Functional Composition of Sensor Web APIs

Ruben Verborgh[1], Vincent Haerinck[2], Thomas Steiner[3], Davy Van Deursen[1], Sofie Van Hoecke[2], Jos De Roo[4], Rik Van de Walle[1], and Joaquim Gabarro[3]

[1] Ghent University – IBBT, ELIS – Multimedia Lab
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg-Ghent, Belgium
{ruben.verborgh,rik.vandewalle}@ugent.be
[2] ELIT Lab, University College West Flanders
Ghent University Association, Graaf Karel de Goedelaan 5, 8500 Kortrijk, Belgium
{vincent.haerinck,sofie.van.hoecke}@howest.be
[3] Universitat Politécnica de Catalunya – Department LSI, 08034 Barcelona, Spain
{tsteiner,gabarro}@lsi.upc.edu
[4] Agfa Healthcare, Moutstraat 100, 9000 Ghent, Belgium
jos.deroo@agfa.com

Abstract. Web APIs are becoming an increasingly popular alternative to the more heavy-weight Web services. Recently, they also have been used in the context of sensor networks. However, making different Web APIs (and thus sensors) cooperate often requires a significant amount of manual configuration. Ideally, we want Web APIs to behave like Linked Data, where data from different sources can be combined in a straightforward way. Therefore, in this paper, we show how Web APIs, semantically described by the light-weight format RESTdesc, can be composed automatically based on their functionality. Moreover, the composition process does *not* require specific tools, as compositions are created by generic Semantic Web reasoners as part of a proof. We then indicate how the composition in this proof can be executed. We describe our architecture and implementation, and validate that proof-based composition is a feasible strategy on a Web scale. Our measurements indicate that current reasoners can integrate compositions of more than 200 Web APIs in under one second. This makes proof-based composition a practical choice for today's Web APIs.

Keywords: Semantic Web, Web APIs, sensors, composition, reasoning

1 Introduction

Sensors are gradually finding their way to the world of Web APIs. The REST principles of resource-oriented API design, as defined by Fielding [15], are gaining momentum on the Web of Things [17,40]. On top of this, Semantic Web technologies are then used to make the sensor data meaningful to machines [34,35]. A uniform way to access semantic sensor data is not the endpoint: machines need a way to select *what* sensor they need for a specific situation. This is the task of semantic Web API descriptions [33,39,42], which capture the functionality of

Web APIs in a semantic format. However, much more innovate power becomes available when different sensors are *combined* to deliver new and unprecedented functionality. Unfortunately, today, this involves a substantial amount of manual work: while Web APIs bare the potential to be composed straightforwardly, they lack the semantics to do this in an automated way [26].

The present paper addresses this issue by introducing a method to automatically compose Web APIs. On the one hand, this allows a faster and easier development of Web applications. On the other hand, it enables on-demand solutions for specific problems and questions, for which it would be impractical or infeasible to create ad-hoc solutions manually. Furthermore, the proposed method does *not* require specific tools or software, but rather works with generic Semantic Web reasoners. This ensures the maintainability and generalizability of the solution towards the future.

In the end, we want to enable for Web APIs what Linked Data [7] does for data: the automated integration of various, heterogeneous sources with the help of semantics, leading to composability. Web APIs are an excellent match for this, because of the many parallels between the Linked Data and REST principles [15,20,44]. Also, Web APIs allow us to move beyond the traditional input/output-based matching from the Web services world, instead delivering integration based on functionality.

This paper is structured as follows. In Section 2, we describe related work on Web API composition. Section 3 introduces a use case and explains the concepts of reasoning-based composition, followed by the principles of composition execution. Section 4 proposes an architecture and implementation to perform composition and execution in an automated way. This approach is evaluated in Section 5, and Section 6 concludes the paper by placing Web API composition in the broader Web context and provides an overview of future work.

2 Related Work

In the next subsections, we discuss related work in the fields of Semantic Web Service description, Web API description, and Semantic Web reasoners.

2.1 Semantic Web Service Description and Composition

Semantic Web service description has been a topic of intense research for at least a decade. There are many approaches to service description with different underlying service models. OWL-S [31] and WSMO [25] are the most known Semantic Web Service description paradigms. They both allow to describe the high-level semantics of services whose message format is WSDL [12]. Though extension to other message formats is possible, this is rarely seen in practice. Semantic Annotations for WSDL (SAWSDL [24]) aim to provide a more light-weight approach for bringing semantics to WSDL services. Composition of Semantic Web services has been well documented, but all approaches require specific software [18,19,32] and none of the solutions have found widespread adoption.

2.2 Web API Description

In recent years, more and more Web API description formats have been evolving. The link between the Semantic Web and Web APIs has been explored many times [37]. Linked Open Services (LOS, [33]) expose functionality on the Web using Linked Data technologies, namely HTTP [14], RDF [21], and SPARQL [38]. Input and output parameters are described with SPARQL graph patterns embedded inside RDF string literals to achieve quantification, which RDF does not support natively. Linked Data Services (LIDS, [39]) define interface conventions that are compatible with the Linked Data principles [7] and are supported by a lightweight formal model. RESTdesc [42] is a hypermedia API description format that describes Web APIs' functionality in terms of resources and links.

The Resource Linking Language (ReLL, [1]) features media types, resource types, and link types as first class citizens for descriptions. The RESTler crawler [1] finds RESTful services based on these descriptions. The authors of ReLL also propose a method for ReLL API composition [2] using Petri nets to describe the machine-client navigation. However, automatic, functionality-based composition is not supported.

Several methods aim to enhance existing technologies to deliver annotations of Web APIs. HTML for RESTful Services (hRESTS, [22]) is a microformat to annotate HTML descriptions of Web APIs in a machine-processable way. SA-REST [16] provides an extension of hRESTS that describes other facets such as data formats and programming language bindings. MicroWSMO [23,29], an extension to SAWSDL that enables the annotation of RESTful services, supports the discovery, composition, and invocation of Web APIs. The Semantic Web sErvices Editing Tool (SWEET, [27]) is an editor that supports the creation of mashups through semantic annotations with MicroWSMO and other technologies. A shared API description model, providing common grounds for enhancing APIs with semantic annotations to overcome the current heterogeneity, has been proposed in the context of the SOA4All project [28].

2.3 Semantic Web Reasoning

Pellet [36] and the various Jena [11] reasoners are likely the most-known examples of publicly available Semantic Web reasoners. Pellet is an OWL DL [8] reasoner, while the Jena framework offers transitive, RDFS [10], OWL [8], and rule reasoners. The rule reasoner is the most powerful, but uses a rule language that is specific to Jena and therefore not interchangeable.

Another category of reasoners use the Notation3 language (N3, [4]), a small superset of RDF that adds support for formulas and quantification, providing a logical framework for inferencing [5]. The initial N3 reasoner is the forward-chaining cwm [3], which is a general-purpose data processing tool for RDF, including tasks such as querying and proof-checking. Another important N3 reasoner is EYE [13], whose features include backward-chaining and high performance. A useful capability of both N3 reasoners is their ability to generate and exchange *proofs*, which can be used for software synthesis or API composition [30,45].

3 Concept

3.1 Example Use Case

To illustrate the theoretical framework, we first introduce an example that will be carried through the paper. The problem statement is as follows:

> *A user wants to reserve a nearby restaurant. He will take a table outside if the weather allows it.*

To solve this problem, the following Web APIs (and several others) are available:

Location API gets the current location;
Temperature API reads a temperature sensor near a specific location;
Pressure API reads an air pressure sensor near a specific location;
Restaurant API makes a restaurant reservation.

If we were to solve this problem manually with the given APIs, a straightforward solution would be to combine them as in Fig. 1. This graph shows how, starting from the **I**nitial state *(the user and his preferences)*, we can reach the **G**oal state *(inside or outside reservation in a nearby restaurant, depending on the weather)*. First, the **L**ocation API needs to look up the current location of the user. Then, the **T**emperature and **P**ressure APIs can be invoked with this location. Based on their results, the details of the reservation can be completed. Finally, the **R**estaurant API uses these parameters to make the reservation, thereby satisfying the **G**oal. The order in which the execution happens is governed by the *dependencies* between the APIs, as depicted in Fig. 1.

This composition can either serve as a one-time solution for a specific situation, or be reused in different scenarios, in which case it becomes a Web API itself. In any case, the goal is to create and execute this composition in a fully automated way. This process will be explained in the next subsections.

3.2 Universally Representing Compositions

In order to automatically create compositions, we need a universal way to represent these compositions and the APIs of which they consist, so machines can manipulate them easily. In essence, a composition can be seen as a logic entailment, since the **I**nitial state must entail the **G**oal state:

$$\mathbf{I}(\mathit{composition}) \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \mathbf{G}(\mathit{composition})$$

This perfectly aligns with the notion of dependencies, since the satisfaction of **G** depends on the satisfaction of **I**. Analogously, each API can be seen as an implication. For instance, given a location of a place, the **T**emperature API allows to obtain its temperature. In that sense, the **T**emperature API fulfills the implication between a **location** and its **temperature**:

$$\mathbf{T} \quad \equiv \quad \mathbf{location}(\mathit{place}) \Rightarrow \mathbf{temperature}(\mathit{place})$$

That way, Web APIs can be represented as implications, and compositions as a chain of implications that leads to entailment.

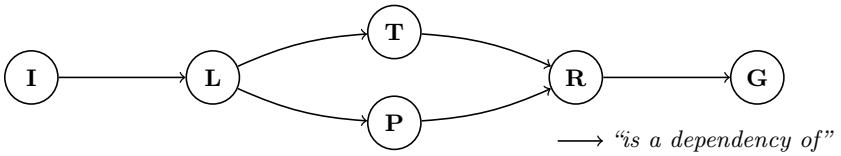

Fig. 1. By combining the **L**ocation, **T**emperature, **P**ressure, and **R**estaurant APIs, and respecting their dependencies, we can reach the **G**oal from the **I**nitial state.

3.3 Deriving Compositions

Not only are implications a straightforward representation to manipulate, the question whether we can solve a certain problem becomes a matter of entailment: *does the **I**nitial state entail the **G**oal state?* However, more important than whether the problem can be solved, is *how* it can be solved, in other words, which APIs are necessary to find the answer. In the logic world, this comes down to providing the *proof* of the entailment: ***why** does the **I**nitial state entail the **G**oal state?* For the restaurant example, a proof might look like this[1]:

$$\mathbf{I} \Rightarrow \mathbf{preferences}(\mathit{user}) \tag{1}$$

$$\mathbf{L} \equiv \mathbf{preferences}(\mathit{user}) \Rightarrow \mathbf{location}(\mathit{place}) \tag{2}$$

$$\mathbf{T} \equiv \mathbf{location}(\mathit{place}) \Rightarrow \mathbf{temperature}(\mathit{place}) \tag{3}$$

$$\mathbf{P} \equiv \mathbf{location}(\mathit{place}) \Rightarrow \mathbf{pressure}(\mathit{place}) \tag{4}$$

$$\mathbf{R} \equiv \mathbf{demand}(\mathit{appointment}) \Rightarrow \mathbf{reservation}(\mathit{appointment}) \tag{5}$$

$$\mathbf{reservation}(\mathit{appointment}) \Rightarrow \mathbf{G} \tag{6}$$

$$\text{consequent}(1) \Rightarrow \text{antecedent}(2) \tag{7}$$

$$\text{consequent}(2) \Rightarrow \text{antecedent}(3) \tag{8}$$

$$\text{consequent}(2) \Rightarrow \text{antecedent}(4) \tag{9}$$

$$\text{consequent}(3), \text{consequent}(4) \Rightarrow \text{antecedent}(5) \tag{10}$$

$$\text{consequent}(5) \Rightarrow \text{antecedent}(6) \tag{11}$$

$$\mathbf{I} \Rightarrow (1), (7), (8), (9), (10), (11), (6) \Rightarrow \mathbf{G} \tag{12}$$

In this proof, we immediately recognize the structure of the composition as depicted in Fig. 1. The characterisations of the **I**nitial and **G**oal states can be seen in (1) and (6) respectively, and the definitions of the APIs in (2) to (5). The dependency relations are contained in (7) to (11). For instance, the fact that the **L**ocation API is a dependency of the **T**emperature API in Fig. 1 corresponds to the implication in (8). Finally, (12) contains the combined proof elements that explain *why* **I** entails **G**, effectively generating the whole composition. This indicates how the proof of entailment explains how APIs can be combined to deliver the requested functionality. As a result, the proof is in fact an alternate and automatically reconstructed representation of the composition graph.

[1] Note that certain background knowledge is assumed (ontologies and/or rules).

3.4 Executing compositions

Once we have obtained a proof, we can execute all Web API calls within it. That way, it becomes a *pragmatic proof*, in which all of the inferences will actually be carried out. The execution order is governed by the dependencies between the APIs. Because the proof starts with the **I**nitial state and ends with the **G**oal state—and cycles are impossible within proofs—we are sure at each step of the execution to find at least one API whose dependencies have been resolved. This is obvious in the example proof, since every proof step only refers to steps with a lower number.

As a result, for the first API call, all parameters are known in advance. In the example, the sole option is to start with the **L**ocation API, since this is the only API with no other dependencies than the **I**nitial state. Its parameter, an address, is already known and will be present in the proof. The situation is different for the **T**emperature and **P**ressure APIs: they both depend on the **L**ocation API and have the resulting geographical coordinates as a parameter. This value is unknown at the time the proof is constructed. However, the proof does tell us how this value can be obtained: it is the result of the **L**ocation API invocation.

There are two ways to deal with these unknown values: A first approach is to do bookkeeping during the execution. Since the proof tells us the dependencies between the APIs, we can assign the values received from previous API calls to the associated variables. A second approach is to repeat the reasoning process after each execution of an API call. The **I**nitial state is thereby augmented with the information returned by the API call, giving rise to a new composition with fewer steps. The benefit is that this approach also works if the API provides a result that is different than expected.

Now that we understand how to manually compose and execute Web APIs, we will have a look at the automation of the process.

4 Architecture and Implementation

4.1 Overview

In this section, we describe how the concepts put forward in Section 3 have been automated and realized in a software platform. Fig. 2 shows an overview of the platform's architecture. It consists of three main components:

- the **reasoner**, which generates the composition;
- the **executor**, which governs the execution of compositions;
- the **client**, offering the interface to coordinate the above two components.

The platform expects the following inputs:

- various **APIs** and corresponding **descriptions**;
- a request, consisting of the **I**nitial and **G**oal states.

The APIs and descriptions are likely to be part of a reusable collection, for instance, an API repository. In contrast, the **I**nitial and **G**oal states will probably differ between invocations.

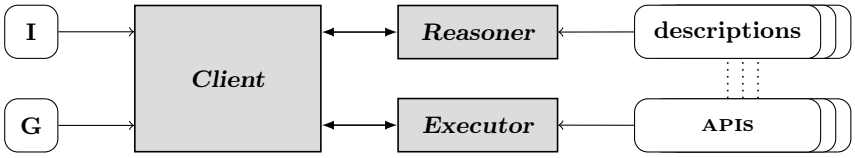

Fig. 2. The principal platform architecture, showing the client that interacts with the reasoner and executor.

Upon receiving a request, the client instructs the reasoner to verify whether the **I**nitial state entails the **G**oal state, reckoning with the provided API descriptions. At the same time, if this entailment holds, the client will ask for the proof. This proof will contain all details needed by the executor to actually invoke the APIs and obtain the desired result. If we apply this to the restaurant example, the available APIs are **L**ocation, **T**emperature, **P**ressure, **R**estaurant, and possibly many others, all of them with their corresponding description. To start the process, the user gives the address of his preferred restaurant to the client, along with the instruction to reserve this restaurant on the next sunny day for his nearby friends. In Subsections 4.3 and 4.4, we will zoom in on the implementation of the reasoner and the executor, but first, Subsection 4.2 will introduce and justify the description technology used in this implementation.

4.2 Description Technology

The first decision to make is what technology will describe the Web APIs, since the platform can only be as powerful as the expressivity of the description method permits. Candidate technologies should possess these characteristics:

- support **REST** or **hypermedia** APIs [15] (as opposed to RPC-style services);
- explain the **functionality** of the API in a machine-processable way (as opposed to detailing only input and output parameters);
- allow **composition** of any number of APIs.

For the implementation, we selected RESTdesc [42,43], since it explicitly targets hypermedia APIs and focuses on functionality. Furthermore, RESTdesc descriptions are expressed in Notation3 (N3, [4]), a Semantic Web logic language put forward by Tim Berners-Lee, which allows *generic* Semantic Web reasoners that take N3 as input to interpret RESTdesc descriptions directly. As a result, RESTdesc-described APIs can easily be composed with the proof-based technique.

Listing 1 displays the RESTdesc description of the **R**estaurant API. RESTdesc descriptions are N3 rules whose antecedent contains the preconditions and whose consequent contains the request and postconditions . The hypermedia nature of RESTdesc can be clearly seen: starting from a restaurant resource that has a `reservationList` link to a reservations resource ❶, a client can `POST` ❸ reservation details ❷ to attach a reservation to the restaurant resource ❹. RESTdesc descriptions indeed focus on resources and the links between them, which makes them an excellent fit to describe hypermedia APIs.

```
@prefix resto: <http://example.org/restaurant#>.
@prefix http:  <http://www.w3.org/2011/http#>.
{
  ?restaurant resto:reservationList ?reservations.  1
  ?place resto:isOutside ?outside.
  ?day resto:hasDate ?date.
}
=>
{
  _:request http:methodName "POST";  3
            http:requestURI ?reservations;
            http:body (?date ?outside);
            http:resp [ http:body ?reservation ].

  ?restaurant resto:hasReservation ?reservation. 4
  ?reservation resto:onDate ?date;
               resto:place [ resto:isOutside ?outside ] .
}.
```

Listing 1. This example RESTdesc description explains the part of the **R**estaurant API that allows to make a reservation.

4.3 Reasoner

Since RESTdesc can be interpreted by generic N3 reasoners, we do not need to implement a specific reasoner for RESTdesc composition. This offers a considerable benefit in terms of portability and sustainability. Performance-wise, this choice is also beneficial, because several implementations of N3 reasoners exist [5], giving rise to an ongoing competition of reasoner developers who continue to improve reasoner performance. Reusing the implementation efforts and experience of the wider reasoning community is a faster and more durable decision than developing and maintaining a specific composition algorithm from the ground up.

We have tested our implementation with the EYE [13] and cwm [3] reasoners, both of which have the ability to generate a proof, such as the one we have crafted manually in Subsection 3.3. This proof must be understood by the client, because it represents the composition the executor will run.

Since the full proof of the example restaurant composition would be too lengthy to discuss, we will use another example from the sensor domain.[2] In the example, the background knowledge is that all temperature sensors are sensors, and that *MySensor* is a temperature sensor. In N3, this is expressed as:

```
<MySensor> a s:TemperatureSensor.
{ ?something a s:TemperatureSensor. } => { ?something a s:Sensor. }.
```

The reasoner also needs a goal query. In this case, we will ask to find all sensors:

```
{ ?x a s:Sensor. } => { ?x a s:Sensor. }.
```

Note that this is *not* an inference, but a query similar to SPARQL `CONSTRUCT`. The answer to this query is, after inference:

```
<MySensor> a s:Sensor.
```

To generate the proof, we invoke the reasoners with a command similar to:

```
eye sensors.n3 --query query.n3
cwm sensors.n3 --think --filter=query.n3 --why
```

```
@prefix s:   <sensors#>.
@prefix var: <var#>.
@prefix r:   <http://www.w3.org/2000/10/swap/reason#>.
@prefix n3:  <http://www.w3.org/2004/06/rei#>.

[ a r:Proof, r:Conjunction;                                    (P)
  r:component
    [ a r:Inference;                                           (Q)
      r:gives {<MySensor> a s:Sensor.};
      r:evidence (
        [ a r:Inference;                                       (R)
          r:gives {<MySensor> a s:Sensor};
          r:evidence ([ a r:Extraction;
                        r:gives {<MySensor> a s:TemperatureSensor.};
                        r:because [ a r:Parsing;
                                    r:source <sensors.n3>]]);
          r:binding [ r:variable [ n3:uri "var#x0"];
                      r:boundTo  [ n3:uri "MySensor"]];
          r:rule [ a r:Extraction;
                   r:gives {@forAll var:x0.
          {var:x0 a s:TemperatureSensor.} => {var:x0 a s:Sensor.}.};
                   r:because [ a r:Parsing; r:source <sensors.n3>]]);
      r:binding [ r:variable [ n3:uri "var#x0"];
                  r:boundTo  [ n3:uri "MySensor"]];
      r:rule [ a r:Extraction;
               r:gives {@forAll var:x0. {var:x0 a s:Sensor.}
                                         => {var:x0 a s:Sensor.}.};
               r:because [ a r:Parsing; r:source <query.n3>]];
  r:gives {
    <MySensor> a s:Sensor.
  }].
```

Listing 2. This example proof illustrates the important proof concepts.

Listing 2 shows the resulting proof. As we can see, a `Proof` (P) consists of a `Conjunction` of components that `gives` the answer to our query. In this example, the only component is an `Inference` (Q) that applies the query rule to the statement "`<MySensor> a s:TemperatureSensor.`", which is done by `binding` the `?x` variable to `<MySensor>`. Then of course, the question is where this statement came from. The `evidence` relation explains it as another `Inference` (R), for which the "`{ ?s a s:TemperatureSensor. } => { ?s a s:Sensor. }`" rule was applied to the statement "`<MySensor> a s:TemperatureSensor.`", `binding` the `?s` variable to `<MySensor>`, and thereby leading to the desired result. The proof then indicates that all further evidence are `Extraction`s as from `Parsing`.

This explanation reveals the dependency-oriented nature of proofs: the validity of the proof depends on the validity of an inference, which in turn depends on another inference, which ultimately depends on parsing an original source. This perfectly aligns with the dependencies of compositions. Since every Web API in the composition will be described by RESTdesc, which captures functionality in inference rules, the inferences in the proof will correspond to API invocations.

[2] The example is available at `http://notes.restdesc.org/2012/sensors/`.

Furthermore, the variable bindings detail the parameters that need to be used for each invocation. Eventually, the parsed source will correspond to the **I**nitial state, and the result of the proof will be the **G**oal state. To create compositions, it is therefore sufficient to start a reasoner with a command similar to:

```
eye initial.ttl descriptions.n3 --query goal.n3
cwm initial.ttl descriptions.n3 --think --filter=goal.n3 --why
```

Note that Web API calls do not have to be the only inferences in the proof: traditional implications (such as ontological constructs) can be carried out, too. This opens up the possibility to combine results from different API calls, and to compose APIs that have been expressed in different ontologies.

4.4 Executor

In order to execute a composition, the executor does not only need to know what APIs to execute, but also all details of what each HTTP request to these APIs should like. Fortunately, by using RESTdesc descriptions as part of the proof process, the variables in the descriptions will be instantiated with concrete values. For example, as part of the proof, the description from Listing 1 will be instantiated as in Listing 3.

As explained in Subsection 3.4, not all parameter values are known in advance. In Listing 3, we can see that the concrete URI of the restaurant and the reservation list have been instantiated: the executor thus already knows that it will have to perform a `POST` request to the URI `http://resto.example.org/reservations/`. It also know the date, which is part of the **I**nitial state. However, it does not know yet the concrete value of `?outside`, so it represents it as a blank node instead. Because this blank node will be linked to blank nodes in the instantiation of the **T**emperature and **P**ressure APIs, the executor understands that it has to use the output of these APIs as the input of the **R**estaurant API.

Since at each step at least one API request will be fully instantiated, the executor will always be able to proceed. Partially instantiated requests can be completed with the result of executed API requests, either by the executor or by performing subsequent reasoner runs with the new data.

```
<http://resto.example.org/> resto:reservationList
                                <http://resto.example.org/reservations/>.
_:place1 resto:isOutside _:outside1.
_:day1 resto:hasDate "2012/11/11".

_:request1 http:methodName "POST";
           http:requestURI <http://resto.example.org/reservations/>;
           http:body ("2012/11/11" _:outside1);
           http:resp [ http:body _:reservation1 ].

<http://resto.example.org/> resto:hasReservation _:reservation1.
_:reservation1 resto:onDate _:date1;
               resto:place [ resto:isOutside _:outside1 ].
```

Listing 3. The instantiation of the **R**estaurant API description by the reasoner.

Another important aspect of the executor is that it is not limited to a certain content representation format. While the RESTdesc descriptions are expressed in N3 or in RDF (when instantiated), the Web APIs do not have to produce or consume RDF. The executor acts as a hypermedia client that negotiates content types at runtime. This is why RESTdesc purposely does not describe the format of the exchanged messages. Such flexibility enables the APIs to communicate in any format the executor supports. For instance, the **T**emperature sensor could interact using JSON, while the **L**ocation sensor could provide answers in GML [35].

5 Evaluation

The crucial statement in this paper is that generic reasoners are able to create Web API compositions in an automated way. Our evaluation verifies whether this concept works on a Web scale, *i.e.*, with a large number of APIs and associated descriptions, assuming that any given task will require a combination of a relatively small subset of all available APIs. To this extent, we have developed a benchmark framework[3] that consists of two main components:

- a **description generator**, which is able to generate an arbitrary-length chain of RESTdesc descriptions that can form a composition;
- an **automated benchmarker**, which tests a reasoner for compositions of varying lengths and complexity.

These variations in complexity are obtained by modifying the number of dependencies between different descriptions. In the simplest case, every API exactly depends on one previous API in the chain. More complex cases involve multiple dependencies. We have tested three scenarios for n going from 2 to 1024:

1. a chain of n APIs with 1 dependency between each of them;
2. a chain of n APIs with 2 dependencies between each of them;
3. a chain of n APIs with 3 dependencies between each of them.

Additionally, we looked at the composition of a 1-dependency chain of 32 APIs, in presence of a growing number of "dummy" APIs that are meant to test how fast the reasoner can discriminate between relevant and non-relevant descriptions.

It is important to understand that most real-world scenarios will be a mixture of the above situations: compositions generally consist of API calls with a varying number of dependencies, created in presence of a non-negligible number of descriptions that are irrelevant to the composition under construction. Therefore, by measuring these aspects independently, we can predict how well a reasoner will perform in those situations.

The measurements have been split in *parsing*, *reasoning*, and *total* times. Parsing represents the time during which the reasoner internalizes the input into an in-memory representation. This was measured by presenting the inputs

[3] The RESTdesc composition benchmark suite is freely available for download at `http://github.com/RubenVerborgh/RESTdesc-Composition-Benchmark`.

#descriptions (n)	2	4	8	16	32	64	128	256	512	1024
n APIs (1 dep.)										
parsing	53	53	54	55	58	64	78	104	161	266
reasoning	2	4	5	7	10	20	43	77	157	391
total	55	57	58	62	68	84	121	181	318	657
n APIs (2 deps.)										
parsing	53	53	59	56	60	67	85	117	184	331
reasoning	3	6	69	41	45	56	84	174	461	1,466
total	56	59	128	97	104	123	169	292	645	1,797
n APIs (3 deps.)										
parsing	53	53	68	56	61	70	90	129	208	371
reasoning	3	12	45	49	61	99	200	544	1,639	6,493
total	57	66	114	105	122	169	290	673	1,847	6,864
32 APIs, n dummies										
parsing	59	60	62	64	65	72	88	134	170	278
reasoning	10	10	10	10	11	12	12	12	12	14
total	69	70	72	74	76	84	100	146	182	292

Table 1. The EYE reasoner manages to create even lengthy compositions in a timely manner (*average times of 50 trials, expressed in milliseconds*).

to the reasoner, without asking for any operation to be performed on them. Since the parsing step can often be cached and reused in subsequent iterations, it is worthwhile evaluating the actual reasoning time separately. Parsing and reasoning together make up for the total time.

Table 1 shows the benchmark results of the EYE reasoner, as generated on a consumer computer (2.66 GHz Intel Core i7, 4GB RAM). The results in the first column teach us that a start-up overhead of $\approx 50ms$ is involved for starting the reasoner. This includes process starting costs and is highly machine-dependent. When looking at the parsing times for all 4 cases, we see that they increase linearly with the number and size of the descriptions, as expected from any parser. In each of the benchmarks for n APIs with 1 to 3 dependencies, we see that the composition time increases linearly with the number of descriptions. As a consequence, the total time also increases linearly.

Finally, if we look at the composition of 32 APIs in presence of dummy APIs, we can see that the influence of the dummies on the reasoning time is minimal. Compared to the case where 32 APIs with one dependency were composed without dummies, we see that most overhead is introduces by the parsing of the extra descriptions, which can be cached. The reasoning time remains fairly close to the original time, even in presence of a large number of dummies.

6 Conclusion and Future Work

In this paper, we presented a novel approach to compose Web APIs, integrating sensors APIs with others. We showed how the description format RESTdesc enables functionality-based compositions, which are automatically created using the proof-generating capabilities of a generic Semantic Web reasoner. We described the architecture and implementation of a platform that is able to compose and execute these Web API compositions. It features a client that accepts API descriptions, an **I**nitial state and a **G**oal, which a reasoner then uses to create a composition that is carried out by an executor.

Based on the results of our evaluation, we can say that proof-based composition of Web APIs by generic reasoners is a feasible approach, even on a Web scale where thousands of sensors could be involved. Reasoning time evolves linearly with the number and size of the descriptions, with response times far below one second for typical composition sizes, even in presence of a large amount of descriptions. Moreover, the reasoner-based approach is much more sustainable than composition methods that are tied to a specific description method.

If we situate RESTdesc composition in the Semantic Web Stack [9] in Fig. 3, we see it is based on the fundamental elements. As a light-weight Web API description method, RESTdesc strives to express the functionality of APIs with the goal of integration and composability, maximizing technology reuse. In that way, we hope to put one of the steps required to bring Web APIs towards the high level of integration and composability of today's Linking Open Data cloud [6].

In the future, we want to improve composition further by taking optimization strategies into account. If several compositions can provide similar solutions, we need a mechanism to select the optimum, given a specified set of constraints. Our work on defining quality parameters of APIs and compositions [41] shows part of our progress in this field. An important challenge is dealing with the heterogeneity of Web APIs and the many different representations they offer. While RESTdesc is not tied to a specific representation format, the executor must be able to deal with a variety of formats across the Web. We strongly believe in content type negotiation, since the Web has been and always will be a diverse environment. Finally, we aim to improve the client so that it becomes usable by a wide audience. We consider browser-based implementations for computers and mobile devices, to bring the power of Web API composition to everyone.

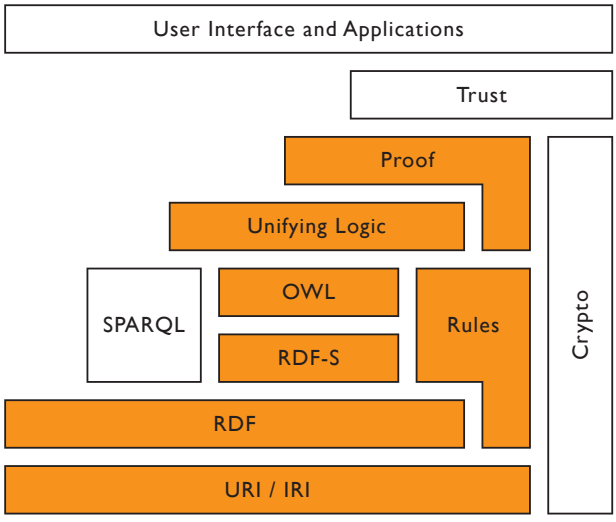

Fig. 3. Proof-based Web API composition, based on RESTdesc functional descriptions, maximizes technology reuse in the Semantic Web Stack.

Acknowledgments The authors would like to thank Maria Maleshkova for proofreading this paper thoroughly and providing us with invaluable suggestions.

The described research activities were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union. This work was partially supported by the European Commission under Grant No. 248296 FP7 (I-SEARCH project). Joaquim Gabarró is partially supported by TIN–2007–66523 (FORMALISM), and SGR 2009–2015 (ALCOM).

References

1. Alarcón, R., Wilde, E.: RESTler: crawling RESTful services. In: Proceedings of the 19th international conference on World Wide Web. pp. 1051–1052. ACM (2010), http://doi.acm.org/10.1145/1772690.1772799
2. Alarcón, R., Wilde, E., Bellido, J.: Hypermedia-driven RESTful service composition. In: Service-Oriented Computing, Lecture Notes in Computer Science, vol. 6568, pp. 111–120. Springer (2011), http://dx.doi.org/10.1007/978-3-642-19394-1_12
3. Berners-Lee, T.: cwm. Semantic Web Application Platform (2000–2009), available at http://www.w3.org/2000/10/swap/doc/cwm.html
4. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax. W3C Team Submission (Mar 2011), http://www.w3.org/TeamSubmission/n3/
5. 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), http://arxiv.org/abs/0711.1533
6. Bizer, C., Jentzsch, A., Cyganiak, R.: State of the LOD cloud (2011), http://www4.wiwiss.fu-berlin.de/lodcloud/state
7. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – The Story So Far. International Journal On Semantic Web and Information Systems 5(3), 1–22 (2009), http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf
8. Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: OWL 2 Web Ontology Language. W3C Recommendation (Oct 2009), http://www.w3.org/TR/owl2-syntax/
9. Bratt, S.: Semantic Web, and other technologies to watch. INCOSE International Workshop (Jan 2007), available at http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)
10. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (Feb 2004), http://www.w3.org/TR/rdf-schema/
11. Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the Semantic Web recommendations. In: Proceedings of the 13th international World Wide Web conference. pp. 74–83. ACM (2004), www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf
12. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL). W3C Note (Mar 2001), http://www.w3.org/TR/wsdl
13. De Roo, J.: Euler proof mechanism (1999–2012), available at http://eulersharp.sourceforge.net/
14. Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 (Jun 1999), http://www.ietf.org/rfc/rfc2616.txt

15. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. Transactions on Internet Technology 2(2), 115–150 (May 2002), http://dl.acm.org/citation.cfm?id=514185
16. Gomadam, K., Ranabahu, A., Sheth, A.: SA-REST: Semantic Annotation of Web Resources. W3C Member Submission, http://www.w3.org/Submission/SA-REST/
17. Guinard, D., Trifa, V., Wilde, E.: A resource-oriented architecture for the Web of Things. In: Internet of Things (Dec 2010), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5678452
18. Hashemian, S., Mavaddat, F.: A graph-based approach to Web services composition. In: Proceedings of the 2005 Symposium on Applications and the Internet. pp. 183–189 (2005), http://www.cin.ufpe.br/~redis/intranet/bibliography/middleware/hashemian-composition05.pdf
19. Hristoskova, A., Volckaert, B., De Turck, F.: Dynamic composition of semantically annotated Web services through QoS-aware HTN planning algorithms. In: Fourth International Conference on Internet and Web Applications and Services. pp. 377–382. IEEE (2009), http://dl.acm.org/citation.cfm?id=1586263
20. Kjernsmo, K.: The necessity of hypermedia RDF and an approach to achieve it. In: Proceedings of the Linked APIs workshop at the 9th Extended Semantic Web Conference (May 2012), http://lapis2012.linkedservices.org/papers/1.pdf
21. 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/
22. Kopecký, J., Gomadam, K., Vitvar, T.: hRESTS: An HTML microformat for describing RESTful Web services. In: Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology. pp. 619–625. IEEE Computer Society (2008), http://dx.doi.org/10.1109/WIIAT.2008.379
23. Kopecký, J., Vitvar, T.: MicroWSMO. WSMO Working Draft (Feb 2008), http://www.wsmo.org/TR/d38/v0.1/
24. Kopecký, J., Vitvar, T., Bournez, C., Farrell, J.: Semantic Annotations for WSDL and XML Schema. IEEE Internet Computing 11, 60–67 (2007), http://cms-wg.sti2.org/doc/IEEEIC2007-KopeckyVBF.pdf
25. Lausen, H., Polleres, A., Roman, D.: Web Service Modeling Ontology (WSMO). W3C Member Submission (Jun 2005), http://www.w3.org/Submission/WSMO/
26. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating Web APIs on the World Wide Web. In: Proceedings of the 8th European Conference on Web Services. pp. 107–114. IEEE (2010), http://sweet-dev.open.ac.uk/war/Papers/mmaWebAPISurvey.pdf
27. Maleshkova, M., Pedrinaci, C., Domingue, J.: Semantic annotation of Web APIs with SWEET (May 2010), http://oro.open.ac.uk/23095/
28. Maleshkova, M., Pedrinaci, C., Li, N., Kopecky, J., Domingue, J.: Lightweight semantics for automating the invocation of Web APIs. In: Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (Dec 2011), http://sweet.kmi.open.ac.uk/pub/SOCA.pdf
29. Maleshkova, M., Kopecký, J., Pedrinaci, C.: Adapting SAWSDL for semantic annotations of RESTful services. In: Proceedings of the On the Move to Meaningful Internet Systems Workshops, Lecture Notes in Computer Science, vol. 5872, pp. 917–926. Springer (2009), http://dx.doi.org/10.1007/978-3-642-05290-3_110
30. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. Transactions on Programming Languages and Systems (TOPLAS) 2(1), 90–121 (1980), http://dl.acm.org/citation.cfm?id=357084.357090

31. 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/
32. Milanovic, N., Malek, M.: Current solutions for Web service composition. Internet Computing, IEEE 8(6), 51–59 (2004), http://ieeexplore.ieee.org/iel5/4236/29773/01355922.pdf
33. Norton, B., Krummenacher, R.: Consuming dynamic Linked Data. In: Proceedings of the 1st International Workshop on Consuming Linked Data (Nov 2010), http://ceur-ws.org/Vol-665/NortonEtAl_COLD2010.pdf
34. Page, K., Frazer, A., Nagel, B., Roure, D.D., Martinez, K.: Semantic access to sensor observations through Web APIs. In: 5th IEEE International Conference on Semantic Computing. IEEE (September 2011), http://eprints.soton.ac.uk/272695/
35. Page, K., De Roure, D., Martinez, K., Sadler, J., Kit, O.: Linked sensor data: RESTfully serving RDF and GML. In: Semantic Sensor Networks (Oct 2009), http://eprints.soton.ac.uk/271743/
36. Parsia, B., Sirin, E.: Pellet: An OWL DL reasoner. In: Proceedings of the Third International Semantic Web Conference (2004), http://iswc2004.semanticweb.org/posters/PID-ZWSCSLQK-1090286232.pdf
37. Pedrinaci, C., Domingue, J., Krummenacher, R.: Services and the Web of Data: An unexploited symbiosis. In: Proceedings of the AAAI Spring Symposium on Linked Data Meets Artificial Intelligence (2010), http://people.kmi.open.ac.uk/carlos/wp-content/uploads/downloads/2010/09/linkedServices-AAAI.pdf
38. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (Jan 2008), http://www.w3.org/TR/rdf-sparql-query/
39. Speiser, S., Harth, A.: Integrating Linked Data and Services with Linked Data Services. In: The Semantic Web: Research and Applications, Lecture Notes in Computer Science, vol. 6643, pp. 170–184. Springer (2011), http://people.aifb.kit.edu/aha/2012/sms/lids-eswc2011.pdf
40. Stirbu, V.: Towards a RESTful plug and play experience in the Web of Things. In: IEEE international Conference on Semantic Computing. pp. 512–517. IEEE (2008), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4597240
41. Verborgh, R., Steiner, T., Gabarró Vallés, J., Mannens, E., Van de Walle, R.: A social description revolution—describing Web APIs' social parameters with RESTdesc. In: Proceedings of the AAAI 2012 Spring Symposia (Mar 2012), http://www.aaai.org/ocs/index.php/SSS/SSS12/paper/download/4283/4665
42. Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró Vallés, J., Van de Walle, R.: Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In: Proceedings of the Third International Workshop on RESTful Design. ACM (Apr 2012), http://www.ws-rest.org/2012/proc/a5-9-verborgh.pdf
43. Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., Gabarró Vallés, J.: Capturing the functionality of Web services with functional descriptions. Multimedia Tools and Applications (2012), http://www.springerlink.com/index/d041t268487gx850.pdf
44. Verborgh, R., Steiner, T., Van de Walle, R., Gabarró Vallés, J.: The missing links–How the description format RESTdesc applies the Linked Data vision to connect hypermedia APIs. In: Proc. of the Linked APIs workshop at the 9th Extended Semantic Web Conference (May 2012), http://lapis2012.linkedservices.org/papers/3.pdf
45. Waldinger, R.: Web agents cooperating deductively. In: Formal Approaches to Agent-Based Systems, Lecture Notes in Computer Science, vol. 1871, pp. 250–262. Springer (2001), http://dx.doi.org/10.1007/3-540-45484-5_20