# Designing a Document Retrieval Service with Onto⇔SOA

Maksym Korotkiy and Jan Top

Vrije Universiteit Amsterdam, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
maksym@cs.vu.nl jltop@cs.vu.nl

**Abstract.** We describe *Onto⇔SOA* – an approach to integrate ontologies and Service-Oriented Architectures (SOA) to provide a simple yet effective mechanism for representing and exploiting both *conceptual* and *behavioral* domain aspects. We interpret SOA as *an architectural style* constrained to induce the *domain alignment* and *loose coupling* characteristics on a compliant architecture. By introducing ontologies into SOA we enhance these characteristics and supply an explicit *conceptual domain model* facilitating interoperability between a service and a consumer. We apply *Onto⇔SOA* to the document retrieval task that provides a basis for comparing the proposed approach to Semantic Web Services.

## 1 Introduction

The potential of true integration of Service-Oriented Architectures (SOA) and ontologies has been recognized by the Semantic Web Services (SWS) research community, with OWL-S [1] and WSMO [2] approaches being the two most well-known representatives. Both OWL-S and WSMO provide extensive ontology-based description frameworks for Web Services. The frameworks are meant to automate service-related tasks such as discovery, invocation, choreography and orchestration.

SWS researchers face a number of difficulties. We believe that these are caused by the fact that a *formal* ontology language is applied to a *very broadly* defined notion of a service. Moreover, the SWS approaches simultaneously target *a number of tasks* each of which is complex in itself. Our intuition is that integration of ontologies and SOA does not inherently require the level of complexity observable in the SWS approaches. We illustrate this with Onto⇔SOA – a framework that integrates ontologies and services to provide a simple yet effective mechanism for representing and exploiting both *conceptual* and *behavioral* domain aspects. In Onto⇔SOA we introduce a number of assumptions about services and associated ontologies that reduce the challenges met by the SWS approaches.

We employ the Software Architectures perspective to analyze the properties of a service and SOA. Software Architectures [3] aim to provide guidelines for design and analysis of software systems that possess certain characteristics. In [4] we have described Onto⇔SOA as *an architectural style that combines ontologies and SOA in a technology and ontology language independent manner*.

In Onto⇔SOA we propose to employ an ontology-based domain model as a direct input to a service. This differs from the more traditional data-oriented approach in which a conceptual domain model is considered as an intermediate design artefact rather than a direct input. We argue that a service, and more generally SOA, is well suited for processing of domain models because SOA explicitly address *domain-aligned* functionality.

We have implemented Onto⇔SOA in *MoRe* [1] – a Java framework that facilitates integration of RDF/S [10, 11] ontologies and REST Web services. We have employed *MoRe* in a number of use cases from the e-Science domain [5]. In this paper we describe the Document Retrieval case in which we employ *MoRe* to design and implement an ontology-enabled service-oriented solution to the problem of finding documents that match a given query.

In Sec. 2 we outline the Document Retrieval case which will be a running example throughout the paper. In Sec. 3 we define SOA as an architectural style and introduce the Onto⇔SOA framework. Sec. 4 briefly compares traditional Web Services to the SOA style and outlines the main differences between Onto⇔SOA and Semantic Web Services. After that, we return to the Document Retrieval case in Sec. 5 to illustrate the Onto⇔SOA solution. Finally, we conclude with Sec. 6.

## 2 Document Retrieval Case

We employ the task of retrieving documents to demonstrate the main characteristics of SOA and illustrate the Onto⇔SOA approach. The case addresses the design of a service-oriented software system capable of solving the Document Retrieval case. From the user perspective the case can be defined as follows. *Given a collection of documents and a text query, find all documents from that collection that match the given query*. We will refer to this user's view as the Document Retrieval *application domain* – a *conceptual description of the problem domain as perceived by the user*. In this case, the application domain can be described with concepts directly extracted from its definition: *document*, *document collection*, *query*, *retrieved documents* (the left-hand side of Fig. 1).

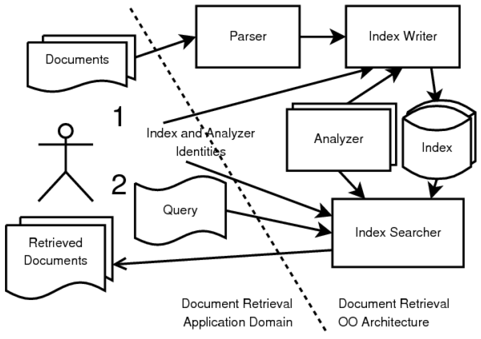
Parser
Index Writer
Documents
1
Index and Analyzer Identities
Analyzer
Index
2
Query
Index Searcher
Retrieved Documents
Document Retrieval Application Domain
Document Retrieval OO Architecture

**Fig. 1.** The Document Retrieval application domain and a corresponding OO architecture.

We use Lucene [2] – a well-known open source Java API for document retrieval – as a reference OO design and implementation. In Web Services it is common practice to design a service as a rather thin wrapper around an, usually existing, OO design. The resulting approach preserves most of the original OO characteristics and differs primarily in data serialization and communication layers. This allows us to

[1] *MoRe* used to be an abbreviation, however by now it has lost the original meaning. Presently we employ it as a label only.

[2] http://lucene.apache.org/

directly employ the Lucene's OO design as an approximation of Web Services practices, to which we will contrast SOA defined as an architectural style.

Lucene's approach to document retrieval will reappear in the alternative designs throughout the paper. The approach consists of two main steps implemented by a number of objects, as depicted in the right-hand side of Fig. 1:

1. *Document Indexing* is supported by the Parser, Analyzer and IndexWriter objects. Parser extracts the structure and content from a document. Then, Analyzer applies natural language processing techniques (stop-words filtering, stemming etc) to the document's content. IndexWriter pre-computes statistical information (term and document frequencies) and stores it in an index along with the term-document map. The main purpose of the index is to improve the speed of the document retrieval process by providing quick access to the pre-computed statistics and documents containing a given term.
2. *Document Search* is supported by the IndexSearcher object that takes a text query, processes it with Analyzer and accesses the index to obtain the pre-computed statistics and documents containing query terms. The statistics is used to compute the ranks of the retrieved documents.

In Lucene the described steps must be coordinated in several ways. *Document Indexing* must precede *Document Search*, otherwise the index might contain no data about the documents being queried. The same type of Analyzer must be employed in each of the steps, otherwise processed document terms might not match processed query terms. Since multiple indexes are possible, an index to be employed must be identified at each step as well.

The user must be aware of the peculiarities of Lucene's approach in order to successfully employ the API. Lucene's approach to document retrieval can be seen as a refinement of the user's view on the *application domain*. The main purpose of this refinement is to construct a more flexible domain model that enables implementation of a configurable software system. This flexibility is obtained by exposing internal details and requires the user's conceptualization to include the corresponding concepts (*analyzer*, *index* etc). However, strictly speaking, these concepts are irrelevant to the user and, therefore, do not belong to the Document Retrieval application domain specified above.

## 3 Onto⇔SOA: an Ontology-enabled SOA

In this section we outline the main ideas behind Onto⇔SOA. In Sec. 3.1 we define SOA as an architectural style that emphasizes *domain alignment* and *loose coupling* characteristics of a service. In Sec. 3.2 we introduce ontologies into the SOA style to enhance the above-mentioned characteristics.

### 3.1 SOA as an Architectural Style

A number of viewpoints on and definitions of *a service* and *SOA* exist in different communities [6] as well as within the Software Engineering field [7]. To cope with this ambiguity, in Onto⇔SOA we explicitly restrict services to *business-aligned* and *loosely-coupled* entities only. We consider these two characteristics to be the key properties of a service rather than being merely slogans.

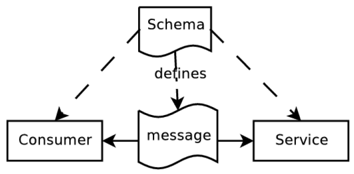

**Fig. 2.** SOA elements.

Surprisingly, we have not found any attempts to define SOA from the viewpoint of Software Architectures. While defining SOA as an architectural style, we analyze relationships among the key characteristics of a service, its internal properties and the target application domain. This will determine how we can further support the target characteristics by means of ontologies.

In [8] *an architectural style* is defined as "a coordinated set of constraints on architectural elements and relationships among those elements within any architecture that conforms to that style". The same reference defines *a software architecture* as "an abstraction of run-time characteristics of a software system during some phase of its operation". As an abstraction, an architecture provides a simplified view on a software system with only relevant characteristics highlighted. Since there exist a virtually infinite number of combinations of architectural characteristics, a software system can have many architectures. However, there is a limited number of characteristics which are relevant in practice.

A *software architecture* provides the means to analyze the characteristics of a system but it does not instruct on how an architecture should be designed to possess those characteristics. An *architectural style* addresses this issue by imposing constraints on architectural elements, thus, inducing desired characteristics on an architecture.

The most general architectural elements are *processing components*, *connectors* and *data*. Processing components can transform data elements. Connectors provide an abstract mechanism that mediates communication, coordination or cooperation among components [9]. From the processing component perspective, connectors transfer data without modifying them. Nevertheless, internally a connector can contain a complex subsystem that subjects the data to a number of intermediate transformations. We interpret SOA as an architectural style that restricts its elements shown in Fig. 2 to induce the *domain alignment* and *loose coupling* characteristics upon which we elaborate in the rest of this section.

**Loose Coupling** *Loose coupling* implies reducing dependencies between system components. This is beneficial for a system subjected to frequent changes. To reduce dependency between a service and a consumer we constrain the connector and data elements.

We require connectors to be simple, generic and application independent. This allows one to deploy the connector across different application domains or make them unaffected by changes in the application domain. We require data elements to contain descriptive messages because descriptive messages require less assumptions than prescriptive ones. Normally, in a prescriptive message we must specify the operation name, its inputs, outputs, preconditions and effects. A descriptive message is specified by inputs only. A schema language defines a unified syntax and structure of messages. Additionally, it must be able to address a wide range of application domains. A schema provides vocabulary required to express and interpret messages. Since we keep the connectors as simple as possible, a schema only addresses data elements (messages) which accommodate all service-specific details.

In *document-oriented* messaging [3] a service and a consumer exchange messages that can be directly linked to artifacts in an application domain (a purchase order or a document corpus, for example). This contrasts to the data-oriented messaging style such as SOAP-RPC. In SOA we favor *document-oriented* messaging because of its *descriptive* nature.

In SOA there is a uni-directional dependency between a consumer and a service: a consumer depends on the functionality provided by a service but a service is independent from its consumers. Decoupling between a consumer and a service can be increased by favoring *session stateless services* [4]. In this way we ensure that a service does not rely on a client to perform a defined sequence of actions.

**Domain Alignment** In SOA services are often referred to as *business aligned*. We translate *business alignment* into the more general *domain alignment* characteristic defined as the ability of a service to have a direct relationship (support, facilitate, enable etc) with domain elements (processes, requirements etc). The SOA style, therefore, constrains its elements to establish a direct link to domain elements.

In a system consisting of multiple components some of them are better aligned to the system domain than others. We propose to use the degree of a component's granularity as an indication of its alignment to the system domain (*domain alignment*). A *coarsely-grained* component encapsulates complex functionality that is likely to have a direct connection to the domain of discourse. We assume that the more *coarsely-grained* component is – the better it is aligned to the system domain, and, therefore, the better it is suited to become a service. Since we induce *domain alignment* via *coarsely-grained* processing components, we assume that the most *coarsely-grained* component is the best candidate to become a service. This implies that it should be always possible to define exactly one most coarsely-grained processing component for a given application domain.

*Omnipotence* of a service is a direct consequence of the statement above. *Omnipotence* means that a service is self-contained in the sense that it requires no other services within the same domain to provide its functionality. This does not restrict interaction between services that belong to *different domains*.

Since a service schema defines which messages a service is able to process, a schema should also be *domain aligned*. An ontology is by definition a *domain-aligned* entity, and, therefore, has the potential to support *domain alignment* of a service if employed as a service schema.

[3] Document-oriented messaging should not be confused with the document-based encoding style supported by SOAP.

[4] A desirable characteristic that facilitates monitoring, recovering after failures, design, implementation and reuse of services

## 3.2 Ontology-enabled SOA

In [4] we have proposed Onto⇔SOA as a derivation of the SOA style that assumes a direct exchange of *document-oriented*, *ontology-based* messages between a service and a consumer. We have derived Onto⇔SOA by introducing additional constraints on the connector and data elements. In Onto⇔SOA we require that an ontology language is used to express a schema underlying messages. Both a service and its consumer commit to this underlying ontology. In this paper we will refer to a schema expressed using an ontology language as *a service ontology* or *an ontology*, for short.

Additionally, we restrict interaction between a service and a consumer to a unified protocol. A consumer employs an ontology to *as completely as possible* describe a domain situation (i.e. a problem or case) at hand. This description is sent to a service that applies its domain knowledge to complete the initial situation with inferred facts (i.e. the problem solution). Finally, a description of the completed situation is sent back to a consumer.

An ontology language provides a unified syntax and data model as well as *a minimal set of conceptual primitives*. The primitives must be simple and *intuitively understandable* to the user. We do not require the language to have a well-defined *formal* semantics because in Onto⇔SOA we ground the *application semantics* in the behavior of a service and the user's understanding of the service domain.

We do not restrict Onto⇔SOA to a certain set of *conceptual primitives*. So far, in all cases we have employed *Class* - *Property* - *Instance* as *conceptual primitives*, however, it should be equally possible to employ other *conceptual primitives* such as *Entity* - *Relations* or *Subject* - *Predicate* - *Object* as long as there is a sufficient consensus about their meaning.

We distinguish two categories of concepts employed in *a service ontology*:

1. *conceptual primitives* that provide basic modeling building blocks (e.g. *Class* - *Property* - *Instance*) not affecting service behavior. The primitives are grounded in the "real world" and enable interfacing with the user.
2. *domain concepts* that affect service behavior. The *application semantics* of *domain concepts* is captured in the behavior of a service. The concepts enable communication with a service. For a service it is sufficient to be aware of *domain concepts* only and *conceptual primitives* such as *Class* - *Property* - *Instance* are not required to implement a domain-specific task.

In Onto⇔SOA a service ontology may not contain concepts that do not fall under either of the two categories.

Additionally, an ontology language employed to express a schema must be able to capture *application semantics* for a wide range of domains. If we intend to respect *formal* semantics of an ontology language, then the more extensive and restrictive it is – the more difficult its application in Onto⇔SOA due to a likely conflict between *formal* and *application semantics*. For example, since RDFS has much less restrictive *formal* semantics than any language from the OWL-family, the former can be employed in Onto⇔SOA with less concerns about possible semantic conflicts.

## 4 Existing Approaches

In Onto⇔SOA we interpret SOA (and a service) in a more restricted way than commonly employed in the Web Services and Semantic Web Services fields. In the coming subsections we outline the main differences between the two interpretations and some of the implications.

### 4.1 Web Services

WSDL/SOAP Web Services is the most popular application of SOA on the Web. WSDL provides a description framework for Web Services and is primarily intended for service discovery and invocation. SOAP [12] defines a standard way to structure messages that can be carried over a variety of transport protocols with HTTP being the most frequently used.

Our interpretation of SOA as an architectural style is more restricted than, but still compatible with, the broader view on Web Services. We do not assume that any software component can be transformed into a service regardless of its internal properties and architectural context (application domain). We require that in order to become a service, a software component must be sufficiently well aligned with a target service domain.

Web Services in most cases employ the RPC communication style. From the SOA perspective, RPC introduces an additional dependency between a consumer and a service and, therefore, hinders *loose coupling* between these components. This dependency results from the RPC interaction protocol that requires a consumer to be aware of the name of the operation, its input arguments, external effects of invocation etc. Furthermore, RPC messages tend to be *prescriptive* rather than *descriptive*. The *prescriptive* nature of RPC Web Services often leads to *stateful sessions*. This results in fragile interaction protocols and increased dependency between a service and a consumer. The dependency caused by RPC can be reduced by employing a *document-oriented* communication style which we favor in Onto⇔SOA.

### 4.2 Semantic Web Services

The state-of-the-art approaches to Semantic Web Services (SWS) [13] such as OWL-S [1] and WSMO [2] employ ontologies to provide *formal* descriptions of Web services to automate discovery, invocation and composition of such services. One of the main differences between SWS and Onto⇔SOA is that the former targets *multiple* architectures (SOAs) whereas the latter addresses a *single* architecture and a corresponding domain.

From the Software Architectures perspective, SWS can be seen as a complex connector between a service and its consumer that ensures semantic compatibility of messages. The connector can perform complex intermediate transformations of a message, however, a service still operates on data-level requests (SOAP-RPC in most cases) instead of conceptual ontology-based messages. Although the WSMO approach has a potential to do so, in practice the SWS approaches rarely address direct exchange of ontology-based messages between processing components in SOA.

In SWS there is a tendency (more visible in OWL-S than in WSMO) to disregard the actual properties of a service and to assume that any software component can be modelled within a SWS framework adequately well to support the target tasks. This results in a fairly extensive framework that requires a large amount of meta-data to describe a service. Moreover, since the internal properties are disregarded, it is difficult to provide guidelines on how to translate the internal service properties and a corresponding SWS model into each other: how to design a service or provide a meta-data description for an existing one.

In Onto⇔SOA we focus on the invocation task only. By means of the SOA style we constrain the internal properties of a service to induce the *domain alignment* and *loose coupling* characteristics. This simplifies the model of a service, reduces the amount of meta-data required to describe it and provides guidelines on design of Onto⇔SOA services.

## 5 Onto⇔SOA Solution for Document Retrieval Case

In [5] we have introduced *MoRe* – a derivation, supported by a prototype implementation, of Onto⇔SOA. It combines the RDF/S languages with the elements of REST services. We employ *MoRe* to evaluate Onto⇔SOA and provide a simple, yet efficient framework for ontology-enabled application development.

In this section we describe a *MoRe*-based solution for the Document Retrieval case introduced in Sec. 2. The solution consists of a *MoRe*-based service [5] and a demonstration application [6]. In coming sections we elaborate on the design of two major Onto⇔SOA artifacts: the service (Sec. 5.1) and the service ontology (Sec. 5.2).

### 5.1 Document Retrieval Service

By applying the Onto⇔SOA constraints to the Document Retrieval architecture shown in Fig. 1 we determine that none of the components is properly aligned to the Document Retrieval domain defined in Sec. 2. The IndexWriter and IndexSearcher components require *an index identifier* – a concept absent from the Document Retrieval domain. Neither the Parser nor Analyzer components perform functions directly related to the Document Retrieval domain. Moreover, the components are of the same granularity and depend on other components. This does not allow us to distinguish a single, *omnipotent* component among them. To find documents matching a given query the user has to follow a rather complex interaction protocol that is likely to require a *stateful session*: the user has to maintain the index and the analyzer identities across component invocations, and must properly order invocations.

We could disregard the Onto⇔SOA constraints and design an architecture in which each of the components is transformed into a service by merely providing a Web-enabled interface to them. The misalignment between these components and the Document Retrieval domain would, however, cause a service schema to expose concepts

[5] http://swpc333.cs.vu.nl:8080/DRDemo/pages/Service.jsp
[6] http://swpc333.cs.vu.nl:8080/DRDemo/pages/DescribeProblem.jsp

(index, analyzer, parser etc) alien to the target Document Retrieval domain. The non-aligned concepts forced on a consumer unnecessarily complicate the document retrieval task and, ultimately, compromise *loose coupling* by exposing implementation details.

In particular, we consider an index to be an implementation detail irrelevant to the functionality of the Document Retrieval service. The sole *non-functional* purpose of an index is to contain pre-computed data to improve the performance of the retrieval process. Even if an index contains data relevant to a consumer (e.g. term frequencies), the associated functionality (computation of frequencies) should be exposed rather than the way the data is stored (an index). Moreover, if we decide to expose this additional functionality then the service domain must be redefined accordingly to include the concepts of term and document frequencies. This effectively results in an application domain distinct from the domain defined in Sec. 2 and, therefore, in a new SOA.

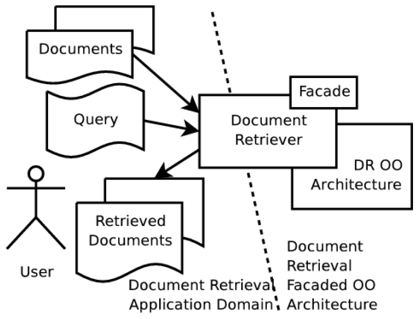
Documents
Query
Facade
Document Retriever
DR OO Architecture
User
Retrieved Documents
Document Retrieval Application Domain
Document Retrieval Facaded OO Architecture

**Fig. 3.** Document Retrieval case: an architecture with a Facade object that encapsulates the architecture depicted on Fig. 1.

To fulfill the Onto⇔SOA constraints we extend the original architecture for the Document Retrieval system. To provide a *domain-aligned* interface, we employ a Facade object [14] that confronts the user with the concepts from the Document Retrieval application domain only, while hiding the peculiarities of the Lucene's approach to document retrieval (Fig. 3).

The Document Retriever Facade object fulfills the SOA constraints. The object is well aligned to the target domain. Its interface exposes only concepts that occur in the Document Retrieval domain. It is the most *coarsely-grained* component consisting of a number of finer-grained components (IndexWriter, Analyzer etc). The object is *omnipotent*: it does not depend on other components of the same level of granularity. It does not require a *stateful session*. Since neither the index nor Analyzer concepts are exposed, there is no need to maintain their identities across service invocations.

We employ the *MoRe* framework to transform the Facade object into a *domain-aligned*, *session-stateless*, *omnipotent* and *document-oriented* service. The service accepts an RDF description of an instance of the Document Retrieval case and returns another RDF document with a solution. The terminology for both types of documents is defined in the Document Retrieval *service ontology* expressed in RDFS.

## 5.2 Document Retrieval Service Ontology

The Document Retrieval service ontology is a specification of the Document Retrieval domain. The Document Retrieval service defines how a particular domain model (an instance of the Document Retrieval case) must be interpreted by a machine: the service must compute a solution for an instance of the Document Retrieval case. To describe an instance of the Document Retrieval case and a corresponding solution we introduce a number of RDFS Classes and Properties summarized in Fig. 4.

| **Concept** | **Description** | **Provided by** |
|---|---|---|
| DocumentRetrievalCase class | the top-level container for the Document Retrieval domain | consumer |
| – hasCorpus property | points to a document corpus | consumer |
| – hasQuery property | points to a query to be matched against documents | consumer |
| – hasSolution property | points to a solution to a DR problem | service |
| Corpus class | contains a set of documents | consumer |
| – containsDocument property | points to a document that belongs to this corpus | consumer |
| Query class | represents a query | consumer |
| – hasQueryString property | contains a literal value with a query | consumer |
| Document class | | consumer |
| – hasURL property | contains an URL of a document | consumer |
| Solution class | contains a collection of documents that match the given query | service |
| – hasRetrievedDocumen | points to a retrieved document that belongs to a solution | service |
| RetrievedDocument class | represents a matched document | service |
| – hasDocument | points to a document from the problem Corpus | service |
| – hasScore | contains match score for the retrieved document | service |

**Fig. 4.** The Document Retrieval service ontology.

The Document Retrieval service ontology is aligned to the Document Retrieval problem domain: all concepts introduced by the ontology can be readily found in the original domain definition. In this way we expect to improve the usability of the service by bridging the conceptual gap between a consumer's view on the service domain (this was the basis for the definition of the Document Retrieval domain in the first place) and functionality provided by the service.

The Document Retrieval service ontology provides the terminology required by a consumer to express an instance of the Document Retrieval case (Fig. 5): a corpus consisting of a collection of documents and a text query. The ontology is also employed by the service to express the solution to the case (Fig. 5): a collection of ranked documents that match the query. The solution document contains what can be seen as facts inferred by the service from the original description of the case.

### 5.3 Implementation

Fig. 6 outlines the main processing steps taking place inside an Onto⇔SOA service and a consumer. Fig. 7 illustrates the processing steps typical for a Web service implementation enabling us to compare the two approaches.

The process cycle of the Document Retrieval service, and more generally of an Onto⇔SOA service and a consumer, consists of the following steps.

**1.** The service receives an RDF document containing a description of the Document Retrieval problem. The document is expressed using terms from the Document Retrieval service ontology and serialized in the RDF-XML syntax. We refer to such RDF-XML serialization as an *external conceptual model*. Fig. 5 contains an example of an *external conceptual model* of an instance of the Document Retrieval problem.

```
-DRProblem
  type  DocumentRetrievalProblem
  hasQuery aQuery
  hasCorpus aCorpus
-aQuery
  type Query
  hasQueryString ‘‘Onto-SOA’’
-aCorpus
  type Corpus
  containsDocument document1
  containsDocument document2
  containsDocument document3
-document1
  type Document
  hasURL ‘‘.../˜maksym’’
-ppdocument2
  type Document
  hasURL ‘‘.../Onto-SOA-WEBSA.pdf’’
-document3
  type Document
  hasURL ‘‘.../OntologiesAndQualities.pdf’’
```

```
-DRProblem
  hasSolution aSolution
-aSolution
  type Solution
  hasRetrievedDocument rDocument1
  hasRetrievedDocument rDocument2
-rDocument1
  type RetrievedDocument
  hasDocument document1
  hasScore 0.65255654
-rDocument2
  type RetrievedDocument
  hasDocument document2
  hasScore 0.9764538
```

**Fig. 5.** An example of a Document Retrieval problem description (left-hand side) and a corresponding solution (right-hand side).

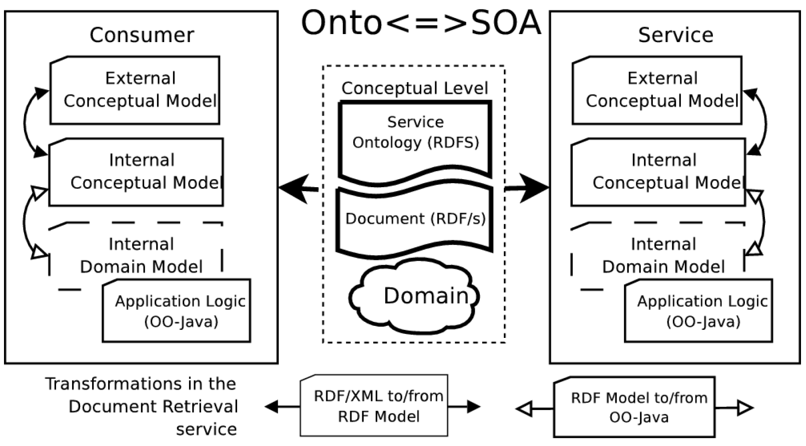
Onto<=>SOA
Consumer
External Conceptual Model
Internal Conceptual Model
Internal Domain Model
Application Logic (OO-Java)
Conceptual Level
Service Ontology (RDFS)
Document (RDF/s)
Domain
Service
External Conceptual Model
Internal Conceptual Model
Internal Domain Model
Application Logic (OO-Java)
Transformations in the Document Retrieval service
RDF/XML to/from RDF Model
RDF Model to/from OO-Java

**Fig. 6.** The Onto⇔SOA process cycle. In parenthesis we refer to a specific technology employed in the Document Retrieval service and demo application.

**2.** The received RDF-XML document must be parsed and converted by the service into an *internal conceptual model* to enable programmatic access to its content. In the Document Retrieval service we employ the in-memory RDF model provided by Jena API [15]. A database-backed model can also be employed as long as the RDF API of choice supports it. The *internal* and *external conceptual models* are equivalent and, therefore, we can automatically transform them into each other by means of Jena API. Both conceptual models are *domain aligned* because this characteristic is enforced in Onto⇔SOA.

**3.** The in-memory RDF model of an instance of the Document Retrieval case is linked to an *internal domain model* (*internal model*, for short) of the service implementation. The *internal domain model* must be computationally feasible and, consequently, is effected by such *non-domain-aligned* factors as an implementation language (Java), available APIs (Lucene), non-functional requirements (performance, security etc) and the algorithms employed. The *conceptual* and *internal domain models* in most cases are not equivalent because the latter is effected by the factor listed above, whereas the former is not. The mapping between the two models (the Document Retrieval service ontology and the Java OO model of the Document Retrieval service implementation) cannot be determined automatically.

**4.** The Document Retrieval service logic (based on the Lucene API) is applied to the *internal model* of the Document Retrieval case. A solution to the case extends the *internal model* which then undergoes the above steps in reverse order. The process cycle results in an *external conceptual model* of the solution sent to a consumer. Fig. 5 displays an example of such an extended model. A consumer then follows steps 1-3 to interpret the response, and reverses the order of steps to invoke a service.

Generally speaking, the *internal models* of a consumer and a service are different. In the Document Retrieval demonstration application, the *internal model* is very thin. The user's actions are translated almost directly into the *conceptual model*. The overhead of generality of a *conceptual model* can be ignored. At the same time, the Document Retrieval service cannot ignore the overhead because it must be able to efficiently handle multiple Document Retrieval requests, and therefore, requires an optimized *internal model*.

An *internal model* can be reduced by implementing application logic to operate as directly as possible over a *conceptual model*. Ontology middleware (query languages, inference engines, rule languages etc) simplifies the use of *conceptual models* from within application logic. For example, by means of a query language (e.g. SPARQL [16]) RDF/S models can be processed in a rather efficient way directly from application logic. However, an application-specific *internal model* still offers better performance. Additionally, even in the case of advanced ontology-aware middleware it is possible that an *internal domain model* is not reduced but rather transferred from a programming language to an ontology query language.

To reduce the possibility of mis-alignment between *conceptual* and *internal models* a direct operation over a *conceptual model* should be preferred. If this is impossible because of performance considerations, then a direct operation can be approximated with intermediate coarse updates to the *conceptual model*. The discrete updates allow to validate intermediate computational states against a *domain-aligned conceptual model*.

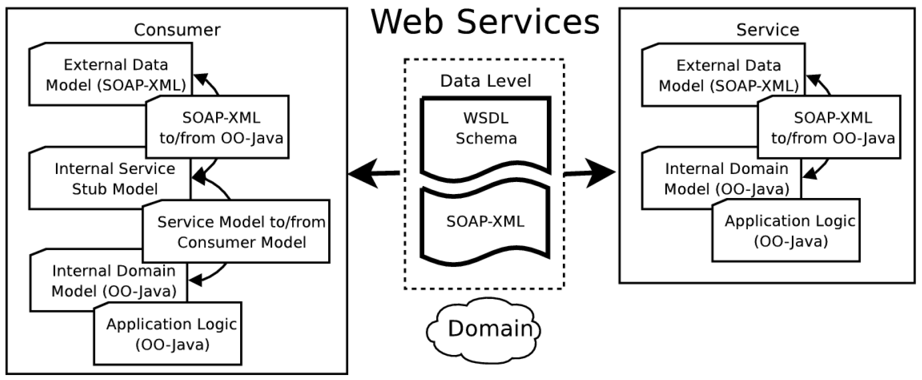
Web Services
Consumer
External Data Model (SOAP-XML)
SOAP-XML to/from OO-Java
Internal Service Stub Model
Service Model to/from Consumer Model
Internal Domain Model (OO-Java)
Application Logic (OO-Java)
Data Level
WSDL Schema
SOAP-XML
Domain
Service
External Data Model (SOAP-XML)
SOAP-XML to/from OO-Java
Internal Domain Model (OO-Java)
Application Logic (OO-Java)

**Fig. 7.** Outline of the Web Services processing cycle.

In Web Services (Fig. 7) the communication between a consumer and a service takes place on the data level. At the service side an external data model (XML data-type definitions) is automatically generated from an *internal model* by means of programming language-specific Web Service middleware (e.g. Apache Axis [7], Java EE Web Service tools [8]). At the consumer side the XML data-type definitions contained in a WSDL description of a service are used to automatically generate an *internal stub model*. This model is then connected to a consumer's *internal model*. Since *conceptual models* are not explicitly present in Web Services, the conceptual gap exists between the *internal models* of the service and the consumer.

In SWS an ontology provides a model of a Web service. Since *domain alignment* is not enforced in Web Services, the ontology is likely to address implementation details of the service rather than the *conceptual model*, thus, compromising the *domain alignment* and *loose coupling* characteristics. Onto⇔SOA addresses the potential conceptual mis-alignment between a service and its consumer by explicitly introducing *conceptual models* into their processing cycle.

## 6 Conclusions

We have described *Onto⇔SOA* – a framework that integrates ontologies and Service-Oriented Architectures. *Onto⇔SOA* enables a direct exchange of ontology-based messages between a service and its consumer. In the proposed framework we interpret SOA as an *architectural style* that constrains internal properties of a service to induce *domain alignment* and *loose coupling* characteristics. We enhance these characteristics by employing an ontology as a *service schema*. This introduces an explicit *conceptual domain model* into a service and a consumer facilitating conceptual interoperability between them. By constraining the notion of a service we simplify its model, reduce the amount

[7] http://ws.apache.org/axis/
[8] http://java.sun.com/webservices/

of meta-data required to describe it and provide guidelines on design of Onto⇔SOA services. We have elaborated on an *Onto⇔SOA*-based design and implementation of a service for the document retrieval task and outlined the differences between the proposed approach and Semantic Web Services.

## 7 Acknowledgements

We would like to thank Jacek Kopecky and Borys Omelayenko for their invaluable suggestions, comments and criticism that helped us to improve this paper.

## References

1. W3C: OWL-S: Semantic Markup for Web Services. (http://www.w3.org)
2. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontology **1** (2005) 77–106
3. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17** (1992) 40–52
4. Korotkiy, M., Top, J.: Onto-SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In: International Conference on Internet and Web Application and Services (ICIW'06). Guadeloupe. (2006)
5. Korotkiy, M., Top, J.: MoRe Semantic Web Applications. In: Proceedings of the ESWC'05 workshop on User Aspects of the Semantic Web. (2005)
6. Baida, Z., Gordijn, J., Omelayenko, B., Akkermans, H.: A shared service terminology for online service provisioning. In: Proceedings of the Sixth International Conference on Electronic Commerce (ICEC04), ACM Press (2004)
7. Hotle, M.: A conceptual evolution: From process to web services. Gartner Group Research Note TU-16-1420 (2003)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, UNIVERSITY OF CALIFORNIA (2005)
9. Shaw, M., Clements, P.C.: A field guide to boxology: Preliminary classification of architectural styles for software systems. In: COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference, Washington, DC, USA, IEEE Computer Society (1997) 6–13
10. W3C: RDF Semantics. (http://www.w3.org/TR/rdf-mt/)
11. W3C: RDF Vocabulary Description Language 1.0: RDF Schema. (www.w3.org)
12. W3C: Web Services: Recommendations, Specifications and Documents (WSDL, SOAP, etc). (http://www.w3.org/2002/ws)
13. Cabral, L., Domingue, J., et al: Approaches to semantic web services: an overview and comparisons. In Bussler, C., Davies, J., Fensel, D., Studer, R., eds.: ESWS. Volume 3053 of Lecture Notes in Computer Science., Springer (2004) 225–239
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns : Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
15. HP Labs Semantic Web Activity: Jena Semantic Web Toolkit. (http://www.hpl.hp.com/semweb/)
16. W3C: SPARQL Query Language for RDF. W3C Candidate Recommendation. (http://www.w3.org/TR/rdf-sparql-query/)