Description and Interaction of RESTful Services for Automatic Discovery and Execution

Abstract

Many have left their footprints on the field of semantic RESTful Web service description. Albeit some of the propositions are even W3C Recommendations, none of the proposed standards could gain significant adoption with Web service providers. Some approaches were supposedly too complex and verbose, others were considered not RESTful, and some failed to reach a significant majority of API providers for a combination of the reasons above. While we neither have the silver bullet for universal Web service description, with this paper, we want to suggest a lightweight approach called RESTdesc. It expresses the semantics of Web services by pre- and post-conditions in simple N3 rules, and integrates existing standards and conventions such as Link headers, HTTP OPTIONS, and URI templates for discovery and interaction. This approach keeps the complexity to a minimum, yet still enables service descriptions with full semantic expressiveness. A sample implementation on the topic of multimedia Web services verifies the effectiveness of our approach.

1 Introduction

The immense diversity of various multimedia analysis and processing algorithms makes it difficult to integrate them in an automated platform to perform compound tasks. Yet, recent research has indicated the importance of the fusion of different techniques [2]. It is impossible to make different algorithms interoperate if there are no agreements or guidelines on how communication should happen. A coordinating platform can only select algorithms based on their capabilities in presence of a formal description detailing their preconditions and postconditions.

In this paper, we show how to lift multimedia algorithms to the level of Semantic Web services[1] with a formal description mechanism that follows a pragmatic approach. Rather than reinventing the existing methodologies, which focus on *technical* process details, we want to express an algorithm's *functionality* in a way that captures its functionality without requiring lengthy specifications. Our intention is to use existing standards such as the HTTP protocol, Link headers, and URI templates and apply common best practices for implementing multimedia algorithms as true Semantic Web services. The aim is a versatile description and communication model, enabling fully automated service discovery and execution, even under changing conditions. The sole starting point is a Web address of a server, required additional information is gathered at runtime.

[1] We use the terms "API" and "(Web) service" synonymously throughout this paper.

Can a client just follow its nose—like humans do—and access the right service by reasoning? We will explain our approach by three real-world multimedia use cases, each of which represents challenges that are currently not fully addressed by alternative techniques. They will illustrate the power of our method and demonstrate to the reader that its apparent simplicity accommodates far-reaching possibilities.

The remainder of this paper is structured as follows: Section 2 gives an overview on related work. Section 3 introduces RESTful Multimedia Web services. Section 4 describes our RESTdesc approach for Semantic Web service description. Section 5 details how our approach can be used for automatic service discovery. Section 6 shows how our approach is able to adapt to change and react dynamically on errors. The paper terminates with Section 7, which provides a conclusion and gives an outlook on future work.

We have implemented a sample multimedia Web service with mock data that follows our description approach. It is available at our RESTdesc testing website[2].

2 Related Work

2.1 Web Service Description Language

The description of Web services has a long history. The XML-based Web Service Description Language (WSDL, [9, 10]) provided one of the first models. WSDL focuses on the communicational aspect of Web services, looking from a message-oriented point of view. The details of the message format are written down in a very verbose way and concretized to actual bindings such as SOAP [19] or plain HTTP [13]. Finally, the description can contain endpoints, which implement the specified bindings.

For our use case, we spot two major problems with the use of WSDL. First, WSDL only provides the mechanisms

[2] RESTdesc testing website: `http://restdesc.no.de/`.

to characterize the technical implementation of Web services. It does not provide the means to capture the functionality of a service. For example, a service that counts the number of words in a text will be described by WSDL as an interface, which accepts a string and outputs an integer. Clearly, an infinite number of algorithms share those input and output properties, so this information is insufficient to infer any meaning or functionality.

Secondly, in practice, a WSDL description is used to generate module source code automatically, which is then compiled into a larger program. If the description changes, the program no longer works, even if such a change leaves the functionality intact. A concrete example of such brittleness is the switch from 32 to 64 bit integer identifiers that occurred at some point in Google's AdWords API, a small change in the service's WSDL file that required the complete recompilation of the relevant pieces of source code [33]. This indicates that WSDL is not well adapted to real-world circumstantial changes.

The above problems indicate why WSDL cannot offer automatic service discovery at runtime and why we should investigate other possibilities.

2.2 Semantic Annotations for WSDL

The W3C Recommendation named Semantic Annotations for WSDL and XML Schema (SAWSDL, [24]) describes a way how to add semantic annotations to various parts of a WSDL document such as interfaces and operations, and input and output message structures. In addition to that, Web services can be assigned a category with the objective of making them discoverable in a central registry of Web services. SAWSDL also defines an annotation mechanism for specifying the data mapping of XML Schema [15, 34] types to and from ontologies, often referred to as *up-* and *down-lifting*. While the standard fulfills parts of our requirements, it inherits all the disadvantages from WSDL, specifically its brittleness and verbosity.

2.3 Web Application Description Language

The Web Application Description Language (WADL, [20]) is another Web service description format, also XML-based, which does not degrade HTTP to a tunneling mechanism for SOAP, but advocates proper use of all the aspects of the HTTP protocol. Services that behave in this way are oftentimes (incorrectly[3]) called RESTful [14], the properties of which we will explain in Section 3. While WSDL 2.0 is also capable of specifying bindings to RESTful endpoints, it still requires the abstractions that enable bindings to SOAP and others. WADL, on the other hand, was tailored to the needs of RESTful services, but only exists as a W3C Member Submission and will most likely never reach the W3C Recommendation status of WSDL 2.0 [27].

[3] We prefer the term "HTTP interface", where most API providers use the buzzword "RESTful API".

In addition to that, WADL still suffers from the same problem: it does emphasize the technical properties of the underlying service and does not leave any room for the semantics of the task it performs. This also means that there is no way to automatically discover services based on the desired functionality. Therefore, there is no reason why WADL would be used any differently than WSDL, as also argued by Joe Gregorio in [18].

The main criticism by the REST community, however, is that WADL does document beforehand what, according to the REST principles [14], should be discovered dynamically at run-time. One of the fundamental properties of REST is the so-called *hypermedia constraint*, which basically can be summarized as the constraint that each server response should contain the possible next steps the client can take, since the application state is not stored on the client. It should be noted that WADL could be used in this way at run-time, yet most current usage continues to be beforehand.

2.4 Semantic Markup for Web Services

OWL-S [29] is a an OWL [32] ontology for describing Semantic Web services in RDF [22]. A service description consists of three parts: a profile, a model and a grounding. Some aspects of profile and model are very similar, in the sense that they both describe input, output, preconditions and effects. The difference is that the profile is used for discovery, while the model is used to control the interaction.

Here, for the first time, we have an actual focus on the functionality of a service which is separate from how the interaction happens. However, whether this separation was successful is debatable, since there is no way to enforce the consistency of profile and model of a single service. Finally, the grounding part specifies the implementation of the service. The OWL-S submission defines a grounding to WSDL, but other groundings are possible (e.g., [36]). This means that the OWL-S description describes the functionality, whereas its grounding describes the communication.

At least, this is what it is supposed to do. OWL-S input and output types provide more or less the equivalent of what a WSDL message format contains, albeit with RDF types, so there is only a minimal added semantic value on that level. The real possibilities lie in the use of preconditions and postconditions (the latter under the form of result effects), which allow to express complex relationships between input and output values, finally capturing the semantics and functionality of the service.

Unfortunately, there is no obligation to use these conditions and the OWL-S submission only mentions the rule languages KIF [16], DRS [30] and SWRL [21], in order of increasing verbosity. Extensions to more light-weight rule languages, such as Notation3 Logic [6], are possible [36]. The real problem here is that none of those languages are integrated into the main service description, but rather form subdocuments within it, which require a separate interpretation. The conditions thus live in another context, which are solely linked by identifiers but not by semantics. As a result, agents lacking support for the used rule language could parse the OWL-S constructs in a service description document and skip the parts they fail to understand—effectively ignoring important conditions they should reckon with[4].

Furthermore, while OWL-S offers functional descriptions capable of automatic discovery of the capabilities of a single service, it does not provide mechanisms to express its relation to other services. Also, descriptions contain redundancies and require a fair amount of vocabulary, even to express conceptually simple services, and rely on external groundings for technical implementations.

2.5 Linked Open Services

The obligation to make explicit the relation between input and output is present within the Linked Open Service (LOS, [26]) principles. However, these principles also put constraints on how the service should behave, dictating its interaction pattern. In essence, it requires wrapping a SPARQL endpoint around a service. While we have used a similar approach for multimedia algorithms in the past [36], regular REST services offer far more flexibility and target more general Web data consumption.

2.6 Resource Linking Language

The Resource Linking Language (ReLL, [1]) aims to provide a natural mapping from RESTful services to RDF. The authors recognize the issues regarding RESTful service descriptions in general and provide an excellent discussion thereof. ReLL differs from our approach in that it only offers *"static description of RESTful services that does not cover [...] new resources or identification and access schemes"*, whereas we specifically aim to address these cases in the context of automated service discovery and consumption. Our work therefore strives to *"include the set of preconditions that must be satisfied by a client to be able to consume a service"*, together with the postconditions that occur as a result of the service call, as detailed in Subsection 4.2.

2.7 Universal Description, Discovery, and Integration

The XML-based OASIS standard Universal Description, Discovery, and Integration (UDDI, [4]) was developed to enable the definition of a set of services supporting the discovery and description of (i) businesses, organizations, and other Web service providers, (ii) the Web services that those institutions offer, and finally (iii) the technical interfaces, which may be used to access those services. UDDI is based on a (at the time of writing of the specification) common set of industry standards, including HTTP, XML, XML Schema, and SOAP. The standard was designed to allow for the description and discovery of both public services and non-public in-house services. It was meant to be used as a service broker where parties interested in a special service could go to and retrieve a list of service providers offering the desired service (for example, shipping address verification). Such services would be described in the so-called Green Pages, including not only technical details, but also contact details of the Web service provider.

While for various reasons out of scope of this paper UDDI could not gain the adoption its creators had hoped for, the overall idea of automatically being able to select a service from a (not necessarily central) registry of services still seems useful to us. We will show in Section 5 how we imagine this idea to work decentralized and dynamically using our approach.

3 RESTful Multimedia Services

When we say RESTful service invocation, we refer to the following REST principles [14]:

- Servers and clients are separated from each other by a uniform interface. Both servers and clients have well-defined responsibilities, also referred to as *separation of concerns*. This is to guarantee maximum independence from the one and the other.
- All client requests are *stateless*, this means that each request from a client has all the information that the server needs to process it.
- Responses must define themselves as *cacheable* or not using standard HTTP caching techniques.
- When layered systems (like load-balancing) are used, this fact must not be exposed to the API user.

In RESTful APIs, resources are identified by URIs. A resource is to be differentiated from its representation. For example, a set of RDF triples (the resource) might be represented in different serializations (syntaxes), such as RDF/XML or Turtle. When one of these representations gets manipulated, there is enough information to manipulate the represented resource, given the permission to do so. Messages need to be self-descriptive, for example, the media type of a message needs to make clear what can be done with this message. Each representation needs to communicate relevant related resources, or next steps the client can take at each state.

To make this clearer, we introduce two related multimedia services, one for face detection, and the other for face recognition. A user agent can upload a photo to the face detection service and use it to check for the existence of faces in the uploaded image. If faces are found, the user agent can use the face recognition service to try to find out more details on the persons whose faces are contained in the image. Each image is considered a

[4] This behavior resembles that of Web browsers without JavaScript support: they parse and render HTML but ignore any `script` tags. Service descriptions, in contrast, *always* require a full interpretation for correct functionality.

resource, for example represented by a binary image file (like `/photos/1`). Each face is a resource, for example represented by an RDF document serialized in Turtle, or a cropped version of the entire image showing only the particular face (like `/photos/1/faces/1`). Each person is a resource (like `/photos/1/persons/1`), for example represented by a string of the person's name. Some of the potential next steps after detecting faces could be, to follow a link to a Web service that allows for recognizing these faces, or starting from the first person on an image, to follow a link to the next person on the image. We will use these two sample Web services, namely a face detection and a face recognition Web service, throughout the paper.

4 RESTdesc Semantic Description

4.1 Motivation

The answer to the question whether Semantic Web service description and Web service discovery are necessary needs to be split up in two parts.

On the one hand there is the question whether Web service description is needed. In RESTful systems, the common opinion is that each message should be self-descriptive enough so that user agents can make sense of each message, given a documented media type that the message is serialized in. On a pure technical layer this works well. For example, let us imagine a very simple image search engine that simply returns the most adequate image of media type `image/gif` as the result to a query, similar to Google's *"I'm feeling lucky"* functionality. This gives the user agent enough information to process the response with its Graphics Interchange Format (GIF) library, however, a priori it is not clear that the image stands in a relation to a search query that the user agent has used as an input. Therefore OpenSearch [11] defines a description format, which can be used to describe a search engine so that it can be used by search client applications. While we could perfectly use OpenSearch to describe this search API, even slight variations of the API semantics render its use impossible. For instance, let us imagine a Web font preview API where you give the name of a Web font as an input, and get a GIF image with a preview of the text *"The quick brown fox jumps over the lazy dog"* in that very Web font as an output. There is currently no universal way to describe the exact functionality of such API, and yet it might be desirable for a Web font vendor to announce its availability.

The second question is whether automatic discovery of Web services is needed. The first approach for automatic service discovery was UDDI, outlined in Section 2.7. It was driven by the vision that central service registries would serve as so-called Green Pages for parties interested in a specific service. The problem with this approach, however, is that companies do not work this way. There is always a human being involved in the process [25]. We see the potential of service discovery in the generation and run-time supervision of automatic execution plans as outlined in [37], a task that can highly profit from discoverable service descriptions.

The before-mentioned OpenSearch protocol allows for an interesting use case of service discovery. Web pages can reference an OpenSearch description that user agents can process and offer site-specific search automatically. If we adapt this idea to our approach, user agents could dynamically offer services related to a current resource, if the resource points to a service description.

4.2 Deriving a functional description

By now, it is clear that we aim to provide a semantic method to express the functionality of a service—as well as its communication—in a concise way that appeals to humans and can be processed automatically. The word "semantic" obviously hints at the Semantic Web [7] and its core language RDF [22].

Let us first revise what we actually want to express. Continuing the example of Section 3, an informal expression for photo retrieval could be:

I can retrieve a photo by going to `/photos/` and appending its identifier. (1)

An intuitive formalization of the above would be:

$$\begin{aligned} &\mathit{hasUri}(\mathit{request}, \{\text{"/photos/"}, \mathit{id}\}) \wedge \\ &\mathit{hasResponse}(\mathit{request}, \mathit{resp}) \wedge \mathit{represents}(\mathit{resp}, \mathit{photo}) \\ &\qquad \wedge \mathit{photoId}(\mathit{photo}, \mathit{id}) \end{aligned} \tag{2}$$

This is straightforward to represent in RDF:

```
:request :uri ("/photos/" :id);
   :response [ :represents [:photoId :id] ].
```
(3)

Upon closer inspection, it is clear that the formalization (2)—and thus its RDF counterpart (3)—does not contain all the semantics of the informal expression (1). While (1) implies (2), the opposite implication (2) ⇒ (1) is broken, and thus the equivalence does not hold. Indeed, fragment (3) states that there exists *some* request which returns the photograph with the identifier specified in its URI. It does however not convey the implicit intention of (1) that *all* requests with this URI structure behave the same way. This is a problem of existential ∃ versus universal ∀ quantification, which has important consequences that should be dealt with formally.

Revising (4) with quantifiers gives:

$$\begin{aligned} &\forall \mathit{photo} : \exists\, \mathit{id}, \mathit{request}, \mathit{uri}, \mathit{resp} : \\ &\quad \mathit{hasUri}(\mathit{request}, \{\text{"/photos/"}, \mathit{id}\}) \wedge \\ &\quad \mathit{hasResponse}(\mathit{request}, \mathit{resp}) \wedge \mathit{represents}(\mathit{resp}, \mathit{photo}) \\ &\qquad \wedge \mathit{photoId}(\mathit{photo}, \mathit{id}) \end{aligned} \tag{4}$$

However, this still remains insufficient, because the universal quantification introduces the claim that *every* photograph in the world possesses an identifier—a false statement for the majority of photographs, with the exception of those uploaded to the server. Similarly, requests exist for such photographs only. Looking back at the informal expression (1), we now spot the (again, implicit) assumption that the photograph we want to retrieve has a known identifier.

Therefore, our last revision of the formal expression takes into account this notion as follows:

$$\begin{aligned}&\forall\, photo, id : photoId(photo, id) \Rightarrow \exists\, request, uri, resp :\\ &\quad hasUri(request, \{\text{“}\texttt{/photos/}\text{”}, id\})\\ &\quad \wedge\, hasResponse(request, resp)\\ &\quad \wedge\, represents(resp, photo)\end{aligned} \tag{5}$$

The above expression now corresponds to the intended meaning of (1): that a representation of every photograph with an identifier can be retrieved by following the constructed URI. Now the issue of expressing (5) in RDF remains. The original RDF specification [22] does not include a form of quantifiers. Although some attempts have been made in the past (e.g., [31]), the most successful intuitive is the W3C submission Notation3 (N3, [5]), which as an added benefit also includes syntactical support for implications.

Expressing (5) in Notation3 gives:

```
@forAll :photo, :id.
@forSome :request.
 {:photo :photoId :id.}
     :implies
 {:request :uri ("/photos/" :id);
           :response [ :represents :photo ].}.  (6)
```

Note the automatic existential quantification of blank nodes. By turning the request also in a blank node and using the full expressive power of Notation3, we can conveniently write (6) as:

```
 {?photo :photoId ?id.}
 =>
 {_:request :uri ("/photos/" ?id);
            :response [ :represents ?photo ].}.  (7)
```

This minimal syntax fully reflects the functionality of the service as intended by (1).

4.3 RESTdesc description format

With the syntax and required concepts in mind, we now look at existing recommendations, proposals, and vocabularies that we can integrate to obtain an interchangeable description format.

Since RESTful services are centered around the correct use of the HTTP protocol, one of the obvious elements we need is a way to describe HTTP requests. The *HTTP Vocabulary in RDF* [23] is already widespread, and in addition to that also is in W3C Working Draft status. It defines all the necessary concepts to rigorously describe HTTP messages, their structure, and their relationships.

The resource-oriented nature of RESTful services implies the use of descriptive URIs, based on a structure specific to each server. Therefore, we should be able to express the relationship between a resource and its URI. In an Internet-Draft, the IETF describes the concept of URI templates [17] to refer to a category of resources.

Below is an example of a URI template for a person in a photograph:

```
http://example.org/photos/{photoId}/persons/{personId}
```

The identifiers between the curly braces are variables, which can be assigned a value. For example, to get the person with identifier 3 on photograph 241, the URI gets expanded to:

```
http://example.org/photos/241/persons/3
```

While URI templates are still in draft status, many implementations and applications exist. In consequence we decided that we should include them in our design.

Finally, we need a way to tie the URI templates to HTTP request parameters such as the request URI. Also, some additional template semantics are required, for instance to describe what the response body contains. Since such a vocabulary was not available yet, we created the *HTTP template* ontology[5].

Listing 1 shows the final description of the photo retrieval service. On a high level, we see the precondition, followed by the request and the postcondition. Concepts detailing precise semantics of the service are expressed in a *server-specific vocabulary*[6] (in this case, photo identifiers) or by reusing publicly available vocabularies (here, for people and depictions). The precondition thus states that an object with a photo identifier is required. In the postcondition, we use the HTTP vocabulary to describe a `GET` request and its associated response. Finally, we use the HTTP template ontology to specify the URI template, and the contents of the response.

Contrary to its appearance, this short description conveys a vast amount of semantic information. Of course, most importantly, there is the explicit relation expressing precisely how the input relates to the output. An alternative way to look at the implication is to state that the specified request only exists in presence of a photograph. The semantics of the quantification have been highlighted in Listing 2, which contains the same description with the explicit quantifier syntax (prefixes from this and further listings omitted for clarity). The incorporation of the URI template is also particularly strong: the variables in the URI have been bound to the actual values that will be present during execution. Interesting here is that these variables, due to the server-specific ontology, do not only have an *associated data type*, but *fully linked semantics*. For instance, if the server describes the `photoId` predicate by specifying its range as integers and its domain as photographs, this information is propagated into the URI template. Also note that

[5] Located at `http://purl.org/restdesc/uri-template`.

[6] It is not obligatory to detail the server-specific vocabulary in an ontology. Its consistent use across different descriptions may suffice for interpretation and composition.

we do not need an ontology for services: the description is complete by the expression of its functionality.

Listings 3 to 5 show example descriptions of other services on the same server. For photo upload (Listing 3), we see the prerequisite is to have an image. Note that the service description author is free to use any vocabulary, in this case the *FOAF* ontology [8]. Since the request URI is fixed, no URI template was used. The response, in contrast, will have a `Location` header with a URI containing the photo identifier. For the request, we specify the format of the `POST` body. Note how the precondition of photo retrieval (Listing 1) naturally follows from the postcondition of photo upload, hinting at a possible causality.

This effect is also visible in Listing 4 and Listing 5, which both demonstrate the ease of expressing complex conditions. The required expressions involve a complicated indirection (e.g., "the photograph contains a region that depicts a person"), yet they can be understood quite easily, while the formal semantics are sound.

When we overlook all of the above, it becomes apparent that RESTdesc descriptions are a simple and elegant way of describing Web services in an integrated semantic manner. They capture the functional aspects formally without resorting to complex artifices. The use of the HTTP vocabulary and semantic identifiers was taken from previous work [35], as well as the use of Notation3 conditions [36], both which were extended and combined into a single method. The resulting RESTdesc descriptions can be used for automatic discovery, service composition, and execution. These and other aspects will be demonstrated in Subsection 5.2.

4.4 Automated interpretation and composition

An interesting fact about Notation3 implications is that, besides the *descriptive/declarative* semantics we have used so far, they also entail *operational* semantics. This means that, given a reasoner that is able to make *modus ponens* inferences, the following action takes place:

$$\frac{P \Rightarrow Q, P}{Q} \tag{8}$$

This is a very relevant property for RESTdesc descriptions, which *enables context-based service discovery.*

For example, we might want to know what we can do on a server given the situation where we have an image. RESTdesc makes this a trivial task. The triple (9) below expresses our current condition:

```
<http://example.org/photo.jpg> a foaf:Image.
```
(9)

It is also the precondition of photo upload (Listing 3). Consequently, using modus ponens (8), we can derive the postcondition of photo upload. Yet it does not stop there. The statements of the postcondition can also trigger other inferences. In the end, the result chain is:

– we can upload the photo, upon which it will receive an identifier;

```
@prefix : <http://restdesc.no.de/ontology#>.
@prefix http: <http://www.w3.org/2006/http#>.
@prefix tmpl: <http://purl.org/restdesc/http-template#>.

{
  ?photo :photoId ?id.
}
=>
{
  _:request http:methodName "GET";
            tmpl:requestURI ("/photos/" ?photoId);
            http:resp [ tmpl:represents ?photo ].
}.
```

Listing 1: RESTdesc description of photo retrieval

```
@prefix log: <http://www.w3.org/2000/10/swap/log#>.

@forAll :photo, :id.            # ∀ photo, id :
@forSome :request, :response.
{
  :photo :photoId :id.          # photoId(photo,id)
}
log:implies                     # ⇒ ∃ request,r : resp(request,r)
{                               #     ∧ represents(r,photo) [...]
  :request [...] http:resp :response.
  :response tmpl:represents :photo.
}.
```

Listing 2: Listing 1 with explicit quantifiers

```
@prefix foaf: <http://xmlns.com/foaf/>.

{
  ?photo a foaf:Image.
}
=>
{
  _:request http:methodName "POST";
       http:requestURI "/photos";
       http:body [ tmpl:formData ("photo=" ?photo) ];
       http:resp [ tmpl:location ("/photos/" ?photoId) ].

  ?photo :photoId ?photoId.
}.
```

Listing 3: RESTdesc description of photo upload

```
{
  ?photo :photoId ?photoId.
}
=>
{
  _:request http:methodName "GET";
          tmpl:requestURI ("/photos/" ?photoId "/faces");
          http:resp [ tmpl:representsMultiple ?region ].

  ?region foaf:depicts [ a foaf:Person ];
          :regionId _:regionId;
          :belongsTo ?photo.
}.
```

Listing 4: RESTdesc description of face detection

```
{
  _:region foaf:depicts ?person;
           :regionId ?regionId;
           :belongsTo [:photoId ?photoId].
}
=>
{
  _:request http:methodName "GET";
            tmpl:requestURI
              ("/photos/" ?photoId "/people/" ?regionId);
            http:resp [ tmpl:represents ?person ].

  ?person foaf:name _:personName.
}.
```

Listing 5: RESTdesc description of face recognition

– we can use this identifier to receive the photo;
– we can use this identifier to detect faces within it;
– we can then ask the server to recognize these faces.

In addition to *what* steps we can take, the inference process also tells us *how* to take this steps by listing the concrete HTTP requests.

These and other examples can be verified online using the EYE Semantic Web reasoner [12] on the interactive RESTdesc test website[2].

An even more interesting approach is to add a goal, in addition to a —starting point (9). If we indeed want to know who is depicted in the photograph, our query might be:

```
<http://example.org/photo.jpg> foaf:depicts ?person.
```

The proof of the reasoner for this query forms a list of ordered steps to obtain the desired results, again with detailed instructions on how to execute these steps. This differs from the previous output, which was just an unordered list of possible actions. Here, the result is an actual execution plan, instructing to first upload the photo, then ask for detected faces, and finally find out the associated persons [37].

It is apparent that RESTdesc descriptions provide a powerful and instant way to deal with automated interpretation and composition of different Web services.

4.5 Compatibility and automatic translation

One outstanding property of RESTdesc is that it, capturing the complete functionality of the service, can be converted into a multitude of other formats. After all, the functionality description, combined with invocation information, contains the maximal superset of what needs to be known to machines about a service.

Is it important to realize that RESTdesc starts from RESTful principles from which the invocation aspects follow. For instance, the fact that `GET` and `HEAD` operations are safe and idempotent, whereas `POST` is not, conveys necessary information to generate certain descriptions.

One relevant example is the translation to the Composition of Identifier Names vocabulary (CoIN, [28]). The CoIN vocabulary *"defines a set of classes and properties used to describe what properties of a resource constitute components of a URI"*. This information is present in the RESTdesc service descriptions, due to its use of URI templates which are bound to concrete variables. Automatic translation can be seen in action on our website[2].

It is also possible to generate WSDL, WADL, or OWL-S descriptions, as all the information—besides human-targeted elements such as labels and textual information—are readily available. In fact, human-centric textual descriptions could also be generated based on RESTdesc and ontological information. Listing 1 could translate to "given an identifier, retrieves a representation of the photograph with that identifier."

By means of content negotiation, a description format understood by the client can be returned upon request.

5 RESTdesc Service Discovery

In this section we will show with the help of the concrete example of a face detection and recognition API introduced before how starting from a single URI one can follow one's nose to explore the capabilities of an API.

5.1 On Web Service Discovery

Web Service discovery can be seen as the process of locating a suitable Web service for a given task. Typically in classic WS-* architectures there are several options for this process, which can involve Web services registering themselves with a central registry (as in Section 2.7 with UDDI), or Web services exposing their capabilities using the Web Service Description Language (WSDL, [9,10]). On a related note, the Web Services Inspection Language (WSIL, [3]) has been proposed to list groups of Web services and their endpoints. When we say Web service discovery, we currently limit ourselves to enable discovery of Web services by following one's nose from a given start URI by resolving links and making sense of Notation3 service invocation descriptions. However, RESTdesc is a very powerful concept, as given just one starting URI, the full reasoning chain of available Web services is enabled, not constrained to Web services on the same domain.

5.2 Learning About one's OPTIONS

An execution plan can be dynamically created by a user agent that is given a concrete task like "identify all persons in a certain photo" (see Subsection 4.4). The user agent first starts to check out its options on the Web service's base URI `/`, as shown in Listing 6. From there, the user agent discovers that there is a link of type "index" to `/photos`, which in turn it checks its options on. As can be seen in Listing 7 (edited slightly for clearness), the user agent has indicated that it accepts responses of type "text/n3; charset=utf-8", and therefore is given instructions that new items can be added to the index by means of a POST request. Next, the user agent uploads the photo to the server, and is notified that the photo has been stored at the location `/photos/1`, as stated in the Location header of the 201 Created-type response. The user agent then executes an OPTIONS call to that location in order to find out what it can do with the uploaded photo. The response contains Notation3 instructions on how to detect potentially contained faces in the photo by performing a GET request to `/photos/1/faces`. Upon execution of that GET call, each of the detected faces has its own URI, for example `/photos/1/faces/1`, where a cropped image region of just that face is available, as can be derived from the Link header. In addition to that, a different Link header reveals that the persons behind each of the faces can be recognized by navigating to the particular person's URI, for example `/photos/1/persons/1`.

```
$ curl -i -H "Accept: text/n3; charset=utf-8" \\
       -X OPTIONS http://restdesc.no.de
HTTP/1.1 200 OK
Link: <./>; rel=self,
      <./photos>; rel=index; type=text/n3;charset=utf-8
Allow: GET, OPTIONS, HEAD
```

Listing 6: An OPTIONS call on an API's base URI

```
$ curl -i -H "Accept: text/n3; charset=utf-8" \\
       -X OPTIONS http://restdesc.no.de/photos
HTTP/1.1 200 OK
Link: <./>;
      rel=index;
      rel=self
Allow: GET, OPTIONS, HEAD, POST
Content-Type: text/n3; charset=utf-8

@prefix : <http://restdesc.no.de/ontology#>.
@prefix http: <http://www.w3.org/2006/http#>.
@prefix tmpl: <http://purl.org/restdesc/http-template#>.
@prefix foaf: <http://xmlns.com/foaf/>.

{
  ?photo a foaf:Image.
} => {
  _:request http:methodName "POST";
    http:requestURI "/photos";
    http:body [ tmpl:formData ("photo=" ?photo) ];
    http:resp [ tmpl:location ("/photos/" ?photoId) ].
  ?photo :photoId _:photoId. }.
}

{
  ?photo :photoId ?id.
} => {
  _:request http:methodName "GET";
    tmpl:requestURI ("/photos/" ?photoId);
    http:resp [ tmpl:represents ?photo ].
}.
```

Listing 7: An OPTIONS call on a discovered index path from an API's base URI while accepting text/n3 responses

```
$ curl -i -F "photo=@./obama-gillard.jpg" \\
       http://restdesc.no.de/photos
HTTP/1.1 100 Continue

HTTP/1.1 201 Created
Location: http://restdesc.no.de/photos/1
Content-Type: text/html

Your photo was uploaded:
<a href="http://restdesc.no.de/photos/1">
  http://restdesc.no.de/photos/1
</a>
```

Listing 8: A POST call in order to upload a photo discovered via the Notation3 data from the previous request

```
$ curl -i -H "Accept: */*" -X OPTIONS \\
       http://restdesc.no.de/photos/1
HTTP/1.1 200 OK
Content-Type: image/jpg
Allow: GET, OPTIONS, HEAD
Link: <http://restdesc.no.de/restdesc/photos/1/faces>;
      rel="http://dbpedia.org/resource/Face_detection";
      title="contained faces";
      type="text/n3"

{
  ?photo :photoId ?photoId.
} => {
  _:request http:methodName "GET";
    tmpl:requestURI ("/photos/" ?photoId "/faces");
    http:resp [ tmpl:representsMultiple ?region ].

  ?region foaf:depicts [ a foaf:Person ];
    :regionId _:regionId;
    :belongsTo ?photo.
].

{
  _:region foaf:depicts ?person;
    :regionId ?regionId;
    :belongsTo [:photoId ?photoId].
}
=>
{
  _:request http:methodName "GET";
    tmpl:requestURI
        ("/photos/" ?photoId "/people/" ?regionId);
    http:resp [ tmpl:represents ?person ].
  ?person foaf:name _:personName.
}.
```

Listing 9: An OPTIONS call to find out about one's options with a concrete photo

6 Adapting to change and errors

In this section, we describe how our approach reacts to change and errors in a forgiving and tolerant way. We investigate how RESTdesc descriptions ensure clients can adapt to long-term changes and possible errors.

6.1 Focus on runtime decisions

RESTdesc is designed from the start to be consumed at runtime and to make decisions only at the moment this becomes necessary. We want to mimic the flexibility of human beings browsing the Web, who follow hyperlinks to achieve a predefined goal—which is perhaps adjusted along the way. Mostly, humans have a high-level plan, that is refined as each step becomes more and more concrete, and if necessary, steps can be taken back.

6.2 Fluent change coping

This focus on the runtime aspect makes RESTdesc well adapted to changes. The key to that functionality is offered by the operational semantics of the integrated pre- and postconditions: in order for a RESTdesc description to apply, its preconditions must be satisfied. This is inherently different from static descriptions, where the description can be interpreted separately. This adaptive behavior does not only work for small interface changes, even more complex situations can be handled gracefully.

We will briefly consider some examples. For instance, suppose the server changes its URI structure (which is similar to the change of data format presented in 2.1). This does not pose a problem, since the URI templating mechanism fills out the parameters dynamically. A more subtle change, for instance, if the server only wants to accept images with maximum dimensions 1000 × 1000, can be handled on two levels. The preconditions will state this requirement on the image, and should the client attempt a larger image, the server will return an error code. More interestingly, the server can also return hyperlinks to image resizing services, which can help the client to work out a solution on its own. Even changes that affect the process structure can be handled transparently: for example, if the face recognition algorithm needs grayscale input images, the preconditions can list this requirement and the server could return service links in a similar way.

The central idea is that the client uses descriptions in a dynamic way: *"Given a certain input, how can the service descriptions reach my predefined goal?"*. The server furthermore aims to support the client by providing information on how to reach subsequent steps. This vision differs completely from the traditional static approach, which cannot deal with changing contexts.

6.3 Adaptive error handling

WSDL and OWL-S provided very detailed ways to specify error conditions and faults. This does not correspond to the human strategy when browsing the Web: we just try, and if something does not work out as expected, we continue, possibly aided by hyperlinks on last visited pages. The underlying rationale is simple: if we had to anticipate *every* possible error (page not found, irrelevant information, network failure, ...), we might as well give up before we start. Consequently, our approach is to handle errors dynamically as they arise[7], guided by the service itself where applicable.

An important benefit of this pragmatic error handling is that all causes can be dealt with in an uniform manner. Clients assume services will handle their request as described. If an exception or error should occur, it is detected and remedied, irrespective of whether it could have been expected. For example, a face detection request can fail for numerous reasons: the image does not exist or has been deleted, no faces detected in the image, the server is unavailable or crashes, ... The central idea is that there is no point in anticipating foreseeable errors, since errors can always occur. A RESTdesc description details necessary preconditions for executing a request, but it does not strive to handle exceptional situations because it can never cover all of them.

The REST practice of correctly using HTTP status codes forms the corner stone of error detection. They can precisely identify the source of the problem (client, request, or server), its temporal scope (temporary or permanent), and offer additional information (even in case of success). What we suggest is that the service should supply hyperlinks that can help the client to remedy the problem, similar to service discovery in Section 5. For example, depending on the error, the server could list the photo upload API (image does not exist), or an alternative API with a different face detection algorithm (no faces detected), or even another server (server unavailable) in its responses.

[7] This approach works well with actions that do not involve commitment, i.e. information exchange, which we primarily focus on. Data fetching and even state-changing actions, such as image upload, are thus perfectly possible. In case of irreversible actions with binding consequences (e.g., reservations), special care might be necessary. But then again, not all errors can be foreseen, which renders this topic inherently difficult and not generally solvable.

7 Conclusion and Future Work

In this paper, we have shown a proposal for a Web service description and interaction approach for automatic Web service discovery and execution called RESTdesc. Our approach builds on top of RESTful principles and consists of a semantic mark-up model, offering a formal description of a service's functionality, with extensive flexibility, and an HTTP-based discovery method of services, both within a domain of related services, and also beyond. It is to be noted that in order for our approach to work, obeying to REST principles is essential for the APIs that RESTdesc should be applied to. We have demonstrated the feasibility and the pragmatism of our proposal with a concrete implementation. In addition to that, and unlike OWL-S, our approach is integrated in the normal Web service data flow.

Future work will be to prove the applicability of the approach to a broad family of existing RESTful Web services. We are also planning to investigate ways to link to external services that not necessarily follow our approach, including multi-domain-spanning Web services. In addition to that, we want to perform an in-depth study of compatibility and exchangeability with other standards and practices (namely with WSDL, WADL, and OWL-S). Currently we are at the very beginnings of our work towards allowing for complex automated execution plan creation including the creation of automated clients against RESTdesc-described services. With this paper we have laid a humble foundation stone for semantic Web service description. Future versions of RESTdesc will most probably encourage the decoupled use of the method, meaning that instead of relying on URI templates (which allow for a certain degree of freedom, but still introduce a form of tight coupling) we shift the URI descriptions into the Link headers, and only specify the relation of those Link headers to the result in the server response.

References

1. Alarcón, R., Wilde, E.: Linking Data from RESTful Services. Proceedings of the 3rd International Workshop on Linked Data on the Web (2010)
2. Atrey, P., Hossain, M., El Saddik, A., S Kankanhalli, M.: Multimodal fusion for multimedia analysis: a survey. Multimedia Systems (2010), `http://www.springerlink.com/index/E31M71152774R630.pdf`
3. Ballinger, K., Brittenham, P., Malhotra, A., Nagy, W.A., Pharies, S.: Web Services Inspection Language (WS-Inspection) 1.0 , *http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html* (2001), `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`
4. Bellwood, T., Capell, S., Clement, L., Colgrave, J., Dovey, M.J., Feygin, D., Hately, A., Kochman, R., Macias, P., Novotny, M., Paolucci, M., von Riegen, C., Rogers, T., Sycara, K., Wenzel, P., Wu, Z.: UDDI Version 3.0.2 (Oct 2004), `http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm`

5. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax. W3C Team Submission (Mar 2011), http://www.w3.org/TeamSubmission/n3/
6. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3Logic: A logical framework for the World Wide Web. Theory and Practice of Logic Programming 8(3), 249–269 (2008)
7. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284(5), 34 (2001)
8. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.97. Namespace document (January 2010), http://xmlns.com/foaf/spec/20100101.html
9. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation (Jun 2007), http://xml.coverpages.org/wsdl20000929.html
10. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.0 (Sep 2000), http://xml.coverpages.org/wsdl20000929.html
11. Clinton, D.: OpenSearch Specification 1.1 Draft 3 (2007), http://www.opensearch.org/Specifications/OpenSearch/1.1
12. De Roo, J.: Euler proof mechanism, http://eulersharp.sourceforge.net/
13. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. Request for Comments: 2616 (Jun 1999), http://www.ietf.org/rfc/rfc2616.txt
14. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. ACM Transactions on Internet Technology 2(2), 115–150 (May 2002), http://dx.doi.org/10.1145/514183.514185
15. Gao, S., Sperberg-McQueen, C.M., Thompson, H.S., Mendelsohn, N., Beech, D., Maloney, M.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620 (June 2008)
16. Genereth, M.R.: Knowledge interchange format. Draft Proposed American National Standard, http://logic.stanford.edu/kif/dpans.html
17. Gregorio, J., Fielding, R., Hadley, M., Notthingham, M.: URI Template (Mar 2010), http://tools.ietf.org/html/draft-gregorio-uritemplate-04
18. Gregorio, J.: Do we need WADL? (Jun 2007), http://bitworking.org/news/193/Do-we-need-WADL
19. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation (Apr 2007), http://www.w3.org/TR/2007/REC-soap12-part1-20070427/
20. Hadley, M.: Web Application Description Language. W3C Member Submission (Aug 2009), http://www.w3.org/Submission/wadl/
21. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S.: Swrl: A Semantic Web rule language combining OWL and RuleML. W3C Member Submission (May 2004), http://www.w3.org/Submission/SWRL/
22. Klyne, G., Carrol, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (Feb 2004), http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
23. Koch, J., Velasco, C.A.: HTTP Vocabulary in RDF 1.0. W3C Working Draft (Oct 2009), http://www.w3.org/TR/HTTP-in-RDF10/
24. Kopecký, J., Vitvar, T., Bournez, C., Farrell, J.: SAWSDL: Semantic Annotations for WSDL and XML Schema. IEEE Internet Computing 11, 60–67 (2007)
25. Krill, P.: Microsoft, IBM, SAP discontinue UDDI registry effort. InfoWorld, http://www.infoworld.com/d/architecture/microsoft-ibm-sap-discontinue-uddi-registry-effort-777
26. Krummenacher, R., Norton, B., Marte, A.: Towards Linked Open Services and Processes. Proceedings of FIS'2010 pp. 68–77 (Jul 2010)
27. Lafon, Y.: Team Comment on the "Web Application Description Language" Submission (Oct 2009), http://www.w3.org/Submission/2009/03/Comment
28. Lindström, N.: The CoIN Vocabulary (May 2011), http://court.googlecode.com/hg/resources/docs/coin/spec.html
29. Martin, D., Burstein, M., Hobbs, J., Lassila, O.: OWL-S: Semantic Markup for Web Services. W3C Member Submission (Nov 2004), http://www.w3.org/Submission/OWL-S/
30. McDermott, D.: DRS: A Set of Conventions for Representing Logical Languages in RDF (Jan 2004), http://cs-www.cs.yale.edu/homes/dvm/daml/DRSguide.pdf
31. Mcdermott, D., Dou, D.: Representing disjunction and quantifiers in RDF. In: In Proceedings of International Semantic Web Conference 2002. pp. 250–263 (2002)
32. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language Overview. W3C Recommendation (Feb 2004), http://www.w3.org/TR/2004/REC-owl-features-20040210/
33. Minar, N.: Why SOAP sucks (Nov 2006), http://www.somebits.com/weblog/tech/bad/whySoapSucks.html
34. Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C.M., Thompson, H.S.: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. World Wide Web Consortium, Working Draft WD-xmlschema11-2-20080620 (June 2008)
35. Steiner, T., Algermissen, J.: Fulfilling the Hypermedia Constraint via HTTP OPTIONS, The HTTP Vocabulary In RDF, And Link Headers. Proceedings of the Second International Workshop on RESTful design (Mar 2011)
36. Verborgh, R., Van Deursen, D., De Roo, J., Mannens, E., Van de Walle, R.: SPARQL Endpoints as Front-end for Multimedia Processing Algorithms. Proceedings of the Fourth International Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (Nov 2010)
37. Verborgh, R., Van Deursen, D., Mannens, E., Poppe, C., Van de Walle, R.: Enabling context-aware multimedia annotation by a novel generic semantic problem-solving platform. Multimedia Tools and Applications special issue on Multimedia and Semantic Technologies for Future Computing Environments (2011)