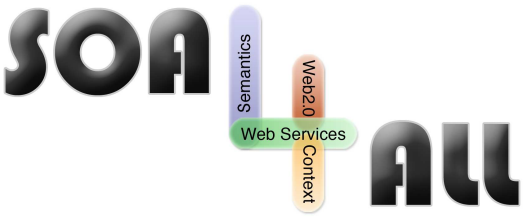

Project Number:	**215219**
Project Acronym:	**SOA4All**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D3.4.6 MicroWSMO v2 – Defining the second version of MicroWSMO as a systematic approach for rich tagging

Activity N:	A2 Core R&D	
Work Package:	WP3 Service Annotation and Reasoning	
Due Date:		28/02/2010
Submission Date:		26/02/2009
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		UIBK
Revision:		1.0
Author(s):	Florian Fischer (UIBK) Barry Norton	
Reviewers:	Carlos Pedrinaci (OU) Patrick Un	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**x**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	10.1.2010	Creation	Florian Fischer
0.2	17.1.2010	Outline	Florian Fischer
0.3	25.1.2010	Initial sections and diagrams	Florian Fischer
0.4	5.2.2010	Service models	Barry Norton, Fischer Florian
0.5	14.2.2010	Formatting, review version	Florian Fischer
1	23.2.2010	Reviewer comments, clarifications, examples	Florian Fischer

Table of Contents

List of Figures

Glossary of Acronyms

Acronym	Definition
D	Deliverable
EC	European Commission
WP	Work Package
API	ApplicationProgrammingInterface
GRDDL	GleaningResourceDescriptionsfromDialectsofLanguages
HTML	HyperTextMarkupLanguage
HTTP	HyperTextTransferProtocol
ID	Identifier
JSON	JavaScriptObjectNotation
OWL	WebOntologyLanguage
RDFS	RDFSchema
RDF	ResourceDescriptionFramework
REST	RepresentationalStateTransfer
SAWSDL	SemanticAnnotationsforWSDLandXMLSchema
SWS	SemanticWebServices
URI	UniformResourceIdentifier
WSDL	WebServicesDescriptionLanguage
WSMO	WebServiceModelingOntology
XHTML	ExtensibleHyperTextMarkupLanguage
XML	ExtensibleMarkupLanguage
XSL	ExtensibleStylesheetLanguage
XSLT	XSLTransformations
MSM	Minimal Service Model
ROA	Resource Oriented Architecture

Executive summary

RESTful web services are one of the rising trends within the Web 2.0. Their perceived simplicity and ease to deploy them makes them an attractive option in place of traditional WS-* service and their associated overhead.

On the one hand, Web APIs that simply use HTTP as a transport protocol are already in widespread use and deployed by well-known players such as Twitter[1] or Flickr[2]. However, those Web APIs often do not really follow REST principles but merely offer their functionalities through an interface accessible via HTTP. More concretely, their specifications are based on the Remote Procedure Call programming style and employ various service bindings, among them HTTP. Those services often support multiple operations on diverse data, all realized through a POST on a single URI.

On the other hand, REST itself is architectural style and principles that already form a cornerstone in the design of the Web itself. RESTful services set themselves apart by focusing on resources and their manipulation through plain HTTP methods, instead of following an RPC-style approach in which each service defines custom operations and for that purpose defines its own vocabulary to talk about those operations.

Those benefits (e.g. a uniform interface operating on resources) are partially lost when the underlying REST principles are discarded and HTTP is only used as a transport protocol. This has resulted in a growing number of applications that leverage a truly resource-oriented architecture (ROA), as a concrete implementation of REST's principles, within prominent examples: Amazon's S3 storage service[3], the Apache CouchDB project[4], the Atom Publishing Protocol (AtomPub) [1], or more concretely Google's GData[5] which is used to interface with various of Google's services such as Google Apps APIs, (Google Calendar Data API, Google Finance Portfolio Data API, etc.).

Both types of APIs, resource-oriented as well as RPC-oriented, are usually not described in a machine-oriented format but rather only through textual documentation aimed at humans.

The original minimal service model underlying MicroWSMO still reflected its origins, namely WSDL, and thus assumes that a service follows an RPC- based modelling style along with corresponding operations supported by the service. While this service model is sufficient and allows to support simple HTTP-based Web APIs it is hard to express truly resource-oriented services adequately in this service model.

We therefore revise and extend the service model underlying MicroWSMO in order to make it more applicable to truly RESTful and resource-oriented services, which are a focus of MicroWSMO. This proposed extension does not lose current functionality or expressivity but merely extends MicroWSMO towards a larger range of services. For this purpose, we present a common core service model and extend it towards a resource-oriented service model as well as towards an RPC-oriented service model, and then discuss how they relate to SOA4All.

[1] http://apiwiki.twitter.com/REST-API-Documentation

[2] http://www.flickr.com/services/api/

[3] http://aws.amazon.com/s3/

[4] http://couchdb.apache.org/

[5] http://code.google.com/apis/gdata/

1. Introduction

This deliverable serves as an extension and an update to the previous deliverable D3.4.4 - "MicroWSMO and hRests" [2]. D3.4.4 defined a microformat called hRESTS for annotating RESTful Web APIs and embedding machine-readable descriptions in HTML documents.

Microfromats [3] are an approach to define structured units of information by using semantic XHTML and readily available HTML attributes. By this means, they make the key information within human-oriented Web content accessible for machines. Typical applications are embedding contact details, licensing information, events etc. in web pages.

An extraction of RDF[4] information from Web pages containing microformats is possible e.g. by means of GRDDL [5], which in turn acts as a bridge between simple XHTML and semantic formalisms.

Microformats are usually defined in a community driven, bottom-up approach in order to solve a very specific and granular problem. The recommended approach to the definition of a microformat[6] is to first determinate the applicability of existing microformats (or a combination of existing microformats) to solve a real-world use-case first, and to start by reusing building blocks from existing schemas and standards if this is not possible. In practice this means that microformats should reuse existing data-formats and -structures and not focus on defining them. As a practical example, the hCard microformat is built on top of the IETF standard for vCard.

hRESTS, and by extension MicroWSMO, in turn build upon an underlying service model that backs them up. In this way, MicroWSMO serves as a principled way to make it possible to semantically annotate service descriptions in websites so that the SOA4All architecture can consume this information about services. Through this approach MicroWSMO makes it possible to include WSDL [6] services, as well as the many services stemming from Web 2.0 platforms, that rather rely REST as underlying architectural style [7].

For that purpose MicroWSMO acts as a SAWSDL-like layer on top of hRESTS, which in turn resembles WSDL, in order to allow a common service ontology, namely the WSMO-Lite ontology [8], to be used as common ground. SAWSDL [9] is an extension of WSDL that specifies how to annotate service descriptions with semantic information, although it does not define the service model itself. SAWSDL merely defines the following three XML attributes:

- modelReference
- liftingSchemaMapping
- loweringSchemaMapping

For practical purposes, those annotations will be separated from the original service at some point. Therefore, MicroWSMO is currently backed by a simple RDF based service model. called the "minimal service model", which it shares with traditional WSDL based Web services. This minimal, RDF-based service model (MSM) can be used to replace the implicit service model with an equivalent RDF representation, and in turn allows e.g. to store service information in triple stores or use it for further processing, reasoning, etc.

The problem in this regard is that, while MicroWSMO can be used for the description of RESTful services, the notion of a truly Resource Oriented Architecture (ROA) is lost after the transition to the RDF based service model. In turn, it is not possible to represent that architectural style adequately in the current service model, which is still based on RPC-style

[6] http://microformats.org/wiki/process

interactions with a service.

Consequently, we restructure and extend the underlying service model of MicroWSMO in order to increase the reach and expressivity towards RESTful services, especially Web 2.0 resources. Components within the SOA4All architecture can then use this refined service model, e.g. for service discovery or reasoning.

1.1 Purpose and Scope

Based on the first version of MicroWSMO developed in D3.4.3, this deliverables further examines the applicability of MicroWSMO to RESTful services. Based on our findings it presents a re-factored underlying service model, which takes the differences between traditional RPC-style services and resource-oriented services into account, and provides for a more natural and expressive modelling of both types of services within MicroWSMO.

In turn it extends the syntax and modelling capabilities available in MicroWSMO accordingly, so that resource-oriented services can be adequately modelled and that information about them can be used within other tools, such as reasoners or discovery components. We furthermore give examples concerning the usage of these additional modelling capabilities.

1.2 Structure of the document

The remainder of this deliverable is structured as follows: In Section 2.1 we introduce and compare different architectural styles of services, namely RPC-based services, RESTful services (REST as an architectural style and resource oriented services as a concrete architecture), and hybrids between those. We then continue to discuss the implications of those different styles of services have for MicroWSMO, in how far they can be adequately handled, and what shortcomings currently exist.

Based on these observations, Section 2.2 details a proposed refactoring of the underlying minimal service model in order to extend MicroWSMOs reach towards RESTful, resource-oriented services.

Finally, Section 3 concludes the document and lays out future work towards service modelling and other activities needing alignment.

1.3 Technical deliverable remarks

MicroWSMO is the description language used in SOA4All for semantic description of RESTful web services.

In SOA4All those descriptions, based on the WSMO-Lite service ontology [8], are then stored and actually further processed along with the underlying RDF service model (the minimal service model) [10], e.g. for service discovery etc. (see iServe[7] as an example).

In turn, every practical implementation relies on a suitable RDF model at some point. For this reason, it is a high priority to formulate an adequate model for RESTful services that can also

[7] http://iserve.kmi.open.ac.uk/

be queried effectively.

However, the extensions proposed in this document cannot be regarded as firmly established yet due to the lack of existing implementations testing them in practice. As a pathway to further standardization and common agreement, it is mandatory to put the results documented in this document to test in practice. For this purpose it is necessary to document those results – also since one of the main issues arising when refactoring and extending the underlying service model is not to break existing components or functionality at this point in the project.

2. MicroWSMO v2 as Annotation Mechanism RESTful Web Services

In this section we review the initial motivation for MicroWSMO and its intended application area: RESTful web services. We then distinguish between three different architectural types of web services and analyse how applicable the current service modelling approach used in SOA4All is in each regard.

MicroWSMO is a description language aimed at the annotation of RESTful web services. These descriptions are then usually stored in some service registry and processed by further tools. For this purpose MicroWSMO is combined with the minimal service model and the WSMO-Lite service ontology, which captures the actual service semantics.

REST [7] itself is actually an architectural style and not a concrete architecture, so HTTP is by no means the only possible implementation of it. REST focuses on the transfer of representations of specific resources. A resource in this sense is an addressable entity, i.e. on the Web identified by an URI, whereas a representation is usually a document in a specific format.

A cornerstone of REST is a *uniform interface* between clients and servers, which in the case of HTTP are established through the HTTP methods (GET, PUT, POST, etc.). This generality of an interface is a fundamental design consideration within RESTful architectures and in turn a RESTful service. The basic consideration is that once a client has access to some concrete *representation* (HTML, XML, etc.) of a *resource* (identified by an URI) it should have enough information to interact with the resource, i.e. modify it, delete it, etc. assuming it has the permission to do so.

If this principle of a uniform interface is violated, e.g. because the server defines custom methods/operations, then the client cannot interact with a resource without further details. In particular the loss of a uniform and general interface by inventing custom vocabulary for operations on resources limits the independent deployment of components and thus the scalability of the overall system. More concretely, a service that does not adhere to the (semantically correct) use of the uniform interface of HTTP is strictly speaking not RESTful and also loses several advantages of this architectural style.

Many Web APIs that claim to be so are in fact not truly RESTful in the sense that their focus is not to use HTTP methods in a resource-oriented architecture but rather the use of HTTP as merely an envelop format for message exchange.

A practical example for this is the Flickr REST API[8], which is in fact a well-designed RPC API that only uses HTTP as protocol along with a custom response format based on XML, and apart from that mirrors Flickr's SOAP and XML-RPC APIs exactly. Interaction with the Flickr REST API is limited to a single endpoint (http://api.flickr.com/services/rest/), i.e. there are no actual resources involved. To request the `flickr.test.echo` service, the REST API would be invoked by sending an HTTP GET request to the following URI:

```
http://api.flickr.com/services/rest/?method=flickr.test.echo&name=value
```

In general, the method-names correspond exactly to those of the SOAP and XML-RPC based versions. In turn, a HTTP POST request using `flickr.favorites.remove` as parameter would be used to remove a photo from a user's favourites list instead of a simple HTTP DELETE. In a similar way there are a lot of "RESTful" Web APIs that do not really leverage resource-oriented architectures to their fullest potential or only apply REST it in a

[8] http://www.flickr.com/services/api/

limited way (e.g. the Twitter API[9]).

To clearly establish the contrast of a RPC-based service with a resource-oriented service, consider a service that allows to manage different items in an e-commerce site. Particular pieces of functionality offered would be i) to list all the items in a product category, and ii) to add an item to a particular product category. A RESTful service realizing this would make use of the HTTP GET and PUT methods to achieve the desired functionality:

- `GET http://example.com/products/someCategory/`

  would list all the products in a particular category, and deliver the representation of this resource in a format as specified e.g. by its MIME type. Similarly

- `GET http://example.com/products/someCategory/1092`

  could return the representation about a particular product within a category. Adding a new product to a certain category would operate in a similar fashion:

- ```
  PUT http://example.com/products/someCategory/
  <product>
    <name>...</name>
    <short-decription>...</short-description>
    <price> ...</price>
  </product>
  ```

- Deleting a product is done through the HTTP DELETE method:

```
DELETE http://example.com/products/someCategory/someID
```

Note that all the functionality is achieved by operating on resources through the uniform interface offered by HTTP.

All the above functionality could have been implemented differently, however would still make use of HTTP:

```
GET http://example.com/listProduct?cat=someCategory
```

```
GET http://example.com/listProduct?cat=someCategory&id=1092
```

```
POST http://example.com/addProduct?cat=someCateogry
```

```
POST http://example.com/deleteProduct?cat=someCategory&id=someID
```

The actual operation to be performed is not expressed by the HTTP method anymore. The operations are rather `listProduct`, `addProduct`, `deleteProduct`.

As a result it is not visible anymore what an operation actually does without further background knowledge. Furthermore it is not visible what data (what resource) an operation is concerned with. This *scoping information* is now contained in the URI parameters, just as arguments in a remote procedure call. While on the technical level this does not make much of a difference, it requires additional documentation and insights regarding the purpose of each of the parameters in order to be able to use such an API. In this way, such an API follows a different architectural style compared to a truly resource-oriented approach – mainly because of the lack of a uniform interface violates one of the principles of a RESTful service.

[9] http://apiwiki.twitter.com/Twitter-API-Documentation
```

Such services, which are very common even though they are not truly RESTful services, essentially still define custom operations with their associated input and output parameters. In the currently minimal service model used underneath MicroWSMO this is well reflected and thus a large number of Web APIs can be represented faithfully within MicroWSMO.

However, the current service model underlying MicroWSMO is not adequate for truly resource-oriented services, since they still have to be mapped back to an RDF representation that actually models an RPC-based service in essence. Problems arise especially because there is no notion of a resource in the RPC-based model, i.e. it is for example very unclear to what a HTTP DELETE request would apply (operation, service, etc.).Moreover, a resource-oriented service does not have a generic method, with arbitrary inputs and outputs. There is a very much predefined and actually more specific structure in place for a resource-oriented service: For example, an input message consists of

- Headerfield parameters,
- URI parameters,
- Body parameter,

which can be either optional or required.

Secondly, MicroWSMO aims at embedding machine-readable service information in ordinary HTML pages, for example the online documentation of a Web API. This is achieved by means of the hRESTS microformat, although an embedding through RDFa [11]is trivially possible as well since the RDF representation of hRESTS, WSMO-Lite, and the underlying service model can simply be used directly in that case.

In the following, we continue to discuss the different possible service architectures in order to clarify the shortfalls of the current RDF service model backing MicroWSMO and then proceed to propose a solution.

2.1 RESTFul Web Services and Resource Oriented Architectures

REST is actually an architectural style defined in [7] with HTTP being its most well known implementation. In turn a concrete architecture for RESTful web services is e.g. the Resource-Oriented Architecture (ROA) presented in [12]. Following we summarize the main criteria for a RESTful architecture and their benefits. A RESTful architecture generally uses the basic building blocks that are already in place on the Web in general:

- **Addressability** closely conforms to *scoping information* being present only in the URI alone. Scoping information defines upon which data an operation should be performed. An application or a service that follows this principle exposes a URI for every resource that clients can interact with and in this information is carried only in the URI.
- **Statelessness** guarantees that operation happens in isolation and that every request includes all the information necessary for a server to fulfil it, i.e. the server does not have to rely on information from previous operations. Combined with addressability this means that it should also be possible to address possible states of the server as resources and that they consequently should have their own URIs.
- **Connectedness** refers to the fact that representations of resources need not only be serialized data structures. They can actually represent documents that contain not just data but also connections to other resources. This leads to modelling of a specific session not as an explicit resource state on the server but rather as implicit application state on the client. This implicit application state is created by the path a client takes along a set of hyperlinks contained in the representations of specific resources served to the client.
- **Uniform Interface** means that only a defined set of operations should be used for the interaction with resources and that a service should refrain from formulating custom operations. For HTTP in a ROA those basic operations are GET, PUT, POST, DELETE. Each of those methods has a well-defined meaning when applied to a specific resource. Without this uniform interface, a Web service introduces a multiplicity of operations that for example only differ in the type of resource they refer to (`getResults`, `getProducts`, `addUser`, `addProduct`, etc.) but otherwise often mimic the functionality of existing methods.

Adherence to those principles serves as a basic guideline to truly RESTful and resource-oriented services.

By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs.

In the concrete case of the Web, If the operation to be performed should be conveyed by the HTTP method and the scoping information defined through the URI, and not rather included e.g. in the body of a request. Otherwise, a service is not truly RESTful and loses the benefits associated with the violated principle.

RPC-style services are not designed accordingly to those architectural principles, i.e. they do not focus on resources but rather on the operations to be performed. In those cases HTTP might be used as underlying protocol (e.g. for a WSDL service using SOAP [13]). RPC-style

services often only use one singular URI as endpoint and thus both the scoping information and the operation to be performed are hidden inside this envelope and do not draw upon the semantics of the HTTP methods, violating the principles of addressability and using a uniform interface. In this sense, every RPC service also defines its own vocabulary to denote operations with their respective input and output parameters.

Obviously, it is possible to identify REST-RPC hybrids. Many Web APIs would actually fall into this category. Those often contain the scoping information in the URI (in the form of parameters) but do not rely solely on the manipulation of resources through the uniform interface offered by HTTP.

In the next section, we continue to show how these different architectural styles map essentially to different RDF service models, to what degree the minimal service model currently backing up MicroWSMO covers them, and where the current shortfalls are.

2.2 Service Models

The minimal service model (MSM), see [10] for an updated version repeated in Listing 1, provides a minimal and common conceptual model formalized in RDF Schema for capturing the semantics of services whether they are WSDL-based or Web APIs (roughly RESTful). In this way, it forms a common ground for treating them homogeneously by subsequent components.

A main reason for annotating services with explicit semantics is to allow service discovery discovery services suitable for a specific task or allow reasoning over formal service descriptions. A very basic approach for this is to use SPARQL [11] in conjunction with a triple store holding service information in RDF form. For this reason, it is desirable to have an underlying service models that captures service information as precisely as necessary.

Furthermore, service information in RDF form should be as complete and self-contained as possible, so that tools do not need to take additional resources like external HTML or WSDL files into account, which again requires a more detailed and accurate service model.

The current MSM is based on the structure of WSDL (wsdl:service, wsdl:operation, wsdl:message, etc.) and in turn existing WSDL-based services map into the MSM in a straightforward way. However, this comes at the price of not being able to take the concrete underlying model (WSDL or RESTful) into account in these components, i.e. it is not easy to use the MSM to capture RESTful services fully since it still mirrors the RPC based model that is implicitly present in WSDL. As mentioned, the lack of a “resource” in the service model means that especially resource creation and deletion is not modelled very faithfully, i.e. it is not clear what exactly an HTTP DELETE would apply to since resources are not identified in the current MSM. Moreover, operations in a resource-oriented service do not match the generic concept of operations in the MSM (using generic inputs and outputs, as well as faults). Rather, they follow the structure imposed by HTTP in our case. While it is possible to rely on lifting and lowering to handle such details, this information is still lost after the transition to the MSM.

Listing 1 contains the minimal service model including SAWSDL’s syntactic properties, WSMO-Lite as a minimal extension to SAWSDL, and hRESTS’s support for Web APIs. The MSM contains SAWSDL elements for linking semantic information through the modelReference and for providing lifting and lowering mechanisms.

Furthermore, the minimal service model defines services as having a number of operations. Each operation in turn has input and output messages and faults.

Web APIs are supported through the addition of hRESTS. hRESTS allows to connect operations, as defined in the minimal service model with a concrete address (defined as URI template [14]). Furthermore, hRESTs also allows denoting the applicable HTTP methods for an operation.

The key issue in this regard is that operations (in the sense of an RPC-based service) are required as basic building block in order to capture information about a service. In a ROA however, it is not necessary to model an operations in such a generic way since, in the concrete case of SOA4All, interaction with RESTful resources is possible through the uniform interface of HTTP. In contrast to this emphasize on operations it would be required to rather model resources explicitly.

```
@prefix xsd:<http://www.w3.org/2001/XMLSchema>.
```

```
@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl:<http://www.w3.org/2002/07/owl#>.
@prefix sawsdl:<http://www.w3.org/ns/sawsdl#>.

#WSMO-Lite
@prefix wsl: <http://www.wsmo.org/ns/wsmo-lite#>.
#hRESTS
@prefix hr: <http://www.wsmo.org/ns/hrests#>.
#minimal service model
@prefix msm: <http://www.wsmo.org/ns/msm#>.

msm:Service rdf:type rdfs:Class .
msm:hasOperation rdf:type rdf:Property ;
      rdfs:domain msm:Service ;
      rdfs:range msm:Operation .
msm:Operation rdft:type rdfs:Class .
msm:hasInputMessage rdf:type rdf:Property ;
      rdfs:domain msm:Operation ;
      rdfs:range msm:Message .
msm:hasOutputMessage rdf:type rdf:Property ;
      rdfs:domain msm:Operation ;
      rdfs:range msm:Message .
msm:hasInputFault rdf:type rdf:Property ;
      rdfs:domain msm:Operation ;
      rdfs:range msm:Message .
msm:hasOutputFault rdf:type rdf:Property ;
      rdfs:domain msm:Operation ;
      rdfs:range msm:Message .
msm:Message rdf:type rdfs:Class .
msm:usesOntology rdf:type rdfs:Property ;
      rdfs:domain msm:Service ;
      rdfs:subPropertyOf rdfs:seeAlso .
msm:hasFunctionalClassification rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
msm:hasNonfunctionalProperty rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
msm:hasCondition rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
msm:hasEffect rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .

#WSMO-Lite
wsl:Ontology rdfs:subClassOf owl:Ontology .
wsl:FunctionalClassificationRoot rdfs:subClassOf rdfs:Class .
wsl:NonFunctionalParameter rdf:type rdfs:Class .
wsl:Condition rdf:type rdfs:Class .
wsl:Effect rdf:type rdfs:Class .

#hRESTS
hr:hasAddress rdf:type rdf:Property;
      rdfs:domain msm:Operation;
```

```
      rdfs:range hr:URITemplate.
hr:hasMethod rdf:type rdf:Property;
      rdfs:domain msm:Operation;
      rdfs:range xsd:string.

#datatype for URI templates
hr:URITemplate rdf:type rdfs:Datatype.

#RDF properties reflecting the SAWSDL service model
sawsdl:modelReference rdf:type rdf:Property.
sawsdl:liftingSchemaMapping rdf:type rdf:Property.
sawsdl:loweringSchemaMapping rdf:type rdf:Property.
```

Listing 1 Original Minimal Service Model

Based on these observations it becomes apparent that it is very hard to model truly resource oriented, RESTful services in an adequate fashion by using the current minimal service model. Therefore, it becomes necessary to re-factor the MSM appropriately and extend it in the direction of resource-oriented architectures in order to cover them with MicroWSMO.

Subsequently we restructure this service model with the former goals for properly modelling RESTful services in mind. Beyond this goal, the resulting service models have to be usable in precisely the same fashion within MicroWSMO, i.e. the only difference is that class names within the microformat-based annotations refer to different elements according to the underlying service model used. Furthermore, it should be possible to describe existing services in same fashion as it is done now (apart from a change of namespaces) and so the re-factored service models have to cover the same expressivity as the MSM does currently.

For that purpose, we can distinguish between different types of services according to the architectural styles identified in the previous section:

- RPC-based services, which refer to operations and their expected input- and output type, etc. This type essentially covers WSDL services.
- RESTful, resource oriented services, which employ the typical HTTP (in the context of SOA4All) methods to operate on resources. What sets this services apart is the usage of a uniform interface operating on a set of resources.
- At the intersection, we can identify services that employ HTTP and Web-APIs, however they are not really resource oriented but rather still work on operations (REST-RPC) and only use HTTP and its associated methods as protocol

Those architectural styles all require different modelling elements to represent service information. Therefore, we abstract the required vocabulary in different, modular RDF-based service models that can be combined appropriately, depending on the architectural style of a service:

- A *core service model (SM)* captures functionality common to any kind of service.
- A *RPC service model (RPCSM)*, applicable to for example to WSDL services.
- A *resource-oriented service model (ROSM),* aimed at resource oriented services and architectures.

In this sense both the RPCSM and the ROSM extend a common basic service model, on

which they cleanly layer, in different directions.

Those service models are then naturally used within service descriptions as required by the underlying service architecture Figure 1 depicts the relationship of different styles of services and how they make usage of the different service models.

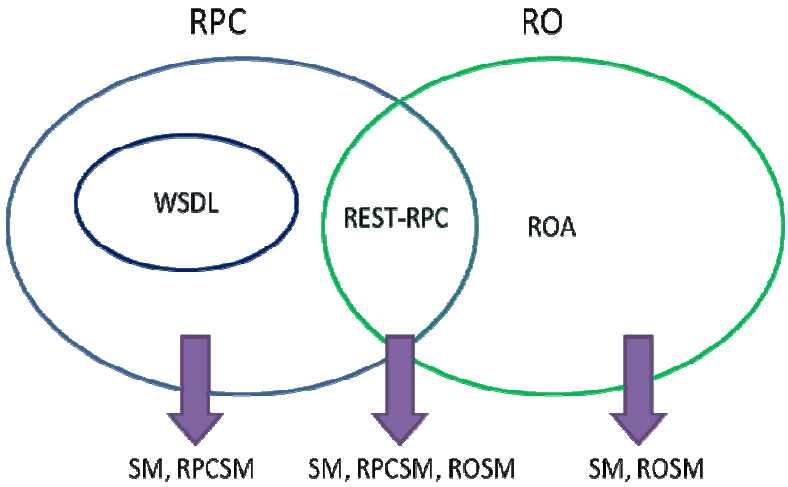

Figure 1 Styles of service paradigms and the applicable RDF Service Models

The clean layering of the different service models also results in typical use cases. Figure 2 depicts the relationship and layering of service models, along with the typical usage scenarios. Naturally, WSDL based services only make use of the core service model, along with the RPC service model. This combination of the RPCSM and the SM essentially corresponds to the expressivity that was covered by the minimal service model.

Within MicroWSMO or RDFa[11] annotations in an HTML page we can cover RPC based services and in addition to that also gain the possibility of annotating resource oriented services. Although it should be noted, that MicroWSMO and RDFa do not operate at precisely the same level. MicroWSMO adopts an ontology as underlying structure and, as usual for a microformat, simply reflects this underlying schema in a rigid way. RDFa however is a generic way of embedding arbitrary RDF information in HTML documents, and so by design more extensible and generic.

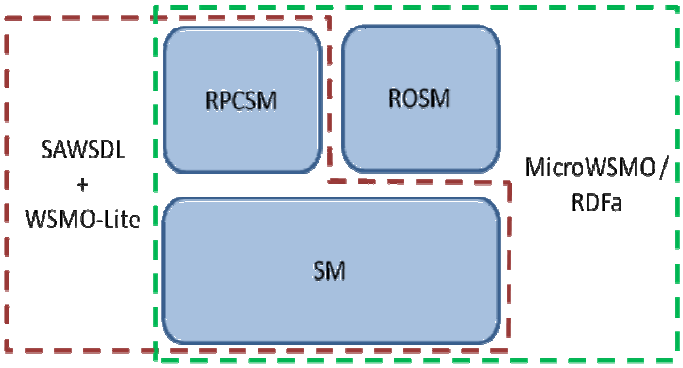

Figure 2 Layering of Service Models and usage within WSMO-Lite and MicroWSMO

In the following we discuss each of the three service models in detail and show complete definitions.

2.2.1 Core Service Model (SM)

Listing 2 shows the common core service model (SM) to be extended for both resources-oriented services as well as services operating in an RPC fashion. This common ground formalizes the WSMO-Lite notions of

- a service,
- conditions,
- effects,
- functional classification of a services,
- non-functional parameters.

Further functionality formalized in the common service model includes an RDF version of SAWSDL properties.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix sawsdl:  <http://www.w3.org/ns/sawsdl#> .
@prefix sm: <http://www.wsmo.org/ns/sm#> .

sm:Service rdf:type  rdfs:Class .

sm:Operation  rdf:type  rdfs:Class .

sm:Condition rdf:type  rdfs:Class .

sm:Effection rdf:type  rdfs:Class .

sm:FunctionalClassificationRoot rdfs:subClassOf  rdfs:Class .

sm:Ontology rdf:type rdfs:Class;
       rdfs:subClassOf  owl:Ontology .

sm:usesOntology rdf:type rdf:Property ;
    rdfs:domain  sm:Service ;
    rdfs:subPropertyOf  rdfs:seeAlso .

sm:NonfunctionalParameter rdf:type  rdfs:Class .

# SAWSDL properties
sawsdl:modelReference  rdf:type  rdf:Property .
sawsdl:liftingSchemaMapping  rdf:type  rdf:Property .
sawsdl:loweringSchemaMapping  rdf:type  rdf:Property .

sm:hasFunctionalClassification rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
sm:hasNonfunctionalProperty rdf:type rdfs:Property ;
```

```
      rdfs:subPropertyOf sawsdl:modelReference .
sm:hasCondition rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
sm:hasEffect rdf:type rdfs:Property ;
      rdfs:subPropertyOf sawsdl:modelReference .
```

Listing 2 Core Service Model

2.2.2 RPC-based Service Model (RPSM)

The RPC-based service model shown in Listing 3 captures much of the functionality formerly associated with the “minimal service model”. It is directly derived from the implicit model stemming from RPC-based services and WSDL in particular, including their associated operations with inputs and outputs.

Therefore, an operation is associated with an address (a URI or URI template [14]) to invoke it, the format of input and output messages, as well as fault messages. Those operations can then be tied to services from the core service model. This is in contrast with the ROSM, which does not define operations in this fashion; the interface of a resource is fixed.

However, operations still have HTTP methods associated with them. For this purpose we do not just use strings to denote the suitable methods but instead draw from the HTTP vocabulary formalized in RDF [15] , which is currently a W3C working draft. The inclusion of HTTP methods is essential in order to be still able to cover Web APIs that are not resource-oriented but still require the usage of particular methods.

The RPCSM layers on the SM by using common vocabulary to talk about services, used ontologies, etc. and furthermore also relies on it to specify service semantics (conditions, effects etc.), or nonfunctional properties (e.g., the price of using the service, QoS guarantees).

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix http: <http://www.w3.org/2006/http#> .
@prefix sm: <http://www.wsmo.org/ns/sm#> .
@prefix rpcsm: <http://www.wsmo.org/ns/rpcsm#> .

rpcsm:Operation rdfs:subClassOf sm:Operation .

rpcsm:hasOperation rdf:type  rdfs:Class ;
          rdfs:domain sm:Service ;
          rdfs:range rpcsm:Operation .

rpcsm:hasAddress rdf:type rdf:Property ;
         rdfs:domain rpcsm:Operation ;
         rdfs:range rpcsm:URITemplate .

rpcsm:hasMethod rdf:type rdf:Property ;
        rdfs:domain rpcsm:Operation ;
```

```
        rdfs:range http:Method.

rpcsm:Message rdf:type rdfs:Class .

rpcsm:hasInputMessage rdf:type  rdf:Property ;
    rdfs:domain sm:Operation ;
    rdfs:range rpcsm:Message .

rpcsm:hasOutputMessage rdf:type  rdf:Property ;
    rdfs:domain  sm:Operation ;
    rdfs:range  rpcsm:Message .

rpcsm:hasInputFault rdf:type  rdf:Property ;
    rdfs:domain  sm:Operation ;
    rdfs:range  rpcsm:Message .

rpcsm:hasOutputFault rdf:type  rdf:Property ;
    rdfs:domain  sm:Operation ;
    rdfs:range  rpcsm:Message .

#datatype for URI templates
rpcsm:URITemplate rdf:type rdfs:Datatype.
```

Listing 3 RPC Service Model

2.2.3 Resource Oriented Service Model (ROSM)

Again layered on the top of the common service model we formalize the resource oriented service model (ROSM) as an extension.

Its focus is to organize resources belonging to a service. Those resources can be organized in collections and have addresses (URIs) associated with them at which they can be accessed. The organization of resource in collections, which again belong to a service, allows capturing an arbitrary number of resources and attaching service semantics to them.

Furthermore, resources can again have certain (HTTP) methods associated with them, which define how it is possible to interact with a resource, which are connected through an operation. These operations are modelled in a much-more fine-grained way, since they basically only have to support the uniform interface of HTTP.

Furthermore, we can explicitly model requests and responses with their associated aspects (e.g. parameters, response codes, etc.). In theory, it is possible to express this same information in the RPCSM. However, operations would be attached directly to a service, whereas in the resource-oriented view they would be attached to ServicedResources, which form a service.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix http: <http://www.w3.org/2006/http#> .
@prefix sm: <http://www.wsmo.org/ns/sm#> .
```

```
@prefix rosm: <http://www.wsmo.org/ns/rosm#> .

rosm:ServicedResource rdfs:subClassOf rdfs:resource .

rosm:hasAddress rdf:type rdf:Property;
             rdfs:domain rosm:ServiceedResource ;
             rdfs:range rosm:Address .

rosm:Address rdf:type rdfs:Datatype .

rosm:ServicedResourceCollection rdfs:subClassOf rosm:ServicedResource .

rosm:containsResource rdf:type rdf:Property ;
                      rdfs:domain rosm:ServicedResourceCollection ;
                      rdfs:range rosm:ServicedResource .

rosm:Operation rdfs:subClassOf sm:Operation .

rosm:supportsOperation rdf:type rdf:Property ;
                       rdfs:domain rosm:ServicedResource ;
                       rdfs:range rosm:Operation .

rosm:basedOnMethod rdf:type rdf:Property ;
             rdfs:domain rosm:Operation ;
             rdfs:range http:Method .

rosm:Parameter rdf:type rdf:Class .

rosm:parameterName rdf:type rdf:Property ;
                   rdfs:domain rosm:Parameter ;
                   rdfs:range rdfs:Literal .

rosm:OptionalParameter rdfs:subClassOf rosm:Parameter .

rosm:RequiredParameter rdfs:subClassOf rosm:Parameter .

rosm:requestBodyParameter rdf:type rdf:Property;
                       rdfs:domain rosm:Operation ;
                       rdfs:range  rosm:Parameter .

rosm:requestHeaderFieldParameter rdf:type rdf:Property;
                                 rdfs:domain rosm:Operation ;
                                 rdfs:range  rosm:Parameter .

rosm:requestURIParameter rdf:type rdf:Property;
                           rdfs:domain rosm:Operation ;
                           rdfs:range  rosm:Parameter .

rosm:hasResponse rdf:type rdf:Property;
                         rdfs:domain rosm:Operation ;
                         rdfs:range  rosm:Response .
```

```
rosm:Response rdf:type rdfs:Class .

rosm:hasResponseCode rdf:type rdf:Property ;
                     rdfs:domain rosm:Response ;
                     rdfs:range http:ResponseCode .

rosm:hasBody rdf:type rdf:Property ;
                     rdfs:domain rosm:Response ;
                     rdfs:range rosm:Parameter .
```

Listing 4 Resource Oriented Service Model

3. Conclusion

In this deliverable, we introduced a revised service model to be used underneath MicroWSMO. The main purpose of this revision is to allow the faithful annotation of resource-oriented, RESTful services while at the same time achieving the following two goals:

1. To keep the functionality of MicroWSMO itself, i.e. the basic approach to using a microformat (or RDFa) as high-level syntax for the annotation of service descriptions remains unchanged. MicroWSMO can still be used in exactly the same fashion, the only source of further expressivity is the underlying service model, which is referenced from HTML class names.
2. To preserve compatibility with the old minimal service model in order to ensure the functionality of existing tools and components within the SOA4All architecture with minor modifications – unless they choose to use the ROSM. This is accomplished because no elements of the MSM are removed, they are merely encapsulated within the RPCSM under a distinct namespace, while overlapping functionality is formalized in a common service model on which the RPCSM cleanly layers.

Further next steps include alignment with ongoing work on service templates and an application for RESTful APIs and in components used within SOA4All, e.g. the crawler or service repository components.

4. References

[1] P. Hoffman and T. Bray, “Atom Publishing Format and Protocol (atompub),” *Retrieved from< http://www. ietf. org/html. charters/atompub-charter. html*, 2006.

[2] J. Kopecky, T. Vitvar, and D. Fensel, “D3.4.3 “MicroWSMO and hRests”.”

[3] R. Khare and T. Çelik, “Microformats: a pragmatic path to the semantic web,” *Proceedings of the 15th international conference on World Wide Web*, 2006, p. 866.

[4] G. Klyne, J.J. Carroll, and B. McBride, “Resource description framework (RDF): Concepts and abstract syntax,” *W3C recommendation*, vol. 10, 2004.

[5] D. Connolly and others, “Gleaning resource descriptions from dialects of languages (GRDDL),” *W3C Candidate Recommendation*, vol. 2, 2007.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web services description language (WSDL) 1.1*, 2001.

[7] R.T. Fielding, “Architectural styles and the design of network-based software architectures,” Citeseer, 2000.

[8] T. Vitvar, J. Kopecky, and D. Fensel, “Wsmo-lite: Lightweight semantic descriptions for services on the web,” *Proceedings of the Fifth European Conference on Web Services*, 2007, pp. 77–86.

[9] J. Kopeckỳ, T. Vitvar, C. Bournez, and J. Farrell, “Sawsdl: Semantic annotations for wsdl and xml schema,” *IEEE Internet Computing*, 2007, pp. 60–67.

[10] C. Pedrinaci, D. Lambert, M. Maleshkova, D. Liu, J. Domingue, and R. Krummenacher, “Adaptive Service Binding with Lightweight Semantic Web Services,” *In Service Engineering: European Research Results (S. Dustdar and F. Li eds.)*, To Appear. 2010.

[11] B. Adida and C. Commons, “RDFa in XHTML: Syntax and processing,” *Recommendation, W3C*, 2008.

[12] L. Richardson and S. Ruby, “RESTful web services,” 2007.

[13] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, H.F. Nielsen, A. Karmarkar, and Y. Lafon, *SOAP version 1.2 part 1: Messaging framework*, June, 2003.

[14] J. Gregorio, M. Hadley, M. Nottingham, and D. Orchard, “URI Template,” *Network Working Group, Internet Draft*, 2006.

[15] J. Koch and C. Velasco, “HTTP Vocabulary in RDF 1.0,” *W3C Working Draft*, Oct. 2009.

