Vers une ontologie des interactions HTTP

Mathieu Lirzin, Béatrice Markhoff

Vers une ontologie des interactions HTTP

Mathieu Lirzin[1,2], Béatrice Markhoff[2]

[1] Néréide, 8 rue des déportés, 37000 Tours, France
mathieu.lirzin@nereide.fr

[2] LIFAT EA 6300, Université de Tours, Tours, France
beatrice.markhoff@univ-tours.fr

Résumé : Les systèmes d'information d'entreprise ont adopté les bases du Web pour les échanges entre programmes hétérogènes. Ces programmes fournissent et consomment via des APIs Web des ressources identifiées par des URIs, dont les représentations sont transmises via HTTP. Par ailleurs HTTP reste au coeur de tous les développements du Web (Web sémantique, données liées, IoT...). Ainsi les situations où un programme doit pouvoir raisonner sur des interactions (requête-réponse) HTTP se multiplient. Cela suppose de disposer d'une spécification formelle explicite d'une conceptualisation partagée de ce qu'est une interaction HTTP. Une proposition de vocabulaire RDF existe, développée dans l'optique de réaliser des tests de conformité d'applications Web et enregistrer les résultats de ces tests. Ce vocabulaire a déjà été réutilisé pour d'autres applications. Dans cet article nous décrivons comment nous l'adaptons pour les besoins de notre application, dans l'optique que cette adaptation serve plus largement.

Mots-clés : HTTP, interaction HTTP, logique de description, ontologie, RDF, OWL

1 Introduction

Le Web repose sur trois standards : l'adressage par liens hypermédia (URI), le langage de balisage HTML et le protocole HTTP [1]. Le protocole HTTP est également utilisé dans les systèmes d'information d'entreprises, en particulier pour les échanges entre programmes hétérogènes : il suffit de mettre en oeuvre un serveur HTTP d'un côté et un client HTTP de l'autre. Malgré sa simplicité apparente ce protocole permet de gérer tous les aspects des communications client-serveur tout en maintenant sa capacité d'évolution pour prendre en compte de nouveaux aspects. Cette large couverture se reflète dans le volume de la spécification de HTTP 1.1, qui comprend 8 RFC de l'IETF, eux-mêmes précisés et étendus par d'autres RFC. Cette spécification doit être respectée non seulement par les concepteurs de serveurs et de navigateurs Web mais aussi par tout développeur de service Web et plus généralement encore tout développeur d'application qui doit utiliser un service accessible par HTTP. L'objectif de la proposition présentée dans cet article est de formaliser la spécification du protocole HTTP pour pouvoir décrire des *interactions* qui le respectent et, plus précisément, pouvoir vérifier automatiquement le respect du protocole. Depuis que le Web existe plusieurs propositions ont été élaborées pour décrire des interactions entre clients et serveurs Web, avec divers moyens et objectifs. La plupart sont exploitables par des programmes, mais *seulement à un niveau syntaxique*. Or *nous avons besoin d'une ontologie des interactions HTTP pour développer une validation d'API Web par rapport à une spécification formelle de ses fonctionnalités*. La description ontologique d'interactions HTTP se heurte à une difficulté importante, celle de représenter les liens hypermédia et leurs usages, à la fois comme représentants (identifiants) de ressources et comme moyens d'accès à des représentations (ou descriptions) de ces ressources, voire à la ressource elle-même (si document Web). Ainsi les usages des URIs dans HTTP relèvent à la fois des données et du contrôle sur ces données. Cet aspect est puissant en matière d'expressivité pour les développeurs qui écrivent des programmes qui interagissent à travers ces données et HTTP. Mais il est ardu de le décrire formellement et il demeure également difficile d'écrire des programmes qui reposent sur HTTP qui soient fiables et robustes face aux évolutions.

1. Standards maintenus par l'IETF (https ://tools.ietf.org) : respectivement, rfc7320, rfc2854 et rfc7230.

Notre proposition dans cet article est d'une part de commencer à formaliser la description d'interactions en HTTP en utilisant une logique de description pour pouvoir valider cette formalisation via un raisonneur, et d'autre part d'apporter des éléments de réponse quant à la difficulté qui vient d'être évoquée. Nous synthétisons ainsi nos contributions :

— Nous utilisons la logique de description $\mathcal{SROIQ}^{(\mathcal{D})}$ pour décrire nos propositions, lesquelles sont construites sur les bases d'un vocabulaire initié dans le cadre du W3C et destiné à représenter HTTP en RDF [2] (Koch *et al.*, 2017a), vocabulaire que par la suite nous nommons HTTPinRDF. La logique de description nous permet de caractériser précisément les classes et les propriétés définies dans ce vocabulaire et d'introduire de façon cohérente celles que nous proposons.

— Les en-têtes d'un message HTTP ont la forme d'une table d'associations (nom, valeur), avec des valeurs dont les types sont hétérogènes. Nous en proposons (section 4.5) une représentation générique, tout en montrant comment en avoir aussi une représentation spécifique plus précise, par exemple pour l'élément d'en-tête standard `Location` dont la valeur est un URI. Cela précise et simplifie la représentation adoptée par le vocabulaire W3C RDF HTTP, où toutes les valeurs sont des littéraux.

— Le contenu du corps d'un message HTTP peut avoir différent formats, ce qui est précisé à l'aide de déclarations dites *types de media*. Ce mécanisme générique offre une grande souplesse en ce qui concerne les types de données échangées, mais il rend délicat la représentation formelle de ces données. Nous proposons (section 4.6) de représenter en RDF à la fois le message (requête ou réponse), ses métadonnées et les données contenues dans son corps, pour un sous-ensemble des types de contenus autorisés par HTTP. Cela rend possible d'exploiter chacun de ces constituants avec les outils du Web sémantique (que ce soit SPARQL, SHACL, STTL, ...).

— Un URI est un identifiant qui respecte une syntaxe bien définie, il identifie une ressource, il permet d'accéder à des représentations de cette ressource, si cette ressource est un service Web il permet de lui communiquer des valeurs de paramètres, etc. Précisément dans le contexte d'une requête HTTP, une partie optionnelle de l'URI est très fréquemment utilisée pour paramétrer le comportement de la procédure responsable du traitement des requêtes côté serveur. Nous proposons (section 4.3) une représentation des différentes parties syntaxiques d'un URI et en particulier de la série de (clé, valeur) représentant ses paramètres, lorsqu'ils existent. Cela permet des références explicites à ces différents composants, ce qui est utile par exemple pour vérifier leur présence.

— Nos propositions sont implémentées [3] avec Protégé en OWL 2 DL, qui correspond à la logique de description $\mathcal{SROIQ}^{(\mathcal{D})}$ où à la fois classes et individus sont identifiés par des URI. Nous utilisons le raisonneur HermiT pour valider la satisfaisabilité de l'ontologie résultante et sa consistence. Nous fournissons également un ensemble d'instances qui représentent des cas concrets d'interaction HTTP, afin de vérifier les requêtes SPARQL qui permettent de répondre aux *Questions de Compétence* (notées **QC**) qui définissent nos besoins en matière de représentation d'interaction HTTP.

Dans la section 2 nous introduisons le protocole HTTP en même temps que sa représentation dans HTTPinRDF. Nous précisons les contours de nos propositions en décrivant les limites de ce vocabulaire par rapport à nos objectifs en section 3. Nous présentons nos propositions en section 4 et leur évaluation en section 5, puis nous les situons dans l'état de l'art concernant la description d'interactions HTTP en section 6, avant de conclure en section 7.

2 Notions préliminaires

HTTP est un standard géré par l'*Internet Engineering Task Force* (IETF). Partant des premières proposition de Tim Berners-Lee, un moyen d'indiquer les formats des données transmises y a rapidement été introduit (en-têtes MIME), puis ont été ajoutées des possibilités de préciser de nombreuses caractéristiques liées à l'efficacité des communications (connexions

2. `https://www.w3.org/TR/HTTP-in-RDF10/`

3. `https://labs.nereide.fr/mthl/http`

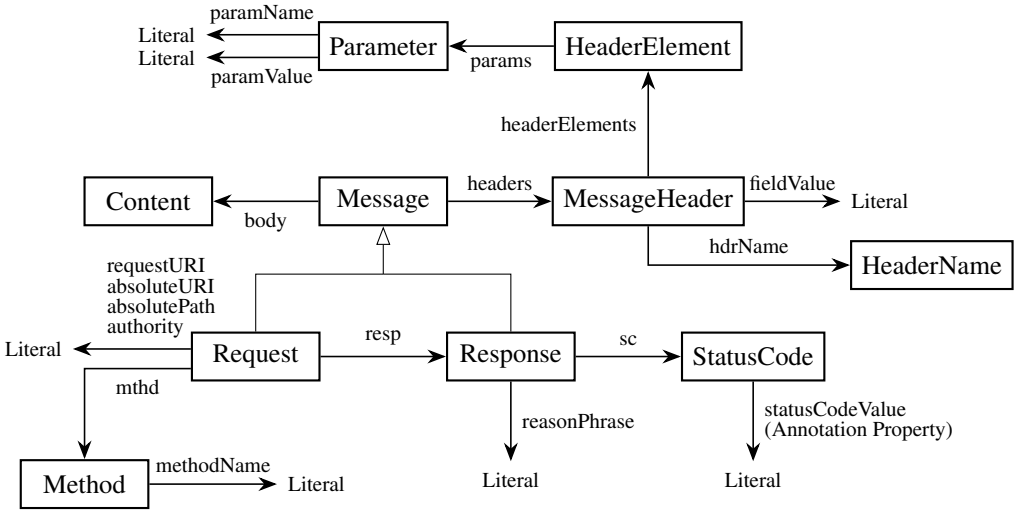

FIGURE *1 – Principaux éléments de HTTPinRDF.*

persistantes, mécanismes de cache) et à leur sécurité (HTTPS, chiffrement). Tout ceci résulte en un pouvoir d'exprimer des informations riches et variées sur les interactions client-serveur. La dernière spécification de HTTP 1.1, produite en 2014, est déclinée en 8 documents appelés RFC, dont les principaux sont la RFC 7230 où sont définis la syntaxe des messages et les aspects du protocole qui concernent le routage, et la RFC 7231 où sont décrits la sémantique des message et leurs contenus. Malgré toutes les précisions apportées dans ces divers documents et les dernières extensions décrites dans ceux associés à HTTP 2 [4], ce standard sur lequel reposent la grande majorité des applications actuelles est volontairement maintenu ouvert pour laisser de la place à d'autres innovations.

La connaissance des RFC de l'IETF est nécessaire pour tout développeur, aussi nous pensons que des outils intelligents devraient exister pour guider les développements qui doivent les respecter. Pour que de tels programmes puissent exploiter non seulement les règles syntaxiques mais aussi la sémantique des interactions HTTP il faut qu'ils disposent d'une ontologie de ces interactions. Un vocabulaire RDF dédié au protocole HTTP (Koch *et al.*, 2017a) a été développé il y a quelques années, dans le cadre d'un outil d'évaluation de l'accessibilité des applications Web. Les interactions de clients Web avec l'application Web testée doivent être enregistrées pour permettre l'évaluation de son accessibilité. Ce vocabulaire RDF fournit la structure de tels enregistrements et pour cela il décrit essentiellement les en-têtes des messages HTTP échangés. Il est utilisé dans EARL, un autre format élaboré dans le même cadre et dédié à l'expression des résultats des évaluations. HTTPinRDF répond aux besoins de ce projet sans pour autant prétendre représenter toute interaction possible via HTTP : il a donc été laissé dans l'état ouvert de *W3C Working Group Note* en 2017.

Cette proposition peut être considérée comme une ontologie d'application (Guarino, 1998), dont le but est de représenter des spécificités propres aux interactions HTTP sur lesquelles reposent les API Web, tandis qu'une ontologie de domaine à laquelle la relier va permettre par exemple de décrire plus généralement un problème, un algorithme ou une fonction, qu'ils soient implémentés par une API Web ou autrement. La figure 1 donne une représentation visuelle des principales classes et propriétés de HTTPinRDF (Koch *et al.*, 2017a). Ce vocabulaire comprend 14 classes et 25 propriétés dans l'espace de noms `http`, plus 11 classes et de nombreuses propriétés (dont celles du vocabulaire Dublin Core DCterm) définies dans quatre autres espaces de noms dédiés respectivement à la représentation des contenus, des en-têtes, des méthodes des message qui sont des requêtes et des codes de statut des messages qui sont des réponses. Ces différents espaces de noms sont présentés dans la table 1.

Le protocole HTTP spécifie une interaction sous la forme d'un échange de messages re-

4. `https://tools.ietf.org/html/rfc7540`

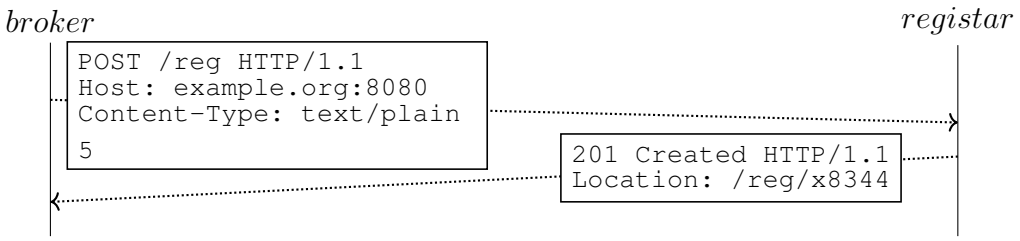

FIGURE *2 – Interaction entre un client "broker" et un serveur "registar".*

TABLE *1 – Préfixes utilisés en Figure 3.*

Prefix	**Namespace**
http :	`http://www.w3.org/2011/http#`
mthd :	`https://www.w3.org/2011/http-methods#`
hdr :	`http://www.w3.org/2011/http-headers#`
sc :	`http://www.w3.org/2011/http-statutCodes#`
cnt :	`http://www.w3.org/2011/content#`
dct :	`http://purl.org/dc/terms/`

quête/réponse : un client envoie un message qui est une requête à un serveur et ce dernier retourne au client un message qui est une réponse. Du point de vue architectural les choses sont un peu plus complexes, il y a des intermédiaires (proxy/passerelles) par lesquels passent les messages, qui peuvent les rediriger, les garder en cache, etc. Mais du point de vue de l'agent client devant utiliser le service offert par le serveur, ces détails n'ont pas d'impact sur le modèle qui lui est utile, celui de l'échange de messages requête/réponse. Comme l'illustre la figure 1, HTTPinRDF représente une interaction HTTP par une relation `resp` depuis un message requête vers un message réponse. Que ce soit une requête ou une réponse, un message a un corps et des en-têtes, lequels forment un ensemble d'associations (nom, valeur). Un message requête est caractérisé en plus par un URI et une méthode, tandis qu'à un message réponse est associé un code de statut. Dans HTTPinRDF, l'URI de la requête est représenté avec un ensemble de propriétés qui toutes ont pour objet un littéral. La méthode de la requête est représentée par une classe. Le code de statut de la réponse est représenté par un littéral et une classe. Pour montrer comment s'utilise ce vocabulaire, nous donnons en figure 2 un exemple d'interaction HTTP. Un agent client nommé *broker* envoie à un serveur nommé *registar* une requête pour que lui soient attribués cinq identifiants. La requête contient le nombre d'identifiants voulus et la réponse comprend le code de statut 201 qui ici dénote la création d'une nouvelle ressource correspondant à la collection d'identifiants ainsi enregistrés. Nous montrons en figure 3 la représentation en RDF de cette interaction (en Turtle), respectant HTTPinRDF. Les définitions des préfixes sont dans la table 1. Le graphe RDF consiste en une paire d'instances des classes `http:Request` et `http:Response`, reliées par la propriété `http:resp` property. Dans cet exemple la propriété utilisée pour représenter l'URI de la requête est `http:absolutePath`, sous-propriété de `http:requestURI`. Pour l'en-tête nous utilisons les propriétés `hdrName` / `fieldValue` parce que ses éléments sont des champs d'en-tête prédéfinis par l'IETF, mais le vocabulaire permet aussi de décrire n'importe quelle paire (nom, valeur) en utilisant la classe `Parameter`.

Un point important de HTTPinRDF est qu'il réifie les méthodes, en-têtes et codes de statut et crée des instances pour tous ceux qui sont définis précisément dans les RFC, tout en les représentant aussi par des littéraux (chaînes de caractères), solution utilisable pour les en-têtes et codes de statut non standards (propres à une application). C'est un moyen de respecter le caractère d'extensibilité du protocole HTTP. Dans l'exemple en Figure 2, nous n'utilisons que des méthodes, en-têtes, et code de statut standards, c'est pourquoi ces éléments sont identifiés par un URI et non un littéral.

```
:req a http:Request ;
   http:mthd mthd:POST ;
   http:absolutePath "/reg" ;
   http:headers [
       http:hdrName hdr:Host ;
       http:fieldValue "example.org:8080"
       ] , [
       http:hdrName hdr:ContentType ;
       http:fieldValue "text/plain"
       ];
   http:resp :resp ;
   http:body [
       a cnt:ContentAsBase64 ;
       dct:isFormatOf [
           a cnt:ContentAsText ;
           cnt:chars "5"
           ]
       ] .

:resp a http:Response ;
   http:sc sc:Created ;
   http:headers [
       http:hdrName hdr:Location ;
       http:fieldValue "/reg/x8344"
       ] .
```

FIGURE *3 – Représentation en Turtle selon HTTPinRDF de l'exemple en Figure 2.*

Une interaction, instance de la propriété `http:resp`, est caractérisée par le code de statut de la réponse. Avec HTTPinRDF il faut identifier la classe associée au numéro de ce code de statut. Un numéro de code de statut doit avoir trois chiffres et sa classe est déterminée par son premier chiffre de la façon suivante :

Classe	Informational	Successful	Redirection	Client Error	Server Error
Codes de statut	$[\![100, 199]\!]$	$[\![200, 299]\!]$	$[\![300, 399]\!]$	$[\![400, 499]\!]$	$[\![500, 599]\!]$

Ces classes sont définies dans l'espace de noms associé au préfixe `sc`. Chaque code de statut est une instance de la classe correspondante. Dans notre exemple, le code de statut est 201, il est représenté par l'individu `sc:Created`, de la classe `sc:Successful`.

Il est important d'observer enfin comment est représenté le corps du message, qui contient le contenu communiqué. Le protocole HTTP permet l'usage de multiple formats pour une même ressource. Un format est identifié par un type de media (Media-Type) dans l'élément d'en-tête `Content-Type`. Dans notre exemple la requête contient le littéral « 5 » avec le Media-Type `text/plain`. La classe `cnt:Content` est un moyen d'associer plusieurs représentations à un même contenu. La propriété `http:body` est toujours présente avec pour valeur `cnt:ContentAsBase64` mais peut aussi être utilisée avec la propriété `dct:hasFormat` pour un contenu `cnt:ContentAsText`.

3 Présentation du problème

D'autres travaux se sont appuyés sur HTTPinRDF, par exemple dans (Verborgh *et al.*, 2017) les auteurs l'utilisent pour définir RESTdesc, un cadre de spécification d'API Web hypermédia (et de composition automatique). « API Web hypermédia » fait ici référence aux API Web qui suivent les principes du style architectural *Representational State Transfer*

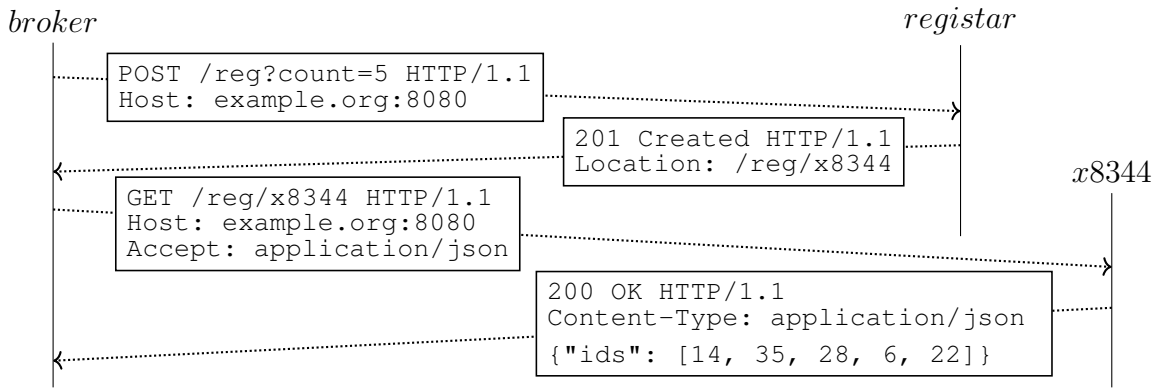

FIGURE *4 – Conversation hypermédia entre un client "broker" et un serveur "registar"*

(REST) [5] (Fielding & Taylor, 2002), et plus particulièrement le principe appelé HATEOAS, qui consiste à utiliser les liens hypermédia pour le contrôle de l'état de l'application client, en d'autres termes pour lui indiquer les continuations possibles des traitements. Nous souhaitons concevoir une vérification automatique de la conformité d'une telle API aux besoins d'une application cliente ou, inversement, de la conformité de l'utilisation de l'API Web par une application cliente. Pour cela nous avons besoin de représenter sous la forme d'un graphe RDF une interaction HTTP dans laquelle le client est amené à utiliser dynamiquement les liens hypermédia fournis par le serveur pour découvrir les actions et ressources possibles pour poursuivre. Or HTTPinRDF ne permet pas de représenter cela.

Pour s'en convaincre, prenons l'exemple de la figure 4, plus proche d'une application réelle que celui de l'exemple de la figure 2. Ici nous avons une conversation composée de deux interactions HTTP, c'est-à-dire deux paires requête/réponse. La première est semblable à l'exemple en Figure 2 à ceci près que le nombre voulu d'identifiants est passé dans une partie de l'URI de la première requête (POST), avec le paramètre de requête `count`. C'est une façon de faire très courante car compatible avec le traitement des formulaires par les navigateurs. La deuxième interaction consiste à interroger le lien fourni par le serveur via l'élément `Location` de l'en-tête de la réponse de la première interaction. Le déréférencement de ce lien fournit une réponse dont le contenu du corps est en format JSON, avec une structure plus complexe que du simple texte (ce à quoi se limitait l'exemple en Figure 2). Il n'est pas possible de représenter la sémantique de cet exemple avec un graphe RDF limité au vocabulaire HTTPinRDF car il ne permet pas de décrire la partie requête d'un URI, ni d'identifier un URI servant dans une interaction suivante, qu'il soit dans un en-tête ou dans un corps de message.

Un autre point qui nécessite de réviser HTTPinRDF est que ses auteurs font référence à la RFC 2616, qui a été supplantée par d'autres documents et en particulier par la RFC 7231, dédiée à la sémantique de HTTP 1.1, laquelle est préservée pour l'essentiel dans les nouvelles versions du protocole. Les changements que nous présentons dans la section suivante sont destinés à prendre en compte la RFC 7231 pour représenter des enchaînements d'interactions HTTP, où le client utilise les liens hypermédia pour fournir des valeurs de paramètres au serveur (en particulier dans l'URI de ses requêtes) et exploite aussi les liens hypermédia que le serveur lui fournit dans une partie ou une autre de ses réponses. Notre objectif est de pouvoir formuler des règles de validité sur les interactions ainsi représentées. Sans prétendre à l'exhaustivité nous exprimons des éléments de telles règles sous la forme de questions de compétence (QC) afin de pouvoir vérifier l'utilité de nos propositions par rapport à nos objectifs. Le principe des QC est de refléter les besoins fonctionnels qui fondent les engagements ontologiques d'une représentation de connaissances (Noy & Mcguinness, 2001; Blomqvist *et al.*, 2010; Hitzler *et al.*, 2016).

5. Où l'application se définit par les ressources (distantes) qu'elle utilise, tandis qu'avec SOAP elle se définit par les appels à des procédures distantes.

QC 1 (Type de média)
Quel est le type de média associé à un corps de message ?

QC 2 (Résultat d'interaction)
Quel est le numéro du code de statut d'une interaction ?

QC 3 (Valeur d'élément d'en-tête, comme Location)
Quel est l'URI fourni via l'élément `Location` *d'un en-tête de réponse ?*

QC 4 (Résultat de conversation)
Quel est le code de statut associé à la réponse de la dernière interaction d'une conversation donnée ? Par exemple celui de la Figure 4 serait 200.

QC 5 (Négociation de contenu)
Est-ce que le type de média du corps d'une réponse correspond à l'un de ceux déclarés par un élément d'en-tête `Accept` *dans la requête correspondante ?*

QC 6 (Contenu du corps de message quand il est en RDF)
Quelles sont les valeurs d'une propriété RDF donnée p *dans le corps d'un message lui-même en RDF ?*

QC 7 (Paramètres de la partie requête dans un URI)
Quelle est la valeur d'un paramètre donné, par exemple `age`*, passé dans l'URI d'un message requête ?*

4 Contributions à une ontologie des interactions HTTP

Nous utilisons la logique de description $\mathcal{SROIQ}^{(\mathcal{D})}$, associée à OWL2 DL, pour décrire l'ontologie résultant de nos propositions, montrée en Figure 5, qui repose en grande partie sur HTTPinRDF. Nous conservons toutefois dans cet article la terminologie RDF en parlant de classes et propriétés plutôt que de rôles et de concepts. Nous notons $\top$ la classe de toutes les instances et $\mathcal{D}$ la classe des littéraux (qui ne peuvent pas être sujet d'une propriété et ont une interprétation unique). Les classes $\top$ and $\mathcal{D}$ sont disjointes. $\bot$ désigne la classe ne contenant aucun élément.

4.1 Message

Un message peut avoir une collection d'éléments d'en-tête, ce que nous représentons comme dans HTTPinRDF par la propriété $headers$ mais dont les objets ici sont de la classe $Header$, et un corps, représenté par la propriété $body$ qui n'apparaît qu'une seule fois et dont l'objet est de la classe $Content$. En-tête et corps sont optionnels.

$$Message \sqsubseteq \forall headers.Header \sqcap \forall body.Content \qquad \top \sqsubseteq \leq_1 body.\top$$

Les détails des classes $Header$ et $Content$ sont donnés en section 4.5 et section 4.6 respectivement. Un message peut être une requête ou une réponse mais pas les deux à la fois.

$$Message \equiv Request \sqcup Response \qquad Request \sqcap Response \sqsubseteq \bot$$

La propriété $resp$ relie une requête à une réponse.

$$\exists resp.\top \sqsubseteq Request \qquad \top \sqsubseteq \forall resp.Response$$

Il peut y avoir plusieurs réponses pour une même requête, avec les restrictions décrites en section 4.4.

4.2 Requête

Une requête a une méthode et un URI. Nous définissons la classe $Method$ comme superclasse de toutes les méthodes de requête standards en utilisant des *nominaux*. Cela n'est pas une équivalence car de nouvelles méthodes peuvent être définies.

$$\{\text{GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH}\} \sqsubseteq Method$$
$$Request \sqsubseteq Message \sqcap \exists mthd.Method \sqcap \exists uri.URI$$
$$\top \sqsubseteq \leq_1 mthd.Method \qquad \top \sqsubseteq \leq_1 uri.URI$$

Les propriétés $mthd$ et uri sont fonctionnelles. La valeur associée à la propriété uri est une instance de la classe URI, que nous introduisons pour décrire les composants syntaxiques d'un URI tels que définis dans la RFC 3986 (Berners-Lee *et al.*, 2005) dont est tiré l'exemple d'URI complet ci-dessous :

$$\underbrace{\texttt{http}}_{scheme}\texttt{://}\underbrace{\texttt{example.com:8042}}_{authority}\underbrace{\texttt{/over/there}}_{path}\texttt{?}\underbrace{\texttt{name=ferret}}_{query}\texttt{\#}\underbrace{\texttt{nose}}_{fragment}$$

Les syntaxes que peut prendre concrètement l'URI d'une requête sont données dans (Fielding & Reschke, 2014a, Section 5.3). Certaines nécessitent de combiner la valeur de l'élément d'en-tête `Host` et du schéma de protocole (`http` ou `https`) pour obtenir l'URI complet. Les différentes parties de l'URI sont représentées avec les propriétés $scheme$, $authorithy$, $path$, $query$, $fragment$ (cf. exemple ci-dessus) de la classe URI, qui ont des valeurs littérales. Elles servent à décrire toute forme que prend concrètement l'URI. Cette représentation diffère de HTTPinRDF, où différentes propriétés sont définies pour la classe $Request$ (`http:requestURI`, `http:absolutePath` et `http:absoluteURI`) dont les valeurs sont des littéraux. Le littéral qui serait valeur de `http:absoluteURI` avec HTTPinRDF est ici représenté en utilisant ensemble $scheme$, $authorithy$, $path$, éventuellement $query$ et peut-être $fragment$. Nous ajoutons une propriété $idRes$ à la classe URI qui prend comme valeur l'URI représentée, sous forme de chaîne de caractères. Ce littéral est l'URI d'une ressource et l'instance de la classe URI sert à représenter ses différentes formes concrètes possibles. Pour pouvoir répondre à **QC 7** nous associons aussi aux instances de la classe URI une propriété `queryParams` afin de détailler la partie $query$, comme précisé dans la section suivante.

4.3 Paramètres de la partie *query* d'une URI

La description d'une requête HTTP nécessite de représenter des paramètres. Les paramètres peuvent être passés de différentes façons, la plus simple étant d'utiliser la partie $query$ de l'URI de la requête, partie qui se situe entre `?` et `#`. Pour une application Web cette partie se conforme au type de media `application/x-www-form-urlencoded` [6] qui permet de passer des paires (clé,valeur) comme arguments, que nous dénotons (k,v) dans l'exemple suivant.

$$\underbrace{\overbrace{\texttt{age}}^{k}\texttt{=}\overbrace{\texttt{54}}^{v}\texttt{\&}\overbrace{\texttt{id}}^{k}\texttt{=}\overbrace{\texttt{XPZIJ4}}^{v}}_{query}$$

Avec la propriété $query$ nous pouvons accéder à la valeur littérale, mais nous voulons aussi accéder aux paires (clé,valeur), pour cela nous définissons la propriété $queryParams$ et la classe $Parameter$ dont les instances ont un nom (et un seul) et une valeur (et une seule).

$$URI \sqsubseteq \forall queryParams.Parameter \qquad Parameter \equiv \exists name.\mathcal{D} \sqcap \exists value.\mathcal{D}$$

6. `https://url.spec.whatwg.org/#concept-urlencoded`

4.4 Réponse

Une réponse a un statut, instance de la classe $Status$ et accessible par la propriété sc, elle-même étant caractérisée par un nombre à trois chiffres via la propriété $code$.

$$Response \sqsubseteq Message \sqcap \exists sc.Status \qquad Status \sqsubseteq \exists code.[\![000, 999]\!]$$
$$\top \sqsubseteq \leq_1 sc.\top \qquad \mathcal{D} \sqsubseteq \leq_1 code.\mathcal{D}$$

Les propriétés sc et $code$ sont fonctionnelles. Nous utilisons la notation compacte $[\![000, 999]\!]$ pour dénoter le type des entiers positifs inférieurs ou égaux à 999 qui s'écrirait ainsi en OWL 2 turtle :

```
:threeDigit a rdfs:Datatype ;
    owl:equivalentClass [
        a rdfs:Datatype ;
        owl:onDatatype xsd:nonNegativeInteger ;
        owl:withRestrictions ([ xsd:maxInclusive 999 ])] .
```

Il existe une bijection entre une instance de $Status$ et son code. Par exemple nous pouvons définir $Created$ comme étant l'unique instance de $Status$ ayant le code 201 (qui indique *une nouvelle ressource a été créée avec succès*).

$$\{Created\} \equiv Status \sqcap \exists code.201$$

Les codes de statut sont des éléments syntaxiques qui dénotent le sens du statut de réponse. Même si chaque instance de $Statut$ a un sens spécifique, elles peuvent être regroupées en sous-classes de $Status$ de la façon suivante.

$$Successful \equiv Status \sqcap \exists code.[\![200, 299]\!] \quad ClientError \equiv Status \sqcap \exists code.[\![400, 499]\!]$$
$$Redirection \equiv Status \sqcap \exists code.[\![300, 399]\!] \quad ServerError \equiv Status \sqcap \exists code.[\![500, 599]\!]$$
$$Informational \equiv Status \sqcap \exists code.[\![100, 199]\!]$$

Les instances de ces classes qualifient des *réponses finales* à l'exception de celles de la classe $Informational$ qui définissent des *réponses intermédiaires* qui seront suivies d'une *réponse finale* (Fielding & Reschke, 2014b, Section 6.2). Ainsi plusieurs réponses peuvent être associées à une requête mais une seule peut être une *réponse finale*.

$$\exists sc.Informational \sqsubseteq Interim \qquad Final \equiv Response \sqcap \neg Interim$$

Definition 1 (Interaction)
Une interaction *est une instance $resp(q, r)$ de la propriété $resp$ telle que q est une instance de $Request$ et r est une instance de $Final$.*

Dans la figure 5 nous ne représentons pas les sous-classes de $Status$, comme nous ne représentons pas non plus celles de $StatusCode$ en Figure 1. Nous ne conservons pas la propriété $reasonPhrase$ car cette précision syntaxique ne porte pas de sens en elle-même.

4.5 Eléments d'en-tête

Un message peut avoir un ensemble d'éléments d'en-tête. Nous représentons ces éléments par une classe $Header$, ses instances étant caractérisées par un unique nom (propriété $hdrName$) et une unique valeur (propriété $fieldValue$), tous deux littéraux.

$$Header \sqsubseteq \exists hdrName.\mathcal{D} \sqcap \exists fieldValue.\mathcal{D}$$

Cette représentation générique utilisant un littéral pour toute valeur peut être précisée pour les éléments d'en-tête prédéfinis comme `Location` ou `Content-Type`, en définissant une

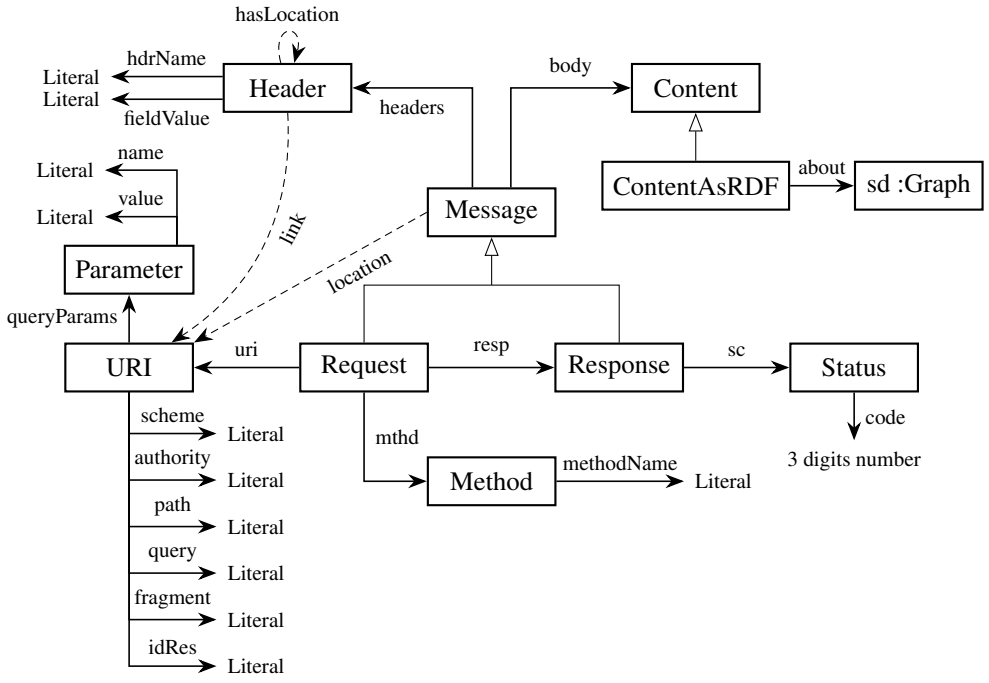

FIGURE *5 – Principaux éléments de notre ontologie des interactions HTTP.*

chaîne de propriétés qui permette de de faire référence à leur valeur directement depuis le message, avec une propriété ayant leur nom. Cela peut grandement simplifier les réponses pour **QC 1** et **QC 3**. Par exemple pour un élément d'en-tête `Location` (c'est-à-dire dont la propriété $hdrName$ a la valeur `"Location"`) nous définissons d'abord une propriété $link$ dont la valeur correspond à une instance de la classe URI représentant la valeur littérale de la propriété $fieldValue$, ce qui rend possible d'isoler des parties de cette valeur littérale si besoin. Puis nous disons qu'une instance de $Header$ ayant pour valeur de $hdrName$ `"Location"` a une propriété $link$.

$$Header \sqcap \exists hdrName.\{\texttt{"Location"}\} \sqsubseteq \exists link.\boldsymbol{URI}$$

De cette manière, d'autres headers pourraient aussi avoir un lien pour valeur. Nous voulons définir une propriété $location$, mais uniquement pour le header `"Location"`. C'est pourquoi nous introduisons d'abord une propriété réflexive $hasLocation$ uniquement pour les instances de la classe $Header$ qui ont `"Location"` pour valeur de $hdrName$.

$$Header \sqcap \exists hdrName.\{\texttt{"Location"}\} \equiv \exists hasLocation.Self$$
$$headers \circ hasLocation \circ link \sqsubseteq location$$

Ce mécanisme est illustré par les flèches en pointillés dans la figure 5 pour l'exemple de `Location`. Nous proposons de faire de même pour tous les éléments d'en-tête prédéfinis, donc aussi pour `Content-Type`. L'approche de HTTPinRDF est différente, la vue littérale des éléments d'en-têtes est raffinée avec la propriété `http:headerElements` dont les objets peuvent être décomposés avec `http:elementName`, `http:elementValue` ou avec `http:params` de façon générique. Nous conservons la généricité tout en proposant des raccourcis pour les en-têtes prédéfinis comme `Location`, ce qui permet une écriture plus succincte des requêtes mais augmente la taille de la TBox.

4.6 Contenu du corps

Comme définie dans la section 4.1, la propriété $body$ donne accès aux données du message. Son objet est instance de la classe $Content$. Dans HTTPinRDF la représentation du

contenu du corps du message est déléguée à un vocabulaire externe nommé Content (Koch *et al.*, 2017b) qui prend en compte le fait qu'une ressource peut être associée à plusieurs représentations dans différents formats. Cependant il restreint le co-domaine de $body$ au format $ContentAsBase64$ ce qui doit pouvoir être précisé puisqu'en HTTP le véritable format est indiqué par les éléments d'en-tête `Content-Type` et `Content-Encoding`. Le premier est obligatoire pour tout message ayant une propriété $body$ et sa valeur est un type de media $\mathcal{M}$. Ces éléments d'en-tête permettent au récepteur d'un message de savoir comment interpréter le contenu du corps. Par exemple lorsque `Content-Type` est `text/plain` et qu'il n'y a pas de `Content-Encoding` la valeur de $body$ devrait être directement une instance de la classe $ContentAsText$.

$$\exists body.Content \sqsubseteq \exists \texttt{content-type}.\mathcal{M}$$

Pour **QC 2** nous aimerions désigner les liens fournis dans un message. Certains sont dans les valeurs d'éléments d'en-tête comme `Location` mais d'autres peuvent être dans le corps lorsque le `Content-Type` correspond à un format hypermédia. RDF étant un moyen naturel de représenter ces liens, nous proposons que le contenu du corps soit disponible en RDF, ce qui, moyennant la précaution de correctement encapsuler ce graphe de contenu, permet de représenter à la fois le message et son contenu en RDF. Pour cela nous introduisons la classe $ContentAsRDF$ comme sous-classe de $Content$. La propriété $about$ sert de lien entre le graphe du message et celui du contenu, encapsulé dans un graphe nommé tel que défini dans la spécification de SPARQL 1.1.

$$ContentAsRDF \sqsubseteq Content \sqcap \exists about.\top \quad \top \sqsubseteq \forall about.Graph \quad \top \sqsubseteq \leq_1 about.\top$$

Cette propriété est fonctionnelle. Par exemple la représentation RDF d'un corps de message décrivant la ressource `:foo` peut se faire avec le langage TriG comme suit :

```
:B {
    :foo :ids (1 2 3) ;
         :date "2003-02-10"^^xsd:date .
}
```

Cette représentation peut ensuite être associée au contenu du corps de message présent dans le graphe par défaut de la manière suivante :

```
:m a http:Message ;
   http:body :b .
:b a cnt:ContentAsRDF ;
   cnt:about :B .
:B a sd:Graph .
```

Pour s'assurer d'avoir des contenus qui soient dans un format RDF (comme RDF/XML, JSON-LD ou Turtle) il est possible d'adopter la notion de *présentation RDF* (Lefrançois, 2018), qui fournit un moyen de transformer un format non-RDF en un graphe RDF et à l'inverse un moyen de ramener un graphe RDF dans le format non-RDF de départ.

5 Evaluation

La question de l'évaluation d'ontologie a reçu beaucoup d'attention et peut être considérée selon différentes catégories : logique, structurelle et fonctionnelle (Tartir *et al.*, 2010). La catégorie logique regroupe les dimensions de qualité qui peuvent être évaluées avec un raisonneur, par exemple la satisfaisabilité. Notre implémentation en OWL 2 DL nous permet de vérifier les inférences possibles avec le raisonneur Hermit, pour en particulier détecter les incohérences. Nous avons aussi vérifié l'absence d'incohérence par la *provocation d'erreur* au moment de l'instanciation de l'ontologie avec un ensemble d'individus représentatif. Une représentation visuelle du graphe RDF des individus correspondant à l'exemple en Figure 4 est fourni en Figure 6.

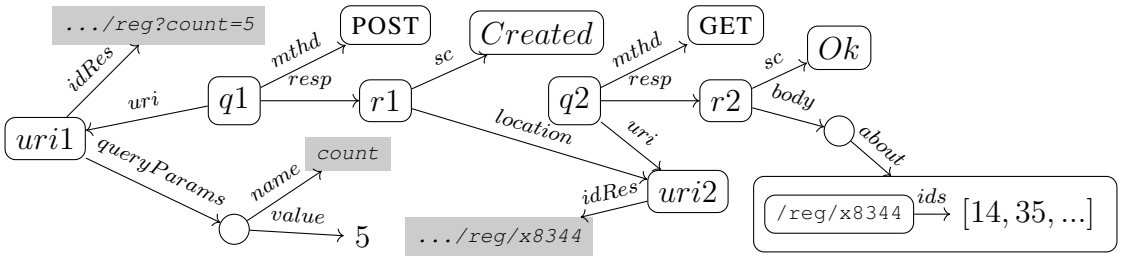

FIGURE *6 – Graphe RDF de l'exemple en Figure 4*

La catégorie *structurelle* est composée de dimensions non contextuelles qui peuvent être mesurées de manière quantitative. Par exemple en utilisant OntoMetrics [7] mais aussi la popularité ou le degré de couplage avec d'autres resources du Web des données. Dans notre cas, des dimensions importantes appartenant à cette catégorie sont les suivantes :

Flexibilité Est-ce que l'ontologie est facilement adapable à une variété d'usages ? Nous répondons à cette question avec notre proposition pour les en-têtes, le corps du message et les paramètres de requêtes, qui améliorent la réutilisabilité potentielle de l'ontologie par rapport à HTTPinRDF.

Transparence Est-ce que l'ontologie est facilement analysable ? Nous répondons à cette question avec notre formalisation en logique de description, implémentée en OWL2 DL, qui permet de l'explorer et la tester avec Protégé et les raisonneurs qu'il intègre.

Ergonomie cognitive Est-ce que l'ontologie est facilement compréhensible et exploitable par l'utilisateur ? Nous répondons à ce point en écrivant cet article pour expliquer nos choix, en documentant cette ontologie, et en la mettant à disposition en ligne.

Conformité à l'expertise Est-ce que l'ontologie est conforme avec les connaissances qu'elle représente ? Dans notre cas nous représentons mieux la sémantique des RFCs en introduisant des classes et propriétés en substitut de littéraux, par exemple pour les URIs. Cette dimension est aussi liée aux propriétés fonctionnelles, qui dans le cadre d'une ontologie d'application comme celle présentée ici sont d'une importance cruciale.

La catégorie *fonctionnelle* regroupe les dimensions de qualité liées aux usages et fonctions contextualisées. Nous addressons cela en écrivant des requêtes SPARQL à partir de notre ontologie dans le but de répondre aux *questions de compétence* exprimées en section 3.

QC 1 Quel est le type de média associé à un corps de message ?

```
SELECT ?m ?mt
WHERE {
  ?m a http:Message .
  ?m http:content-type ?mt .
}
```

QC 2 Quel est le numéro du code de statut d'une interaction ?

```
SELECT ?status
WHERE {
  ?q http:resp ?r .
  ?r http:sc/http:code ?status .
}
```

QC 3 Quel est l'URI fourni via l'élément `Location` d'un en-tête de réponse ?

```
SELECT ?next
WHERE {
  ?q0 http:resp/http:location ?next .
}
```

7. https://ontometrics.informatik.uni-rostock.de/ontologymetrics

QC 4 Quel est le code de statut associé à la réponse de la dernière interaction de la conversation en Figure 4 ?

```
SELECT ?status
WHERE {
  ?q0 http:resp/http:location/http:idRes ?next .
  ?q1 http:uri/http:idRes ?next .
  ?q1 http:resp/http:sc ?status .
}
```

QC 5 Est-ce que le type de média du corps d'une réponse correspond à l'un de ceux déclarés par un élément d'en-tête `Accept` dans la requête correspondante ?

```
ASK {
  ?q http:resp ?r .
  ?q http:accept/http:media-type ?mt1 .
  ?r http:content-type ?mt2 .
  FILTER (CONTAINS(STR(?mt1), STR(?mt2))
          || CONTAINS(STR(?mt2), STR(?mt1)))
}
```

QC 6 Quelles sont les valeurs de la propriété RDF `ex:ids` dans le corps de la deuxième réponse de la conversation en Figure 4 ?

```
SELECT ?ids
WHERE {
  ?m http:body/cnt:about ?G .
  GRAPH ?G { ?x ex:ids/rdf:rest*/rdf:first ?ids } .
}
```

Ici la propriété `ex:ids` associe la racine de la représentation du contenu avec une liste d'identifiants de type `rdf:List`.

QC 7 Quelle est la valeur d'un paramètre donné, par exemple `age`, passé dans l'URI d'un message requête ?

```
SELECT ?age
WHERE {
  ?q http:uri/http:queryParams ?p .
  ?p http:name "age" .
  ?p http:value ?age
}
```

6 Positionnement dans l'état de l'art

Notre objectif est de guider le travail de développeurs d'applications web : pour leur expliquer leurs erreurs un programme doit pouvoir traiter les différents éléments de HTTP et leur sens, il doit donc s'appuyer sur une ontologie les représentant. Cette ontologie peut aussi être un cadre pour la définition de spécification d'API Web, sous la forme de règles de validation. Dans cette section nous considérons donc le problème de spécification d'API Web, qui motive notre travail, sous différents points de vue, à travers les utilisations passées et actuelles des propositions existantes. D'un point de vue historique, aux débuts de la programmation d'applications Web on parlait de services Web, définis, décrits et publiés avec SOAP, WSDL et UDDI (Weerawarana *et al.*, 2005). De nombreux travaux visaient aussi la description de ces services Web à un niveau sémantique, avec OWL-S (Martin & et., 2005) ou encore SAWSDL (Kopecký *et al.*, 2007). Pour les interactions client-serveur, ce type de services Web repose sur une forme de communication dite Remote Procedure Call (RPC), popularisée par la programmation orientée objet, qui d'une part ignore le principe des liens hypermédia au coeur de l'architecture logicielle du Web et d'autre part consiste à définir et exposer des ensembles arbitraires d'opérations au lieu des méthodes définies dans HTTP. Ils ont été beaucoup étudiés et développés et de grandes entreprises les utilisent encore couramment. Pour les développeurs les spécifications en WSDL fournissent un excellent support

pour savoir comment utiliser un tel service, mais à notre connaissance aucune tentative visant à automatiser leur utilisation ne s'est avérée probante. Pourtant WSDL permet de fournir une description lisible par une machine, des moyens de faire appel au service, avec quels paramètres, et quels résultats sont retournés (Weerawarana *et al.*, 2005). Les services y sont décrits comme des collections de points d'accès réseau (ports), qui associent un URL avec une liaison dans laquelle sont décrits le protocole concret et les formats de message, à savoir les opérations supportées et les formes de données échangées (schémas). SAWSDL (Kopecký *et al.*, 2007) représente un ensemble d'attributs supplémentaires pour ajouter des annotations sémantiques aux différentes parties d'un document WSDL. Cela permet aux concepteurs de service Web de relier des déclarations WSDL et des schémas XML à des ontologies.

La suite de l'histoire des applications Web est dominée par les services REST ou les services RESTful (Richardson *et al.*, 2013) dits encore API Web hypermédia (Verborgh *et al.*, 2017), qui ont pris leur essor en même temps que l'émergence des Linked Data et la popularité croissante des formats JSON et JSON-LD liés au langage de développement web Javascript. Avant de considérer les propositions de spécification de tels services, il est important de remarquer l'universalité du besoin de fournir de l'information sur les fonctionnalités d'un service, ainsi que sur la forme et le sens des données échangées. C'est la vocation de SAWSDL pour les services qui suivent SOAP, ou de OWL-S (Martin & et., 2005) qui est proposé pour tout type de services. Ce dernier est d'ailleurs comparable à une autre proposition plus récente d'une ontologie pour la description de fonctions, Function Ontology (De Meester *et al.*, 2016), elle aussi destinée à compléter par un niveau plus abstrait les spécifications de services Web, lesquelles sont très couplées avec la technologie [8]. Il s'agit de permettre de déclarer et décrire n'importe quel problème, fonction ou algorithme, que cela soit implanté avec un service Web ou autrement. OWL-S et Function Ontology ont en commun de répondre aux besoins de descriptions pour la découverte, l'invocation et la composition automatique de services. OWL-S est une proposition ancienne (W3C Member Submission 2004) alors que Function Ontology est à un stade de première ébauche non officielle, conçue pour les applications du Web sémantique. Les deux propositions comprennent un mécanisme pour relier le niveau plus élevé des descriptions fonctionnelles au niveau plus concret des descriptions d'API Web, ces dernières pouvant être spécifiées en utilisant WSDL, ou l'ontologie HTTP, ou encore Hydra (Lanthaler & Gütl, 2013).

Hydra est un formalisme pour décrire et utiliser les API Web hypermédia, qui comme on l'a vu respectent le style architectural REST et sont plus simples à déployer et à utiliser que les services définis avec SOAP (Upadhyaya *et al.*, 2011; Richardson *et al.*, 2013). Ce sont ces API Web qui sont maintenant les plus développées et utilisées. OpenAPI (Initiative, 2018) est une initiative très suivie qui propose d'associer aux API Web une documentation lisible par un programme, qui peut être compilée en une page web, devenant ainsi découvrable avec un navigateur et un moteur de recherche standard. C'est un niveau de description seulement syntaxique qui permet pour ces services ce que WSDL offre pour les services SOAP, ou encore ce que permet WADL pour des services REST (JSON replaçant XML). De son côté Hydra offre un moyen d'exprimer une description sémantique avec l'objectif de simplifier encore plus le développement de services RESTful, tout en exploitant le potentiel du Linked Data (Lanthaler & Gütl, 2013), aspect qui n'est pas considéré par l'initiative OpenAPI. Hydra se présente comme un vocabulaire RDFS qui permet d'un côté de décrire les API Web et de l'autre d'augmenter les Linked Data avec des contrôles hypermédia (il permet de spécifier quels URI dans un graphe RDF sont prévus pour être déréférencés). Le W3C accueille par ailleurs le développement du standard Linked Data Platform (LDP) (Speicher *et al.*, 2015) qui a pour but de permettre aux fournisseurs de données liées d'exposer leurs jeux de données à la manière RESTful et comprend un modèle pour interagir (lire et écrire) avec ces données. Cela rejoint l'intention de Hydra de permettre de décrire des API Web RESTful qui consomment et produisent des données liées. Hydra est la proposition la plus proche de l'ontologie HTTP que nous proposons, elle en diffère toutefois par l'usage de classes et de propriétés pour représenter les composants de HTTP et des URI qui sont plus abstraites, tout en imposant de manipuler des notions propres aux données liées qui ne sont pas forcément familières pour

8. `https://w3id.org/function/spec`

les développeurs d'applications Web actuelles. Hydra est utilisé par plusieurs applications, par exemple l'une d'elles consiste à automatiser la découverte et l'utilisation de services Web grâce à des micro-services SPARQL (Michel *et al.*, 2019), qui permettent aux programmes d'interroger une API Web comme Flickr en utilisant SPARQL.

RESTdesc (Verborgh *et al.*, 2012, 2017) est aussi un format de description sémantique d'API Web hypermédia, qui utilise l'ontologie HTTP plutôt que Hydra, et Notation3 Logic (Berners-Lee *et al.*, 2007) plutôt que RDFS ou OWL. Il est conçu pour la découverte et la composition automatique de tels services, par des agents intelligents. Pour ses auteurs en effet, l'utilisation des liens hypermédia et des données liées doit permettre de développer des agents intelligents qui naviguent à travers les API et choisissent la poursuite de leurs actions au cours de cette navigation, comme le font les humains avec les pages Web. Cette vision est évidemment enthousiasmante mais elle reste éloignée des pratiques actuelles dans les systèmes d'information d'entreprise : la composition des services est encore programmée à la main, à partir d'un ensemble précis de services, et même ainsi, avec tout le soin mis dans ces développements, de nombreux problèmes se posent lorsque les services évoluent. Notre objectif est ainsi plus modeste que celui de RESTdesc, au moins dans un premier temps. Pour assister le travail quotidien des producteurs et consommateurs d'API Web nous cherchons à valider une API Web concrète par rapport à une spécification en expliquant les erreurs éventuelles. C'est pourquoi nous nous attachons à la représentation formelle de la connaissance que partagent ces développeurs, concernant les interactions HTTP.

7 Conclusion

Nous avons présenté une descritpion ontologique des interactions HTTP en nous basant sur une étude en profondeur de la RFC 7231. Notre point de départ était HTTPinRDF proposé par le W3C qui est utilisé dans différent contextes. Comme il s'appuie sur une interprétation limitée des RFC (plus proche de la syntaxe HTTP que de sa sémantique), il ne nous permet pas d'exprimer les requêtes qui correspondent aux questions de compétences représentant nos besoins. Nous avons mené une analyse formelle en utilisant à la fois la logique de description et OWL 2 DL pour introduire notre proposition de son évolution vers une ontologie HTTP qui répond à nos besoins comme montré par notre effort d'évaluation. En ce qui concerne les travaux liés, HTTPinRDF est la seule proposition qui est directement liée à notre objectif de décrire sémantiquement le protocole HTTP, même si l'objectif plus général de décrire des services Web est celui de nombreux travaux. Les réponses actuelles à cet objectif plus général ne sont pas directement utilisables pour valider une API Web concrète par rapport à la spécification d'HTTP, en expliquant aux développeurs leur potentielle erreurs.

La spécification RFC 7231 étant conséquente notre effort de formalisation est incomplet et de nombreux aspects restent à étudier. De plus nous avons observé dans le cas des paramètres de requête que les pratiques usuelles, qui sont importantes à prendre en compte du fait de leur large adoption, reposent sur des extensions *ad-hoc* de la spécification d'un URI. Nous sommes conscients des nombreuses limitations de notre approche. Par exemple nous ne sommes pas en mesure de décrire l'évolution des représentations. Nous n'exprimons pas non plus les relations de dépendance temporelle qui existent entre les messages. Une autre limitation vient de *l'hypothèse du monde ouvert* qui signifie qu'il n'est pas possible de vérifier l'absence de certains type d'instance, ce qui est utile dans un contexte de validation. Par exemple nous pourrions vouloir vérifier que les réponses associées à la méthode `HEAD` n'ont pas de corps. Combiner OWL et SHACL pourrait être la solution et un travail à venir est d'exploiter l'ontologie pour instrumenter l'exécution des interactions HTTP soit au niveau du client, soit au niveau du serveur. Les inférences logiques réalisées par le raisonneur OWL 2 permettront de vérifier la conformité de ces interaction avec la spécification du protocole. Par exemple il pourrait vérifier pour chaque message que l'en-tête `Content-Type` est défini de manière appropriée en présence d'un corps. Pour réaliser cela, nous travaillons sur un programme qui convertit des messages HTTP en graphes RDF utilisant notre ontologie.

Références

BERNERS-LEE T., CONNOLLY D., KAGAL L., SCHARF Y. & HENDLER J. A. (2007). N3logic : A logical framework for the world wide web. *Theory Pract. Log. Program.*, **8**, 249–269.

BERNERS-LEE T., FIELDING R. & MASINTER L. (2005). Uniform resource identifier (uri) : Generic syntax.

BLOMQVIST E., PRESUTTI V., DAGA E. & GANGEMI A. (2010). Experimenting with extreme design. In P. CIMIANO & H. S. PINTO, Eds., *Knowledge Engineering and Management by the Masses*, p. 120–134, Berlin, Heidelberg : Springer Berlin Heidelberg.

DE MEESTER B., DIMOU A., VERBORGH R. & MANNENS E. (2016). An Ontology to Semantically Declare and Describe Functions. In *The Semantic Web*, p. 46–49, Cham : Springer International Publishing.

FIELDING R. & RESCHKE J. (2014a). Hypertext transfer protocol (http/1.1) : Message syntax and routing.

FIELDING R. & RESCHKE J. (2014b). Hypertext transfer protocol (http/1.1) : Semantics and content.

FIELDING R. T. & TAYLOR R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, **2**(2), 115–150.

GUARINO N. (1998). Formal ontology and information systems. In *Proceedings of Formal Ontology in Information Systems*, p. 3–15 : IOS Press.

P. HITZLER, A. GANGEMI, K. JANOWICZ, A. KRISNADHI & V. PRESUTTI, Eds. (2016). *Ontology Engineering with Ontology Design Patterns - Foundations and Applications*, volume 25. IOS Press.

INITIATIVE O. (2018). The OpenAPI Specification.

KOCH J., VELASCO C. & ACKERMANN P. (2017a). HTTP Vocabulary in RDF 1.0. W3C Working Group.

KOCH J., VELASCO C. & ACKERMANN P. (2017b). Representing content in rdf 1.0.

KOPECKÝ J., VITVAR T., BOURNEZ C. & FARRELL J. (2007). SAWSDL : semantic annotations for WSDL and XML schema. *Internet Computing, IEEE*, **11**, 60–67.

LANTHALER M. & GÜTL C. (2013). Hydra : A Vocabulary for Hypermedia-Driven Web APIs. *LDOW*, **996**.

LEFRANÇOIS M. (2018). RDF presentation and correct content conveyance for legacy services and the web of things. In *Proceedings of the 8th International Conference on the Internet of Things*, p.Ã3 : ACM.

MARTIN D. & ET. A. (2005). Bringing Semantics to Web Services : The OWL-S Approach. In *Semantic Web Services and Web Process Composition*, p. 26–42, Berlin, Heidelberg : Springer Berlin Heidelberg.

MICHEL F., FARON-ZUCKER C., CORBY O. & GANDON F. (2019). Enabling automatic discovery and querying of web APIs at web scale using linked data standards. In *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019.*, p. 883–892.

NOY N. & MCGUINNESS D. (2001). Ontology development 101 : A guide to creating your first ontology. *Knowledge Systems Laboratory*, **32**.

RICHARDSON L., AMUNDSEN M. & RUBY S. (2013). *RESTful Web APIs*. O'Reilly Media.

SPEICHER S., ARWE J. & MALHOTRA A. (2015). *Linked Data Platform 1.0, W3C Recommendation*. Rapport interne, W3C.

TARTIR S., ARPINAR I. & SHETH A. (2010). *Ontological Evaluation and Validation*, p. 115–130.

UPADHYAYA B., ZOU Y., XIAO H., NG J. & LAU A. (2011). Migration of SOAP-based services to RESTful services. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, p. 105–114 : IEEE.

VERBORGH R., ARNDT D., HOECKE S. V., ROO J. D., MELS G., STEINER T. & GABARRÓ J. (2017). The pragmatic proof : Hypermedia API composition and execution. *Theory Pract. Log. Program.*, **17**(1), 1–48.

VERBORGH R., STEINER T., VAN DEURSEN D., COPPENS S., VALLÉS J. G. & VAN DE WALLE R. (2012). Functional descriptions as the bridge between hypermedia APIs and the semantic web. In *Proceedings of the third international workshop on RESTful design*, p. 33–40 : ACM.

WEERAWARANA S., CURBERA F., LEYMANN F., STOREY T. & FERGUSON D. F. (2005). *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. USA : Prentice Hall PTR.