Lightweight Semantics for Automating the Invocation of Web APIs

Maria Maleshkova, Carlos Pedrinaci, Ning Li, Jacek Kopecky, John Domingue
Knowledge Media Institute (KMi), The Open University
{m.maleshkova, c.pedrinaci, n.li, j.kopecky, j.b.domingue}@open.ac.uk

***Abstract*—The past few years have been marked by the rapid increase in popularity and use of Web APIs as indicated by the growing number of available APIs and the multitude of applications built on top of them. The development and evolution of applications based on Web APIs is, however, hampered by the lack of automation achievable with current technologies. In this paper, we focus in particular on invocation, which as of now requires manual implementation of custom-tailored clients for each individual API. We present an approach for employing lightweight semantics for supporting the automated invocation of Web APIs. We investigate current Web API description forms and conduct an analysis of the requirements for a description model capable of supporting unified API invocation. In the light of these results, we propose a shared API description model that overcomes the current heterogeneity of the documentation and provides common grounds for enhancing APIs with semantic annotations that facilitate a general automated invocation solution. We evaluate the applicability of our approach by determining the coverage provided by our description model and via a prototypical implementation of an invocation engine.**

***Index Terms*—Web APIs; Semantic Web Services; Service Invocation;**

I. Introduction

The past few years have been marked by a trend towards a simpler approach for developing and exposing Web service APIs, moving away from traditional services based on SOAP and WSDL. Instead of relying on the rather complex WS-* specification stack, current Web service providers are inspired by a technology that is based on adopting the original design principles of the World Wide Web [1] to the world of services on the Web. The result is the current proliferation of Web APIs that rely directly on the interaction primitives provided by the HTTP protocol, with data payloads transmitted directly as part of the HTTP requests and responses.

Despite their growing importance, Web APIs are still facing a number of limitations. In contrast to traditional Web services, whose invocation relies on the information provided in the WSDL [2] documentation and is supported by a stack of specifications, the development of Web APIs is not guided by standards. In fact, only about a third of the APIs are currently conforming to the REST principles, while the majority ignore these best-practices and define interfaces in terms of operations instead of resources [3]. Moreover, providers implement and publish programmable interfaces in any way they see fit, commonly documenting them in human-oriented descriptions such as HTML webpages. The lack of a unified approach directly influences invocation because some APIs require the construction of a URI by filling in specific parameter values, while sometimes key information such as the input datatypes or the used HTTP method is missing. The absence of a common structured language for describing Web APIs is addressed by some initial proposals such as WSDL 2.0 [2] and WADL [4], which would provide a good basis for a common invocation solution, however, unfortunately they are not being adopted. Therefore, lightweight annotations over Web API descriptions [5], [6] have been developed as means for overcoming the existing heterogeneity and providing basic support for service task automation. Both MicroWSMO [5] and SA-REST [6] rely on syntactically structuring the description by marking service properties within the HTML and subsequently liking these to semantic entities, based on adapting the SAWSDL [7] approach. The importance of invocation support has already been recognised by some API provides, who deliver custom client libraries in order to ease the use of individual APIs or a particular type of APIs, such as strictly RESTful ones. However, even with these, further implementation work would still be required when developing applications based on service compositions. There are a number of mashup frameworks and "pipe"-based solutions, where a user interface allows for composing a set of services that are pre-adapted to the platform [1]. Therefore, currently API consumers need to manually process and interpret the available documentation and produce custom invocation solutions that are rarely reusable.

With the proliferation of Web APIs, invocation is becoming a key task. In this paper, we present a pioneering contribution in the area of Web APIs invocation, namely, to the best of our knowledge, the very first description model and invocation engine capable of invoking Web APIs in a unified way. We provide a detailed analysis of the process of invoking Web APIs and of the information necessary to automate it. In the light of these results, we propose a shared API description model that overcomes the current heterogeneity of the documentation and provides a common ground for adding semantic annotations for supporting invocation. Finally, we provide a proof-of-concept implementation in the form of our invocation engine OmniVoke [8] and we evaluate the applicability of our approach by determining its coverage of common types of Web APIs.

[1]Yahoo Pipes (http://pipes.yahoo.com/pipes/), Google App Engine's Mashup Editor (http://code.google.com/appengine/), Deri Pipes (http://pipes.deri.org/)

TABLE I: Requirements Coverage

Description	R1: HTTP Method	R2: Operation Based	R3: Param. URI	R4: Service-Op. Address	R5: Input Grounding
HTML Doc.	60.4%	68%	96.4%[1]	N/A	100%
WADL	Yes	No	Yes	Yes	Yes
WSDL 2.0	Yes	Yes	Yes	Yes	Yes
MicroWSMO	Yes	Yes	Yes	No	No
SA-REST	Yes	No	Yes	No	No

	R6: Lo Mapping	R7: Opt. Param & Further	R8: Msg. Parts	R9: Li Mapping	R10: Custom Errors
HTML Doc.	N/A	61%-opt. param	N/A	N/A	∼50%
WADL	No	Yes	Yes	No	No[2]
WSDL 2.0	No	Yes	Yes	No	No[2]
MicroWSMO	Yes	No	No	Yes	Yes
SA-REST	Yes	No	No	Yes	No

II. Describing Web APIs for Supporting Invocation

Despite their popularity, the use of Web APIs is currently a challenging task. Since the majority of the documentation is provided in a human-oriented form as part of webpages, API users have to invest a lot of manual effort into finding services, interpreting their descriptions and realising hard-wired implementations. In this section we focus on deriving the requirements (marked with '**R**') for a semantic Web API description capable of supporting the automated invocation, so that given its description, an API can directly be invoked by our invocation engine, without further implementation efforts or completion of manual tasks. A Web API request is implementation-wise equivalent to an HTTP request. In fact, no matter what the underlying technology is, manual invocation by the user or programatic, the invocation of an API comprises the following three steps:

1. **Construct HTTP request** (identify the HTTP Method, construct invocation URI, construct HTTP body and header, prepare the input data)
2. **Actual invocation**
3. **Process the HTTP response** (response handling, process the output data, present the output, error handling)

HTTP method. As part of our previous work, we found out that currently about 40% of the APIs do not state the HTTP method to be used [3]. This is possibly because providers assume that the method to use is GET, especially for APIs that can be invoked directly through parameterising the URI. Existing approaches, such as MicroWSMO and SA-REST already include the HTTP method and since it is necessary for invocation, **R1:** the HTTP method should be specified.

Construct invocation URI. Even though, RESTful services and Web APIs are often used as synonyms, actually only about a third of the APIs are truly RESTful [3] and the majority of the APIs (67%) are described in terms of operations. Therefore, **R2:** the description formalism should be based on service and operation definition in order to guarantee a larger coverage. The definition of parameters is also an important part of the URI, therefore **R3:** support for parameterised URIs is also necessary. It is also common that one API has a number of operations, that share the same domain as part of the invocation URI, so **R4:** assign an address to the service and this address can be overwritten or further specialised by the definition of individual operation addresses.

Construct HTTP body and header. Even though, many APIs transmit the input data directly as part of the invocation URI, it is also very common, especially in the cases were entries are created or published, that the data payload is sent as part of the HTTP body. In fact, about one third of the APIs require the construction of the HTTP request [3]. Therefore, **R5:** it is necessary to be able to specify the parts of the HTTP requests that are used to transmit the input.

Input Data. The preparation of the input data is one of the most challenging tasks related to invocation. Providers commonly use parameters with optional values, default values, alternative values (for example: 1, 2 or 3) and coded values ('en' instead of 'english'). In addition, more than two thirds of the APIs do not explicitly state the datatype of the input. In order to avoid having to process data on the syntactic level and to be able to benefit from the automation support provided by describing the APIs and their inputs and outputs semantically, we handle the preparation of the input data with the help of **R6:** definition of input data transformations in terms of lowering schemas. The lowering schema defines how to transform the semantic input into the parameters used in the invocation HTTP message. In addition, more than half of the APIs use optional parameters, which means that the invocation can be completed even if no input values are provided. Therefore, **R7:** the capturing of characteristics that have a direct impact on invocation, such as optional values or output-format parameters is required. Furthermore, this observation leads to the need to be able to describe not only the input and output as a whole but also parts of the message individually. It is necessary to support **R8:** the description of the complete messages as well as the explicit annotation of their message parts.

Actual invocation. Once the HTTP method, the invocation URI, the HTTP body and headers, and the input data are prepared, the actual HTTP message can be constructed. The Web API invocation itself, including the sending of the HTTP request, the waiting for the response as well as the receiving of the response is realised as part of the system implementing the invocation engine (see Section IV).

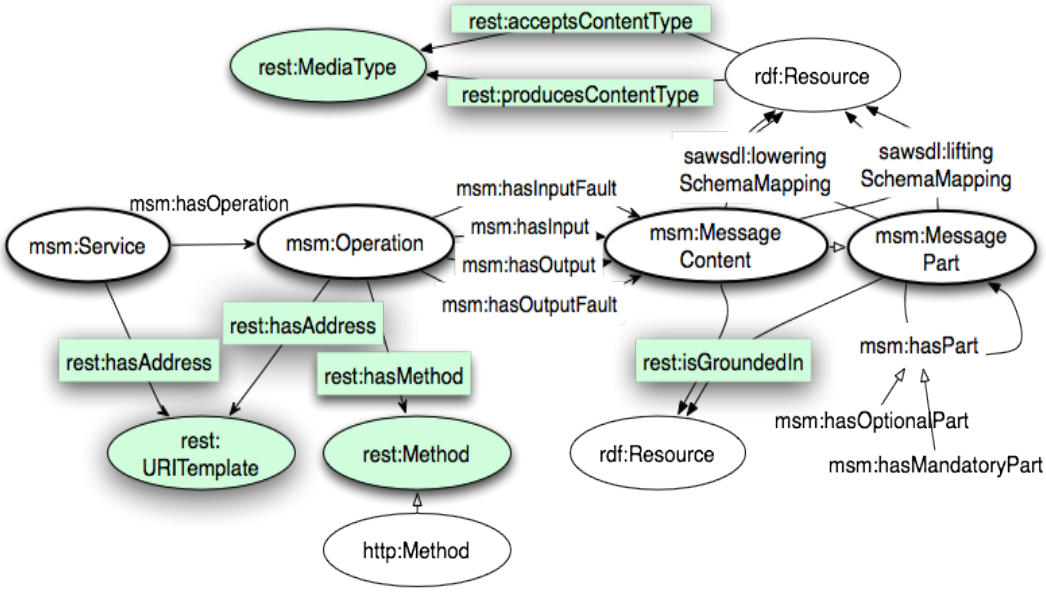

Fig. 1: Web API Grounding Model

Response handling and output data processing. Once the HTTP response is received, the actual output, which is commonly sent as part of the body, needs to be extracted. There are a number of possible formats for the output formats, however, providing support for the use of XML and JSON would cover the majority of the APIs [3]. Since we are basing our approach on the use of lightweight semantics, the **R9:** definition and inclusion of a lifting schema mapping is required. If the API invocation runs smoothly, the output is extracted and lifted to RDF, but if the invocation fails, an appropriate mechanism for error handling needs to be implemented. In the case of custom errors, **R10:** the lifting schema mapping needs to be able to process those and transform them to RDF.

In summary, we have identified the main steps of completing the invocation process and have derived the resulting requirements. However, surprisingly enough as visualised in Table I, currently none of the existing description approaches cover all of the necessary details ([1]Percentage that provides the invocation URI; [2]No support for custom errors.). The two main gaps that we have identified are in providing means to specify the data grounding, i.e. which parts of the input are transmitted via which parts of the HTTP request (in the URI, in the HTTP body or in the HTTP header), and enabling the description of individual input parts. In particular, currently it is not possible to differentiate between input that is simply transmitted as part of the request and input that actually influences the way, in which the invocation is performed, such as output format or authentication credentials parameters.

III. Supporting Automated Invocation

Based on the results of the analysis conducted in the previous section and taking into consideration existing service description approaches, we aggregate the collected data and devise an invocation model, capable of supporting the automated invocation for a large percentage of the APIs. The here presented model is a result of refining and extending the original Minimal Service Model (MSM), which was initially introduced with in [9]. We provide a strict decoupling between the parts that capture the core service properties (*msm:*) and the specific extensions for supporting invocation (*rest:*). Figure 1 visualises the Web API grounding model, consisting of the *msm* namespace elements (http://cms-wg.sti2.org/ns/minimal-service-model#), including *Service*, *Operation*, *MessageContent* and optional or mandatory *MessageParts*, and the *rest* namespace elements (http://purl.org/hRESTS/1.1) that represent the main extensions to the model. The model captures the HTTP method (R1) and is based on defining the service in terms of operations (R2), which have URI Template-based addresses (R3) that can further extend or overwrite the service address definition (R4). The model also provides means for specifying data grounding via the *isGroundedIn* property, in particular defining whether the input values are transmitted as part of the HTTP body or the URI (R5).

Lifting and lowering schema mappings (R6 and R9) can be associated with the inputs and outputs as a whole, i.e. *MessageContent*, but also with individual message parts (R8). In particular, we allow for fine-grain definition of the inputs and outputs (R8) that can have optional or mandatory parts (R7). In its original version, hRESTS expected a single lowering transformation that would apply to the whole input message, without distinguishing between different parameters of the URI. In our extension, we allow finer-grained (and thus more reusable) lowering transformations on individual message parts. Errors are handled via the definition of output faults and as part of the lifting transformation, which can be adjusted to deal with custom errors (R10).

IV. Implementation and Evaluation

The contribution described here is implemented as part of a general invocation engine – OmniVoke, which is exposed as a RESTful API, where the invocation functions are abstracted into resources such as request, response and status, etc., identified by the semantically-described service Unique ID (UID). Therefore, OmniVoke provides a "meta" API that repents a common invocation point for the majority of the APIs. In particular, the service and operation names are exposed in the following form http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/service/{ServiceUID}/operation/{OperationName}/invoke, where the request data is sent to the invocation engine in the HTTP body via POST, since the calling of the invocation API represents the creation of a new request resource.

Listing 1 shows the semantic description of the Last.fm [2] *artist.getInfo* operation, which is created based on its HTML documentation. We provide a tool called SWEET [10] that supports the creation of semantic Web API descriptions. The *LastFMService* contains an *ArtistGetInfo* operation with input *ArtistGetInfoInput*. The input contains message parts *artist*, *api_key*, and their links to external ontology entities, i.e. to Music Ontology, as well as the links to loweringSchemaMapping and liftingSchemaMapping scripts, which are in the form of SPARQL queries. The input data grounding is realised trough the definition of the operation address as a parameterized URI Template, where each of the message parts has a *isGroundedIn* property specifying its place in the URI.

Listing 1: Example RDF Service Description

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/
   services/e8f9548e-bbed-43fe-9d8a-71b7fdef9da#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix msm: <http://cms-wg.sti2.org/ns/minimal-service-model#>.
@prefix rest: <http://purl.org/hRESTS/1.1#>.
:LastFMService a msm:Service;
    msm:hasOperation :ArtistGetInfo;
    rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate.
:ArtistGetInfo a msm:Operation;
    msm:hasInput :ArtistGetInfoInput;
    msm:hasOutput :ArtistGetInfoOutput;
    rest:hasMethod "GET";
    rest:hasAddress "method=artist.getinfo&artist={p1}
&api_key={p2}"^^rest:URITemplate.
:ArtistGetInfoInput a msm:MessageContent;
    msm:hasPart :artist, :api_key.
:artist a msm:MessagePart;
    sawsdl:loweringSchemaMapping http://iserve.kmi.open.ac.uk/
 lilo /ArtistLowering. txt ;
    sawsdl:modelReference "http://purl.org/ontology/mo/MusicArtist";
    rest:isGroundedIn "p1"^^rdf: PlainLiteral .
:api_key a msm:MessagePart;
    sawsdl:loweringSchemaMapping http://iserve.kmi.open.ac.uk/
 lilo /APIKeyLowering.txt;
    sawsdl:modelReference "http://purl.oclc.org/NET/
WebApiAuthentication#API_Key";
    rest:isGroundedIn "p2"^^rdf: PlainLiteral .
:ArtistGetInfoOutput a msm:MessageContent;
    sawsdl:liftingSchemaMapping http://iserve.kmi.open.ac.uk/
 lilo / ArtistGetInfoLifting . txt .
```

Based on this description, the Last.fm API can be invoked by OmniVoke via the POST http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/service/db4b646a-4665-4337-9626-4669cc8bce56/operation/ArtistGetInfo/invoke. We also evaluate our model based on the coverage that it provides, given the heterogeneity of the Web API world. We do this by both annotating actual Web APIs and testing their invocability and by determining the overall percentage of coverage, depending on the different API characteristics. All examples are available at http://purl.org/hRESTS/examples.

[2] http://www.last.fm/api/show?service=267

V. Conclusions and Future Work

Nowadays, finding, interpreting and invoking Web APIs requires extensive human involvement. There are a number of approaches targeted at supporting the invocation of a particular type of APIs, specifically RESTful, and some solutions that rely on the 'pipe'-based approach that enables the composition and invocation of a predefined set of services. Still the invocation of the majority of the Web APIs currently requires the manual implementation of custom solutions that are rarely reusable. Therefore, we propose an approach for capturing invocation-related information by using the Web API Grounding Model, which overcomes Web API heterogeneity and provides the basis for automated invocation handling. We base the annotation approach on an analysis of currently existing Web service and API description forms and on the requirements resulting from the steps involved in sending, processing and receiving HTTP messages. We show the practical applicability of our model by describing how it can be used as input to OmniVoke, implementing automated Web API invocation. Future work will mainly focus on integrating the invocation engine as part of a process engine in order to support the orchestration of semantically annotated Web APIs.

Acknowledgments The work presented in this paper is supported by EU funding under SOA4All (FP7 - 215219).

References

[1] R. T. Fielding: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, 2000.

[2] Web Services Description Language (WSDL) Version 2.0. Recommendation, W3C, June 2007. Available at http://www.w3.org/TR/wsdl20/.

[3] M. Maleshkova, C. Pedrinaci, J. Domingue: Investigating Web APIs on the World Wide Web. In Proc of the 8th European Conf. on Web Services (ECOWS), 2010.

[4] M. J. Hadley: Web Application Description Language (WADL). Technical report, Sun Microsystems, November 2006. Available at https://wadl.dev.java.net.

[5] J. Kopecký, T. Vitvar, D. Fensel, K. Gomadam: hRESTS & MicroWSMO. Technical report, available at http://cms-wg.sti2.org/TR/d12/, 2009.

[6] A. P. Sheth, K. Gomadam, J. Lathem: SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. In IEEE Internet Computing, 11(6):9194, 2007.

[7] J. Kopecký, T. Vitvar, C. Bournez, J. Farrel: SAWSDL: Semantic Annotations for WSDL and XML Schema. IEEE Internet Computing, 11(6):60-67, 2007.

[8] N. Li, C. Pedrinaci, M. Maleshkova, J. Kopecky, J. Domingue: OmniVoke:A Framework for Automating the Invocation of Web APIs. Fifth IEEE International Conference on Semantic Computing, 2011.

[9] T. Vitvar, J. Kopecký, J. Viskova, D. Fensel: WSMO-Lite Annotations for Web Services. In the Semantic Web: Research and Applications, ESWC 2008.

[10] M. Maleshkova, C. Pedrinaci, J. Domingue: Semantic annotation of Web APIs with SWEET. 6th Workshop on Scripting and Development for the Semantic Web at ESWC, 2010.