TLP **17** (1): 1–48, 2017. © Cambridge University Press 2016
doi:10.1017/S1471068416000016 First published online 7 March 2016

The pragmatic proof: Hypermedia API composition and execution

RUBEN VERBORGH, DÖRTHE ARNDT and SOFIE VAN HOECKE
Ghent University – iMinds – Data Science Lab, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
(*e-mail:* {ruben.verborgh,dorthe.arndt,sofie.vanhoecke}@ugent.be)

JOS DE ROO and GIOVANNI MELS
Agfa Healthcare, Moutstraat 100, 9000 Ghent, Belgium
(*e-mail:* {jos.deroo,giovanni.mels}@agfa.com)

THOMAS STEINER and JOAQUIM GABARRO
Department LSI, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain
(*e-mail:* tomac@google.com, gabarro@lsi.upc.edu)

submitted 15 August 2013; revised 18 December 2013, 13 November 2014, 31 August 2015; accepted 9 December 2015

Abstract

Machine clients are increasingly making use of the Web to perform tasks. While Web services traditionally mimic remote procedure calling interfaces, a new generation of so-called hypermedia APIs works through hyperlinks and forms, in a way similar to how people browse the Web. This means that existing composition techniques, which determine a procedural plan upfront, are not sufficient to consume hypermedia APIs, which need to be navigated at runtime. Clients instead need a more dynamic plan that allows them to follow hyperlinks and use forms with a preset goal. Therefore, in this paper, we show how compositions of hypermedia APIs can be created by generic Semantic Web reasoners. This is achieved through the generation of a proof based on semantic descriptions of the APIs' functionality. To pragmatically verify the applicability of compositions, we introduce the notion of pre-execution and post-execution proofs. The runtime interaction between a client and a server is guided by proofs but driven by hypermedia, allowing the client to react to the application's actual state indicated by the server's response. We describe how to generate compositions from descriptions, discuss a computer-assisted process to generate descriptions, and verify reasoner performance on various composition tasks using a benchmark suite. The experimental results lead to the conclusion that proof-based consumption of hypermedia APIs is a feasible strategy at Web scale.

KEYWORDS: composition, proof, reasoning, Semantic Web, hypermedia APIs, Web APIs

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

1 Introduction

1.1 Hard-coded API contracts on a hypermedia-driven Web

The World Wide Web, on which millions of servers together offer billions of information resources, has been designed as a distributed hypermedia application (Berners-Lee *et al.* 1992). "Hypermedia" means that pieces of information can be connected to each other; therefore, consuming information does not require any knowledge of servers' internal information structures. Instead, users of the Web follow *hyperlinks* and fill out *forms* to move from one piece of information to the next. This architectural decision has been essential for the global growth of the Web: people can browse websites by clicking links, regardless of whether they have used them before. Fielding called this principle *"hypermedia as the engine of application state"* (Fielding and Taylor 2002), because servers send hypermedia documents, which clients use to advance the state of the interaction. Rather than relying on a pre-determined set of actions that would have been communicated through a separate channel, such hypermedia documents let clients select steps just-in-time. This ensures the Web's temporal scalability: if a server decides to change its interface, clients do not have to be reprogrammed—they simply receive hypermedia documents with different links.

As more and more people found their way to the Web, it seemed evident that automated clients would also start using the Web autonomously. The immediate barrier was that all resources on the Web were only available in human languages, which machines cannot accurately interpret yet. A logical step for programmers, who deal with Application Programming Interfaces (APIs) in software development, was to retrofit a system for API operations to the Web's HTTP interface (Fielding *et al.* 1999). Instead of navigating links without prior knowledge, machines executed a pre-defined list of commands that they translated into HTTP requests. As such, those APIs follow a proprietary Remote Procedure Calling (RPC) protocol *on top of* HTTP, which—despite the label "Web services" or "Web APIs"—has consequently few in common with the Web.

As expected, such proprietary APIs with a fixed contract cannot withstand evolution very well. If the server changes the API, clients have to be reprogrammed. While incompatible API changes are uncommon in closed environments, servers on the Web are in constant evolution, as witnessed by the short lifespan of websites and APIs. The hypermedia mechanism that guides clients through the Web—and thus allows them to cope with changes—is notably absent from RPC APIs.

1.2 Hypermedia APIs as native Web citizens

A decade after the invention of the Web, Fielding analyzed the architectural principles that contributed to its world-wide growth, which he captured in the Representational State Transfer (REST) architectural style (Fielding and Taylor 2002). Unlike the RPC APIs discussed above, APIs that follow the REST principles are native Web citizens, and thus more resilient to change (Verborgh *et al.* 2015).

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

The distinguishing characteristic of REST is its *uniform interface*, consisting of four constraints (Fielding and Taylor 2002):

Identification of resources: The essential unit of information in REST architectures is a *resource*, a conceptual entity that must be *uniquely identifiable*. On the Web, this means that each resource must have its own URL. For example, `http://example.org/weather/london/2014/05/01` could identify the weather in London on a particular day.

Resource manipulation through representations: Resources are conceptual and cannot be transfered; clients and servers instead exchange *representations*. Depending on the client's capabilities, client and server agree on one of multiple possible representations of each resource (e.g., HTML for humans, JSON or RDF for machines). For example, the weather in London with the above URL would be represented as a JSON document when requested by a JavaScript application.

Self-descriptive messages: Rather than defining custom actions, as is the case with APIs in typical programming languages, REST APIs use a limited set of commands defined by a protocol. For example, the Web uses HTTP's `GET` and `PUT`, which have a universal meaning, rather than `showWeather` or `setWeather`.

Hypermedia as the engine of application state: Also known as the *hypermedia constraint*, this principle indicates that the client should be able to perform the interaction with the server solely through hypermedia. On the Web, this happens through *hypermedia controls* such as hyperlinks and forms. Another perspective on this is that all communication should happen *in-band* instead of *out-of-band*: clients engage in an interaction through hypermedia representations of resources rather than through pre-defined contracts. REST APIs are thus *hypermedia-driven* (Fielding 2008). For example, tomorrow's weather is linked from today's weather, rather than having to craft a new weather request by hand.

Unfortunately, many APIs mistakenly label themselves as "REST", giving a rather unclear meaning to the term "REST API"; in particular, the fourth constraint is often missing. As a result, the term "*hypermedia API*" is used to distinguish those APIs that follow all REST constraints (Richardson *et al.* 2013), and thus inherit their architectural benefits.

An important consequence of the REST architectural constraints is that there is no observable difference between *websites* and *hypermedia APIs*. The term "website" is commonly applied when the consumers are humans, and "hypermedia API" is used for machine clients. However, they only differ in the kind of representations they offer: human-readable or machine-readable. By offering multiple representations, a *single* hypermedia API/website can serve both humans and machine clients, as opposed to the two separated interfaces we commonly see (Verborgh *et al.* 2015).

1.3 Generic clients of hypermedia APIs

Once APIs have been made accessible for machines, the question becomes: how can we create clients that perform tasks using such APIs? And more specifically, can we build generic clients that do not have to be pre-programmed for a specific task?

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

For example, given an API for shipping packages, an API for user profiles, and an API of online bookstore, how can we deliver a book on a user's wish list to their doorstep? Indeed, in addition to interpreting an API's responses, clients should be able to reason about an APIs' functionality and how it can be combined with other APIs to perform a complex task. This question has been the subject of much literature for classical Web services, where the APIs were composed into a static plan that had to be executed step-by-step. Such a plan, and the mechanism to generate it, would only be valid as long as all involved APIs did not undergo any changes, which is a rather unrealistic assumption on the Web.

The situation is fundamentally different for hypermedia APIs: because of the hypermedia constraint, no such static plan can (nor should) be created beforehand, because hypermedia APIs can only be consumed by following hypermedia controls. On the other hand, clients should somehow know what links they must follow, because only certain links will lead them to successful completion of the desired goal. This gives rise to an approach where we need a high-level plan that guides the runtime, hypermedia-driven execution of Web APIs.

In this paper, we introduce and formalize a proof-based method to produce such high-level plans, and detail their execution through hypermedia. In the next section, we describe related work in the domain of Web services and Web APIs, followed by a justification of the chose technologies in Section 3. We introduce to hypermedia API description in Section 4. Section 5 explains the use of proofs to validate hypermedia API compositions, the generation of which is detailed in Section 6. We explain how descriptions can be created in a computer-assisted process in Section 7. The feasibility of the approach is evaluated in Section 8. Finally, we end the paper in Section 9 with conclusions and an outlook on future work.

2 Related work

2.1 Web APIs

Let us start a discussion on related work by defining terms about Web APIs.

A Web server uses the HTTP protocol to offer data and/or actions, which are often exclusively located on this server. In other words, the server performs a data lookup or computation that another device cannot or does not.

A Web client consumes resources offered by a Web server.

A Web service or Web API is a machine-accessible interface to data and/or actions offered by a server through HTTP. Traditionally, "Web service" is used for RPC-based XML interfaces, whereas "Web API" is used for more lightweight APIs, such as those based on JSON.

A Web API operation is a single HTTP request from a Web client to a Web API. Interactions between a client and a server consist of one or more operations.

A hypermedia control is a hyperlink or form that allows clients to navigate from one resource to another.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

A hypermedia API is a Web API that follows all REST architectural constraints as discussed in Section 1.2. In particular, it is designed such that its (human or machine) clients should perform the interaction through hypermedia controls.

A characteristic of hypermedia APIs is that they, in contrast to RPC APIs, are not action-based but resource-based. As such, the border between data and actions is blurred; they are effectively modeled as one and the same. For example, an RPC API might offer access to image resizing functionality by adding a dedicated action named `resizeImage`. A hypermedia API would instead expose a resource for the original image such as `/images/381/`, which would allow access to resized images through a links towards the resource `/images/381/thumbnail/`. This structuring around resources considerably simplifies the planning process, as actions do not have to be considered as separate entities. Consequently, as we will see in Section 4, rules and data need not be treated differently.

Integrating hypermedia APIs into an application now-a-days requires manual development work, such as writing the HTTP request templates and parsing the returned HTTP responses. Instructions on how to write this code can often be found on the API's website in the form of human-readable API documentation. In order to automate this process, *machine-readable documentation* is necessary. On the lowest level, this documentation describes the message format and modalities. On a higher level, it explains the specific functionality offered by the hypermedia API, so a machine can autonomously decide whether the API is appropriate for a certain use case.

2.2 The Semantic Web

To create machine-readable information in general, we can use standards from the Semantic Web (Berners-Lee *et al.* 2001), which is a vision that enables autonomous clients to use the Web. A central standard is RDF (Klyne and Carrol 2004), which is a machine-interpretable language consisting of data triples (*subject*, *predicate*, *object*). For instance, the following triple describes the relationship between Leonhard Euler and Daniel Bernoulli:

```
<http://dbpedia.org/resource/Leonhard_Euler>
    <http://xmlns.com/foaf/0.1/knows>
        <http://dbpedia.org/resource/Daniel_Bernoulli>.
```

Various RDF syntaxes exist; some of them, such as Notation3 (N3, Berners-Lee and Connolly 2011) extend the RDF model with features such as variables and quantification. The proof-based algorithm in this paper uses the N3 reasoner EYE (Verborgh and De Roo 2015) to generate plans. We chose N3 over other formalisms such as Prolog (Clocksin and Mellish 1994) or Datalog (Abiteboul *et al.* 1995) because RDF—and thus N3—is native to the Web. This is exemplified in the triple above: the predicate that relates Euler to Bernoulli can be seen as a (typed) hyperlink from the first URL to the other. Thereby, the hyperlink concept that is crucial for the Web and hypermedia APIs, can be represented in the most straightforward way in RDF/N3. Furthermore, the presence of RDF

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

means that we can reuse ontological constructs from well-known vocabularies such as RDFS (Brickley and Guha 2004) or OWL (Bock *et al.* 2012). For example, the domain and range of the `foaf:knows` predicate are `foaf:Person`. Hence, using the triple above and this ontological knowledge, additional triples can be derived, expressing that Euler and Bernoulli are instances of `foaf:Person`. Such derived knowledge could then be used for hypermedia API operations that require `foaf:Person` instances as input.

2.3 Web service description

Machine-readable documentation of Web services 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 (Martin *et al.* 2004) and WSMO (Lausen *et al.* 2005) are the most well-known Semantic Web service description paradigms. They both allow to describe the high-level semantics of services whose message format is WSDL (Christensen *et al.* 2001). Though extension to other message formats is possible, this is rarely seen in practice. Semantic Annotations for WSDL (SAWSDL, Kopecký *et al.* 2007) aim to provide a more lightweight approach for bringing semantics to WSDL services. Composition of Semantic Web services has been well documented, but all approaches focus on RPC interactions and require specific software (Milanovic and Malek 2004). In contrast, the proposed approach works for REST interactions and exclusively relies on *generic* semantic Web reasoners. While automated approaches to create descriptions are being researched (Ordóñez *et al.* 2012), they are not the focus of this paper.

2.4 Web API description

In recent years, several description formats for the more lightweight Web APIs have emerged (Verborgh *et al.* 2014). Linked Open Services (Norton and Krummenacher 2010) expose functionality on the Web using Linked Data technologies, such as HTTP and RDF. Input and output parameters are described with graph patterns embedded inside RDF string literals to achieve quantification, which RDF does not support natively. Linked Data Services (Speiser and Harth 2011) define interface conventions supported by a lightweight model. None of these methods, however, use the hypermedia principles of REST. Several methods aim to enhance existing technologies to deliver annotations of Web APIs. HTML for RESTful Services (Kopecký *et al.* 2008) is a microformats extension to annotate HTML descriptions of Web APIs in a machine-processable way. SA-REST (Gomadam *et al.* 2010) provides an extension of HTML for RESTful Services that describes other facets such as data formats and programming language bindings. MicroWSMO (Kopecký and Vitvar 2008; Maleshkova *et al.* 2009), an extension to SAWSDL that enables the annotation of RESTful services, supports the discovery, composition, and invocation of Web APIs, but requires additional software. Data-Fu (Stadtmüller *et al.* 2013) uses rules to describe client–server interactions; however, these rules are tightly bound to a server's information structure and are equivalent to a fixed declarative program

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

in RPC style. This paper instead introduces a flexible, hypermedia-driven technique for the REST architectural style.

2.5 Hypermedia API description

The description of hypermedia APIs is a relatively new field. Hydra (Lanthaler and Gütl 2013) is a vocabulary to support API descriptions, but does not directly support automated composition. RESTdesc (Verborgh *et al.* 2012) is a description format for hypermedia APIs that describes them in terms of resources and links. RESTdesc is expressed in N3 and will be used as a description format in this paper, so it is discussed further in Section 4. The Resource Linking Language (Alarcón and Wilde 2010) features media types, resource types, and link types as first-class citizens for descriptions. The authors of Resource Linking Language also propose a method for Resource Linking Language API composition (Alarcón *et al.* 2011) using Petri nets to describe the machine-client navigation. However, in contrast to RESTdesc, it does not support automatic, functionality-based composition.

2.6 Semantic Web reasoning

The Pellet reasoner (Parsia and Sirin 2004) and the various reasoners of the Jena framework (Carroll *et al.* 2004) are the most commonly known examples of publicly available semantic Web reasoners. Pellet is a reasoner on ontological constructs (Bock *et al.* 2012), while Jena offers various ontological and rule-based reasoners. The rule reasoner is the most flexible, as it allows to incorporate custom derivations, but it uses a rule language that is specific to Jena and therefore not interchangeable.

Another category of reasoners uses N3, leveraging the language's support of formulas and quantification for RDF to provide a logical framework for inferencing (Berners-Lee *et al.* 2008). The first N3 reasoner was the forward-chaining *cwm* (Berners-Lee 2009), which is a general-purpose data processing tool for RDF, including tasks such as querying and proof-checking. Another important N3 reasoner is EYE (Verborgh and De Roo 2015), 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 (Manna and Waldinger 1980; Waldinger 2001).

3 Justification of chosen technologies

In this section, we explain and justify the technological choices made in this paper. First, we detail our choice for REST Web APIs, emphasizing the fundamental differences with RPC APIs. Second, we argue why we choose N3 instead of other logic programming languages.

3.1 REST Web APIs

With the exception of RESTdesc, existing techniques for Web service and Web API description (discussed in Sections 2.3 and 2.4) focus exclusively on interactions that

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

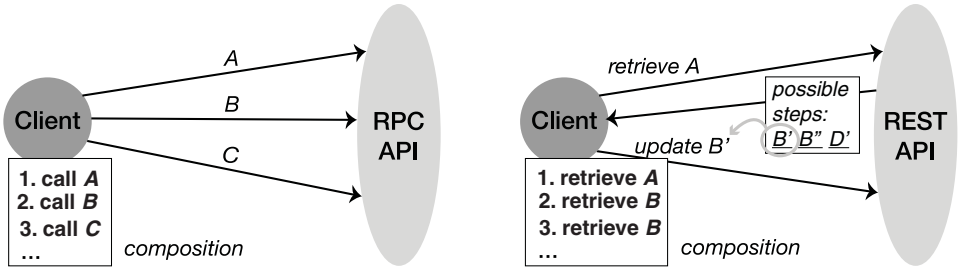


Fig. 1. Interaction differences between clients of RPC and REST APIs (a) RPC composition and interaction, (b) REST composition and interaction.

follow the RPC model. Algorithms that create compositions of such services or APIs will essentially produce a list of calls that have to be issued. This process is visualized in Figure 1(a). Two things are of particular interest in this diagram:

- While subsequent calls might use output from earlier tasks (and might even be conditional based on such output), the type of calls made is not influenced by the RPC API. The *control flow* is dictated by the composition, based on descriptions of each call. The API itself does not provide any information about which next steps can be taken and how these should be performed. The role of descriptions is thus three-fold: (a) explaining which order of calls are possible; (b) describing how to perform calls; (c) expressing the functionality of calls.
- Consequently, all control information is in the descriptions and composition; the server does not send any control information. The client thus fully relies on the descriptions and composition for control information.

We conclude that, apart from their label, Web services or APIs that follow the RPC communication style have only a minor connection to the Web. While technically, they tunnel their calls over the HTTP protocol, nothing in their principled workings is tied to the Web's core principles, which include hypermedia documents and hyperlinks.

In contrast, an API that follows the REST architectural style provides (information) *resources* rather than (procedural) *calls* as its interface. It replies to a request for a resource with a *hypermedia representation*, which contains hyperlinks and forms that provide access to possible next steps. Figure 1(b) illustrates this principle: a client first requests a resource A, which results in a hypermedia document. This documents contains a link to B', which is a concrete instantiation of the B the client should look for according to the composition. Using the control information in the hypermedia document, the client then performs an action on B'. This indicates two major differences compared to RPC:

- The REST control flow is not fully dictated by the composition. The composition only provides a high-level plan of what needs to happen, whereas concrete actions are performed through hypermedia controls. Descriptions have only one function, namely expressing the functionality provided by

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

a certain resource type. The access order or an explanation of how the call is executed are determined at runtime based on the REST API's responses.

- As such, the client needs to derive a control flow at runtime by combining high-level steps from a composition with concrete actions from hypermedia documents obtained from the server.

Note how REST APIs do have a strong connection to the Web's core principles, as clients use hypermedia documents and hyperlinks to perform tasks. Unlike the RPC style, this is similar to human consumption of the Web: when we want to perform a certain task, we also have a high-level plan in mind (e.g., ordering a package involves entering a delivery address) that becomes concrete through hypermedia controls (e.g., using a form to submit that delivery address). This allows for a much looser coupling (Pautasso and Wilde 2009), as descriptions and compositions do not need to contain interaction details.

Furthermore, because of the emphasis on links, REST APIs are closely related to RDF. If an API outputs in an RDF-based format (such as Turtle, JSON-LD, or N3), the client obtains a list of triples, which are essentially typed links (if we interpret the property as a relation specifier). Following these links to realize a concrete action is an example of hypermedia-driven Web API consumption, which is what we will outline in this paper. We stress that this is not possible with existing descriptions and algorithms for RPC Web services and Web APIs, as they cannot incorporate runtime control information originating from the API.

3.2 Notation3

In this paper, we use the N3 language and associated logic framework to generate and execute compositions of REST Web APIs. We justify this choice on both a practical level and a theoretical level, as detailed in the subsections below.

3.2.1 Practical arguments

The Linked Data Cloud contains billions of triples in the RDF model (Bizer *et al.* 2011). In order for this knowledge to be reused, we need a logic with support for triples. Whereas typical programming languages can all support triples in some way, treating them as first-class citizens decreases the effort of working with them. For instance, while the popular Jena library provides triple support for Java, an RDF document is not a Java document, so the RDF will need to be converted in Java-specific objects. Programmatic access and manipulation happens through these objects instead of directly at the RDF level. In contrast, N3 is a superset of the Turtle serialization of RDF. This means that each valid RDF document, when expressed in Turtle, is by definition a valid N3 document. As such, reasoners that support N3 natively support triples, without the impedance mismatch of other languages. This means that all RDF responses on the Web, including those of RDF Web APIs, can be interpreted directly by N3 reasoners. Usage of N3 thus brings direct compatibility with a body of billions of knowledge facts on the Web.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Additionally, hundreds of ontologies are expressed as RDF triples using RDFS and OWL. That means they use RDF triples with RDFS and OWL predicates and objects to define ontological meaning. Because they are expressed as triples, N3 reasoners can process them. However, since many common RDFS and OWL constructs have been captured in publicly available N3 rules (De Roo 2014), N3 reasoners can also apply their semantics without requiring native support for neither RDFS nor OWL. For example, the OWL class `owl:SymmetricProperty` is implemented by stating that for properties p that are symmetric, the existence of a triple (s, p, o) implies the existence of (o, p, s). Therefore, by using N3, we can directly incorporate available ontological knowledge.

Finally, the most important reason to choose N3 as an underlying description format is that it can combine the above aspects together with descriptions of Web APIs. As argued above, the hypermedia-driven nature of REST Web APIs maps well to the RDF model. If we use an N3 rule to describe Web APIs, we can directly reason on the combination of Web API descriptions, existing triples, and existing ontologies. Concretely, if a Web API returns an RDF response, we can directly combine that response with the composition, thereby instantiating a result placeholder with the actual results. This is the core mechanism of the execution process detailed in Section 5.2. Furthermore, the closeness of RDF responses to N3 rules plays an important role in description generation, as discussed in Section 7.

3.2.2 Arguments related to logic programming

In order to be suitable for the approach, we describe in this paper, a logical framework has to fulfill certain requirements:

Semantic Web compatibility: As described above, it is crucial for a format describing RESTful Web APIs to act directly on the semantic Web. Logical knowledge in the semantic Web is usually expressed in RDF triples. The chosen logic should natively support this format.

Rules: Our framework describes the possible use of hypermedia APIs using rules. We therefore need a logic which supports rules.

Proofs: Our approach makes use of proofs produced by a reasoner. These proofs are then used for further reasoning. The logic of choice should be supported by a reasoner which is able to provide such proofs. Furthermore, there needs to be a format to express proofs using the logic.

Conjunctions in the head of a rule: While conjunctions in the head of a rule are normally considered syntactic sugar (Lloyd and Topor 1984), they become important if rules appear in a proof for the following reason: in our approach, we use the instantiated conjunction in the head of a rule as it occurs in a proof to infer concrete practical instructions. To understand this idea, consider a simple example: imagine that we have a source s which is aware of the birth date of any person. In a first order style, we could describe that using the following rule:

$$\forall x : \text{person}(x) \implies (\exists y : \text{birthDateOf}(x, y) \wedge \text{canBeAskedFor}(s, y))$$

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

If we now ask for the birth date of person(leonhard_euler), we get as a result that there exists a birth date:

$$\exists y : \text{birthDateOf}(\text{leonhard_euler}, y).$$

But the present rule derives more, we also get an instruction how to get this particular information:

$$\exists y : \text{birthDateOf}(\text{leonhard_euler}, y) \wedge \text{canBeAskedFor}(s, y)$$

As one rule derives both, the fact that there exists a birth date and an actual instruction how to get it, both things would appear in a proof, while in case of having rules with atomic heads we would have to specifically ask for the instruction. But there could be many ways to obtain that birth date: maybe we could invoke a service calculating it by knowing the exact age of the person and the date of death, maybe it is an even more complex process of first asking for the names of the concrete sources and then asking these sources for information. Such things are not known beforehand. Therefore, it is not a solution to simply include the *canBeAskedFor* predicate in the query. This is the same in our case where these *sources* are Web services. By using only rules with (quantified) atomic heads, we would lose relevant information in the proof. The logic needs thus to support conjunction in the head of a rule.

Existential quantification in the head of a rule: The algorithm presented in this paper employs proofs to make, execute, and adjust a plan of sequent API calls in order to achieve a given goal. The possible calls are described using rules with existentially quantified variables in their head. This can be understood as follows: given a certain situation, there exists a call which can be executed to obtain a new situation. While some details of this "new situation" are already clear before executing the call, some things can only be known afterwards. This kind of uncertainty is crucial for our algorithm. The logical framework should support existentially quantified variables in the head of rules.

Given the first requirement, we take a closer look into semantic Web frameworks: RDF, RDFS, and OWL do not natively support rules, but there are several rule formats defined on top of them. Among them, Semantic Web Rule Language (SWRL) (Horrocks *et al.* 2004), Web Rule Language (WRL) (Angele *et al.* 2005), or Rule Interchange Format (RIF) (Kifer 2008).

The SWRL was built on top of OWL and can be understood as its rule extension. Rules are expressed in terms of OWL concepts (classes, properties, individuals, etc.). SWRL thereby inherits OWL's strong separation between classes and individuals which can form a burden when reasoning over plain RDF data. To the best of our knowledge, there is no reasoner which outputs complete proofs for derivations done applying SWRL rules. SWRL supports conjunctions in the head of a rule but it does not allow new existential variables in that position. Especially, the missing support for existential rules and the fact that there is no exchangeable proof produced by SWRL reasoners made us opt against this rule language.

The WRL and the RIF are both based on Frame Logic (Kifer *et al.* 1995). Frame Logic was invented to combine object oriented and declarative programming. It

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

extends classical predicate calculus with the concepts of objects, classes, and types. Object-oriented ideas such as inheritance are supported as well as classical rule inference. This richness of features made Frame Logic a good choice to base the rule-based ontology language WRL on, and—as it was designed for rule exchange—even a better choice for RIF. Both, WRL and RIF have variants which support conjunction in the head of a rule. Neither the Basic Logic Dialect of RIF nor WRL do allow new existentials in the head of a rule. The direct reasoning support for both formats is very limited, the IRIS system (Bishop and Fischer 2008) provides reasoner for both but does not support proofs. As far as we know there is also no other reasoner which produces proofs based on RIF rules. Nevertheless, RIF can indeed serve as an exchange format for rules.

In contrast to the above mentioned formats, N3 Logic fulfills all of the requirements mentioned above. It is a rule language which easily combines with RDF, RDFS and—via OWL RL—also with OWL. The two most common reasoners for N3, cwm and EYE, both output and check proofs. N3 rules can have conjunctions and existentially quantified variables in their head. Therefore, we chose N3 for our purposes.

Note that by fulfilling the last requirement, support of existential variables in the head of rules, N3 Logic is strongly related to the Datalog$^{\pm}$ framework (Calì *et al.* 2011; Mugnier 2011). Several recent approaches support reasoning with existential rules on top of ontologies as for example, Graal (Baget *et al.* 2015) or IRIS$^{\pm}$ (Gottlob *et al.* 2014). These implementations are very promising, but still under development. Their rule format is not as close to RDF as N3 is and there is no support for proofs yet.

4 Describing hypermedia APIs with RESTdesc

4.1 Example use case

As a guiding example, we will introduce an exemplary hypermedia API in the domain of image processing. It offers functionality such as the following:

- uploading images
- resizing an image
- changing the colors of an image
- combining multiple images
- ...

Since hypermedia APIs follow the REST architectural constraints, this functionality is realized in a resource-oriented, hypermedia-driven way. For instance, the API does *not* offer methods such as `resizeImage`; rather, it exposes `image` and `thumbnail` resources, which are connected to each other through hyperlinks and/or forms. In order to enable machines to interpret responses of this API, these resources are available as representations in the machine-readable format RDF.

The problem statement is to make a generic Web client that can autonomously reach a complex goal such as "obtain a scaled, black-and-white version of `image.jpg`",

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

without hard-coding any knowledge about this API. The client should be able to execute complex instructions on (combinations of) other hypermedia APIs as well.

Since this API is a hypermedia API, we cannot create a detailed plan in advance, because the exact steps are not known at design time. However, at the same time, an automated client cannot only follow hypermedia controls, because there would be no way to know whether the controls it chooses lead to the given goal. In other words, while hypermedia gives machines access to resources via links, it does not explain them the functionality offered through those links.

RESTdesc descriptions (Verborgh *et al.* 2012; Verborgh *et al.* 2013) allow to express the functionality of hypermedia APIs by explaining the role of a hypermedia control in an API. That way, if a machine client encounters a hypermedia control, it can interpret the results of following it. Furthermore, RESTdesc descriptions allow the composition of a high-level plan that can guide machine clients through an interaction with a hypermedia API.

In the remainder of Section 4, we formalize RESTdesc and its underlying N3 logic. Sections 5 and 6 detail hypermedia API composition and execution, using the above image processing hypermedia API as an example.

4.2 Formalization of Notation3

RESTdesc descriptions are expressed in the N3 rule language (Berners-Lee *et al.* 2008; Berners-Lee and Connolly 2011). We will introduce the N3 language and its logic, focusing on the aspects relevant to our purposes. Our formalization is based on the formalization we gave in a previous paper (Arndt *et al.* 2015) and the informal semantic descriptions given in the above mentioned sources. N3 augments the RDF model with symbols for quantification, implication, and statements about formulas:

Definition 1 (*Basic N3 vocabulary*)
An *N3 alphabet A* consists of the following disjoint classes of symbols:

- A set U of URI symbols.
- A set $V = V_E \dot{\cup} V_U$ of (quantified) variables, with V_E being the set of existential variables and V_U the set of universal variables.
- A set L of literals.
- A boolean literal `false`
- Brackets `{`, `}`, `(`, `)`
- Logical implication `=>`
- Period `.`

We define the elements of U as in the corresponding specification (Duerst and Suignard 2005). N3 allows to abbreviate URLs as prefixed names (Beckett *et al.* 2013). Literals are strings beginning and ending with quotation marks '`"`'; existentials start with '`_:`', universals with '`?`'.

N3 does not distinguish between predicates and constants—a single URI symbol can stand for both at the same time—so the first-order-concept of a *term* has

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

a slightly different counterpart in N3: an *expression*. Since the definition of expressions (Definition 2) is closely related to the concept of a formula (Definition 3), the two following definitions should be considered together.

Definition 2 (*Expressions*)
Let A be an N3 alphabet. The set of *expressions* $E \subset A^*$ is defined as follows:

(1) Each URI is an expression.
(2) Each variable is an expression.
(3) Each literal is an expression.
(4) If $e_1, \ldots, e_n$ are expressions, `(`$e_1 \ldots e_n$`)` is an expression.
(5) `false` is an expression.
(6) `{ }` is an expression.
(7) If $f \in F$ is a formula, then `{`f`}` is an expression.

The expression defined by 4 is called a *list*. We call the expressions defined by 5–7 *formula expressions* and denote the set of all formula expressions by FE.

Note that point 7 of the definition above makes use of formulas, which are defined as follows:

Definition 3 (*N3 Formulas*)
The set F of *N3 formulas* over alphabet A is recursively defined as follows:

(1) If $e_1, e_2, e_3 \in E$, then the following is a formula, called *atomic* formula:

$$e_1\ e_2\ e_3.$$

(2) If t_1, t_2 are formula expressions then the following is a formula, called *implication*:

$$t_1 \texttt{=>}\ t_2.$$

(3) If f_1 and f_2 are formulas, then the following is a formula, called *conjunction*:

$$f_1 f_2.$$

We will refer to a formula without any variables as a *ground formula*. Analogously, we call expressions without any variables *ground expressions*. We denote the corresponding sets by F_g respectively E_g. An formula or expression which does not contain universal variables is called *universal free*. The set of universal free formulas (possibly containing existentials) is denoted by F_e, the set of universal free expressions by E_e.

In the examples in the remainder of this paper, we will use the common RDF shortcuts:

Remark 4 (*Syntactic variants*)

- A formula consisting of two triple subformulas starting with the same element `<d> <p> <e>. <d> <q> <f>.` can be abbreviated using a semicolon: `<d> <p> <e>; <q> <f>.`
  Two triple formulas sharing the first two elements `<d> <p> <e>. <d> <p> <f>.` can be abbreviated using a comma: `<d> <p> <e>, <f>.`

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

- `[]` can be used as an expression and is a shortcut for a new existential variable. So `[] <p> <d>.` stands for `_:x <p> <d>.`
- An expression of the form `[<p> <o>]` is a shortcut for a new existential variable `_:x`, which is subject to the statement `_:x <p> <o>.` So `<s> <p> [<q> <o>].` stands for `<s> <p> _:x. _:x <q> <o>.`
- `a` is a shortcut for `rdf:type` (Klyne and Carrol 2004).

To emphasize the difference between brackets which form part of the N3 vocabulary, i.e., "(", ")", "{", and "}", and the brackets occurring in mathematical language, we will underline the N3 brackets in all definitions where both kinds of brackets occur.

Simple N3 triples of the form `:s :p :o` can be understood as a first order formula $p(s, o)$. We call `:p` the predicate, `:s` the subject and `:o` the object of a triple. More complicated constructs often contain variables: in N3 existential and universal variables are implicitly quantified. The scope of this quantification depends on how deeply nested a variable occurs in a formula. To be able to make statements about that we define:

Definition 5 (*Components of a formula*)
Let $f \in F$ be a formula and $c : E \to 2^E$ a function such that:

$$c(e) = \begin{cases} c(e_1) \cup \ldots \cup c(e_n) & \text{if } e = \underline{(}e_1 \ldots e_n\underline{)} \text{ is a list,} \\ \{e\} & \text{otherwise.} \end{cases}$$

We define the set $comp(f) \subset E$ of components of f as follows:

- If f is an atomic formula of the form $e_1\ e_2\ e_3.$, $comp(f) = c(e_1) \cup c(e_2) \cup c(e_3)$.
- If f is an implication of the form t_1`=>` $t_2.$, then $comp(f) = \{t_1, t_2\}$.
- If f is a conjunction of the form $f_1 f_2$, then $comp(f) = comp(f_1) \cup comp(f_2)$.

Likewise, for $n \in \mathbb{N}_{>0}$, we define the components of level n as

$$comp^n(f) := \{e \in E | \exists f_1, \ldots, f_{n-1} \in F : e \in comp(f_1) \wedge \underline{\{}f_1\underline{\}} \in comp(f_2) \wedge \ldots \\ \wedge \underline{\{}f_{n-1}\underline{\}} \in comp(f)\}$$

Now, we can distinguish between direct components and nested components. As an example, take the following N3 formula:

```
:John :says {:Kurt :knows :Albert.}.
```

Direct components are `:John`, `:says` and `{:Kurt :knows :Albert.}` while `:Kurt`, `:knows` and `:Albert` are nested components of level two. N3 allows more complex structures, formulas like

```
{{?x :p :a.} => {?x :q :b.}.} => {{?x :r :c.} => {?x :s :d.}.}.
```

where for example the predicate `:p` occurs as a component of level three are valid in N3. Such deeply nested structures require a careful treatment of scoping for variables occurring in them. Note for example that the above formula should be interpreted as

$$(\forall x_1 : p(x_1, a) \to q(x_1, b)) \to (\forall x_2 : r(x_2, c) \to s(x_2, d))$$

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

and *not* as

$$\forall x : ((p(x,a) \rightarrow q(x,b)) \rightarrow (r(x,c) \rightarrow s(x,d)))$$

Due to this particularities and because deeply nested structures are no requirement for our framework, we limit the considerations of this paper to *simple formulas* and refer the reader interested in more details to the corresponding publication (Arndt *et al.* 2015).

Definition 6 (*Simple formulas*)
We call an N3 formula f *simple* iff for all $n \in \mathbb{N}$, $n > 2$: $\text{comp}^n(f) = \emptyset$

Universal variables in *simple formulas* can be understood as universally quantified on the top level of the formula. The formula

```
{?x :p :o1.} => {?x :q :o2.}.
```

is interpreted as

$$\forall x : p(x, o_1) \rightarrow q(x, o_2)$$

The scope of an existential variable is always the formula expression it occurs in as a direct component. The formula

```
_:x :says {_:x :knows :Albert.}.
```

is interpreted as

$$\exists x_1 : \text{says}(x_1, (\exists x_2 : \text{knows}(x_2, Albert)))$$

As the existential quantification of blank nodes, in contrast to universal quantification, only counts for the direct formula they occur in and not for their subordinated formulas, we define two ways to apply a substitution:

Definition 7 (*Substitution*)
Let A be an N3 alphabet and $f \in F$ an N3 formula over A.

- A *substitution* is a finite set of pairs of expressions $\{v_1/e_1, \ldots, v_n/e_n\}$, where each e_i is an expression and each v_i a variable such that $v_i \neq e_i$ and $v_i \neq v_j$, if $i \neq j$.
- For a formula f and a substitution $\sigma = \{v_1/e_1, \ldots, v_n/e_n\}$, we obtain the *component application* of σ to f, $f\sigma^c$, by simultaneously replacing each v_i which occurs as a *direct component* in f by the corresponding expression e_i.
- For a formula f and a substitution $\sigma = \{v_1/e_1, \ldots, v_n/e_n\}$, we obtain the *total application* of σ to f, $f\sigma^t$, by simultaneously replacing each v_i which occurs as a *direct or nested component* in f by the corresponding expression e_i.

As the definition states, component application of a substitution only changes the direct components of a formula. For a substitution $\mu = \{_:\text{x}/:\text{Kurt}\}$, we obtain

$$(_:\text{x :says } \{_:\text{x :knows :Albert.}\}.)\mu^c = (\text{ :Kurt :says } \{_:\text{x :knows :Albert.}\}.)$$

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

A total application of $\sigma = \{\texttt{_:x}/\texttt{:Kurt}\}$ in contrast, replaces each *occurrence* of a variable in a formula:

$$(\texttt{?x :says } \{\texttt{?x :knows :Albert.}\}.)\sigma^t = (\texttt{:Kurt :says } \{\texttt{:Kurt :knows :Albert.}\}.)$$

For simple formulas, we can now define interpretation and semantics:

Definition 8 (*Interpretation*)
An interpretation $\mathfrak{I}$ of an alphabet A consists of:

(1) A set $\mathscr{D}$ called the domain of $\mathfrak{I}$.
(2) A function $\mathfrak{a} : E \setminus V \to \mathscr{D}$ called the object function.
(3) A function $\mathfrak{p} : \mathscr{D} \to 2^{\mathscr{D}\times\mathscr{D}}$ called the predicate function.

Note that in contrast to the classical definition of RDF-semantics (Hayes and Patel-Schneider 2014), our domain does not distinguish between properties (IP) and resources (IR). The definitions are nevertheless compatible, as we assume $\mathfrak{p}(p) = \emptyset \in 2^{\mathscr{D}\times\mathscr{D}}$ for all resources p which are not properties (i.e., $p \in \text{IR} \setminus \text{IP}$ in the RDF-sense). By extending given RDF ground interpretation functions to N3 interpretation functions, the meaning of all valid RDF triples can be kept in N3 Logic.

Definition 9 (*Semantics of N3*)
Let $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ be an interpretation of A and let f be a *simple* formula over A. Then, the following holds:

(1) If f contains universal variables, $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f\sigma^t$ for every substitution $\sigma : V_U \to E_e$.
(2) If f is universal free and $W = \text{comp}(f) \cap V_E \neq \emptyset$, $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f\mu^c$ for some substitution $\mu : W \to E_g$.
(3) If f is universal free and $\text{comp}(f) \cap V_E = \emptyset$:
  (a) If f is an atomic formula $c_1\, p\, c_2$, then $\mathfrak{I} \models c_1\, p\, c_2$. iff $(\mathfrak{a}(c_1), \mathfrak{a}(c_2)) \in \mathfrak{p}(\mathfrak{a}(p))$.
  (b) If f is a conjunction $f_1 f_2$, then $\mathfrak{I} \models f_1 f_2$ iff $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$.
  (c) If f is an implication
    - $\mathfrak{I} \models \{f_1\}\texttt{=>}\{f_2\}$ iff $\mathfrak{I} \models f_2$ if $\mathfrak{I} \models f_1$.
    - $\mathfrak{I} \models \{f_1\}\texttt{ => false.}$ iff $\mathfrak{I} \not\models f_1$.
    - $\mathfrak{I} \models \texttt{\{ \}=>}\{f_2\}$. iff $\mathfrak{I} \models f_2$.
    - $\mathfrak{I} \models \texttt{false => } e$ and $\mathfrak{I} \models e \texttt{ => \{ \}}.$, for all $e \in E$,

Note that by first handling universal variables (point 1 in the definition) and then treating existentials (point 2), the definition makes sure that in case of conflicts the universal quantifier is outside of the existential. In N3, the statement

```
?x :loves _:y.
```

has to be interpreted as $\forall x \exists y : \text{loves}(x, y)$ and *not* as $\exists y \forall x : \text{loves}(x, y)$.
We now define a model:

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Definition 10 (*Model*)
Let Φ be a set of N3 formulas. We call an interpretation $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ a *model* of Φ iff $\mathfrak{I} \models f$ for every formula $f \in \Phi$.

As in first-order logic, we can define the notion of logical implication:

Definition 11 (*Logical implication*)
Let Φ be a set of N3 formulas and ϕ a formula over the same N3 alphabet A. We say that Φ (logical) implies ϕ ($\Phi \models \phi$) iff every model $\mathfrak{I} \models \Phi$ is also a model of ϕ.

4.3 RESTdesc descriptions

RESTdesc descriptions are designed to explain how a hypermedia API can be used to perform a specific action. Such a process of using an API consists of different steps: given all needed information, the client sends an HTTP request to the hypermedia API on a server. The API interprets the request and, if possible, reacts by fulfilling the indicated task or retrieving the information requested. The server sends a response and thereby creates a new situation. As mentioned in Section 2.1, this process is called an API operation. Hypermedia APIs are commonly able to perform different kinds of operations.

To describe such an operation in N3, a formalization of HTTP is needed. We use the RDF vocabulary as defined in the corresponding W3C Working Draft (Koch *et al.* 2011). In order to facilitate the understanding of the following sections, we give a short overview of the HTTP predicates used in this paper.

Definition 12 (*HTTP predicates*)
Let A be an N3 alphabet, M be a set of HTTP method names, $\{$`"GET"`, `"POST"`, `"PUT"`, `"DELETE"`, `"HEAD"`, `"PATCH"`$\} \subset M \subset L$, $\mathtt{u} \in U$ an HTTP message, $\mathtt{v} \in U \cup L$ and $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ an interpretation of A. Then:

- $\mathfrak{I} \models$ `u http:headers v.` iff `v` is an HTTP header of `u`.
- $\mathfrak{I} \models$ `u http:body v.` iff `v` is the HTTP body of `u`.
- $\mathfrak{I} \models$ `u http:methodName v.` iff `u` is a request and $\mathtt{v} \in M$ its method name.
- $\mathfrak{I} \models$ `u http:requestURI v.` iff `u` is a request and `v` is its URL.
- $\mathfrak{I} \models$ `u http:resp v.` iff `u` is a request and `v` is its responding HTTP message.

This vocabulary can be used to describe HTTP-requests. Such a request must always have a method name and a request URI. Using OWL-vocabulary, such requests can be defined by the following class:

```
http:Request rdfs:subClassOf http:Message,
    [
    rdf:type     owl:Restriction ;
    owl:onProperty http:methodName ;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
    ],
    [
    rdf:type owl:Restriction ;
```

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
owl:onProperty http:requestURI ;
owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] .
```

These two properties also have to be specified in an HTTP request description:

Definition 13 (*HTTP request description*)
Let A be an N3 alphabet which contains a set H of HTTP predicates including those defined in Definition 12.

- An *HTTP request description* is a conjunction $f = f_1 f_2 \ldots f_n \in F$ of atomic formulas with the following properties:
  - — All atomic formulas f_i share the same existential variable $_:\mathtt{x} \in V_E$ as a subject.
  - — The predicate of each formula f_i is an HTTP-predicate, i.e., f_i has the form $_:\mathtt{x}\ :\mathtt{h}_i\ :\mathtt{o}_i\ .$ with $:\mathtt{h}_i \in H$.
  - — The conjunction f contains one atomic formula f_i with the predicate `http:methodName` and one formula f_j with the predicate `http:requestURI`.
  - — The object of every atomic formula $f_i = _:\mathtt{x}\ :\mathtt{h}_i\ :\mathtt{o}_i\ .$ with $\mathtt{h}_i \neq \mathtt{http:resp}$ is either a universal variable, a URI or a literal, $:\mathtt{o}_i \in V_U \cup U \cup L$.
- We call an HTTP request description $f = f_1 \ldots f_n$ *sufficiently specified*, if $:\mathtt{o}_i \in E_g$ for every triple $f_i = _:\mathtt{x}\ :\mathtt{h}_i\ :\mathtt{h}_i$ with $:\mathtt{p}_i \neq \mathtt{http:resp}$.

The definition reflects the syntactical requirements to an HTTP request description, it should contain the URL and the method name of the described request and it can contain additional information which can be described using the HTTP-predicates. If the object of these formulas are instantiated, i.e., sufficiently specified, they can be sent to a server and, if they contain all necessary information, executed by an API which will return the HTTP response. RESTdesc descriptions enable us to specify the intended functionality of a hypermedia API's operations:

Definition 14 (*RESTdesc description*)
Let A be an N3 alphabet containing the predicates defined in Definition 12, F the set of formulas over A. A RESTdesc description $f \in F$ of a hypermedia API operation is a *simple* N3 formula of the form:

```
{ precondition } => { http-request  postcondition }.
```

where `precondition`, `http-request` and `postcondition` are N3 formulas over A with the following properties:

(1) *`precondition`* describes the resources needed to execute the operation and does not contain any existential variable.
(2) *`http-request`* is an HTTP request description which describes a request which can be used to obtain the desired result of executing the operation. It contains no triple having the same subject as the `http-request`. All universal variables which occur in `http-request` do also occur in `precondition`.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix ex: <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.

{ ?image ex:smallThumbnail ?thumbnail. }
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].

  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

Listing 1. RESTdesc description of the action "obtaining a thumbnail" *(desc_thumbnail.n3)*

(3) *postcondition* describes one or more results obtained by the execution of the operation. All universal variables contained in `postcondition` also occur in `precondition`.

By making sure that the subject of any triple in the postcondition is different than the subject of the request, we make both syntactical distinguishable. Note that a RESTdesc description is an existential rule as defined in the Datalog$^{\pm}$ framework (Calì *et al.* 2011): our restriction on universal variables, that all universals in the head of the rule should also occur in the body, is very similar to Datalog (Abiteboul *et al.* 1995) and our rules allow (and expect) new existentials in the consequence.

The reasons for the restrictions will become more clear in Section 5.3, where we show that for every ground instance of the precondition, the HTTP request is sufficiently specified and the postcondition will not contain any universal variables. From an operational point of view, the remaining existential variables in the postcondition are those which are expected to be grounded through the execution of the HTTP request.

Listing 1 shows a description that explains the `smallThumbnail` relation in a hypermedia API. The *precondition* demands the existence of a `smallThumbnail` hyperlink between an `?image` resource and a `?thumbnail` resource. The HTTP request is a `GET` request to the URL of `?thumbnail`. The response to this request will be a representation of `?thumbnail`. These characteristics of this representation are detailed in the remainder of the *postcondition*. This states that the original `?image` will be in a `thumbnail` relationship (the meaning of which is defined by the DBpedia ontology [Auer *et al.* 2007]) with `?thumbnail`. Furthermore, `?thumbnail` will be an `Image` and have a `height` of `80.0`.

There are two different ways to interpret this description: First, the declarative, static way as defined in Section 4.2, which could be phrased as "the existence of the `smallThumbnail` relationship implies the existence of a `GET` request which leads to an 80px-high thumbnail of this image".

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix ex: <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.

{ ?image a dbpedia:Image. }
=>
{
  _:request http:methodName "POST";
            http:requestURI "/images/";
            http:body ?image;
            http:resp [ http:body ?image ].
  ?image ex:comments _:comments;
         ex:smallThumbnail _:thumb.
}
```

Listing 2. RESTdesc description of the action "uploading an image" *(desc_images.n3)*

The second interpretation is the operational, dynamic way. In this case, a software agent has a description of the world, against which the description is *instantiated*, i.e., the rule is applied.

Thus, given a concrete set of triples, such as

```
</photos/37> ex:smallThumbnail </photos/37/thumb>.
```

the description in Listing 1 would be instantiated to:

```
_:request http:methodName "GET";
          http:requestURI </photos/37/thumb>;
          http:resp [ http:body </photos/37/thumb> ].

</photos/37> dbpedia-owl:thumbnail </photos/37/thumb>.
</photos/37/thumb> a dbpedia:Image;
                   dbpedia-owl:height 80.0.
```

Thereby, the description has been instantiated into a concrete HTTP request that can be executed by the agent. Note how this instantiation directly results in RDF triples, which can be interpreted by any RDF-compatible client. The request has been sufficiently specified as defined in Definition 13. In addition, the instantiated postcondition explains the properties realized by this concrete request. Here, an HTTP `GET` request to `/photos/37/thumb` will result in a thumbnail of the image `/photos/37` that will have a height of 80 pixels. This dynamic interpretation is helpful to agents that want to understand the impact of performing a certain action on resources they have at their disposition.

RESTdesc descriptions are not limited to `GET` requests. They can also describe state-changing operations, for instance, those realized through the `POST` method. Listing 2 shows a description for an image upload action. The postconditions contain existential variables that are not referenced (`_:comments` and `_:thumb`), which might appear strange at first sight. However, these triples are important to an agent as they convey an *expectation* of what happens when an image is uploaded. Concretely, any uploaded image will receive a `comments` link and a `smallThumbnail`

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

link. Even though the exact values will only be known at runtime when the actual `POST` request is executed, at design-time, we are able to determine that there will be several links. The meaning of those links is in turn expressed by other descriptions, such as the one in Listing 1 discussed above.

5 Proof of a Composition's correctness

5.1 Compositions of hypermedia APIs

Having introduced a formal way to describe the function of hypermedia APIs in the last chapter, we will now focus on the proofs which can be created using these rules. More concretely, we examine proofs which confirm that a certain combination of API calls brings us to a desired goal. We therefore take a closer look at the problem itself. Given a set of possible API operations, we want to achieve a goal from an initial state. Furthermore, we might have some additional knowledge that can be incorporated. The above can be expressed in N3 as follows:

Definition 15 (*API composition problem*)
Let F be the set of simple N3 formulas over an alphabet A which contains the predicates defined in definition 12. An *API composition problem* consists of the following formulas:

- A set $H \subset F_g$ of ground formulas capturing all resource and application states the client is currently aware of, the *initial state*.
- A formula $g \in F$ with $\text{comp}^2(g) = \emptyset$, which does not contain existential variables, the *goal state* which indicates on a symbolic level what the client wants to achieve.
- A set R of RESTdesc descriptions or conjunctions of RESTdesc descriptions, describing all hypermedia APIs available to the client, the *description formulas*.
- A (possibly empty) set of N3 formulas B, the *background knowledge*, where each $b \in B$ is either a ground formula or an implication e_1=>e_2. which does not contain existential variables and where each universal variable e_2 contains does also occur in e_1.

Note that we put syntactical restrictions on our definitions: as already mentioned in Section 4.3, RESTdesc descriptions are existential rules. The constraints put on the background knowledge make the rules contained in it expressible in Datalog (Abiteboul *et al.* 1995). The initial state contains only ground formulas and as the goal does not contain nested constructions or existential variables it can also be expressed in a Datalog rule. This makes the whole problem at our disposal expressible in Datalog$^{\pm}$ (Calì *et al.* 2011). The reason for this restrictions will become clear in Section 5.3,
If we talk about a proof as explained above, we mean evidence for the fact that from $H \cup R \cup B$ follows g', where g' is a valid instance of g. As a normal hypermedia API composition problem tries to actually achieve real states and obtain instantiated objects, our final target is to make g' ground.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

5.2 Pre-proofs versus Post-proofs

The focus of this section is not how to create proofs but how to verify their correctness, given that creation has already been performed. This is not unlike the notion of proof in the classical semantic Web vision (Berners-Lee *et al.* 2001), where it is defined as a means to assert the validity of a piece of (static) information. In this paper, we extend this classical notion or proofs to also include *dynamic* information, *i.e.,* data generated by Web APIs. As a consequence of this dynamic nature, we introduce two different kinds of proofs for an API composition problem (H, g, R, B) as defined in Definition 15:

a pre-execution proof *("pre-proof")*, in which the assumption is made that execution of all API operations will behave as expected, i.e., a proof in a classical sense which provides evidence for

$$H \cup R \cup B \models g'$$

a post-execution proof *("post-proof")*, in which an additional evidence for the goal is provided by the API operations' actual execution results, which are purely static data. This means the resulting proof itself confirms that

$$H \cup R \cup B \cup \{\text{execution results}\} \models g''$$

with g' and g'' being instances of g. Note that technically speaking, for the same API operation every pre-proof is also a post-proof but, if its execution actually yields a non-empty result, not every post-proof is a pre-proof. We are especially interested in the post-proofs of an API operation which are not its pre-proofs, and call those proofs **proper post-execution proofs**. In other words, proper post proofs are those proofs that actually make use of the information gained by the API call. Intuitively, we expect those proofs to be shorter than the initial pre-proof, as they have more relevant knowledge at their disposal.

The distinction between pre- and (proper) post-proof exists because, although error handling is possible, one can never *guarantee* that a composition that has proven to work in theory will *always* and reliably achieve the desired result in practice, since the individual steps can fail. Some errors (such as disk failures or power outages) cannot be predicted and may cause a composition not to reach a goal that would normally be possible. Furthermore, a composition can only be as adequate as its individual descriptions, which could contain mistakes. Therefore, the pre-proof necessarily has to make the additional assumption that all APIs will function according to their description. The pre-proof's objective thus becomes: *"assuming correct behavior of all APIs, the composition must lead to the fulfillment of the goal."*

While a pre-proof can be validated before a composition's execution, the creation and validation of a proper post-proof can only happen when the execution's results are available. At that stage, however, the environment's nature is no longer dynamic, since the APIs' results are effectively available as data. A proper post-proof is therefore equivalent to a data-based proof, wherein the executed API operations contribute to the provenance information. This provenance can be used to link the

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

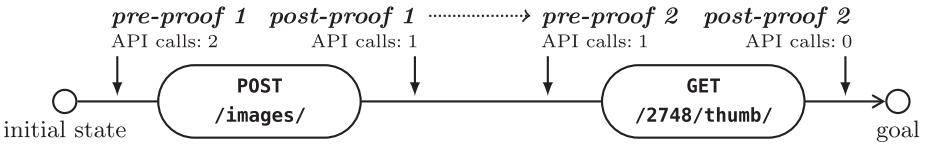


Fig. 2. A proper post-proof of one operation becomes the pre-proof of the next.

proper post-proof to the pre-proof, indicating whether the non-failure assumption has corresponded to reality. In the ideal case, this assumption indeed holds and the proper post-proof is essentially a revision of the pre-proof in which the actual values returned by the hypermedia APIs are filled in. The objective of the post-proof is thus *"given the execution results of some API operations, the composition must lead to the fulfillment of the goal"*. Thereby, a proper post-proof after one operation becomes the pre-proof of the next operation, as indicated in Figure 2.

Regular proofs do not contain dynamic information that needs to be obtained at runtime. The extension to pre-proofs that contain dynamic information, necessary to verify the correctness of a composition *before* it is executed, requires a mechanism to express when API operations are performed. RESTdesc descriptions can be considered rules that *simulate* the execution of a hypermedia API, using existentially quantified variables as placeholders for the API's results, which are still unknown at the time the pre-proof is to be verified.

5.3 Anatomy of a hypermedia API composition proof

The N3 proof vocabulary created in the context of the Semantic Web Application Platform (Berners-Lee 2000) enables us to formalize proofs in a machine-readable way. This subsection gives a short introduction into the terminology used and the resulting proofs, focusing on the aspects relevant to our purposes.

A proof is a conjunction of N3 formulas describing inference steps a reasoner has performed to come to a certain conclusion, so called *proof steps*.

The vocabulary distinguishes between four different kinds of proof steps. We write them as deduction rules, using "$\vdash$".

Definition 16 (*Proof steps*)
Let F be the set of simple formulas over an N3 alphabet A, $\Gamma \subset F$ a set of formulas and $f, f_1, f_2, g \in F$. A *proof step* is one of the following inference rules:

(1) *Axiom:* If $f \in \Gamma$, then $\Gamma \vdash f$.
(2) *Conjunction elimination:* If $\Gamma \vdash f_1 f_2$, then $\Gamma \vdash f_1$ and $\Gamma \vdash f_2$.
(3) *Conjunction introduction:* Let $\Gamma \vdash f_1$ and $\Gamma \vdash f_2$ and let

$$\sigma : V_U \to V_U \setminus (\mathrm{comp}^1(f_1) \cup \mathrm{comp}^2(f_1)) \text{ and } \mu : V_E \to V_E \setminus \mathrm{comp}(f_1)$$

be substitutions. Let $f_2' = f_2 \sigma^t \mu^c$, then

$$\Gamma \vdash f_1 f_2'$$

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

(4) *Generalized modus ponens:* If $\Gamma \vdash \{f_1\}{=}{>}\{f_2\}.$ and $\Gamma \vdash g$ and there exists a substitution $\sigma : \text{comp}(f_1) \cap V_U \rightarrow E_e$ such that

$$(\{f_1\}{=}{>}\{f_2\}.)\sigma^t = (\{f_1'\}{=}{>}\{f_2'\}.) \text{ and } f_1' = g$$

then $\Gamma \vdash f_2'$.

Theorem 17 (*Correctness of proof calculus*)
Let Φ be a set of N3 formulas and ϕ a formula over the same N3 alphabet A. Then, the following holds:

$$\text{If } \Phi \vdash \phi \text{ then } \Phi \models \phi.$$

Proof
We prove that every proof step is correct.

(1) *Axiom:* For the axiom step, the claim is trivial, as it corresponds to Definition 11.
(2) *Conjunction elimination:* Let $\Phi \models f_1 f_2$ and let $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ be a model for Φ and $f_1 f_2$. If $f_1 f_2$ is universal free and $\text{comp}(f_1 f_2) \cap V_E = \emptyset$, the claim follows immediately from Definition 9.3b.
If $f_1 f_2$ universal free and $\text{comp}(f_1 f_2) \neq \emptyset$ let $\mu : \text{comp}(f_1 f_2) \rightarrow E_g$ be a substitution such that $\mathfrak{I} \models (f_1 f_2)\mu^c$. Then, $\mathfrak{I} \models (f_1\mu^c)(f_2\mu^c)$, thus $\mathfrak{I} \models (f_1\mu^c)$ and $\mathfrak{I} \models (f_2\mu^c)$.
If $f_1 f_2$ are not universal free, then $\mathfrak{I} \models (f_1 f_2)\sigma^t$ for all substitutions $\sigma : V_U \rightarrow E_e$. The claim follows by the same argument as above.
(3) *Conjunction introduction:* Let $\Phi \models f_1$, $\Phi \models f_2$ and let $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ be a model for Φ, f_1 and f_2. As the renaming substitutions σ and μ do not change the meaning of a formula, for $f_2' = f_2\sigma^t\mu^c$ the following holds: $\mathfrak{I} \models f_2'$. It immediately follows that $\mathfrak{I} \models f_1 f_2'$.
(4) *Generalized modus ponens:* the claim follows directly from Definitions 9.1 and 9.3c.

□

Applied on a API composition problem, we get the following consequence:

Corollary 18 (*Correctness of API composition proofs*)
Let (H, g, R, B) be an API composition problem and g' an instance of g then the following holds:

$$\text{If } H \cup R \cup B \vdash g' \text{ then } H \cup R \cup B \models g'$$

We will examine the generalized modus ponens in more detail, as this is the proof step where implication rules, such as RESTdesc descriptions, are applied.

Lemma 19
Let A be an N3 alphabet, $f \in F_g$ a simple ground formula and $\{f_1\}{=}{>}\{f_2\} \in F$, a simple implication formula where all universal variables which occur in f_2 also occur in f_1. If the generalized modus ponens is applicable to f and $\{f_1\}{=}{>}\{f_2\}$, then the resulting formula does not contain universal variables.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Proof
Let $\sigma : V_U \to E_e$ be a substitution such that $(\{f_1\}\texttt{=>}\{f_2\}.)\sigma^t = \{f\}\texttt{=>}\{f_2'\}.$ As f is a ground formula, $\text{range}(\sigma|_{V_U \cap (\text{comp}^1(f_1) \cup \text{comp}^2(f_1))}) \subset E_g$. Due to the condition that every universal variable of f_2 is also in f_1, i.e.,

$$((\text{comp}^1(f_2) \cup \text{comp}^2(f_2)) \cap V_U) \subset (\text{comp}^1(f_1) \cup \text{comp}^2(f_1)) \cap V_U)$$

the claim follows. □

As HTTP requests in RESTdesc descriptions only contain one leading existential to represent the HTTP message, and RESTdesc descriptions fulfill the conditions of Lemma 19, we arrive at the following consequence:

Corollary 20
Every application of a restdesc description to a ground formula results in a sufficiently specified HTTP request and a postcondition which does not contain any universal variables.

The first step of Definition 16 includes from a technical point of view also the parsing of a source. In the N3 proof vocabulary[1] we will discuss next, this step is therefore named after this action.

Definition 21 (*Proof vocabulary*)
Let A be an N3 alphabet and $\mathfrak{I} = (\mathscr{D}, \mathfrak{a}, \mathfrak{p})$ be an interpretation of its formulas. Let $\mathtt{x}, \mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_n \in U$ be N3 representations of proof steps and $\mathtt{z}_1, \mathtt{z}_2, \mathtt{z}_3 \in U$.

(1) Proof step types:

- $\mathfrak{I} \models$ `x a r:Proof.` iff `x` is the proof step which leads to the proven result.
- $\mathfrak{I} \models$ `x a r:Parsing.` iff `x` is a parsed axiom.
- $\mathfrak{I} \models$ `x a r:Conjunction.` iff `x` is a conjunction introduction.
- $\mathfrak{I} \models$ `x a r:Inference.` iff `x` is a generalized modus ponens.
- $\mathfrak{I} \models$ `x a r:Extraction.` iff `x` is a conjunction elimination.

(2) Proof predicates:

- $\mathfrak{I} \models$ `x r:gives {`f`}.` iff $f \in F$ is the formula obtained by applying `x`.
- $\mathfrak{I} \models$ `x r:source u.` iff `x` is a parsed axiom and $\mathtt{u} \in U$ is the URI of the parsed axiom's source.
- $\mathfrak{I} \models$ `x r:component y.` iff `x` is a conjunction introduction and `y` is a proof step which gives one of its components.
- $\mathfrak{I} \models$ `x r:rule y.` iff `x` is a generalized modus ponens and `y` is the proof step which leads to the applied implication.
- $\mathfrak{I} \models$ `x r:evidence (`$\mathtt{y}_1, \ldots, \mathtt{y}_n$`).` iff `x` is a generalized modus ponens and $\mathtt{y}_1, \ldots, \mathtt{y}_n$ are the proof steps which lead to the formulas used for the unification with the antecedent of the implication.
- $\mathfrak{I} \models$ `x r:because y.` iff `x` is a conjunction elimination and `y` is the proof step which yields the to-be-eliminated conjunction.

[1] The vocabulary's RDF definition can be found at: `http://www.w3.org/2000/10/swap/reason#`.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
1 @prefix dbpedia: <http://dbpedia.org/resource/>.
2
3 <lena.jpg> a dbpedia:Image.
```

Listing 3. The initial knowledge of the agent *(agent_knowledge.n3)*

```
1 @prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
2
3 { <lena.jpg> dbpedia-owl:thumbnail ?thumbnail. }
4 =>
5 { <lena.jpg> dbpedia-owl:thumbnail ?thumbnail. }.
```

Listing 4. A filter rule expressing the agent's goal *(agent_goal.n3)*

(3) Substitutions:

- $\mathfrak{I} \models$ `x r:binding` z_1. iff `x` includes a substitution z_1.
- $\mathfrak{I} \models z_1$ `r:variable` z_2. iff z_1 is a substitution whose domain is $\{z_2\}$.
- $\mathfrak{I} \models z_1$ `r:boundTo` z_3. iff z_1 is a substitution whose range is $\{z_3\}$.

To produce a proof for an API composition problem, the reasoner needs to be aware of all formulas at its disposal (in our case $H \cup R \cup B$) and of the goal which it is expected to prove. The latter is given to the reasoner as the consequence of a *filter rule* `{`f`} => {`g`}.` This triggers the reasoner to prove an instance of f and in case of success, return each provable ground instance of g if possible, or a provable instance containing existentials otherwise. For brevity, not all reasoners display every proof step in a proof: especially conjunction elimination and introduction are often omitted. However, to the best of our knowledge, all reasoners' proofs contain all applications of `r:Inference` leading to a goal g, which allows us to measure a proof's length by counting applications of the generalized modus ponens.

5.4 Discussion of a composition proof example

5.4.1 Overview

Listing 5 displays an example of a hypermedia API composition proof. In addition to the two RESTdesc descriptions from Listings 1 and 2 (R in an API composition problem), it involves the goal file Listing 4 (`{g} => {g}.`) and the input file Listing 3 (the initial knowledge H). The proof, generated by EYE (Verborgh and De Roo 2015), explains how, given an image, a thumbnail of this image can be obtained. Its main element is `r:Proof` [*line 8*], which is a conjunction of different components, indicated by `r:component`. In this example, there is only one, but there can be multiple. The conclusion of the proof (object of the `r:gives` relation) is the triple `<lena.jpg> dbpedia-owl:thumbnail _:sk3` (with `_:sk3` an existential variable). This captures the fact that `lena.jpg` has a certain thumbnail, which is the consequent of the filter rule in Listing 4.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix ex: <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix n3: <http://www.w3.org/2004/06/rei#>.
@prefix r: <http://www.w3.org/2000/10/swap/reason#>.

<#proof> a r:Proof, r:Conjunction;
  r:component <#lemma1>;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. }.

<#lemma1> a r:Inference;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. };
  r:evidence (<#lemma2>);
  r:binding [ r:variable [ n3:uri "var#x0"]; r:boundTo [ n3:nodeId "_:sk3"]];
  r:rule <#lemma7>.

<#lemma2> a r:Inference;
  r:gives { _:sk4 http:methodName "GET".
            _:sk4 http:requestURI _:sk3.
            _:sk4 http:resp _:sk5.
            _:sk5 http:body _:sk3.
            <lena.jpg> dbpedia-owl:thumbnail _:sk3.
            _:sk3 a dbpedia:Image.
            _:sk3 dbpedia-owl:height 80.0. };
  r:evidence (<#lemma3>);
  r:binding [ r:variable [ n3:uri "var#x0"]; r:boundTo [ n3:uri "lena.jpg"]];
  r:binding [ r:variable [ n3:uri "var#x1"]; r:boundTo [ n3:nodeId "_:sk3"]];
  r:binding [ r:variable [ n3:uri "var#x2"]; r:boundTo [ n3:nodeId "_:sk4"]];
  r:binding [ r:variable [ n3:uri "var#x3"]; r:boundTo [ n3:nodeId "_:sk5"]];
  r:rule <#lemma4>.

<#lemma3> a r:Inference;
  r:gives { _:sk0 http:methodName "POST".
            _:sk0 http:requestURI "/images/".
            _:sk0 http:body <lena.jpg>.
            _:sk0 http:resp _:sk1.
            _:sk1 http:body <lena.jpg>.
            <lena.jpg> ex:comments _:sk2.
            <lena.jpg> ex:smallThumbnail _:sk3. };
  r:binding [ r:variable [ n3:uri "var#x0"]; r:boundTo [ n3:uri "lena.jpg"]];
  r:binding [ r:variable [ n3:uri "var#x1"]; r:boundTo [ n3:nodeId "_:sk0"]];
  r:binding [ r:variable [ n3:uri "var#x2"]; r:boundTo [ n3:nodeId "_:sk1"]];
  r:binding [ r:variable [ n3:uri "var#x3"]; r:boundTo [ n3:nodeId "_:sk2"]];
  r:binding [ r:variable [ n3:uri "var#x4"]; r:boundTo [ n3:nodeId "_:sk3"]];
  r:evidence (<#lemma6>);
  r:rule <#lemma5>.

<#lemma4> a r:Extraction; r:because [ a r:Parsing; r:source <desc_thumbnail> ].
<#lemma5> a r:Extraction; r:because [ a r:Parsing; r:source <desc_images> ].
<#lemma6> a r:Extraction; r:because [ a r:Parsing; r:source <agent_knowledge> ].
<#lemma7> a r:Extraction; r:because [ a r:Parsing; r:source <agent_goal> ].
```

Listing 5. Example hypermedia API composition proof

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

5.4.2 Extractions and inferences

EYE represents `r:extractions` and `r:inferences` as lemmata, which can be reused in different proof steps. The extractions, Lemmata 4 to 7 [*lines 49–52*], are fairly trivial as none of the underlying formulas is a real conjunction and will therefore not be discussed in detail. In this proof, they just give the formulas specified in Listing 1 to 4 with renamed variables. The inferences describe the actual reasoning carried out and thus merit a closer inspection. We will follow the path backwards from the proof's conclusion, tracing back inferences until we arrive at the atomic facts that are the starting point of the proof.

5.4.3 Filter rule

The justification for the conclusion consists in this case of a single component, namely Lemma 1 [*line 12*]. This lemma is an application of the modus ponens using the implication of Lemma 7, which is the file `agent_goal.n3` shown in Listing 4. This rule has been instantiated according to the variable bindings indicated by `r:binding`: here, the variable `?thumbnail` (`var#x0`) is bound to the existential variable `_:sk3`. The unification uses a part of the result of Lemma 2 as indicated by `r:evidence`, the performed conjunction elimination is not listed in this proof. Substituting `_:sk3` for `?thumbnail` in Listing 4 indeed gives the desired conclusion `<lena.jpg> dbpedia-owl:thumbnail _:sk3`, which contributes to the final result.

5.4.4 Thumbnail operation

Lemma 2 [*line 18*] is another `r:inference`, this time applying the rule from Lemma 4, which is the RESTdesc description in Listing 1 that explains what happens when an image is uploaded. The instantiation is again detailed with `r:binding` statements. The `?image` variable (`var#x0`) is bound to `lena.jpg`, the `?thumbnail` variable (`var#1`) to the existential `_:sk3`. This is only possible because a statement `<lena.jpg> ex:smallThumbnail _:sk3` is proven; its derivation will be detailed shortly. The other variables `var#x2` and `var#x3` refer to the request and response resources in the consequent of Listing 1 and are instantiated with new existentials (`_:sk4` and `_:sk5`). This lemma is in turn made possible by another one.

5.4.5 Upload operation

Lemma 3 [*line 33*] explains the derivation of the triple that is a necessary condition for Lemma 2: `<lena.jpg> ex:smallThumbnail _:sk3`. The rule used for the inference is that from the upload action in Listing 2 (Lemma 5), instantiated with the initial knowledge of the agent that it has access to an image (Listing 3/Lemma 6). Substituting this knowledge into the rule by binding `?image` (`var#x0`) to `lena.jpg`, gives the triples of the instantiated consequent, as shown by the `r:gives` predicate. Note in particular the newly created existential `_:sk3`; this entails the triple `<lena.jpg> ex:smallThumbnail _:sk3` which is needed for Lemma 2. Lemma 3 itself is justified by Lemma 6 (Listing 3), which is a simple extraction that stands on itself. Hence, we have reached the starting proof's starting point.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

5.4.6 From start to end

As a summary, we will briefly follow the proof in a forward way. Extracted from the background knowledge in Listing 3, the triple `<lena.jpg> a dbpedia:Image` triggers the image upload rule from Listing 1, which yields `<lena.jpg> dbpedia-owl:thumbnail _:sk3` for some existential variable `_:sk3`. This triple in turn triggers the thumbnail rule from Listing 1, which yields `<lena.jpg> ex:smallThumbnail _:sk3`. Finally, this is the input for the filter rule from Listing 4, which yields the final result of the proof: the image has some thumbnail. So given the background knowledge and the two hypermedia API descriptions, this proof verifies that the goal follows from them.

5.5 Hypermedia API operations inside a proof

The proof in Listing 5 is special in the sense that some of its implication rules, namely Listing 1 and 2, are actually hypermedia API descriptions. That means they do not fulfill an actual ontological implication. Instead, they convey dynamic information. Therefore, those steps in the proof can be interpreted as HTTP requests that should be performed in order to achieve the desired result. This proof is indeed a pre-proof: it is valid under the assumption that the described HTTP requests will behave as expected, which can never be guaranteed on an environment such as the Internet. The instantiation of a hypermedia API description turns it into the description of a concrete API operation. For instance, Lemma 3 [*line 33*] contains the following operation:

```
_:sk0 http:methodName "POST".
_:sk0 http:requestURI "/images/".
_:sk0 http:body <lena.jpg>.
_:sk0 http:resp _:sk1.
_:sk1 http:body <lena.jpg>.
<lena.jpg> ex:comments _:sk2.
<lena.jpg> ex:smallThumbnail _:sk3.
```

This instructs an agent to perform an HTTP `POST` request (`_:sk0`) to `/images/` with `lena.jpg` as the request body. This request will return a response (`_:sk1`) with a representation of `lena.jpg`, which will contain `ex:comments` and `ex:smallThumbnail` links. Since this is a hypermedia API, the link targets are not yet known at this stage. Therefore, they are represented by generated existential variables `_:sk2` and `_:sk3`. At runtime, the HTTP request will return the actual link targets, but at the pre-proof stage, it suffices to know that some target for those links will exist.

Indeed, the outcome of this rule—the fact that *some* `smallThumbnail` links will exist—serves as input for the next API operation in Lemma 2 [*line 18*]. The consequent of this rule is instantiated as follows:

```
_:sk4 http:methodName "GET".
_:sk4 http:requestURI _:sk3.
_:sk4 http:resp _:sk5.
_:sk5 http:body _:sk3.
```

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

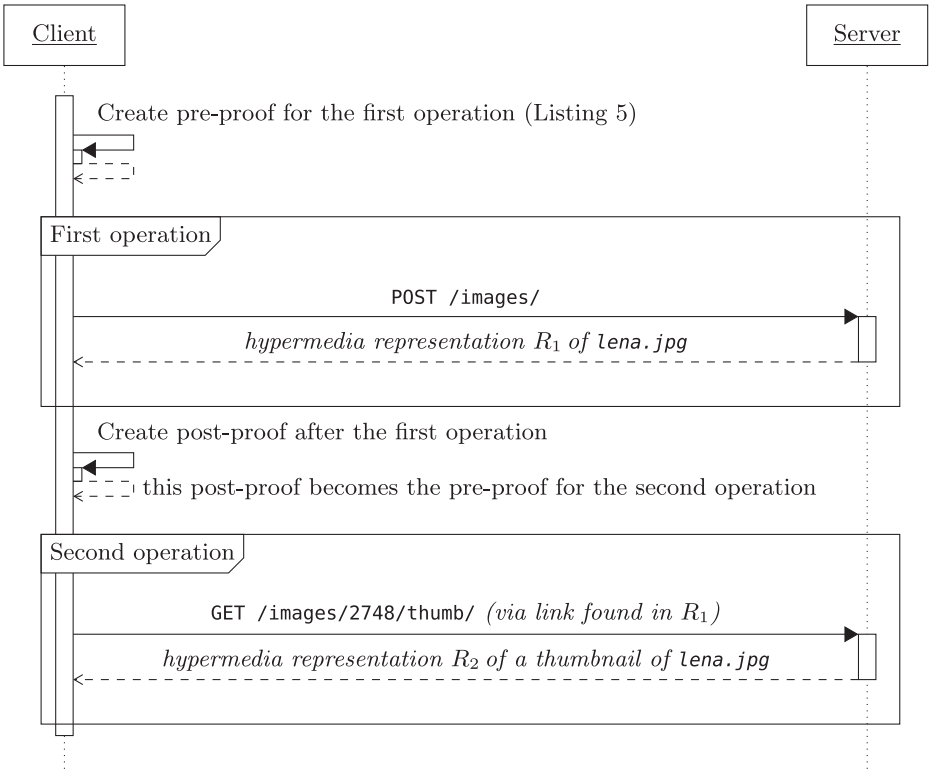


Fig. 3. The *a posteriori* UML sequence diagram of an example execution shows how the second operation happened through a link in the first operation.

```
<lena.jpg> dbpedia-owl:thumbnail _:sk3.
_:sk3 a dbpedia:Image.
_:sk3 dbpedia-owl:height 80.0
```

This describes a `GET` request (`_:sk4`) to the URL `_:sk3`, which will return a representation of a thumbnail that is 80 pixels high. This request is interesting because it is *incomplete*: `_:sk3` is not a concrete URL that can be filled in. However, this identifier is the same variable as the one in Lemma 3, so this description essentially states that whatever will be the target of the `smallThumbnail` link in the previous `POST` request should be the URL of the present `GET` request. The existential variables thus serve as placeholders for values that will be the result of actual API operations.

While the proof above is a pre-proof, a proper post-proof can be obtained by actually executing the `POST` HTTP request, which has all values necessary for execution (as opposed to the `GET` request where the URL is still undetermined). This execution will result in a concrete value for the `comments` and `smallThumbnail` link placeholders `_:sk2` and `_:sk3`. They lead to a proper post-proof that uses these concrete values, and hence that proof does not need the assumption that the `POST` request will execute successfully (because evidence shows it did). Figure 3 shows the UML sequence diagram of an example interaction between the client and the server.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

As stated in its definition, a pre-proof implicitly assumes that each API will indeed deliver the functionality as stated in its RESTdesc description. The proof thus only holds under that assumption. For example, if a power outage occurs during the calculation of the aspect ratio, the placeholder will not be instantiated with an actual value during the execution, which can pose a threat to subsequent hypermedia API operations that depend on this value. However, the failure of a single API operation does not necessarily imply the intended result cannot be achieved. Rather, it means the assumption of the pre-proof was invalid and an alternative pre-proof—a new hypermedia API composition—should be created, starting from the current application state. Such a pragmatic approach to proofs containing hypermedia API operations is unavoidable: no matter how low the probability of a certain operation to fail, failures can never be eliminated. Therefore, pragmatism ensures that planning in advance is possible. Each proof should be stored along with its assumptions in order to understand the context in which it can be used.

6 Hypermedia-driven composition generation and execution

In contrast to fully plan-based methods, the steps in the composition obtained through reasoner-based composition of hypermedia APIs are not executed blindly. Instead, the interaction is driven by the hypermedia responses from the server; the composition in the proof only serves as *guidance* for the client, and as a guarantee (to the extent possible) that the desired goal can be reached. The composition that starts from the current state helps an agent decide what its next step towards that goal should be. Once this step has been taken, the rest of the pre-proof is discarded because it is based on outdated information. After the request, the state is augmented with the information inside the server's response. This new state becomes the input for a new pre-proof that takes into account the actual situation, instead of the expected (and incomplete) values from the hypermedia API description. In this section, we will detail this iterative composition generation and execution.

6.1 Goal-oriented composition generation

Creating a composition that satisfies a goal comes down to generating a proof that supports the goal. Inside this proof, the necessary hypermedia API operations will be incorporated as instantiated rules. Proof-based composition generation, unlike other composition techniques, requires no composition-specific tools or algorithms. A generic reasoner that supports the rule language in which the hypermedia APIs are described is capable of generating a proof containing the composition. For example, since RESTdesc descriptions are expressed in the N3 language, compositions of hypermedia APIs described with RESTdesc can be performed by any N3 reasoner with proof support. The fact that proof-based composition can be performed by existing reasoners is an advantage in itself, because no new software has to be implemented and tested. Furthermore, this offers the following benefits.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Incorporation of external knowledge: Existing RDF knowledge can directly be incorporated into the composition process. Whereas composition algorithms that are specifically tailored to certain description models usually operate on closed worlds, generic semantic Web reasoners are built to incorporate knowledge from various sources. For example, existing OWL and RDF ontologies can be used to compose hypermedia APIs described with different vocabularies.

Evolution of reasoners: Many implementations of reasoners exist and they continue to be updated to allow enhanced performance and possibilities. The proof-based composition method directly benefits from these innovations. This also counters the problem that many single-purpose composition algorithms are seldom updated after their creation because they are so specific.

Independent validation: When dealing with proof and trust on the Web, it is especially important that the validation can happen by an independent party. Since different reasoners and validators exist, the composition proof can be validated independently. This contrasts with other composition approaches, whose algorithms have to be trusted.

In order to make a reasoner generate a pre-proof of a composition, it must be invoked with the initial state, the available hypermedia APIs, and the desired goal. Here, we will examine the case for N3 reasoners and hypermedia APIs described with RESTdesc N3 rules, but the principle of proof-based composition is generalizable to all families of inference rules.

The hypermedia API descriptions include all those APIs available to the client. In practice, the number of supplied available hypermedia APIs would be substantially higher than the number of APIs in the resulting composition. The background knowledge can, for example, consist of ontologies and business rules. The reasoner will try to infer the goal state, asserting the other inputs as part of the ground truth. The initial state and background knowledge should correspond to reality, regardless of the results of the actual execution, provided the descriptions are accurate. In contrast, the API description rules only hold under the assumption of successful execution, due to the nature of the pre-proof.

If the reasoner can infer the goal state given the ground truth, we can conclude that a composition *exists*. To obtain the details of the composition, the reasoner must return the proof of the inference, i.e., the data and rules applied to achieve the goal. Inside this proof, there will be placeholders for return values by the server that are unknown at design time. The proof will be structured as in Listing 5, where the initial state was Listing 3, the goal state Listing 4, and the descriptions Listings 1 and 2. No background knowledge was needed, but it could have been useful for instance if the image of the initial state was described in different ontology, in which case the conversion to the DBpedia ontology would be necessary.

6.2 Hypermedia-driven execution

In order to achieve a certain goal in a hypermedia-driven way, the following process steps can be followed.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Definition 22 (*Pragmatic proof algorithm*)
Given an API composition problem with an initial state H, goal g, description formulas R, and background knowledge B, we define the pragmatic proof algorithm as follows:

(1) Start an N_3 reasoner to generate a pre-proof for (R, g, H, B).
  (a) If the reasoner is not able to generate a proof, halt with *failure*.
  (b) Else scan the pre-proof for applications of rules of R, set the number of these applications to n_{pre}.
(2) Check n_{pre}:
  (a) If $n_{\text{pre}} = 0$, halt with *success*.
  (b) Else continue with Step 3.
(3) Out of the pre-proof, select a sufficiently specified HTTP request description which is part of the application of a rule $r \in R$.
(4) Execute the described HTTP request and parse the (possibly empty) server response to a set of ground formulas G.
(5) Invoke the reasoner with the new API composition problem $(R, g, H \cup G, B)$ to produce a post-proof.
(6) Determine n_{post}:
  (a) If the reasoner was not able to generate a proof, set $n_{\text{post}} := n_{\text{pre}}$.
  (b) Else scan the proof for the number of inference steps which are using rules from R and set this number of steps to n_{post}.
(7) Compare n_{post} with n_{pre}:
  (a) If $n_{\text{post}} \geqslant n_{\text{pre}}$ go back to Step 1 with the new API composition problem $(R \setminus \{r\}, g, H, B)$.
  (b) If $n_{\text{post}} < n_{\text{pre}}$, the post-proof can be used as the next pre-proof. Set $n_{\text{pre}} := n_{\text{post}}$ and continue with Step 2.

Before having a more theoretical look at the results of this algorithm, let us run through a possible execution of the composition example introduced previously.

- *(Step 1)* Given the background knowledge, initial state, and goal, the reasoner generates the pre-proof from Listing 5, which contains $n_{\text{pre}} = 2$ API operations.
- *(Step 2)* $n_{\text{pre}} \neq 0$, so continue with Step 3.
- *(Step 3)* The HTTP request to upload the image is the only one that is sufficiently instantiated, so it is selected.
- *(Step 4)* Execute the HTTP request by posting the image to `/images/`, and retrieve a hypermedia response. Inside this hypermedia response, there is a `comments` link to `/comments/about/images/37` and a `smallThumbnail` link to `/images/37/thumb/`. They are added to G.
- *(Step 5)* A post-proof is produced from the new state, revealing that the goal can now be completed with one API operation. Indeed, only an HTTP `GET` request to `/image/37/thumb` is needed.
- *(Step 6)* The above means that $n_{\text{post}} = 1$.
- *(Step 7)* $n_{\text{post}} = 1 < n_{\text{pre}} = 2$, so set $n_{\text{pre}} := 1$ and continue with Step 2.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

- *(Step 2)* $n_{\text{pre}} = 1 \neq 0$, so continue with Step 3.
- *(Step 3)* Select the only remaining HTTP request in the pre-proof.
- *(Step 4)* Execute the `GET` request to `/image/37/thumb` and thereby obtain a representation of the thumbnail of the image.
- *(Step 5)* Generate the post-proof; it consists entirely of data as the necessary information to reach the goal has been obtained.
- *(Step 6)* No rules of R are applied, so $n_{\text{post}} = 0$.
- *(Step 7)* $n_{\text{post}} = 0 < n_{\text{pre}} = 1$, so set $n_{\text{pre}} := 0$ and continue with Step 2.
- *(Step 2)* $n_{\text{pre}} = 0$, so halt with *success*.

This example shows how the proof guides the process, but hypermedia drives the interaction. For instance, the URL needed for the `GET` request was not hard-coded: it was obtained as a hypermedia control from the server. This means that, even if the server changes its internal URL structure or the layout of the representation, the interaction can still take place. The client needs the RESTdesc descriptions to find out whether the complex goal is possible and what first steps it should take. Otherwise, it would have no way of knowing that the upload of an image results in a link to the thumbnail. However, once this expectation is there, the client navigates through hypermedia. We can compare this to driving with a map: the map gives the overall picture, but the actual way finding happens based on the actual roads and scenery when somebody undertakes the journey.

There are several reasons, why the situations in 1(a) or 6(a) can occur: the reasoner could have a technical problem, it could detect inconsistencies (a fulfilled antecedent of a rule with false in the consequent), it could simply be that there is no instance g' of g which is a logical consequence of $H \cup R \cup B$, but even if such a g' exists the reasoning problem is undecidable. This is because RESTdesc descriptions are rules with new existential variables in the consequence, which makes the problem in general undecidable, as discussed by Baget *et al.* (2011).

However, we can show that the following holds:

Theorem 23
Given an execution of the algorithm in Theorem 22 that requires n executions of the reasoner (in Steps 1 and 5). If all n reasoning runs terminate, the algorithm terminates as well. Furthermore, if the algorithm halts with *success*, its output is a ground instance of the goal state.

Proof
We first show termination: if the algorithm does not terminate, it especially never reaches the Steps 1(a) and 2(a) and Steps 1 and 2 always result in option (b). All formulas in $H \cup B$ are either ground formulas or simple rule formulas which do not contain existentials and which fulfill Lemma 19. Therefore, no atomic formula or conjunction of atomic formulas which can be obtained by applying the proof steps from Definition 16 on $H \cup B$ contains universals or existentials. If for a pre-proof $n_{\text{pre}} > 0$ holds, it must by Corollary 20 contain at least one sufficiently specified HTTP request description. So, Step 3 and the following Steps 4–6 can always be executed. Step 7(a) reduces the set of RESTdesc descriptions, it can only

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

be performed $|R|$ times. Starting from a fixed pre-proof pre_0, Step 7(b) can only be applied n_{pre_0} times. Thus, the algorithm terminates.

As every operation which changes the API composition problem for the pre-proof to be checked in Step 2, preserves the syntactic properties of the sets of formulas involved, it is enough to show that the result of every pre-proof of an API composition problem which does not contain applications of RESTdesc rules is ground. This follows by the same arguments as above. As $H \cup B \cup \{\{\underline{\text{g}}\} \Rightarrow \{\underline{\text{g}}\}.\}$ only contains ground formulas and rules which fulfill the conditions of Lemma 19, it is not possible to derive atomic formulas or conjunctions of atomic formulas which contain existentials or universals thus the result of the proof is ground. □

7 Semi-automated description generation

One of the bottlenecks with traditional description-based methods of Web APIs and services is that these descriptions have to be created, which is mostly a manual task. Consequently, without a method that facilitates the creation of such descriptions, the overall concept of description-based composition and execution might not successfully transition from theory to practice. This section explains how RESTdesc descriptions can be created by a computer-assisted process.

We define *semi-automated RESTdesc description generation* as a process that takes as input a series of HTTP requests and responses performed by one or more persons, and provides as output skeletons for RESTdesc descriptions, which a user can then further refine to a final description. We rely on the fact that RESTdesc descriptions capture the *expectations* of resources. For example, Listing 2 captures the expectation that the upload of an image will result in links to comments and a thumbnail. RESTdesc thus describes the generic hypermedia controls that will be available on such image resources.

The idea behind the process is to extract the hypermedia controls from a series of requests performed by people. We will describe the strategy using an exemplary series of interactions, displayed in Listing 6. To create this particular series, a user uploaded two images and obtained their thumbnails.

First, the process needs to identify the different kinds of steps and thus resources. This happens by performing clustering on the HTTP responses with, for instance, a string similarity algorithm as distance function. In this case, responses 1 and 3 are highly similar, and so are responses 2 and 4. Therefore, they are assigned into two different clusters. Note that such clustering is especially realistic because Web APIs typically generate responses based on templates, which contributes to a high structural similarity for responses of the same kind that hence follow the same template. User interaction can influence the clustering sensitivity, and possibly manually change assignments. Since the two clusters, corresponding to two types of operations, are correct in this example, nothing needs to change.

Next, for each cluster, the common elements in responses and their corresponding requests are identified. In this case, the first cluster contains two `POST` requests to `/images/`, and both responses contain some resource that is an image, and has

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
HTTP request 1: POST to /images/ with image1.jpg as body
HTTP response 1:
@prefix ex: <http://example.org/image#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.

</images/24> a dbpedia:Image.
             ex:comments </images/24/comments>.
             ex:smallThumbnail </images/24/thumbnail>.


HTTP request 2: GET to /images/24/thumbnail
HTTP response 2:
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.

</images/24> dbpedia-owl:thumbnail </images/24/thumbnail>.
</images/24/thumbnail> a dbpedia:Image;
                       dbpedia-owl:height 80.0.


HTTP request 3: POST to /images/ with image2.jpg as body
HTTP response 3:
@prefix ex: <http://example.org/image#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.

</images/25> a dbpedia:Image.
             ex:comments </images/25/comments>.
             ex:smallThumbnail </images/25/thumbnail>.


HTTP request 4: GET to /images/25/thumbnail
HTTP response 4:
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.

</images/25> dbpedia-owl:thumbnail </images/25/thumbnail>.
</images/25/thumbnail> a dbpedia:Image;
                       dbpedia-owl:height 80.0.
```

Listing 6. Example user interaction with a Web API, used as input for the RESTdesc description generation process.

a link to comments and images. This allows the algorithm to produce a skeleton as follows:

```
{ ?object1 a :localFile. }
=>
{
  _:request http:methodName "POST";
            http:requestURI "/images/";
            http:body ?object1;
            http:resp [ http:body _:object2 ].
  _:object2 a dbpedia:Image;
            ex:comments _:object3;
            ex:smallThumbnail _:object4.
}
```

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Note how this is very close to Listing 2. The user can still improve this description by specifying that `?object1` and `_object2` are the same entity, and that `?object1` is a `dbpedia:Image` already in the antecedent. After these changes, and possibly renaming variables and blank nodes for legibility, the description is the same as that of Listing 2 and thus correct.

Identifying common elements in the second cluster leads to the following skeleton:

```
{}
=>
{
  _:request http:methodName "GET";
            http:requestURI _:object1;
            http:resp [ http:body _:object1 ].
  _:object2 dbpedia-owl:thumbnail _:object1.
  _:object1 a dbpedia:Image;
            dbpedia-owl:height 80.0.
}.
```

The fact that the request URI and the body refer to the same entity (`_:object1`) can be deduced from the properties of the `GET` method in the HTTP specification. These properties do not apply to `POST`, which is why the previous skeleton could not make this assumption. The current skeleton is still incomplete, since it does not have a precondition. More specifically, we need to obtain the request URI somehow. Given the sequence of the requests in Listing 6, the process can detect that the concrete instances of `_:object1` (`/images/24/thumbnail` and `/images/25/thumbnail`) were already mentioned in a previous response body. Hence, it can place the pattern containing those instances in the antecedent and connect the components that had identical values with the same variable names:

```
{ ?object2 ex:smallThumbnail ?object1. }
=>
{
  _:request http:methodName "GET";
            http:requestURI ?object1;
            http:resp [ http:body ?object1 ].
  ?object2 dbpedia-owl:thumbnail ?object1.
  ?object1 a dbpedia:Image;
           dbpedia-owl:height 80.0.
}.
```

The user can now optionally rename the variable placeholders to obtain the exact same description as in Listing 1.

This shows how descriptions such as Listings 1 and 2 can be generated. Note that this process is not fully automated, but human-assisted. That is, it requires a repeated sequence of human steps as input in order to sufficiently cluster and generalize descriptions. Furthermore, hints and corrections from users might be necessary, as over- or under-generalizations will occur inevitably in some cases. Yet, such an assisted process significantly decreases the burden of full manual description. The fact the RESTdesc descriptions focus on REST APIs facilitates the process: it can make additional assumptions on the behavior of the uniform interface, and hyperlinks from previous responses can be reused.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Most important for this assisted generation of descriptions is the close relationship between N3 and RDF. As we have shown above, the RDF triples in the Web API's responses serve as a direct prototype for the consequent of the generated skeletons. For example, the triple `</images/24> dbpedia-owl:smallThumbnail </images/24/thumbnail>.` directly leads to the inclusion of `_:object2 dbpedia-owl:smallThumbnail _:object4` in the first skeleton. Not only does this simplify the generation process, the connection between the generated description and the original response is also apparent for users, which makes it easier for them as well.

8 Evaluation

8.1 Composition algorithm benchmark

This paper discusses a proof-based method to compose and execute hypermedia APIs. Specific to this method, in contrast to traditional Web service composition methods, is that the composition should be regenerated at each step. This makes the feasibility of the approach depend on whether the composition cost is within reasonable limits. Therefore, an evaluation should assess whether composition happens sufficiently fast for realistic composition lengths and in presence of a realistic number of possible hypermedia APIs that can be used in compositions.

To verify this, we developed a benchmark framework[2] for hypermedia API composition, consisting of two main components:

a hypermedia API description generator, which deterministically generates single- or multi-connected chains of example hypermedia API descriptions with a chosen length, specifically tailored to enable compositions;

an automated benchmarker, testing how well a reasoner performs on creating proofs for compositions of varying lengths and complexity.

Below is one of the example descriptions, generated by the tool with parameters 3 (description chain length) and 2 (number of needed connections per description).

```
{
  ?a1 ex:rel2 ?b1.                # Generated input conditions
  ?a2 ex:rel2 ?b2.
}
=>
{
  _:request http:methodName "GET"; # Generated HTTP request
            http:requestURI ?a1;
            http:resp [ http:body ?b1 ].
  ?a1 ex:rel3 ?b1.                # Generated output conditions
  ?a2 ex:rel3 ?b2.
}.
```

The goal would be a filter rule of the following form:

[2] Available at `http://github.com/RubenVerborgh/RESTdesc-Composition-Benchmark`.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

```
{
  ?a1 ex:rel3 ?b1.                # Output conditions of the
  ?a2 ex:rel3 ?b2.                # last API operation in the chain
}
=>
{
  ?a1 ex:rel3 ?b1.
  ?a2 ex:rel3 ?b2.
}.
```

Herein, the conditions in the antecedent are only satisfiable by creating a chain from the first description towards the last.

As this example shows, descriptions are structurally identical to RESTdesc descriptions of existing hypermedia APIs and therefore representative examples. Other descriptions will be generated such that the input and output conditions can be matched, so the composition algorithm can form chains (in this example, with links to two previous descriptions).

8.2 Parameters and measurements

The main parameters that determine the difficulty of generating a composition are:

(1) the **number** of API operations in the resulting composition: n
(2) the number of **dependencies** between hypermedia API operations in the composition: d
(3) the **total** number of hypermedia APIs supplied to the reasoner (not all necessarily part of the composition): t

To measure the influence of the first two parameters, we test the generation of a composition with a resulting length of n and with d dependencies between each hypermedia API operation, where n ranges from 2 to 1,024 and d from 1 to 3. These ranges have been chosen such that their upper bounds exceed those of compositions for regular use cases, which typically involve only a few API operations with few interdependencies. The goal is therefore whether performance is acceptable for small values—success on larger values comes as an added bonus.

To test the third parameter t, we will keep n and d fixed at 32 and 1 respectively (*i.e.,* already a large composition) and add a number of *dummy* APIs ($n' = t - n$) that can be composed with the other APIs, but are not needed in the resulting composition. It is important to understand that most real-world scenarios will be a mixture of the above situations: compositions are generally graphs 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 the performance 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 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

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

the actual reasoning time separately. Parsing and reasoning together make up for the total time.

8.3 Results

The benchmark was executed on one 2.4 GHz core of an Intel Xeon processor on Ubuntu Server 12.04. The results are summarized below; full results are available at `http://github.com/RubenVerborgh/RESTdesc-Composition-Benchmark-Results`.

8.3.1 EYE reasoner

Tables 1 and 2 show the benchmark results achieved by the EYE reasoner (Verborgh and De Roo 2015), version 2014-09-30 on SWI-Prolog 6.6.6. The results in the first column teach us that starting the reasoner introduces an overhead of $\approx$ 40 ms. This includes process starting costs, which are highly machine-dependent.

Inspecting Table 1 from left to right, we see the reasoning time increases with the composition length n and remains limited to a few hundred milliseconds in almost all cases. The absolute increase in reasoning time for a higher number of dependencies d never crosses 150 ms for small to medium values of n, but becomes larger for high n. Table 2 shows that the reasoning time hardly increases in presence of dummies.

8.3.2 cwm reasoner

The same experiments have been performed with the *cwm* reasoner (Berners-Lee 2009), whose results are shown in Tables 3 and 4. The *cwm* reasoner is not as strongly performance-optimized as EYE, which is clearly visible in the results. Also, we were only able to test for values of n up to 256, because out-of-memory errors appeared for large values. Despite this fact, we still see acceptable results for small-to-medium-sized compositions. We note a higher start-up time of $\approx$140 ms. Reasoning time increases faster than linearly in n, which is also the case for increasing d. The presence of dummies bothers *cwm* more than EYE, and serious issues start to appear at $n' = 512$.

8.3.3 Analysis

The main cause of the difference in performance between EYE and *cwm* are due to the different reasoning mechanisms. EYE is a backward-chaining reasoner, which starts from the goal and works towards the initial state, whereas *cwm* is forward-chaining, exploring inferences from the initial state onwards until the goal has been reached. This explorative behavior demands more processor time, since all possible paths have to be tried, even those that do not contribute to the composition. This is most apparent in the experiment with dummies: *cwm* tries to use them and eventually finds they are not necessary; EYE will only parse them but never tries to use them in the composition.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

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

#APIs n	4	8	16	32	64	128	256	512	1,024
d=1 **dep.**									
parsing	39 ms	41 ms	45 ms	55 ms	74 ms	114 ms	183 ms	323 ms	597 ms
reasoning	*16* ms	*19 ms*	*33* ms	*21* ms	*43* ms	*48* ms	*108* ms	*268* ms	*903* ms
total	56 ms	61 ms	79 ms	76 ms	118 ms	162 ms	292 ms	591 ms	1,500 ms
d=2 **dep.**									
parsing	39 ms	44 ms	59 ms	61 ms	90 ms	126 ms	212 ms	383 ms	728 ms
reasoning	*22* ms	*60* ms	*22* ms	*56* ms	*45*ms	*113*ms	*311*ms	*1,044* ms	*3,640* ms
total	61 ms	105 ms	82 ms	117 ms	136 ms	239 ms	524 ms	1,427 ms	4,369 ms
d=3 **dep.**									
parsing	41 ms	50 ms	53 ms	73 ms	97 ms	146 ms	250 ms	460 ms	890 ms
reasoning	*29* ms	*18* ms	*39* ms	*48* ms	*105* ms	*263* ms	*787* ms	*2,660* ms	*9,105* ms
total	70 ms	68 ms	93 ms	122 ms	203 ms	409 ms	1,037 ms	3,121 ms	9,995 ms

Table 2. *Even a large number of dummies does not significantly disturb EYE's reasoning time (average times of 50 trials)*

# dummies n'	2,048	4,096	8,192	16,384	32,768	65,536	131,072
n=**32,** d=**1 dep.**							
parsing	1,221 ms	2,405 ms	4,586 ms	8,959 ms	17,700 ms	35,823 ms	74,394 ms
reasoning	*47* ms	*66* ms	*71* ms	*177* ms	*403* ms	*633* ms	*1,042* ms
total	1,268 ms	2,471 ms	4,657 ms	9,136 ms	18,104 ms	36,456 ms	75,437 ms

Table 3. *The cwm reasoner performs worse than EYE, but still creates smaller compositions acceptably fast (average times of 50 trials)*

#APIs n	4	8	16	32	64	128	256
d=1 **dep.**							
parsing	139 ms	155 ms	186 ms	271 ms	566 ms	1,721 ms	6,119 ms
reasoning	*43* ms	*106* ms	*320* ms	*1,149* ms	*4,456* ms	*18,266* ms	*82,856* ms
total	183 ms	262 ms	506 ms	1,421 ms	5,022 ms	19,987 ms	88,975 ms
d=2 **dep.**							
parsing	143 ms	161 ms	201 ms	300 ms	637 ms	1,904 ms	6,509 ms
reasoning	*119* ms	*473* ms	*1,946* ms	*8,016* ms	*31,189* ms	*136,806* ms	*582,810* ms
total	263 ms	634 ms	2,147 ms	8,317 ms	31,827 ms	138,710 ms	589,319 ms
d=3 **dep.**							
parsing	147 ms	168 ms	216 ms	328 ms	701 ms	2,088 ms	7,396 ms
reasoning	*190* ms	*739* ms	*2,843* ms	*10,708* ms	*43,429* ms	*182,949* ms	*776,569* ms
total	337 ms	908 ms	3,059 ms	11,036 ms	44,130 ms	185,037 ms	783,965 ms

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Table 4. *The cwm reasoner does not perform well when the number of dummies increases (average times of 50 trials)*

#dummies n'	16	32	64	128	256	512
n=32, d=1 dep.						
parsing	406 ms	576 ms	1,057 ms	2,548 ms	7,682 ms	26,678 ms
reasoning	*1,107* ms	*1,083* ms	*1,116* ms	*2,173* ms	*12,908* ms	*110,775* ms
total	1,513 ms	1,659 ms	2,173 ms	4,721 ms	20,590 ms	137,453 ms

8.3.4 Discussion

The main question of these experiments is whether the results are generalizable to real-world hypermedia API compositions. On the one hand, we have to investigate the difference between the generated API descriptions and actual hypermedia API descriptions. On the other hand, we have to verify if the resulting reasoning times are acceptable for realistic compositions in a Web-scale environment.

First, the generated descriptions have been tailored to closely mimic actual RESTdesc descriptions. The following characteristics of actual descriptions are also found in the generated ones. They contain a number of pre-conditions, determined by the parameter d, and an equal number of post-conditions. They describe the HTTP request that has to be performed to execute a hypermedia API operation. The parameters of the request are obtained through placeholders from the pre-conditions, so they have to be instantiated.

In contrast, some characteristics are different. All requests are `GET` requests, whereas real-world APIs also use other HTTP verbs. However, this has no impact on the reasoner. Furthermore, all descriptions employ predicates with a shared URI namespace. This is done to ensure that a composition always exists. However, this too has no impact on reasoning or parsing time. Therefore, the generated descriptions simulate real-world descriptions reliably.

Second, we note that only the reasoning times are important, because the parsing results can be cached. The maximum tested composition length n is large compared to what one could expect from realistic compositions. In practice, compositions of only a few API operations will be necessary, yet both reasoners perform acceptably on small to medium composition sizes. Furthermore, EYE is capable of creating compositions of a few hundred API operations in just a few hundred milliseconds. Also, the number of dependencies d of each API will likely be limited, with most calls only depending on a single other operation. Yet, even if this is not the case, EYE can fluently cope with multiple dependencies. The final parameter we have to check is the total number of APIs t, since reasoners should be able to create compositions out of large API repositories. Given that ProgrammableWeb contains 10,000 APIs (Berlind 2013), the fact that EYE merely needs ≈230 ms to create a composition in presence of more than 130,000 dummy APIs, indicates that proof-based composition is a viable strategy for the years to come.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

9 Conclusion

In this paper, we explained a novel solution to automated composition and execution of hypermedia APIs. A crucial part in generating a composition is the ability to determine whether it will satisfy a given goal without any undesired effects. This has led us to the approach of a pragmatic proof, wherein hypermedia API operations are incorporated as inference rules. We distinguish between a pre-execution proof and a post-execution proof, where the former has the additional assumption that all hypermedia API operations will succeed, hence the "pragmatic" label of the method.

We selected an RDF-based method and logic for this task, in order to bridge between existing Web technologies and concepts from logic programming. A benefit of proof-based composition is that it does not require new algorithms and tools, but can be applied with existing semantic Web reasoners. Those reasoners can easily incorporate external sources of knowledge such as ontologies or business rules. Furthermore, the performance of composition generation improves with the evolution of those reasoners. Also, the fact that a third-party tool is used allows independent validation of the composition.

Our approach is a special use case for proofs, which have traditionally been regarded as a part of trust on the semantic Web. While pre-proofs partly contribute to this, they also have the added functionality of generating a composition during that process. It will be interesting to explore other opportunities to exploit the power of proof creation and the mechanisms behind it. This application can serve as an example of how to apply such ideas.

In the past, we have already employed the method in the domain of sensor APIs (Verborgh *et al.* 2012), yet we want to extend the approach to other domains such as multimedia analysis and transcoding (Verborgh *et al.* 2013; Van Lancker *et al.* 2013). In the longterm, we aim at offering the composition method described in this paper as a hypermedia API itself, so it can be used for dynamic mashup and composition generation.

Another interesting path is to explore the limits of the used logic. For instance, it would currently be impossible to express the deletion of resources, even though this is a common operation on the Web and even has a designated HTTP method `DELETE`. We are currently experimenting with capturing explicitly described states inside RESTdesc descriptions to account for these situations.

A crucial part of the proof-based method is that the interaction remains driven by hypermedia. In contrast to traditional approaches, where a plan determines the full interaction, the composition here serves as a guideline to complete the interaction. Until the moment machines are able to autonomously interpret the meaning of following a hyperlink—like we humans can—guiding them through a hypermedia application with descriptions and proofs can be the pragmatic alternative.

Acknowledgements

Ruben Verborgh is a Postdoctoral Fellow of the Research Foundation Flanders. The research activities were funded by Ghent University, the Institute for the

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Promotion of Innovation by Science and Technology in Flanders (IWT), and the European Union.

References

Abiteboul, S., Hull, R. and Vianu, V., Eds. 1995. *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Alarcón, R. and Wilde, E. 2010. RESTler: Crawling RESTful services. In *Proc. of the 19th International Conference on World Wide Web*, M. Rappa, P. Jones, J. Freire, S. Chakrabarti, Eds. ACM, Raleigh, North Carolina, USA, 1051–1052.

Alarcón, R., Wilde, E. and Bellido, J. 2011. Hypermedia-driven RESTful service composition. In *Service-Oriented Computing*, E. M. Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig and M. Fantinato Eds. Lecture Notes in Computer Science, Vol. 6568. Springer, Berlin Heidelberg, 111–120.

Angele, J., Boley, H., de Bruijn, J., Fensel, D., Hitzler, P., Kifer, M., Krummenacher, R., Lausen, H., Polleres, A. and Studer, R. 2005. Web rule language (WRL). *W3C Member Submission 9*. `https://www.w3.org/Submission/WRL/` [Accessed on 21/01/2016].

Arndt, D., Verborgh, R., De Roo, J., Sun, H., Mannens, E. and Van De Walle, R. 2015. Semantics of Notation3 logic: A solution for implicit quantification. In *Rule Technologies: Foundations, Tools, and Applications*, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke and D. Roman, Eds. Springer, Berlin Heidelberg, 127–143.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. and Ives, Z. 2007. DBpedia: A nucleus for a Web of open data. In *The Semantic Web*, K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, P. Cudrè-Mauroux, Eds. Lecture Notes in Computer Science, Vol. 4825. Springer, Berlin Heidelberg, 722–735. URL: *http://dx.doi.org/10.1007/978-3-540-76298-0_52* [Accessed on 21/01/2016].

Baget, J.-F., Leclère, M., Mugnier, M.-L., Rocher, S. and Sipieter, C. 2015. Graal: A toolkit for query answering with existential rules. In *Rule Technologies: Foundations, Tools, and Applications*, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke and D. Roman, Eds. Springer, Berlin Heidelberg, 328–344.

Baget, J.-F., Leclère, M., Mugnier, M.-L. and Salvat, E. 2011. On rules with existential variables: Walking the decidability line. *Artificial Intelligence 175,* 9–10, 1620–1654.

Beckett, D., Berners-Lee, T., Prud'hommeaux, E. and Carothers, G. 2013. Turtle – terse RDF triple language. W3C Candidate Recommendation. URL: *http://www.w3.org/TR/turtle/* [Accessed on 21/01/2016].

Berlind, D. 2013. ProgrammableWeb's directory hits 10,000 APIs. And counting. ProgrammableWeb blog. URL: `http://blog.programmableweb.com/2013/09/23/programmablewebs-directory-hits-10000-apis-and-counting/` [Accessed on 21/01/2016].

Berners-Lee, T. 2000. Semantic Web Application Platform. URL: *http://www.w3.org/2000/10/swap/* [Accessed on 21/01/2016].

Berners-Lee, T. 2000–2009. URL: *cwm. http://www.w3.org/2000/10/swap/doc/cwm.html* [Accessed on 21/01/2016].

Berners-Lee, T., Cailliau, R. and Groff, J.-F. 1992. The world-wide web. *Computer Networks and ISDN Systems 25,* 4–5, 454–459.

Berners-Lee, T. and Connolly, D. 2011. Notation3 (N3): A readable RDF syntax. W3C Team Submission. URL: *http://www.w3.org/TeamSubmission/n3/* [Accessed on 21/01/2016].

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y. and Hendler, J. 2008. N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming 8,* 3, 249–269.

Berners-Lee, T., Hendler, J. and Lassila, O. 2001. The semantic web. *Scientific American 284,* 5, 34.

Bishop, B. and Fischer, F. 2008. IRIS-integrated rule inference system. In *Proc. of International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008).*

Bizer, C., Jentzsch, A. and Cyganiak, R. 2011. State of the LOD cloud. URL: *http://www4.wiwiss.fu-berlin.de/lodcloud/state* [Accessed on 21/01/2016].

Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U. and Smith, M. 2012. Owl 2 web ontology language. W3C Recommendation. URL: `http://www.w3.org/TR/owl2-syntax/` [Accessed on 21/01/2016].

Brickley, D. and Guha, R. V. 2004. RDF vocabulary description language 1.0: RDF schema. W3C Recommendation. URL: `http://www.w3.org/TR/rdf-schema/` [Accessed on 21/01/2016].

Calì, A., Gottlob, G., Lukasiewicz, T. and Pieris, A. 2011. Datalog+/-: A family of languages for ontology querying. In *Datalog Reloaded*. O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Berlin Heidelberg: Springer, 351–368.

Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A. and Wilkinson, K. 2004. Jena: Implementing the semantic web recommendations. In *Proc. of the 13th International World Wide Web Conference*, S. Feldman, M. Uretsky, M. Najork, C. Wills, Eds. ACM, New York, NY, USA, 74–83. URL: `www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf` [Accessed on 21/01/2016].

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. 2001. Web services description language (WSDL). W3C Note. URL: `http://www.w3.org/TR/wsdl` [Accessed on 21/01/2016].

Clocksin, W. F. and Mellish, C. S. 1994. *Programming in PROLOG*. Springer, Berlin Heidelberg.

De Roo, J. 1999–2014. Euler proof mechanism. URL: `http://eulersharp.sourceforge.net/` [Accessed on 21/01/2016].

Duerst, M. and Suignard, M. 2005. Internationalized resource identifiers (IRIs). URL: `http://www.ietf.org/rfc/rfc3987.txt` [Accessed on 21/01/2016].

Fielding, R. T. 2008. REST APIs must be hypertext-driven. Untangled – Musings of Roy T. Fielding. URL: `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven` [Accessed on 21/01/2016].

Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. 1999. Hypertext transfer protocol – HTTP/1.1. URL: `http://www.ietf.org/rfc/rfc2616.txt` [Accessed on 21/01/2016].

Fielding, R. T. and Taylor, R. N. 2002. Principled design of the modern Web architecture. *Transactions on Internet Technology 2,* 2 (May), 115–150.

Gomadam, K., Ranabahu, A. and Sheth, A. 2010. SA-REST: Semantic annotation of web resources. W3C Member Submission. URL: `http://www.w3.org/Submission/SA-REST/` [Accessed on 21/01/2016].

Gottlob, G., Orsi, G. and Pieris, A. 2014. Query rewriting and optimization for ontological databases. *ACM Transactions on Database Systems (TODS) 39,* 3, 25.

Hayes, P. J. and Patel-Schneider, P. F. 2014. RDF 1.1 Semantics. W3C Recommendation. URL: `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/` [Accessed on 21/01/2016].

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B. and Dean, M. 2004. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission. URL: http://www.w3.org/Submission/SWRL/ [Accessed on 21/01/2016].

Kifer, M. 2008. Rule interchange format: The framework. In *Web Reasoning and Rule Systems*. D. Calvanese and G. Lausen, Eds. Springer, Berlin Heidelberg, 1–11.

Kifer, M., Lausen, G. and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM (JACM) 42,* 4, 741–843.

Klyne, G. and Carrol, J. J. 2004. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ [Accessed on 21/01/2016].

Koch, J., Valesco, C. A. and Ackermann, P. 2011. HTTP vocabulary in RDF 1.0. W3C Working Draft. URL: http://www.w3.org/TR/HTTP-in-RDF10/ [Accessed on 21/01/2016].

Kopecký, J., Gomadam, K. and Vitvar, T. 2008. hRESTS: An HTML microformat for describing RESTful Web services. In *Proc. of the International Conference on Web Intelligence and Intelligent Agent Technology*, IEEE Computer Society, Washington, DC, USA, 619–625.

Kopecký, J. and Vitvar, T. 2008. MicroWSMO. WSMO Working Draft. URL: http://www.wsmo.org/TR/d38/v0.1/ [Accessed on 21/01/2016].

Kopecký, J., Vitvar, T., Bournez, C. and Farrell, J. 2007. Semantic annotations for WSDL and XML schema. *IEEE Internet Computing 11*, 60–67.

Lanthaler, M. and Gütl, C. 2013. Hydra: A vocabulary for hypermedia-driven Web APIs. In *Proc. of the 6th Workshop on Linked Data on the Web*, C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas and S. Auer, Eds. 6th Workshop on Linked Data on the Web. Rio de Janeiro, Brazil.

Lausen, H., Polleres, A. and Roman, D. 2005. Web service modeling ontology (WSMO). W3C Member Submission. URL: http://www.w3.org/Submission/WSMO/ [Accessed on 21/01/2016].

Lloyd, J. W. and Topor, R. W. 1984. Making Prolog more expressive. *The Journal of Logic Programming 1,* 3, 225–240.

Maleshkova, M., Kopecký, J. and Pedrinaci, C. 2009. Adapting SAWSDL for semantic annotations of RESTful services. In *Proc. of the On the Move to Meaningful Internet Systems Workshops*, R. Meersman, P. Herrero and T. Dillon, Eds. Lecture Notes in Computer Science, Vol. 5872. Springer, Berlin Heidelberg, 917–926.

Manna, Z. and Waldinger, R. 1980. A deductive approach to program synthesis. *Transactions on Programming Languages and Systems (TOPLAS) 2,* 1, 90–121.

Martin, D., Burstein, M., Hobbs, J. and Lassila, O. 2004. OWL-S: Semantic markup for web services. W3C Member Submission. URL: http://www.w3.org/Submission/OWL-S/ [Accessed on 21/01/2016].

Milanovic, N. and Malek, M. 2004. Current solutions for Web service composition. *Internet Computing, IEEE 8,* 6, 51–59.

Mugnier, M.-L. 2011. Ontological query answering with existential rules. In *Web Reasoning and Rule Systems*, S. Rudolph and C. Gutierrez, Eds. Springer, Berlin Heidelberg, 2–23.

Norton, B. and Krummenacher, R. 2010. Consuming dynamic linked data. In *Proc. of the 1st International Workshop on Consuming Linked (COLD2010) Workshop at the 9th International Semantic Web Conference (ISWC2010) – ISWC 2010 Workshops*, O. Hartig, A. Harth and J. Sequeda, Eds. Shanghai, China, vol. II.

Ordóñez, L., Suarez, L. J., Rojas, L. A. and Corrales, J. C. 2012. Towards the automation of the semantic annotation process for Web services. In *Proc. of the 2012 Conference on Semantic Web & Web Services.*

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press

Parsia, B. and Sirin, E. 2004. Pellet: An OWL DL reasoner. In *Proc. of the 3rd International Semantic Web Conference*, V. Haarslev, R. Möller, Eds.

Pautasso, C. and Wilde, E. 2009. Why is the Web loosely coupled? – A multi-faceted metric for service design. In *Proc. of the 18th International Conference on World Wide Web*, J. Quemada, G. León, Y. Maarek and W. Nejdl, Eds. ACM, New York, 911–920.

Richardson, L., Amundsen, M. and Ruby, S. 2013. *RESTful Web APIs*. O'Reilly, Sebastopol, CA.

Speiser, S. and Harth, A. 2011. Integrating linked data and services with linked data services. In *The Semantic Web: Research and Applications*, G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, and J. Pan, Eds. Lecture Notes in Computer Science, Vol. 6643. Springer, 170–184.

Stadtmüller, S., Speiser, S., Harth, A. and Studer, R. 2013. Data-Fu: A language and an interpreter for interaction with read/write linked data. In *Proc. of the World Wide Web Conference*, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1225–1236.

Van Lancker, W., Van Deursen, D., Verborgh, R. and Van de Walle, R. 2013. Semantic media decision taking using N3Logic. *Multimedia Tools and Applications 63,* 1 (Mar.), 7–26.

Verborgh, R. and De Roo, J. 2015. Drawing conclusions from linked data on the web. *IEEE Software 32,* 5 (May), 23–27.

Verborgh, R., Haerinck, V., Steiner, T., Van Deursen, D., Van Hoecke, S., De Roo, J., Van de Walle, R. and Gabarró Vallés, J. 2012. Functional composition of sensor Web APIs. In *Proc. of the 5th International Workshop on Semantic Sensor Networks*, R. Alarcon, C. Pautasso and E. Wilde, Eds. New York, NY, USA.

Verborgh, R., Harth, A., Maleshkova, M., Stadtmüller, S., Steiner, T., Taheriyan, M. and Van de Walle, R. 2014. Survey of semantic description of REST APIs. In *REST: Advanced Research Topics and Practical Applications*, C. Pautasso, E. Wilde, and R. Alarcón, Eds. Springer, Berlin Heidelberg, 69–89.

Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró Vallés, J. and Van de Walle, R. 2012. Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In *Proc. of the 3rd International Workshop on RESTful Design*, C. Henson, K. Taylor, O. Corcho, A. Sheth, and M. Hauswirth, Eds. ACM, 33–40.

Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R. and Gabarró Vallés, J. 2013. Capturing the functionality of Web services with functional descriptions. *Multimedia Tools and Applications 64*, 365–387.

Verborgh, R., van Hooland, S., Cope, A. S., Chan, S., Mannens, E. and Van de Walle, R. 2015. The fallacy of the multi-API culture: Conceptual and practical benefits of representational state transfer (REST). *Journal of Documentation 71,* 2 (Mar.), 233–252.

Waldinger, R. 2001. Web agents cooperating deductively. In *Formal Approaches to Agent-Based Systems*. Lecture Notes in Computer Science, Vol. 1871. Springer, 250–262.

https://doi.org/10.1017/S1471068416000016 Published online by Cambridge University Press