this chapter. Chapter Four describes the design and solution and also defines an E-Commerce use case that demonstrates the need for such research. Chapter Five discusses the semantic OWL-S framework that is the basis for this Dissertation, including the design rationale and implementation decisions and reasons for these decisions. Chapter

Six evaluates the framework, and details our experiences. Chapter Seven concludes the findings of our research.

Chapter 2

BACKGROUND

2 Background

This chapter provides background information relating to Web Services and Semantic Web technologies. Section 2.1 discusses the history and motivation for the Semantic Web, as well as explaining relevant Semantic Web technologies. Section 2.2 reviews current Web Service standards and Section 2.3 presents an overview of the main concepts of OWL-S 1.0.

2.1 Semantic Web

There has been extensive research in the area of intelligent systems that can represent human knowledge. These systems are characterised as knowledge-based system. Description Logics are a family of knowledge representation languages that have been studied extensively in Artificial Intelligence over the last two decades. They are embodied in several knowledge-based systems and are used to develop various real-life applications [24]. Description Logics is a method for representing and ordering human knowledge in knowledge based systems [24]. Description Logics were developed out of semantic network and frame systems. These systems arranged data in network graphs. This arrangement allowed computer programs to traverse the graphs in different ways, using different algorithms.

As with most network structures, a link can be characterised as a bridge from one point to another. Each link normally connects two nodes together. Nodes are normally characterised as concepts in Description Logics. A connecting link defines the “is a” relationship between two nodes.

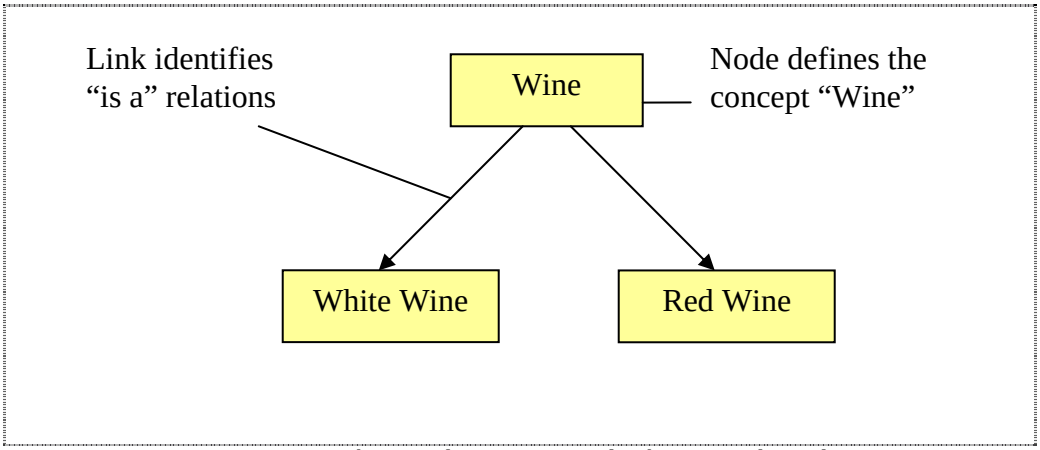

Figure 2.1.1: simplistic graph representation detailing a Wine hierarchy.

One of the important features of Description Logic that is evident in the Semantic Web is that unambiguous conclusions can be ascertained from a description logic graph hierarchy. In Figure 2.1.1 we can infer that White Wine is a Wine, and Red Wine is a Wine also. This kind of relationship is known as a subsumption relationship. The subsumption relationship is discussed in Section 2.1.5.3.

The concepts behind the Semantic Web can also be traced back to work done on Conceptual Dependencies [40]. Conceptual Dependencies, like Description Logic express human knowledge in an unambiguous form and was also applied to traditional models of Artificial Intelligence computing. Conceptual Dependencies are abstract hierarchies of human concepts encoded into a computer readable form [40]. The definition could also be applied to ontologies. **Ontologies are used in the Semantic Web framework to encode human concepts into a human readable form that can be interpreted by a machine or Agent**.

Ontologies are a specification of a conceptualization [12]. They are a description (like a formal specification of a program) of the concepts and relationships that can exist for an Agent or a community of Agents [12].

2.1.1 Concepts

The World Wide Web consortium[2] (W3C) is an Internet task force with the goal of standardising protocols and rules for Internet technologies. W3C standardises the majority of Semantic Web technologies and specifications. This Section, 2.1; presents an overview of some of the W3C specifications.

[2] http://www.w3.org/ - World Wide Web Consortium Home Page

2.1.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) [13] is a general ontology language in which syntax conforms to well-formed XML. XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents. RDF expands XML's Schema to allow for this type of expression.

RDF was developed by the W3C for expressing semi-structured metadata in order to enable knowledge-management applications. An RDF document contains its knowledge in the form of data constructs called triples. Triples identify relationships between abstract concepts. RDF allows for the notion of Classes and Properties. A Class is a definition of some "Thing", where as a Property is an attribute of a "Thing". Uniform Resource Identifiers (URI) [26] is the glue that allows Agents to traverse the Semantic Web. URI's are strings that identify resources in the web [25]. Ontologies use URI's to reference other concepts that exist in perhaps other ontologies on some other endpoint on the Internet. The mark-up in Figure 2.1.2 shows a simple RDF triple. It is possible to deduce from the triple that there exists a concept called Drink, and Wine "is a" Drink.

```
…
<Class ID="Wine">
      <subClassOf resource="#Drink"/>
</Class>
…
```

Figure 2.1.2: simplistic concept for Wine.

A shortcoming of RDF, with regards the Semantic Web is the inability to express sophisticated constructs, such as data-typing of properties, characteristics of properties, enumerations and usage constraints on concepts. These constructs allows ontology providers to encode how a concept should be used and how a computer should represent it in memory. OWL [8] is an extension to RDF. It addresses the RDF short comings and introduces the necessary syntax to express these kinds of constraints.

2.1.3 Web Ontology Language (OWL)

Web Ontology Language (OWL) is an ontology mark-up language that enables the creation of ontologies for any domain and the instantiation of these ontologies for the description of specific Web Resources [6]. It extends the concepts defined in the RDF schema. OWL was initially developed by the

Defense Advanced Research Projects Agency (DARPA) Group[3] under the guise of DAML+OIL. DARPA is an organisation linked to the US Department of Defence with the aim of investigating Agent technologies, among other things. Ontology Inference Layer (OIL) was initially a European initiative, which was merged with the DARPA Agent Mark-up Language (DAML) to form DAML-OIL mark-up. W3C renamed DAML+OIL to OWL and standardised the OWL language in 2003.

2.1.4 Intentional and Extensional Knowledge

Within an ontologies knowledge base, there is a clear distinction between intentional knowledge and extensible knowledge. Intentional knowledge is general terminology built using OWL declarations about the knowledge domain. It expresses general concepts and properties. For example, in a Wine ontology, properties like hasColour, hasFlavour and hasBody are intentional properties. The OWL mark-up in Figure 2.1.3 shows intentional knowledge about White Wine. It states that White Wine must have the colour White.

```
…
<owl:Class rdf:ID="WhiteWine">
   <owl:intersectionOf rdf:parseType="Collection">
     <owl:Class rdf:about="#Wine" />
     <owl:Restriction>
       <owl:onProperty rdf:resource="#hasColor" />
       <owl:hasValue rdf:resource="#White" />
     </owl:Restriction>
   </owl:intersectionOf>
 </owl:Class>
…
```

Figure 2.1.3: OWL class definition for White Wine.

Extensional knowledge introduces the notion of instances of concepts. An instance of a concept is known as an individual. Individuals assert properties and facts about an instance of a concept. The mark-up in Figure 2.1.4 identifies St Genevieve Texas white wine, and assigns values to the general properties of Wine.

[3] http://www.darpa.mil/ - DARPA Group Home Page

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

Figure 2.1.4: individual (instance) of the class Wine.

In Description Logics, intentional and extensional knowledge are referred to as TBox and ABox logic respectively. This is analogous to the terms Class and Object in Object Oriented (OO) [41] development. A class defines the semantics of a thing, almost like a blueprint. An object assigns values to the class and is a real representation of a class type.

2.1.5 OWL Example

To help readers understand OWL syntax and mark-up, a simple example is presented in Figure 2.1.5. The mark-up expresses a concept for “Wine”. It is not a complete definition of what Wine is and some important concepts about Wine have been omitted for clarity.

```
…
<owl:Class rdf:ID="Wine">
    <rdfs:subClassOf rdf:resource="&drink;Drink" />
    <rdfs:subClassOf>
       <owl:Restriction>
          <owl:onProperty rdf:resource="#hasFlavour" />
          <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
       </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
       <owl:Restriction>
           <owl:onProperty rdf:resource="#hasBody" />
           <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:label xml:lang="en">wine</rdfs:label>
   <rdfs:label xml:lang="fr">vin</rdfs:label>
 </owl:Class>

<owl:ObjectProperty rdf:ID="hasWineDescriptor">
    <rdfs:domain rdf:resource="#Wine" />
    <rdfs:range  rdf:resource="#WineDescriptor" />
 </owl:ObjectProperty>

<!--some properties of Wine-->
<owl:ObjectProperty rdf:ID="hasBody">
    <rdf:type rdf:resource="&owl;FunctionalProperty" />
    <rdfs:subPropertyOf rdf:resource="#hasWineDescriptor" />
    <rdfs:range rdf:resource="#WineBody" />
 </owl:ObjectProperty>

 <owl:ObjectProperty rdf:ID="hasFlavour">
    <rdf:type rdf:resource="&owl;FunctionalProperty" />
    <rdfs:subPropertyOf rdf:resource="#hasWineDescriptor" />
    <rdfs:range rdf:resource="#WineFlavour" />
 </owl:ObjectProperty>
…
```

Figure 2.1.5: OWL representation for the concept of Wine.

The information that can be extracted from the mark-up is as follows. The mark-up defines an OWL class called “Wine”. Wine “is a” sub class of Drink. Wine has two functional[4] properties, hasFlavour and hasBody. A property is similar to a realisation of the “has a” relationship used in OO development. The properties extend the hasWineDescriptor property which is in the “domain” of Wine; therefore we can infer that hasFlavour and hasBody are also in the domain of Wine. hasFlavour must have a value of

[4] Functional Property: for each instance of the property there is at most one value for the property

type WineFlavour and hasBody must have a type of WineBody. Each instance of Wine is constrained to only have one value for hasBody and one value for hasFlavour.

It is possible for the class of Wine to be replicated all over the Semantic Web. Each class of Wine can be made distinct and separate due to a namespace typing mechanism using URI's. Agents can decide that the OWL class located at http://somewhere/concept#Wine is not the same concept as http://nowhere/other#Wine. The Semantic Web is an attempt to share knowledge and unify concepts for things. It is more beneficial to have only a few concepts for Wine, rather then several conflicting definitions.

2.1.6 Enforcing Semantic Relationships

OWL allows ontology providers to link two different ontologies together through well defined relationships. Some of the most common constructs allowing this are "equivalency", "same as", "sub class of" and "all different".

2.1.6.1 OWL Equivalency

Equivalency allows ontology providers express that two concepts are equal by intent. It allows Agents to infer that both concepts refer to the same human concept. For example, if two ontologies exist at different URI, one has a class "Wine" and the other has a class "Vino". Let's assume that these classes are similar by nature and express the same concepts but use different naming convention, perhaps different languages. The concepts of both can be linked by using the <owl: equivalentClass> mark-up tag. The properties of class Wine can also be made equivalent to properties in class Vino. Figure 2.1.6 shows OWL mark-up for "equivalency".

```
…
<owl:Class rdf:ID="Wine">
      <owl:equivalentClass rdf:resource="http://somewhere#Vino" />
</owl:Class>
…
```

Figure 2.1.6: how to link a concept defined in an ontology to another equivalent concept.

2.1.6.2 OWL Same Individual As

An OWL instance document can be enhanced with an OWL property to indicate that it is the same as another instance. Using the "same as" relationship allows ontology providers to bind one individual to another individual. An individual is the term used to describe an instance of an ontology concept. This construct would allow two individuals to be substituted for each other with out any conceptual difference. Figure 2.1.7 shows OWL mark-up for a "same individual as" concept.

```
…
<WhiteWine rdf:ID="StGenevieveTexasWhite">
   <locatedIn rdf:resource="#CentralTexasRegion" />
   <hasMaker  rdf:resource="#StGenevieve" />
   <hasSugar  rdf:resource="#Dry" />
   <hasFlavor rdf:resource="#Moderate" />
   <owl:sameIndividualAs rdf:resource="http://somewhere#StGenTexW"/>
 </WhiteWine>
 …
```

Figure 2.1.7: how one instance of a concept can be the same as another.

2.1.6.3 OWL Sub Class Of

A "sub class of" relationship is an expression of classic inheritance. This construct can be used to build hierarchies of human concepts. A sub class relationship is also classed under a subsumption relationship. Subsumption can be seen as the determination of sub concept and super concept relationships between concepts of a given terminology [24]. An example of a sub class relationship is the notion that Wine "is a" Drink. Figure 2.1.8 shows some example mark-up for an OWL subsumption relationship.

```
…
<owl:Class rdf:ID="Wine">
      <rdfs:subClassOf rdf:resource="http://somewhere#Drink" />
</owl:Class>
…
```

Figure 2.1.8: how to specify the "is a" relationship.

2.1.6.4 OWL All Different

Using the OWL "All Different" construct one can assert that a group of individuals are different. This is useful for enforcing the concept that an individual may be of the same type, but the concept it refers to is totally different to the any other concept of the same type. The example in Figure 2.1.9 enforces that real world fact that Wine has distinct colours which are not the same conceptually.

```
…
<owl:AllDifferent>
   <owl:distinctMembers rdf:parseType="Collection">
     <wine:WineColor rdf:about="#Red" />
     <wine:WineColor rdf:about="#White" />
     <wine:WineColor rdf:about="#Rose" />
   </owl:distinctMembers>
 </owl:AllDifferent>
…
```

Figure 2.1.9: how to group a set of different concepts that are of the same class.

2.2 Web Services

Web Services are self-contained, self-describing modular applications that can be published, located and invoked in a dynamic fashion over the Internet. Web Services can be local, distributed or Web-based. They interact with and/or invoke each other, fulfilling specific tasks and requests that, in turn, carry out specific parts of complex transactions or workflows [5].

One goal of Web Services is to facilitate inter-organisational distributed computing using traditional protocols such as HTTP and SMTP. By using such protocols, company firewalls can be negated through the use well known open ports. This allows services that are initially unaware of each others existence; invoke remote messages on each other.

Typically, Web Services are end points to business services and advertise their functionality using WSDL[5] mark-up [5].

2.2.1 Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery, and Integration, (UDDI) is a specification that defines a service registry of available Web Services, serving as a global electronic "yellow pages" for business services.

It is intended to allow companies to advertise and publish a description of their business/service to a directory. Businesses that register with a UDDI server are known as service providers. Service requesters use the service registry to discover a service provider for obtaining arbitrary goods or services. UDDI defines data-types that abstract the concept of a business and a service. It is therefore possible for any class of service to register their business and provide a way for electronic clients to connect to and use the advertised services.

[5] Defined in Section 2.2.2

Classification systems exist in UDDI registries to allow business to be associated with classes of organizations. UDDI is used as a method for publishing and finding service descriptions [5]. As with most new web technologies, a custom XML schema is used to communicate with a UDDI registry. This means that like SOAP, UDDI is technology neutral, and can be implemented in any programming language. Most UDDI servers are exposed as Web Services and can therefore leverage the open architecture nature of SOAP Services.

2.2.2 Web Service Definition Language (WSDL)

WSDL is an XML-based specification schema for describing the operational information of a Web Service such as interface and end points. WSDL defines XML grammar for describing contracts between a set of endpoints exchanging messages. Contracts provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication [5]. WSDL is somewhat comparable to IDL [36] [42] in its description of methods; argument types and return values, in that it describes the implementation details for clients who want to talk to the service. Using WSDL, Web Services can be enabled to access and invoke remote applications and databases [5].

WSDL has recently become wide spread in its use. There are several tools that can automate the creation of WSDL documents, and it has proven to be a very robust and descriptive deployment mechanism. Major vendors of technologies like Microsoft, IBM, Sun, Oracle and BEA have incorporated WSDL into their tool sets and technology standards, as well as a healthy proportion of open source vendors.

2.2.3 Simple Object Access Protocol (SOAP)

The glue that makes Web Services a reality it the transport technology, Simple Object Access Protocol (SOAP). SOAP is a model of using simple request and response messages written in XML as the basic protocol for electronic communication. SOAP messaging is often modelled as a platform-neutral remote procedure call mechanism, but it can be used for the exchange of any kind of XML information [5].

Any server or client can theoretically interpret the XML message and process a SOAP messages payload. SOAP messages can also be transported across networks using standard protocols like HTTP and SMTP, so no object level transport mechanisms like RMI [37] or CORBA [36] need to be used.

2.3 OWL-Services (OWL-S)

OWL-Services (OWL-S) [6] is an ontology defined specifically for marking up Web-Services semantically. It defines concepts that can be used to describe a Web Service such that humans and

computers can reason over the description. It is a DARPA inspired mark-up language built using OWL syntax.

2.3.1 Introducing OWL-S 1.0

During the time period this Dissertation was being researched, OWL-S 1.0 was the latest release of the OWL-S specification [6]. The specification[6] is a good starting point in understanding what the motivation for OWL-S is. This specification document was formulated by the OWL Services coalition (formulary the DAML Services coalition). The coalition's publications detail the current state of development for the OWL-S specification. It is currently at version 1.1, but 1.1 features were not examined during this research. The paper also discusses the motivation for developing OWL-S and what areas of the specification are yet to be defined. There are three main ontologies defined in OWL-S specification [6]. They are called the service profile, the process model and the service grounding.

OWL-S simply provides a mechanism and mark-up language for expressing human knowledge about a Web Service. This knowledge describes non-functional and functional attributes about a Web Service. An organisation or developer that adopts OWL-S to mark-up Web Services is obliged to provide four OWL ontologies to complete an OWL-S service. These ontologies are normally located at separate URI's but can also be defined in the same ontology. Three of the user defined ontologies expand on concepts defined in the core OWL-S specification. These core ontologies are defined in the Sections 2.3.2, 2.3.3 and 2.3.4.

The OWL-S specification will be the basis for designing any beta OWL-S API because it defines the main elements in the OWL-S specification and what the concerns should be. It also blueprints the relationships between them, for example; it states that a Service "presents" a ServiceProfile which indicates that the Service abstraction has a ServiceProfile element as an attribute. Figure 2.3.1 shows a diagrammatic overview of the OWL-S ontology hierarchy and how they inter-relate.

[6] Available to download at http://www.daml.org/service

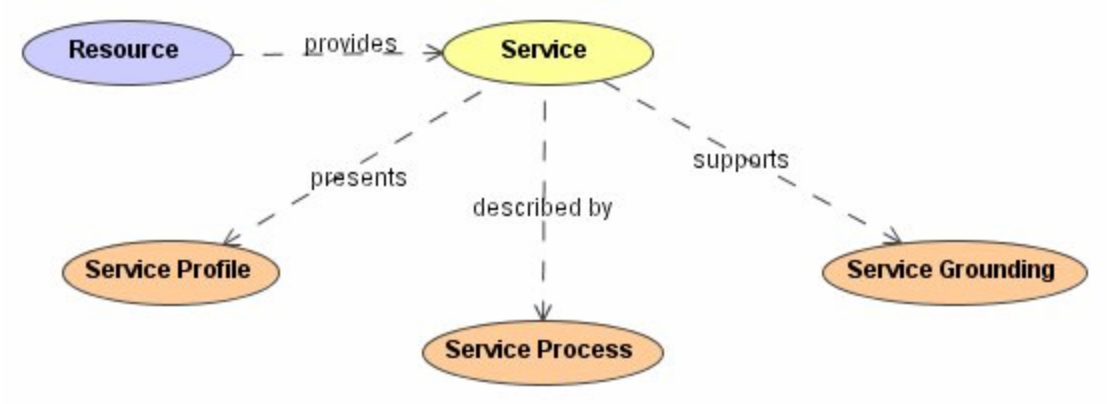

Figure 2.3.1: OWL-S service ontology model relationships.

2.3.2 Service Profile Ontology

The service profile ontology defines the core concepts for advertising and classifying an OWL-S service [6]. An OWL-S service provider typically defines custom profile ontologies, specific to a Web Services domain. The service provider's ontology uses and expands the concepts defined in the DARPA service profile ontology to ensure it type conforms to the specification.

The profile ontology is consulted primarily to establish what type of services a particular OWL-S ontology offers and under what industrial classification it is classed under. The profile ontology answers the question "What does this service offer me?" Essentially, a Web Services functional and non-functional attributes are advertised in the profile model. The service profile model also advertises its functional inputs, outputs, preconditions and effects (IOPE) that an Agent needs to know about.

Inputs are conceptual definitions for parameters that a concrete service needs as input in order to be invoked. Outputs are return data values that are returned from Web Services. Effects are conceptual concepts that relate to real-world things. Outputs values may trigger an effect in the real world; this effect can be conceptualised using OWL constructs. Preconditions have only being sufficiently defined in OWL 1.1 [43] and therefore will not be in scope of this Dissertation. These functional attributes are defined in the process model, a description of which follows in Section 2.3.3. Collectively, they are normally referred to as IOPE.

As well as IOPE, the service profile can also be consulted to look up contact information for human beings or physical locations that have some relevance to the service. A name and also a human readable description of the service are normally provided in the profile model also.

Advertising services in this context of a profile model means, expressing the real world functionality of a Web Service with OWL concepts; or more simplistically, semantic concepts. For example, a profile

model defined in an OWL-S service may contain information that it outputs weather forecasts, or perhaps that it is a currency converter service. The concepts for what weather is or what currency is may exist in some other ontology repository. But, using this embedded knowledge; software Agents can decide whether or not a service is suitable for achieving their goals.

The process of classifying a service in relation to the profile model is, binding a service to a specific third party service category. Again, Agents can determine if the service uses an industrial classification mechanism. Typically, a separate ontology will define the semantics of the Web Services domain, defining what the domain is or industry category[7] it operates in and other useful concepts.

Figure 2.3.2 shows a mark-up segment from an OWL-S Profile Model used to annotate an E-Commerce Web Service. It can be inferred from this mark-up that the service is categorised under the NAICS classification schema as a Business to Consumer (B2C) E-Commerce trader. The service advertises that it is an E-Commerce Service and also that its currency type is Euro. The service also exhibits the real world effects of "Browsing a product inventory" and "Confirmation of Purchase", two concepts associated with B2C E-Commerce. An Agent can examine these concepts and upon further reasoning, it is possible to deduce that this Web Service is a B2C E-Commerce Web Service that supports browsing and purchasing of items.

[7] Perhaps NASIC or UNSPSC Classification Schemes

```
…
<profile:serviceCategory>
      <profile:NAICS rdf:ID="NAICS-category">
            <profile:name>E-Commerce</profile:name>
            <profile:value>Business to Consumer retail sales Internet sites</profile:value>
            <profile:code>454111</profile:code>
            <profile:taxonomy>NAICS</profile:taxonomy>
      </profile:NAICS>
</profile:serviceCategory>

<profile:serviceParameter>
      <concepts:E-CommerceService rdf:ID="E-CommerceService">
      <profile:serviceParameterName>E-
CommerceService</profile:serviceParameterName>
      <concepts:taxonomy
rdf:resource="&concepts;#http://someplace/ont/Wine.xml#Wine"/>
      </concepts:E-CommerceService>
</profile:serviceParameter>

<profile:serviceParameter>
      <concepts:Currency rdf:ID="EStore-currencyType">
      <profile:serviceParameterName>Currency Type</profile:serviceParameterName>
      <concepts:currency rdf:resource="&concepts;#EuroCurrency"/>
      </concepts:Currency>
</profile:serviceParameter>

<profile:hasEffect rdf:resource="&pm_file;#PurchaseItems"/>
<profile:hasEffect rdf:resource="&pm_file;#BrowseInventoryOfProducts"/>
…
```

Figure 2.3.2: segment of mark-up from an example OWL-S Profile Model.

2.3.3 Process Model Ontology

The process model ontology defines the concepts that abstract how a service is intended to be used. Service Providers mark-up custom process model ontologies that give Agents rules and instructions for using the services advertised in the profile model. The process model answers the question "How can I use this service in the correct, prescribed manner?"

The Service Process Model is concerned with how the service can and should be processed, specifically in what order. The class ProcessModel defined by DARPA [6] provides generic constructs and structures that aid in the composition of services. For example, an Agent can understand (by consulting a process model) what inputs a collection of services needs and in what order or sequence a service is meant to be processed to ensure it is operated correctly.

There are three types of processes that can be included in a process model description. These are Atomic processes, which have no sub-processes and operate in a single step [6], Composite processes which are decomposable into sub-processes whose operations are controlled via control structures [6] and finally Simple Processes, which are not invocable, but intended to act as an abstraction or specialisation of some other process.

The process model generally provides semantic descriptions for the input parameters of a service. The process model is entitled to use OWL constructs defined in ontologies to represent web service parameters. These concepts can be as simplistic or complex as desired. Using WSDL, parameters can only be defined using XSD [44] data types. This limits the typing of parameters to such primitive values as Booleans, Strings, Integers and others. By using OWL concepts, data types can be represented in richer, more descriptive ways. Figure 2.3.3 shows WSDL mark-up for a "purchase item" operation, compare this to Figure 2.3.4 which shows the same operation marked up using OWL-S but with the functional parameters represented as concepts rather then XSD data types.

```
…
<wsdl:message name="purchaseItemResponse">
      <wsdl:part name="purchaseItemReturn" type="xsd:boolean"/>
 </wsdl:message>

<wsdl:message name="purchaseItemRequest">
       <wsdl:part name="itemId" type="xsd:int"/>
       <wsdl:part name="amount" type="xsd:int"/>
 </wsdl:message>

<wsdl:operation name="purchaseItem" parameterOrder="itemId amount">
      <wsdl:input message="impl:purchaseItemRequest" name="purchaseItemRequest"/>
      <wsdl:output message="impl:purchaseItemResponse" name="purchaseItemResponse"/>
</wsdl:operation>
 …
```

Figure 2.3.3: WSDL mark-up which only describes functional parameters.

Figure 2.3.3 defines an OWL-S Atomic process, which outputs a semantic type, "PurchaseConfirmation" which is defined in some other ontology and is shown in Figure 2.3.4. The functional attributes of the operation is given some semantic meaning.

```
…
<process:UnConditionalOutput rdf:ID="purchaseItem_Output">
      <process:parameterType rdf:resource="&my_concepts;#PurchaseConfirmation" />
</process:UnConditionalOutput>

<process:UnConditionalEffect rdf:ID="ShippingID_Effect">
      <process:ceEffect rdf:resource="&my_concepts;#ShippingID"/>
</process:UnConditionalEffect>

<process:Input rdf:ID="purchaseItem_ammount_IN">
      <process:parameterType rdf:resource="&my_concepts;#ItemAmount" />
</process:Input>

<process:Input rdf:ID="purchaseItem_itemId_IN">
      <process:parameterType rdf:resource="&product;#UniqueId" />
</process:Input>

<process:AtomicProcess rdf:ID="Operation_purchaseItem">
      <process:hasInput rdf:resource="# purchaseItem_ammount_IN"/>
      <process:hasInput rdf:resource="# purchaseItem_itemId_IN"/>
      <process:hasOutput rdf:resource="# purchaseItem _Output"/>
      <process:hasEffect rdf:resource="# ShippingID_Effect "/>
</process:AtomicProcess>
…
```

Figure 2.3.4: Atomic Process definition containing an output and effect.

Figure 2.3.5 details the OWL class for PurchaseConfirmation. It also defines a data type property associated with the class and constrained to the XSD type Boolean.

```
…
<owl:Class rdf:ID="PurchaseConfirmation"/>

 <owl:DatatypeProperty rdf:ID="hasConfirmation">
       <rdfs:range rdf:resource="&xsd;boolean"/>
       <rdfs:domain rdf:resource="#PurchaseConfirmation"/>
       <rdfs:comment>has true if confirmation exists</rdfs:comment>
  </owl:DatatypeProperty>
…
```

Figure 2.3.5: definition of a Purchase Confirmation concept.

An Agent can evaluate and reason over what a “PurchaseConfirmation” concept is, by using knowledge that exists in its run-time environment coupled with the OWL concept defined in some ontology. This allows an Agent to infer that this service is perhaps appropriate to meet its objectives.

A good metaphor for understanding the motivation for the process model is to think of it as well structured instructions for some arbitrary item, maybe a car owner's manual. The "item" represents the web service. The "instructions" are analogous to the rules for operating a process, and are "well structured", because there is almost always a finite way to use a certain item.

2.3.4 Service Grounding Ontology

The class ServiceGrounding defines concepts that can be used to map an OWL-S abstraction of a service to a concrete realisation of a service. The OWL-S specification defines a grounding ontology as one which specifies the details of how to access a service [6]. The details include what message formats to use and what protocols are expected. A grounding ontology can answer the question "Where and in what way (protocol) do I communicate with this service". The class ServiceGrounding allows OWL-S to remain technology independent and concentrate on generic concepts rather then concrete guidelines. Figure 2.3.6 shows the grounding mark-up for the Atomic Process "purchaseItem" defined in Figure 2.3.4.

```
…
<grounding:WsdlAtomicProcessGrounding rdf:ID="WSDL_purchaseItem">
      <grounding:owlsProcess rdf:resource="&pm_file;#purchaseItem"/>
      <grounding:wsdlDocument>&wsdl;</grounding:wsdlDocument>
      <grounding:wsdlOperation>
            <grounding:WsdlOperationRef>
                  <grounding:portType>&wsdl;#EStore</grounding:portType>
                   <grounding:operation>&wsdl; #purchaseItem</grounding:operation>
            </grounding:WsdlOperationRef>
      </grounding:wsdlOperation>
  <grounding:wsdlInputMessage>&wsdl;#purchaseItemRequest</grounding:wsdlInputMessage>
<grounding:wsdlInputMessageParts rdf:parseType="Collection">
      <grounding:WsdlMessageMap>
           <grounding:owlsParameter rdf:resource="&pm_file;#purchaseItem_ammount_IN"/>
           <grounding:wsdlMessagePart>&wsdl;#ammount</grounding:wsdlMessagePart>
      </grounding:WsdlMessageMap>

      <grounding:WsdlMessageMap>
            <grounding:owlsParameter rdf:resource="&pm_file;#purchaseItem_itemId_IN"/>
            <grounding:wsdlMessagePart>&wsdl;#itemId</grounding:wsdlMessagePart>
      </grounding:WsdlMessageMap>
</grounding:wsdlInputMessageParts>

<grounding:wsdlOutputMessage>&wsdl;#purchaseItemResponse</grounding:wsdlOutputMessage>
      <grounding:wsdlOutputMessageParts rdf:parseType="Collection">
            <grounding:WsdlMessageMap>
                  <grounding:owlsParameter rdf:resource="&pm_file;#purchaseItem _OUT"/>
                  <grounding:wsdlMessagePart>&wsdl;#purchaseItemReturn
                  </grounding:wsdlMessagePart>
            </grounding:WsdlMessageMap>
       </grounding:wsdlOutputMessageParts>

       <grounding:wsdlReference>http://www.w3.org/TR/2001/NOTE-wsdl-20010315
       </grounding:wsdlReference>
</grounding:WsdlAtomicProcessGrounding>
…
```

Figure 2.3.6: grounding mark-up for the Atomic Process “Purchase Item”.

2.3.5 OWL-S complementing WSDL standards

Another key strength of OWL-S is that it can utilise industry standards, namely WSDL [6]. This is an important link to allow OWL-S conform to existing industry standards and technologies, while allowing legacy systems to be integrated. Other XML based standards could also be incorporated due to the fact that the Service Grounding class is not constrained by any particular technology.

2.3.6 Service Description Ontology

The final ontology necessary to compose an OWL-S service is a Service Description. The service description is essentially an entry point into a specific OWL-S ontologies collection. A Service Description can have one or many services defined, that individually provide URI locations for where the profile, process and grounding models exist for the service.

Chapter 3

STATE OF THE ART

3 State of the Art

Currently the most active areas of research in the Semantic Web Service domain[8] are service invocation, planning and response interpretation using process descriptions, protocol interpretation and execution, semantic translation and mediation between processes, candidate service identification and selection, automated process composition, process status tracking, failure semantics, ontology creation, management and access; trusted reputation services, inter-service negotiation and possibly the most important ambiguity, security, including identification, authentication, and policy based authorization [15].

The state of the art is focused on reviewing research efforts underway elsewhere which partially address our research objectives defined in Section 1.4, but not all of them in entirety. The format of each sub-section is as follows; firstly we will provide an overview of each research efforts aims. We will conclude each sub-section by evaluating what the merits of the research are and what objectives of our research it does not address.

3.1 Semantic Service Discovery

3.1.1 E-Speak

The E-speak model was one of the first service architectures proposed. It was developed by Hewlett-Packard [18]. E-speak and UDDI both have complimentary goals, as they both allow the advertisement and discovery of services. E-speck acts as a portal for discovering E-speak services using keywords searches. E-speak concepts are not as inter-operable as UDDI, because it needs an E-speak engine to be running on all client machines. Moreover the E-speak model does not facilitate execution monitoring in its current state. There is currently no support for Semantic Matching of keywords in E-speak [18].

3.1.2 DAML-S Matchmaker

The DAML-S Matchmaker is a semantically enhanced UDDI registry [22]. It extends the UDDI specification to facilitate its goals [9]. The matchmaker allows service providers to publish OWL-S ontologies semantically into the registry by using the OWL-S ontologies Profile Model to express its

[8] Listed in no particular order

capabilities [9]. The matchmaker aims to improve the service discovery process by allowing location of services based on their capabilities [10]. Inference is used to select UDDI entries that have a subsumption relationship to a certain request parameter.

3.1.3 UDDI Enhancement

Akkiraju et al. [48] [49] have also proposed models to enhance UDDI to support capability searches. This work extends the findings proposed by Paolucci et al [9]. They propose an extension to the UDDI inquiry API specification to enable requesters to specify the required capabilities of a service. Secondly, the service discovery capabilities of UDDI are enhanced by performing semantic matching and automatic service composition using planning algorithms [48]. Finally, it is proposed that these service compositions should be presented in BPEL4WS, described in Section 3.3.1.

3.1.4 Evaluation of Semantic Service Discovery Research

E-Speak currently does not facilitate capability based queries and thus, fails to enable our research objectives for semantic discovery. E-Speak also involves installing proprietary software in order for an E-Speak engine to be used effectively; We regard this as a limitation for examining semantic discovery mechanisms.

The DAML-S Matchmaker and Akkiraju et al. have proposed solutions to enable semantic discovery through the extension of the UDDI specification. Both efforts do provide a mechanism to semantically discover Web Services and are based on UDDI, a technology we decided to use. Because of this research's maturity, we wished to follow an alternative branch of research that did not seek to extend UDDI, thus maintaining the open ethos of the Semantic Web.

3.2 Candidate Selection

3.2.1 DAML Dining

DAML dining [23] is a semantically enabled web tool that allows users to search a DAML ontology repository. This repository contains ontologies that describe the characteristics of restaurants. This tool is analogous to a semantic search portal. It is of interest to this research, as it allows users to interact and select attributes for search criteria that are compared to information that exists in OWL/DAML ontologies.

This application is also in the domain of E-Commerce, but does not offer any reservation service, or sell any tangible goods. It is obvious from this scenario how OWL-S ontologies could benefit from such an application by allowing Web Service technologies to be incorporated in the dining application. This could extend the application to facilitate on-line bookings and perhaps take-away services. There

are no details on the implementation of this tool, or its accuracy or success. It is intended to be seen as a showcase technology demo.

3.2.2 Evaluation of Candidate Selection

DAML Dining is not service orientated; this research however does address the area of candidate selection using knowledge encoded in ontologies. Although the research is promising, it is not related to the Web Service domain and thus fails to fully satisfy our research objectives described in Section 1.4.

3.3 Semantic Service Composition

3.3.1 BPEL4WS

The Business Process Execution Language for Web Services (BPEL4WS) [16] specification was co-authored by IBM, Microsoft and BEA. It supersedes work done by Microsoft on XLANG and also IBM's Web Service Flow Language (WSFL). It provides a notation for describing the interactions of Web Services, following in the tradition of workflow modelling [17]. Workflow in BPEL4WS is directed by traditional control structures like if, then, else; and while-loop. Services are integrated by treating them as partners that fill roles in a BPEL4WS process model. WSDL is used to notate the functional attributes of a web service. A process model, in BPEL4WS describes a workflow that orchestrates the interaction of the service partners. It attempts to choreograph the planning and protocol interpretations in the Web Services domain.

BPEL4WS focuses on representing compositions where all of the process and the bindings between services are known a priori. A more challenging problem is to compose services dynamically, on demand [6]. OWL-S intent is to use ontologies to provide Agents with the semantic knowledge necessary to dynamically discover and evaluate Web Services, BPEL4WS is more aligned to deployment time composition. However it is clear that BPEL4WS and OWL-S have broad and similar objectives [10].

Essentially BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration both in the intra-corporate and the business-to-business spaces. [16]

3.3.2 Semi-Automatic Service Composer Tool

Siren et al. have developed a semi-automatic web service composer [10]. This tool allows a user to control the selection of OWL-S annotated services and provides WSDL invocation capabilities. The services used are pre-configured and it is not the goal of the tool to discover services. The composer is

capable of reading DAML-S descriptions and presents the service attributes to a user as a list. This allows the user to filter this list using the ontological information encoded in service descriptions, effectively to select a candidate service. The user can create a composition by choosing a service to supply the input of another service [10].

3.3.3 Pizza and a Movie Selection and Composition

One future use case for OWL-S is to help enrich applications in the E-Commerce domain [10] [46]. Dale et al. proposes an evening organisation service, whose design is presented together with its reasoning flow [20]. This service is actually a collection of smaller programs, which are Web accessible and composed together to form the evening organizer. [20]. DAML-S is used to annotate how this composition should work.

The motivation for this example was to demonstrate how a user (customer) could benefit from having an intelligent Agent compose and invoke Web Services on the customer's behalf that meet the customer functional requirements. It presents the potential of advanced Web Services through the case study [20]. In this case, the user wants to find a pizza restaurant in the San Francisco area and also a showing of a Sci-Fi movie for afterwards, including the relevant transport to and from the locations.

The scenario starts by asking the customer to fill out a template form which guides the user to a Restaurant and Shows template. The answers supplied to these questions are used to formulate a user profile; which the Agent will analyze when deciding which service to select for invocation. An event organiser component then returns a completed itinerary to the Agent [20]. The Agent, in turn will present the itinerary back to the user, asking for a final confirmation to proceed. The services theoretically, will be invoked depending on the users signal and the outcome of this process is once again presented to the user. If any service fails to complete the user can be asked for further input or perhaps the reasoning engine can suggest an alternative. This process will continue until the user is satisfied with the Agents work.

3.3.4 Evaluation of Service Composition

BPEL4WS is an elegant solution to Web Service composition, but fails to address our research objective of achieving composition at run-time.

Siren et al. research supports composition and invocation of Web Services using OWL-S but is not intended to facilitate discovery, and as such does not answer our research objectives in entirety. The research identifies the need for Agents to be aware of what is needed as input into services and to

dynamically compose several services together to achieve a goal. It uses simplistic planning techniques to formalise a composition.

The Pizza and a Movie research does not incorporate semantic discovery methods for Web Services, and as such does not answer our research objectives in entirety. However, the research is a good E-Commerce composition scenario. It focuses purely on enabling composition using semantic concepts defined in Process Model instances. This composition is partly achieved by employing static composition methods, but using OWL-S to describe a composition. It does not use profile model attributes to aid service selection. This means that non-functional service attributes are not used in the composition process, as these are intended to be defined in the a Profile Model instance. It provides no details on how candidate identification and selection is achieved.

3.4 Semantic Service Invocation

3.4.1 OWL-S API

Currently, the most comprehensive OWL-S API has been released by Mindswap[9]. This API provides functionality to enable the invocation of OWL-S annotated Web Services using the SOAP protocol. The API can be used to parse an OWL-S ontology into memory and provides interfaces to access the OWL-S data structures defined in the OWL-S specification.

3.4.2 Web Service Description Framework

Eberhart proposes using the Web Service Description Framework (WSDF) to provide both a representation mechanism and run-time system architecture for semantically enriched Web Services [63]. The approach allows a client to invoke a service based solely on a shared ontology, i.e. without prior knowledge on a specific API. WSDF requires client and server to provide a mapping from the local structures to some common domain ontology at design-time. This research states that OWL-S does not currently address such issues as relationships between parameters, result processing and shared data-type mediation [63].

3.4.3 Evaluation of Service Invocation

The OWL-S API does not enable automated invocation, as a programmer must write some client code to load parameters and reconcile data-types.

While some of Eberhart's critique has merit, OWL-S is still an emerging standard and deficiencies with the invocation processes will likely be resolved in later OWL-S versions, by introducing a technique to

[9] http://www.mindswap.org/ - Mindswap Home Page

express a more elaborate representation of knowledge, perhaps using an OWL based rule language. Eberhart's research does not incorporate OWL-S and will not be evaluated further in this research.

3.5 Conclusion

In this chapter, we have profiled a selection of research efforts that in its entirety, encompasses our research objectives. We have shown that for every research effort profiled, no single one completely addresses all of our research objectives outlined in Section 1.4 completely. We believe however, that the research contained in this dissertation has merit in aiding the semantic discovery, candidate selection, semantic composition and invocation of Web Services domain.

Chapter 4

DESIGN

4 Design

To facilitate the research objectives defined in Section 1.4 a framework providing semantic discovery, candidate selection, service composition and invocation is proposed. Our framework can be used as the building block for semantically enabled applications. Our framework's capabilities are described in Section 4.1. An E-Commerce use case is proposed in Section 4.2 which demonstrates the need for this kind of semantic framework.

4.1 Framework Overview

Figure 4.1.1 shows a component overview of the semantic framework. The main components are a semantic discovery service, candidate selection service, composition service and an invocation service.

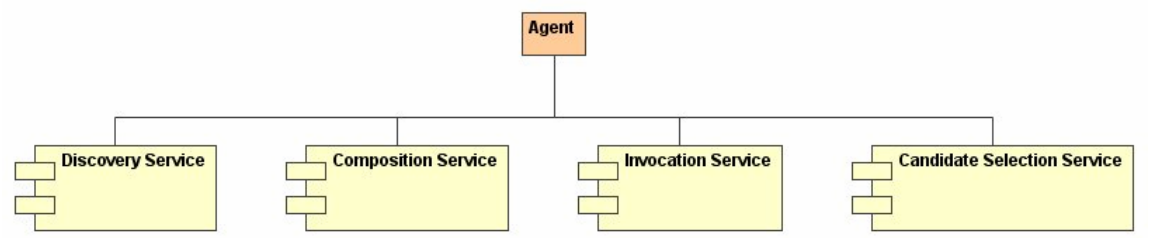

Figure 4.1.1: component overview of the semantic framework.

4.1.1 UDDI Integration for Semantic Discovery

Current Web Services technology based on UDDI and WSDL does not make any use of semantic information and therefore fails to address the problem of matching between capabilities of services and allowing service location on the basis of what functionalities are sought, failing therefore to address the problem of locating web services [9]. Using descriptive keywords and business names are the only means of currently searching a UDDI registry. To facilitate a capability search on a UDDI registry, the semantics of an OWL-S service need to be mapped to a UDDI registry.

The solution used in this framework was to register the Access Point[10] of a UDDI Service entry to map to the URI of the OWL-S Service ontology. Identifying names for OWL concepts were extracted from

[10] Abstract data type used to provide a method for a client to contact to a Business Service.

the Service Profile model and used as keyword descriptions for the Service. For example, if an OWL-S Service Profile ontology is classified under an "E-Commerce" ontological concept, "E-Commerce" will be extracted and used as a descriptive keyword.

Using a pre-defined keyword search, businesses entries will be returned that have matching descriptive keywords. For each service returned, the Access Point URI for the service is validated to check whether or not it is an OWL-S Ontology. If it is an ontology, then its capabilities are examined to check whether it conforms to the Agents requirements.

4.1.2 Candidate Selection using Semantic Reasoning

To provide service identification based on capabilities, a reasoner component was developed to facilitate semantic inference of OWL concepts. The reasoner is used in this context to match a set of non-functional or functional user requirements against the discovered OWL-S services. The output of the reasoning process is a candidate priority queue of OWL-S services. The service which registers the most semantic matches against the user's requirements is returned first. This process allows an Agent to identify Web Services and processes that best match the user's goal.

4.1.3 Service Composition using Capability Reasoning

To achieve service composition an Agent needs to be able to dynamically assemble an execution plan capable of achieving a user's goal. This kind of assembly and orchestration of services is commonly known as choreographing services. To achieve this functionality, an Agent must identify services whose capabilities match the user's goals to some degree. This can be achieved at run-time by using the candidate service selection functionality described in Section 4.1.2. Effectively, an Agent needs to assemble a collection of services that are capable of fulfilling user requirements.

Service Composition functionality is reliant also on automated invocation of services described in Section 4.1.4. An Agent cannot compose a set of services if it does not have the required information and logic to use them. By examining the IOPE's of the composed service prior to execution, it is possible to predict whether an Agent can supply the required data to successfully invoke the services. It is possible to identify this fact by consulting an OWL-S Process Model. Outputs from one service may be suitable inputs to another service. This kind of deduction makes it possible to plan an order for service invocation.

The framework facilitates a simplistic model of composition. No consideration for service monitoring, transactional characteristics or security has been factored into this solution.

4.1.4 Semantic Service Invocation

The framework automates service invocation by consulting the services OWL-S Process Model. Using the Process Model, Web Service functionality can be dynamically invoked by matching a user's goal to the IOPE as expressed in the Profile Model. For invocation, the OWL-S process that best provides the user's goal is selected and invoked. Input parameter types are dynamically reconciled, based on converting user input values and attributes to semantic concepts. These semantic concepts are compared to the selected processes inputs to try and find a suitable candidate. If all parameters can be set, the OWL-S Grounding is used to invoke a SOAP message on the Web Service.

If the Web Service returns an output the Agent will interpret the appropriate response. This may involve further user input, or perhaps further invocation.

4.2 Application Description and Use Cases

To demonstrate how the framework can be applied, an E-Commerce application was built. This application demonstrates the benefits that semantic applications could realise for Internet users. The application domain was an E-Commerce Portal B2C Website.

The portal scenario revolves around a web site that enables users locate, search and purchase fictional products from different virtual E-Stores. An assumption made by this example is that vendors have subscribed to the portals notion of the world, and are a satisfied with their semantic representation. It is possible to link in other ontologies, but some semantic bridge would need to be defined between the new and old ontologies to render them compatible.

The portal was built using the functionality provided by the framework components described in Section 4.1. Fictional E-Commerce Services were grounded into a UDDI registry and the discovery component was used to dynamically discover these entries. The candidate selection component was used to identify suitable candidate services. It was also used as a generic OWL reasoner for inference. Finally, the invocation service provided functionality to allow automated invocation of the Portals Web Services.

As stated in Section 1.4, non-functional service attributes were used to identify candidate services. Section 4.2.1 identifies these non-functional service attributes.

4.2.1 Non-Functional Service Attributes

The importance of any non-functional attribute will depend on what the user deems as important for their means. Some users may perceive service availability to be more important to them then the

geographical location of a service. The concepts ontology explained in Section 4.2.6 defines the non-functional service concepts that Portal users can choose from.

4.2.1.1 Quality of Service

Service Reputation is a research area that is beyond the scope of this Dissertation. It is proposed that in the future, Service Providers will be able to rate their service, via trusted reputation monitors [19]. This scenario can be simplified by defining a "ServiceQuality" concept and assigning a rating to each service. If a service exhibits "High Quality", it can be inferred that the service is of a reputable nature.

4.2.1.2 Reliability

If a service is down or inaccessible frequently, it is an unreliable service. A service that is available 24/7 is a desirable service to use. To model reliability, an ontological concept can be defined to model reliable services and unreliable services. For example, a reliable service has a "Constant" availability, where as an unreliable service has an "Infrequent" availability. As with Service Quality, monitoring the reliability of services by third parties is out of scope of this research.

4.2.1.3 Geographical Constraints

A user may prefer to access services where the brick depots are located in a specific geographical area. For example a Cork based customer may have source a service provided by an E-Shop in Belfast but wants instead to source the service in the Republic of Ireland and therefore deal in Euro. This ability can be incorporated into the geo-coding requirement. Location concepts provide a method for adding location aware information into OWL-S Service descriptions.

4.2.1.4 Service Classification

Industrial classification services have been formulated to allow companies to classify their service in relation to others. NASIC and UUNISPC are two such classification services. An OWL-S service can be classified in some domain to help express its capabilities. For example, if a service is classified under some E-Commerce B2C classification, it is a plausible assumption that the service participates in some sort of consumer orientated E-Commerce.

4.2.1.5 Currency Classification

A particular service may want to constrain the types of currency it deals in. For example, a British based company may offer its products in Sterling and Dollar prices. A user may only have access to Euro Currency; this means that the British service is not suitable for the user.

4.2.1.6 Temporal Classification

If a service is not performing satisfactorily or not as "advertised" then this will impact negatively on the probability that a client will use the service. If a service is perceived to be slow, this will make it less desirable than a quick and effective service. A client should be interested in services that are timely and responsive. The compulsion to advertise truthfully on this matter is beyond the scope of this research.

4.2.1.7 Geo-coding

Geo-coding is the process of taking an address that exists in the "real world" and adding longitude and latitude coordinates to the data. It would be interesting to factor this data into OWL-S Profile Model instances, so as location aware decisions can be made by the composition framework. This Dissertation has not factored in Geo-coding data.

4.2.1.8 Pre-conditions

Pre-conditions are evaluations that must hold prior to a service being invoked. A Pre-condition can involve evaluating an expression, which may even involve invoking a process. A Pre-condition is used to ensure that a client is capable of using a service in its intended way.

For example, a minimum of 12 bottles of wine must be ordered to avail of the delivery service; in this case the client must give direction to the Agent on how to handle such situations.

4.2.1.9 Post-conditions

Post-conditions are concerned with whether or not the overall result of the service meets the expectations of the user was expecting (based on input parameters). Version 1.1 of the OWL-S specification details more comprehensive rules regarding Post-conditions. Post-conditions have not been examined further in this research due to the limitations of the OWL-S 1.0 specification.

4.2.2 Domain Description

…Internet user wants to buy a product from an online store…

The chosen use-case for this Dissertation is a portal web site for selling fictitious products. The chosen products groups are Wine, Cheese and Cameras. Ontologies representing the concepts were already defined and incorporated in to the research to demonstrate the flexibility of the framework.

These products are also very diverse with their own individual semantics and character. This implies that one variety of cheese may not complement every subset of cheese, and obviously camera specifications will vary resulting in different classes of quality and reliability. Wine and Cameras are

sufficiently different in their semantics to prove that the framework can be used for diverse use-cases, concepts and ontologies.

4.2.3 Domain Issues

Currently, a common model for E-Commerce B2C shopping involves a user manually searching through on-line catalogues of products until a satisfactory product is located. The user then has to submit some credentials to electronically pay for the product. This kind of model is used by such E-Commerce companies as Buy 4 Now[11], E-Bay[12] and Amazon[13]. It is noted that E-Bay's payment model is an auction based consumer to consumer model, but the discovery of products revolves around keyword searches and browsing product categories.

The process described above involves human intelligence to realise the objective of purchasing a desirable E-Commerce product. A human user is obliged to control the discovery, selection and instigating of purchase operations for products.

Using semantic technologies, we have shown it is possible to automate this entire process. Agent technology and Web Services will enable E-Commerce services participate in global scale portal networks. Of course, even with semantics, this kind of automated transaction is not currently possible due to the limitations that exist in current security, trust and reputation authorities [19].

4.2.4 Actors & Goals

- Customer (Actor)
- Identify Candidate Products (Goal)
- Discover Services (Goal)
- Purchase Product (Goal)
- Select Products (Goal)

[11] http://www.buy4now.ie/

[12] http://www.ebay.com/

[13] http://www.amazon.com/

4.2.5 Use Case UML

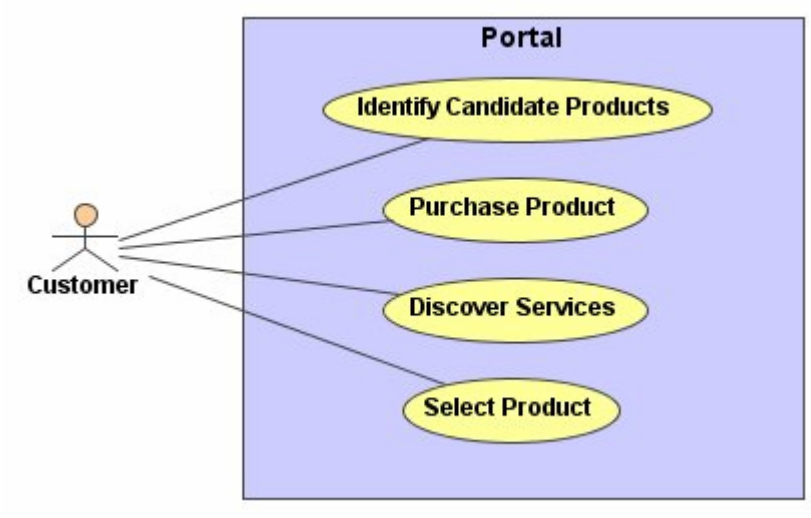

Figure 4.2.1: Use Case UML definition.

4.2.6 Ontologies and Semantic Descriptions

There are some domain specific ontologies defined for use by the application that describe the E-Commerce world. These ontologies are used to encode knowledge about the application domain. In the following sub-sections, we will discuss the core ontologies used to describe the Portals semantics.

Concepts: The concepts ontology among other things, defines meta-data about the characteristics of Web Services that have subscribed to the Portal. The concepts ontology is used by the portals OWL-S instances to advertise their respective services capabilities. General domain knowledge such as Money, what it is to browse products and the concepts of purchasing a product are also defined.

The concepts ontology defines OWL classes that inherit the OWL-S Profile Model concept, ServiceParameter[14]. This allows reasoners to infer what Service Attributes a Profile Model advertises. This enables an Agent to match a Profile Model to non-functional user goals.

The concepts ontology is by no means a comprehensive domain description, but provides enough realism for use by this demo application. For example, “Confirmation” is an ontological concept defined in the concepts ontology. Using inference techniques and consultation of the semantic concept, purchase confirmation can be understood to be the result of buying an item from an E-Commerce vendor. Figure 4.2.2 shows a UML class diagram which shows the concepts ontologies semantic relationships.

[14] http://www.daml.org/services/owl-s/1.0/Profile.owl#ServiceParameter – definition of OWL concept

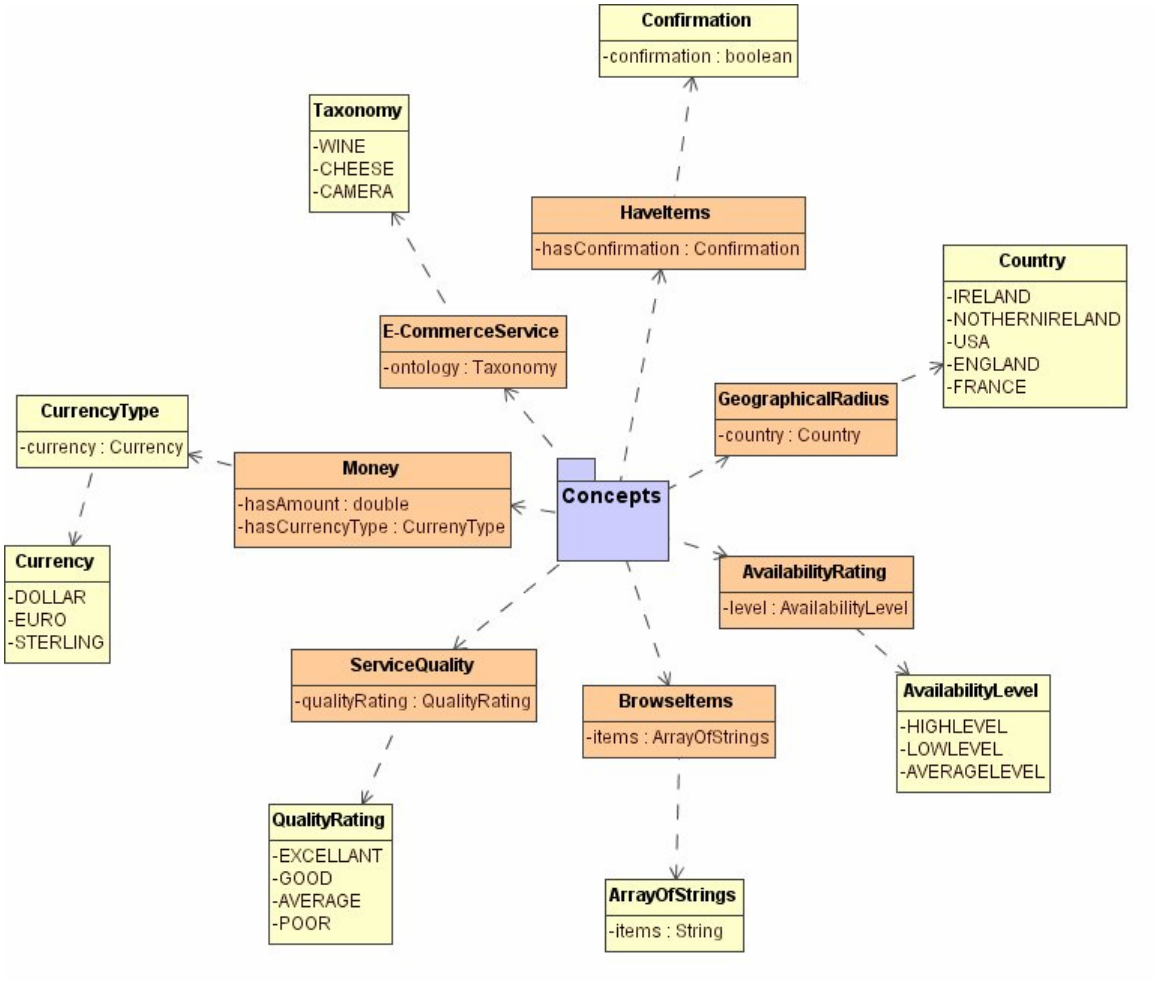

Figure 2.2.2: concept ontologies relationships express in UML notation.

Customer: The customer ontology represents the main characteristic of a human customer. It defines properties such as a customer balance and a customer's username. The customer's ontology is the basis for the in-memory representation of the frameworks user. This user profile provides the Agent with state. The instantiation of a customer is referred to as the user profile in this Dissertation from now on.

Portal: The portal ontology abstracts the concepts of an E-Commerce portal and groups together non-functional service properties for use in candidate service identification.

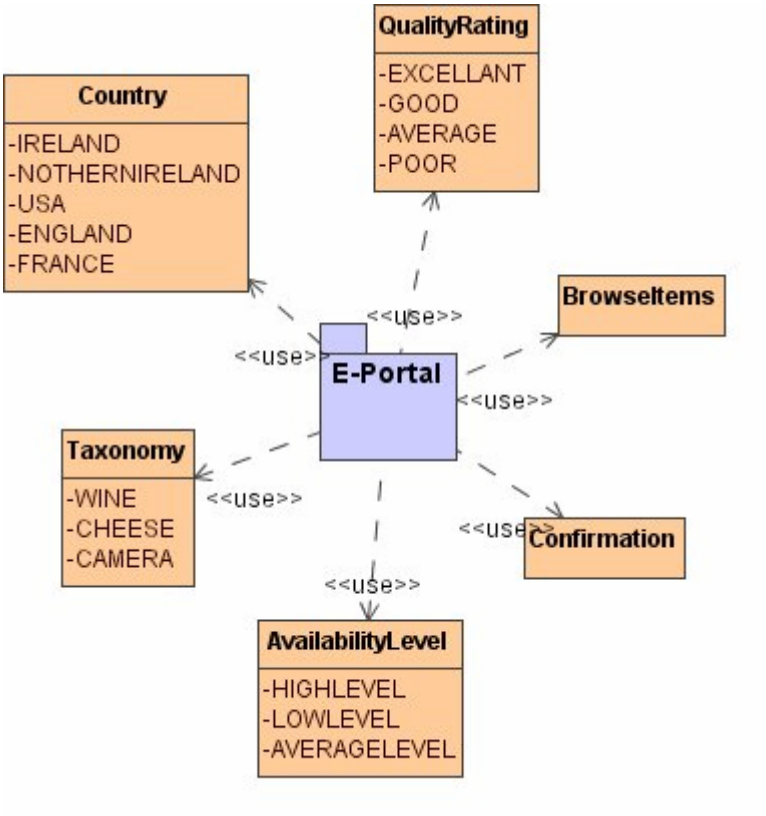

Figure 4.2.3: UML representation of the Portal Ontology.

The OWL <owl: AllDifferent> construct allows non-functional service attributes to have several possible values. This mechanism de-couples the definition of the concepts from the concepts usage. The AllDifferent mark-up can be seen in Figure 4.2.4.

```
…
<owl:AllDifferent>
   <owl:distinctMembers rdf:parseType="Collection">
      <concepts:QualityRating rdf:about="Excellent"/>
      <concepts:QualityRating rdf:about="Good"/>
      <concepts:QualityRating rdf:about="Average"/>
      <concepts:QualityRating rdf:about="Poor"/>
   </owl:distinctMembers>
</owl:AllDifferent>
…
```

Figure 4.2.4: OWL definition to denote different Quality Rating concepts.

Products: There are three product ontologies that represent the E-Portals product range; these are a Wine, a Cheese and Camera ontology. All three ontologies contain semantic knowledge about the make up of the individual products and also define individual instances of the products. For example, the

wine ontology defines a class Wine, with properties such as hasColour, hasBody, hasTaste and other wine characteristics. The ontology also defines individuals of wines such as Pinot Noir and Cotturi Zinfandel. Ontology management is not in the scope of this framework. This means that the product instances defined in the products ontologies represent all the possible product types that this portal is aware of. Section 7.3.1 discusses some possible improvements to this scenario by introducing ontology management techniques.

4.2.7 Scenario Steps

1. A user identifies themself by selecting a user profile. Once a profile is selected, the portal initialises the user's details by consulting a database and the customer ontology.

2. A user selects non-functional service attributes from a finite set. A user can assign a priority to their selection – High Priority, Medium Priority or Low Priority.

3. After submission, the selected attributes are added to the in-memory representation of the user's profile.

4. A UDDI keyword query is executed in order to find a selection of appropriate Web Services. The access points for the services are validated to ensure they are valid OWL-S service ontologies. Each ontology returned is examined to see whether or not they have the capability to fulfil the portals use cases that is to browse an inventory of products and purchase an item.

5. The identified ontologies are reasoned over to assemble a list of candidate services that best match the users' preferences.

6. To proceed, the user must select a candidate service recommended by the discovery process. Once a service is selected, the user is asked to select functional properties that relate to the product that the e-store sells. For example, a wine service will ask a user to select the functional properties of wine, such as colour and smell. A priority can be assigned to each property also, as in step 1. To extend this scenario, several services could be selected to demonstrate service composition.

7. The "find all products operation" is invoked on the chosen Web Service. This returns an array of product instances stored in the e-store's database. These product instances are matched to their equivalent product defined in the ontology and merged with there ontological properties.

The inference engine is invoked again to find a candidate group of products that match the user's preferences.

8. If a user decides to purchase a product recommended by the portal, then the purchase operation is invoked and the scenario ends. A Boolean value of true represents a successful purchase transaction.

4.2.8 Use Case Deployment Architecture

A high level architecture, showing the main application components is presented in Figure 4.2.5. It shows the Portal abstraction which contains components that are capable of Discovery, Candidate Selection, Composition and Invocation. The Agent also operates inside the Portal deployment.

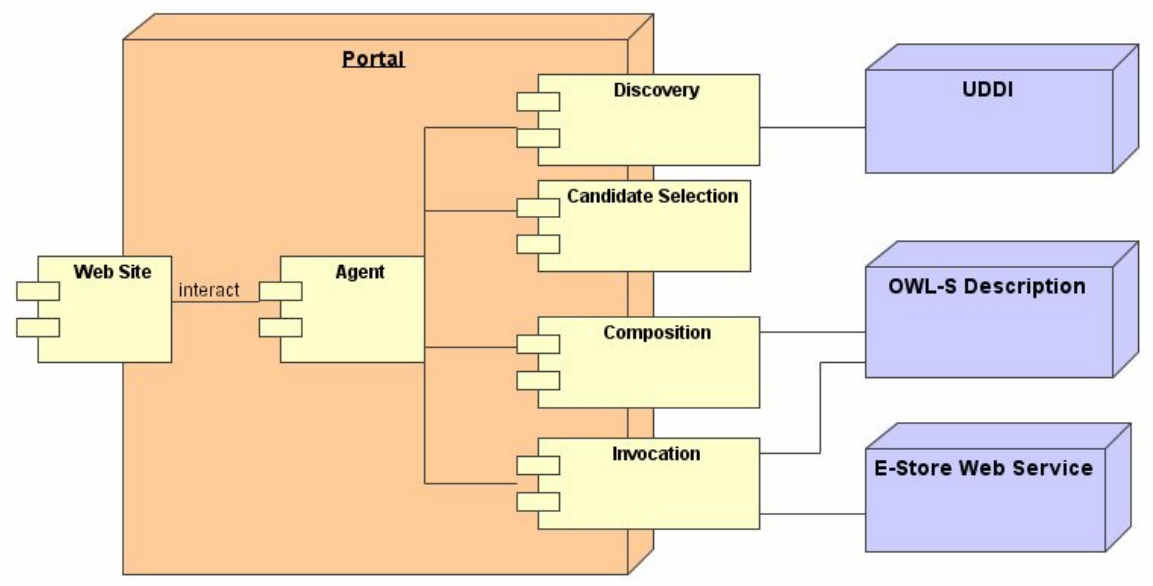

Figure 4.2.5: high level component architecture for Portal.

Chapter 5

FRAMEWORK IMPLEMENTATION

5 Framework Implementation

This chapter presents a technical description of the frameworks implementation. A high level overview of the framework is presented in Section 5.1, followed by a detailed description of the discovery, candidate selection, service composition and invocation functionality in Sections 5.4, 5.5, 5.6 and 5.7 respectively.

5.1 Overview

The framework can ultimately be used in the context of a user controlled Agent or an automated Agent controlled by some other application, perhaps a Rule Based planner engine. For this Dissertation's demo application, the framework is instantiated and executed inside an Application Server and controlled by a user via a JSP [29] Web Application.

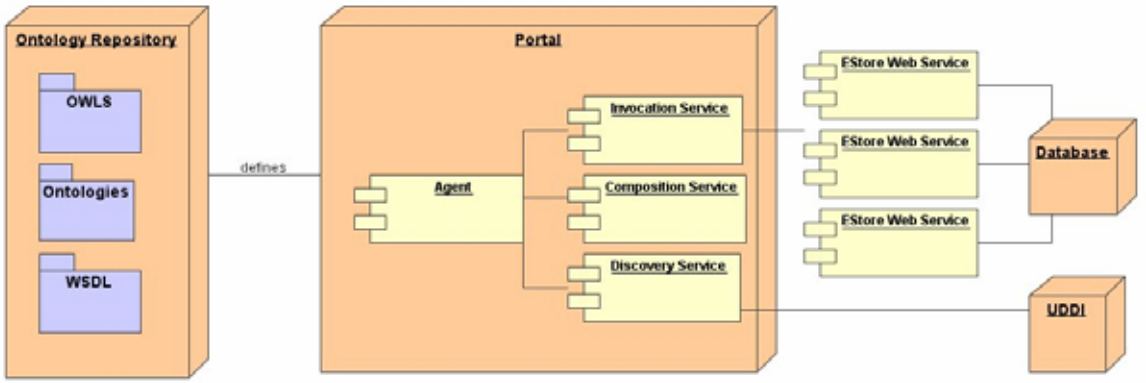

Figure 5.1.1: main deployments involved in the demo application.

Figure 5.1.1 shows deployment architecture for the semantic framework. The deployment consists of an Ontology Repository, storing the OWL-S ontologies, domain concepts and the WSDL for the E-Commerce services. This is logically grouped together, but in reality could be located across multiple servers. The E-Commerce Portal is hosted on one server. The Agent and semantic framework is deployed on this server. The E-Store Web Services and UDDI registry can reside in some other server(s).

5.2 Assisting Technologies

This Section details some of the third party software used or extended for this implementation.

JDOM: JDOM [56] is an XML parser for Java. The JDOM parser parses an XML file into the Document Object Model, which is a tree-like representation of the parsed XML document. From this DOM the XML document can be evaluated and traversed by browsing its tree structure.

JDOM was used to parse and manipulate XML documents in memory. It's functionality was not extended or modified.

Jena: Jena [to do] is a Java framework for enabling Semantic Web applications. Jena is open source software and was developed at the HP Semantic Labs. It provides a programmatic environment for RDF and OWL.

Jena converts RDF documents into a memory resident RDF model using subject-verb-object triples. It supports information retrieval through an iteration mechanism and the RDQL [57] query language. It also provides interfaces to allow semantic relationships be inferred between types.

Jena was used in this framework as an underlying Ontology API. Jena functionality enabled the initialisation of ontologies into memory, and provided software interfaces that allowed for semantic relationships to be tested and inferred between OWL semantic concepts.

OWL-S: OWL-S is an API providing OWL-S functionality for version 1.0. The underlying API provides implementation for much of the data types used in OWL-S, including the core ontology abstractions, Profile, Process and Grounding models.

The API was sourced from the Mindswap[15] organisation. The API lacked the functionality to parse and extract OWL-S Effects and Preconditions from process models. It also did not implement data structures for Profile Model concepts such as Actors, Service Parameters and Service Categories. The API did not include any implementation for the `ChoiceOf` OWL-S Process execution.

The `ChoiceOf` construct allows for Atomic or Composite processes to be grouped together and invoked separately. `ChoiceOf` facilitates an Agent choosing a process based on its IOPE's. The `ChoiceOf` construct is one method to allow Agents to assemble a sequence of Processes from different `ChoiceOf` constructs in different OWL-S ontologies. `ChoiceOf` is used in the E-Store

[15] http://www.mindswap.org/ - Mindswap Home Page

application to allow Web Services offer a choice of services. In the sample scenario described in Section 4.2, the choice is between browsing a product inventory and purchasing an item.

To provide these features, the OWL-S API was extended. The execution engine component used for invoking OWL-S services needed an operation to support “Choice” Processes. The functionality allowed for a client to choose a process, based on what IOPE was desired. The existing execution engine was extended to support `ChoiceOf` invocations. This process will be largely un-documented as is not a core part of the framework. It is also recognised that subsequent releases of the API will solve these shortcomings and can be reintegrated into the framework at a later date.

Figure 5.2.1 is a class diagram showing the extension to the Profile Model interface of the OWLS API. The Profile Model interfaces were changed to support accessing Lists of Actors, Service Parameters and Service Categories. It is currently legal to have several of these types defined in a single Profile Model, so a List structure is used to hold potentially several attributes.

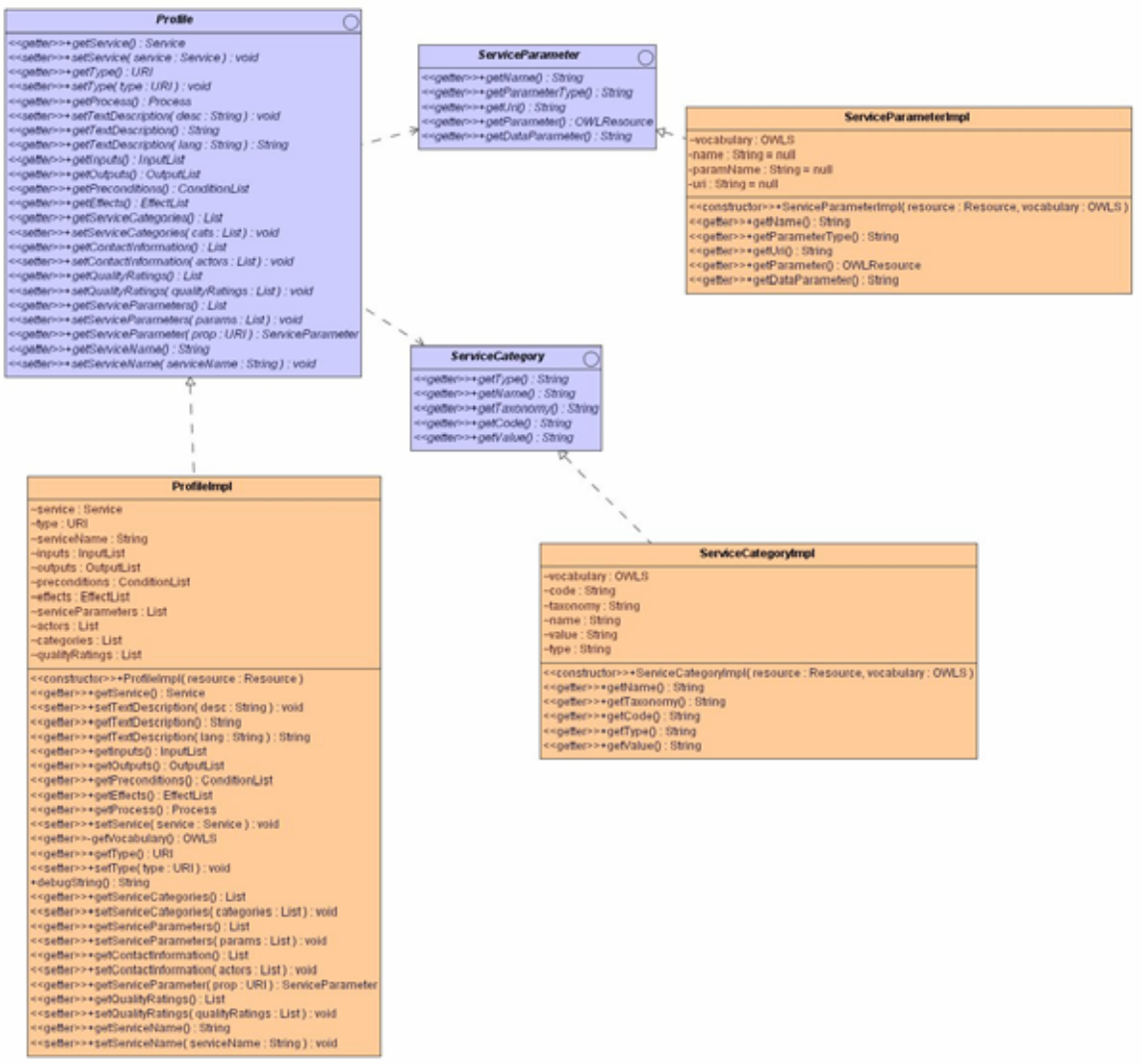

Figure 5.2.1: OWL-S Profile Model classes.

XML vocabulary was also added to the API which allowed the IO Parser extract the Profile Model attributes. This vocabulary was added to the relevant OWL classes in the vocabulary package of the OWL-S API. The IO parser capability was also extended to facilitate reading these new features.

Business Exploration for Web Services (BE4WS): Business Exploration for Web Services (BE4WS) [51] is an open source API provided by IBM which facilitates searching multiple UDDI registries simultaneously using a preconfigured set of key words.

BE4WS was incorporated into this framework and exposed as a Web Service to enable UDDI keyword queries.

5.3 Custom XML Schema

The framework uses a custom XML schema to transport semantically encoded data. The reasons for using a custom format, as opposed to transporting entire ontologies are detailed in Section 7.3.2. During the inference process described in Section 5.5.2, OWL classes or properties are compared to other classes or properties to check whether they are the same resource, equivalent, or related via subsumption. The important information to encode is the semantic type of the resource and the value of the resource. The reasoner component described in Section 5.5 expects input data to be structured in this custom format. This means that the User Profile document containing user preferences, subscribes to this format; as does the list of potential candidates to be reasoned over. The mark-up in Figure 5.3.1 shows an example of the XML custom format.

```
<?xml version="1.0"?>
<item>
…
<attrib type=”http://somelocation/ontologies/profileModel.owl#CurrencyType”>EUR</attrib>
<attrib type=”http://somelocation/ontologies/profileModel.owl#PaymentType”>Visa</attrib>
…
</item>
```

Figure 5.3.1: XML document segment detailing the custom format.

5.4 Capability Advertisement and Service Discovery

The advertisement and discovery functionality is documented in this Section. Capability Advertisement is achieved by publishing the semantics of an OWL-S Profile Model into a UDDI registry. Service Discovery is provided by facilitating a keyword search on UDDI registries and using the reasoner component to examine the discovered services capabilities.

5.4.1 Publishing OWL-S Semantically in JUDDI

For the example scenario, the JUDDI [32] UDDI registry was used to store the fictional E-Businesses. JUDDI is a Java based open source UDDI registry that supports version 2.0 of the UDDI specification. The fictional services are published into the UDDI registry by linking extracted keywords that describe the Service to the UDDI entry. The class `UDDI2OWLSTranslator` provides this functionality. In Figure 5.4.1, the text EuroCurrency would be extracted from the example Profile Model, and mapped as a descriptive keyword for describing this service in the UDDI registry.

```
…
<profile:serviceParameter>
      <concepts:Currency rdf:ID="EStore-currencyType">
            <profile:serviceParameterName>Currency
Type</profile:serviceParameterName>
            <concepts:currency rdf:resource="&concepts;#EuroCurrency"/>
       </concepts:Currency>
 </profile:serviceParameter>
…
```

Figure 5.4.1: service parameter mark-up segment from a Profile Model.

5.4.2 UDDI Discovery

For discovery, a Web Service enabling a collection of UDDI registries to be queried was developed. This search abstraction de-couples the Agent from the underlying OWL-S discovery mechanism. This will allow a semantic discovery API to be integrated into the framework at a later time, when work in this area has matured. The current discovery implementation uses the IBM UDDI query component BE4WS [51]. The implementation is configured by an XML document which defines what UDDI registries to search and what keywords are to be used.

After the discovery process, a list of URI's are returned which point to the Services Access Points. This implementation can easily be substituted for the capability based approach at a later time by substituting new functionality for the Class `UDDISearchImpl`. Figure 5.4.2 details the main classes used for discovery and publication.

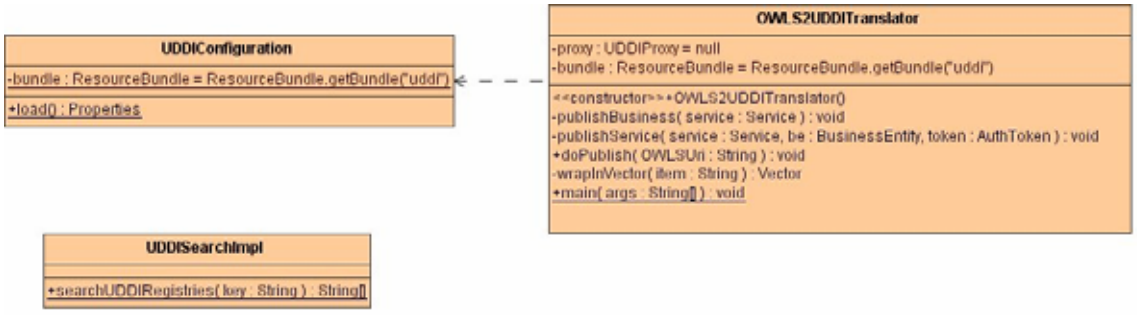

Figure 5.4.2: class diagram showing discovery functionality.

5.4.3 Discovery Sequence

The discovery process involves invoking the UDDI search component which is deployed as a Web Service. The configuration for the search process is stored in an XML document on the server which is read into memory prior to searching. This document tells the search engine what UDDI servers to query and what predicate and keywords to use. The framework caches this for future use. The service

discovery operation is only invoked when the user has selected non-functional service requirements. A Servlet handling HTTP web request invokes the `invokeDiscovery` method.

The next phase of the process instantiates an Agent object and requests it to invoke the UDDI search module. The Agent uses the generic Web Service invoker `ServiceInvoker`[16] to send a SOAP call to the UDDI module. The results returned are validated by the Agent to check whether the Access Points of each UDDI result are valid OWL-S documents. The services capabilities are then examined to ensure they meet the frameworks requirements.

The valid ontologies are added to the frameworks global ontologies collection, and are then returned to the Servlet client as a cloned collection. Figure 5.4.3 details the sequence of methods involved in the discovery process. The sequence only details the Servlet and Agent interaction, some components are left out for clarity.

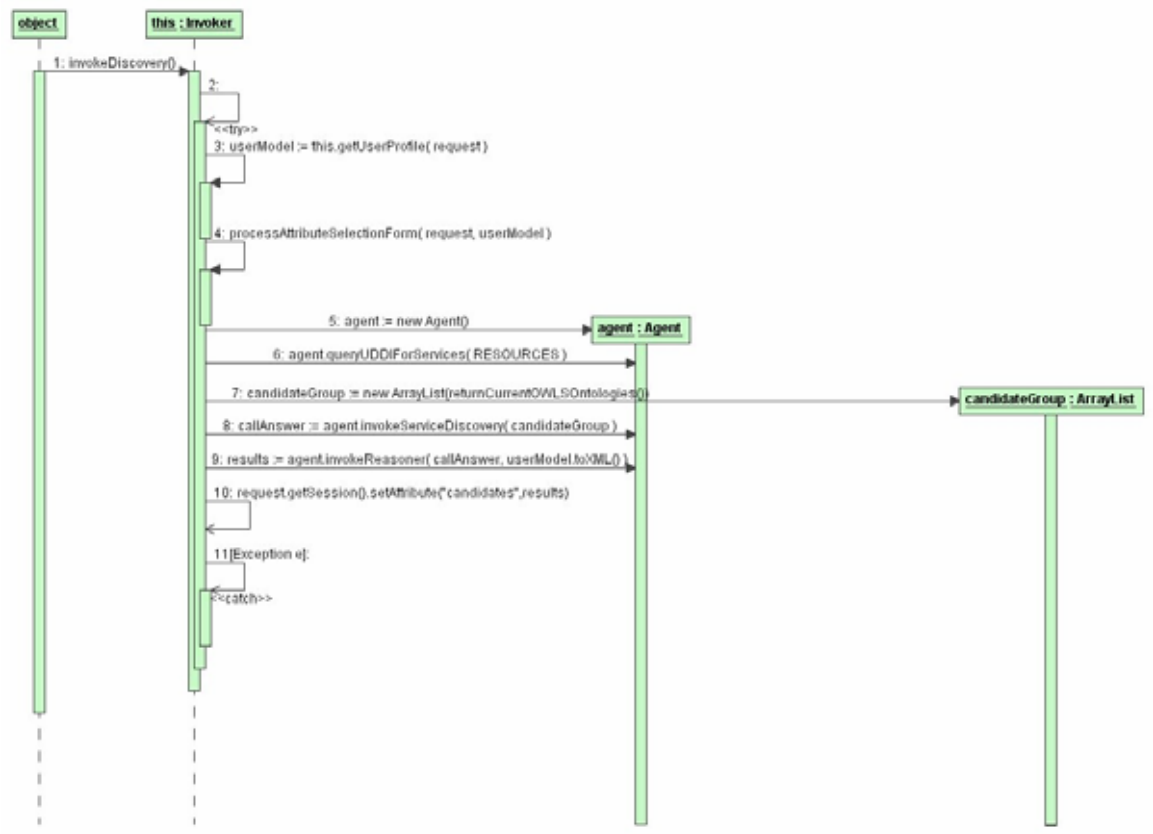

Figure 5.4.3: sequence of method calls in the discovery process.

[16] See Section 5.8

Once Services are discovered, the Agent then extracts the non-functional service attributes from all the Profile Models of the OWL-S ontologies, and returns them as a collection of XML documents compliant to the frameworks XML schema. This custom schema is explained in Section 5.3. A reasoner component is instantiated and used to select candidate services that match the user's non-functional preferences. The candidate services are returned to the user for further selection. Section 5.4 explains the workings of the candidate selection and the reasoner component in more detail.

5.5 Candidate Selection

To provide candidate selection functionality, a custom semantic reasoner component was developed. The selection process involves examining a user's profile, extracting the individual elements details and comparing them to a collection of semantically encoded knowledge.

5.5.1 Semantic Reasoner

The "Reasoner" package and sub packages contain the Reasoner interface and implementations and a Reasoner Event Model. There are also two inference implementations that can analyse semantic relationships between OWL classes. The reasoner package also contains a Web Service Parameter broker component that can resolve what input parameters an OWL Process requires and resolves and extracts them from the clients XML Profile. This process assumes that a client has the required inputs stored in their profile. This aids with automated invocation of services. Figure 5.5.1 shows an overview of the reasoner and inference classes.

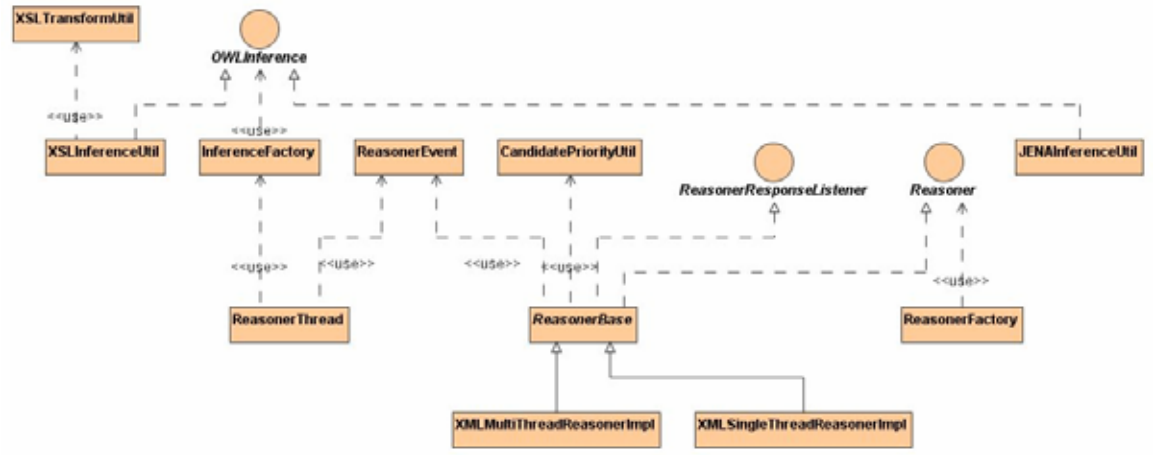

Figure 5.5.1: class diagram showing the reasoner and inference packages.

5.5.2 Inference Functionality

The inference functionality is used during candidate selection, service composition and automated service invocation. Figure 5.5.2 shows the main classes providing inference functionality. The inference

classes are called `JENAInferenceUtil` and `XSLInferenceUtil`. `JENAInferenceUtil` is implemented using the Jena API [54] and `XSLInferenceUtil` is implemented by using Extensible Style Sheets (XSL) [35] to extract the semantic properties. Jena and other third party components used or extended during this research are details in Section 5.2. XSL is explained fully in Section 6.2.1.3. The inference algorithm used is shown in Figure 5.5.2.

```
      public boolean hasSemanticMatch ( URI comparator )
        throws Exception {

        if ( isSameClassAs ( comparator ) ) {
            return true;
        } else if ( isSamePropertyAs ( comparator ) ) {
            return true;
        }

        this.resourceModel = createOntModel ( comparator );
        if ( hasEquivalentClass ( resourceModel, comparator ) ) {
            return true;
        } else if ( hasSuperClass ( resourceModel, comparator ) ) {
            return true;
        } else if ( hasSubClass ( resourceModel, comparator ) ) {
            return true;
        } else if ( hasEquivalentProperty ( resourceModel, comparator ) ) {
            return true;
        } else if ( hasSubProperty ( resourceModel, comparator ) ) {
            return true;
        } else if ( hasSuperProperty ( resourceModel, comparator ) ) {
            return true;
        } else {
            return false;
        }
    }
```

Figure 5.5.2: details the Java code for the matching algorithm.

Both inference classes have the same public interface by implementing the `OWLInference` interface. The framework uses a factory capable of producing either object type at run-time. Figure 5.5.3 shows a more detailed class diagram of the classes involved in inference.

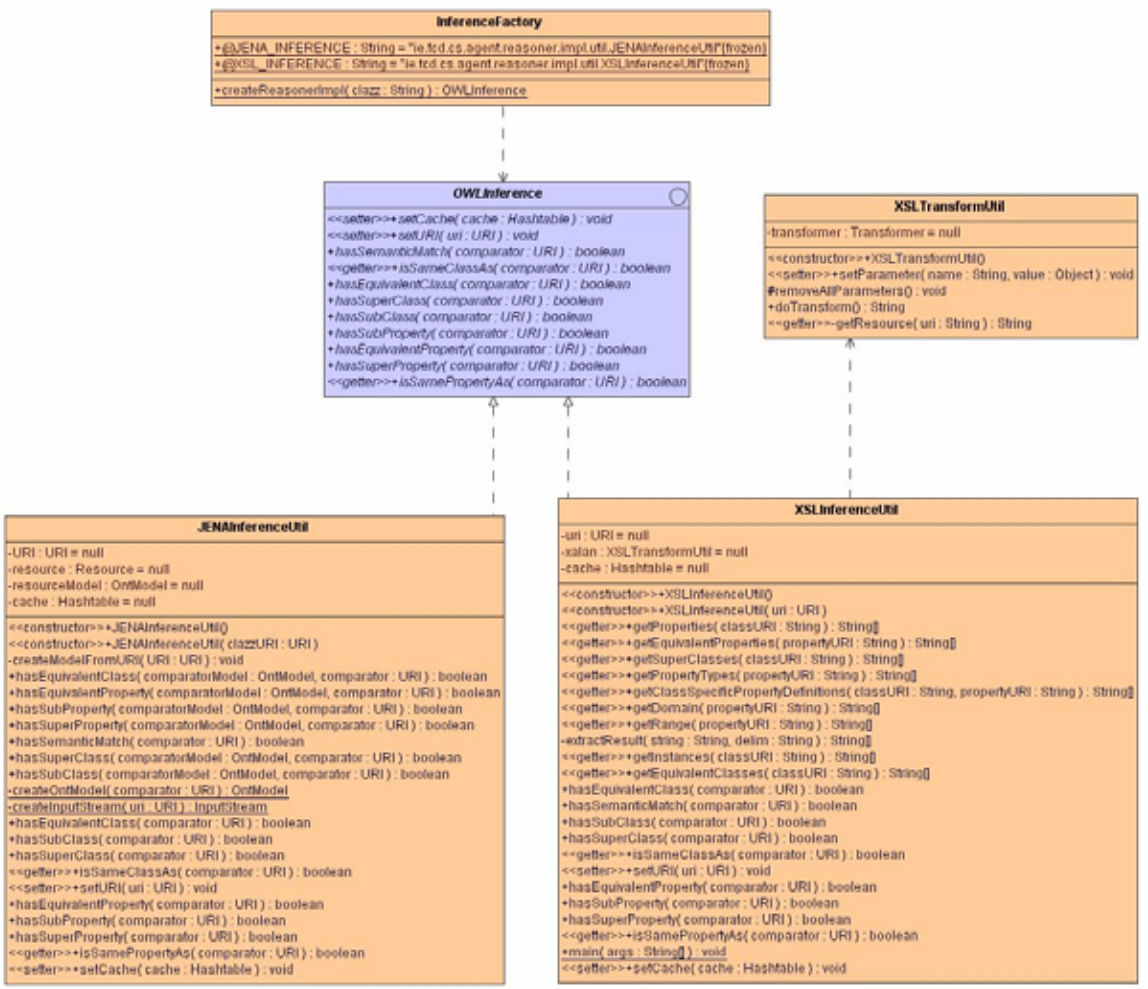

Figure 5.5.3: class diagram showing the inference functionality.

5.5.3 Reasoner Component Overview

The Reasoner component is used by the framework to automate candidate selection. Figure 5.5.4 shows an overview of the elements used in the reasoning process. The Reasoner requires a set of user preferences, in the form of a semantic user profile to be loaded into the component. The reasoner will use this profile as the knowledge base to assert during inference execution. The XML documents that encode the semantic data are also loaded in as input into the Reasoner.

The Reasoner calls out to the inference component to evaluate potential candidates that match the user's preferences. The output of the Reasoner is a list of XML documents that are relevant to the user's preferences, sorted by the highest number of matches first. This list is the selected candidates in priority ordering.

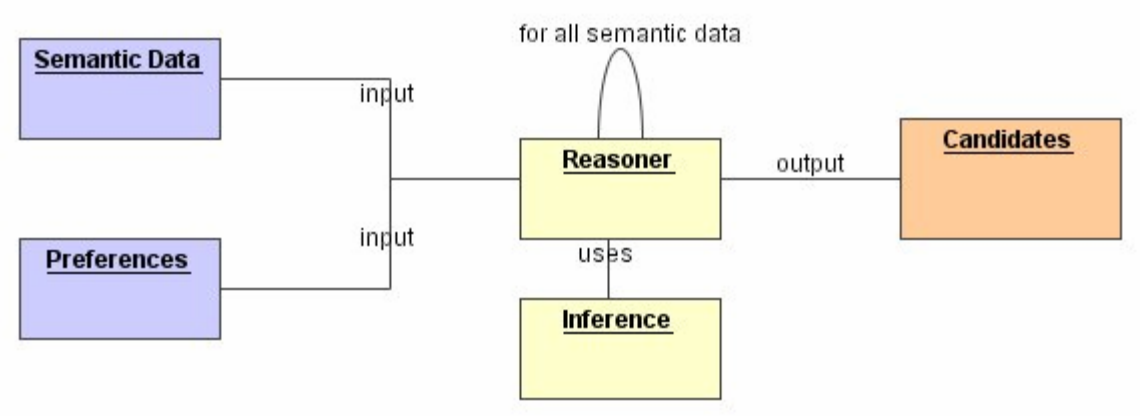

Figure 5.5.4: reasoner component overview and interaction.

5.5.4 Reasoner Functionality

There are two Reasoner implementations included in the framework. One implementation is multi-threaded and the other is a single thread model. Both reasoners extend the abstract class `ReasonerBase` which defines operations and functionality that are common to both implementations.

The inference functionality is provided by the class `ReasonerThread`. `ReasonerThread` uses the `InferenceFactory` to instantiate an inference engine at run-time.

An event model [31] is used to facilitate notification when an inference cycle terminates. The event model is similar to events models defined in the Java Beans [55] component architecture.

`ReasonerResponseListener` is an interface that allows implementing classes be notified when the inference cycle ends. It defines one method for implementation. The `ReasonerEvent` class is used as an event object and stores the results of an inference cycle. `ReasonerBase` implements the `ReasonerResponseListener` interface so as it can receive the event notifications.

`ReasonerBase` also implements the `Reasoner` interface, which defines three lifecycle methods. `Reasoner` is the frameworks interface that allows potentially other reasoners to be plugged into the framework. The methods signatures are detailed in Figure 5.5.5.

```
        // method interface to add a XML user profile as a knowledge base.
        public void addUserProfile(String profile);

        // start the reasoner cycle against a set of candidates.
        public void startReasoner(List candidates);

        // returns the result of the reasoner (XML document).
        public String getAnswer();
```

Figure 5.5.5: reasoner interface signature.

The `addUserProfile` interface allows state to be added to a reasoner prior to execution in the form of a XML based user profile. The `startReasoner` method runs the reasoning logic and when the call returns, the client can access the candidate results in XML format, using the `getAnswer` method.

The Java code for the reasoning logic is shown in Figure 5.5.6. This code is shared by both the single and multi threaded reasoner as it is defined in their super class. The method `startReasonerInstance` called inside the `startReasoner` method is an abstract method and requires implementation in a subclass before the reasoner can be instantiated. This mechanism allows the single thread model to provide different functionality then the multi thread model.

```
public void startReasoner ( List ontologies ) {
        … start omitted for clarity

           // atribIterator is an iterator containing users profile
           // XML elements
            while ( attribIterator.hasNext (  ) ) {
                Element element = ( Element ) attribIterator.next (  );
                Hashtable cache = new Hashtable();
                startReasonerInstance (
                    element,
                    ontologies.toArray (  ),
                    cache );
            }

            // Loop to halt execution while multithreads are running.
            // This will not infinite loop in single thread model,
            // because execution will already be finished when it
            // arrives here.
            while ( !executionStatus ) {
                ;
            }
            // sort list
            prioritiseCandidates (
                map,
                results );

        … end omitted for clarity
 }
```

Figure 5.5.6: details the Java code that controls inference execution.

5.5.5 Reasoner Inference Thread

`ReasonerThread` is a class concerned with executing an inference cycle against a set of potential candidates. The `ReasonerThread` allows its execution to be monitored via the reasoner event model. This allows the class `ReasonerBase` to be notified of the inference cycles results. On successful execution of an inference cycle or an application error, an event is fired to all objects monitoring the execution of the inference.

The listener interface `ReasonerResponseListener` defines an event call back method `reasonerResponse`. `ReasonerBase` implements functionality for this method. By using status codes the application can interpret the events response. Figure 5.5.7 shows the code from the `ReasonerBase` class that provides event handling.

```
public synchronized void reasonerResponse ( ReasonerEvent e ) {
        if ( e.getCode (  ) == ReasonerThread.DONE ) {
            this.map.putAll ( e.getMap (  ) );
            this.finishedThreads++;
            this.threads.remove ( e.getSource (  ) );
        } else if ( e.getCode (  ) == ReasonerThread.ERROR ) {
            this.finishedThreads++;
            this.threads.remove ( e.getSource (  ) );
        }

        if ( this.finishedThreads == this.threadNumber ) {
            executionStatus = true;
        }
}
```

Figure 5.5.7: details the Java code for handling inference thread notification.

If the status code signals DONE, the encapsulated candidates are added to the global candidate matches. When every inference cycle has signalled that it has completed execution, an execution flag is set to true. This flag signals to the reasoner to continue execution and merge all the candidates into a prioritised list. The priority ordering is dependent on the amount of matches a candidate XML document registered with the user's preferences elements. The documents with the greatest number of semantic matches will be returned at the top of the queue.

5.5.6 Candidate Priority Ordering

This priority ordering of candidate's is achieved by the class `CandidatePriorityUtil`. It is used in ranking the inference results. It orders the candidates based on a "most matches" count. The object encapsulates a document and an integer value which is incremented every time a match is evaluated to

true on the document. The interface `java.util.Comparator` is implemented by `CandidatePriorityUtil` in order to allow candidates to be sorted against each other.

Figure 5.5.8 is a class diagram showing a subset of the reasoner classes. Detailed are the `ReasonerFactory` for Reasoner instantiation and the `Reasoner` interface. The abstract `ReasonerBase` class, the `CandidatePriorityUtil` helper class and the event model classes are also shown.

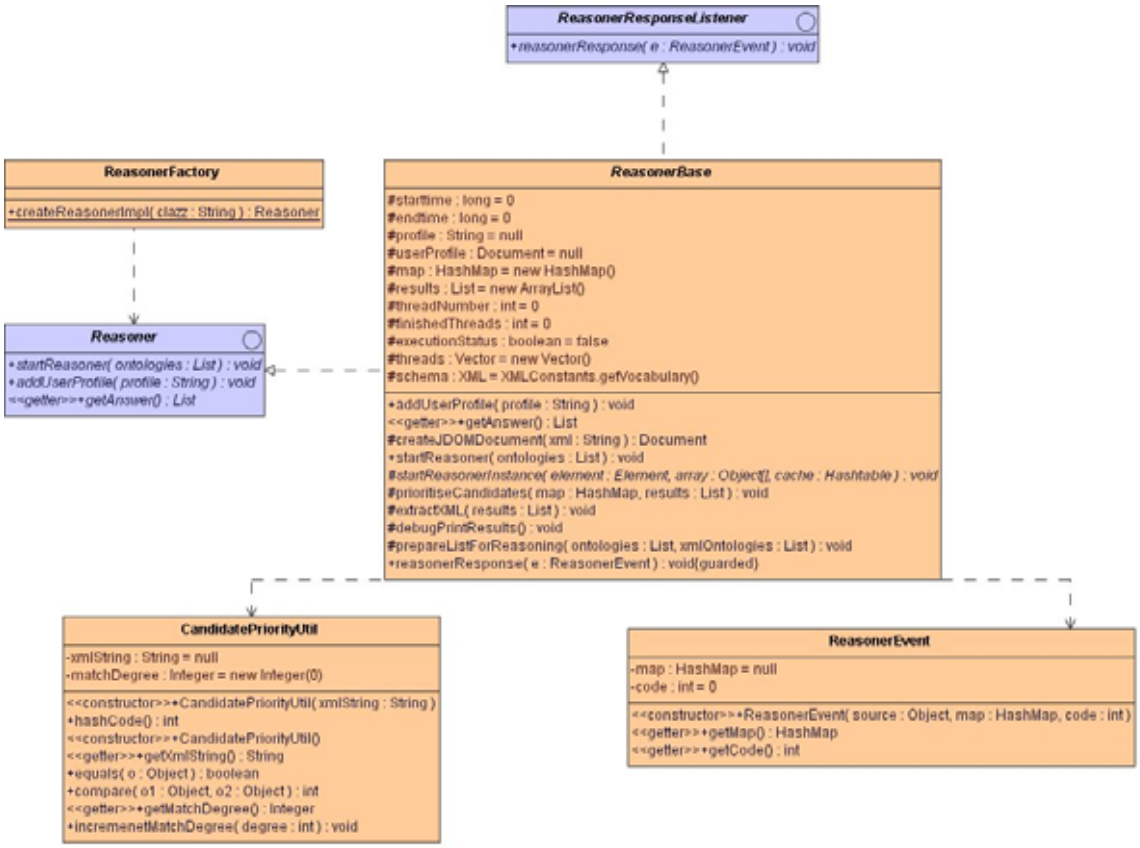

Figure 5.5.8: class diagram showing a subset of the reasoner classes.

Figure 5.5.9 shows the rest of the classes involved in reasoning; they are omitted from Figure 5.5.8 for clarity. The two reasoner implementations extend the abstract `ReasonerBase` class. `ReasonerThread` is used by the reasoner implementations to carry out inference. It is bound to the `OWLInference` interface for inference functionality. Note also how `ReasonerThread` is only bound to the OWL inference interface and not the Jena and XSL implementations. This allows substitution of inference implementations at run-time.

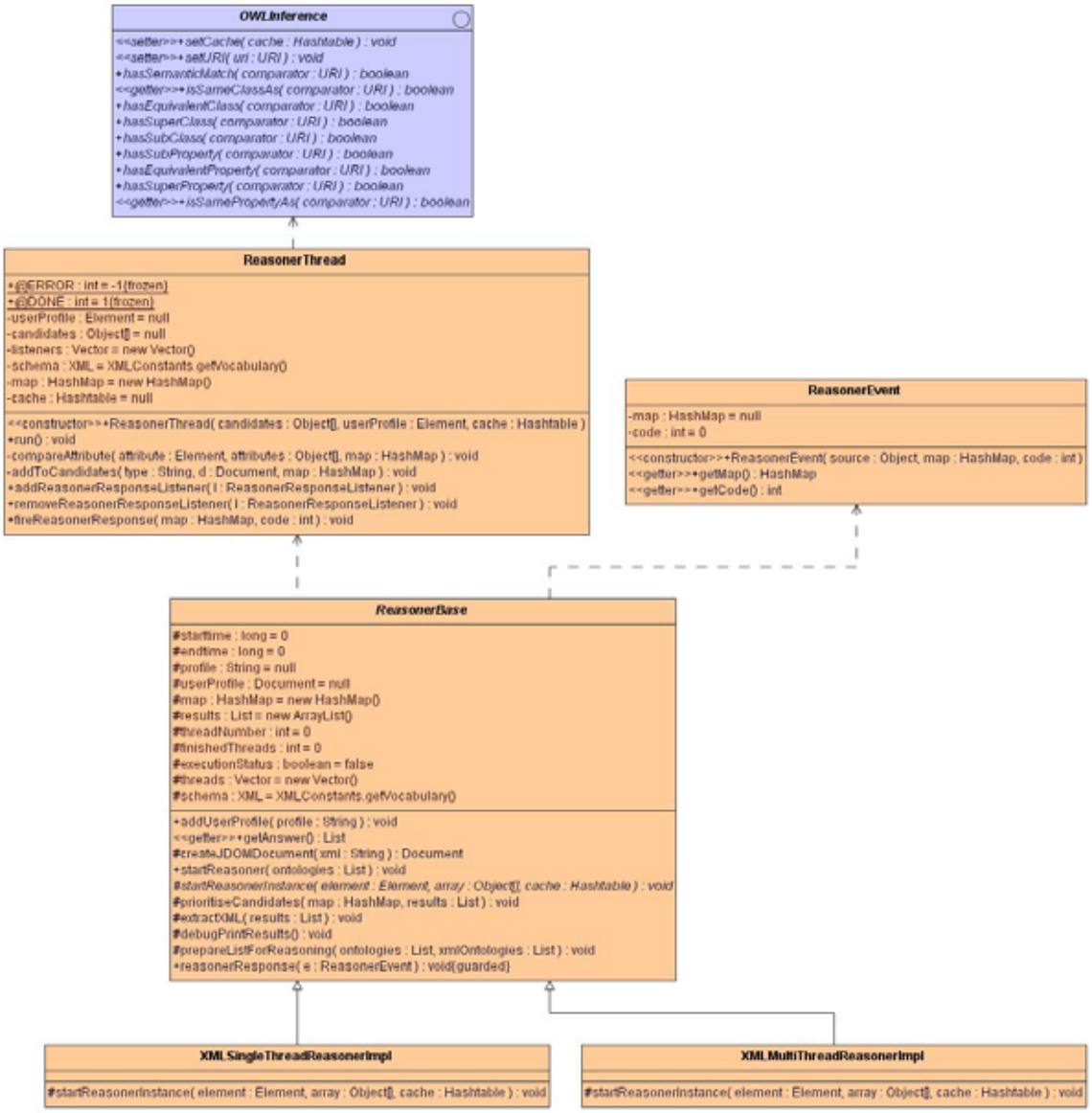

Figure 5.5.9: class diagram showing the reasoner and inference relationship.

5.5.7 Reasoner Sequence

The sequence diagram in Figure 5.5.10 explains the execution of the XML reasoner currently in use. The execution begins when a client invokes the `startReasoner` method, which takes one parameter, a list of XML strings. These XML strings are parsed into XML document objects, using JDOM. It is assumed that the XML strings were formed using the frameworks custom XML schema. The users profile, provided by the Agent is also converted to a JDOM object, and for each element in the user profile, a new `ReasonerThread` is initialised to iterate over the collection of XML objects in the candidate group.

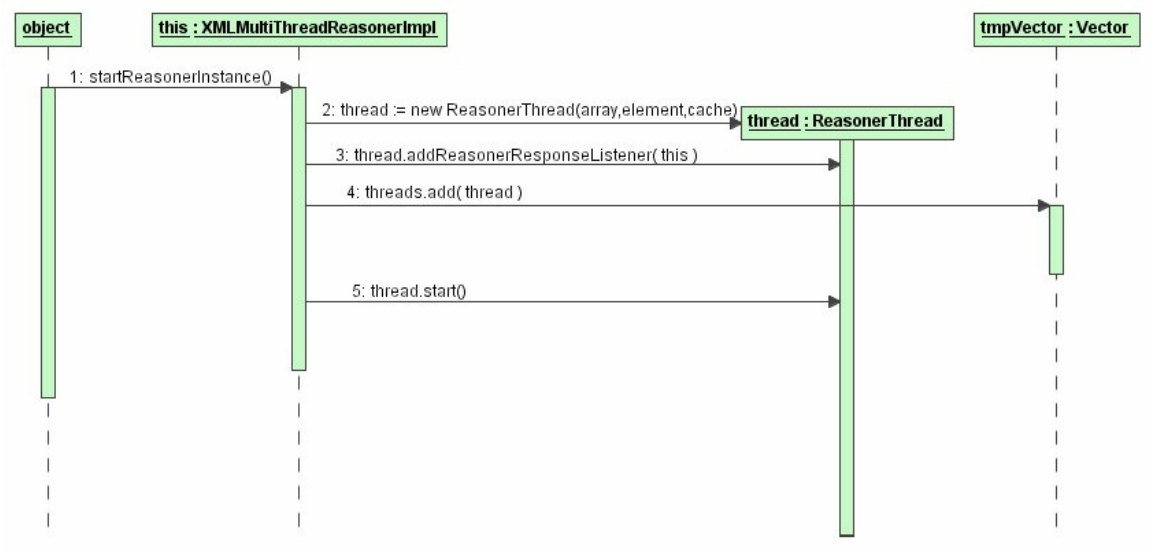

Figure 5.5.10: sequence of messages involved in the "startReasonerInstance" implementation.

The parent thread (current root thread) adds itself as a listener to the child thread by using the `ReasonerResponseListener` interface provided by the framework. The parent thread suspends its execution until the threads call back by starting a while loop on a flag that will only become true when all threads return status notifications.

The `ReasonerThread` uses the `OWLInference` interface to test whether the URI in each element of the user profile XML nodes[17], is a semantic equal, equivalent or subsumption of the comparator. If the result of that operation is true, the XML comparator document is added to the candidate list. And as mentioned earlier this queue is prioritised. Figure 5.5.11 shows the sequence of calls during inference.

[17] that defines the elements semantic type

Figure 5.5.11: sequence of message sent during the inference process

5.6 Service Composition Functionality

This section provides a description of how service composition is achieved. Section 5.6.1 explains how OWL-S functionality was incorporated into the framework. Section 5.6.2 details the composition capabilities.

5.6.1 OWL-S Functionality

The OWL-S package contains classes that can carry out functionality specific to OWL-S operations. Functionality for IO operations, validation of services, process selection and invocation of OWL-S enabled services, as well as ontology management are provided. The package provides an abstraction to an underlying OWL-S API described in Section 5.2 [50].

This API has been adapted into the framework, but is not exposed directly. The framework class `OWLSOntology` provides interfaces to allow clients invoke OWL-S services dynamically ascertain whether a service has desired IOPE's and a method of returning a Services Profile Model description in XML format. It also allows for loading and validating OWL-S ontologies. This class uses the API to achieve its functionality, but encapsulates the inner-workings of the API. This enables substitution of APIs at a later time if necessary.

The valid OWL-S ontologies that are discovered during the discovery process are stored in memory by using the `OWLSOntologies` singleton instance. Figure 5.6.1 shows the main classes in the frameworks owl-s package.

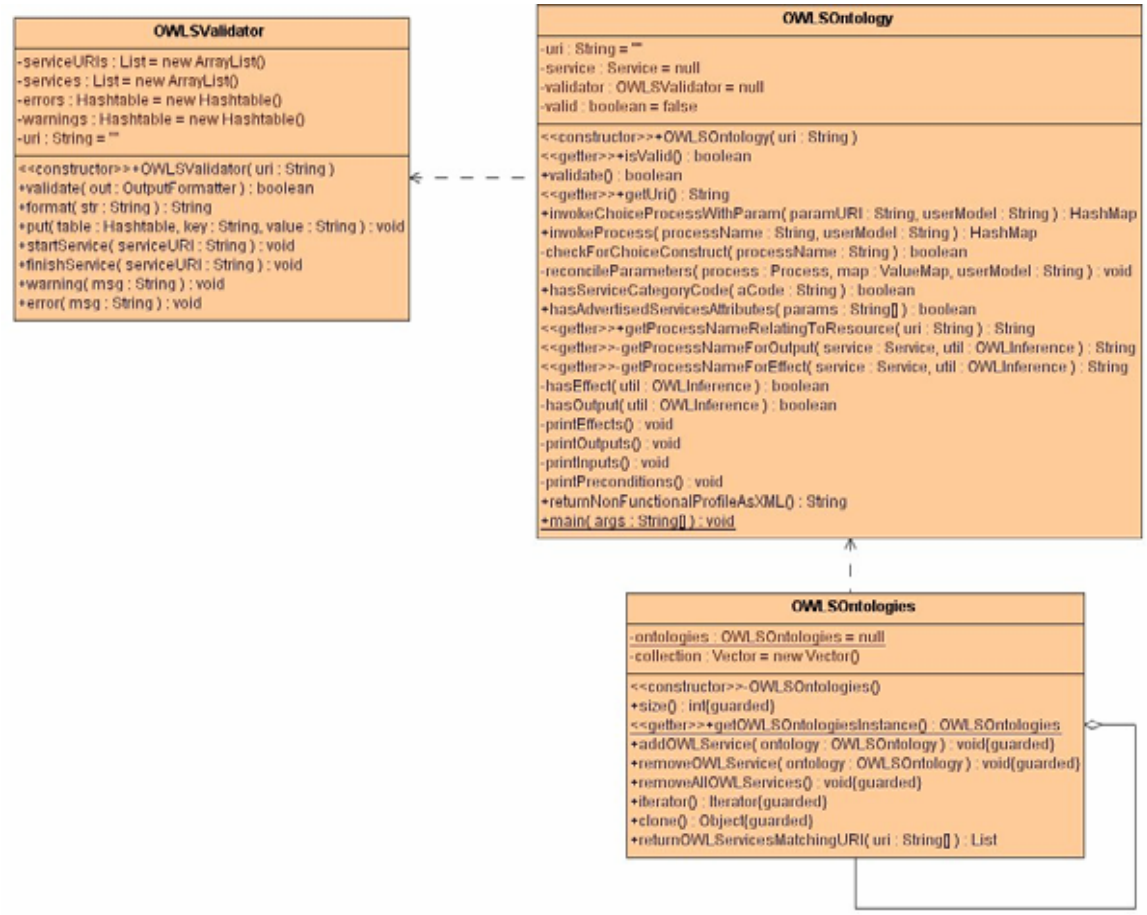

Figure 5.6.1: the OWL-S functionality is realised by the OWL-S package.

5.6.2 Executing a Service Composition

The framework facilitates assembling and executing a service composition plan. However, automation of service composition at runtime involves introducing a planning component capable of determining what actions are necessary to achieve a desired outcome. This planning functionality is out of scope of this research. Service composition is only supported by formulising a semantic plan a priori to framework deployment. A plan is represented by expressing desired semantic concepts in some order. The framework is able to discover and arrange a set of services that best match each of the goals, and invoke them sequentially using the execution engine described in Section 5.7. Figure 5.6.2 shows the sequence of calls in the service composition process.

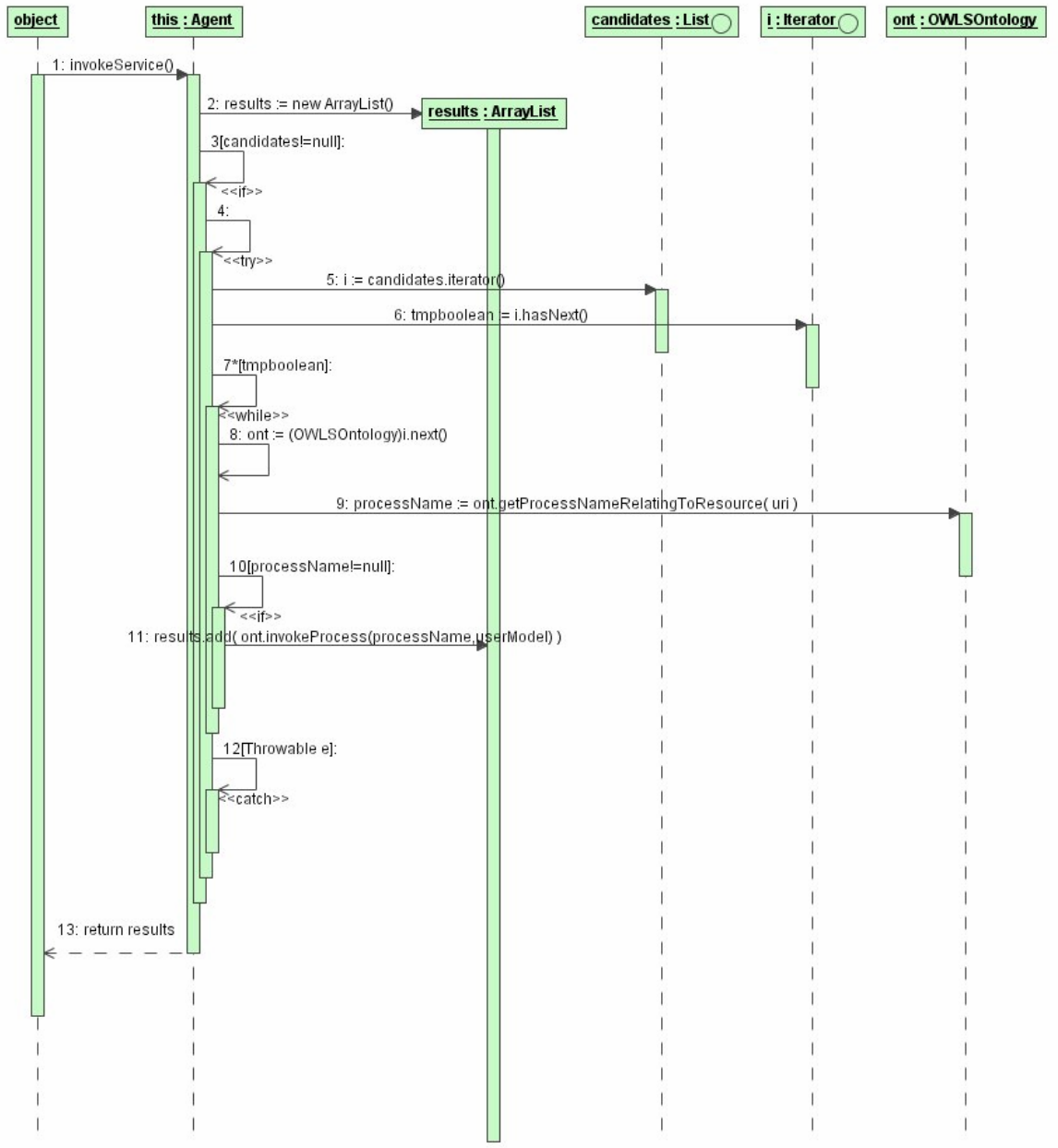

Figure 5.6.2: sequence for how the Agent invokes a composition of processes.

5.7 Automated Invocation Functionality

The framework is capable of dynamically invoking a Web Service using semantic descriptions supplied in the Process Model. The OWL-S functionality described in 5.6.1 allows for OWL-S Processes to be identified which exhibits some known IOPE's. Prior to execution, the user's semantic profile is passed into the execution engine. The parameter broker attempts to reconcile each input parameter required against some value in the OWL-S Process Model ontology. This process involves using the inference functionality explained in Section 5.5.2 to examine whether each attribute in the user profile is the same

as, equivalent to or a sub class of the inputs semantic type. Figure 5.7.1 provides the Java code that enables this reconciliation. It is assumed that the user profile will contain all the domain knowledge required to invoke a service. This knowledge may be ascertained via asking users for input.

```
public void reconcileParams ( HashMap possibleParams,
                              Process process,
                              ValueMap map )
        throws Exception {
        InputList inputs = process.getInputs ();
        Iterator i = inputs.iterator ();

        while ( i.hasNext () ) {
            Input input = ( Input ) i.next ();
            Iterator p = possibleParams.keySet ().iterator ();
            while ( p.hasNext () ) {
                String param = ( String ) p.next ();
                try {
                    URI paramURI = new URI ( param );

                    OWLInference util =
                                InferenceFactory.createInferenceImpl
                                       ( InferenceFactory.JENA_INFERENCE );
                    util.setURI( paramURI );

                    StmtIterator it =
                                input.getJenaResource ().listProperties ();

                    while ( it.hasNext () ) {
                        Statement statement = it.nextStatement ();
                        try {
                           String objURI = statement.getObject ().toString();
                           URI uri = new URI(objURI);

                           if ( util.hasSemanticMatch ( new URI(objURI) ) ) {
                               URI name = input.getURI ();
                               map.put ( inputs.getParameter ( name ),
                               possibleParams.get ( paramURI.toString () ) );
                           }

                        }
                        catch ( Exception e ) { }
                    }
                }
                catch (Exception e){ }
            }
        }
    }
```

Figure 5.7.1: details the Java code showing the parameter reconciliation algorithm.

The output of an invocation is encapsulated in an Object array. An application developer is required to use the output, perhaps by rendering it to a user interface for a user to interpret, or perhaps using the invocation output as input into the semantic reasoner for further inference. The result of this invocation is added to the user profile, so as subsequent invocations can use the data outputted as possible input for the next operation. The sequence diagram shown in Figure 5.7.2 shows the message calls involved.

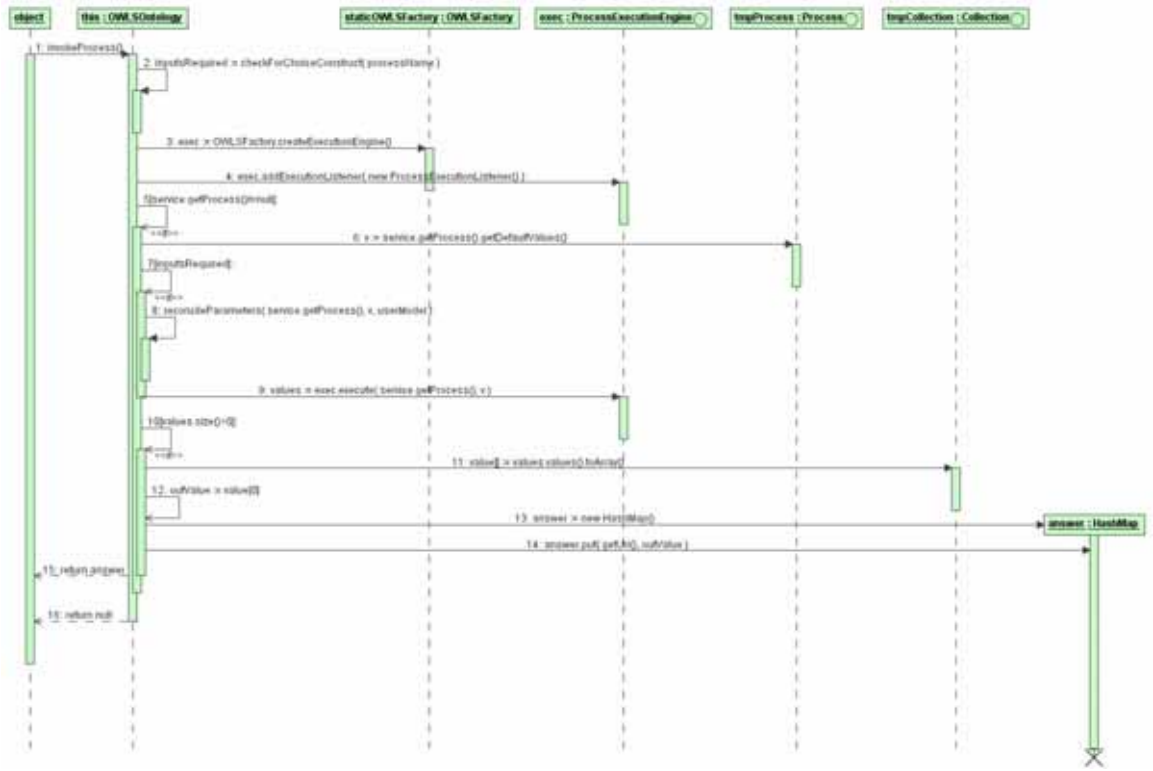

Figure 5.7.2: sequence for executing an OWL-S Process.

5.8 Universal Functionality

The utility package contains classes that are generic to the entire framework. These classes include `XMLConstants` which defines the XML schema strings used in the custom XML format. It is advised that all classes reference the String variables in this class to allow the syntax to be changed with out breaking the entire application.

There is also a class called `OntologyUtil` that can parse, merge and discover semantic knowledge from inspecting ontologies. There is also a generic Web Service invoker that is used to invoke Web Services that are not marked up in OWL-S, for example the UDDI Discovery component. The service invoker examines a WSDL file and constructs a SOAP message using parameters supplied by a calling client.

The `UserProfile` class enables an XML user profile document become an object in memory. It can be instantiates by passing it a valid XML document that adheres to the frameworks schema. Figure 5.8.1 is a class diagram showing the utility classes the sub system shares.

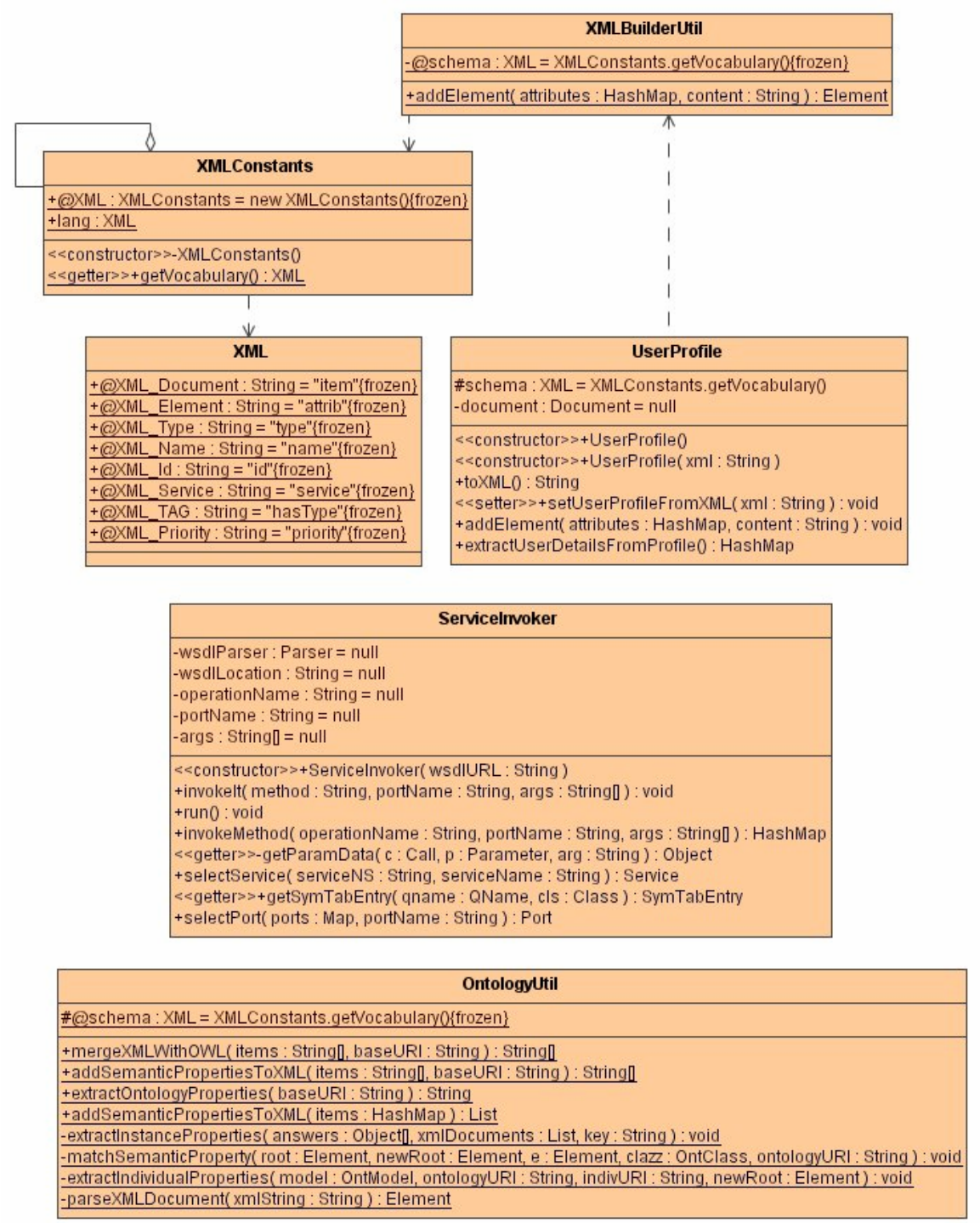

Figure 5.8.1: class diagram showing the generic utility classes.

Chapter 6

FRAMEWORK EVALUTION

6 Framework Evaluation

This research has proposed and demonstrated a method for applying Semantic Web concepts to automate and facilitate the discovery, selection, composition and invocation of Web Services. It has used a specification called OWL-S to achieve this, as well as leveraging current Web Service technologies such as UDDI, WSDL and SOAP and incorporating them into a semantic framework.

Section 6.1.1 discusses the discovery portal, Section 6.1.2 discusses the candidate selection implementation, Section 6.1.3 discusses the service composition functionality and Section 6.1.4 discusses the automated invocation functionality. Section 6.2.1 details the performance of the inference engine used in this implementation. Section 6.2.2 evaluates the performance of the reasoner engine used in this implementation. Section 6.2.3 analyses the trade offs between storing an ontology in flat file format and a relational database. Section 6.3 presents an evaluation of the APeLS reasoner component developed by KDEG, which was considered for usage to provide reasoning functionality during design phase of the framework, but not incorporated in this implementation.

6.1 Implementation Evaluation

The following sub-sections of Section 6.1 takes an identical format. The evaluation begins with referring back to Sections 1.4 where we describe the problems that we sought to address. We continue by briefly describing the solutions we have developed to address these problems. A more detailed description of our approach is contained in Chapter 5. We conclude by evaluating the approach taken, considering the design decisions made and the consequences of those decisions.

6.1.1 UDDI Discovery Portal

The discovery process required allowing semantically annotated Web Services to be found which could satisfy known requirements. These requirements were expressed as semantic concepts. UDDI was chosen as the technology for discovery due to its industry backing and apparent adoption by technology vendors.

The solution implemented was to extract the details and relevant keywords from an OWL-S Profile Model and map these to a UDDI Business Entry. The Access Point of the UDDI Business Entry was mapped to the OWL-S Service URI being published. A subset of these keywords was then used as

search keys in a UDDI discovery query. This method facilitated a targeted keyword search on a UDDI registry. The results returned were validated to check whether the Access Points of each result were valid OWL-S ontologies. The services capabilities were then examined to ensure they meet the portals requirements.

This solution has a number of drawbacks. Notably, the capability assessment of the services was done after the services were returned from the UDDI registry. This means that it is possible that the query will return services that do not meet the portals requirements, which can be inefficient. A more elegant approach would be to extend a UDDI registries capabilities to allow for inference and capability matching to be performed on the UDDI server and extending the UDDI protocol to facilitate semantic capability searches. This approach has been adopted by the DAML-S Matchmaker project. It is proposed that such an approach could be integrated into this framework.

6.1.2 Candidate Service Selection

The candidate selection requirement involved identifying Web Services in some priority ordering that best satisfy a set of user preferences. This process is typically done at discovery and/or composition time. A candidate service is classed as any service that matches any of the capabilities sought. A candidate that meets all the users' requirements will be ranked as the most suited candidate.

To provide candidate selection functionality, a custom semantic reasoner component was developed. The selection process involves examining a user's profile, extracting the individual elements semantic details and comparing them to a collection of semantically encoded knowledge.

The reasoner relies on XML documents that are formed using a custom XML schema. One limitation to this approach is that semantic data stored in ontologies must be stripped out and encoded in this XML format prior to reasoning. This requires extra processing time. Section 7.3.1 details methods for optimising and enhancing the speed of semantic inference. Optimisation methods negate the need to examine entire ontologies when only concerned with specific data. This implies that extracting only the relevant data for a particular concept is an acceptable step to take.

6.1.3 Service Composition

The service composition process requires selecting two or more Web Services and performing a union to provide new functionality. The framework allows a service composition plan to be executed by the service invocation module. A list of services that match a set of capabilities can be identified and this list can be passed to the invocation engine for invocation. The candidate selection component can be used to identify the most appropriate service(s) to achieve a plan. When this candidate list is formulated, the

list can be passed to the execution engine for sequential execution. The output of each invocation is added to the user's profile, for possible use in subsequent invocations.

The framework currently lacks the sophistication to allow a plan to be formulated and execution to be monitored; it currently expects a plan to have been formulated previously. This functionality could be included by using the reasoner and service invocation components to demo a trial run of the composition a priori to execution. It would then be possible to deduce whether or not a composition can be realised, or whether a user needs to provide more input or perhaps the services need to be ordered in a different priority.

6.1.4 Automated Service Invocation

The service invocation requirement involved executing Web Services dynamically, using the encoded semantic metadata as an instruction set. This process involved selecting the appropriate operation to achieve a known outcome and also dynamic reconciliation of input parameters and interpretation of message return types.

The framework supports semantic service invocation by allowing OWL-S Processes to be identified and selected based on user's preferences and the processes capabilities. Dynamic invocation is made possible by facilitating parameter reconciliation at run-time. A user's semantic representation is stored in memory, which is used by the invocation engine to extract appropriate input for a particular service at run-time. All semantic knowledge that is generated by the framework is stored in the users profile to expand the user's representation of the domains world. The invocation engine supports the invocation of different types of composite processes, and uses the OWL-S Process Model to formulate an execution plan.

The solution is dependent on the information encoded in a user's profile. An application developer, using the framework must ensure that the user profile has all the required domain knowledge available in their run-time state. This information can be accumulated by prompting for user input, which can then be converted to the appropriate semantic concept. Using the OWL same as, equivalent, and subsumption relationship it is possible to determine what user profile concept is needed for an input parameter.

For example, in the E-Portal demo application developed using the framework, a user is asked to specify the quantity of a product they wish to purchase prior to invoking the purchase operation. This inputted value is converted to the semantic concept for quantity, defined in the product ontology and

then stored in the user's profile. The profile will then have this meta-data encoded when the execution engine attempts to reconcile the quantity parameter.

6.1.5 Discovery using Non-Functional Service Attributes

It was also an aim of this Dissertation to examine methods for incorporating functional and non-functional service requirements into the candidate service selection process. Non-Functional attribute selection was achieved by defining semantic concepts in a global ontology, and referencing these concepts from OWL-S Profile Models as custom Service Parameters. Section 4.2.1 identified these parameters. The OWL-S Profile Models are consulted during the service identification and selection phase to find services that best suited user's preferences. Functional attribute selection was achieved by grouping together different functional properties and allowing a user select combinations of these properties. A candidate group was then identified containing items that exhibited the user's preferences.

This method for incorporation of non-functional services properties used is not a realistic model for use in an enterprise setting. It does provide a mechanism for identifying services that best match a user's goals, but there is no guarantee that these services have advertised their attributes truthfully. Regulation and reputation management is not enforced and there is no Internet scale enforcement protocol yet standardised. Ontology providers can advertise there services in what ever way they see fit, even if this means providing false information.

Assured Reputation Information Exchanged Securely (ARIES) is a protocol developed by IBM to provide consumers with reliable reputation information and per-transaction assurances when selecting new service providers [19]. It gives web service providers the opportunity to compete on objective qualitative and quantitative metrics such as price and availability rather than subjective and opinionative criteria [19]. ARIES secure reputation and assurance services are intended to be incorporated into existing service acquisition[18] models for both manually and automatically initiated transactions.

It is possible, in the context of the Semantic Web that reputation authorities could be incorporated and used to publish ratings for arbitrary services as an ontology hierarchy. Agents can reason over such ontologies when selecting candidate services and report their experiences of using the service back to the reputation authority. It is argued in [19] that service requestors have been slow to exploit the benefits of a global e-marketplace due to very real concerns about their authenticity.

[18] For example UDDI

6.2 Performance Tests

To access the practical run-time performance of this framework, tests were performed to find out what software configurations and implementations exhibited the best performance. The tests detailed in Section 6.2.1 focused on different methods for performing semantic inference. Tests in Section 6.2.2 examine and contrast different reasoner implementations. Tests in Section 6.2.3 compare methods for persisting ontologies.

All tests were carried out using the following Hardware and Software specifications:

Operating System:	Window XP Professional
Memory:	256 Ram
Processor:	Pentium M 1300MHZ
Ontology Server:	Tomcat 5.1 Web Server running on "localhost".
Java Software:	Java version 1.4.1-4

6.2.1 Inference Overhead

One of the major drawbacks with evaluating semantic relationships between OWL Resources is the overhead involved in IO operations and building object representations of ontologies. When large numbers of inferences are evaluated the requirements on network resources and processor power is intense leading to degradation of performance.

6.2.1.1 Inference Methods

Inference in this semantic frameworks context is the process of testing whether an OWL Resource is the same as, equivalent to, or subsumes another Resource. There are two inference implementations contained in this framework.

This framework uses the two inference implementations for parsing and extracting knowledge from OWL ontologies. An interface is defined to allow different implementations to be plugged in to the framework and a factory is used to instantiate the implementations. The two implementations are discussed in Sections 6.2.1.2 and 6.2.1.3.

6.2.1.2 Jena API Approach

The Jena API developed at Hewlett-Packard labs is a toolkit for building semantic applications. The Jena API provides an implementation of an OWL parser to allow OWL documents and resources to be loaded into an in-memory graph structure. Once loaded, methods providing inference capabilities can be called on the in memory object to allow semantic relationships to be inferred.

6.2.1.3 XSL Approach

As mentioned in chapter two, OWL documents conform to well formed XML syntax. The Extensible Style sheet Language (XSL) family is a W3C standardised specification for defining XML document transformation and presentation [35]. It consists of three parts, XSL Transformation (XSLT), XML Path Language (XPATH) and XSL Formatting Objects (XSL-FO).

XSLT defines a method for extracting XML statements and values from an XML document. An XSLT style sheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses a formatting vocabulary [35].

By defining a style sheet that can identify specific OWL mark-up such as equivalent relations and sub class relations, relationships can be inferred between different OWL concepts. Using this style sheet, an XSL transform engine can load the ontology into memory and apply the XSL file to the ontology. Once applied, data can be extracted from the ontology using an XSL transform API.

6.2.1.4 Inference Performance Tests

As mentioned in Section 6.1.3 and 6.1.4, one inference implementation uses the Jena API and the other uses an XSL implementation to extract desired relationships from an ontologies Document Object Model (DOM) [38]. The majority of the inference cycles carried out by this framework will involve testing a collection of candidate resources against a single resource, in a continuous, exhaustive process. This process may be repeated several times during one inference session on different resources.

6.2.1.5 Jena Vs XSL

There were three test cases identified to compare the performance of the two inference implementations. All tests used the same OWL ontologies as input. Each inference model was initialised with a URI to an OWL Class resource. The OWL Class used was “Wine” from the sample applications Wine ontology. The Wine ontology is a reasonably large file in ontology terms. This OWL class was tested against the class “Product”, also from the E-Commerce demo application. The class Wine is a sub class of Product, so the result of the inference will be true each time. The inference was carried out 200 times, using a control loop. We choose this figure as we believed that for any instantiation of the reasoner component in a real application would incur an inference threshold of 200. All tests were carried out three times and the results were averaged to get the results. The list below is a formal definition of the test cases.

1. When the test case starts, an inference object is instantiated. For all iterations of the control loop, the same inference object is used. This test showed the overhead involved in evaluating a semantic relationship. The result is shown under the *Cached Instance* heading in table 6.2.1.

2. This test loaded a new instance of the Jena and XSL implementations for all iterations of the control loop. This tests the initialisation overhead for each implementation. The result is shown under the *New Instance* heading in table 6.2.1.

3. Finally, a third test carried out two inferences for each loop iteration to see if there is any effect on carrying out multiple inferences. The same inference object was used for all iterations as in test case 1. The result is shown under the *Two Inferences on Cached Instance* heading in table 6.2.1.

The results are compiled in the table 6.2.1 below. Data is represented in seconds and milliseconds.

	Cached Instance	*New Instance*	*Two Inferences on Cached Instance*
Jena	3.68	31.45	4.03
XSL	15.99	16.25	30.01

Figure 6.2.1: results for inference implementations tests.

6.2.1.6 Jena Conclusion

The results show that Jena performance is degraded when it is forced to load a new ontology into memory for each inference cycle[19]. The initialisation time is directly affected by the size and complexity of the ontology. Once loaded, Jena can carry out repeated inferences quickly (3.68 for 200 inferences, 4.03 for 400 inferences).

Jena's implementation builds up an object graph of an ontology into memory, and uses an iterator [31] to traverse the object model. The main overhead in this process is not the iteration mechanism but the initialisation process.

[19] Test Case 2

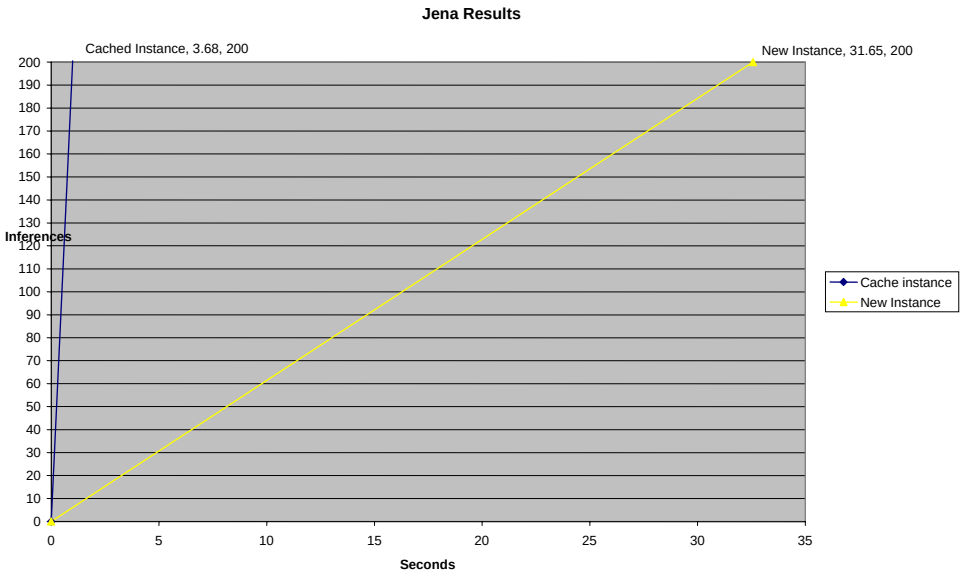

Figure 6.2.1: Jena test results in graph format.

6.2.1.7 XSL Conclusion

The XSL implementation exhibits more predictable performance. If requires a fixed amount of time to carry out an inference and this appears to grows steadily relative to inferences (15.99 for 200 inferences, 30.01 for 400 inferences). By doubling the amount of inferences, an inference cycle takes almost twice as long to complete.

The XSL results are symptomatic of the overhead involved with XSL processing. XML's flexibility places significant demands on network and hardware infrastructure, consuming as much as 80% of server processing power to perform CPU-intensive tasks such as transformation [34].

The XSLT engine will be unaware of the semantic links and relationships between OWL resources in any given ontology, therefore does not need to build a complex object graph like the Jena API. This can lead to a performance improvement on initialisation, but because there is no in memory state of relationships in the ontology, each inference requires more computation then simply testing a variable value, as in Jena.

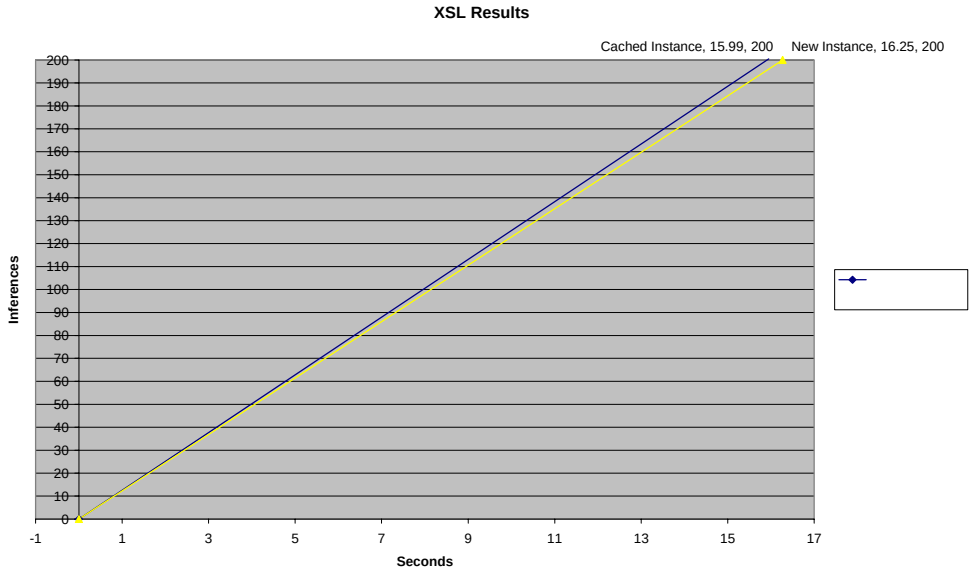

Figure 6.6.2: XSL test results in graph format.

6.2.2 Reasoner Implementations

As described in Section 5.5.1, there are two different inference implementations. A reasoner in this framework is a component responsible for controlling the execution of an inference cycle.

There are four separate implementations to choose from when selecting a reasoner engine at run-time. The options are a single thread model with Jena inference, single thread model with XSL inference, multi threaded model with Jena inference and multi threaded model with XSL inference.

6.2.2.1 Multi Threaded Vs Single Threaded Implementation

The Reasoner starts a new inference cycle for every XML element in a user profile. The multi-thread reasoner implementation spawns a new thread of execution for every inference cycle. This implementation allows several inference cycles to run simultaneously and independently.

The single-thread reasoner executes all inference cycles sequentially. This means that only one XML element in a users profile can be in execution at any particular instance. When one inference cycle returns a new cycle can begin.

The tests involve instantiating a Reasoner component and initialising it with 50 candidate XML documents with 5 elements each and a user profile with 5 elements. This will result in a total of 5 inference cycles totalling 1000 inferences altogether.

There were two sample user profiles used as input for the reasoner to provide variation. One user profile included several concepts found in the demo applications Wine ontology, the other profile contained elements from the Product ontology, which is a considerably smaller file size. By using both sample sets it can be ensured that all the inferences will result in true.

1. This test involved running two inference cycles using the multi thread reasoner implementation. One inference cycle used the Jena inference engine, while the other used the XSL implementation.

2. This test involved running two inference cycles using the single thread reasoner implementation. One inference cycle used the Jena inference engine, while the other used the XSL implementation.

The results are compiled in the two tables below. Data is represented in seconds and milliseconds. Table 6.6.2 shows the reasoner execution using the Wine based XML profile, while table 6.2.3 shows the execution based on the Product ontology concepts.

	Jena Inference Plug in	*XSL Inference Plug in*
Multi-Threaded	252.863	78.68
Single-Threaded	116.34	34.49

Table 6.2.2: results of the first reasoner implementations tests.

	Jena Inference Plug in	*XSL Inference Plug in*
Multi-Threaded	25.019	70.85
Single-Threaded	9.02	32.31

Table 6.2.3: results of the second reasoner implementations tests.

6.2.2.2 Conclusion

The results show that the sequential single thread model executed faster then the multi thread model for all implementation combinations. The Jena and XSL based single thread implementations execute at least twice as fast as their multi thread counterparts. One possible reason for this performance difference is that the test computers RAM limitations hindered the processing capabilities of several threads executing simultaneously. Another possible conclusion is that the cost of passivating and activating a thread is expensive, and not proportionate to the level of processing that needs to be done.

We can be inferred from the results that the Jena implementation performs poorly when the Reasoner is invoked against a set of large ontologies. Table 6.2.2 shows the results of the reasoning cycle using

concepts from the Wine ontology only. The multi thread model took on average 4 seconds for each inference. The single thread model took on average 1 second. Compare these figures to the test results in table 6.2.3, which used the smaller Product ontologies concepts. It is apparent that Jena's performance is reliant on the size of the ontologies used in a candidate set. Table 6.2.3 confirms that using the concepts from the smaller Product ontology improves Jena's multi and single thread model by at least a factor of 10.

The XSL implementations performance was almost identical for the two tests. The single thread model performed nearly twice as fast of the multi thread model. The size of the ontology had no significant relevance to performance. It appears that processing an XSL file in memory is the main performance overhead using this model, and not initialisation of the ontology XML resources.

6.2.3 Ontology Persistence Storage Tests

There were two methods evaluated for storing and accessing ontologies in this framework. One method involved storing the ontology data in a database as relational data. The other model involved storing the ontologies on a Web Server as files.

The two models were implemented using the Jena API. Jena supports reading in ontologies into memory by providing a URI to a Web Server address or a URI to a database server instance.

6.2.3.1 Jena DB Vs Jena HTTP

Two test cases were identified to compare the cost of initialisation of an ontology into memory using flat files and database methods. The applications Wine ontology was loaded into memory for both methods. Prior to the test, the Wine ontology was stored into a MySQL [58] database, using Jena's Database API. The tests cases are defined below.

1. The Wine ontology was loaded into memory twenty five times from the database model and also twenty five times from a flat file. The total time taken was recorded and can be seen in column 1. HTTP Server represents the flat file method and RDBMS is the database method.

2. The Wine ontology was loaded into memory two hundred and twenty five times from the database model and also twenty five times from a flat file. The total time taken was recorded and can be seen in column 2. HTTP Server represents the flat file method and RDBMS is the database method.

	25 times	*250 Times*
HTTP Server	9.02	32.31
RDBMS	25.019	70.85

Table 6.2.4: results of the persistence tests using the Jena API.

6.2.3.2 Conclusion

It is conceivable that by storing ontology data in some kind of persistent database, and by using database indexes and keys, databases could perform quicker then flat file access, due to the assumption that the data has been stored in a valid data format that ensures ontologies are valid. Concepts could be loaded as needed, thus avoiding loading redundant concepts. This lazy loading negates the need for validation of ontologies every time they are accessed. Also, by employing optimisation techniques which target specific RDF triples that are only relevant to a certain query could further enhance inference performance. It would also reduce the level of XML string processing. Validation of ontologies has been identified as a performance bottle neck in Section 7.2.

The tests show that the Jena database implementation does not perform faster then the flat file model. Storing ontologies in flat files allows for faster in-memory initialisation. This can be attributed to its implementation and the overhead of storing ontologies in a relational form.

6.3 Adaptive Personalised E-Learning Service Evaluation

The KDEG group in Trinity College has developed a rule based inference component built on top of JESS [53]. The component is called Adaptive Personalised E-Learning Service (APeLS) [52]. APeLS was designed to allow XML documents to be reasoned over using custom JESS functions specific to DOM models and X-Path queries. The component has been applied to the field of E-Learning. The current implementation of APeLS is built on top of an XML database [33].

It was proposed that the APeLS component would be incorporated / extended to allow it to infer over OWL based ontologies for use in this framework. The APeLS component is capable of producing a candidate result(s) from a set of ontologies, due to its support of X-Path queries. After evaluating the APeLS component, it was not deemed as a suitable inference engine for the composition framework for several reasons.

6.3.1 XML Database constraint

APeLS is constrained to use an XML database implementation. This means that for every instance of APeLS used in the semantic framework, an XML database would need to be pre-populated with the candidate XML documents as well as the current user's profile. This data would also need to be deleted

after execution. The nature of using any database is for persistent storage of data. This is not a requirement for the composer as the inference cycle is done dynamically at run-time with different datasets that have no relevance after the inference cycle.

6.3.2 Pre-Run-time Configuration

APeLS is configured for use by a Resource Bundle[20]. This Resource Bundle specifies the location of the XML database in use and other global properties like the location of the JESS narrative in use. This Resource Bundle is configured prior to deployment of the application.

The semantic framework does not know the make-up of a user's profile prior to execution, and is not aware of what the candidate ontologies are. It is therefore impossible to efficiently pre-configure a specific database.

A knowledge base used during the reasoning process will be built by a user while selecting non-functional and functional requirements. It is not possible to pre-compose a narrative because the process is dynamic by nature.

It is accepted that it is possible to programmatically interact with the JESS shell, at run-time and load into memory a narrative, but the benefit of this was not deemed sufficient for incorporation of APeLS into the framework.

6.3.3 Scalability issues

The framework must support concurrent user requests. Performing write and read operations on one database (preconfigured in the Resource Bundle) by multiple users will pose a concurrency risk. There is currently no interface that allows a developer to hook in a custom "Learner" model. The current Learner model is specific to an E-Learning context, which is not generic enough for general use.

One proposal is that a more abstract interface is provided in future releases, so as developers can plug-in other user models for use in different domains. However, it is accepted that APeLS was designed for use in the E-Learning domain and genericity and re-use was not a primary requirement.

6.3.4 APeLS Builds New Knowledge

The result of APELS run-time execution is to build a new XML document from the results of an inference process, using knowledge encoded into narratives by domain experts. The semantic framework has no need to build a new document or encode any more knowledge that does not already

[20] A configuration file for use in Java applications.

exist in the applications ontologies. It merely needs to select from a collection of documents that exist and use this sub collection to output as a result.

Chapter 7

CONCLUSION

7 Conclusion

In accordance with the goals of this work, a semantic framework has been developed to support discovery, candidate service selection, service composition and automated service invocation. During this research, we first explored the limitations of current Web Service technologies with regards to expressing their semantics. We identified OWL-S as an emerging technology that could address these limitations. An E-Commerce use case was devised that showed how an Internet user could benefit from semantic automation of E-Commerce processes. We developed an application using the semantic framework which demonstrated this claim.

7.1 Research Review

This Section, 7.1; summarises our research output with regards to our initial objective detailed in Section 1.4 and compares and contrasts it to other research profiled in Chapter 3.

7.1.1 Semantic Service Discovery

We identified and implemented a method to enable capability based discovery of Web Services using UDDI as a discovery mechanism and incorporating non-functional service parameters. The method we proposed does not extend the UDDI specification, something that Akkiraju et al. and Paolucci et al. propose.

A limitation to our approach is that it is still keyword based, and requires the publication component to identify keywords that can be used to discover services. Semantics are only introduced after a UDDI inquiry has been processed. This solution is acceptable in a closed world scenario, such as the E-Commerce use case, but is unlikely to scale well when discovery or varied services is required.

7.1.2 Service Selection

We implemented several reasoner implementations to identify suitable technologies that could be used for semantic inference. The implementations showed that Jena is a more suited approach to XSL, but is by no means a perfect model for inference. The overhead involved in inference is concluded in Section 7.3.

Our reasoner did not operate inside an expert system shell, which is the approach taken by Siren et al. Semi-Automatic Composer tool described in Section 3.3.2. Expert system shells normally require some domain specific narrative or rule set to be loaded into the shell. This narrative is often encoded by a domain expert prior to application usage. Our research identified that inference techniques need to be configurable at runtime, meaning reasoners that require static configuration or pre defined narratives are not as suitable as our robust model, which is configurable to any semantic domain.

7.1.3 Semantic Service Composition

We identified that service composition is only feasible by incorporating A.I. planning techniques. This is because it is necessary to understand the outcome of a composition in advance to infer whether or not a collection of services are possible to be composed together, using the current domain knowledge and resources.

Service composition frameworks like SHOP2 [62] and Sirens et al. Semi-Automatic Composer tool use A.I planning to identify viable compositions. It is future work to evaluate these approaches and incorporate such techniques into our framework which will enable dynamic composition of services.

7.1.4 Semantic Service Invocation

We extended an execution engine to facilitate a method for identifying Web Service operations based on the capabilities they offer, and allowed the dynamic construction of SOAP messages used to invoke operations on concrete Web Services. This implementation used a user profile knowledge base as a means for reconciling input parameters.

We extended Siren et al. work on developing an execution API using the beta OWL-S API. We implemented functionality to support the ChoiceOf OWL-S composition construct and also extended the engine to support the interpretation of Array XSD data types in response and request operations.

One important merit of our approach to semantic invocation is that complex OWL concepts are modelled and used to describe outputs and effects rather then using primitive XSD data types. For example, our execution engine can determine that an OWL concept may be mapped to an XSD array type without explicit instructions, by consulting the OWL concept.

7.2 OWL-S

During this research, OWL-S 1.1 beta superseded OWL-S 1.0. OWL-S 1.1 includes more detailed guidelines for expressing semantic rules and semantic conditions. Using the new standard, it would be

possible to extend the frameworks support for evaluating pre and post conditions, as well as other composition rules and techniques.

7.3 The Bigger Picture

Section 7.3 concludes by outlining some limitations that were identified with this research approach to solving the research objectives. Section 7.3.1 proposes some future work needed to improve inference performance and Section 7.3.2 highlights the necessary extension to the framework in order for it to scale to enterprise computing demands. Section 7.3.3 identifies some current shortcomings with ontology creation and versioning, and proposes a necessary course of action by the Semantic Web community to remedy these issues.

7.3.1 Inference Overhead

It has been identified in 6.2.1 that semantic inference comes with a severe performance overhead due to the processing requirements of XML, as well as the need to validate ontologies to ensure the concepts they represent are complete and legal. The scalability of the framework is severely hampered due to this factor.

One measure that would improve performance is to introduce some optimisation of OWL queries to minimise the processing required. Firstly, validation of ontologies every time they are initialised in memory needs to be irradiated. It is not an ideal situation to allow developers load and use ontologies that are not known to be legal, because it is one guaranteed way to introduce application bugs into a system. However, if performance is critical, this trade off may be practical.

To ensure ontologies are legal, a publishing and versioning component could be introduced into this framework. This step would ensure that the integrity of a knowledge base is legal and correct at all times. Data can be kept consistent by enforcing cardinality restrictions and other OWL concepts that enforce usage constraints. This can be done by employing object wrappers or database constraints techniques. We proposed that databases are an ideal method for storing ontologies due to their inherent support for transactions, security, indexing optimisation and proven ability to scale to enterprise requirements.

For applications that connect to large databases and/or ontologies, it will not be feasible to load the entire set of available information into working memory. Performance tests in Chapter 6 support this theory. Although extra RAM, processing power and bandwidth may improve performance, this solution is flawed as it does not address the core reasons for poor performance, which is XML string processing and semantic evaluation on ontologies at initialisation time. Due to the level of XML string

processing involved with XSLT methods for inference it is unlikely that this method can be used in large scale semantic applications.

It is necessary to target a query specifically to the source for appropriate information as it is needed. In addition, the task of any query optimiser is to not only to optimise the retrieval of information from ontology sources, but also coordinate queries that span multiple sources [39]. SNOBASE [39] by IBM is one initiative to address the problem of integrating and scaling ontologies into enterprise applications. IBM claim that the greatest barrier to more wide spread use of ontologies for storing meta-information is the lack of support in the currently available middle-ware stacks used in business applications [39]. SNOBASE is an API that provides optimisation techniques to rapidly enhance the speed of semantic inferences.

This is interesting with regards to OWL individuals. OWL individuals can be thought of as instances of OWL classes. Individuals contain values and data that represent real instances of OWL concepts. Using the E-Commerce scenario in this Dissertation, it is easy to imagine a case where instance data relating to an OWL individual needs to be updated. For example, after a purchase operation, the amount of a particular instance of Wine may need to be decremented. A flat file structure is not an ideal way to manage this kind of operation, due to potential problems with concurrency and random access of files. Databases, on the other hand provide locking mechanisms to ensure data integrity. Databases are a proven technology that scales to enterprise computing demands.

7.3.2 Internet Scalability

This solution proposed in this research to address our objectives should be regarded as a technology showcase, rather than an Internet scale application. There are some notable issues that would limit its usage in an enterprise scale. For example, ontologies are stored in flat file format. This approach is suitable when the ontologies data is being modified iteratively during development but flat files lack the ability to support usage by several clients concurrently. Using an enterprise grade database would improve this issue.

To improve performance and scalability, it is recommended that a distributed component architecture such as the J2EE standard be used to implement a semantic framework such as the one described in this research. The Agent components used in this scenario could be migrated to EJB Session Beans, where as Ontology access and persistence could be mapped to EJB Entity Beans. Integration with database back-ends could be achieved to allow for query optimisation.

7.3.3 Semantic Tools

We used the Jena API to enable in-memory initialisation and manipulation of OWL ontologies at runtime. Jena is a comprehensive and mature API for developing semantic applications, but we believe that it is not suitable for use inside a reasoner component, unless it's functionality is extended to allow for query optimisation, outlined in Section 7.3.1.

The OWL-S API used during this research will, when completed provide developers with a suitable, robust API for use with OWL-S applications. The API provides functionality to allow parsing and invocation of OWL-S processes.

During Ontology creation, we identified a short-coming in current ontology mark up tools like Protégé [59] and OILED [60]. This limitation is being addressed with OWL-S plug-in development for these ontology creation environments. Also, tools like WSDL2OWLS [61] which can create OWL-S ontology skeletons from WSDL files will ease the burden on ontology providers. We propose that Protégé should incorporate a WSDL2OWLS tool to aid ontology providers with expressing Profile Model concepts and also with incorporating complex ontological descriptions in to OWL-S instances.

7.4 Final Remarks

OWL-S offers a rich and expressive framework for adding semantics to Web Services. Its full potential has not yet been realised. As OWL-S matures, semantic frameworks will be easier to design and implement, due to improvements in development tools, advances in Web Service computing and alignment of current research domains referenced in Section 3.1. We believe traditional methods of software development will eventually incorporate methods to semantically annotate software at development time to reduce human involvement in the annotation of ontologies and semantics.

When the Internet exploded at the turn of the nineties, it was impossible to imagine the impact and benefits it brought to society. We believe that this will also be true for the Semantic Web. Our research has shown that enriched applications[21] can be built using semantic technologies and specifically OWL-S, but it is obvious that there is still more work to be done[22].

Our research also identified performance issues relating to semantic inference[23]. We have proposed employing query optimisation techniques using databases to remedy inference performance[24]. We have

[21] Discussed in Section 4.2

[22] Discussed in Section 7.3

[23] Discussed in Section 6.2.1

[24] Discussed in Section 7.3.1

also demonstrated how OWL-S can be used to enable Agents infer and understand the semantic similarities between disparate Web Services and how reasoners can be used to automate this.

With regards to our research effort, further work includes investigating A.I. planning techniques to aid in service composition. With A.I. planning, our framework can be run inside a planning engine, perhaps using a JESS shell or developing custom JESS functions to extend the API's functionality, similar to the APeLS reasoner profiled in Section 6.3.

We believe that with research into OWL-S based service composition planning, security and transaction monitoring, coupled with more powerful reasoning capabilities incorporating OWL rule based mark-up languages, semantic frameworks will enable new and improved models for Ad-hoc E-Commerce transactions and underpin the next generation of Internet based E-Commerce B2C Applications.

Chapter 8

APPENDIX

8 Appendix

This chapter documents how the Java web application was built using the framework. It also details a method for controlling the Agent operations using HTTP parameters, which is used in the demo application.

8.1 Model – View – Controller

The Model-View-Controller pattern [47] is applied to the Web Application. This pattern allows for the clean separation of the user interface and the underlying sub-system. The View in this application relates to the user interface that a user interacts with. The Controller is a business logic component that controls the process flow of the application and the Model is the underlying data representation of the application. The framework is analogous to the Model, the Agent is analogous to the Controller and the Web Pages are the View component.

8.1.1 Web Layer

The JSP pages render the content produced by the Agent component. The JSP is the View component in this architecture. A HTTP Servlet [30] is used to allow a user to interact with the Agent via the JSP Web Pages. The Servlet invokes the Agents Lifecycle methods.

8.1.2 Agent Control Layer

Figure 8.1.1 is a class diagram detailing the relationship between the Servlet Web Layer and the Agent subsystem. The Agent acts as a Mediator [31] to the Discovery, Candidate Selection, Reasoner, Composition and Invocation functionality. The Agent decouples clients (in this case the Servlet) from knowing the specifics of the subcomponents and abstracts the underlying technologies the Agent uses to achieve its goals. The Agent is stateless and when state is needed, this is passed into the Agent after initialisation in the form of the XML user profile, stored in this case inside the Servlet HTTP Session.

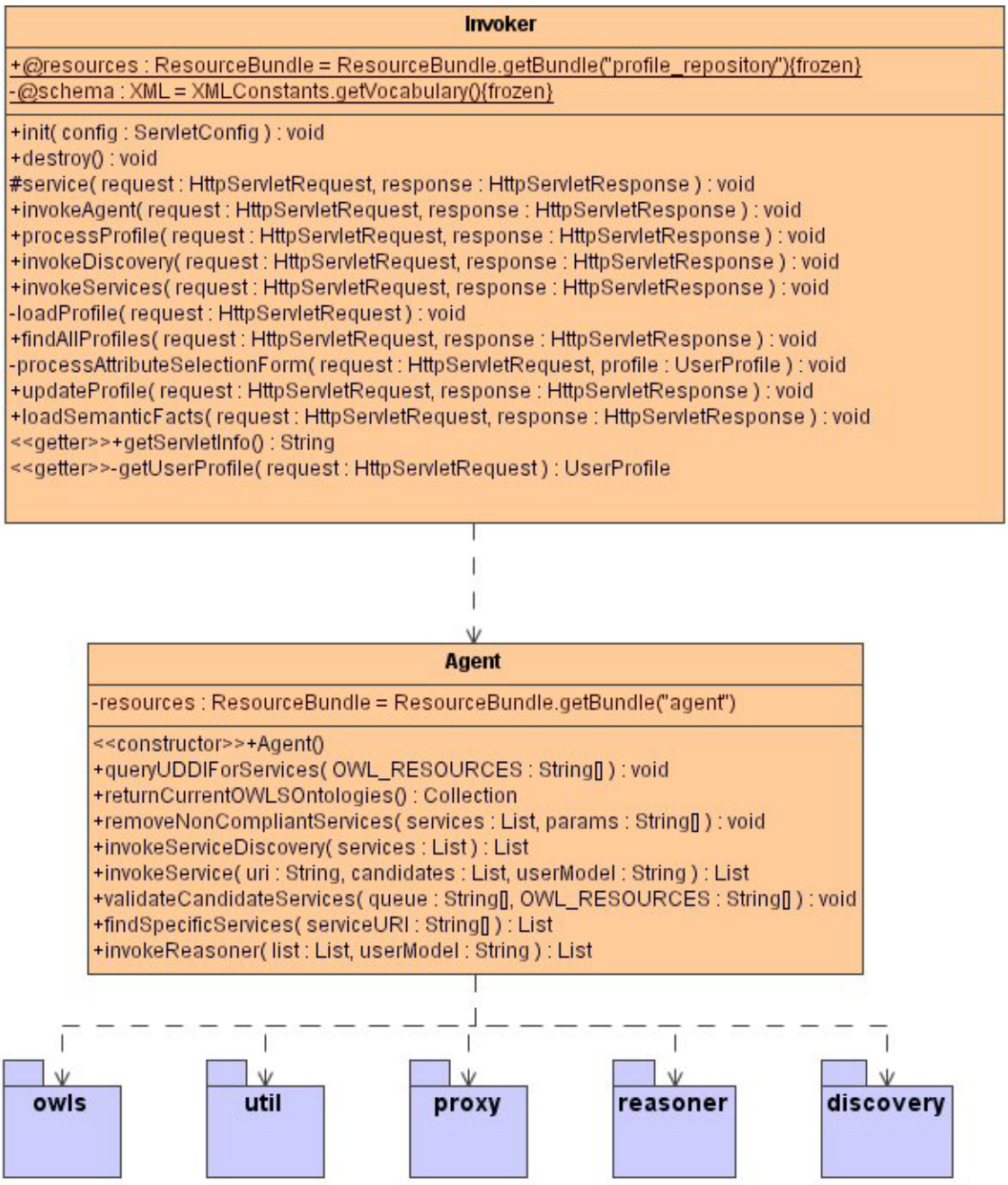

Figure 8.1.1: class diagram sowing the Agents relationship to the client Servlet and sub system.

Figure 8.1.1 shows the Invoker Servlet class is dependent on the Agents interface but is not aware of the subsystems specifics. The Agent mediates object communication between client and sub system.

8.2 HTTP Framework Control Parameters

The Servlet exposes the generic "service" method defined in the `javax.servlet.GenericServlet` super class. The implementation of the service method

examines the HTTP Request object for a “page” and “action” HTTP parameter. These parameters should be passed into the Servlet as “hidden” input parameters.

8.2.1 HTTP Redirect Parameter

The “page” parameter tells the Dispatcher Servlet which JSP page is responsible for processing the output of this request. The Servlet uses the Request Dispatcher controller provided by the Java Web Container to redirect requests.

8.2.2 HTTP Action Parameter

The “action” parameter tells the Dispatcher Servlet what Servlet operation to invoke. The Java reflection API is used to invoke the named method on the Servlet. This operation will typically carry out some sort of processing, resulting in an XML document which can be rendered by the JSP page specified by the “page” parameter. It is assumed that the operation exists otherwise a `java.lang.NoSuchMethod` Exception is raised.

8.3 E-Commerce System

The E-Store System stores a collection of product items (wine, cheese or cameras) in a relational database, and maps fictional vendors to certain items. The resulting data model is basically a set of virtual e-sellers with warehouses of tangible products, but of course fictional stock levels. A client program to this Web Service can purchase goods from the Warehouse by invoking SOAP messages on the service. There is no financial transactions taking place as this is only a demonstration, therefore security is not a concern.

Bibliography

[1] UDDI. The UDDI Technical White Paper. http://www.uddi.org/, 2000.

[2] W3C. Extensible mark-up language (xml) 1.0 (second edition). http://www.w3.org/TR/2000/REC-xml-20001006, 2000.

[3] W3C. Soap version 1.2, w3c working draft 17 December 2001. http://www.w3.org/TR/2001/WD-soap12-part0-20011217/, 2001.

[4] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1.

[5] IBM Corporation. Web Services Introduction. Published with IEEK Technology Toolkit. 2003

[6] The OWL Services Coalition. OWL-S, Semantic Mark-up for Web Services, Version 1.0, http://www.daml.org/services/owl-s/

[7] W3C. The Semantic Web, http://www.w3.org/2001/sw/.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. Scientic American, 284(5):34{43, May 2001.

[9] Massimo Paolucci et al. Importing the Semantic Web in UDDI, In Proceedings of E-Services and the Semantic Web Workshop, 2002

[10] Erven Sirin, James Hendler and Bijan Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions, In Web Services: Modelling, Architecture, and Infrastructure workshop in ICEIS 2003, Angers, France, April 2003.

[11] Brian McBride. Four Steps Towards the Widespread Adoption of a Semantic Web, In Proceedings of the First International Semantic Web Conference on The Semantic Web, Pages 419 – 422, 2002.

[12] T. R. Gruber. A translation approach to portable ontologies. Knowledge Acquisition, 5(2):199-220, 1993.

[13] W3C. RDF/XML Syntax Specification, http://www.w3.org/TR/rdf-syntax-grammar/

[14] W3C. OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/

[15] Semantic Web Services Initiative Architecture Committee (SWSA). SWSA Roadmap http://www.daml.org/services/swsa/

[16] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weer- awarana. Business Process Execution Language for Web Services, Version 1.0, July 2001. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[17] W.M.P van der Aalst et al. Advanced Workflow Patterns, In Cooperative Information Systems (CoopIS) , Edinburgh, 2000

[18] Alan H. Karp. E-Speak E-xplained, Hewlett-Packard Laboratories, Palo Alto, California

[19] Christopher Gibson and Chris Sharp. ARIES: Assured Reputation Information Exchanged Securely, IBM UK Laboratories. 2003

[20] Jonathan Dale and Luigi Ceccaroni. Pizza and a Movie: A Case Study in Advanced Web Services, In Agencities: Challenges in Open Agent Environments Workshop, Autonomous Agents and Multi-Agents Systems Conference 2002, Bologna, Italy, July 2002.

[21] Dan Wu et al. Automating DAML-S Web Services Composition Using SHOP2, In Proceedings of 2nd International Semantic Web Conference (ISWC2003), Sanbiel Island, Florida, October 2003.

[22] Naveen Srinivasan. Matchmaker DAML-S UDDI Client, http://www.daml.ri.cmu.edu/matchmaker/.

[23] Darpa Organisation, Daml Dining Demo, http://orl04.drc.com/damldining/find.asp.

[24] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider. The Description Logic Handbook, Theory, Implementation and Applications, January 2003.

[25] W3C. Naming and Addressing: URIs, URLs, http://www.w3.org/Addressing/

[26] W3C. Uniform Resource Identifiers (URI): Generic Syntax, http://www.ietf.org/rfc/rfc2396.txt

[27] W3C. Hypertext Transfer Protocol -- HTTP/1.1, http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf.

[28] Jonathan B. Postel. Simple Mail Transfer Protocol, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, August 1982.

[29] Sun Microsystems. Java Server Pages Technology, http://java.sun.com/products/jsp/.

[30] Sun Microsystems. Java Servlet Technology, http://java.sun.com/products/servlet/.

[31] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns, 21st Printing, November 2000.

[32] Apache Group. JUDDI UDDI registry, http://ws.apache.org/juddi/.

[33] Apache Group. Xindice XML database, http://xml.apache.org/xindice/.

[34] Network World Fusion. XML appliances speed Web services, http://www.nwfusion.com/news/tech/2003/1208techupdate.html.

[35] W3C. The Extensible Stylesheet Language Family (XSL), http://www.w3.org/Style/XSL/.

[36] OMG. Common Object Request Broker Architecture: Core Specification, http://www.omg.org/docs/formal/04-03-01.pdf.

[37] Sun Microsystems. Java Remote Method Invocation (Java RMI), http://java.sun.com/products/jdk/rmi/.

[38] W3C. Document Object Model (DOM), http://www.w3.org/DOM/.

[39] Juhnyoung Lee, Richard Goodwin, Yiming Ye, Rama Akkiraju. Towards Enterprise-Scale Ontology Management, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, Yet To Be Published.

[40] Conceptual Dependency. A Pocket Guide To Conceptual Dependency, http://www.cc.gatech.edu/computing/classes/cs3361_96_spring/Fall95/Notes/cd.html.

[41] Bertrand Meyer, Object-oriented software construction (2nd ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, 1997

[42] OMG. OMG IDL: Details, http://www.omg.org/gettingstarted/omg_idl.htm.

[43] DAML Services Coalition. OWL-S 1.1 Beta Release, http://www.daml.org/services/owl-s/1.1B/.

[44] W3C. XML Schema, http://www.w3.org/XML/Schema.

[45] Naveen Srinivasan. Matchmaker OWL-S 1.0 2 UDDI, http://projects.semwebcentral.org/frs/?group_id=31&release_id=49.

[46] Darpa Organisation. SWSA Use Cases, http://www.daml.org/services/use-cases/architecture/.

[47] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in small talk-80. Journal of Object Orientated Programming. 1(3):26-49, August/September 1988.

[48] Rama Akkiraju, Richard Goodwin, Prashant Doshi, Sascha Roeder. A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, 2004. Yet To Be Published

[49] Prashant Doshi, Richard Goodwin, Rama Akkiraju and Sascha Roeder. Extending Semantic Matching for Application in Business Process Integration, Dept. of Computer Science, Univ. of Illinois, 851 S. Morgan St, Chicago, IL 60607, 2004.

[50] Mindswap Java API, http://www.mindswap.org/2004/owl-s/index.shtml.

[51] Liang-Jie Zhang & Qun Zhou.Aggregate UDDI Searches with Business Explorer for Web Services, IBM Developer Works, March 2002

[52] Declan Dagger, Owen Conlan, Vincent P. Wade. An Architecture for Candidacy in Adaptive eLearning Systems to Facilitate the Reuse of Learning Resources, Trinity College Dublin.

[53] JESS, http://herzberg.ca.sandia.gov/jess/.

[54] Hewlett-Packard Development Company. Jena, http://jena.sourceforge.net/.

[55] Sun Microsystems. Java Beans, http://java.sun.com/products/javabeans/.

[56] JDOM. JDOM API, http://www.jdom.org/.

[57] W3C. RDQL – A query language for RDF, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[58] MySQL Database. http://dev.mysql.com/

[59] Protégé Ontology Editor. Stanford Medical Informatics, http://protege.stanford.edu/.

[60] OilEd Ontology Editor. http://oiled.man.ac.uk.

[61] Massimo Paolucci & Naveen Srinivasan.WSDL2OWL, http://projects.semwebcentral.org/projects/wsdl2owl-s/.

[62] University of Maryland. SHOP2, http://www.cs.umd.edu/projects/shop/.

[63] Andreas Eberhart. Ad-hoc Invocation of Semantic Web Services, In Proceedings of the IEEE International Conference on Web Services (ICWS'04) – Volume 00, June 2004.

Abbreviations

A.I.	Artificial Intelligence
APeLS	Adaptive Personalised E-Learning Service
API	Application Programming Interface
ARIES	Assured Reputation Information Exchanged Securely
B2C	Business to Consumer
BE4WS	Business Exploration for Web Service
BPEL4WS	Business Process Execution Language for Web Services
CPU	Central Processing Unit
DAML	DARPA Agent mark-up Language
DAML-S	DARPA Agent Mark-up Language for Services
DARPA	Defense Advanced Research Projects Agency
DOM	Document Object Model
EJB	Enterprise Java Beans
HTTP	Hyper Text Transfer Protocol
JESS	Java Expert System Shell
JDOM	Java Document Object Model
JSP	Java Server Pages
J2EE	Java 2 Enterprise Edition
KDEG	Knowledge & Data Engineering Group
IBM	International Business Machines
IOPE	Input Output Precondition Effect
OIL	Ontology Inference Layer
OO	Object Orientated
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
RAM	Random Access Memory
RDF	Resource Description Framework
RDQL	Resource Description Query Language
SMTP	Simple Mail Transfer Protocol
UDDI	Universal, Description, Discovery and Integration
UML	Unified Modelling Language
URI	Uniform Resource Identifier
WSDF	Web Service Description Framework

WSDL	Web Service Description Language
WSFL	Web Service Flow Language
W3C	World Wide Web Consortium
XML	Extensible Mark-up Language
XPATH	Extensible Mark-up Path Language
XSL	Extensible Style Sheet Language
XSLT	Extensible Style Sheet Language for Transformation
XSL-FO	XSL Formatting Objects