SAPS: Semantic AtomPub-based Services

Markus Lanthaler [1]
[1] Institute for Information Systems and Computer Media
Graz University of Technology
Graz, Austria

Christian Gütl [1, 2]
[2] School of Information Systems
Curtin University of Technology
Perth, Australia

***Abstract*—The utopian promise of a uniform service landscape in the form of SOAP, WSDL, and UDDI made by service-oriented architectures (SOA) built on Web services has proven elusive. Instead more and more prominent Web service providers opted to expose their services based on the REST architectural style. Nevertheless there are still problems on formal describing, finding, and orchestrating RESTful services. While there are already a number of different approaches, none so far has managed to break out of its academic confines. Thus, we propose a novel approach to build Semantic RESTful Services based on proven technologies. The combination of those proven technologies leads to scalable and loosely coupled systems and has the additional advantage, that developers are already familiar with its functioning.**

Keywords—Semantic Web; Web services; REST; Web 3.0; SAPS; Atom; OpenSearch; Internet; SOA

I. INTRODUCTION

Service-oriented architectures (SOA) have gained increasing interest in both, research and business. Much effort has been invested to make information accessible through Web services. But, while most current and previous research efforts mostly concentrate on traditional SOAP-based Web services, the utopian promise of uniform service interface standards, metadata, and universal service registries, in the form of the SOAP, WSDL, and UDDI standards has proven elusive. This and other centralized, registry-based approaches were overwhelmed by the Web's rate of growth and the lack of a universally accepted classification scheme. Thus, the usage of SOAP-based services is mainly limited to company-internal systems and to the integration of legacy systems. Instead of SOAP-based services with their high perceived complexity, prominent Web service providers like Microsoft, Google, Yahoo, and others have opted to use lightweight, REST-style APIs.

REST is an architectural style developed specifically for the Internet. It specifies constraints to enhance performance, scalability, and resource abstraction within distributed hypermedia systems [1], [2]. But there are still problems on formal describing, finding and orchestrating RESTful services. Research on these issues has already started but none of the approaches so far has managed to break out of its academic confines.

In this paper we introduce a new approach based on Atom Publishing, OpenSearch, SAWSDL, and the Linked Open Data vocabulary to build Semantic RESTful Services. The combination of those proven, standardized technologies leads to scalable and loosely coupled systems and has the additional advantage that developers are already familiar with its functioning.

The remainder of this paper is organized as follows. First we outline REST's main advantages. In section III and IV we discuss different proposals for service interface description formats respectively proposals for semantic annotation of (RESTful) Web services. In section V we explain our approach based on a simple example. Finally, the concluding remarks are presented in section VI.

II. REPRESENTATIONAL STATE TRANSFER (REST)

Even though many successful distributed systems have been built on RPC and RPC-oriented technologies, such as SOAP, it is known for quite some time that this approach is flawed because it ignores the differences between local and remote computing [2]. Another aspect is that SOAP breaks intermediaries for caching, filtering, monitoring, etc. by abusing HTTP as a pure transport protocol while it is in fact an application protocol. However, to ensure good performance, scalability, and maintainability in Internet-scale systems these intermediaries are "must haves".

REST [1] addresses exactly these issues allowing the creation of extensible, maintainable, and loosely-coupled distributed systems at Internet-scale. The fact that the whole Web, the largest and most successful distributed system, is built on REST's principles should be sufficient evidence of REST's superior scalability and interoperability.

The central feature that distinguishes REST's architectural style from other network-based styles is its emphasis on a uniform interface. REST is defined by four interface constraints: 1) identification of resources, 2) manipulation of resources through representations, 3) self-descriptive messages, and 4) hypermedia as the engine of application state.

REST's *identification of resources* constraint, which specifies that every resource has to be addressable, makes REST a natural fit for the Semantic Web vision [3] and creates a network of Linked Data; no parallel exists for SOAP's remote method invocation. It follows that REST-based Web services are an ideal carrier for semantic data that would even provide the additional benefit of resource resolvability in human-readable formats.

The *hypermedia as the engine of application state* (HATEOAS) constraint refers to the use of hyperlinks in resource representations as a way of navigating the state machine of an application. According to Fielding [4], "a REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types. […] From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations."

While the "human Web" is unquestionably based on this type of interaction and state-control flow where very little is

known a priori, machine-to-machine communication is often based on static knowledge and tight coupling. The challenge is thus to bring some of the human Web's adaptivity to the Web of machines to allow the building of loosely coupled, reliable, and scalable systems. After all, a Web service can be seen as a special Web page meant to be consumed by an autonomous program as opposed to a human being. The Web already supports machine-to-machine communication, what is not machine-processable about the current Web is not the protocol (HTTP), it is the content.

III. Interface Description

In order for two (or more) systems to communicate successfully there has to be an agreement or contract on the used interfaces and data formats. Such a contract can either be static or dynamic. In a static contract, knowledge about the other system's model is embedded and cannot be updated without changing all involved clients. In a dynamic contract on the other hand, clients are capable of discovering knowledge about the contract at runtime and adjust accordingly. Hadley et al. [5] furthermore divide dynamic contracts into contextual and non-contextual contracts. According to them, contextual contracts can be updated in the course of a conversation depending on the application's state; conversely, non-contextual contracts are fixed and independent of the application's state.

In the traditional Remote Procedure Call (RPC) model, where all differences between local and distributed computing are hidden, usually static contracts in the form of an Interface Description Language (IDL) are used to specify those interfaces. In SOAP this is usually done by using WSDL and XML Schema. That way, automatic code generation on both, the client and the server side, are possible.

In contrast REST's HATEOAS constraint is characterized by the use of contextual contracts where the set of actions varies over time [5]. Additionally the interface variability is almost eliminated due to REST's uniform interface. In consequence REST-based services are almost exclusively described by human-readable documentation describing the URLs and the data expected as input and as output. Even though it would be possible to describe REST services with WSDL 2.0, different other approaches have been proposed. Most of them were more or less ad-hoc inventions designed to solve particular problems and haven't been updated for many years. The most recent, respectively only regularly updated proposals are, to our best knowledge, hRESTS (HTML for RESTful Services) [6] and WADL (Web Application Description Language) [7]. While WADL's approach is closely related to WSDL (the developer creates a monolithic XML file containing all the information about the service interface), hRESTS idea is quite different. hRESTS uses microformats to annotate the human-readable HTML documentation with microformats to make it machine-processable. Despite the number of proposals, none of the approaches has managed to gain wide acceptance. In practice most RESTful services are still solely described by a human-readable documentation in the form of an HTML document.

Given REST's constraints, it is arguable whether REST even needs a interface description. We argue, in contrast to the before mentioned approaches, that the description of the resource representations, i.e., the data format, combined with the use of hypermedia should be enough to achieve a high degree of automation for RESTful services.

IV. Semantic Annotation

Most of the time, the syntactic description of a service's interface is not enough as two services can have exactly the same syntactic definition but perform significantly different functions. Thus, services have to be annotated semantically; the resulting service is called a Semantic Web Service (SWS) or a Semantic RESTful Service (SRS). These supplemental semantic descriptions of the service's properties can in consequence lead to higher level of automation for tasks like discovery, composition, and invocation. Since most SWS technologies use ontologies as the underlying data model they also provide means for tackling the interoperability problem at the semantic level and, more importantly, enable the integration of Web services within the Semantic Web. After a number of efforts (we would like to refer you to [9] for an extensive survey of the different approaches) semantic annotation of SOAP-based services is now preferably addressed by the W3C recommendation Semantic Annotations for WSDL and XML Schema (SAWSDL) [8].

SAWSDL defines how to add semantic annotations to various parts of a WSDL document such as inputs, outputs, interfaces, and operations but it does not specify a language for representing the semantic models. It just defines how references to semantic models, e.g. ontologies, can be made. This lack of formal semantics hinders logic-based discovery and composition of Web services described with SAWSDL and calls for magic mediators outside the framework to resolve the semantic heterogeneities. One of the approaches to solve this issue is WSMO-Lite [10].

WSMO-Lite has been created as a lightweight service ontology to fill the SAWSDL annotations with concrete service semantics to allow bottom-up modeling of services. It describes four aspects of a Web service: 1) the *Information Model*, which defines the data model for input, output, and fault messages; 2) the *Functional Semantics*, which define the functionality, which the service offers; 3) the *Behavioral Semantics*, which define how a client has to talk to the service; and 4) the *Non-functional Descriptions*, which define non-functional properties such as quality of service or price. A major advantage of WSMO-Lite's approach is that it is not bound to a particular service description format, e.g., WSDL. As a result WSMO-Lite can be used to integrate approaches like, e.g., hRESTS (in conjunction with MicroWSMO) or SA-REST with the traditional WSDL-based service descriptions.

All the above mentioned approaches try to be as general as possible to allow the creation of suitable service annotations for a wide range of application domains. In contrast to that, there exist a number of service description formats such as OpenSearch [12] and the Atom Publishing Protocol [13] which are tailored for very specific application domains. The semantics of those descriptions are implicit, i.e., all services described by such an approach are created for the same or very similar use cases. As these two technologies are two of the key ingredients of SAPS, our approach proposed in chapter V, we describe them here in more detail.

A. Atom

The Atom suite consists of the Atom Syndication Format [14] and the Atom Publishing Protocol (also known as AtomPub) [13]. The Atom Syndication Format is a XML-based format for publishers to syndicate content in the form of

so called Web or news feeds. The Atom Publishing Protocol, on the other hand, is an application-level protocol for publishing, editing, and deleting Web resources.

The Atom Syndication Format consists of two kinds of documents: feed documents and entry documents. A feed document is, as the name suggests, the representation of an Atom feed. It contains metadata about the feed and some or all of the entries associated with the feed. An entry document describes exactly one feed item outside the context of an Atom feed. It is worth mentioning that Atom documents must be well-formed XML but are not required to be valid XML because the specification does not include a Document Type Definition (DTD) for them. Atom is designed to be an extensible format and so foreign markup (markup which is not part of the Atom vocabulary) is allowed almost anywhere in an Atom document.

The Atom Publishing Protocol describes how a feed can be manipulated by a client. The so called service document describes the location and capabilities of one or more collections, i.e., feeds, which are grouped into workspaces. That information is needed by clients for authoring to commence.

Both, the Atom Syndication Format as well as the Atom Publishing Protocol, are fully based on the REST architectural style and thus extremely Web-friendly. This, and the above mentioned extensibility, led to the adoption of AtomPub for the implementation of various kinds of Web services. The most prominent examples might be the Google Data Protocol (GData) [15] and Microsoft's Open Data Protocol (OData) [16]. They use Atom's extensibility to implement APIs for their services. It is thus arguable that Atom is one of the world's most successful RESTful Web service stories.

B. OpenSearch

OpenSearch [12] is a collection of simple formats that allow the description of search engines' interfaces as well as the publishing of search results in a format suitable for syndication and aggregation. OpenSearch allows search clients, such as Web browsers, to invoke search queries and process the responses. By now all major Web browsers support OpenSearch and use it to add new search engines to the browser's search bar. This way the user can invoke a query directly from the browser without first having to load the search engine's homepage.

OpenSearch consists of four formats: 1) the description document, 2) the URL template syntax, 3) the response elements, and 4) the Query element. The OpenSearch description document describes the web interface of a search engine in the form of a simple XML document. It may also contain some metadata such as the name of the search engine and its developer. The URL template syntax represents a parameterized form of the URL by which a search engine is queried. Simply speaking it describes the used GET parameters to invoke a query. An example of such a template looks as follows: `http://example.com/search?q={searchTerms}`. All parameters are enclosed in curly braces and are by default considered to be part of the OpenSearch template namespace. By using the XML namespace prefix convention it is possible to add new parameter names, which brings extensibility. The OpenSearch response elements are used by search engines to augment existing XML formats such as Atom and RSS with search-related metadata. Finally, the OpenSearch Query element can be used to define specific search requests that can be performed by a search client, e.g., related queries in a search result element.

V. SAPS: SEMANTIC ATOMPUB-BASED SERVICES

The attempt to standardize Web services has taken years, but still there are no clear definitions of what constitutes a service at a conceptual level. While a number of different approaches have been proposed, none so far has managed to break out of its academic confines. We argue that the lack of acceptance of those approaches stems from the fact that they do not provide any imminent incentive and thus experience a classic chicken-and-egg problem. No services are semantically described because there are no applications making use of that information and no applications are developed because there are no semantically annotated services. Facebook's recently introduced Open Graph Protocol clearly shows the willingness of Web site publishers to semantically annotate their content if it is easy enough and if there is an imminent incentive to do so; more than 50,000 sites implemented the protocol within the first week of its publication [17]. Another factor for the lacking acceptance is the high complexity of most of the approaches.

To solve those and other problems, we propose *SAPS (Semantic AtomPub-based Services)*, an approach which tries to create the machine counterpart of the human Web by combining different proven technologies. This will lead to scalable and loosely coupled systems and has the additional advantage that developers are already familiar with its functioning. This way, apart from the knowledge about the application domain, the client needs to know only the entry point (the URL of the service's homepage) and the used media types. The client then proceeds through the service by looking at one response at a time, each time evaluating how best to proceed given its overall goal and the available transitions. This strictly follows the HATEOAS constraint as described in section II. The most challenging part is how to communicate the domain knowledge without overburdening developers.

The basic idea of SAPS is the use of semantically annotated schemas to allow clients to understand the payload. To ease the implementation of SAPS-based services as well as to enhance their interoperability, Atom Publishing and OpenSearch, two widely known technologies, are used.

A. Motivating Example

In order to illustrate the principles of the proposed approach, we use a service similar to the well-known Restbucks [18]. It is basically an online shop which allows looking for products, ordering them, and of course paying the order. The inspiration for that problem domain came from Gregor Hohpe's observation [19] on how a busy coffee shop works. In his popular article Hohpe talks about synchronous and asynchronous messaging, transactions, exception handling, and scaling the message-processing pipeline in an everyday situation.

B. Basic Concepts and Principles

SAPS combines the Atom Syndication Format, the Atom Publishing Protocol, and the OpenSearch format and adds a semantic layer on top of them as shown in Figure 1. SAPS' semantic layer consists of semantically annotated schemas describing the payload (the exchanged data), e.g., XML Schema with SAWSDL annotations or a semantically annotated JSON schema as described at the end of section C. This allows the usage of these normally domain-specific application formats in application domains different from content syndication and interface description of search engines.

SAPS follows strictly the Atom specification. The use of most Atom elements is self-explanatory, but some elements require further clarification in the context of SAPS. E.g., in most of the cases it is not obvious how `atom:title`, `atom:author`, and `atom:summary` shall be used. SAPS takes a pragmatic approach for these fields: the title element should be used to create some-kind of human-readable representation of the item (e.g., if a product is represented its name could be used), the same applies for the summary element where required. The author element is a bit trickier; most of the time this element will either be empty or set by the server to some constant value. The other element which it might not be obvious how to use, is the `app:categories` element, in fact, it is not even obvious how to use it in traditional Atom documents as the specification does not assign any meaning to the content of this element [14]. In SAPS the categories element is used to give hints about the data in a collection by setting the scheme to the used ontology (e.g. the GoodRelations ontology) and the term to a used concept (e.g. GoodRelations' Offering class). To describe the service's search interface, OpenSearch is used.

Given that Atom was designed as an extensible format, it is easy to integrate OpenSearch. The OpenSearch documents describing the search interface can either be directly embedded in Atom documents or be linked to by using Atom's `link` tag. Atom uses Internet Media Types to define the acceptable payload formats. Unfortunately often this is not enough as media types can be too general to be of practical use and most of the time it is not practical to create a new proprietary media type. E.g., the popular `application/xml` media type is not concrete enough to allow the automatic construction of messages. Thus, more granular information in the form of schemas has to be added. This is done by SAPS' newly defined attribute `saps:schema`. This allows specifying, additionally to the media type, a schema which can then in turn be used to automatically generate the payloads.

To improve the efficiency SAPS introduces a `saps:etag` attribute which can be used in feeds to specify the ETag of every item. The `saps:etag` is equivalent to HTTP's `ETag` header and can be used for conditional retrievals (`GET` using the `If-None-Match` HTTP header) as well as for optimistic concurrency control when updating or deleting entries (`PUT` or `DELETE` requests using HTTP's `If-Match` header).

C. Illustrative Implementation of the Motivating Example

In this section we will show an illustrative implementation of the motivating example presented in section A to give a better understanding of SAPS.

First of all, we need to define the different involved resources. By analyzing the workflows described in the example we are able to extract the following resources: orders, products, and payments. Those resources are accessed by three actors: the customers, the cashiers, and the baristas. It is obvious that the actors have different privileges and need to authenticate themselves to the system, but for the sake of simplicity this example does not address those issues and assumes that every actor performs only the actions he is allowed to.

Just like for a normal website we start by building the service's homepage. This is done by creating an Atom service document containing a reference to an OpenSearch description document; both enhanced with semantic information. The service homepage includes collections for the available products and the orders. Payments are not included in the homepage because there is no use case to justify that.

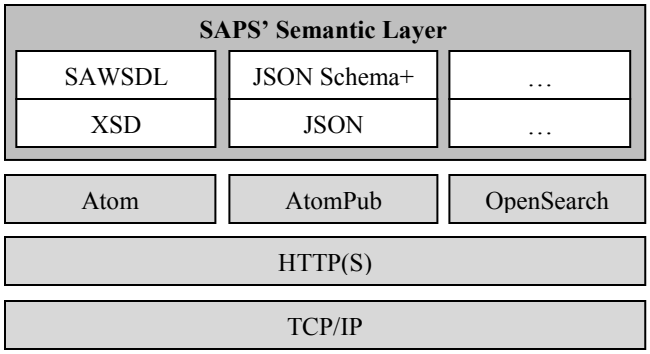

Figure 1. SAPS' layer cake

The products collection has an empty `accept` element which specifies that the products collection does not support the creation of new entries. The category of the collection is set to GoodRelations' [20] `Offering` class so that clients are able to understand the meaning (the semantics) of this collection. These categories could also be used to store information about the behavior as well as non-functional descriptions of the service.

```
<?xml version="1.0" encoding='utf-8'?>
<service xmlns="http://www.w3.org/2007/app"
        xmlns:atom="http://www.w3.org/2005/Atom"
        xmlns:saps="http://www.purl.org/saps">
 <workspace>
   <atom:title>Coffee Shop Service</atom:title>
   <collection href="/products" >
     <atom:title>Products</atom:title>
     <accept />
     <categories fixed="yes">
       <atom:category
         scheme="http://purl.org/goodrelations/v1#"
         term="http://purl.org/goodrelations/v1#Offering"
         label="Products" />
     </categories>
    <atom:link rel="search"
      type="application/opensearchdescription+xml"
      href="http://api.example.com/productsearch.xml"/>
   </collection>
   <collection href="/orders" >
     <atom:title>Orders</atom:title>
     <accept saps:schema="/schemas/purchase-order.xsd">
       application/xml
     </accept>
     <categories fixed="yes">
       <atom:category scheme=
          "http://www.purl.org/net/ontology/order.owl#"
         term="http://www.purl.org/net/ontology/
           order.owl#PurchaseOrder"
         label="Orders" />
     </categories>
   </collection>
 </workspace>
</service>
```

Furthermore, the products collection contains a link to an OpenSearch description document which allows searching for products either by generic search terms (full-text search) or by an EAN code. As shown in the following OpenSearch description, the EAN code is linked to GoodRelations' `hasEAN_UCC-13` property to make it understandable to a client.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription
     xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Product Search</ShortName>
  <Description>Search for products</Description>
  <Url xmlns:gr="http://purl.org/goodrelations/v1#"
    type="application/atom+xml;type=feed"
    template="http://api.example.com/?q=
     {searchTerms?}&amp;ean={gr:hasEAN_UCC-13?}" />
</OpenSearchDescription>
```

The orders collection is, in contrast to the products collection, writable. It accepts new orders in the form of XML documents complying with the `purchase-order.xsd` XML schema. The category is set to SchemaWeb's [21] `PurchaseOrder` class. We intentionally used two different ontologies for the products and the orders to highlight that the client of a service has to implement mappings between the different ontologies, i.e., it has to "understand" their meaning. If there is a one-to-one mapping between different ontologies the service provider is able to use both ontologies in the semantic annotation to point this out.

A client is now able to search for products and create a purchase order. It can, e.g., search for a product with the EAN code *2300000010015* by issuing a GET request as described in the OpenSearch description document:

```
GET /?q=&ean=2300000010015 HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/atom+xml;type=feed

<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
 xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
  <title type="text">Search for EAN 2300000010015</title>
  <updated>2010-02-08T12:29:29Z</updated>
  <author>
    <name>example.com</name>
  </author>
  <link rel="search"
    type="application/opensearchdescription+xml"
    href="http://api.example.com/productsearch.xml"/>
  <opensearch:totalResults>1</opensearch:totalResults>
  <opensearch:startIndex>1</opensearch:startIndex>
  <opensearch:itemsPerPage>10</opensearch:itemsPerPage>
  <opensearch:Query
     xmlns:gr="http://purl.org/goodrelations/v1#"
     role="request"
     searchTerms=""
     ean="gr:hasEAN_UCC-13"
     startPage="1" />
  <entry>
    <title>Cappuccino</title>
    <summary>A hot cappuccino for 1.99 EUR</summary>
    <id>tag:example.org,2003:3.2397</id>
    <link rel="alternate" type="text/html"
       href="http://example.org/products/P4197.html" />
    <link rel="alternate" type="application/xml"
       href="http://api.example.org/products/P4197.xml"/>
    <updated>2009-07-31T12:29:29Z</updated>
    <published>2008-12-13T08:29:29-04:00</published>
    <content type="application/xml"
      <product xmlns="http://example.com/products"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-
           instance"
        xsi:schemaLocation="http://example.com/products
          http://example.com/products.xsd">
        <id>http://api.example.org/products/P4197</id>
        <label>Cappuccino</label>
        <ean>2300000010015</ean>
        <sku>P4197</sku>
        <price>1.99</price>
        <description>...</description>
      </product>
    </content>
  </entry>
</feed>
```

The associated product XML schema is defined as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
  <xsd:element name="product" type="ProductType"
     sawsdl:modelReference="http://purl.org/goodrelations/
       v1#Offering" />
  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:sequence>
      <xsd:element name="id" type="xsd:anyURI" />
        <xsd:element name="label" type="xsd:string"
           sawsdl:modelReference="http://www.w3.org/2000/
              01/rdf-schema#label" />
        <xsd:element name="ean" type="xsd:string"
           sawsdl:modelReference="http://purl.org/
              goodrelations/v1#hasEAN_UCC-13" />
        <xsd:element name="sku" type="xsd:string"
           sawsdl:modelReference="
              http://purl.org/
                goodrelations/v1#hasStockKeepingUnit
              http://www.purl.org/net/
                ontology/order.owl#PartNumber" />
        <xsd:element name="price" type="xsd:decimal"
           sawsdl:modelReference="http://purl.org/
              goodrelations/v1#hasCurrencyValue" />
        <xsd:element name="description" type="xsd:string"
           sawsdl:modelReference="http://www.w3.org/2000/
              01/rdf-schema#comment" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Due to the SAWSDL annotations in the product and the order XML schema, the client is able to extract the SKU of the product and create a purchase order by sending an order XML document to the server. Obviously the client has to have knowledge about desired quantities and acceptable price ranges but that is part of the client's business logic and thus beyond the scope of this paper.

Finally the server will guide the client to the payment by returning a link for the payment:

```
<link rel="next http://example.com/ontology/payment"
   href="/order/1684/payment" type="application/xml"
   saps:schema="/schemas/payment.xsd" title="Payment" />
```

In order to guarantee a loose coupling of the client and the server, the schemas have to be retrieved and interpreted on-the-fly at runtime and not at design time. This is in contrast to the traditional SOAP-practice where the schemas are used at design time to generate static proxy classes to interact with the service.

It should be highlighted that this approach is not limited to XML representations. Any other media type can be used as well as long as the client is able to automatically create representations in that format. It is, e.g., imaginable to use JSON as the transport format by using the `application/json` media type in conjunction with semantically annotated JSON schemas. A possible solution would be to enhance proposals like Kris Zyp's JSON Schema [22] with a `modelReference` attribute.

```
{
  "name": "Product",
  "properties": {
    "id": {
      "description": "Product identifier (URL)",
      "type": "string"
    },
    "label": {
      "description": "Name of the product",
      "type": "string",
      "modelReference": "http://www.w3.org/2000/01/
          rdf-schema#label"
    },
    "ean":{
      "description": "The EAN code of the product",
      "type": "string",
      "modelReference": "http://purl.org/
          goodrelations/v1#hasEAN_UCC-13"
    }
  }
}
```

D. Comparison to Related Work

Instead of trying to semantically describe existing services, as most previous proposals do, SAPS introduces a new model to create semantic Web services encouraging developers to reuse existing ontologies as much as possible. The services created according to SAPS are, by design, completely RESTful.

Since SAPS is based on the Atom protocol suite it is similar to previous efforts such as Google's Data Protocol (GData) [15] or Microsoft's Open Data Protocol (OData) [16]. The difference to GData is that the allowed elements are described in a machine-readable manner in the form of a schema instead of defining them just in a human-readable form. This makes it more similar to Microsoft's OData. But, in contrast to that approach, SAPS uses standardized building blocks such as XML Schema (XSD) and SAWSDL to do so instead of defining a completely new and proprietary data model as Microsoft does. This not only creates interoperable and standard-conform services, but also aligns SAPS to the vision of a Semantic Web.

E. Current Limitations

This paper introduces a first version of SAPS. Due to its early development stage it still has a number of limitations, especially regarding efficiency. The current version of SAPS does not support any kind of partial requests or responses, i.e., retrieving or updating just part of a representation instead of the whole representation. This is closely related to the limited expressiveness for describing the search interface. OpenSearch currently does not support any operators in search queries. Last but not least no batch operations are currently supported; we are currently evaluating the use of multipart requests to do so.

VI. Conclusions and Future Work

In this paper we proposed SAPS, a new approach to create semantically annotated Web services by combining different proven technologies. We believe its simplicity and nativity to both the Web and the Semantic Web will result in higher acceptance compared to previous efforts for building semantic RESTful services. The design of our approach strictly follows REST's constraints leading to reliable, scalable, loosely coupled, and, in consequence, reusable systems. By basing the approach on well-known technologies we hope to lower the barrier for developers to provide semantic Web services.

For future work we aim to develop prototype modeling-tools as well as client libraries and conduct performance evaluations in scenarios of different complexity. The next version of SAPS should clearly address the current limitations described in the previous section as well as examine other optimizations.

References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Dept. Inform. Comput. Sci., Univ. California, Irvine, USA, 2000.

[2] M. Lanthaler and C. Guetl, "Towards a RESTful service ecosystem - perspectives and challenges," in Proc. of the 4th IEEE Int. Conf. on Digital Ecosystems and Technologies (DEST), Dubai, U.A.E, 2010.

[3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," Scientific Amer., vol. 284, no. 5, pp. 34-43, May 2001.

[4] R. T. Fielding, "REST APIs must be hypertext-driven," Untangled musings of Roy T. Fielding. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[5] M. Hadley, S. Pericas-Geertsen, and P. Sandoz, "Exploring hypermedia support in Jersey," in Proc. of the 1st Int. Workshop on RESTful Design - WS-REST '10, pp. 10-15, Raleigh, North Carolina.

[6] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: an HTML microformat for describing RESTful Web services", in Proc. 2008 IEEE/WIC/ACM Int. Conf. Web Intelligence and Intelligent Agent Technology, pp. 619-625, 2008.

[7] M. J. Hadley, "Web Application Description Language (WADL)", Aug. 2009. [Online]. Available: http://www.w3.org/Submission/2009/SUBM-wadl-20090831/.

[8] Semantic Annotations for WSDL and XML Schema (SAWSDL), W3C Recommendation, 2007.

[9] M. Lanthaler, M. Granitzer, and C. Gütl, "Semantic Web services: state of the art," in Proc. of the IADIS Int. Conf. on Internet Technologies & Society (ITS 2010), Perth, Australia, 2010.

[10] T. Vitvar, J. Kopecký, J. Viskova, and D. Fensel, "WSMO-Lite annotations for Web services," in Proc. 5th European Semantic Web Conf., LNCS 5021, pp. 674-689, Tenerife, Spain, 2010.

[11] A. P. Sheth, K Gomadam, and J. Lathem, "SA-REST: semantically interoperable and easier-to-use services and mashups", IEEE Internet Computing 11, vol. 6, pp. 84-87, Nov./Dec. 2007.

[12] C. DeWitt, "OpenSearch 1.1 Draft 4" [Online]. Available: http://www.opensearch.org/Specifications/OpenSearch/1.1.

[13] The Atom Publishing Protocol. [Online]. Available: http://tools.ietf.org/html/rfc5023.

[14] The Atom Syndication Format. [Online]. Available: http://tools.ietf.org/html/rfc4287.

[15] Google Data Protocol. [Online]. Available: http://code.google.com/apis/gdata/.

[16] Open Data Protocol. [Online]. Available: http://www.odata.org/.

[17] S. L. Huang, "After f8 - resources for building the personalized Web," Facebook Developer Blog. [Online]. Retrieved from http://developers.facebook.com/blog/post/379 on July 7, 2010.

[18] J. Webber, S. Parastatidis, and I. Robinson, "How to GET a cup of coffee," InfoQ. [Online]. Retrieved on January 5, 2010 from http://www.infoq.com/articles/webber-rest-workflow.

[19] G. Hohpe, "Your coffee shop doesn't use two-phase commit," IEEE Software, vol. 22, issue 2, pp. 64-66, 2008.

[20] M. Hepp, "GoodRelations language reference," E-Business and Web Science Research Group. Retrieved on March 11, 2010 from http://www.heppnetz.de/ontologies/goodrelations/v1.

[21] SchemaWeb PurchaseOrder Ontology. Retrieved on March 11, 2010 from http://www.schemaweb.info/schema/SchemaInfo.aspx?id=253.

[22] K. Zyp, "A JSON media type for describing the structure and meaning of JSON," IETF Internet-Draft. [Online]. Available: http://tools.ietf.org/html/draft-zyp-json-schema-02.