Abstract

BILGIN, AHMET SOYDAN. Semantic Web Services Query and Manipulation Language for Quality Attributes of Web Services (Under the direction of Munindar P. Singh).

The Web is moving toward a collection of interoperating Web services. Achieving this interoperability requires dynamic discovery of Web services on the bases of their capability. The capability of a service can be best determined by its functional description attributes, which describe the service interface such as input and output of the service, and quality attributes, which provide additional information to evaluate the service.

This thesis defines an approach where these quality attributes can be advertised and queried by using DAML as the query, ontology, and service description language. As a core part of this system, we modify and extend an existing DAML query language. We develop this system as an extension of current Web service registries so our system will be fully synchronized with any available Web service registry.

SEMANTIC WEB SERVICES QUERY AND MANIPULATION LANGUAGE FOR QUALITY ATTRIBUTES OF WEB SERVICES

BY

AHMET SOYDAN BILGIN

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF

NORTH CAROLINA STATE UNIVERSITY

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH

AUGUST 2003

APPROVED BY:

PETER R. WURMAN

PENG NING

MUNINDAR P. SINGH

CHAIR OF ADVISORY COMMITTEE

Biography

Ahmet Soydan Bilgin was born on September 4th, 1978 in KDZ Eregli, Turkey. He got his Bachelor of Engineering degree in Computer Engineering from Bilkent University, Ankara in 2000. He was a masters student in North Carolina State University in the Department of Computer Science from August 2001 to August 2003.

Acknowledgement

I would like to thank my advisor Dr. Munindar P. Singh for his constant support, guidance, and patience. I would also like to thank my committe members Dr. Peter R. Wurman and Dr. Peng Ning. I would like to thank Mike for all the help and for those discussions on various topic. I'm also grateful to Pinar for her patience while answering my questions. I thank Ashok, Ergun, Amit, and Yathi for their support and friendship. Finally I am thankful to my family for always being there whenever things get rough.

This research was partially supported by the National Science Foundation under grant ITR-0081742.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web services are modular and self-describing applications, which can either be composed with other Web services to create businesses or simply deployed as reusable components to provide some functionality to their clients. An important characteristics of a Web service is that it is a black box as far as any client agent (human or artificial) is concerned. In this respect, service interface-based discovery including the matching of service inputs and outputs seem to be a good starting point in service discovery process [Paolucci et al., 2002a]. On the other hand, most of the time, discovering a service based on its outer skin may not give the desirable match; because of the service's inner underlying decisions and workings that can affect the service's quality. For example, a *DocToPdf Converter* Web service may take a *.doc* file as input and produce a *.pdf* file as output. However, this information will not be enough for a user who wants all the hyperlinks or the characters written in *Tahoma* format in his document to be converted properly.

The past behavior of the Web service also plays an important role in discovering the right service [Maximilien and Singh, 2002]. Service consumers can evaluate the past behavior of a service by using attributes from different domains such as security, reputation, broadness of feature set, price, location, and developer support. A service provider can

also advertise some of these attributes. Some well-known sites such as *bindingpoint.com* [bindingpoint.com, 2001] and *salcentral.com* [salcentral.com, 2001] already provide an interface to rank Web services.

Standards such as UDDI [UDDI, 2000] and ebXML [OASIS, 2001] address the problem of Web service discovery. In a UDDI registry, Web services are published and discovered through four core data structures (businessEntity, businessService, bindingTemplate, and tModel) and their properties, which we review in Chapter 2.

Briefly, UDDI is an XML-based standard. XML-based standards represent syntactic properties of services. Human intervention is necessary to correctly interpret the intended meaning of syntactic properties. There is an important difference in the perspectives taken by syntactic and semantic representations. Syntactic representation is about the number of elements of certain types, their attributes and ordering, and the sorts of text that can appear in datatypes, while semantic representation is about things in the world that are expressed in shared ontologies—people who have names, car rental service who has rating attribute, and so on. Semantic representation provides a way for formal interpretation of intended meaning.

UDDI cannot represent quality attributes of Web services. Our first task in this thesis is to provide a way in which this information is represented as a supplementary Web service to current UDDI registries. We would like to represent Web services according to their category and quality attributes in unambiguous and computer interpretable form. To achieve this, we need to use Semantic Web standards.

The Semantic Web requires that the data be not only machine readable, but also machine understandable [Lee et al., 2001]. This is important for us, because the purpose of our work is to extend Web service descriptions by including several quality attributes so that artificial client agents may use an expressive and flexible service querying mechanism to

discover the right service according to their criteria. Our system has an ontology of service categories and quality attributes pertaining to each service category. The main requirements of our ontology for Semantic Web service description are:

1. The advertiser (service provider) must have freedom to add new service categories and new quality attributes. Different advertisers may want to describe their services with different degrees of complexity and completeness.

2. We need support for the reasoning of subsumption and equivalence properties of both service categories and quality attributes. Each service category has its own specific attributes. In addition to this, the child service category (e.g., `CarRental`) inherits its parent service category attributes (e.g., `VehicleRental`, `Rental`, and other upper level service categories). In this scenario even attributes can have sub-super or equality relationships in the same service category. For example, `average response time` is the child attribute of the `response time` attribute. Also `cost` and `fee` are equivalent attributes.

3. Each service is represented with its quality attribute values provided by the service providers and service consumers.

Another relevant effort is DAML-S [DARPA, 2003], which represents the subset of the quality attributes as non-functional properties of a service. A recent study mentions these attributes as qualitative service attributes that provide information about the service provider, the intended service customers, geographic availability, and so on [Peer, 2003].

However, simply having the above ontology is not enough to accomplish our overall goal of advertising and querying quality attributes. What we really need is a query language that Web service providers and Web service consumers can use to query resources in the schema and the data, or insert resources into the schema and the data. In this respect,

the main contribution of our work is the implementation of a flexible and simple ontology query language, which uses the existing ontology language primitives, and may be used to query our ontology and the Web service data based on this ontology.

DAML is one of the most powerful Semantic Web standards, which we review in Chapter 2. DAML provides representation for semi-structured knowledge objects with machine-processable data. It provides modeling primitives commonly found in frame-based languages while its formal semantics is defined in description logic. It is also the starting point for the Ontology Web Language [van Harmelen et al., 2003], which has been just released as a draft in March 2003. Because of the existence of DAML-specific tools, we have used DAML instead of OWL for describing queries and ontologies. Our work can be easily migrated to OWL as its popularity increases in terms of tool support.

The main aspects of our work are:

1. Adopting DAML as a query description language.

2. The specification of a DAML query language that enables SQL-like features such as selection and insertion of DAML resources.

3. The specification of a DAML query language that can be used independent of the way data is stored. For example, the name of DAML instances is just the DAML language detail, so these names may or may not be stored.

4. Our specific purpose is to express a Semantic Web services Query and Manipulation Language that can be used to advertise and query quality attributes of Web services. This purpose provides a sample usage of the proposed DAML query language.

This thesis extends the DAML Query Language, which was proposed by deVos [deVos, 2002]. deVos proposes a DAML query language that has a simple basis but can handle

very general queries by incorporating DAML class expressions. His thinking of "What do I add to DAML to create a query" results in a general DAML query language that uses all the primitives of DAML itself. We first explain some of the insufficiencies of this language and then propose the necessary extensions and modifications to be able to query our ontology schema and the data stored in a relational database. We define a mapping between the relational database table structure and DAML classes, so those special agents use this mapping information to query the database as if the data were converted to DAML instances. We also insert new classes and properties to the existing ontology and database. We implement the basic properties of the proposed DAML query language by using existing DAML-specific tools such as DAMLJessKB for parsing and reasoning and the Jena DAML API for creating DAML classes and properties.

Thesis Organization This thesis outlines the development of a query language based on DAML, which queries and inserts Web service (meta)data related to the quality attributes of a service. The remainder of the thesis is organized as follows: Chapter 2 gives background information on Web service description standards and Semantic Web standards. It also presents an overview of the DAML Query Language proposed by deVos. Chapter 3 briefly explains service categories and quality attributes ontology with a sample service instance. Chapter 4 presents the proposed extensions for this query language to support querying and inserting service (meta)data. Chapter 5 explains the implementation details of our querying system. Chapter 6 explains the related work and some further research areas.

Chapter 2

Background

This chapter gives brief information about the current Web service description standards and Semantic Web standards. We mention DAML-S and OWL-S as standards, which achieve the common purposes of both Web service description and Semantic Web standards. We also give detailed information about the current DAML query language, which was proposed by deVos.

2.1 Web Service Description Standards

Service description is a broad term subject to different interpretations. For example, WSDL descriptions focus on the behavioral aspects of a service. UDDI (Universal Description, Discovery and Integration of Business for the Web) descriptions are based on three different types of information: contact details, classification with respect to a certain taxonomy, and technical information. E-Speak, DAML-S, and OWL-S are standards that provide functionality of both UDDI and WSDL. DAML-S and OWL-S supports the semantic description of Web services.

2.1.1 UDDI (Universal Description, Discovery, and Integration of Web Services)

Universal Description, Discovery and Integration (UDDI) [UDDI, 2000] was jointly proposed as a standard by IBM, Microsoft, and Ariba. Having being proposed by leading vendors made more readily accepted as a standard. UDDI uses a service registry architecture that presents a standard way for businesses to register their Web services and to discover other Web services. Conceptually, the information provided in UDDI registries consists of three components: *white pages* of company contact information; *yellow pages* that categorize businesses by standard taxonomies; and *green pages* that provide the technical information about services by referencing the WSDL documents.

UDDI describes businesses and services by their physical attributes such as name and address. Services are also associated with a set of attributes called tModels. tModels can be associated with description standards such as WSDL to provide green page functionality, or with taxonomies such as NAICS (North American Industry Classification System) [Bureau, 1997] to provide yellow page functionality.

UDDI doesn't represent service capabilities and doesn't take into account quality of service attributes, so it is of no help to search for services on the basis of what they provide and what qualities they have. For example, UDDI doesn't support a query such as *Find a car rental service that has good rating, is available 24 hours a day, and doesn't charge an extra fee for drivers under 25 years of age*.

Figure 2.1 is a sample business entity which categorizes itself as a car rental service provider located in California by using NAICS and ISO 3166 Geographic Taxonomies [ISO, 2001].

UDDI doesn't provide any semantics for service definitions [Ankolekar et al., 2002].

```
(1)<?xml version="1.0" encoding="utf-8"?>
(2)<businessEntity businessKey="…" xmlns="urn:uddi-org:api_v2">
  (3) <name>Company</name>
  (4) <categoryBag>
      (5) <keyedReference
          (6) tModelKey="uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
          (7) keyName="NAICS: Car Rental"
          (8) keyValue="51121" />
      (9) <keyedReference
          (10) tModelKey="uuid:4e49a8d6-d5a2-4fc2-93a0-0411d8d19e88"
          (11) keyName="California"
          (12) keyValue="US-CA" />
  (13) </categoryBag>
(14)</businessEntity>
```

Figure 2.1: A car rental service provider

If you query the services located in the US or if you look for a vehicle rental service, the UDDI registry will not find the service of Figure 2.1 although car is a subtype of vehicle and California is inside US [Januszewski, 2002].

2.1.2 WSDL (Web Service Definition Language)

WSDL [Christensen et al., 2001] is an XML format, closely associated with UDDI as the language for describing interfaces to business services registered within a UDDI database. WSDL is a kind of content language for advertisement of Web service functional description attributes (i.e., inputs and outputs of the service operations). The green page functionality of UDDI is achieved by providing link to WSDL of the Web service.

Like UDDI, WSDL does not support semantic description of services [Ankolekar et al., 2002].

2.1.3 E-Speak

E-Speak [Apte and Mehta, 2001] [Ankolekar et al., 2002] is the E-services (electronic services) platform developed by HP for the creation of dynamic, intelligent interaction of E-services. E-Speak provides for discovery, negotiation, management, security, and

utilization of E-services. Although E-Speak was not widely accepted commercially, it was the first platform that provided an attribute framework, which allowed for a robust description of services. For example, when talking about a car, it would not be enough to say that the car is an Audi. You would want to know the color, the engine power, the number of doors, whether it has a sunroof, and numerous other characteristics.

E-Speak provides the vocabulary architecture to create an attribute framework to describe services. E-Speak and UDDI have similar goals in that they both facilitate the advertisement and discovery of services. E-Speak like WSDL supports the description of service and data types.

E-Speak Vocabulary An E-Speak vocabulary can be thought of as a real-world vocabulary. A vocabulary is a set of words that can be used to describe a service. This type of service description lends itself to forming structured queries to find those services within a certain category.

A complete E-Speak vocabulary specification consists of property names and the type of values they have associated with them. A vocabulary is required to find a service. The E-Speak vocabulary doesn't make a distinction between functional description and quality attributes of a service. There is a default base vocabulary, which defines a set of generic attributes such as name, type, description, keywords, and version. If we choose to use anything other the above generic attributes, we will need to do some work to create a user-defined vocabulary. As an example, for the car rental domain, we can include attributes related to range of cars, legal restrictions for renting a car, and extra fees. This vocabulary will allow a client to form structured queries when trying to find an appropriate service.

Car rental companies that decide to deploy their Web services can use the above vocabulary by providing values for these attributes. Whenever clients want to rent a car, they can

search for the appropriate car rental service by specifying their search criteria including the same set of attributes. E-Speak provides several classes to support the creation, registration, and use of vocabularies.

Currently, no semantics is attached to any of the attributes defined in the E-Speak vocabularies. Service requests and advertisements are matched over the service description attributes, which do not distinguish between further subtypes. For example, a service can be advertised with attributes *Response Time = 10* and *sells = Computer*. A service consumer can query for a service that has *Average Response Time* < 15 or for a service that *sells = Desktop*. Although *Desktop* is a subtype of *Computer* and *Average Response Time* is a subproperty of *Response Time*, neither query will return a result.

2.2 Semantic Web Standards and DAML-S

A limitation shared by the XML-based standards is their lack of an explicit semantics: two identical XML descriptions may mean different things depending on when and who uses them. Service advertisers and service requestors can have different perspectives and different knowledge about the same service. Syntactically the advertised service may not match with the requested service, but semantically they can match with each other.

The DARPA Agent Markup Language for Web services (DAML-S) is an ontology description language for Web services. DAML-S provides a mechanism to describe the capabilities and properties of a Web service in a machine interpretable form. OWL-S (Ontology Web Language for Web Services) is equivalent to DAML-S in terms of functionality, but is built upon OWL, which is still under development. To put DAML-S or OWL-S in perspective, it is necessary to understand their foundations: RDF, RDF Schema, DAML, and OWL.

2.2.1 RDF and RDF Schema

Resource Description Framework [W3C, 1999], as its name implies, is a framework for describing and interchanging metadata. All things described by RDF expressions are called resources. Resources are identified by unique URIs. Thus a resource may be a Web page, a person, or any other object we wish to describe. A resource is associated with zero or more properties. RDF specifies that information be represented in the form of *triples* of the form $\langle$*subject, predicate, object*$\rangle$. A triple corresponds to *Statement* in the RDF Syntax specification document. The object of a statement can be another resource or a literal value. Hence, if we want to state that *Soydan Bilgin* is the *creator* of the resource *http://vegas.csc.ncsu.edu/wsap/ServiceTypes.daml*, we would have the triple: $\langle$*http://vegas.csc.ncsu.edu/wsap/ServiceTypes.daml*, *creator*, *Soydan Bilgin*$\rangle$.

RDF's vocabulary description language, RDF Schema [W3C, 2002], is an extension of RDF. RDF Schema vocabulary consists of modeling primitives such as *Class*, *subClassOf*, *Property*, *subPropertyOf*, *domain*, and *range*. These modeling primitives provide a mechanism for describing groups of related resources and the relationships between these resources. Such a conceptualization of a domain is termed an *ontology*.

2.2.2 DAML and OWL

While RDF Schema provides us with a formalism for describing the structure of data, its descriptive power is limited for building a completely machine understandable Web. DAML [DARPA, 2001] provides more sophisticated classifications and properties of resources than RDF Schema. DAML provides properties such as equivalence via the *sameClassAs* and *samePropertyAs* primitives and uniqueness via the *UniqueProperty* primitive.

It broadens the concept of a class by providing built-in support for enumerations and allowing local restrictions on properties. DAML also adds facilities for data typing based on the type definitions provided in XML Schema Definition Language.

OWL [van Harmelen et al., 2003] has three sub-languages: *Lite*, *DL*, and *Full*. OWL-DL is the one that supports existing description logic constraints to provide computational properties for reasoning systems. OWL-DL doesn't add more descriptive power upon DAML. OWL is simply the W3C initiative version of DAML. OWL removes the synonyms in DAML (i.e., *DAML:range* is same as *RDF-Schema:range*), adds some properties and classes to support versioning, and adds *AllDifferent* class and *distinctMembers* property to address the Unique Names Assumption. The intended usage of adding AllDifferent class and distintMembers property is that all individuals in a list formed by AllDifferent class are different from each other. These constructs facilitate the statement of uniqueness of individuals, which would otherwise be captured via statements of pairwise disjointness using the *DAML:differentIndividualFrom* primitive.

2.2.3 DAML-S or OWL-S

Web mark-up languages such as DAML and OWL enable the creation of ontologies for any domain and the instantiation of these ontologies for the description of web resources. DAML-S is a specific ontology expressed in DAML, which provides a vocabulary to describe properties and capabilities of Web services in computer-interpretable form. The basic goal of DAML-S is to facilitate the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition, and execution monitoring [Ankolekar et al., 2002]. OWL-S, in essence, is same as DAML-S, but expressed in OWL.

At the root of the DAML-S ontology is the generic class *Service*, which has subclasses

ServiceProfile, *ServiceModel*, and *ServiceGrounding*. The ServiceProfile outlines the capabilities of the service. The ServiceModel models the service as a process and specifies how the service works. The ServiceGrounding specifies how the service can be accessed and its operations that can be invoked.

The profile provides the essential knowledge needed for an agent to discover a service that meets its requirements. It presents information regarding a given service provider, the functionality, and the functional attributes of a service. The inputs, outputs, preconditions, and the effects of the service represents the functionality of the service. Two examples of such functional attributes are quality rating and geographic radius. DAML-S makes a distinction between functional description attributes and functional attributes. Functional description attributes describe the interface. On the other hand functional attributes don't deal with the process that the service implements. Functional attributes provide additional information about the service and the constraints on its use.

In this respect, our definition for quality attributes also covers the definition for functional attributes. the profile can represent limited number of quality attributes such as *Quality Rating*, *Maximum Response Time*, *Average Response time*, and *Geographic Radius*. In addition to these attributes, the latest release of DAML-S specification (0.9 Beta Draft Release) [DARPA, 2003] provides profile-based class hierarchies to describe the service and its functionality by positioning the service within some domain. Profile-based class hierarchies are constructed with inheritance of properties by subclasses. Constructing class hierarchies enable the categorization of services and thus support more powerful forms of query. This specification (with profile-based class hierarchies) meets the requirements for our ontology except for the diverse range of quality attributes, as explained in Chapter 1.

DAML-S is still a draft. It provides the functionality of UDDI, WSDL and more. DAML-S and OWL-S seem to be promising solutions for the semantic description of Web

services. DAML-S provides limited number of quality attributes and simple profile-based class hierarchy. In our work, we develop an ontology to describe and categorize services by using quality attributes. Our ontology corresponds to the DAML-S profile with an emphasis on quality attributes.

Research is going on about the automated service capabilities matching and the automated service composition by using DAML-S as the service description standard. A recent study shows how to map DAML-S service profiles into UDDI records to provide a way to record semantic information within UDDI records [Paolucci et al., 2002b]. [Li and Horrocks, 2003] implements a prototype matchmaker, which uses a DL reasoner to match service advertisements and requests based on a DAML-S based ontology. They also argue that service profiles contain too much information for effective matching.

2.3 Basics of DAML Query Language

deVos proposes the DAML Query Language [deVos, 2002], which allows us to query DAML instances. A query in this language is formulated with an expression of the form `select <property expression> from <class expression>`. The query results are triples. The `from` clause, which is an expression describing a DAML class can be used for expressing complex DAML concepts. The overall goal of deVos was to build a simple query language that could be expressed in DAML and be implemented over various simple conventional (i.e., non-DAML) data sources. Even though it is simple, this query language would be sufficient for extracting statements from many data sources. Actually, the ease of use, power or lack of power of the proposed language would be that of DAML class expressions.

Figure 2.2 shows the basis of the query language as expressed in DAML. The result of

```
<Query>
        <select rdf:resource="#manager"/>
        <select rdf:resource="#name"/>
        <from rdf:resource="#Departments">
</Query>
```

Figure 2.2: Find the `name` and the `manager` of all subjects of type `Departments`

the query in Figure 2.2 is the set of all statements with a subject of type `Departments` and a predicate of `name` or `manager`.

In general, the `from` clause in the query can accept any DAML class. deVos uses `DAML:Restriction` for narrowing the search space. As a first step, consider a restriction query in Figure 2.3:

```
<Query>
    <select rdf:resource="#manager"/>
    <from>
        <daml:Restriction>
            <daml:onProperty rdf:resource="#name"/>
            <daml:hasValue>cosmetics</daml:hasValue>
        </daml:Restriction>
    </from>
</Query>
```

Figure 2.3: Find the `manager` of all subjects that has `name` = “cosmetics”

The result of the above query is the set of statements with predicate `manager` with subjects that have a property `name` equal to “cosmetics”. The object value of `DAML:hasValue` can be a literal or a reference to a resource. Multiple restrictions can be expressed by `DAML:subClassOf` expressions. The boolean combination of all these restrictions yields the projected subject. It is also possible to enumerate the instances to be queried by using `DAML:oneOf` property. Different classes can be joined by the using `DAML:hasClass` primitive as in Figure 2.4 .

It is obviously useful to reduce query results to a set of resources by nesting queries.

```
<Query>
        <select rdf:resource="#name"/>
          <from>
                <daml:Restriction>
                        <daml:onProperty rdf:resource="#address"/>
                         <daml:hasClass>
                              <daml:onProperty rdf:resource="#street"/>
                              <daml:hasValue>Avent Ferry</daml:hasValue>
                         </daml:hasClass>
                </daml:Restriction>
        </from>
</Query>
```

Figure 2.4: Find the `name` of all subjects, which has `address` property whose objects has `street` = "Avent Ferry"

Figure 2.5 shows an example:

```
<Query>
        <select rdf:resource="#name"/>
        <from>
              <SubQuery>
                    <select rdf:resource="#department"/>
                    <from rdf:resource="#Items"/>
              </SubQuery>
        </from>
</Query>
```

Figure 2.5: Find the name of all departments where some items are sold

The `SubQuery` is defined as a `Class` and a `Query`. The triples returned by the inner query form the input for the outer query.

The foregoing queries all produce homogenous results, that is, the same properties are queried for every instance in the result set. However, it is often desirable to query different properties from different sets of instances in a combined query. Moreover, it is usually desirable to batch several queries together if the query service is remote and throughput is a consideration. The example envelope in Figure 2.6 is used to combine several queries and shared class expressions:

A `Request` class contains a series of `declare` and `evaluate` terms. Each `declare` term introduces a class expression. It is expected that each declared class has a name so that it can be referenced in another class declaration or query.

Each `evaluate` term contains a query. The result of a `Request` class is the union of the results of individual queries. Figure 2.6 is an elaboration of the fictitious department store as in [deVos, 2002] that returns the name of the department manager as well as the item prices in the cosmetics department .

```
<Request>
  <!-- declarations for Departments, name, etc. omitted for brevity -->
                    <declare>
                    <daml:Class rdf:ID="CosmeticsDept">
                        <daml:subClassOf rdf:resource="#Departments"/>
                        <daml:Restriction>
                            <daml:onProperty rdf:resource="#name"/>
                            <daml:hasValue>cosmetics</daml:hasValue>
                        </daml:Restriction>
                    </daml:Class>
                </declare>
                <evaluate>
                    <Query>
                        <select rdf:resource="#manager"/>
                        <from rdf:resource="#CosmeticsDept"/>
                    </Query>
                </evaluate>
                <evaluate>
                    <Query>
                        <select rdf:resource="#price"/>
                        <from>
                            <daml:Restriction>
                                <daml:onProperty rdf:resource="#department"/>
                                <daml:hasClass rdf:resource="#CosmeticsDep"/>
                            </daml:Restriction>
                        </from>
                    </Query>
                </evaluate>
</Request>
```

Figure 2.6: Find the name of manager as well as the prices of items in the cosmetics department

Chapter 3

Service Categories and Attributes Ontology

Our system incorporates a simple ontology for service categories and service quality attributes. In this ontology each service category corresponds to a `DAML:Class` and each quality attribute corresponds to a `DAML:ObjectProperty`. A hierarchical representation of a sample ontology is shown in Figure 3.1:

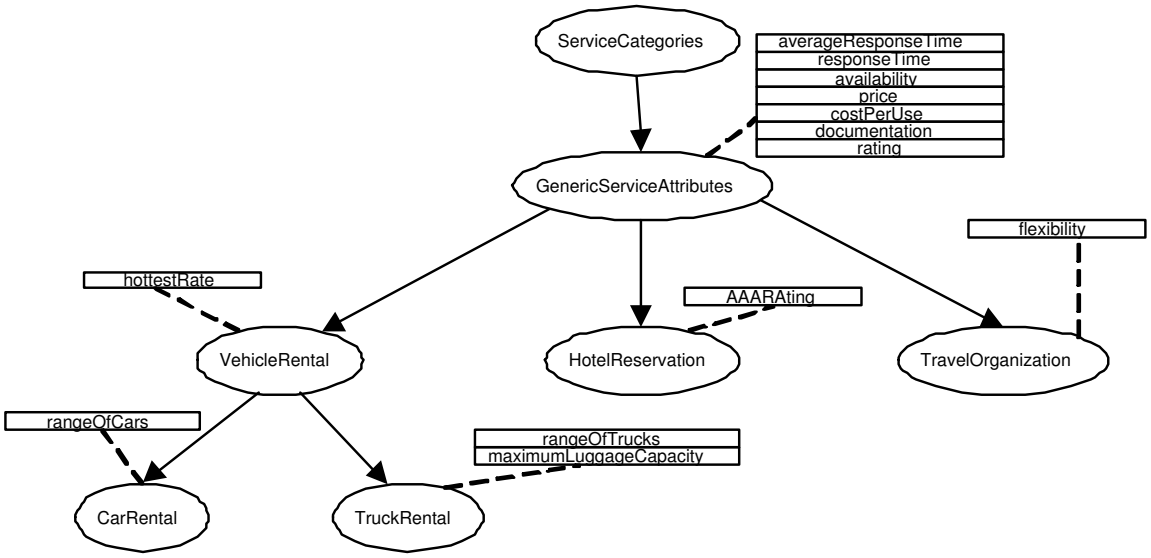

Figure 3.1: Representation of service categories and quality attributes ontology

In this representation, ellipses represent classes and each row in the rectangles represents an object property. The `ServiceCategories` is the abstract service category. The `GenericServiceAttributes` is a subclass of `ServiceCategories`. The hierarchy of service categories is constructed via `DAML:subClassOf` as shown in Figure 3.2.

```
<daml:Class rdf:ID="CarRental">
   <daml:subClassOf rdf:resource="#VehicleRental/>
</daml:Class>
```

Figure 3.2: The `CarRental` service category

Each service has one `serviceKey` (a unique identifier of the service provider by a public registry like Microsoft business UDDI Registry), one `publishedRegistryUrl` (the URL of the UDDI Registry), and one `serviceProvider` (the provider of the service and is identified by the `partyIdentifier`). In real life, a service can be published in more than one registry, but for the sake of simplicity we use just one service key of the service.

```
<daml:ObjectProperty rdf:ID="rangeOfCars">
    <rdfs:label>rangeOfCars</rdfs:label>
    <daml:domain rdf:resource="#CarRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>
```

Figure 3.3: The `rangeOfCars` attribute

Each quality attribute belongs to at least one service category. For example, `rangeOfCars` is formed as in Figure 3.3.

There can be more than one `DAML:domain` property for each service quality attribute. Quality attributes can also have `DAML:subPropertyOf` or `DAML:samePropertyAs` properties. The range of each quality attribute is `AttributeValues`. The

properties associated with `AttributeValues` are `value`, `unit`, `predicate`, `numberOfSubmission`, `lastSubmissionTime`, and `submittedBy`. According to our ontology, values of attributes can be submitted by either service providers or service consumers. Each service provider or service consumer party has a name and an identifier property.

Sample Service Instance In the proposed approach, data pertaining to a Web service is stored in a relational database. This data can be converted to DAML instances if a mapping between the structure of tables and DAML classes is defined. Chapter 5 defines and explains this mapping. After this mapping is applied, agents can query the data as if it were stored as DAML instances. Figure 3.4 describes a sample service as a DAML instance after the mapping is applied. In this representation, each service is an instance of one of the service category classes in the ontology. Each service (service category class instance) has attribute values submitted by either the service provider or service consumers. For the sake of simplicity of implementation, we assume that a service provider or a service consumer can submit at most one value for each service attribute. This assumption proves to be important while inserting or modifying service data. Also resources of type `AttributeValues` class that don't have any `submittedBy` property are assumed to be submitted by the service provider

Figure 3.5 gives the abstract UML representation of the `VehicleRental` service. Each service has three main frames corresponding to three main tags in its service description. The first-level frame corresponds to a service category class instance. There can be several second-level frames, each of which corresponds to a resource of type `AttributeValues`. There can be several third-level frames, each of which corresponds to a resource of type `Party`. In Figure 3.5, we denote properties with range of

```
<ont:VehicleRental rdf:ID="vehicleRental1">
    <ont:serviceProvider rdf:resource="#p3"/>
    <ont:serviceName>Ergun_VehicleRental</ont:serviceName>
    <ont:serviceKey>159734c6-bed1-417c-b74f-066dcf3f13b5</ont:serviceKey>
    <ont:publishedRegistryUrl>http://test.uddi.microsoft.com </ont:publishedRegistryUrl>
    <ont:availability rdf:resource="#v1"/>
    <ont:availability rdf:resource="#v2"/>
    <ont:price rdf:resource="#v3"/>
    <ont:hotRate rdf:resource="#v6"/>
</ont:VehicleRental>

<ont:AttributeValues rdf:ID="v1">
    <ont:value>90</ont:value>
    <ont:unit>percentage</ont:unit>
    <ont:predicate rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#GREATER"/>
   <ont:numberOfSubmission>1</ont:numberOfSubmission>
    <ont:lastSubmissionTime>2003-05-01</ont:lastSubmissionTime>
    <ont:submittedBy rdf:resource="#p1"/>
</ont:AttributeValues>

<ont:AttributeValues rdf:ID="v2">
    <ont:value>70</ont:value>
    <ont:unit>percentage</ont:unit>
    <ont:predicate rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#GREATER"/>
    <ont:numberOfSubmission>1</ont:numberOfSubmission>
    <ont:lastSubmissionTime>2003-05-02</ont:lastSubmissionTime>
    <ont:submittedBy rdf:resource="#p2"/>
</ont:AttributeValues>

<ont:AttributeValues rdf:ID="v3">
      <ont:value>1</ont:value>
      <ont:unit>dollar</ont:unit>
      <ont:predicate      rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#EQUAL"/>
</ont:AttributeValues>

<ont:AttributeValues rdf:ID="v6">
    <ont:value>15</ont:value>
     <ont:unit>dollar</ont:unit>
    <ont:predicate rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#EQUAL"/>
</ont:AttributeValues>

<ont:ServiceConsumer rdf:ID="p1">
    <ont:partyIdentifier>1</ont:partyIdentifier>
    <ont:partyName>Munindar  Singh</ont:partyName>
</ont:ServiceConsumer>

<ont:ServiceConsumer rdf:ID="p2">
    <ont:partyIdentifier>2</ont:partyIdentifier>
    <ont:partyName>Pinar Yolum</ont:partyName>
</ont:ServiceConsumer>

<ont:ServiceProvider rdf:ID="p3">
    <ont:partyIdentifier>uuid:8993643</ont:partyIdentifier>
    <ont:partyName>Ergun Bicici</ont:partyName>
</ont:ServiceProvider>
```


Figure 3.4: A vehicle rental service

`DAML:Thing` as attributes with `Object` type.

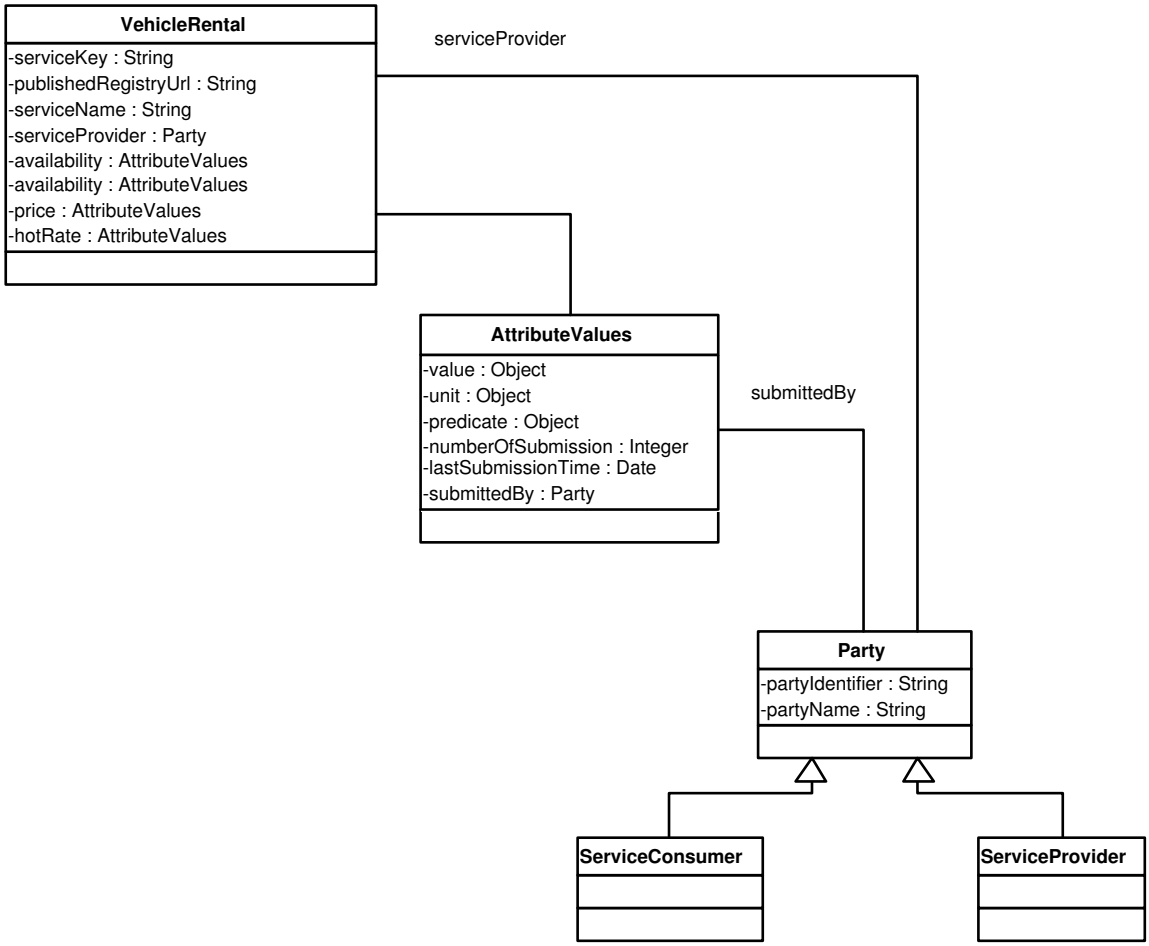

Figure 3.5: An abstract UML representation of the vehicle rental service

Chapter 4

Semantic Web Services Query and Manipulation Language

4.1 Basics of the Query Language

Our system incorporates a simple Semantic Web services Query and Manipulation Language (SWSQML), which the service providers and consumers use to query, insert, or modify both the quality attributes ontology and the service description data based on this ontology. The ontology is stored as triples in memory and the service description data is stored in a relational database. We use a relational database because of the SQL-like functionality that our language has. A major advantage of using a relational database is that it provides a scalable off-the-shelf solution. For the rest of the chapter, the metadata part of our system corresponds to our ontology and the data part corresponds to our database.

Our language should allow the following query and manipulation templates for our ontology:

1. Find the direct attributes (properties) of a specific service category.

2. Find the transitive attributes of a specific service category.

3. Find the direct subclasses of a service category (class).

4. Find the transitive subclasses of a service category (class).

5. Find the property values (e.g., domain) of an attribute (property).

6. Insert a new service category (class).

7. Insert a new attribute (property).

Our language should allow the following query and manipulation templates for the service data stored in the relational database :

1. Find the attributes where a specific service has values.

2. Find the attributes and their values for a specific service.

3. Find the services which have a specific value for a specific attribute.

4. Find the services having a given attribute.

5. Find the type of a service.

6. Find the services which are of a specific service category.

7. Find the services which have values for the attributes whose range/domain is a specific category.

8. Insert a new service.

9. Insert or modify values for an attribute of a specific service.

Each service corresponds to an instance of a service category class in our ontology. The following example shows the way service instances can be created.

⟨`DAML:Class RDF:ID= "CarRental"`⟩ represents the `CarRental` service category. To create a new car rental service, the above class has to be instantiated, e.g., as ⟨`CarRental RDF:ID= "Soydan CarRental Service"`⟩

All the data related to service instances is stored in a relational database, which can be either a DAML repository or a non-DAML repository. Our implementation depends on a non-DAML repository as explained in Chapter 5.

4.2 Motivation for Proposed Language

The DAML Query Language proposed by deVos is useful in terms of querying DAML class instances. It can handle the following two query templates well:

1. Select properties from the instances of a class having a given property.
    - Find the attributes where a specific service has values.
    - Find the services having a given attribute.
2. Select properties from the instances of a class having a property with a specific value(s).
    - Find the attributes and their values for a specific service.
    - Find the services which have a specific value for a specific attribute.

The two query templates above are the core query templates for our system. Obtaining attribute values from the services or checking whether a service has a value for the specified attributes is crucial. However the DAML query language is inadequate for meeting our

system requirements, especially the query templates listed in Section 4.1. Specifically, the following extensions are needed:

1. Queries should be at the semantic level; not only at the structure level. At the structure level, the (meta)data only consists of a set of triples. This is consistent with deVos' motivation for using this query language for non-DAML data sources without reasoners. However, we require a query language that is sensitive to the semantics of the DAML and RDF Schema primitives. SWSQML needs to elaborate subsumption and equivalence relationships between classes and properties, e.g., SWSQML has to aware of the transitivity of the subclass relation.

2. We need to answer the following queries, which use our ontology:

   - Find the subclass of `VehicleRental` service.
   - Find the properties whose range is `AttributeValues`.

   deVos' approach cannot evaluate the above queries, because it cannot query the DAML classes that have name. This situation is ideal for the data part of our system; because we don't store the name of the classes in RDBMS tables. For the metadata part, querying via the name of the class or property is crucial. For example, in Figure 4.1, we have a `DAML:Class` with name `VehicleRantal` and a DAML instance of type `VehicleRental` with name `vehicleRental1`

```
<ont:VehicleRental rdf:ID="vehicleRental1">

    *****

<daml:Class rdf:ID="VehicleRental">
  <rdfs:label>VehicleRental</rdfs:label>
  <rdfs:subClassOf rdf:resource="#GenericAttributes"
</daml:Class>
```

Figure 4.1: The `vehicleRental` class and an instance of the `vehicleRental` class

We may not build a query by using `vehicleRental1` if we store service instances in a non-DAML data source. This makes sense, but we should be able to build queries by using `VehicleRental`. Named classes are not important in instances; because they are just DAML language detail and will be transparent to the user while joining different classes by using `DAML:hasClass` primitive as explained in Chapter 2.3. if we had stored the name of the classes in a relational database, it would have been infeasible to determine while inserting new instances whether the name for a DAML class instance has been used before.

3. A mechanism to insert (meta)data expressed by DAML. This is one of the reasons we decided to use and extend the DAML Query Language. Because of its dependency on DAML primitives, it is very straightforward to insert metadata.

The four main reasons listed above made us modify and extend the existing work. The next section explains our modifications and extensions by using the queries for our ontology file (ServiceTypes.daml) and two service instances (a vehicle and a car rental service). As an example, the mapping of the vehicle rental service from relations to DAML are given in Figure 3.4. As mentioned before, service instances are stored in a relational database, but mapping from both sources is easily done.

4.3 Modification and Enhancements

4.3.1 Identification of Services

Service providers can advertise millions of services. If we abstract the system as a huge table built according to our ontology (metadata part), each service data will correspond to a tuple containing attribute values in a relational model. In this respect, each service

(class instance) should be uniquely identified. Candidates for such an identifier for each service are `serviceKey` + `publishedRegistryUrl`, which can be obtained from a real public registry (i.e., UDDI). We add an `identifier` term to our query description ontology. This term resembles the `where` term of SQL except that it can only be used for the unique identifier properties. The example in Figure 4.2 uses `serviceKey` + `publishedRegistryUrl` as a unique identifier for the service. We can use the query in Figure 4.2 to find all the details of the `availability` property of the service whose `serviceKey` is "159734c6-bed1-417c-b74f-066dcf3f13b5" and which is published in the Microsoft UDDI Test Registry.

```
<Request>
    <identifier>
      <daml:Class>
        <daml:subClassOf>
          <daml:Restriction>
            <daml:onProperty
              rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
            <daml:hasValue>159734c6-bed1-417c-b74f-066dcf3f13b5</daml:hasValue>
          </daml:Restriction>
        </daml:subClassOf>
        <daml:subClassOf>
          <daml:Restriction>
            <daml:onProperty
              rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                ServiceTypes.daml#publishedRegistryUrl"/>
            <daml:hasValue>http://test.uddi.microsoft.com</daml:hasValue>
          </daml:Restriction>
        </daml:subClassOf>
      </daml:Class>
    </identifier>
    <evaluate>
      <Query>
        <select
          rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#availability"/>
      </Query>
    </evaluate>
</Request>
```

Figure 4.2: Find all the values of a given service for the given property

4.3.2 Evaluating Class Instances

According to deVos' proposal, ⟨`from RDF:resource = "class"`⟩ is evaluated as *with a subject of type class*. This kind of structure always evaluates instances of a class (data part), but if we want to evaluate the actual class itself (metadata part), DAML Query Language has no constructs, so we add the following terms:

- **RestrictedTo:** This is subclass of `DAML:Restriction` and can have three properties: `select`, `objectType`, and `subjectType`. The `RestrictedTo` is used for the type declarations of instances.
- **objectType:** This is a property with range of `DAML:Class`. It is used to restrict the search space to triples that have object type = *class*.
- **subjectType:** This is a property with range of `DAML:Class`. It is used to restrict the search space to triples that have subject type = *class*.

To find the `serviceKey` and `availability` of vehicle rental services, our query should be:

```
<Query>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#availability"/>

      <from>
          <RestrictedTo>
            <subjectType
                rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#VehicleRental" />
          </RestrictedTo>
      </from>
</Query>
```

Figure 4.3: Find the properties of a service with the given type

The query in Figure 4.4 finds the `partyIdentifier` of all parties that are either service consumer or service provider:

```
<Query>

        <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#partyIdentifier"/>
        <from>
          <RestrictedTo>
            <subjectType>
              <daml:Class>
                <daml:oneOf rdf:parseType="daml:collection">
                <daml:Class
                    rdf:about="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#ServiceConsumer"/>
                <daml:Class
                   rdf:about="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#ServiceProvider"/>
                </daml:oneOf>
              </daml:Class>
            </subjectType>
          </RestrictedTo>
        </from>

</Query>
```

Figure 4.4: Find the property of parties with given types

`RestrictedTo` inherits all the properties of `DAML:Restriction` (i.e., `DAML:onProperty`). The query in Figure 4.5 finds the `serviceKey` of all car rantal services that have `rangeOfCars` property with value equal to "Average" (Possible rating values are defined in QualityRating ontology).

If we do not need to query according to the type of the class, then `RestrictedTo` should not be used. The query in Figure 4.6 returns the `partyIdentifier` and `partyName` of all parties who submitted something to the given service.

```
<Query>
        <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
        <from>
            <RestrictedTo>
                <subjectType
                    rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#CarRental" />
                <daml:onProperty
                    rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#rangeOfCars"/>
                <daml:hasClass>
                    <daml:Restriction>
                      <daml:onProperty
                          rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#value"/>
                      <daml:hasValue
                          rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/QualityRating.daml#Average"/>
                    </daml:Restriction>
                </daml:hasClass>
            </RestrictedTo>
        </from>
</Query>
```

Figure 4.5: Find the property of services with the given type and the value for the given property

```
<Request>
   <identifier>…</identifier>
   <evaluate>
       <Query>
           <select
               rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#partyIdentifier"/>
           <select
               rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#partyName"/>
           <from>
                   <daml:Restriction>
                       <daml:onProperty
                           rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                               ServiceTypes.daml#submittedBy"/>
                   </daml:Restriction>
           </from>
       </Query>
  </evaluate>
</Request>
```

Figure 4.6: Find the properties of parties who submitted something to a given service

4.3.3 Querying Properties

One of the drawbacks of deVos' work is its inability to query metadata. To rectify this problem, we first added `RestrictedTo` to get rid of the ambiguity between class and class instance projection in the `from` part. However, we still would not be able to answer two simple query templates listed below:

1. Find the range (domain) of a specified property.

2. Find resources whose range is a specified class.

The `from` term in deVos' DAML Query Language can only include `DAML:Class`. We first changed the range of this term to `DAML:Thing`, so we are able to put properties in the `from` part of the query. For example, to find the domain of the `availability` property, we have the following query:

```
<Query>
      <select rdf:resource="http://www.w3.org/2000/01/rdf-schema#domain"/>
      <from rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#availability" /
</Query>
```

Figure 4.7: Find the domain of a property

Notice the `from` term of the above query. As another example, we try to find the domain of the properties whose ranges are `AttributeValues` in ServiceTypes ontology.

```
<Query>
    <select rdf:resource=" http://www.w3.org/2000/01/rdf-schema#domain "/>
    <from>
       <daml:ObjectProperty>
          <daml:range
               rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                    ServiceTypes.daml#AttributeValues" />
       </daml:ObjectProperty>
    </from>
</Query>
```

Figure 4.8: Find the domain of properties whose range is the given class

In the the query in Figure 4.8, we use `DAML:ObjectProperty` to narrow the search space of the subject resources to object properties.

4.3.4 Returning Name of Classes and Properties

The language proposed by deVos was a triple-based language. It returns all the statements, which include properties in the `select` part over the resources specified in the `form` part of the query. Instead of returning the reply in triples form, we decided to return only values of properties specified in the `select` part of the query as key-value pairs. This approach, which resembles resource-centric query languages, reduces the useless information that would be returned if we query services (data portion of the system). As we discussed above, service data is formed by instantiating one of the service category classes in our ontology. In this formulation, we are interested only in properties and their values. We do not need the name of the subject or the object. This is reasonable, because the name of an object or a subject is only the implementation detail of DAML, and we don‘t store service data as a DAML-based data source. As shown in Figure 4.9, we are not interested in the name of the resource `value-1`. We only want to know that some values exist for `availability` property.

```
<ont:availability rdf:resource="#value1"/>

<ont:AttributeValues rdf:ID="value1">
   <ont:value>90</ont:value>
   <ont:unit>percentage</ont:unit>
   <ont:predicate rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#GREATER"/>
</ont:AttributeValues>
```

Figure 4.9: A sample class and a class instance

As a consequence of the above argument, the result of the following query cannot return the name of the object whose predicate is `availability`.

```
<select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#availability"/>
```

Figure 4.10: Find the property of all services

Normally the object value of the `availability` property is the resource with name `value-1`; but as we mentioned above we were not interested in the resource identifier (name of the class). The above query returns the answer as if resources are joined automatically by using the `RDF:ID` as the shared key between these resources.

Querying the service data is just one side of the coin. On the other side of the coin, we need to query service metadata. While querying the service metadata, we need the obtain the name of classes and properties. For example, if we want to find properties whose range is `AttributeValue` in our ontology, we have the following query:

```
<Query>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/dql.daml#subjectRDFID"/>
    <from>
        <daml:ObjectProperty>
            <daml:range
                rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                  ServiceTypes.daml#AttributeValues" />
        </daml:ObjectProperty>
    </from>
</Query>
```

Figure 4.11: Find the name of the resources whose `range` is the given class

In the above query `subjectRdfID` is a property whose domain is the RDF-Schema property. The above query returns the name of all subject resources of triples, which satisfy the condition in the `from` part. Similar to `subjectRdfID`, we also define `objectRdfID` and `predicateRdfID` properties as part of our DAML query description syntax. The `objectRdfID` is used to return the name of all object resources. The `predicateRdfID` is used to return the name of all predicate resources of triples that satisfy the condition in the `from` part.

4.3.5 Transitive Properties

One of the most important features of our system are subsumption and equivalence relationships between concepts (classes or properties). DAML has no primitives to query such relations. If we want to find the domain of a property by using `DAML:domain` or `RDF-Schema:domain` properties in the `select` statement, we can only find classes that are declared to be the domain of a specific property. We cannot find the transitive domain of a property by using the current primitives. For example, if the domain of the `availability` property is `GenericAttributes` (as specified in our ontology) and `VehicleRental` is the subclass of `GenericAttributes`, we cannot figure out that `VehicleRental` is also the domain of the `availability` property by using the current language primitives. For this reason we added some primitives that can be used to query transitivity of properties. Each of these primitives is defined as a `DAML:Property`. All of the properties below can either be used in the `select` or the `from` parts of a query:

- **transitiveDomain:** The domain is `DAML:Property` and their range is `DAML:Class`.
- **transitiveRange:** The domain is `DAML:Property` and their range is `DAML:Class`.
- **transitiveSubPropertyOf:** The domain is `DAML:Property` and the range is `DAML:Property`.
- **transitiveSubClassOf:** The domain is `RDF-Schema:Resource` and the range is `DAML:Class`. The domain of this property is declared as `RDF-Schema:Resource`; because the `select` statement can only accept `DAML:Property`. On the other hand, like `DAML:subClassOf`, `transitiveSubClassOf` can also be used as a property whose domain is `DAML:Class`. Both `DAML:Class` and `DAML:Property` is subclass of

RDF-Schema:Resource.

As an example, the query in Figure 4.12 finds the name of classes, that are subclass of VehicleRental:

```
<Query>
  <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/dql.daml#subjectRdfID"/>
  <from>
      <daml:Class>
          <transitiveSubClassOf  rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                ServiceTypes.daml#VehicleRental"/>
       </daml:Class>
  </from>
</Query>
```

Figure 4.12: Find the names of resources that are subclasses the given class

4.3.6 Inserting New Data

We also need to insert meta (data). We added the `insert` property whose domain is `Query` and range is `DAML:Thing`. The `insert` property resembles to the functionality of SQL `insert` function. If the meta(data) with the specified identity already exists, then it does not insert it; otherwise it inserts the new meta(data). Insertion is done at the class or the property levels. In the metadata, consistency checking is done through the name of the class or the property. In the database in which service data is stored, consistency is checked through the primary keys of tables, which can be mapped to the identifiers of the subjects, e.g., the `serviceKey` and the `publishedRegistryUrl` attributes of service. We either insert a tuple to the database or we insert a new class and new property to our ontology. Updates are not allowed. We cannot merely change the specific column value, or the properties of the `DAML:Class` and `DAML:Property` resources. This restriction is due to the hard consistency requirement between the ontology and the data. For the insertion request, class or property definitions are given using the `declare` property.

For example, the query in Figure 4.13 attempts to insert the `price` property for the service whose `serviceKey` is "159734c6-bed1-417c-b74f-066dcf3f13b5" and is published in the Microsoft UDDI Test Registry. In this figure, new resources, which can either be the subjects or the objects of the insertion statement in the `insert` part of the query, are defined by the `declare` term.

The query in Figure 4.13 inserts a new tuple. If we want to insert metadata, we have the same query formulation except the `identifier` part. The query in Figure 4.14 inserts `maximumBaggageCapacity` in our ontology.

4.3.7 Intersection, Union and Except

In deVos' language multiple restrictions were achieved by using `DAML:subClassOf` in the `from` part of the query . The default set operator was intersection in these queries and it was implicitly defined. By using DAML set operator primitives like `unionOf` and `disjointWith`, we are able to perform more algebraic set operations on result sets. In Figure 4.15, we query all the services whose price is not expensive by using `DAML:disjointWith`. We use the `disjointWith` primitive in SWSQML to query the resources that don't satisfy the given restriction.

```
<Request>
  <identifier>
    <daml:Class>
      <daml:subClassOf>
        <daml:Restriction>
          <daml:onProperty rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
          <daml:hasValue>159734c6-bed1-417c-b74f-066dcf3f13b5</daml:hasValue>
        </daml:Restriction>
      </daml:subClassOf>
      <daml:subClassOf>
        <daml:Restriction>
          <daml:onProperty rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                ServiceTypes.daml#publishedRegistryUrl"/>
          <daml:hasValue>http://test.uddi.microsoft.com</daml:hasValue>
        </daml:Restriction>
      </daml:subClassOf>
    </daml:Class>
  </identifier>

  <declare>
      <ont:ServiceProvider rdf:ID="p1">
        <ont:partyIdentifier>uuid:8993643</ont:partyIdentifier>
        <ont:partyName>Soydan Bilgin</ont:partyName>
      </ont:ServiceProvider>
  </declare>

  <declare>
      <ont:AttributeValues rdf:ID="v3">
        <ont:value>2</ont:value>
        <ont:unit>dollar</ont:unit>
        <ont:predicate rdf:resource="http://www.damlsmm.ri.cmu.edu/data/Math.daml#EQUAL"/>
        <ont:submittedBy rdf:resource="#p1">
      </ont:AttributeValues>
  </declare>

  <evaluate>
      <Query>
          <insert>
            <daml:Restriction>
                <daml:onProperty  rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                    ServiceTypes.daml#price"/>
                <daml:hasValue rdf:resource="#v3"/>
            </daml:Restriction>
          </insert>

      </Query>
  </evaluate>

</Request>
```

Figure 4.13: Insert values for the new attribute of a given service

```
<Request>
   <declare>
      <daml:ObjectProperty rdf:ID="maximumBaggageCapacity">
         <rdfs:label>maximumBaggageCapacity</rdfs:label>
         <rdfs:domain
            rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#CarRental"/>
         <rdfs:range
            rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#AttributeValues"/>
      </daml:ObjectProperty>
   </declare>
   <evaluate>
      <Query>
         <insert rdf:resource="#maximumBaggageCapacity"/>
      </Query>
   </evaluate>
</Request>
```

Figure 4.14: Insert `maximumBaggageCapacity` property to the ontology

```
<Query>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
    <from>
        <daml:Class>
            <daml:disjointWith>
                <daml:Restriction>
                    <daml:onProperty
                        rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#price"/>
                    <daml:hasClass>
                        <daml:Restriction>
                            <daml:onProperty
                                rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                                    ServiceTypes.daml#value"/>
                            <daml:hasValue
                                rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
                                    QualityRating.daml#Expensive"/>
                        </daml:Restriction>
                    </daml:hasClass>
                </daml:Restriction>
            </daml:disjointWith>
        </daml:Class>
    </from>
</Query>
```

Figure 4.15: Find the services which do not satisfy the restriction

Chapter 5

Implementation

We have implemented basic properties of the existing DAML Query Language with proposed enhancements. We have denoted this enhanced language as Semantic Web services Query and Manipulation Language (SWSQML). We have implemented the SWSQML as a Web service. Just as UDDI is used to discover and publish Web services catalog information, the SWSQML can be used to query quality attributes of a Web service as well as schema of these quality attributes. Our Web service also provides limited publishing functionality.

The SWSQML has two parts. The data part corresponds to the representation of Web services by quality attributes and the metadata part corresponds to the schema of the quality attributes. Our quality attributes ontology is limited size and is requested very frequently by both parts. At the same time, ontology data is the most difficult to query from a relational database because of the transitivity of the `subClassOf` and the `subPropertyOf`. Because of these reasons, we store and process our ontology in memory. On the other hand, we store Web services data in a relational database as a non-DAML data source. There are two main reasons to use a relational database for storing our data portion of the system. First, there can be thousands of Web services that cannot fit in main memory. Second,

mapping from DAML classes to database tables is trivial. The users of SWSQML service can use the mapping information, which is explained in Section 5.4, to query the database as if the data were converted to DAML instances.

5.1 Implementation Environment

We implemented the service in Java. We have developed and tested this application on Windows 2000 environment. We used an open source Java IDE environment called Eclipse from eclipse.org. We‘ve used MySql database to store our data. We've also used DAML specific tools such as DAML Validator [Rager, 2003], DAMLJessKB [Kopena and Regli, 2003], and the Jena DAML API [HPL, 2001]. The brief descriptions of these tools are given in Section 5.2.

5.2 Basic Architecture

We implemented the whole service as a RPC-based Web service. The input to our service is the URL for the query file and the output is the URL for the answer file generated by the service. The input query file is expressed in DAML, so we first validate it by using the DAML Validator, which is provided by daml.org. This validator uses ARP Parser of the Jena Framework to parse input files. After validation, we read the input query file by using DAMLJessKB, which converts a DAML file into a set of equivalent subject-predicate-object triples. We‘ve used DAMLJessKB instead of the Jena DamlModel to read the query file because of its more structured representation of anonymous classes, which are heavily used in our queries. We also used DAMLJessKB to figure out subsumption and equivalence relations. The basic architecture of our DAML query service is shown in Figure 5.1

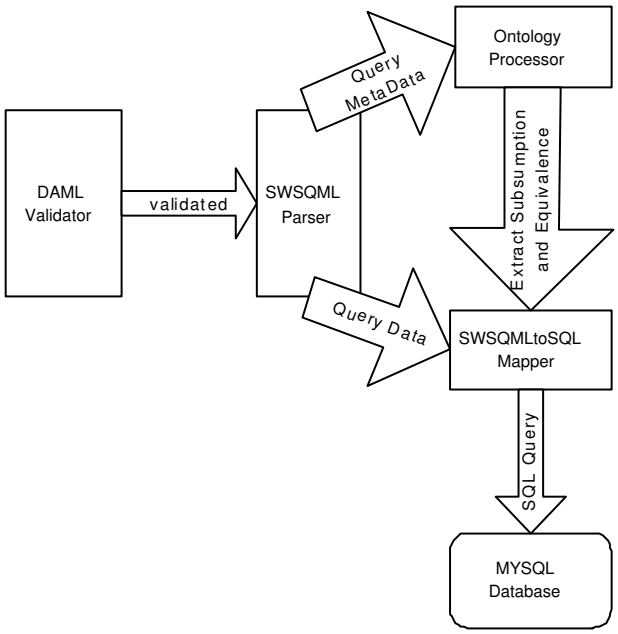

Figure 5.1: Basic architecture of DAML query service

We decide which part of the system to query depending on the properties used in `select` term of the query. The properties listed in the left side of Table 5.1 are used in querying the relational database. The properties listed in the right side of Table 5.1 are used in querying the ontology. It can be easily observed that there are no common properties used for querying both parts. In our system, we have just one ontology file, so we haven't used a term to identify the ontology namespaces that will be queried. We don't have a term like `using namespace` as in RDQL [HPL, 2002], which is the query language used in the Jena Framework. In RDQL, it is necessary to figure out sources that need to be used while querying. The `using namespace` term can also be introduced to our DAML query description syntax to support querying more than one ontology file.

After parsing the input query file, if we have to query the metadata, we use the Ontology Processor. The Ontology Processor is initialized by reading our quality attributes ontology into main memory by using DAMLJessKB package, so that we obtain the triple-based

Querying RDBMS	Querying Ontology
Properties, that correspond to table columns	Properties, defined in DAML and RDF-Schema specifications
'RDF:Type' property	SUBJECT_RDF_ID OBJECT_RDF_ID PREDICATE_RDF_ID
Properties, that are quality attributes	TRANSITIVE_DOMAIN TRANSITIVE_RANGE TRANSITIVE_SUBCLASSOF TRANSITIVE_SAMECLASSAS TRANSITIVE_SUBPROPERTYOF TRANSITIVE_SAMEPROPERTYAS

Table 5.1: Properties used while querying RDBMS and the ontology

representation of our ontology. These triples are then asserted into Jess [Friedman-Hill, 1997] knowledge base and the query is applied on this knowledge base. The usage of the Jess and the DAMLJessKB reduced our development time and facilitated reasoning about subsumption and equivalence relations between concepts. We also used the Jena DAML API to insert new classes and properties into our ontology.

If we have to query the data, which corresponds to a MySQL database, then we use SWSQMLtoSQL Mapper that converts DAML query into SQL query. SWSQML-toSQLMapper relies on the relational schema of underlying database to generate SQL statements. This module also uses the methods of the Ontology Processor to make inferences.

5.3 Querying Metadata

Querying metadata consists of processing the ontology triples in the memory by using DAMLJessKB. DAMLJessKB knowledge base contains the triples in ⟨*subject, predicate,*

object⟩ format. DAMLJessKB also defines additional axioms to find transitive closure of concepts. It is straightforward to separate DAML query into subject, predicate and object subparts to form the input for knowlege base. According to the name of the predicate, the Ontology Processor may need to scan knowlege base recursively. For example, the `TRANSITIVE_SUBCLASSOF` predicate requires recursively scanning the knowledge base for each `subClassOf` predicate of given subjects or objects.

5.4 Querying Data

Querying data consists of two main tasks. These are translation from DAML query to SQL query, and reasoning to find subsumption and equivalence relations between concepts. The translation task is performed by the SWSQMLtoSQLMapper component and the rasoning task is performed by the Ontology Processor component. Therefore querying RDMS also requires processing the ontology file in memory as explained in Section 5.3

We've tried to make the translation from DAML query to SQL query as simple as possible. Making it simple lets us to adapt modified ontology structures to database schema easily. Translation from DAML query to SQL query requires the definition of a mapping between database schema and the ontology. By using this mapping information, the user of our service can query the database as if the data were converted to DAML instances.

5.4.1 Database Schema and Mapping

The relationship between database tables and a mapping from the DAML instance structure to table structure is shown in Figure 5.2 and Figure 5.3, respectively. We've used three tables: the *services* table stores subset of the UDDI-related registry information, the *attributes* table is a vertically designed table, which stores values of quality attributes related

to a service, the *parties* table stores some basic information about the potential service providers and service consumers. In summary, the idea is to map the tables directly to DAML classes. Table rows become instances and each row can be viewed as a resource where columns are literals or links to other resources.

In Figure 3.4, in Chapter 3, we showed a DAML instance for a complete `VehicleRental` service as if it were stored as a file. The mapping from this DAML instance file to database table is trivial. It is almost one to one mapping from classes to tables, properties to columns, and instances to rows except three important differences, which can be easily observed from Figure 5.2 and Figure 5.3, respectively.

1. Addition of the *attributeName* column to the attributes table. We add this column to store names of quality attributes in a vertically designed attributes table instead of services table. We've designed the attributes table vertically because the number and the type of quality attributes assigned to a service can change for each service.

2. Addition of the *serviceType* column to the services table. As a result of this additon, the type of the service is stored as a column value, not as a table name.

3. Addition of the *partyType* column to parties table. As a result of this additon, the name of classes, which are subclass of `Party`, is stored as a column value.

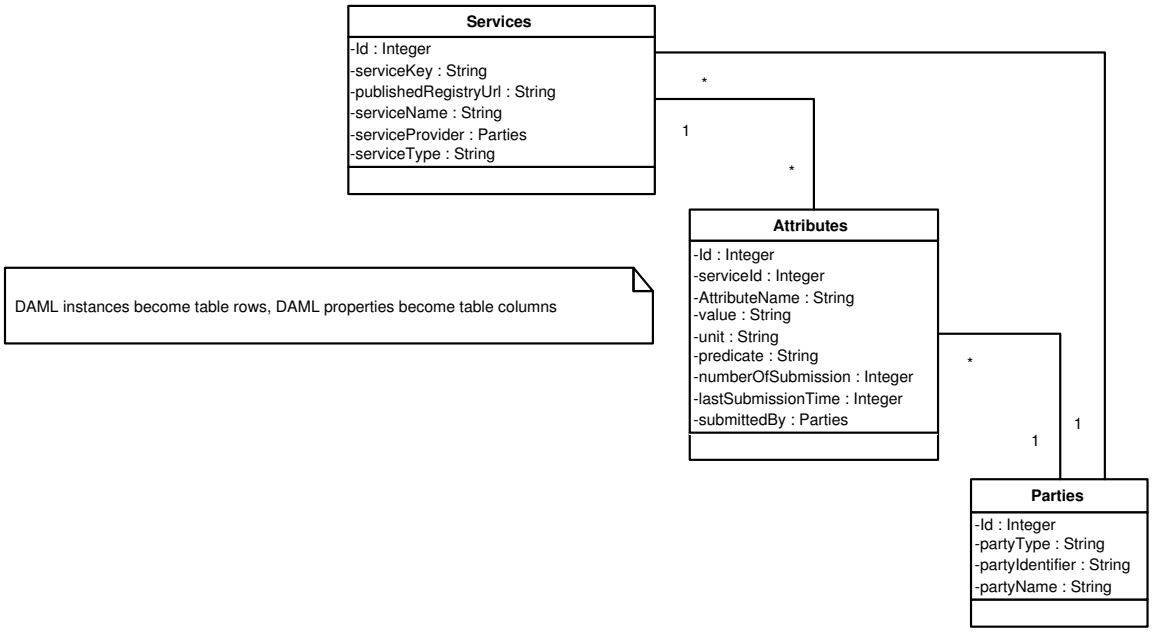

Figure 5.2: Representation of RDBMS table structure

5.4.2 Conversion from DAML Query Language to SQL

We have used MySQL database as our relational database; so we make the translation according to the MySQL SQL syntax. We‘ve defined a mapping function, which maps properties to the columns and the tables of the database. For example, the `availability` is mapped to the attributeName column of the attributes table, the `serviceKey` is mapped to the serviceKey column of services table. We use joins, self-joins and table aliasing heavily. For example, two cases that we use the table aliasing are:

1. If multiple properties that refer to the same table are selected, e.g., submittedBy and serviceProvider properties both refer to the parties table

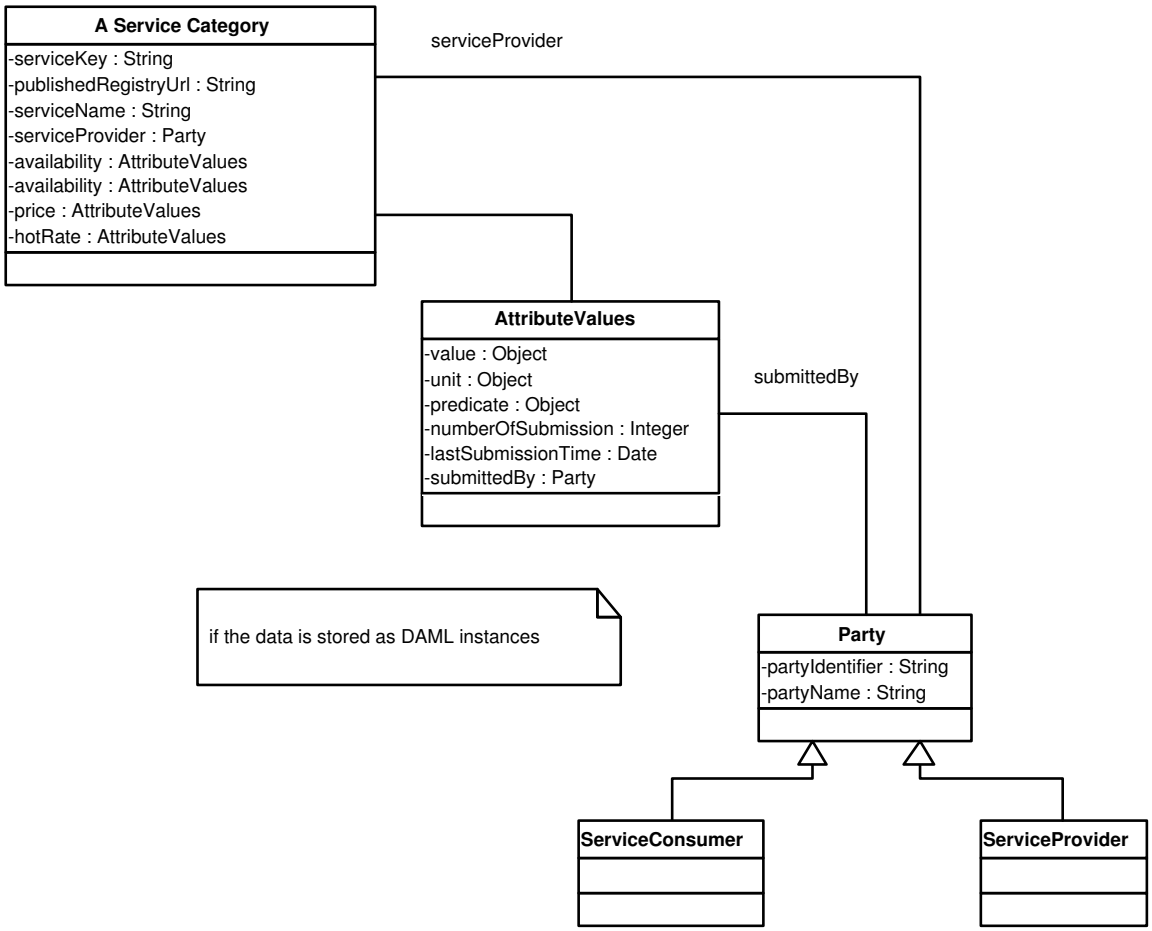

Figure 5.3: Representation of DAML classes

2. If multiple properties reference to the same column more than once, e.g., *finding the availability property of services which has the price property* requires self-join and table aliasing.

As we mentioned in Chapter 4, we don't store the name of DAML class instances. Therefore the queries such as in Figure 4.10 prefetches the data from the tables whose key is used as a foreign key in the original table. For example, in Figure 4.10, the original table, which is obtained from mapping information is attributes table. This table has to be

joined with parties table by using the value for submittedBy property to fetch the values for partyIdentifier and partyName.

The mapping of set operators such as `unionOf` and `intersectionOf` is trivial. For the `disjointWith` operator, we use the corresponding *NOT IN* SQL operator.

For queries that requires reasoning about subsumption and equivalence of concepts, we process our ontology by using the Ontology Processor component. For each quality attribute restriction in the `from` part of the query, we check all `subPropertyOf` and `samePropertyAs` relationships of the property until the requested restriction is satisfied. We seperate such implicit semantic relationships by using *OR* SQL operator in the translated query. With the same approach, for each service with the given service category type, we check all `subClassOf` and `sameClassAs` relationships of the service category to return the services which are of same-type or super-type. For example, the equivalent SQL translation for the query in Figure 5.4 is given in Figure 5.5.

```
<Query>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#serviceKey"/>
    <from>
        <RestrictedTo>
            <subjectType  rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#CarRental" />
            <daml:onProperty
                rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#costPerUse"/>
            <daml:hasClass>
                <daml:Restriction>
                    <daml:onProperty
                        rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#value"/>
                    <daml:hasValue
                        rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/QualityRating.daml#Cheap"/>
                </daml:Restriction>
            </daml:hasClass>
        </RestrictedTo>
    </from>
</Query>
```

Figure 5.4: Find the `serviceKey` of car rental services which has `costPerUse` = “cheap”

```
SELECT s1.serviceKey
FROM services AS s1, attributes AS t1
WHERE (s1.serviceType='CarRental' OR s1.serviceType='VehicleRental')
   AND s1.Id=t1.serviceId
   AND (t1.attributeName='costPerUse'  OR  t1.attributeName='price')
   AND t1.value='http://vegas.csc.ncsu.edu:8080/wsap/QualityRating.daml#Cheap'
```

Figure 5.5: Equivalent SQL translation for the query in Figure 5.4

We generate one SQL select statement for each `evaluate` term in the query request. On the other hand, we can generate multiple SQL insert statements for each `insert` term. For insertion requests, the values for lastSubmissionTime and numberOfSubmissions columns are updated automatically by the system.

5.5 Limitations of the Implementation

We have some limitations in our implementation. Most of these limitations are due to our intent to preserve the simplicity of the implementation. We list these limitations here.

In our implemantation, we provide limited insertion functionality for both the relational database and the ontology. On the other hand, we do not provide any modification (update) or deletion functionality due to hard consistency problems. For example, when updating a `subClassOf` relation between two existing classes, the class hierarchy changes and needs to be rebuilt again. Adding new `subClassOf` relations can cause cyclic relations between classes in the ontology and deleting a class can affect the complete ontology. All these kind of modifications also affect the data stored in the database. For example, if `CarRental` class is deleted, then the services which has `CarRental` service category type has to be deleted to preserve the consistency between the ontology and the data based on this ontology. As a result of these drawbacks, we have decided to provide only insertion functionality with limited properties as explained in Section 4.3.6.

We provide a class hierarchy of service categories defined in our ontology. In our implementation, a service can structurely have only one type; but semantically can have many types by the usage of `subClassOf` relation. We don't provide multiple instantiation modeling mechanism for the same service. A service cannot structurely have two types. For example, the same service cannot both have `CarRental` and `TruckRental` types. This feature can be provided by using `DAML:sameIndividualAs` property. For example, as shown in Figure 5.6, a service with two different types can be inserted into the database. The problem reveals, when we try to insert two class instances that refer each other, with different insert requests, because it is not possible to refer class instances with their names. The names of the class instances are not stored in RDBMS as explained in Section 4.2.

```
<CarRental rdf:ID="Soydan-CarRental">
    <serviceKey>159734c6-bed1-417c-b74f-066dcf3f13b5</serviceKey>
</CarRental>

<TruckRental rdf:ID="Soydan-TruckRental">
    <daml:sameIndividualAs rdf:resource="#Soydan-CarRental"/>
</TruckRental>
```

Figure 5.6: The declaration of a service with two different types

In our implementation, we query only one ontology. If more than one ontology has to be queried, then we need to add the functionality to declare the namespaces of ontologies that will be queried.

In our implementation, each service can be represented with three different frame typess, where each frame type corresponds to a table in the relational database. This is shown in Figure 5.2. Our implementation depends on the uniqueness of properties that can be used in different frames. The usage of properties with multiple domains that correspond to different frames in Figure 3.5 or the usage of multiple ontologies, where the properties with the same name can be defined, result in ambiguous queries. For example, if we

rename `serviceProvider` as `submittedBy` property, then both `VehicleRental` and `AttributeValues` classes in Figure 3.5 will have `submittedBy`. In this case, the top query of Figure 5.7 can be ambiguous. To eliminate this ambiguity, we need to add functionality to alias different frames as in the bottom query of Figure 5.7.

```
<select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#submittedBy"/>
<from>.....</from>

<select rdf:resource="AttributeValues.submittedBy"/>
<from>.....</from>
```

Figure 5.7: Queries with and without aliasing

In this implementation, we don‘t support features like aggregate functions (min, max, average, sum, count), grouping (e.g., groupBy clause functionality as in SQL), and sorting (for ordering querying results)

Chapter 6

Discussion

This chapter describes some relevant literature, summarizes our contribution, and concludes with a discussion about future directions.

6.1 Related Work

The main focus of our work is Semantic Web query languages. Artificial or human agents have to use and query ontologies and the resources based on them due to wide acceptance of Semantic Web. [Plexousakis et al., 2002] presents a detailed study on current ontology querying languages by providing an evaluation framework, which uses criterias such as modeling constructs, inference support, the closure, the generality, and the orthogonality of the language, update support, and API support for describing language characteristics.

In this section, we will focus on query languages for RDF, RDF Schema, and DAML description bases. As already explained in Section 2.2, these three standards and OWL represents the evolution of description bases used for Semantic Web. RDF with the aid of RDF Schema provides a rich infrastructure to create ontologies. DAML and OWL extends RDF Schema with richer modeling primitives. We describe RQL (RDF Query Language) and

RDQL (RDF Data Query Language) as the most popular querying languages for RDF and RDF Schema description bases, DQL (DARPA DAML Query Language) as the query description language, which provides agent-to-agent query answering dialogues using knowledge represented in DAML, and Sesame as the most efficient and generic RDF-Schema based repository and querying facility, which provides RQL and RDQL implementations. We will also briefly mention RDF Query Specification as the first SQL-like approach, viewing RDF description bases as a relational base.

Our Semantic Web services Query and Manipulation language resembles to RDQL in terms of SQL-like approach, but we have a different query description syntax, which uses DAML primitives and doesn't assume an underlying DAML-based repository. Like RQL, we also provide basic inference support and subquerying where the output of the subquery can be the input for the outer query.

6.1.1 RDF Query Language

[Karvounarakis et al., 2002] presents RQL as the language for retrieving information represented in RDF and RDF Schema from the Web. It adopts the functionality of XML query languages to RDF and RDF Schema description bases. They provide ontology querying capability assuming an underlying RDF-Schema based repository. RQL provides features like aggregate, grouping, and sorting functions, nested query, existential and universal quantifier support, set-based, and arithmetic operations on data values. RQL has the ability to combine ontology and data querying in a transparent way, which enables the agents to query resources described according to their preferred ontology. We also provide the similar and basic version of this feature while translating queries that requires reasoning about subsumption and equivalence of concepts. RQL doesn't provide inference support for DAML description bases and update support to modify the contents of existing ontology

and the data.

Our query language mainly differs from RQL by its syntax, which depends on DAML primitives. For a better understanding, we give a simple example expressed in both RQL and our query language in Figure 6.1.

RQL version:

```
select $P, $Y
from {$X}$P{$Y}
where $X<=VehicleRental
```

DAML Query Language version:

```
<Query>
    <select rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/dql.daml#SUBJECT_RDF_ID"/>
    <select rdf:resource="http://www.w3.org/2000/01/rdf-schema#range"/>
    <from>
      <RestrictedTo>
        <subjectType rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
        <TRANSITIVE_DOMAIN rdf:resource="http://vegas.csc.ncsu.edu:8080/wsap/
            ServiceTypes.daml#VehicleRental"/>
      </RestrictedTo>
    </from>
</Query>
```

Figure 6.1: Find all the property types and their corresponding range, which can be used on a resource of type `VehicleRental` or any of its subclasses

6.1.2 RDF Data Query Language

RDQL [HPL, 2002] is a query language, which is being developed by the Hewlett Semantic Web Group. RDQL is part of the Jena framework. It uses SQL-like constructs for the query description. RDQL regards RDF model as a set of triples and can only query at the structure level. There is no inference being done by RDQL. [Sirin, 2002] presents a database agent that translates RDQL queries into SQL queries by defining a mapping function from DAML classes to database table structure. The functionality of the database

agent resembles to our SWSQMLtoSQLMapper module, but we provide the mapping from a query described by DAML primitives to SQL and we consider the results of inference being done via the constructs such as `sameClassAs`, `samePropertyAs`, `subClassOf`, and `subPropertyOf` while mapping.

6.1.3 RDF Query Specification

RDF Query Specification [Malhotra and Sundaresan, 1998] is the first proposed SQL style approach, viewing RDF description bases as a relational database. It resembles to RDQL in terms of the querying capabilities except that RDF Query Specification doesn't cover RDF Schema description bases. The specification allows the usage of RDF primitives while constructing queries like our SWSQML uses DAML primitives.

6.1.4 Sesame

Sesame is a RDF Schema based repository and querying facility, which is being developed by Aidministrator Nederland [Broekstra et al., 2002]. Sesame is an architecture that provides a persistent storage capability for RDF-Schema description bases. It supports the basic inference capability needed for supporting RDF-Schema, such as transitivity of the `subClassOf` and the `subPropertyOf`. Sesame also enables adding new RDF statements to the database. It provides implementations for RQL and RDQL; but it doesn't support DAML description bases.

6.1.5 DARPA DAML Query Language

DARPA Dql is the query language, which supports query-answering dialogues in which the answering agent may use automated reasoning methods on the knowledge represented

in DAML [Fikes et al., 2003]. DARPA Dql provides a description model, which allows hypothesizing an object by using a query premise. It allows < `if...else...` > kind of queries. `If` part hypothesize the object and `else` part makes questions by taking into account the hypothesis. It also allows queries on DAML description bases without a predicate. There are two implementations of this query language provided by Stanford KSL and UMBC. Both implementations process the ontology in memory. Stanford KSL provides a schema for the query specification in XML Schema. DARPA Dql is a kind of query description language for rule engines like Jess. DARPA Dql is a language for querying and reasoning metadata instead of data. Every subject and object is referenced by its name.

6.2 Main Contribution and Directions

This thesis presents the SWSQML, which uses DAML primitives for describing the queries. We have extended the previous DAML Query Language to facilitate querying quality attributes ontology and the Web services description expressed by the concepts from this ontology. We have implemented a basic query service for DAML description bases in a relational database for services and in memory for the ontology. We also provided a simple methodology for representing Web services with their quality attributes and identified the required query and manipulation templates to accommodate use cases for service selection and manipulation.

Several enhancements are possible to our work. One of them is to adapt our query service to query Web services based on OWL-S ontology. We can also look for the possible new OWL modeling primitives that can be useful for query descriptions. Another desirable enhancement can be to provide more functionality of SQL such as updating and deleting metadata and data resources. This will be a really tough task. For example, deleting a class

may require deleting the properties of other classes and properties that refer to the deleted class. Also deleting a class (a service category) in the ontology will require a referential integrity check for the instances of type deleted class. Updating the properties of a class will require a consistency check on the ontology to remove the cyclic relationships. The SWSQML can be the starting point for a query description standard that can be used among distributed agents on multiple data sources.

Bibliography

Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, David L. Martin, Ora Lassila, David L. Martin, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terence Payne, Katia Sycara, and Honglei Zeng. DAML-S: Web service description for the Semantic Web. In *The First International Semantic Web Conference (ISWC*, Sardinia, Italy, June 2002.

Naresh Apte and Toral Mehta. *Web services: A Java developer's guide using E-Speak*. Prentice Hall, New Jersey, 2001.

bindingpoint.com, 2001. http://www.bindingpoint.com.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. *Lecture Notes in Computer Science*, 2342:54–76, 2002. ISSN 0302-9743.

US Census Bureau. North American Industry Classification System (NAICS), 1997. http://www.census.gov/epcd/www/naics.html.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. Technical report, World Wide Web Consortium, 2001.

DARPA. Reference description of the DAML+OIL ontology markup language. Technical report, DARPA, 2001.

DARPA. DAML-S (and OWL-S) 0.9 beta draft release. Technical report, DARPA, 2003.

Arnold deVos. An RDF query language based on DAML, 2002. http://www.langdale.com.au/RDF/DAML-Query.html.

Richard Fikes, Pat Hayes, and Ian Horrocks. DAML query language (DQL). Abstract specification, DARPA, 2003.

Ernest J. Friedman-Hill. Jess, the rule engine for the Java platform, November 1997. http://herzberg.ca.sandia.gov/jess/docs/61/.

HPL. Jena Semantic Web toolkit, 2001. http://www.hpl.hp.com/semweb/jena.htm.

HPL. RDQL: RDF data query language, 2002. http://www.hpl.hp.com/semweb/rdql.htm.

ISO. 3166 counry codes, 2001. http://www.iso.ch.

Karsten Januszewski. The importance of metadata: Reification, categorization and UDDI. Technical report, MSDN, 2002.

Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A declarative query language for RDF. In *Proceedings of the Eleventh International World Wide Web Conference*, pages 592–603, Honolulu, Hawaii, 2002. ACM Press.

Joseph Kopena and William C. Regli. DAMLJessKB: A tool for reasoning with the Semantic Web. *IEEE Intelligent Systems*, 18(3):74–77, June 2003.

T. Berners Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284 (5):34–43, 2001.

Lei Li and Ian Horrocks. A software framework for matchmaking based on Semantic Web technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331–339, 2003. URL `download/2003/p815-li.pdf`.

Ashok Malhotra and Neel Sundaresan. RDF query specification. Technical report, IBM, 1998.

E. Michael Maximilien and Munindar Singh. Conceptual model of Web service reputation. In *ACM SIGMOD*, December 2002.

OASIS. ebXML, 2001. http://www.ebxml.org/.

Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Importing the Semantic Web in UDDI. In *Proceedings Web Services, E-Business and Semantic Web Workshop, CAiSE 2002.*, pages 225–236, Toronto, Canada, 2002b.

Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of Web service capabilities. In *The Semantic Web - ISWC 2002*, pages 333–347, Springer-Verlag, Berlin, 2002a.

Joachim Peer. Semantic annotation, matchmaking and composition of Web services. In *International Semantic Web Conference (ISWC-03)*, 2003. Poster.

Dimitris Plexousakis, Aimilia Magkanaraki, Vassilis Christophides, Grigoris Karvounarakis, and Ta Tuan Anh. Ontology storage and querying. Technical Report no. 308, ICS-FORTH, Heraklion, Crete, Greece, April 2002.

David Rager. DAML validator, 2003. http://www.daml.org/validator/.

salcentral.com, 2001. http://www.salcentral.com.

Evren Sirin. Agents for Semantic search, 2002. http://www.mindswap.org/.

UDDI. The UDDI technical white paper. Technical report, OASIS, 2000.

Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web ontology language reference. Working draft, World Wide Web Consortium, 2003.

W3C. Resource description framework (RDF) model and syntax specification. Technical report, World Wide Web Consortium, 1999.

W3C. RDF vocabulary description language 1.0: RDF Schema. Working draft, World Wide Web Consortium, 2002.

Appendix A

Ontology files

A.1 A Sample Quality Attributes Ontology

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:oiled="http://img.cs.man.ac.uk/oil/oiled#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
    xmlns:exd="http://vegas.csc.ncsu.edu:8080/wsap/type-dt.xsd#"
    xmlns    ="http://vegas.csc.ncsu.edu:8080/wsap/ServiceTypes.daml#">

    <daml:Ontology rdf:about="">
        <dc:title>Ontology for Service Types
         and their Properties</dc:title>
        <dc:date>April 25, 2003</dc:date>
        <dc:creator>Soydan Bilgin</dc:creator>
        <dc:description></dc:description>
        <dc:subject></dc:subject>
```

```
    <daml:versionInfo>V 1.0</daml:versionInfo>
    <daml:imports
       rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>

<daml:Class rdf:ID="AttributeValues">
    <rdfs:label>Values</rdfs:label>
</daml:Class>

<daml:Class rdf:ID="VehicleRental">
    <rdfs:label>VehicleRental</rdfs:label>
    <daml:subClassOf rdf:resource="#GenericAttributes"/>
</daml:Class>

<daml:Class rdf:ID="ServiceProvider">
    <rdfs:label>ServiceProvider</rdfs:label>
    <daml:subClassOf rdf:resource="#Party"/>
</daml:Class>

<daml:Class rdf:ID="ServiceConsumer">
    <rdfs:label>ServiceConsumer</rdfs:label>
    <daml:subClassOf rdf:resource="#Party"/>
</daml:Class>

<daml:Class rdf:ID="GenericAttributes">
    <rdfs:label>GenericAttributes</rdfs:label>
    <daml:subClassOf rdf:resource="#Attributes"/>
</daml:Class>

<daml:Class rdf:ID="Attributes">
```

```
    <rdfs:label>Attributes</rdfs:label>
    <rdfs:comment><![CDATA[]]></rdfs:comment>
      <daml:subClassOf>
        <daml:Restriction daml:cardinality="1">
          <daml:onProperty rdf:resource="#serviceKey"/>
        </daml:Restriction>
    </daml:subClassOf>
     <daml:subClassOf>
        <daml:Restriction daml:cardinality="1">
          <daml:onProperty rdf:resource="#publishedRegistryUrl"/>
        </daml:Restriction>
    </daml:subClassOf>
    <daml:subClassOf>
        <daml:Restriction daml:cardinality="1">
          <daml:onProperty rdf:resource="#serviceProvider"/>
        </daml:Restriction>
    </daml:subClassOf>
</daml:Class>

<daml:DatatypeProperty rdf:ID="publishedRegistryUrl">
    <rdfs:label>publishedRegistryUrl</rdfs:label>
    <daml:domain rdf:resource="#Attributes"/>
    <daml:range
      rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</daml:DatatypeProperty>

<daml:ObjectProperty rdf:ID="serviceProvider">
    <rdfs:label>ServiceProvider</rdfs:label>
    <daml:domain rdf:resource="#Attributes"/>
    <daml:range rdf:resource="#ServiceProvider"/>
```

```
</daml:ObjectProperty>

<daml:DatatypeProperty rdf:ID="serviceName">
    <rdfs:label>serviceName</rdfs:label>
    <daml:domain rdf:resource="#Attributes"/>
    <daml:range
      rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</daml:DatatypeProperty>

<daml:DatatypeProperty rdf:ID="serviceKey">
    <rdfs:label>serviceID</rdfs:label>
    <daml:domain rdf:resource="#Attributes"/>
    <daml:range
      rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</daml:DatatypeProperty>

<daml:Class rdf:ID="TruckRental">
    <rdfs:label>TruckRental</rdfs:label>
    <daml:subClassOf rdf:resource="#VehicleRental"/>
</daml:Class>

<daml:Class rdf:ID="Party">
    <rdfs:label>Party</rdfs:label>
</daml:Class>

<daml:Class rdf:ID="CarRental">
    <rdfs:label>CarRental</rdfs:label>
    <rdfs:comment><![CDATA[]]></rdfs:comment>
    <daml:subClassOf rdf:resource="#VehicleRental"/>
</daml:Class>
```

```
<daml:Class rdf:ID="ThirdPartyProvider">
    <rdfs:label>ThirdPartyProvider</rdfs:label>
    <daml:subClassOf rdf:resource="#Party"/>
</daml:Class>

<daml:Class rdf:ID="TravelOrganization">
    <rdfs:label>TravelOrganization</rdfs:label>
    <daml:subClassOf rdf:resource="#GenericAttributes"/>
</daml:Class>

<daml:Class rdf:ID="HotelReservation">
    <rdfs:label>HotelReservation</rdfs:label>
    <daml:subClassOf rdf:resource="#GenericAttributes"/>
</daml:Class>

<daml:ObjectProperty rdf:ID="maximumPayLoad">
    <rdfs:label>maximumPayLoad</rdfs:label>
    <daml:domain rdf:resource="#TruckRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="availability">
    <rdfs:label>availability</rdfs:label>
    <daml:domain rdf:resource="#GenericAttributes"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="predicate">
    <rdfs:label>predicate</rdfs:label>
```

```
    <daml:domain rdf:resource="#AttributeValues"/>
    <daml:range
      rdf:resource="http://www.damlsmm.ri.cmu.edu/data/
          Math.daml#Relation"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="maximumLuggageCapacity">
    <rdfs:label>maximumLuggageCapacity</rdfs:label>
    <daml:domain rdf:resource="#TruckRental"/>
    <daml:domain rdf:resource="#CarRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rangeOfCars">
    <rdfs:label>rangeOfCars</rdfs:label>
    <daml:domain rdf:resource="#CarRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

 <daml:UniqueProperty rdf:ID="unit">
    <rdfs:label>unit</rdfs:label>
    <daml:domain rdf:resource="#AttributeValues"/>
    <daml:range
       rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
 </daml:UniqueProperty>

<daml:DatatypeProperty rdf:ID="numberOfSubmission">
    <rdfs:label>numberOfSubmission</rdfs:label>
    <daml:domain rdf:resource="#AttributeValues"/>
    <daml:range
```

```
       rdf:resource="http://www.w3.org/2000/10/XMLSchema#integer"/>
</daml:DatatypeProperty>

<daml:ObjectProperty rdf:ID="rangeOfTrucks">
    <rdfs:label>rangeOfTrucks</rdfs:label>
    <daml:domain rdf:resource="#TruckRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="submittedBy">
    <rdfs:label>submittedBy</rdfs:label>
    <daml:domain rdf:resource="#AttributeValues"/>
    <daml:range rdf:resource="#Party"/>
</daml:ObjectProperty>

<daml:UniqueProperty rdf:ID="partyIdentifier">
    <rdfs:label>partyIdentifier</rdfs:label>
    <daml:domain rdf:resource="#Party"/>
    <daml:range
       rdf:resource="http://www.w3.org/2000/10/XMLSchema#anyURI"/>
</daml:UniqueProperty>

<daml:ObjectProperty rdf:ID="AAARating">
    <rdfs:label>AAARating</rdfs:label>
    <daml:domain rdf:resource="#HotelReservation"/>
    <daml:range rdf:resource="#AttributeValues"/>
 </daml:ObjectProperty>

<daml:DatatypeProperty rdf:ID="lastSubmissionTime">
    <rdfs:label>lastSubmissionTime</rdfs:label>
```

```
    <daml:domain rdf:resource="#AttributeValues"/>
    <daml:range
       rdf:resource="http://www.w3.org/2000/10/XMLSchema#date"/>
</daml:DatatypeProperty>

<daml:ObjectProperty rdf:ID="hotRate">
    <rdfs:label>hotRate</rdfs:label>
    <daml:domain rdf:resource="#VehicleRental"/>
    <daml:range rdf:resource="#AttributeValues"/>
</daml:ObjectProperty>

 <daml:Property rdf:ID="value">
    <rdfs:label>value</rdfs:label>
    <daml:domain rdf:resource="#AttributeValues"/>
 </daml:Property>

<daml:Property rdf:ID="objectTypeValue">
    <rdfs:label>ObjectTypeValue</rdfs:label>
    <daml:subPropertyOf rdf:resource="#value"/>
    <daml:range
       rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</daml:Property>

<daml:Property rdf:ID="dataTypeValue">
    <rdfs:label>DataTypeValue</rdfs:label>
    <daml:subPropertyOf rdf:resource="#value"/>
    <daml:range
       rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</daml:Property>
```

```
    <daml:ObjectProperty rdf:ID="costPerUse">
        <rdfs:label>costPerUse</rdfs:label>
        <daml:domain  rdf:resource="#GenericAttributes"/>
        <daml:range rdf:resource="#AttributeValues"/>
    </daml:ObjectProperty>

    <daml:DatatypeProperty rdf:ID="partyName">
        <rdfs:label>partyName</rdfs:label>
        <daml:domain rdf:resource="#Party"/>
        <daml:range
           rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
    </daml:DatatypeProperty>

    <daml:Class rdf:ID="SLAAttributeValues">
        <rdfs:label>ServiceLevelAgreements</rdfs:label>
        <daml:subClassOf rdf:resource="#AttributeValues"/>
    </daml:Class>

    <daml:Class rdf:ID="EvaluationAttributeValues">
        <rdfs:label>EvaluationAttributes</rdfs:label>
        <daml:subClassOf rdf:resource="#AttributeValues"/>
    </daml:Class>

     <daml:ObjectProperty rdf:ID="price">
        <rdfs:label>price</rdfs:label>
        <daml:samePropertyAs rdf:resource="#costPerUse"/>
     </daml:ObjectProperty>

</rdf:RDF>
```

A.2 Query Description Ontology for SWSQML

```
<?xml version='1.0'?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
xmlns="http://vegas.csc.ncsu.edu:8080/wsap/dql.daml#">

    <daml:Ontology rdf:about="">
        <daml:versionInfo>2002-02-15</daml:versionInfo>
        <daml:imports
          rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
    </daml:Ontology>

    <daml:Class rdf:ID="Request"></daml:Class>

    <daml:Property rdf:ID="declare">
        <daml:domain rdf:resource="#Request"/>
        <daml:range
          rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
    </daml:Property>

    <daml:Property rdf:ID="evaluate">
        <daml:domain rdf:resource="#Request"/>
        <daml:range rdf:resource="#Query"/>
    </daml:Property>
```

```
    <daml:Class rdf:ID="Query">
         <daml:subClassOf>
              <daml:Restriction>
                 <daml:onProperty rdf:resource="#select" />
              </daml:Restriction>
         </daml:subClassOf>
</daml:Class>

<daml:Property rdf:ID="select">
             <daml:range
                 rdf:resource="http://www.w3.org/1999/02/
                       22-rdf-syntax-ns#Property"/>
</daml:Property>

<daml:Class rdf:ID="RestrictedTo">
         <daml:subClassOf>
             <daml:Restriction>
                  <daml:onProperty rdf:resource="#select" />
                  <daml:maxCardinality>1</daml:maxCardinality>
             </daml:Restriction>
         </daml:subClassOf>
         <daml:subClassOf>
             <daml:Restriction>
                  <daml:onProperty rdf:resource="#objectType" />
                  <daml:maxCardinality>1</daml:maxCardinality>
             </daml:Restriction>
         </daml:subClassOf>
         <daml:subClassOf>
             <daml:Restriction>
                  <daml:onProperty rdf:resource="#subjectType" />
```

```
                    <daml:maxCardinality>1</daml:maxCardinality>
                </daml:Restriction>
            </daml:subClassOf>
            <daml:subClassOf
                  rdf:resource="http://www.daml.org/2001/03/
                     daml+oil#Restriction"/>
</daml:Class>

<daml:Property rdf:ID="objectType">
            <daml:range
               rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</daml:Property>

<daml:Property rdf:ID="subjectType">
            <daml:range
               rdf:resource="http://www.w3.org/2000/01/
                   rdf-schema#Class"/>
</daml:Property>

<daml:UniqueProperty rdf:ID="from">
            <daml:domain rdf:resource="#Query"/>
            <daml:range
               rdf:resource="http://www.daml.org/2001/03/
                   daml+oil#Thing"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="SUBJECT_RDF_ID">
            <daml:domain
               rdf:resource="http://www.w3.org/1999/02/
                   22-rdf-syntax-ns#Property"/>
```

```
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="OBJECT_RDF_ID">
         <daml:domain
            rdf:resource="http://www.w3.org/1999/02/
                22-rdf-syntax-ns#Property"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="PREDICATE_RDF_ID">
         <daml:domain
            rdf:resource="http://www.w3.org/1999/02/
                22-rdf-syntax-ns#Property"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="TRANSITIVE_DOMAIN"/>

<daml:UniqueProperty rdf:ID="TRANSITIVE_RANGE"/>

<daml:UniqueProperty rdf:ID="TRANSITIVE_SUBCLASSOF">
         <daml:domain
             rdf:resource="http://www.w3.org/2000/01/
               rdf-schema#Resource"/>
         <daml:range
             rdf:resource="http://www.w3.org/2000/01/
               rdf-schema#Class"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="TRANSITIVE_SAMECLASSAS">
         <daml:domain
             rdf:resource="http://www.w3.org/2000/01/
```

```
                rdf-schema#Resource"/>
          <daml:range
              rdf:resource="http://www.w3.org/2000/01/
                rdf-schema#Class"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="TRANSITIVE_SAMEPROPERTYAS">
          <daml:subPropertyOf rdf:resource="#TRANSITIVE_SUBPROPERTYOF"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="TRANSITIVE_SUBPROPERTYOF">
          <daml:domain
              rdf:resource="http://www.w3.org/1999/02/
                22-rdf-syntax-ns#Property"/>
          <daml:range
              rdf:resource="http://www.w3.org/1999/02/
                22-rdf-syntax-ns#Property"/>
</daml:UniqueProperty>

<daml:UniqueProperty  rdf:ID="identifier">
          <daml:domain rdf:resource="#Request"/>
          <daml:range rdf:resource="http://www.daml.org/2001/03/
                daml+oil#Thing"/>
</daml:UniqueProperty>

<daml:UniqueProperty rdf:ID="replace">
          <daml:domain rdf:resource="#Query"/>
          <daml:range rdf:resource="http://www.daml.org/2001/03/
                daml+oil#Thing"/>
</daml:UniqueProperty>
```

```
<daml:Class rdf:ID="SubQuery">
         <daml:subClassOf
            rdf:resource="http://www.w3.org/2000/01/
               rdf-schema#Class"/>
         <daml:subClassOf rdf:resource="#Query"/>
         <daml:subClassOf>
         <daml:Restriction>
             <daml:onProperty rdf:resource="#select"/>
             <daml:cardinality>1</daml:cardinality>
             </daml:Restriction>
          </daml:subClassOf>
</daml:Class>

</rdf:RDF>
```