# Semantic Navigation on the Web of Data: Specification of Routes, Web Fragments and Actions

Valeria Fionda
KRDB, Free University of Bozen-Bolzano, Italy
fionda@inf.unibz.it

Claudio Gutierrez
DCC, Universidad de Chile, Santiago, Chile
cgutierr@dcc.uchile.cl

Giuseppe Pirró
KRDB, Free University of Bozen-Bolzano, Italy
pirro@inf.unibz.it

## ABSTRACT

The massive semantic data sources linked in the Web of Data give new meaning to old features like navigation; introduce new challenges like semantic specification of Web fragments; and make it possible to specify actions relying on semantic data. In this paper we introduce a declarative language to face these challenges. Based on navigational features, it is designed to specify fragments of the Web of Data and actions to be performed based on these data. We implement it in a centralized fashion, and show its power and performance. Finally, we explore the same ideas in a distributed setting, showing their feasibility, potentialities and challenges.

## Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval; H.3.4 [**Information Systems**]: Systems and Software; E.2 [**Data**]: Data Storage and Representation

## Keywords

Navigation, Web of Data, Linked Data, Semantic Web

## 1. INTRODUCTION

Classically the Web has been modelled as a huge graph of links between pages [5]. This model included Web features such as links without labels and *only* generated by the owner of the page. Although Web pages are created and kept distributively, their small size and lack of structure stimulated the idea to view searching and querying through single and centralized repositories (built from pages via crawlers). With the advent of the Web of Data, that is, semantic data at massive scale [4, 18], these assumptions, in general, do not hold anymore. First, links are semantically labelled (thanks to RDF triples) thus can be used to orient and control the navigation. Besides, they are generated distributively and can be part of any data source thus enabling –using the words of Tim Berners-Lee– *anyone* to say *anything* about *anything* and publish it *anywhere* [3]. Second, data sources have a truly distributed nature due to their huge size, autonomous generation, and standard RDF structure.

In this setting, navigation along the nodes of the Web of Data, using the semantics stored in each data source, becomes significant. To model these issues, rather than as a graph, the Web of Data is better represented as a set of nodes plus data describing their semantic structure attached to each node (see Fig. 1). This model permits to better express the distributed creation and maintenance of data, and the fact that its structure is provided by dynamical and distributed data sources. In particular, it reflects the fact that at each moment of time, and for each particular agent, the whole network of data on the Web is unknown [22].

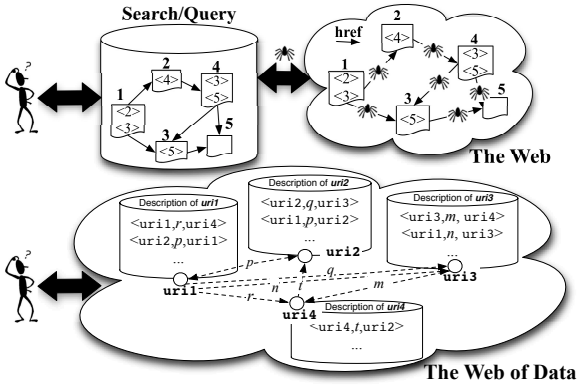



**Figure 1: Classical Web versus Web of Data. Size, distributive character, and semantic description of data gives navigation a prominent role.**

This scenario and the new underlying semantic data space set the stage for a new class of applications. At the core, lies the possibility of instructing software agents to navigate the Web of Data with an initial specification, which is then adjusted according to the local data encountered during the navigation. This dynamic and open high-level specification cannot be systematically simulated by current languages that do not exploit *online* data found during the navigation (see examples below and Related Work section). The desideratum is to have a simple language that can perform at least the following basic tasks in a declarative and integrated manner: *(i)* semi-automatic navigation "driven" by local information; *(ii)* specification of navigation charts, that is, semantic descriptions of fragments of the Web; *(iii)* specification of actions one would like to perform over the data encountered during the navigation (e.g., retrieving data, sending messages, etc.).

This paper presents such a language, called NAUTILOD, and shows that it can be readily implemented on the current Linked Open Data (LOD) network [18]. NAUTILOD has been implemented in the `swget` tool, which exploits current Web protocols and works on LOD. The distributed version of `swget` has been explored in a proof-of-concept implementation to show its feasibility, potentialities and challenges.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
*WWW 2012*, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

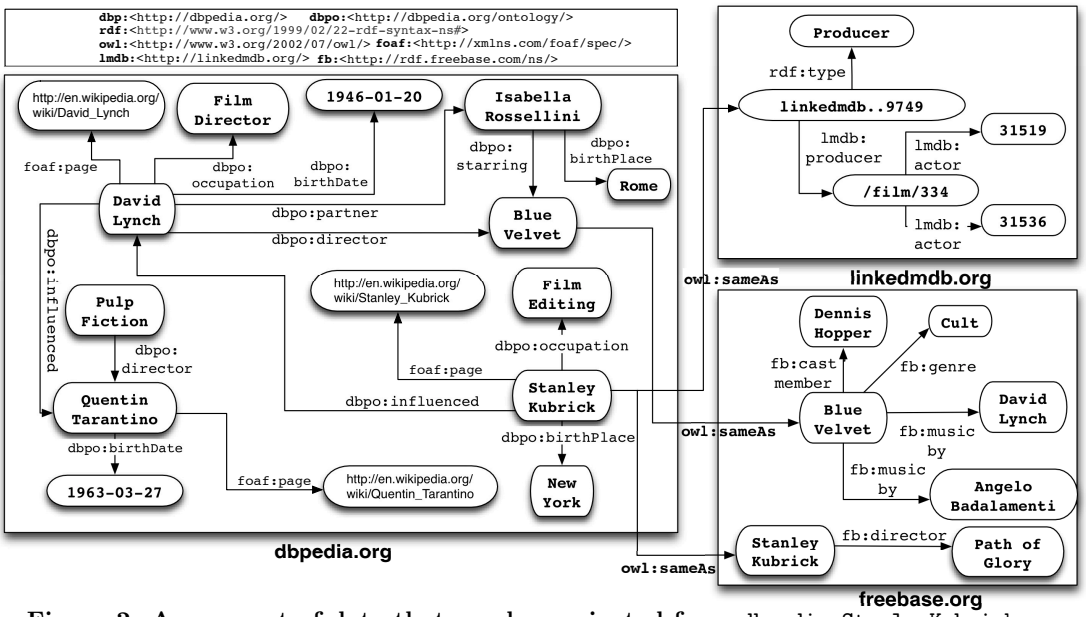



**Figure 2: An excerpt of data that can be navigated from `dbpedia:StanleyKubrick`.**

**NautiLOD by example**. To help the reader to get an idea of the language and its potentialities, we present some examples using the excerpt of real-world data shown in Fig. 2. The syntax and semantics are introduced in Section 3.

**Example** 1.1. *(**Aliases via** `owl:sameAs`) Specify what is predicated from Stanley Kubrick in DBpedia and also consider his possible aliases in other data sources.*

The idea is to have `<owl:sameAs>`-paths, which start from Kubrick's URI in DBpedia. Recursively, for each URI `u` reached, check in its data source the triples $\langle$`u`, `owl:sameAs`, `v`$\rangle$. Select all `v`'s found. Finally, for each of such `v`, return all URIs `w` in triples of the form $\langle$`v`, `p`, `w`$\rangle$ found in `v`'s data source. The specification in NautiLOD is:

$$(\texttt{<owl:sameAs>}) * /\texttt{<\_>}$$

where `<_>` denotes a wild card for RDF predicates. In Fig. 2, when evaluating this expression starting from the URI `dbp:StanleyKubrick` we get all the different representations of Stanley Kubrick provided by `dbpedia.org`, `freebase.org` and `linkedmdb.org`. From these nodes, the expression `<_>` matches any predicate. The final result is: {`dbp:DavidLynch, dbp:New York, dbp:FilmEditing,lmdb:Producer,lmdb:/film/334,fb:Path of Glory,http://en.wikipedia.org/wiki/Stanley_Kubrick`}. Note that the naive search for Kubrick's information in DBpedia, would only give {`http://en.wikipedia.org/wiki/Stanley_Kubrick, New York, David Lynch, Film Editing`}.

A more complex example, which extends standard navigational languages with actions and SPARQL queries is:

**Example** 1.2. *URIs of movies (and their aliases), whose director is more than 50 years old, and has been influenced, either directly or indirectly, by Stanley Kubrick. Send by email the Wiki pages of such directors as you get them.*

This specification involves `influence`-paths and aliases as in the previous example; tests (expressed in NautiLOD using ASK-SPARQL queries) over the dataset associated to a given URI (if somebody influenced by Kubrick is found, check if it has the right age); and actions to be performed using data form the data source. The NautiLOD specification is:

$$(\texttt{<dbpo:influenced>})\texttt{+[Test]/Act/<dbpo:director>/}\\ \texttt{/(<owl:sameAs>)*}$$

where the test and the action are as follows:
`Test= ASK ?p <dbpo:birthDate> ?y. FILTER(?y<1961-01-01).`
`Act= sendEmail(?p)[SELECT ?p WHERE {?x <foaf:page> ?p.}].`

In the expression, the symbol `+` denotes that one or more levels of influence are acceptable, e.g., we get directors like David Lynch and Quentin Tarantino. From this set of resources, the constraint on the age enforced by the ASK query is evaluated on the data source associated to each of the resources already matched. This filter leaves in this case only `dbp:DavidLynch`. At this point, over the elements of this set (one element in this case), the action will send via email the Wiki page (obtained from the SELECT query). The action `sendEmail`, implemented by an ad-hoc programming procedure, does not influence the navigation process. Thus, the evaluation will continue from the URI `u` =`dbp:DavidLynch`, by navigating the property `dbpo:director` (found in the dataset $\mathcal{D}$ obtained by dereferencing `u`). For example, in $\mathcal{D}$ we found the triple $\langle$`u`, `dbpo:director`, `dbp:BlueVelvet`$\rangle$. Then, from `dbp:BlueVelvet` we launch the final part of the expression, already seen in Example 1.1. It can be checked that the final result of the evaluation is: (1) the set {`dbp:BlueVelvet, fb:BlueVelvet`}, that is, data about the movie Blue Velvet from `dbpedia.org` and `freebase.org`; (2) the set of actions performed, in this case one email sent.

**Contributions of the paper.** The following are main the contributions of this paper:

(1) *We define a general declarative specification language*, called NautiLOD, whose navigational features exploit regular expressions on RDF predicates, enhanced with existential tests (based on ASK-SPARQL queries) and actions. It allows both to specify a set of sites that match the semantic description, and to orient the navigation using the information that these sites provide. Its basic navigational features

are inspired both by `wget` and `XPath`, enhanced with semantic specifications, using SPARQL to filter paths, and with actions to be performed while navigating. We present a simple syntax, a formal semantics and a basic cost analysis.

(2) *We implement a version of the language*, by developing the application `swget` that evaluates NautiLOD expressions in a centralized form (at the distinguished initial node). Being based on NautiLOD, `swget` permits to perform semantically-driven navigation of the Web of Data as well as retrieval actions. This tool relies on the computational resources of the initial node issuing the command and exploits the Web protocol HTTP. It is readily available on the current Linked Open Data (LOD) network. Its limitation is, of course, the scalability: the traffic of data involved could be high, making the navigation costly.

(3) *We implement* `swget` *in a distributed environment.* Based on simple assumptions on third parties (a small application that each server should run to join it, and that in many ways extends the idea of current *endpoints*), we show the feasibility of such an application that simulates a travelling agent, and hint at the powerful uses it can have. From this proof-of-concept, we explore the potentialities of this idea and its challenges.

The paper is organized as follows. Section 2 provides a quick overview of the Web of Data. In Section 3 the NautiLOD language is introduced: syntax, semantics and its evaluation cost. In Section 4 `swget`, a centralized implementation of NautiLOD is introduced: its architecture, pseudo-code and experimental evaluation. Section 5 deals with the distributed version of `swget`, showing the feasibility and potentialities of this application. Section 6 discusses related work. Finally, in Section 7 we draw conclusions and delineate future work.

## 2. PRELIMINARIES: THE WEB OF DATA

This section provides some background on RDF and Linked Open Data (LOD) that are at the basis of the Web of Data. For further details the reader can refer to [4, 12].

**RDF.** The Resource Description Framework (RDF) is a metadata model introduced by the W3C for representing information about resources in the Semantic Web. RDF is built upon the notion of *statement*. A statement defines the *property* $p$ holding between two resources, the *subject* $s$ and the *object* $o$. It is denote by $\langle s, p, o\rangle$, and thus called *triple* in RDF. A collection of RDF triples is referred to as an *RDF graph*. RDF exploits Uniform Resource Identifiers (URIs) to identify resources. URIs represent global identifiers in the Web and enable to access the *descriptions* of resources according to specific protocols (e.g., HTTP).

### 2.1 Web of Data - the LOD initiative

The LOD initiative leverages RDF to publish and interlinking resources on the Web. This enables a new (semantic) space called *Web of Data*. Objects in this space are *linked* and looked-up by exploiting (Semantic) Web languages and technologies. LOD is based on some principles, which can be seen more as best practices than formal constraints:

(1) Real world objects or abstract concepts must be assigned names on the form of URIs.

(2) In particular, HTTP URIs have to be used so that people can look them up by using existing technologies.

(3) When someone looks up a URI, associated information has to be provided in a standard form (e.g., RDF).

(4) Interconnections among URIs have to be provided by including references to other URIs.

An important notion in this context is that of dereferenceable URI. A dereferenceable URI represents an identifier of a real world entity that can be used to retrieve a representation, by an HTTP GET, of the resource it identifies. The client can negotiate the format (e.g., RDF, N3) in which it prefers to receive the description.

### 2.2 Data in the LOD

Data in the LOD are provided by sites (i.e., servers), which cover a variety of domains. For instance, `dbpedia.org` or `freebase.org` provide cross-domain information, `geonames.org` publishes geographic information, `pubmed.org` information in the domain of life-science.

Theoretically in each server resides an RDF triple-store (or a repository of RDF data). In order to obtain information about the resource identified by a URI `u`, a client has to perform an HTTP GET `u` request. This request is handled by the Linked Data server, which answers with a set triples. This is usually said to be the *dereferencing* of `u`. To simplify the presentation of the ideas, in this paper we consider only plain RDF and no blank nodes.

In the Web of Data, resources are not isolated from one another, in spirit with the fourth principle of LOD, but are linked. The interlinking of these resources and thus of the corresponding sites in which they reside forms the so called *Linked Open Data Cloud* [1].

## 3. A NAVIGATION LANGUAGE FOR THE WEB OF DATA

As we argued in the Introduction, there are data management challenges emerging in the Web of Data that need to be addressed. Particularly important are: *(i)* the specification of parts of this Web, thus of semantic fragments of it; *(ii)* the possibility to declaratively specify the navigation and exploit the semantics of data placed at each node of the Web; *(iii)* performing actions while navigating. To cope with this needs, this section presents a *navigation language* for the Web of Data, inspired by two non-related languages: `wget`, a language to automatically navigate and retrieve Web pages; and `XPath`, a language to specify parts of documents in the world of semi-structured data. We call it *Navigational language for Linked Open Data*, NautiLOD.

NautiLOD is built upon *navigational expressions*, based on regular expressions, filtered by tests using ASK-SPARQL queries (over the data residing in the nodes that are being navigated), and incorporating actions to be triggered while the navigation proceeds. NautiLOD allows to: *(i)* semantically specify collections of URIs; *(ii)* perform recursive navigation of the Web of Data, controlled using the semantics of the RDF data attached to the URIs that are visited (obtained by dereferencing these URIs); *(iii)* perform actions on specific URIs, e.g., selectively retrieve data from them.

Before presenting the language, we present in Section 3.1 an abstract data model of the Web of Data. Then we present the syntax of NautiLOD (Section 3.2), and the formal semantics (Section 3.3). Finally, we provide a basic cost model for the complexity of evaluating NautiLOD expressions.

[1] `http://richard.cyganiak.de/2007/10/lod/`

## 3.1 Data model

We define a minimal abstract model of the Web of Data to highlight the main features required in our discussion.

Let $\mathcal{U}$ be the set of all URIs and $\mathcal{L}$ the set of all literals. We distinguish between two types of triples. *RDF links* $\langle s, p, o\rangle \in \mathcal{U}\times\mathcal{U}\times\mathcal{U}$ that encode connections among resources in the Web of Data. *Literal triples*, $\langle s, p, o\rangle \in \mathcal{U} \times \mathcal{U} \times \mathcal{L}$, which are used to state properties or features of the resource identified by the subject $s$. Note that the object of a triple, in the general case, can be also a blank node. However, here we will not consider them to simplify the presentation of the main ideas (note also that the usage of blank nodes is discouraged [18]). Let $\mathcal{T}$ be the set of all triples in the Web of Data. The following three notions will be fundamental.

**Definition** 3.1 (Web of Data $\mathcal{T}$). *Let $\mathcal{U}$ and $\mathcal{L}$ be infinite sets. The Web of Data (over $\mathcal{U}$ and $\mathcal{L}$) is the set of triples $\langle s, p, o\rangle$ in $\mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$. We will denote it by $\mathcal{T}$.*

**Definition** 3.2 (Description Function $\mathcal{D}$). *A function $\mathcal{D} : \mathcal{U} \to P(\mathcal{T})$ associates to each URI $\mathtt{u} \in \mathcal{U}$ a subset of triples of $\mathcal{T}$, denoted by $\mathcal{D}(\mathtt{u})$, which is the set of triples obtained by dereferencing* u.

**Definition** 3.3 (Web of Data Instance $\mathcal{W}$). *A Web of Data instance is a pair $\mathcal{W} = \langle \mathcal{U}, \mathcal{D}\rangle$, where $\mathcal{U}$ is the set of all URIs and $\mathcal{D}$ is a description function.*

Note that not all the URIs in $\mathcal{U}$ are dereferenceable. If a URI $\mathtt{u} \in \mathcal{U}$ is not dereferenceable then $\mathcal{D}(\mathtt{u}) = \emptyset$.

## 3.2 Syntax

NautiLOD provides a mechanism to declaratively: (i) define *navigational expressions*; (ii) allow *semantic control* over the navigation via test queries; (iii) *retrieve data* by performing actions as side-effects along the navigational path.

The navigational core of the language is based on regular path expressions, pretty much like Web query languages and XPath. The semantic control is done via existential tests using ASK-SPARQL queries. This mechanism allows to redirect the navigation based on the information present at each node of the navigation path. Finally, the language allows to command actions during the navigation according to decisions based on the original specification and the local information found.

| | |
|---|---|
| `path` ::= | `pred` \| `pred`$^{-1}$ \| `action` \| `path/path` |
| | \| (`path`)? \| (`path`)$*$ \| (`path`\|`path`) \| `path`[`test`] |
| `pred` ::= | **<RDF predicate>** \| **<_>** |
| `test` ::= | **ASK-SPARQL query** |
| `action` ::= | **procedure[Select-SPARQL query]** |

**Table 1: Syntax of the NautiLOD language.**

The syntax of the language NautiLOD is defined according to the grammar reported in Table 1. The language is based on *Paths Expressions*, that is, concatenation of base-case expressions built over *predicates*, *tests* and *actions*. The language accepts concatenations of basic and complex types of expressions. Basic expressions are predicates and actions; complex expressions are disjunctions of expressions, expressions involving a number of repetitions using the features of regular languages, and expressions followed by a test. The building blocks of a NautiLOD expression are:

(1) *Predicates.* The base case. `pred` can be an RDF predicate or the wildcard `<_>` used to denote *any* predicate.

(2) *Test Expressions.* A `test` denotes a query expression. Its base case is an ASK-SPARQL query.

(3) *Action Expressions.* An `action` is a procedural specification of a command (e.g., send a notification message, PUT and GET commands on the Web, etc.), which obtains its *parameters* from the data source reached during the navigation. It is a side-effect, that is, it does not influence the subsequent navigation process.

If restricted to (1) and (2), NautiLOD can be seen as a declarative language to describe portions of the Web of Data, i.e., set of URIs conform to some semantic specification.

## 3.3 Semantics

NautiLOD expressions are evaluated against a Web of Data instance $\mathcal{W}$ and a URI `u` indicating the starting point of the evaluation. The meaning of a NautiLOD expression is a set of URIs defined by the expression plus a set of actions produced by its evaluation. The resulting set of URIs are the leaves in the paths according to the NautiLOD expression, originating from the seed URI `u`.

For instance, the expression `<type>`, evaluated over `u`, will return the set of URIs $\mathtt{u}_k$ reachable from `u` by "navigating" the predicate `<type>`, that is, by inspecting triples of the form $\langle \mathtt{u}, \texttt{<type>}, \mathtt{u}_k\rangle$ included in $\mathcal{D}(\mathtt{u})$. Similarly, the expression `<type>`[q] will filter, from the results of the evaluation of `<type>`, those URIs $\mathtt{u}_k$ for which the query `q` evaluated on their descriptions $\mathcal{D}(\mathtt{u}_k)$ is true. Finally, the evaluation of an expression `<type>`[q]/a will return the results of `<type>`[q] *and* perform the action `a` (possibly using some data from $\mathcal{D}(\mathtt{u}_k)$). The formal semantics of NautiLOD is reported in Table 2. The fragment of the language without actions follows the lines of formalization of XPath by Wadler [30]. Actions are treated essentially as side-effects and evaluated while navigating. Given and expression, a Web of Data instance $\mathcal{W} = \langle \mathcal{U}, \mathcal{D}\rangle$, and a seed URI `u` the semantics has the following modules:

- $E[\![\texttt{path}]\!](\mathtt{u}, \mathcal{W})$: evaluates the set of URIs selected by the navigational expression `path` starting from the URI `u` in the Web of Data instance $\mathcal{W}$. Additionally, it collects the actions associated to each of such URIs.
- $U[\![\texttt{path}]\!](\mathtt{u}, \mathcal{W})$: defines the set of URIs specified by the expression `path` when forgetting the actions.
- $A[\![\texttt{path}]\!](\mathtt{u}, \mathcal{W})$: executes the actions specified by the evaluation of the navigational expression `path`.
- $Sem[\![\texttt{path}]\!](\mathtt{u}, \mathcal{W})$: outputs the meaning of the expression `path`, namely, the ordered pair of two sets: the set of URIs specified by the evaluation of `path`; and the set of actions performed according to this information.

*Note on some decisions made.* Any sensible real implementation can benefit from giving an order to the elements of the output action set. As far as the formal semantics, at this stage we assumed that actions are inde1pendent from one another and that the world $\mathcal{W}$ is static during the evaluation (to avoid to overload our discussion with the relevant issue of synchronization, that is at this point orthogonal to the current proposal). Thus, we decided to denote the actions produced by the evaluation of an expression as a set. It is not difficult to see that one could have chosen a list as the semantics for output actions.

$$
\begin{array}{rcl}
E[\![\texttt{<p>}]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u}',\bot) \mid \langle \mathtt{u},\texttt{<p>},\mathtt{u}'\rangle \in \mathcal{D}(\mathtt{u})\} \\
E[\![(\texttt{<p>})^{-1}]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u}',\bot) \mid \langle \mathtt{u}',\texttt{<p>},\mathtt{u}\rangle \in \mathcal{D}(\mathtt{u})\} \\
E[\![\texttt{<\_>}]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u}',\bot) \mid \exists \texttt{<p>}, \langle \mathtt{u},\texttt{<p>},\mathtt{u}'\rangle \in \mathcal{D}(\mathtt{u})\} \\
E[\![\mathtt{act}]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u},\mathtt{act})\} \\
E[\![\mathtt{path}_1/\mathtt{path}_2]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u}'',a) \in E[\![\mathtt{path}_2]\!](\mathtt{u}',\mathcal{W}) : \exists b, (\mathtt{u}',b) \in E[\![\mathtt{path}_1]\!](\mathtt{u},\mathcal{W})\} \\
E[\![(\mathtt{path})?]\!](\mathtt{u},\mathcal{W}) & = & \{(u,\bot)\} \cup E[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}) \\
E[\![(\mathtt{path})*]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u},\bot)\} \cup \bigcup_1^\infty E[\![\mathtt{path}_i]\!](\mathtt{u},\mathcal{W}) \mid \mathtt{path}_1 = \mathtt{path} \wedge \mathtt{path}_i = \mathtt{path}_{i-1}/\mathtt{path} \\
E[\![\mathtt{path}_1|\mathtt{path}_2]\!](\mathtt{u},\mathcal{W}) & = & E[\![\mathtt{path}_1]\!](\mathtt{u},\mathcal{W}) \cup E[\![\mathtt{path}_2]\!](\mathtt{u},\mathcal{W}) \\
E[\![\mathtt{path}[\mathtt{test}]]\!](\mathtt{u},\mathcal{W}) & = & \{(\mathtt{u}',a) \in E[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}) : \mathtt{test}(\mathtt{u}') = \mathtt{true}\} \\
U[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}) & = & \{v : \exists a, (v,a) \in E[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W})\} \\
A[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}) & = & \{\mathtt{Exec}(a,v) : (v,a) \in E[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W})\} \\
\mathrm{Sem}[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}) & = & (U[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}), A[\![\mathtt{path}]\!](\mathtt{u},\mathcal{W}))
\end{array}
$$

**Table 2: Semantics of NautiLOD. The semantics of an expression is composed of two sets: (1) the set of URIs of $\mathcal{W}$ satisfying the specification; (2) the actions produced by the evaluation of the specification. $\mathtt{Exec}(a,u)$ denotes the execution of action $a$ over $u$. $\bot$ indicates the empty action (i.e., no action).**

## 3.4 Evaluation of Cost and Complexity

We present a general analysis of cost and complexity of the evaluation of NautiLOD expressions over a Web of Data instance $\mathcal{W}$. We can separate the cost in three parts, where $E$ are expressions, $\overline{E}$ action-and-test-free expressions, $A$ actions and $T$ tests:

$$cost(E,\mathcal{W}) = cost(\overline{E},\mathcal{W}) + cost(A) + cost(T). \quad (1)$$

Since actions do not affect the navigation process we can treat their cost separately. Besides, in our language, tests are ASK-SPARQL queries having a different structure from the pure navigational path expressions of the language. Even in this case we can treat their cost independently.

**Actions.** NautiLOD is designed for acting on the Web of Data. In this scenario, the cost of actions has essentially two components: *execution* and *transmission*. The execution cost boils down to the cost of evaluating the SELECT SPARQL query that gives the action's parameters. As for transmission costs, a typical example is the `wget` command, where the cost is the one given by the GET data command.

**Action-and-test-free.** This fragment of NautiLOD can be considered essentially as the PF fragment of *XPath* (location paths without conditions), that is well known to be (with respect to combined complexity) **NL**-complete under **L**-reductions (Thm. 4.3, [11]). The idea of the proof is simple: membership in **NL** follows from the fact that we can guess the path while we verify it in time **L**. The hardness essentially follows from a reduction from the directed graph reachability problem. Thus we have:

Theorem 3.4. *With respect to combined complexity, the action-and-test-free fragment of* NautiLOD *is* **NL***-complete under* **L***-reductions.*

*Combined* refers to the fact that the input parameters are the expression size and the data size. Note that what really matters is not the whole Web (the data), but only the set of nodes reachable by the expression. Thus it is more precise to speak of expression size plus set-of-visited nodes size. The worst case is of course the whole size of the Web.

**Tests.** The evaluation of tests (i.e., ASK-SPARQL queries) has a cost. This cost is well known and one could choose particular fragments of SPARQL to control it [24]. However, tests will possibly reduce the size of the set of nodes visited during the evaluation. Thus the $cost(\overline{E},\mathcal{W})$ has to be reduced to take into account the effective subset of nodes reachable thanks to the filtering performed by the tests. Let $\mathcal{W}_T$ be the portion of $\mathcal{W}$ when taking into account this filtering. We have:

$$cost(E,\mathcal{W}) = cost(\overline{E},\mathcal{W}_T) + cost(A) + cost(T). \quad (2)$$

Section 4.2 will discuss some examples on real world data by underlining the contribution of each component of the cost.

**Remark.** In a distributed setting, with partially unknown information and a network of almost unbound size, the notion "cost of evaluating an expression e" appears less significant than in a controlled centralized environment. In this scenario, a more pertinent question seems to be: "given an amount of resources $r$ and the expression $e$, how much can I get with $r$ satisfying $e$?". This calls for optimizing (according to some parameters) the navigation starting from a given URI $u$, according to equation (2).

# 4. IMPLEMENTATION OF NautiLOD

This section deals with `swget` [9], a tool implementing NautiLOD. `swget` implements all the navigational features of NautiLOD, a set of actions centred on retrieving data, and adds (for practical reasons) a set of ad-hoc options for further controlling the navigation from a network-oriented perspective (e.g., size of data transferred, latency time).

`swget` has been implemented in Java and is available as: *(i)* a developer release, which includes a command-line tool that is easily embeddable in custom applications; *(ii)* an end user release, which features a GUI. Further details, examples, the complete syntax along with the downloadable versions are available at the `swget`'s Web site [2].

## 4.1 Architecture

The high level architecture of `swget` is reported in the left part of Fig. 3. The *Command interpreter* receives the input, i.e., a seed `URI`, a NautiLOD expression and a set of options. The input is then passed to the *Controller* module, which checks if a network request is admissible and possibly passes it to the *Network Manager*. A request is admissible if it complies with what specified by the NautiLOD expression and with the network-related navigation parameters (see Section 4.1.1). The *Network Manager* performs HTTP GET

[2] `http://swget.wordpress.com`

requests to obtain streams of RDF data. These streams are processed for obtaining Jena RDF models, which will be passed to the *Link Extractor*. The *Link Extractor* takes in input an automaton constructed by the NAUTILOD *interpreter* and selects a subset of outgoing links in the current model according to the current state of the automaton. The set is given to the *Controller Module*, which starts over the cycle. The execution will end either when some navigational parameter imposes it (e.g., a threshold has been reached) or when there are no more URIs to be dereferenced.

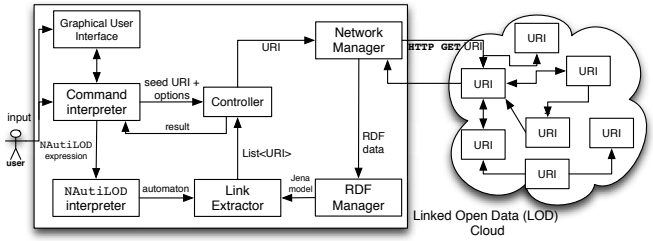



**Figure 3: `swget` architecture and scenario.**

### 4.1.1 Network-based controlled navigation

NAUTILOD is designed to semantically control the navigation. However, it can be the case that a user wants to control the navigation also in terms of network traffic generated. A typical example is a user running `swget` from a mobile device with limited Internet capabilities. This is why `swget` includes features to add more control to the navigation through the parameters reported in Table 3. Each option is given in input to `swget` as a pair $\langle$`param`, `value`$\rangle$.

**Table 3: Network params to control the navigation**

| Parameter | Value | Meaning |
|---|---|---|
| `maxDerTriples` | int | max number of triples allowed in each dereferencing |
| `saveGraph` | boolean | Save the graphs dereferenced |
| `maxSize` | int | traffic limit (in MBs) |
| `timeoutDer` | long | connection time-out |
| `timeout` | long | total time-out |
| `domains` | List<String> | trusted servers |

To illustrate a possible scenario where the navigation can be controlled both from a semantic and network-based perspective consider, the following example.

**Example** 4.1. *(**Controlled navigation**) Find information about Rome, starting from its definition in DBpedia and includes other possible definitions of Rome linked to DBpedia but only if their description contains less than 500 triples and belongs to DBpedia, Freebase or The New York Times.*

```
swget < dbp:Rome> (<owl:sameAs>)* -saveGraph
  -domains {dbpedia.org,rdf.freebase.com,
    data.nytimes.com} -maxDerTriples 500
```

The command, besides the NAUTILOD expression, contains the `-domains` and `-maxDerTriples` parameters to control the navigation on the basis of the trust toward information providers and the number of triples, respectively.

## 4.2 Evaluating NAUTILOD expressions

Given a NAUTILOD expression `e`, the transitions between states of the automaton associated to `e` handles the navigation process. Algorithm 1 reports the `swget` controlled navigation algorithm while Table 4 describes the high level primitives used to interact with the automaton.

The algorithm takes as input a *seed* `URI`, a NAUTILOD expression and a set of network parameters, and it returns a set of URIs and literals conform to the expression and the network parameters. For each URI involved in the evaluation, possible tests (line 9) and actions (line 12) are considered.

The procedure `navigate` extracts links (line 3) from a resource identified by `p.uri` toward other resources. Note that `p.uri` is considered either when appearing as the *subject* or the *object* of each triple to comply with the Linked Data initial proposal [2] [section on browsable graphs].

**Algorithm 1:** `swget` pseudo-code

**Input** : e=NAUTILOD expression; seed=URI; par=Parms<n,v>
**Output**: set of URIs and literals conform to e and par;

```
1  a = buildAutomaton(e);
2  addLookUpPair(seed, a.getInitial());
3  while (∃ p=<uri,state> to look up and checkNet(par)=OK) do
4     desc=getDescription(p.uri);
5     if (a.isFinal(p.state)) then
6        addToResult(p.uri);
7     if (not alreadyLookedUp(p)) then
8        setAlreadyLookedUp(p);
9        if (t=getTest(p.state)≠ ∅ and evalT(t,desc)=true) then
10          s=a.nextState(p.state,t);
11          addLookUpPair(p.uri,s);
12       if (act=getAction(p.state)≠ ∅) then
13          if(evalA(act.test,desc)=true) then exeC(act.cmd);
14          s=a.nextState(p.state,act));
15          addLookUpPair(p.uri,s);
16       out=navigate(p,a,desc);
17       for (each URI pair p'=<uri,state> in out) do
18          addLookUpPair(p');
19       for (each literal pair lit=<literal,state> in out) do
20          if (a.isFinal(lit.state)) then
21             addToResult(lit.literal);
22 return Result;
```

**Function** navigate(`exp`,`a`,`desc`)

**Output**: List of <uri,state> and <literal,state>

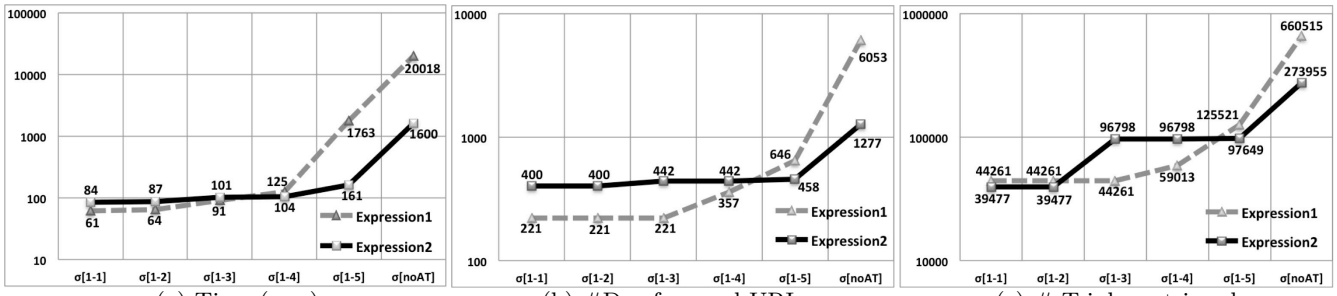
```
1  for (each pred in a.nextP(p.state)) do
2     nextS=a.nextState(p.state,pred);
3     query= "SELECT ?x WHERE
      {{ ?x pred p.uri} UNION{ p.uri pred ?x}}";
4     for (each res in evalQ(query, desc)) do
5        addOutput(res,nextS);
6  return Output;
```

**Table 4: Primitives for accessing the automaton.**

| Primitive | Behaviour |
|---|---|
| `getInitial()` | returns the initial state $q_0$ |
| `nextP(q)` | returns the set $\{\sigma \mid \delta(q,\sigma) = q_1\}$ of tokens (i.e., predicates) enabling a transition from $q$ to $q_1$ |
| `getTest(q)` | returns the test to be performed into the current automaton state |
| `getAction(q)` | returns the action to be performed into the current automaton state |
| `nextState(q,`$\sigma$`)` | returns the state that can be reached from `q` by the token $\sigma$ |
| `isFinal(q)` | returns `TRUE` if `q` is an accepting state |

## 4.3 Experimental Evaluation

To show real costs of evaluating the different components of `swget` expressions over real-world data, we choose two complex expressions (shown in Fig. 5) to be evaluated over the Linked Open Data network. We report the results of `swget` in terms of execution time ($t$), URIs dereferenced ($d$) and number of triples retrieved ($n$). Each expression has been divided in 5 parts (i.e., $\sigma_i, i \in \{1..5\}$). They have been executed as whole (i.e., $\sigma_{[1\text{-}5]}$) and as action-and-test-free expressions (i.e., $\sigma_{[noAT]}$), which correspond to $\overline{E}_1$ and $\overline{E}_2$,




(a) Time (secs) (b) #Dereferenced URIs (c) # Triples retrieved

**Figure 4: Evaluation of `swget`. Each expression has been executed 4 times. Average results are reported.**

respectively (see Section 3.4). Moreover, the various sub-expressions (i.e., $\sigma_{[1\text{-}i]}, i \in \{1..4\}$) have also been executed. This leads to a total of 12 expressions. For each expression, the corresponding *sub-Web* has been locally retrieved. That is, for each reachable URI the corresponding RDF graph has been locally stored. The aim of the evaluation is to investigate how the various components in the cost model presented in Section 3.4 affect the parameters $t$, $d$ and $n$. The results of the evaluation, in logarithmic scale, are reported in Fig. 4(a)-(c). In particular, in the x-axis are reported from left to right: the 4 sub-expressions, the full expression (i.e., $\sigma_{[1\text{-}5]}$) and the action-and-test-free expression (i.e., $\sigma_{[noAT]}$).

seed URI: =**http://dbp:Stanley_Kubrick** **$E_1$**
$\sigma 1$: <dbpo:influenced><3>
$\sigma 2$: [ASK ?p <dbpo:birthDate> ?y.FILTER(?y > 1961-01-01)]
$\sigma 3$: {sendEmail(?p)[SELECT ?p WHERE {?x <foaf:name> ?p. }]}
$\sigma 4$: <dbpo:director>
$\sigma 5$: <owl:sameAs>?

seed URI:=**http://dbp:Italy** **$E_2$**
$\sigma 1$: <dbpo:homeTown>
$\sigma 2$: [ASK ?person <rdf:type> <dbpo:Person>.
?person <rdf:type> <dbpo:MusicalArtist>]
$\sigma 3$: <dbpo:birthPlace>
$\sigma 4$: [ASK ?t <dbpo:populationTotal> ?p. FILTER(?p<15000)]
$\sigma 5$: <owl:sameAs>*

**Figure 5: Expressions used in the evaluation**

The first expression ($E_1$) starts by finding people influenced by Stanley Kubrick up to a level 3 (subexpr. $\sigma_{[1\text{-}1]}$). This operation requires about 61 secs., for a total of 221 URIs dereferenced. On the description of each of these 221 URIs, an ASK query is performed to select only those entities that were born after 1961 (subexpr. $\sigma_{[1\text{-}2]}$). The execution time of the queries is of about 4 secs. (i.e., $\simeq 0.02$ secs., per query) and 31 entities have been selected. Then, an action is performed on the descriptions of these 31 entities by selecting their `<foaf:name>` to be sent via email (subexpr. $\sigma_{[1\text{-}3]}$). In total, the select, the rendering of the results in an HTML format and the transmission of the emails cost about 25 secs. The navigation continues from the 31 entities before the action to get movies through the property `<dbpo:director>` (subexpr. $\sigma_{[1\text{-}4]}$). The cost is of about 34 secs., for a total of 136 movies. Finally, for each movie only one level of possible additional descriptions is searched by the `<owl:sameAs>` property (the whole expr. $\sigma_{[1\text{-}5]}$) whose cost is 1638 secs., for a total of 409 new URIs available from multiple servers (e.g., `linkedmdb.org`, `freebase.org`) of which only 289 were dereferenceable.

By referring to the cost model in Section 3.4 we have that $cost(E_1, \mathcal{W})$=$cost(\overline{E}_1, \mathcal{W}_{T_1})$+$cost(A_1)$+$cost(T_1)$=1763. The factor $cost(A_1) \simeq 25$ secs., whereas $cost(T_1) \simeq 4$ secs., and $cost(\overline{E}_1, \mathcal{W}_{T_1}) \simeq 1738$ secs. If we consider the test-and-action-free expression executed over the whole Web of Data (i.e., $\mathcal{W}$), we have that $cost(\overline{E}_1, \mathcal{W}) \simeq 20018$ secs. Note that the ASK queries costs about 4 secs., and permits to reduce the portion of the Web of Data navigated by $\overline{E}_1$, which enables to save about 20018-1738=18280 secs. Such a larger difference in the execution time is justified by the fact that the 222 initial URIs, selected by $\sigma_{[1\text{-}1]}$ are not filtered in the case of $(\overline{E}_1, \mathcal{W})$ and then cause a larger amount of paths to be followed at the second level. Indeed, the total number of dereferenced URIs for $(\overline{E}_1, \mathcal{W})$ is 6053, with about 660K triples retrieved, while for $(\overline{E}_1, \mathcal{W}_{T_1})$ is 646, with 125K triples retrieved.

The second expression ($E_2$) starts by navigating the property `<dbpo:homeTown>` to find entities living in Italy (subexpr. $\sigma_{[1\text{-}1]}$) with an execution time of about 84 secs., and a total of 400 dereferenced URIs, one seed and 399 URIs of entities. On the description of each of these 399 URIs, an ASK query filters entities that are of type `<dbpo:Person>` and `<dbpo:MusicalArtist>` (subexpr. $\sigma_{[1\text{-}2]}$). All the queries are performed in about 3.8 secs., with an average time per query of 0.01 secs., to select 156 persons. The navigation continues through the property `<dbpo:birthPlace>` to find the places where these persons were born (subexpr. $\sigma_{[1\text{-}3]}$), which costs about 14 secs. In total, 43 new URIs have been reached. The navigation continues with a second ASK query to select only those places in which live less than 15000 habitants (subexpr. $\sigma_{[1\text{-}4]}$). The cost of performing the 43 ASK queries is of about 3 secs and 5 places are selected. Finally, for each of the 5 places additional descriptions are searched by navigating the `<owl:sameAs>` property (the whole expr. $\sigma_{[1\text{-}5]}$). This allows to reach a total of 29 URIs, some of which are external to `dbpedia.org`. The cost for this operation is of about 57 secs.

As for the cost, $cost(E_2, \mathcal{W})$=$cost(\overline{E}_2, \mathcal{W}_{T_2})$+$cost(A_2)$ + $cost(T_2)$=161. The factor $cost(A_2)$=0 since $E_2$ does not contain any action whereas $cost(T_2) \simeq 6$ secs., which gives $cost(\overline{E}_2, \mathcal{W}_{T_2})$=155 secs. The cost of the test-and-action-free expression (i.e., $\overline{E}_2$) over $\mathcal{W}$ is $cost(\overline{E}_2, \mathcal{W}) \simeq 1600$ secs., with 1277 URIs dereferenced. This is because the expression is not selective since it performs a sort of "semantic" crawling only based on RDF predicates. In fact, the number of triples retrieved (see Fig. 4(c)) is almost three times higher than in the case of the expression with tests. By including the tests, the evaluation of $\overline{E}_2$ is 1445 secs. faster.

## 5. A PROPOSAL FOR DISTRIBUTED `swget`

This section presents and overview of Distributed `swget` (`Dswget`), which has the peculiarity that the processing of NautiLOD expressions is carried out cooperatively among LOD information providers.

## 5.1 Dswget: making LOD servers cooperate

swget enables controlled navigation but it heavily relies on the client that initiates the request. However, one may think of the Linked Data servers storing RDF triples as to peers in a Peer-to-Peer (P2P) network, where links are given by URIs in RDF triples. For instance ⟨dbp:Rome, owl:sameAs, fb:Rome⟩ links dbpedia.org with freebase.org. Indeed, there are some differences w.r.t. a traditional P2P network. First, Linked Data servers are less volatile than peers. Second, it is reasonable to assume that the computational power of Linked Data servers is higher than that of a traditional peers. This enables to handle a higher number of connections with the associated data.

Our proposal is to leverage the computational power of servers in the network to cooperatively evaluate swget commands. This enables to drastically reduce the amount of data transferred. In fact, data is not transferred from servers to the client that initiates the request (in response to HTTP GETs). Servers will exchange swget commands plus some metadata and operate on their data locally. This can be achieved by installing on each server in the network a Dswget engine and coordinating the cooperation by an ad-hoc distributed algorithm.

A Dswget command is issued by a Dswget client to the server to which the *seed* URI belongs. Each server involved in the computation will receive, handle and forward commands and results by using the Procedure handle. Note that in this procedure there are calls to some primitives reported in Table 4 and to the function navigate described in Section 4.2. The specific primitives needed by Dswget are reported in Table 5.

**Procedure** handle(client_id,e,URIs, metadata)

**Input**: client_id=address of the client; e=NautiLOD expression; URIs=set of pairs <URI,A_state>; metadata=additional data (e.g., current state of the automaton, request id)

```
1  a=buildAutomaton(e);
2  for (each p=<uri,state> in URIs) do
3     desc=getDescription(p.uri); //local call no deref. needed
4     if (not alreadyLookedUp(p)) then
5        setAlreadyLookedUp(p);
6        if (t=getTest(p.state)≠ ∅ and evalT(t,desc)=true) then
7           s=a.nextState(p.state,t));
8           addLookUpPair(p.uri,s);
9        if (act=getAction(p.state)≠ ∅) then
10          if(evalA(act.test,desc)=true) then exeC(act.cmd);
11       out=navigate(p,a,desc);
12       for (each URI pair p'=<uri,state> in out) do
13          if (a.isFinal(p'.state)) then
14             addtoResults(p'.uri);
15             else addLookUpPair(p');
16       for (each literal pair lit=<literal,state> in out) do
17          if (a.isFinal(lit.state)) then
18             addToResult(lit.literal);
19 sendResults(client_id);
20 fwdToServers(client_id,e);
```

**Table 5: Primitives of Dswget**

| Primitive | Behaviour |
|---|---|
| sendResults | sends to the original client (partial) results, which are URIs (line 14) and literals (line 18) |
| fwdToServers | forwards to other servers, the initial client address, the NautiLOD expression and a set of pairs <URI, A_state>. For each pair, the computation on a URI will be started from the corresponding A_state |

### 5.1.1 A Running example

To see an example of how Dswget works, consider the following request originated from a Dswget client:

**Example** 5.1. *(Dswget) Starting from DBpedia, find cities with less than 15000 persons, along with their aliases, in which musicians, currently living in Italy, were born.*

The Dswget command is reported in Fig. 6, which also reports a possible Dswget interaction scenario. On each linked data server a Dswget engine has been installed. Each server exposes a set of dereferenceable URIs for which the corresponding RDF descriptions are available. RDF data enable both internal references (e.g., dbp:Rome and dbp:uri1) and external ones (e.g., fb:Enrico and geo:Paris). In Fig. 6 references between URIs are represented by dotted arrows. When not explicitly mentioned, it is assumed that the reference occurs on a generic predicate. The automaton associated to this expression, having q4 and q5 as accepting states, is also reported. The state(s) of the automaton on which a server is operating is(are) reported in grey. Dswget protocol messages have been numbered to emphasize the order in which they are exchanged. The command along with some medatada (e.g., the address of the client) is issued by the client's Dswget engine toward the server to which the seed URI belongs (i.e., dbpedia.org in this example). The Dswget engine at this server, after locally building the automaton, starts the processing of the NautiLOD expression at the state q0. It obtains from its local RDF store, the description of Rome $\mathcal{D}$(dbp:Rome) and looks for URIs having dbpo:hometown as a predicate. In Fig 6, the URI fb:Enrico satisfies this pattern.

The Dswget engine at dbpedia.org performs the first transition of state, that is, $\delta(q0, \sigma 1) = q1$. The automaton does not reach a final state, and then the process has to continue. Since the URI fb:Enrico belongs to another server, the Dswget engine at dbpedia.org, communicates with that at freebase.org by seeding the initial NautiLOD expression, the URI for which freebase.org is involved in the computation (i.e., fb:Enrico) and the current state of the automaton. If multiple URIs have to be sent, they are packed together in a unique message.

With a similar reasoning the request reaches the Dswget engine at geonames.org, which checks if it is possible to reach the next state of the automaton starting from the URI passed by freebase.org. It has to check on $\mathcal{D}$(geo:Solarolo) if the query represented by $\sigma_4$ can be satisfied, that is, whether this city has less than 15K habitants. Then, the state q4 is reached, which is a final state. The Dswget engine at geonames.org contacts directly the Dswget engine of the client that issued the request and send the result (i.e., the URI geo:Solarolo). The address of the client is passed at each communication among Dswget engines.

Note that the automaton has another final state, that is, q5 that can be reached if there exist some triples in $\mathcal{D}$(geo:Solarolo) having an owl:sameAs predicate. Such a triple is ⟨geo:Solarolo, owl:sameAs, yago:Solarolo⟩. Therefore, the Dswget engine at geonames.org sends to the client the result (i.e., yago:Solarolo) and continues the process by sending to the engine at yago.org the URI in the object of this triple, the expression and the current state of the automaton. In this case since in $\mathcal{D}$(yago:Solarolo) there are no more triples having owl:sameAs as predicate, the process ends.

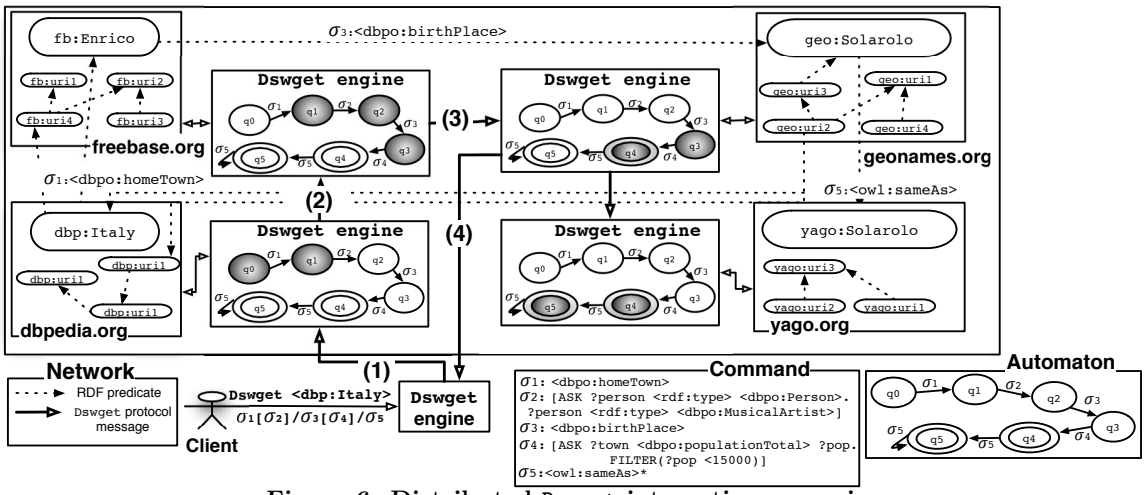



**Figure 6: Distributed Dswget interaction scenario.**

## 5.2 Dswget Design issues: an overview

In designing **Dswget** several issues, typical of the distributed systems, have been faced. Here we briefly report on the main of them without getting into too technical details.

In the Web of Data, a client to get information about a resource issues an HTTP GET request toward the HTTP server where the resource is hosted. In the standard case, the HTTP protocol offers a blocking semantics for its primitives, which means that once a request is issued the client has to wait for an answer or until a time-out. In **Dswget**, since engines exchange messages and data in a P2P fashion, a blocking semantics for communications would block the whole execution. To face this issue, specific asynchronous communication primitives and a *job delegation* mechanism are needed. With job delegation we mean that the sending **Dswget** engine delegates part of the execution and evaluation of a (sub)NautiLOD expression to the receiving engine(s). In this respect, since a request, through the mechanism of *job delegation* is spread among multiple **Dswget** engines it is necessary to handle the termination of requests to avoid to keep consuming resources in an uncontrolled way. **Dswget** tackles this issue from two different perspectives:

(1) *Loop detection*: each **Dswget** engine keeps track, for each request, of each URI along with the state of the automaton on which it has been processed.

(2) *Termination*: this problem can be addressed by each **Dswget** engine which, for each request it receives informs the client that initially issued the request about the fact that it has operated on this request and whether it has delegated other **Dswget** engines. Then, the client can keep track of the list of the active engines on a particular request. The **Dswget** engine may additionally send back to the client the state of the automaton on which it is operating, thus enabling the client to know how far the execution is from a final state.

# 6. RELATED WORK

Many of the ideas underlying our proposal have been around in particular settings. We owe inspiration to several of them.

Languages for the *navigation and specification* of nodes in a graph have a long tradition. Among them, XPath for XML and some proposals extending SPARQL with navigational features (e.g., [1, 21, 25, 31] and SPARQL 1.1). All these approaches allow the evaluation of path expressions but do not navigate *online* according to the data found during the navigation. Besides, most of them assume that data is stored in a central repository, typically a single RDF graph. Several query languages for the Web have been proposed (see [10] for a survey) but these are not grounded on semantic technologies and languages. Nonetheless, all these languages have inspired the navigational core of NautiLOD.

Specification (and retrieval) of collections of sites was early addressed, and a good example is the well known tool **wget**. Besides being non-declarative, it is restricted to almost purely syntactic features. At semantic level, Harth et al. [20] proposed LDSpider, a crawler for the Web of Data able to retrieve RDF data by following RDF links according to different crawling strategies. LDSpider has little flexibility and is not declarative. The execution philosophy of **wget** was a source of inspiration for the incorporation of actions into NautiLOD and to the design of **swget**. Distributed data management has been explored and implemented by P2P and similar approaches [29]. For RDF, RDFPeers [6] and YARS2 [15] use P2P to answer RDF queries while systems like DIASPORA [27] handle distributed query processing on the Web. Our distributed version of **swget** borrows some ideas from these approaches. Finally, it is important to stress the fact that there is a solid body of work on query processing and navigation on the Web of Data. Three lines of research can be identified:

(1) Load the *desired* data into a single RDF store (by *crawling* the LOD or some sub-portions) and process queries in a centralized way. There is a large list of Triple Stores [19]. There have been also developments in indexing techniques for semantic data. Swoogle [8], Sindice [23] and Watson [7] among the most successful. Recently, an approximate index structure for summarizing the content of Linked Data sources has been proposed by Harth et al. [14].

(2) Process the queries in a distributed form by using a federated query processor. DARQ [26] and FedX [28] provide mechanisms for transparently query answering on multiple query services. The query is split into sub-queries that are forwarded to the individual data sources and their results processed together. An evaluation of federated query approaches can be found in [13].

(3) Extend SPARQL with navigational features. The SERVICE feature of SPARQL 1.1 and proposals like SQUIN. [16, 17] extend the scope of SPARQL queries with navigational features. SQUIN is a query execution paradigm based on *link-traversal*, which discovers on the fly data sources relevant for the query and permits to automatically navigate to other sources while executing a query.

As it can be seen, our approach has a different departure point: it focuses on *navigational* functionalities, thus departing from querying as in (2); emphasizes specification of autonomous distributed sources, as opposed to (1); uses SPARQL querying to enhance navigation, while (3) proceeds in the reverse direction; and incorporates actions that in some sense generalize procedures implicit in the evaluation over the Web (*e.g.*, "get data" in crawlers and "return data" in query languages).

# 7. CONCLUSIONS AND FUTURE WORK

We presented the NautiLOD language to navigate, specify fragments and perform actions on the Web of Data. NautiLOD explicitly exploits the semantics of the data "stored" at each URI. We discussed `swget`, an implementation of NautiLOD in a centralized setting that works over real-world data, namely the LOD network. We also developed a distributed version of `swget` as proof-of-concept of its feasibility, potentialities and challenges.

The most important conclusion we can draw from this research and development is that the semantics given by RDF specifications *can be used* with profit to navigate, specify places and actions on the Web of Data. NautiLOD can be used as the basis for the development of agents, that can work immediately over LOD, to get data, navigate and report while navigating.

A second relevant finding is that there are some limitations for taking full advantage of the language and tools we developed. They refer essentially to (1) lack of standards in the sites regarding the dereferencing of data; (2) lack of standard RDF metadata regarding properties of the sites themselves (e.g., provenance, summary of contents, etc.); (3) weak infrastructure to host delegation of execution and evaluation (of the language) to permit distribution. Tackling these issues is our wish list to leverage the Web of Data.

**Acknowledgements.** Authors were supported by Marie Curie action IRSES under Grant No. 24761 (Net2). Gutierrez was supported by FONDECYT grant No.1110287.

# 8. REFERENCES

[1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with Regular Expression Patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.
[2] T. Berners-Lee. Linked data design issues.
[3] T. Berners-Lee. What the Semantic Web Can Represent, 1998.
[4] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *IJSWIS*, 5(3):1–22, 2009.
[5] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.
[6] M. Cai and M. Frank. RDFPeers: a Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In *WWW*, 2004.
[7] M. d'Aquin and E. Motta. Watson, more than a Semantic Web Search Engine. *Semantic Web*, 2(1):55–63, 2011.
[8] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, and J. Sachs. Swoogle: A Search and Metadata Engine for the Semantic Web. In *CIKM*, 2004.
[9] V. Fionda, C. Gutierrez, and G. Pirró. Semantically-driven Recursive Navigation and Retrieval of Data Sources in the Web of Data. In *Posters-ISWC*, 2011.
[10] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World-Wide Web: a survey. *SIGMOD Rec.*, 27:59–74, 1998.
[11] G. Gottlob, C Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *PODS*, 2003.
[12] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez. Foundations of Semantic Web Databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.
[13] P. Haase, T. Mathäb, and M. Ziller. An Evaluation of Approaches to Federated Query Processing over Linked Data. In *I-SEMANTICS*, 2010.
[14] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-demand Queries over Linked Data. In *WWW*, 2010.
[15] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC*, 2007.
[16] O. Hartig. Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution. In *ESWC*, 2011.
[17] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *ISWC*, 2009.
[18] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.
[19] K. Hose, R. Schenkel, M. Theobald, and G. Weikum. Database Foundations for Scalable RDF Processing. In *Reasoning Web*, 2011.
[20] R. Isele, A. Harth, J. Umbrich, and C. Bizer. LDspider: An open-source crawling framework for the Web of Linked Data. In *Posters-ISWC*, 2010.
[21] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *ESWC*, 2007.
[22] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.
[23] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello. Sindice.com: A document-oriented lookup index for open linked data. *Int. J. of Metad., Semant. and Ontolog.*, 3(1), 2008.
[24] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM TODS*, 34(3), 2009.
[25] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. *JWS*, 8(4), 2010.
[26] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*, 2008.
[27] M. Ramanath and J. R. Haritsa. DIASPORA: A Highly Distributed Web-Query Processing System. *World Wide Web*, 3(2):111–124, 2000.
[28] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, 2011.
[29] P. Valduriez and E. Pacitti. Data Management in Large-Scale P2P Systems. In *VECPAR*, 2004.
[30] P. Wadler. Two semantics for XPath, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.
[31] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: Expressive Navigation in RDF Graphs. In *RR*, 2010.