Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Management of Semantic Web Services

Roberto De Virgilio

RT-DIA-153 **September 2009**

devirgilio@dia.uniroma3.it

ABSTRACT

Web Services Description Language (WSDL) allows a structured way to standardize the description of Web Services, exploiting XML for the exchange of structured information. However XML supports little interoperability between services, expected when WSDL documents have to be combined. In this context, the Semantic Web has become a promising research field, which tries to automate and support sharing and reuse of the data and metadata representing the growing amount of digital information available to our society. RDF and OWL have been conceived to represent, publish and share Web resources and ontologies over them. These models allow knowledge from many different sources to be easily combined so that unexpected data connections can be used. In this paper we provide a description of WSDL documents into a metamodel representation of Semantic Models. We present a tool and experiments on a real dataset for the efficient management of Semantic Web Services Descriptions. Our framework provides advanced and up-to-date ontology reasoning capabilities supported by a rule-based engine.

1 Introduction

In the last ten years we have observed an impressive evolution of the Web. From a static repository for text and multimedia it became a provider of information services such as flight-booking programs, e-commerce and business-to-business applications.

Web services promise easy access to remote content and application functionality, independently of the provider's platform, the location, the service implementation, or the data format. However, the description of a service is syntactic. Therefore all tasks associated with Web services application development have to be carried out by humans (i.e. discovery, composition and invocation). We face with problems of scalability and representation heterogeneity. In particular, as Larry Ellison[1] said, "*semantic differences remain the primary roadblock to smooth application integration, one which Web Services alone won't overcome*". Large-scale interoperation of Web services is the need to make such services machine processable, in order to build a Semantic Web of services whose properties, capabilities, interfaces, and effects are encoded in an unambiguous, machine-understandable form.

In 2001 Tim Berners Lee [8] opened new interesting perspectives by proposing the concept of Semantic Web, a world where information is also processable by machines. In order to let rough data be unambiguously interpreted by a computer, it is necessary to describe them by means of metadata that associate a well defined meaning to the entities involved. For this purpose the W3C developed `RDF` (Resource Description Framework) [21] and `OWL` (Web Ontology language) [24], languages that are commonly used for representing Semantic Web data today: RDF provides a simple way to represent any kind of data and metadata while OWL is used to define and instantiate Web ontologies, that is, conceptualizations of a specific domain. In this respect, the Semantic Web initiative exploits ideas from the (re-born) field of knowledge engineering using knowledge representation languages such as `DL` (Description Logics) [6], which provide theoretical and practical tools for the representation of ontologies.

Following this direction, several initiatives (e.g. `WSMO` [14], `OWL-S` [23], `SAWSDL` [20], `WSDL-S` [1]) provided a semantic description of services. The most prominent efforts focus on extending the WSDL schema introducing special tags, responsible to semantic mark-up the description of a service. Nevertheless, these approaches force the adoption of their particular extensions of the WSDL specification through ad-hoc semantic representations. Currently, in order to merge WSDL information with semantic models, the World Wide Web Consortium (W3C) proposed an RDF mapping from the XML-based WSDL documents [18, 19], where a semi-structured WSDL representation based on RDF triples is given. However the use of RDF model, as it is, provides relevant issues of efficient data management (i.e. storage and scalability).

With respect to this work, in this paper we propose a semantic description of WSDL 2.0 based on a metamodel [13] representation of semantic models (i.e. `RDF`, `RDF(S)`, `OWL-Lite` and `OWL-DL`). We exploit `Nyaya` [13], a tool for the efficient management of Semantic Web data, which provides advanced and up-to-date ontology reasoning capabilities based on Datalog$^{\pm}$ [11], a Datalog-like formalism that allows querying and reasoning over ontologies. Our storage system is specifically suited to work with huge RDF and OWL documents. We address the issues of storing and querying Web services by exploiting a metamodel technique. Differently from other approaches in the literature, which

[1] Oracle Chairman and CEO

provide implementation solutions, we follow a process that starts with the definition of a meta-representation of the chosen data model at a conceptual level. We select a fragment of our metamodel and we map the WSDL description into that fragment. The next step is a logical translation for a relational database. Eventually, at the physical level, we choose optimization techniques based on indexing and partitioning. Because of the storage model foundations, based on the metamodel approach, the system can easily represent structural aspects of complex models like RDF(S) and OWL. Therefore we can introduce any semantic annotations, assigned to the Web service and expressed by a Semantic Model, through the same process. Finally, we use Datalog$^{\pm}$ both to query and reason over our data.

The paper is organized as follows. Section 2 introduces a motivating scenario and a running example. Section 3 illustrates the state of the art about semantic representation of a Web Service description. Section 4 provides an architecture of reference for our process. Section 5 briefly shows the metamodel, describes in details the mapping between WSDL and a fragment of our metamodel, and introduces the rule based system to query Web services. Finally Section 6 presents some experiments, and Section 7 sketches conclusions and future works.

2 A Motivating Scenario

The semantic markup of Web services enables the automation of different tasks, in particular service discovery, and composition and interoperation.
Automatic Web service discovery automatically locates Web services that provide a particular service and matches properties required by the user. Currently this process is semi-automatic: firstly a human has to use a search engine to find the possible service and then verifies whether it fits the constraints given (i.e. reading the resulting Web page or executing the service itself). Through association of semantics, information necessary to service discovery is machine-interpretable and the entire process is achieved in an automatic way by the search engine. In this context efficiency and effectiveness exploit semantic markup.
Automatic Web service composition and interoperation automatically select, compose and support interoperation of Web services to support complex task objectives. Currently, the composition of services has to be driven by the selection of services from the user. Then the user has to compose manually the composition, ensuring the interoperation and providing the input at intermediate steps of the process (e.g. the correct bus for a landing flight). Also in this case, association of semantics to Web services can support the automation of the process.

In this framework, these two tasks are not entirely realizable with respect to current Web technology. It is due to the lack of Semantic markup and high heterogeneity of representation for markup. From academic research on Web, many efforts focus on Semantic representation of Web service descriptions. In particular last solutions focus on providing mapping between WSDL description and RDF representation.

Let's consider the following running example

```
<service name="reservationService"
      interface="tns:reservationInterface">
      <endpoint name="reservationEndpoint"
```

```
            binding="tns:reservationSOAPBinding"
            address ="http://greath.example.com/2004/reservation"/>
</service>
```

It is a fragment from a WSDL description of a reservation service. W3C individuated principal *elements* occurring in a WSDL schema and mapped them into Resources and Properties of RDF model. This mapping [19] introduces a mechanism to generate the URI references of each resource. Briefly, it is composed by three parts: (i) the namespace of the document, (ii) information about the considered component (e.g. an operation, a message and so on), and (iii) the path to reach the component following the XPointer syntax [12]. For instance we can represent the previous fragment into RDF, using the N-TRIPLE syntax, as follows

```
<URIService> rdf:type    <http://www.w3.org/ns/wsdl-rdf#Service>;
<URIService> rdfs:label "reservationService";
<URIService> wsdl:endpoint <URIEndpoint>
<URIEndpoint> wsdl:implements <URIInterface>
<URIInterface> rdf:type <http://www.w3.org/ns/wsdl-rdf#Interface>
<URIEndpoint>  rdf:type     <http://www.w3.org/ns/wsdl-rdf#Endpoint>;
<URIEndpoint>  rdfs:label  "reservationEndpoint";
<URIEndpoint> wsdl:address <http://greath.example.com/2004/reservation>;
<URIEndpoint> wsdl:usesBinding <URIBinding>
<URIBinding> rdf:type <http://www.w3.org/ns/wsdl-rdf#Binding>
```

Therefore the natural storage model followed by RDF is a three column table (*Subject*, *Property*, *Object*) as follows

Subject	**Property**	**Object**
URIService	rdf:type	`http://www.w3.org/ns/wsdl-rdf#Service`
URIService	rdfs:label	*"reservationService"*
...	...	...
URIBinding	rdf:type	`http://www.w3.org/ns/wsdl-rdf#Binding`

The visible inefficiency of this representation presents relevant issues of scalability and management capabilities.

3 State of the art

The most relevant solutions to semantic markup of Web services are in *OWL-based Web Service Ontology* (`OWL-S`), *Web Services Modeling Ontology* (`WSMO`) and *Web Service Semantics* (`WSDL-S`). The OWL-S and WSMO introduce an ad-hoc rich semantic model to describe Web services, while WSDL-S defines semantic annotations in an external domain model, preserving the information already present in the WSDL. Moreover, an alternative direction is to provide a mapping between RDF and WSDL, enabling the ability to read WSDL documents as RDF input.

OWL-S. OWL-S [23] defines an own ontology on top of OWL. This upper level ontology describes a minimal set of semantic markup language constructs describing main properties and capabilities of a Web service. The description is in terms of how the service can interact with the user (*service grounding*), how the service has to be used (*service model*) and how the service can be discovered (*service profile*). Therefore each instance of a Semantic Web service is described by an Individual of type *Service*, having properties *presents*, *describedBy* and *supports* pointing to Individuals of types *ServiceProfile*, *ServiceModel* and *ServiceGrounding* respectively. OWL-S uses SWRL [17] to process service content.

WSMO. WSMO [14] provides a conceptual model for Semantic Web Services. It is based on an ontology described in a formal language `WSML` [22] and implemented into an environment `WSMX` [16]. WSMO defines an ontology based on *Goals*, representing user desires (constraints) to match by executing a service, and *Mediators*, describing interoperability connections between WSMO elements. The Mediation crosses three levels: (i) Data Level (interoperability between data sources), (ii) Protocol Level (interoperability between communication patterns), and (iii) Process Level (interoperability between Business processes). As OWL-S, WSMO uses a rule based system supported by Description Logics and Logic Programming.

WSDL-S. Differently from OWL-S and WSMO, WSDL-S [1] defines annotations about capabilities and properties of Web services, described in WSDL, in an external domain model. WSDL-S exploits the extensibility of WSDL schema in terms of elements and attributes. For instance referring to the running example of Section 2, WSDL-S introduces a reference to an external ontology (i.e. we can have more than one ontology of reference) to annotate the associated concept, as follows.

```
<service name="reservationService"
      interface="tns:reservationInterface"
      wssem:modelReference="ReferenceOntology#Reservation">
    ...
</service>
```

In this WSDL-S document the service `"reservationService"` is annotated by `wssem:modelReference="ReferenceOntology#Reservation"` indicating that it is an instance of the concept `Reservation` with respect to a `ReferenceOntology`. The association of this annotation supports the automation of Web services tasks such as discovery or composition.

The creation of a WSDL-S document is semi-automatic. Gomadam et al. [15] proposed a tool allowing user to annotate manually existing WSDL files. Certainly, WSDL-S provides several advantages. Given the use of WSDL standard, existing frameworks can introduce WSDL-S annotation quickly. Developers can build different domain models by using preferred language (not necessary OWL, RDF or RDFS) or reuse existing domain models.

RDF Mapping. Since all above approaches force the adoption of particular extensions of the WSDL specification through ad-hoc semantic representations, Kopecky [18, 19]

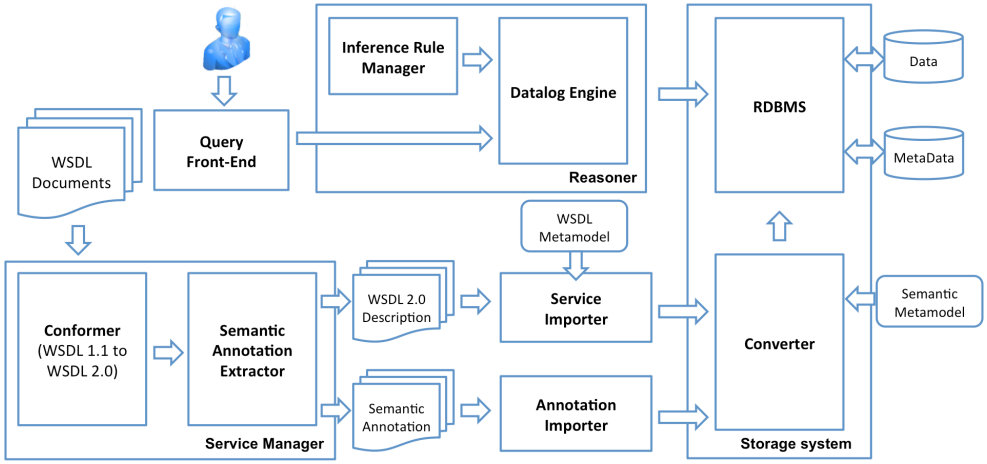


Figure 1: The basic architecture of the system

defines a representation of WSDL 2.0 into RDF and OWL models, and illustrates a mapping procedure for transforming particular WSDL descriptions into their RDF form. As introduced in Section 2, this mapping provides a generation procedure of URIs to assign to resources occurring into the RDF representation. Since this representation is independent of WSDL, it is extensible and supports any mechanism to annotate Web services through a uniform language to precess service content. Moreover interoperability is achieved by merging different Semantic Web data and enabling the ability to read WSDL documents as RDF input.

4 An Architecture of reference

A flexible architecture of our system is shown in Figure 1. It serves as a logical view of how the system looks like. We extended Nyaya, introducing components to process WSDL documents. A typical use scenario of the system follows:

- A collection of WSDL documents is captured by the *Service Manager* (SM). These documents could be expressed in WSDL 1.1 or WSDL 2.0. Therefore SM presents a *Conformer* to standardize the WSDL 1.1 representation into 2.0. This component exploits the converter available at `http://www.w3.org/2006/02/WSDLConvert.html`. Then a *Semantic Annotation Extractor* extracts the semantic annotation associated to the service description. We assume that each WSDL document presents a semantic annotation, embedded into the description or in an external document. We consider these annotations expressed in OWL.

- A *Service Importer* (SI) and an *Annotation Importer*(AI) parse respectively the WSDL 2.0 document and the corresponding semantic annotation, coming from SM. The former extracts elements and attributes from the service description using the SAX parser[2] and a metamodel describing the *constructs* of interest in the WSDL schema, as will be shown in Section 5. The latter extracts the TBox (terminological) axioms and the ABox (assertional) facts.

[2] `http://www.saxproject.org/`

- The *Storage System* populates the back-end database with the incoming WSDL elements and axioms and facts. All the elements are stored in a relational database according to a metamodel-based organization [4], sketched in Figure 2. This metamodel is made of a set of primitives, each of which properly represents a construct of a Semantic Model (e.g RDF, RDFS, OWL). Thus, each primitive is used to properly gather elements of the knowledge source having the same semantics. A relational DBMS is used to actually store all the imported elements. The main advantage of this storage model is that different and possibly heterogeneous formats of semantic data (such as the various dialects of OWL and other representation languages) can be managed. The *Converter* is in charge of associating each construct of the source model with the corresponding primitive of the metamodel and storing the elements accordingly. Therefore we can select different fragments of the metamodel corresponding to different semantic representations of a service (e.g RDF, RDFS or OWL).

- The *Reasoner* is composed by a rule-based engine and a basic set of inference rules defined by means of a Rule Manager that dictate how new information can be inferred from ground facts and ontological domain knowledge. More specifically, the engine embeds a Datalog-based reasoning engine running a basic program that specifies how to infer new elements of both basic classes and properties according to assertions and constraints defined over them (e.g., subclass hierarchies between classes and properties, transitivity and symmetry relationships, restrictions and so on). This program is also written in Datalog$^{\pm}$ and changes with the specific ontological language adopted.

- The user poses queries to the system through a *front-end* user interface. The idea here is that different users with different expertise can specify simple queries in different languages: the final user can take advantage from a high level user interface in which conjunctive queries are expressed in a Query-By-Example fashion, whereas an advanced user can formulate conjunctive queries in the rule-based language. All the queries are delivered to the Reasoner that, relying on the inference rules, generates the queries to be actually executed by the storage system to retrieve all the elements included in the answer of the original query.

A relevant feature of this approach is that it is easily extensible: new primitives can be easily added to the metamodel-based storage model to incorporate more advanced features as well as new formalisms for the representation of the semantics of data sources.

The rest of the paper is devoted to a more detailed description of the semantic representation of a WSDL document and the efficient and scalable processing of WSDL content through advanced and powerful reasoning capabilities.

5 Management of Semantic Web Services

5.1 A Metamodel Approach

The storage system of Nyaya is characterized by a three steps process as follows:

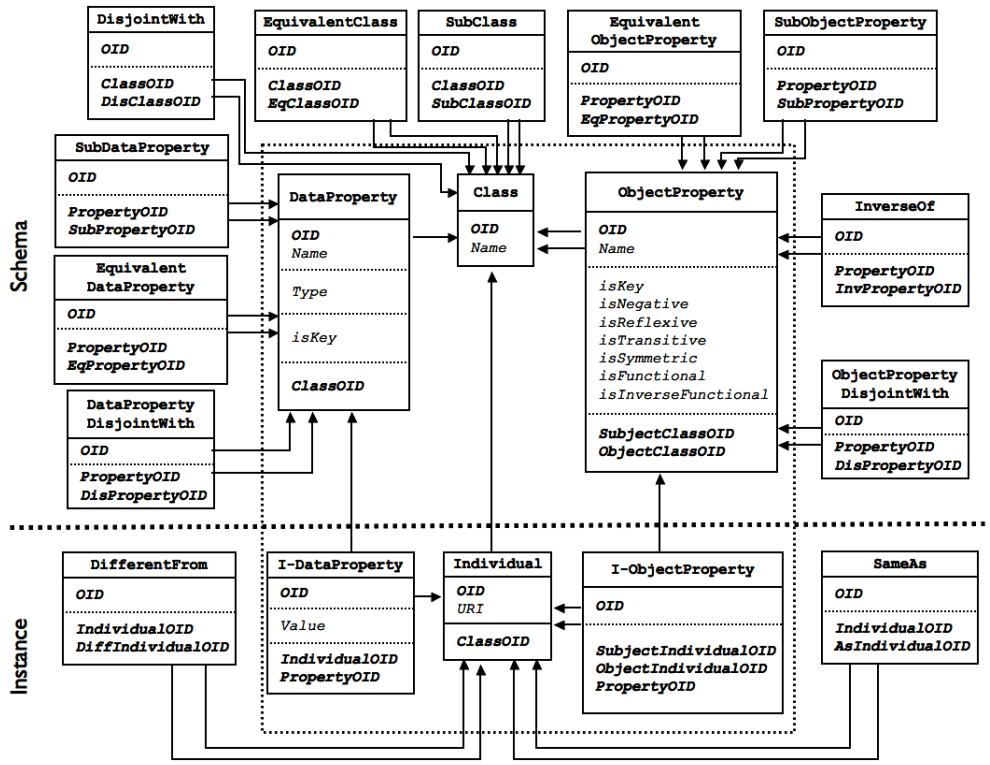


Figure 2: The Metamodel core

- a *conceptual level*, proposing a simple conceptual model where a set of constructs properly represents semantic concepts. Each construct is used to properly represent elements of documents, with the same semantics;
- a *logical level*, implementing our conceptual model into a logical one. In our case we use the relational model;
- a *physical level*, defining the physical design of the logical representation of previous level. There, we illustrate a special partitioning technique that relevantly increases the performance of the entire process.

Conceptual level. Our approach is inspired by works of Atzeni et al. [4, 5] that propose a framework for the management of heterogeneous data models in an uniform way. They leverage on the concept of metamodel that allows a high level description of models by means of a generic set of constructs. Formally, a model can be represented as $M = \{C_1, C_2, \ldots, C_n\}$. C_i's are constructs, that have the following structure: $C = (OID, attr_1, \ldots, attr_f, ref_1, \ldots, ref_m)$ where OID is the object identifier, the $attr_j$'s are the properties of the construct and the ref_k's are the references of the construct with others.

Following this idea, we propose a simple model where a set of constructs properly represent concepts expressible with Semantic Models. A difference with respect to the approach of Atzeni et al. is that they consider a well marked distinction between the schema level and the instances while, for our purposes, we need to manage at the same time schemes and instances.

At schema level we distinguish three main constructs: *Class*, *ObjectProperty* and *DataProperty*. The concept of Class is clearly defined while ObjectProperty and DataProperty represent binary relations where the former is a relation between Classes and the latter a relation between a Class and a literal with a primitive type value. At instance level three more constructs are introduced to represent instances, namely: *Individual* (resources, with URI), *i-ObjectProperty* and *i-DataProperty* (with value).

Constructs of the instance level have a reference to the constructs of schema level they correspond to and inherit from them the same references, toward instance level constructs corresponding to schema level constructs pointed by those references. Formally we define

$$\begin{aligned} M_{basic} &= \{C_{Class}, C_{DataProperty}, C_{ObjectProperty}, C_{Individual}, C_{i-DataProperty}, \\ &\quad C_{i-ObjectProperty}\} \end{aligned}$$

where the constructs have the following structure:

$$\begin{aligned} C_{Class} &= (OID, Name) \\ C_{DataProperty} &= (OID, Name, Type, isKey, ClassOID) \\ C_{ObjectProperty} &= (OID, Name, IsKey, isNegative, isTransitive, isSymmetric, \\ &\quad isFunctional, isInverseFunctional, SubjectClassOID, \\ &\quad ObjectClassOID) \\ C_{Individual} &= (OID, URI, ClassOID) \\ C_{i-DataProperty} &= (OID, Name, Value, IndividualOID, PropertyOID) \\ C_{i-ObjectProperty} &= (OID, Name, SubjectIndividualOID, ObjectIndividualOID, \\ &\quad PropertyOID) \end{aligned}$$

Figure 2 sketches an UML-like diagram of our metamodel, where the dashed line divide schema level from instance level. Enclosed in the dashed box, there is the core M_{basic} of our model.

We remark a strong point of the approach: extendibility. If one discovers that metamodel is not detailed enough (i.e. expressive enough) for his purposes, he can overcome such mismatch just adding constructs with references and properties and/or adding new properties to existing ones. For instance, this simple model can be extended in order to manage also subclasses and subproperties. At schema level it suffices to add three new constructs, namely *SubClass*, *SubDataProperty* and *SubObjectProperty*, each one with two references toward *Class* and *Properties*, respectively: for *Subclass*, the first one (*SubClassOID*) references the class that is a subclass of the class referenced by the second one (*ClassOID*); the use of references is analogous for subproperties (*SubPropertyOID*, *PropertyOID*). Following, we can represent Collections (e.g. Intersections, Unions, ...), Restrictions and so on. For sake of lackness of space, Figure 2 presents a subset of our metamodel.

Logical Level. We use a relational implementation of our conceptual model. Basically, the schema of the underlying database includes a table for each construct of the metamodel. For each field of a construct we add a column to the table corresponding to such

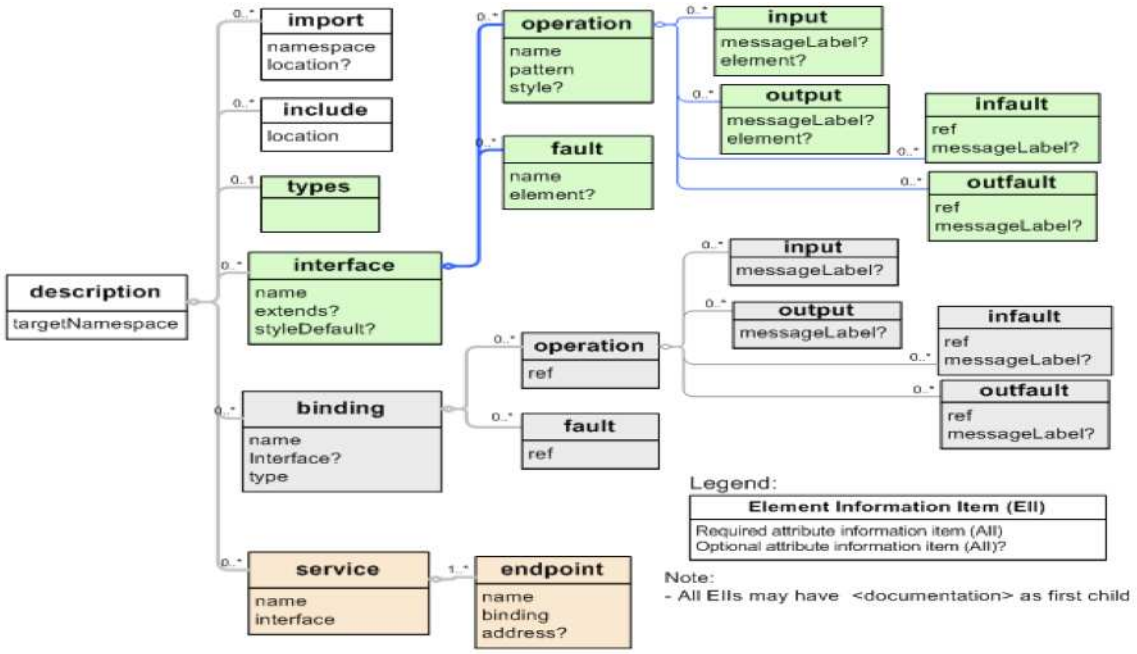


Figure 3: The WSDL Metamodel

construct. The OID attribute is the primary key for each table, and we add an integrity constraint for each reference, from the pointer construct to the pointed one (i.e. from the column corresponding to the reference field toward the OID column of the referenced construct). Notice that, since we exploit different levels of abstraction, the logical level could be implemented also by an object oriented model.

Physical level. The storage model is supported by special optimization methods that increase the performance of the system in accessing data and answering queries. These methods guarantee that only the data that is strictly needed is actually accessed.

The resulting tables of logical level could be very large. To this aim we use a partitioning technique referring to splitting what is logically one large table into smaller physical pieces. The partitioning is based on table *inheritance*. Each partition represents a child table of a single parent table. Normally the parent table itself is empty; it exists just to represent the entire data set. The child table inherits the structure of the parent (i.e. attributes). The partitioning of a table is processed by the *range* defined on it. In other words the table is divided into different partitions defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions.

In details we set up a partitioned table by following steps:

1. We create the parent table, from which all of the partitions will inherit.

2. The parent table will contain no data. We can define several constraints on this table (e.g. key, foreign key and so on) to be applied equally to all partitions.

3. We choose the range from the parent table (a single attribute or a set). Then based on this range, we create several child tables that inherit the structure (i.e. attributes and constraints) from the parent table. Normally, these tables will not add any columns to the set inherited from the parent.

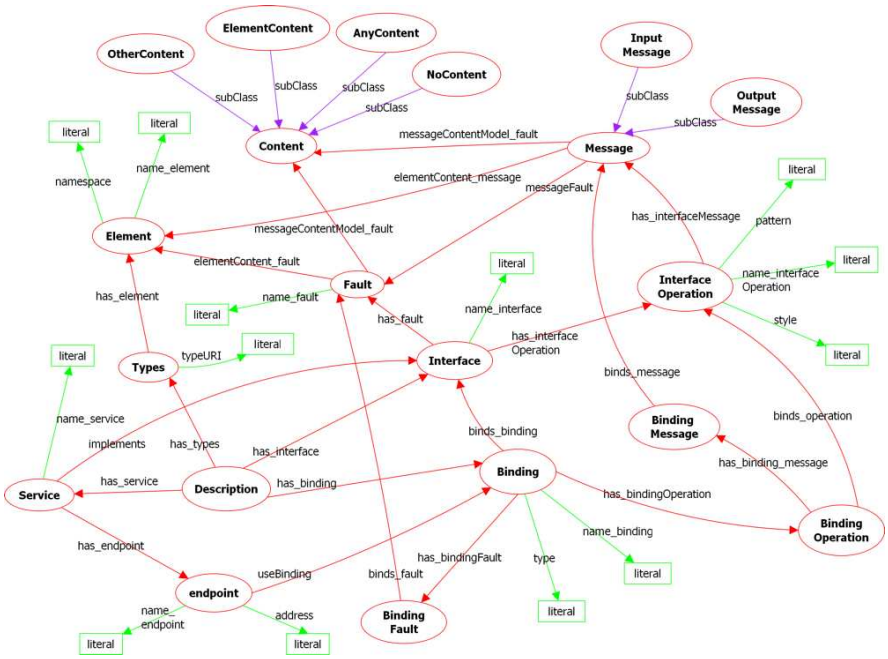


Figure 4: Semantic representation of WSDL Schema Elements

4. We add table's *check* constraints to the child tables to define the allowed key values in each partition.

5. We create an index on the key column(s) in each partition (as well as any other indexes we might want).

6. Finally we define a trigger or rule to redirect data inserted into the parent table to the appropriate partition.

7. Optionally we can iterate the partitioning on the resulting child tables.

In our case let us consider the following ranges:

- $C_{i-ObjectProperty}$: the key column is the $PropertyOID$ attribute. It redirects data into respective partition respect to a unique value. Then we can create indexes on the $SubjectIndividualOID$ and $ObjectIndividualOID$ attributes.

- $C_{i-DataProperty}$: the key column is the $PropertyOID$ attribute and we can create indexes on the $IndividualOID$ and $Value$ attributes.

- $C_{Individual}$. the key column is the $ClassOID$ attribute.

5.2 Importing of WSDL documents

To import a WSDL document into our internal organization, we have to characterize the set of basic constructs to describe a WSDL document, that we will call *WSDL metamodel*, to select a fragment of the metamodel illustrated above and define a mapping between

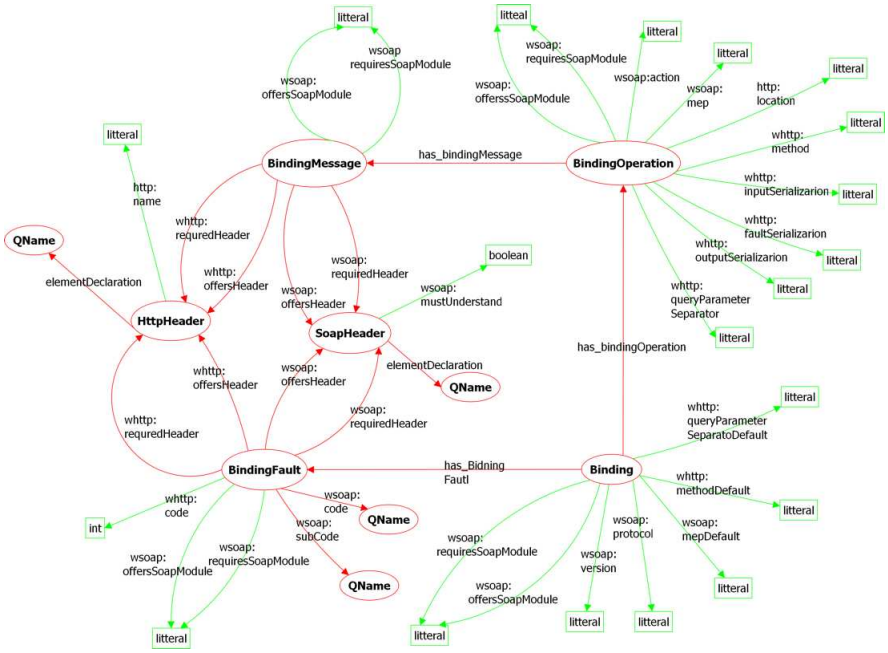


Figure 5: Semantic Representation of WSDL Schema Binding

the two collections of constructs. In Figure 3 an UML diagram of the WSDL Metamodel is represented. Formally we define

$$M_{WSDL-basic} = \{C_{description}, C_{types}, C_{interface}, C_{binding}, C_{service}\}$$

$M_{WSDL-basic}$ represents the core set of constructs to describe a WSDL document. The main component is $C_{description}$ having the attribute *targetNamespace* to identify the WSDL components occurring in the document. $C_{description}$ is related to four top-level elements. C_{types} describes the format of messages used by the service as input and output. $C_{interface}$ describes the abstract functionalities of the service. $C_{binding}$ specifies the implementation details necessary to access the service, that is a concrete message format and communication protocols of the interface for each operation. Finally $C_{service}$ describes where the service can be accessed. Since the extensibility of the representation, we can enrich the model in M_{WSDL} adding other constructs such as $C_{operation}$, $C_{message}$ and so on. Then we select the following fragment of our metamodel:

$$\begin{aligned} M_{fragment} = \{&C_{Class}, C_{DataProperty}, C_{ObjectProperty}, C_{SubClass}, \\ &C_{Individual}, C_{i-DataProperty}, C_{i-ObjectProperty}\} \end{aligned}$$

Therefore the semantic representation of a WSDL document results from the mapping between M_{WSDL} and $M_{fragment}$. Figure 4 shows the resulting semantic representation of WSDL Elements at schema level. We used a graph representation where nodes correspond to C_{Class}, red edges to $C_{ObjectProperty}$, green edges to $C_{DataProperty}$ and purple edges to subclass relations between classes. Figure 5 illustrates the resulting semantic representation of WSDL Binding Elements at schema level following the same legend.

Let's consider the running example of Section 2. Figure 6 shows the resulting semantic representation, at conceptual level, according to the mapping above discussed. At logical level, we import the document at schema level as illustrated in Figure 7.

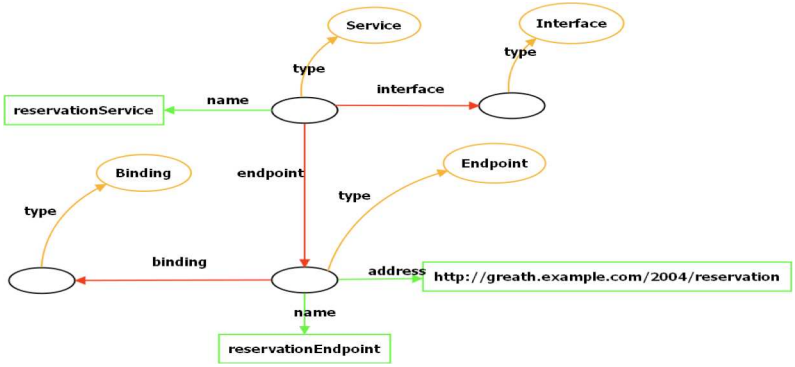
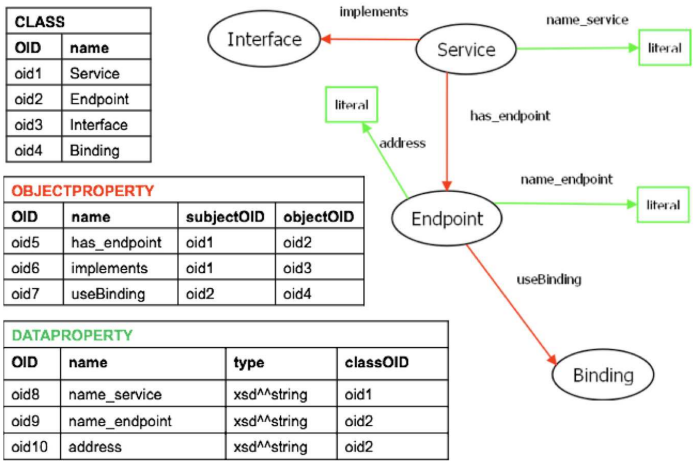


Figure 6: An example of Conceptual Modeling

In the right side the Figure shows the WSDL schema portion corresponding to the information contained in the document. In the left side the Figure provides the relational implementation of corresponding constructs in our metamodel. For the sake of notation simplicity, we omitted some attributes. Figure 8 describes the relational implementation at instance level. Each table presents the references (marked in red) to the corresponding entries at schema level. URIs assigned to entries of table `Individual` are generated by using the same procedure in [19]. In the Figure we used short names as *uriService* or *uriEndpoint* to simplify the notation.



CLASS

OID	name
oid1	Service
oid2	Endpoint
oid3	Interface
oid4	Binding

OBJECTPROPERTY

OID	name	subjectOID	objectOID
oid5	has_endpoint	oid1	oid2
oid6	implements	oid1	oid3
oid7	useBinding	oid2	oid4

DATAPROPERTY

OID	name	type	classOID
oid8	name_service	xsd^^string	oid1
oid9	name_endpoint	xsd^^string	oid2
oid10	address	xsd^^string	oid2

Figure 7: An example of Logical Modeling at Schema level

We remind the example WSDL-S document in Section 3. Figure 9 shows how the semantic annotation associated to the WSDL document is imported in Nyaya. At schema level we introduced one entry *Concept* in `Class` representing the abstract class of concept in an external ontology and one entry *modelReference* in `ObjectProperty`, representing the reference annotation between a WSDL component and a concept. In this case we had to introduce the entry *Thing* because the domain of *modelReference* is generic (unknown). At instance level we have two entries in `Individual` and `i-ObjectProperty` representing instances of *Concept* and *modelReference*. Finally Figure 10 shows the physical organization of the table `Individual` for instance. The optimization is based on the partitioning with respect to the attribute *ClassOID*. In this way we have a direct access to entries of table `Individual` representing instance of the same class (e.g. *Service* or *EndPoint*).

INDIVIDUAL		
OID	URI	ClassOID
ind1	uriService	oid1
ind2	uriEndpoint	oid2
ind3	uriInterface	oid3
ind4	uriBinding	oid4

I-OBJECTPROPERTY				
OID	name	subjectIndOID	objectIndOID	ObjectpPropertyOID
obj1	has_endpoint	ind1	ind2	oid5
obj2	implemets	ind1	ind3	oid6
obj3	useBinding	ind2	ind4	oid7

I-DATAPROPERTY				
OID	name	value	individualOID	DataPropertyOID
data1	name_service	reservationService	ind1	oid8
data2	name_endpoint	reservationEndpoinf	ind2	oid9
data3	address	http://greath.example.com/..	ind2	oid10

Figure 8: An example of Logical Modeling at Instance level

CLASS	
OID	name
oid0	Thing
oid1	Service
oid2	Endpoint
oid3	Interface
oid4	Binding
oid11	Concept

OBJECTPROPERTY			
OID	name	subjectOID	objectOID
oid5	has_endpoint	oid1	oid2
oid6	implements	oid1	oid3
oid7	useBinding	oid2	oid4
oid12	modelReference	oid0	oid11

INDIVIDUAL		
OID	URI	ClassOID
ind1	uriService	oid1
ind2	uriEndpoint	oid2
ind3	uriInterface	oid3
ind4	uriBinding	oid4
ind5	Reservation	oid11

I-OBJECTPROPERTY			
OID	subjectIndOID	objectIndOID	ObjectPropertyOID
obj1	ind1	ind2	oid5
obj2	ind1	ind3	oid6
obj3	ind2	ind4	oid7
obj4	ind1	ind5	oid12

Figure 9: Importing of Semantic Annotations

5.3 Three levels Datalog Programs

Nyaya allows access for both naive users, through a user-friendly interface, and expert users through Datalog$^{\pm}$ programs that use, as base predicates, classes and roles of the ontology of interest. To each concept naturally corresponds a unary predicate, and to each role a binary predicate. For instance, to the concepts *Service* and *Interface*, correspond the atoms $Service(X)$ and $Interface(Y)$, and to the role *implements* corresponds the predicate $implements(X, Y)$. These are called *first-level ontology predicates (FLOPs)*. Moreover, by means of the rules, users can define their own predicates, of any arity.

Initially, ontology instances are stored in the Nyaya database. The Datalog$^{\pm}$ program produces new atoms – among which there may be more ontology instances – which do not get materialized into the database. This is because for the moment we do not want to incur into view maintenance problems, and would rather let newly produced knowledge last only for the current program run. The user program thus defined is called *first-level Datalog$^{\pm}$ program*.

Queries are specified by coupling the first-level Datalog$^{\pm}$ program with a goal. Referring to our running example, Example 1 shows a user program.

Example 1 *First-level program*

q(X,Y) :- ReservationInterface(X), name_interface(X,Y)

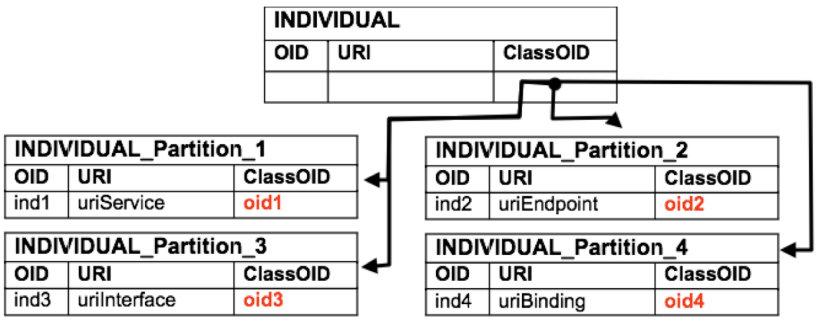


Figure 10: An example of Physical Modeling

ReservationInterface(Y) :- Service(X), implements(X,Y), name_service(X,'reservationService')

A first-level Datalog$^{\pm}$ programs is translated into a second-level Datalog$^{\pm}$ program which expresses the FLOPs in terms of the predicates (constructs) stored in the database, called *second-level ontology predicates (SLOPs)*. In these programs we adopt an extension of Datalog, expressing for each predicate the name of attributes, before values (constant or variable) and using the prefix `TMP` to indicate predicates corresponding to temporary knowledge.

Example 2 shows the second-level Datalog$^{\pm}$ program which is associated with the program of Example 1.

Example 2 *Second-level program*

```
TMP_RES(OID: X, Name: Y) :- i-DataProperty(IndividualOID: X, value: Y, PropertyOID: P3),
          DataProperty(OID: P3, name: 'name_interface'), TMP_ReservationInterface(OID: X)

TMP_ReservationInterface(OID: Y) :- Individual(OID: X, ClassOID: C1), Class(OID: C1, name: 'Service'),
           i-DataProperty(IndividualOID: X, value: 'reservationService', PropertyOID: P1),
           DataProperty(OID: P1, name: 'name_service'),
           i-ObjectProperty(SubjectIndividualOID: X, ObjectIndividualOID: Y, PropertyOID: P2),
           ObjectProperty(OID: P2, name: 'implements')
```

The final Datalog$^{\pm}$ program run by the Datalog$^{\pm}$ engine consists of the union of the first- and second-level Datalog$^{\pm}$ programs, along with the *reasoning-level program*, which provides the Nyaya reasoning capabilities. This represents the *third-level program* of our rule-based system. The *reasoning-level program* does not vary with the user program but changes across different ontological formalisms. Actually we implemented the inference rules for $\mathcal{EL}^{++}$ ontologies. For more details look at [13].

Let's consider a more complete WSDL document

```
<description
    xmlns="http://www.w3.org/ns/wsdl"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsdlx= "http://www.w3.org/ns/wsdl-extensions">
  <interface  name = "reservationInterface" >
    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
```

```
            wsdlx:safe = "true">
        <input messageLabel="In"
              element="ghns:checkAvailability" />
         ...
     </operation>
  </interface>
  <service name="reservationService"
       interface="tns:reservationInterface">
...
  </service>
  ...
</description>
```

In the following we show a more complex example of Datalog$^{\pm}$ program to discover the name of services with an interface providing an operation `opCheckAvailability` with an input `checkAvailability`.

```
TMP_IO(OID: Y1) :- Class(OID: X1, name: 'InterfaceOperation'), Individual(OID: Y1, classOID: X1),
                i-DataProperty(name: 'name', IndividualOID: Y1, value: 'opCheckAvailability'),
                Class(OID: X2, name: 'InputMessage'), Individual(OID: Y2, classOID: X2),
                Class(OID: X3, name: 'Element'), Individual(OID: Y3, classOID: X3),
                i-ObjectProperty(name: 'elementContent_Message', SubjectOID: X2, ObjectOID: X3),
                i-DataProperty(IndividualOID: Y3, value: 'checkAvailability', propertyOID: DP);

TMP_RES(serviceName: K3) :- Class(OID: K1, name: 'Service'), Individual(OID: K2, classOID: K1),
                 TMP_IO(OID: Y1), i-DataProperty(name: 'name', IndividualOID: K2, value: K3),
                 i-ObjectProperty(name: 'implements', SubjectOID: K2, ObjectOID: K6, propertyOID: K4),
                 i-ObjectProperty(name: 'has_InterfaceOperation', SubjectOID: K6, ObjectOID: Y1,
                                 propertyOID: K5);
```

6 Implementation

Several experiments have been done to evaluate the performance of our framework. In this section we relate on them.

6.1 Platform Environment

Our benchmarcking system is a dual quad core 2.66GHz Intel Xeon, running Linux Gentoo, with 8 GB of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array. We implemented our experiments using PostgreSQL 8.3. As Beckmann et al. [7] experimentally showed, it is relevantly more efficient respect with commercial database products. We used the default Postgres configuration: we preferred to no calibrate the system for specific needs.

SDB was tested on PostgreSQL. This implementation provides the main table `triples` containing three columns: s (i.e. subject), p (i.e. property) and o (i.e. object). It defines a primary key on the columns triple (s, p, o) and uses two unclustered B+ tree indices. One index is on (s, o), one is on (p, o). The table contains hashing values coming from URIs and property names, stored into the table `nodes`.

Sesame was tested with stable 2.2.4 version on a PostgreSQL store that utilizes B+tree indices on any combination of subjects, properties, and objects. We will show that scalability on Sesame is still an issue as it must perform many self-joins like all triple-stores.

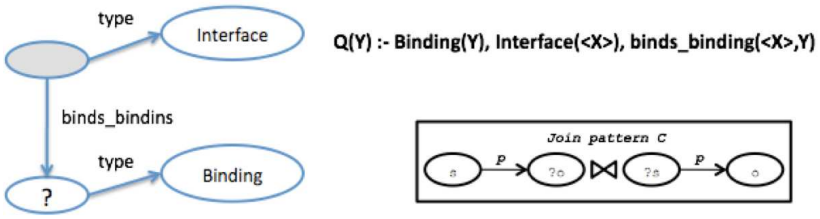


Figure 11: Query Example Q1

Our storage system exploits the native partitioning technique of PostgreSQL. We use the M_{basic} set of constructs as described in Section **??**. At instance level, on the `Individual` table (and relative partitions), we used a constraint of unique value on the *URI* attribute. On the `i-DataProperty` and `i-ObjectProperty` tables we used an unclustered B+ tree index on the *PropertyOID* attribute and for each partitions we used clustered B+ tree indexes on the *SubjectIndividualOID* and *IndividualOID* attributes, and unclustered B+ tree indexes on the *ObjectIndividualOID* and *Value* attributes. Each partition has a check constraint and a trigger to redirect data inserted into the parent table and query processing into the children. In our implementation we generate OIDs and translated textual data (e.g *value* of i-DataProperty) by MD5[3] coding to gain more performance.

6.2 Benchmark Environment

We used the public available *QWS Dataset* [2, 3]. The public dataset consists of 5,000 Web services of which 365 services are real implementations that exist on the Web today. The goal of our experiments is the comparison between our methodology and the RDF triple storage system. We used the RDF mapping in [19] and imported the resulting RDF representations of QWS dataset into SDB Jena [26] and Sesame [10] implementations. The resulting RDF dataset is 1.5 million triples.

Most of the common RDF query languages are based on *triple query pattern matching*. A pattern is a triple $\langle subject, property/predicate, object\rangle$ $\langle s, p, o\rangle$ where any s, p or o can be constant or variable (denoted, in the latter case, by $?s$, $?p$ and $?o$). We can combine these basic patterns into more complex ones through join conditions. As in [25] we identify three different *join patterns*, denoted by A, B, and C.

We implemented three representative queries, resulting from different combinations of A, B, and C (C; A, C; B, C). For SDB Jena and Sesame we expressed the queries in SPARQL, while for Nyaya in first-level Datalog$^{\pm}$ programs. More in detail the full queries are described in the following.

Q1. This query returns all Bindings associated to a certain Interface. In other terms It returns URIs of Individuals of type Binding associated to an Individual of type Interface with a particular URI. Figure 11 depicts Q1 by a graph representation. At the right side of the Figure it is described the first-level Datalog$^{\pm}$ program, and at the bottom side we indicate the occurring join patterns. In the program we indicate with $< X >$ a value to take as input. In this case we provide three different URIs as shown in Table 1. In

[3] `http://tools.ietf.org/html/rfc1321`

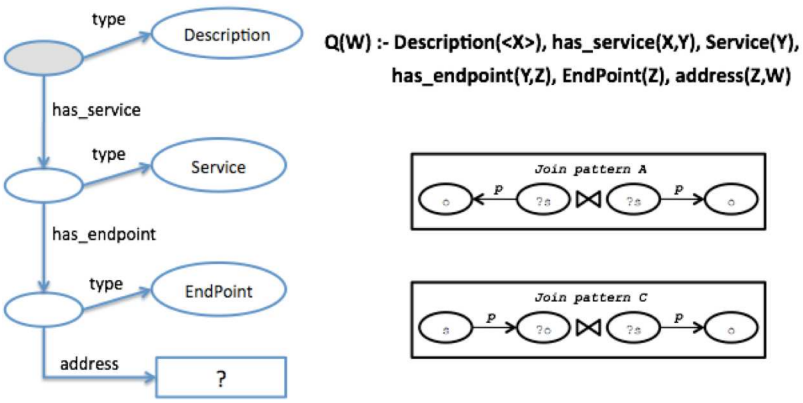


Figure 12: Query Example Q2

the graph $< X >$ corresponds to the grey circle. Q1 uses the join pattern C (i.e. a join condition between subjects and objects). Here we support the triple pattern $(s, p, ?o)$.

Q2. This query provides address of the EndPoint for a Service with a certain Description. As for Q1, Figure 12 describes Q2 using a graph representation, a first-level Datalog$^{\pm}$ program and the occurring join patterns. Q2 uses a combination between the join patterns A (i.e. a join between subjects) and C and presents the triple pattern $(?s, p, ?o)$.

Q3. This query searches Interface Operations that can be composed. In other terms Q3 discovers Interface Operations where one Interface presents the content (Element) of the outMessagge equal to the content of the inputMessage of another Interface. The query uses a combination between the join patterns B (i.e. a join between objects) and C and provides the triple patterns $(?s, p, o)$ and $(s, p, ?o)$.

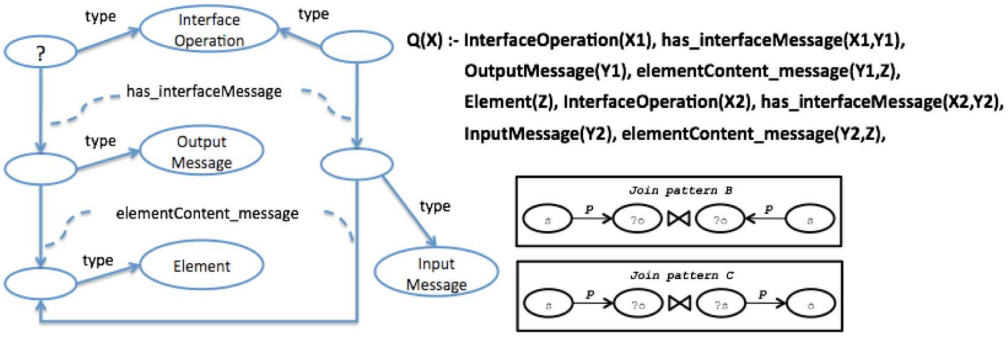


Figure 13: Query Example Q3

6.3 Performance Results

The three queries have been tested on all systems. Q1 and Q2 implement a service discovery task, while Q3 a composition and interoperation task. We report the number of resulting records (#Res) and response times, expressed in seconds, for each system. Each time is the average of three runs of the query. Internal database cache and operating

system cache can cause false measurements then before each query execution we have restarted the database and cleaned any cache.

In the following we will explain the obtained results for the execution of each query.

Q1. Q1 shows that the expensive object-subject joins of the triple-store approach is crucial since it performs much more slowly than our system, as shown in Table 1. SDB and Jena has to operate expensive joins (i.e. self and merge joins respectively) between huge amount of data. The respective optimizations do not support that systems. Nyaya exhibits the effectiveness of the construct-based approach, supported by the partitioning optimization. Nyaya accesses directly to the data portion of interest (i.e. the partitions corresponding to *Interface*, *Binding* and *binds_binding*).

Table 1: Q1 Response Times

URI	**# Res**	**Jena**	**Sesame**	Nyaya
`http://www.strikeiron.com#wsdl.interface(SDPCensusSoap)`	1	0.24	0.579	0.08
`http://tempuri.org/#wsdl.interface(serviceSoap)`	2	0.31	0.59	0.11
`http://intelinet.no/idresolver#wsdl.interface(IdResolverSoap)`	2	0.35	0.61	0.1

Q2. Q2 performs join conditions subject-subject and object-subject at the same time. This makes computationally complex the query execution (i.e. may be the most complex computation). Table 2 reports the response times at different URI input for Description. SDB Jena shows dramatic response times due to reasons expressed above. Sesame exploits indices on any combination of subjects, properties, and objects. Nyaya confirms the efficiency of the approach reducing relevantly the query search space and join complexity through the partitioning on *EndPoint*, *Service* and *Description*.

Table 2: Q2 Response Times

URI	**# Res**	**Jena**	**Sesame**	Nyaya
`http://tempuri.org/#wsdl.description()`	221	6.3	0.05	0.006
`http://www.strikeiron.com#wsdl.description()`	46	4.22	0.011	0.0012
`http://www.ebi.ac.uk/WSClustalW#wsdl.description()`	2	1.67	0.011	0.001

Q3. Q3 query performs join conditions object-object and subject-object presenting similar join complexity with respect to Q2. Table 3 shows the resulting response times. As in Q2, SDB Jena can not exploit the property table technique while Sesame is not well supported by indexing. Again Nyaya exploits the construct-based approach and it is supported by the partitioning optimization.

Summarizing, the triple-store approach presents relevant performing issue. SDB and Sesame demonstrated its unprofitable storage model. The results of Nyaya are better in

Table 3: Q3 Response Times

# Res	Jena	Sesame	Nyaya
26	2.3	0.4	0.001

the most of cases due to its internal data organization at each level of abstractions (i.e. construct-based aggregations and partitioning).

7 Conclusion and Future Work

Generating a semantic representation of Web services to realize the vision of a Semantic Web of services has received a lot of attention in the recent years. Since WSDL covers only syntactic functional description, while the semantics of Web Services is not covered by any specification, several initiatives (e.g. `WSMO`, `OWL-S`, `SAWSDL`, `WSDL-S`) provided a semantic description of services by extending the WSDL document through complex semantic models. Nevertheless these approaches force the adoption of particular ad-hoc constructs to extend the WSDL specifications. In particular, WSDL-S proposal underlines the necessity of providing a semantic representation that is a slight extension of the WSDL specification, a language that the developer community is familiar with, to be compliant with existing service repositories, and of the semantic domain models, to take an agnostic approach to ontology representation languages. Currently, in order to merge WSDL information with semantic models, the World Wide Web Consortium (W3C) proposed an RDF mapping from the XML-based WSDL documents[4], where a semi-structured WSDL representation based on RDF triples is given. However data must be rapidly accessed in order to be effectively used. Then Storing and maintaining these data represent crucial activities to achieve this complex purpose. The classical "triple-store" approaches are not good enough because most of the queries require a high number of self-joins on the triples table.

In order to overcome these problems we proposed a metamodel-based approach to describe the semantics associated to a Web service. and discussed Nyaya to store and maintain large amount of Semantic Web Services. Experimental results demonstrated the effectiveness and efficiency of the proposal. Future directions provide for the implementation of advanced service discovery algorithms (such as the one presented in [9]) on the existing metamodel representation. Through sets of Datalog$^{\pm}$ programs we retrieve suitable services for a given service request. In this way, capabilities of semantic models are merged together with well-known, efficient techniques of database management systems and Datalog query engines.

References

[1] R. Akkiraju et al. Web Service Semantics - WSDL-S. Technical Report at `http://www.w3.org/Submission/WSDL-S/`, 2005.

[4] `http://www.w3.org/TR/wsdl20-rdf/`

[2] E. Al-Masri, and Q. H. Mahmoud. Discovering the best Web service (poster). In *Proc. of the 16th International Conference on World Wide Web (WWW'07)*, 2007.

[3] E. Al-Masri, and Q. H. Mahmoud. QoS-based Discovery and Ranking of Web Services. In *Proc. of the 16th International Conference on Computer Communications and Networks (ICCCN'07)*, 2007.

[4] P. Atzeni and R. Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Information Systems*, 18(6): 349–362, 1998.

[5] P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 17(6): 1347–1370, 2008.

[6] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[7] J. Beckmann, A. Halverson, R. Krishnamurthy, and J. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In *Proc. of the 22rd Int. Conference on Data Engineering (ICDE'06), Atlanta, USA*, 2006.

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

[9] D. Bianchini, V. De Antonellis and M. Melchiori. Flexible Semantic-Based Service Matchmaking and Discovery. In *World Wide Web Journal, 11(2): 227-251*, 2008.

[10] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. of the first Internation Conference on Semantic Web (ISWC'02), Sardinia, Italy*, 2002.

[11] A. Calì, G. Gottlob, and T. Lukasiewicz. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. In *Proc. of the 28th ACM Symp. on Principles of Database Systems (PODS'09)*, 2009.

[12] S. De Rose et al. XML Pointer Language (XPointer). Technical Report at `http://www.w3.org/TR/xptr/`, 2002.

[13] R. De Virgilio, G. Orsi, L. Tanca and R. Torlone. Reasoning over Large Semantic Datasets. Technical Report, RT-DIA-149, Università Roma Tre, 2009.

[14] D. Fensel, C. Bussler. The Web Service Modeling Framework WSMF. In *Electronic Commerce Research and Applications (ECRA) 1(2):113-137*, 2002.

[15] K. Gomadam, K. Verma, D. Brewer, A. Sheth, J. Miller. Radiant: A tool for semantic annotation of Web Services. In *Proc. of 4th International Semantic Web Conference (ISWC'05), Osaka, Japan*, 2005.

[16] A. Haller, E. Cimpian, A. Mocan, E. Oren, C. Bussler. WSMX - A Semantic Service-Oriented Architecture. In *Proc. of International Conference on Web Services (ICWS'05), Orlando, FL, USA*, 2005.

[17] I. Horrocks et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical Report at `http://www.w3.org/Submission/SWRL/`, 2004.

[18] J. Kopecky. WSDL RDF Mapping: Developing Ontologies from Standardized XML Languages. In *Proc. of the 1st International ER Workshop on Ontologizing Industrial Standards (OIS'06), Tucson, AZ, USA*, 2006.

[19] J. Kopecky. Web Services Description Language (WSDL) Version 2.0: RDF Mapping. Technical Report at `http://www.w3.org/TR/wsdl20-rdf/`, 2007.

[20] J. Kopecky, T. Vitvar, C. Bournez, J. Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. In *IEEE Internet Computing, 11(6): 60–67*, 2007.

[21] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax Internet document: `http://www.w3.org/TR/rdf-concepts/`.

[22] H. Lausen, J. de Bruijn, A. Polleres, D. Fensel. WSML - a Language Framework for Semantic Web Services. In *W3C Workshop on Rule Languages for Interoperability, Washington, DC, USA*, 2005.

[23] D. Martin et al. OWL-S: Semantic Markup for Web Services. Technical Report at `http://www.w3.org/Submission/OWL-S/`, 2004.

[24] D. L. McGuinness and F. van Harmelen OWL Web Ontology Language Overview Internet document: `http://www.w3.org/TR/owl-features/`.

[25] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, S. Manegold. Column-store support for RDF data management: not all swans are white. IN *Proc. of the Int. Conf. on Very Large Database (VLDB'08)*, 2008

[26] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of the first International Workshop on Semantic Web and Databases (SWDB'03), Berlin, Germany*, 2003.