LDP-DL: A language to define the design of Linked Data Platforms

Noorani Bakerally, Antoine Zimmermann, Olivier Boissier

Univ Lyon, IMT Mines
Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023 Saint-Étienne, France
{noorani.bakerally,antoine.zimmermann,olivier.boissier}@emse.fr

Abstract. Linked Data Platform 1.0 (LDP) is the W3C Recommendation for exposing linked data in a RESTful manner. While several implementations of the LDP standard exist, deploying an LDP from existing data sources still involves much manual development. This is because there is currently no support for automatizing generation of LDP on these implementations. To this end, we propose an approach whose core is a language for specifying how existing data sources should be used to generate LDPs in a way that is independent of and compatible with any LDP implementation and deployable on any of them. We formally describe the syntax and semantics of the language and its implementation. We show that our approach 1) allows the reuse of the same design for multiple deployments, or 2) the same data with different designs, 3) is open to heterogeneous data sources, 4) can cope with hosting constraints and 5) significantly automatizes deployment of LDPs.

Keywords: RDF, Linked Data, Linked Data Platform

1 Introduction

In the context of open data, data sets are made available through Web data portals with the intention to offer innovative third party developers the opportunity to provide new services to end users. In particular, today's smart cities usually publish urban data openly. However, exploitation of urban open data is made very difficult by the current heterogeneity of data sets. The problem becomes even more prominent when considering multiple data portals from several metropolises.

Semantic Web technologies are largely addressing heterogeneity issues, with a uniform identification mechanism (URIs), a uniform data model (RDF), and a standard ontology language (OWL). Recently, a new standard was added to provide a uniform data access mechanism for linked data, based on RESTful principles: the Linked Data Platform 1.0 standard (LDP [18]). Considering the alledged advantages of the Semantic Web and Linked Data, we believe that LDPs can ease the path towards achieving Tim Berners-Lee's 5-star[1] open data scheme.

Yet, while LDPs greatly simplify the work of data users (developers, analysts, journalists, etc.), it puts a heavy burden on the data publishers, who have to make use of these new, unfamiliar technologies. The move towards LDPs requires a redesign of how the data is organized and published. Moreover, current LDP implementations are

[1] `http://5stardata.info/en/`, last accessed 23 March 2018

in their early stage and provide no automatic support for deploying data whether it is static, dynamic or heterogeneous. This paper is an attempt to provide a solution targeted towards data publishers to facilitate the use and deployment of linked data platforms conforming with the LDP standard from existing data sources.

We start by identifying requirements that any solution to the problem should satisfy (Sec. 2.1), then show how the state of the art is failing to satisfy them (Sec. 2.2), and subsequently propose a general approach to more easily design and deploy LDPs (Sec. 2.3). An important and central part of the approach is to provide a language, LDP Design Language (LDP-DL), for declaratively defining the design of data organization and deployment. This paper is mostly focused on describing this language, with Sec. 3.3 dedicated to its abstract syntax and Sec. 3.4 to its semantics. Our implementation is explained in Sec. 4.1 followed by Sec. 4.2 that describes what experiments we conducted to highlight the requirements that the approach satisfies. Finally, immediate directions towards overcoming the limitations are presented in the concluding section (Sec. 5).

2 Foundations and Motivations

In order to better understand the problem that we consider, we start by listing the requirements that the automatic generation of LDP from existing data sources should satisfy (cf. Sec. 2.1). To that aim, we will use the motivating example of LDP deployment in smart cities that we are considering in the OpenSensingCity project[2]. However, let's keep in mind that these requirements are also relevant for other application domains as well. We analyze and point current limitations in the current approaches with respect to these requirements (cf. Sec. 2.2). We end this section then by presenting a global view of the approach that we propose in this paper (cf. Sec. 2.3)

2.1 Requirements

To better explain the requirements on automatic LDP generation, let's take the example of a city governmental institution that decides to expose the data (open or not) produced in the city, in order to enable their exploitation by smart city applications to support citizens in their activities in the city.

To that aim, it decides to deploy a data platform. In order to enhance interoperability and homogenize access, the choice has been made to use a data platform which is compliant with Semantics Web standards including the LDP standard. However, in order to deploy such a Linked Data Platform in this context, the following requirements must be satisfied :

- **Handling heterogeneous data sources. (*Heterogeneity*)** A city is a decentralized and open ecosystem where data come from different organizations that are normally heterogeneous. As such, the city LDP may have to exploit and aggregate data from these sources and must be therefore open to heterogeneous data sources.
- **Handling hosting constraints. (*Hosting Constraints*)** Smart city and open data consist of data sources whose exploitation can give rise to hosting constraints that prevent from hosting a copy of the data in a third-party environment. Such constraints can be on the data itself (e.g. license restrictions), or it can be a limitation of the third-party software environment (e.g. bandwidth or storage limitations to continuously

[2] http://opensensingcity.emse.fr/

verify and maintain fresh copies of dynamic or real-time data). Thus, the city LDP has to be able to cope with hosting constraints.

- **Reusable design. (*Reusability*)** If LDPs are spreading in different cities, one can easily imagine that there may be a city wishing to reuse the design of another city LDP to expose their data in a similar way. One potential reason for doing so may be to enhance integration and access of their date to cross-city applications. Such applications may exploit any city LDP as long as the LDPs use both a design and vocabulary known by the application.
- **Automated LDP generation. (*Automatization*)** Finally, the use of existing LDP implementations (discussed in next section) necessitates much manual development and thus requires time and expertise that the city may not want to invest when putting in place its LDP. Also, having automated solutions may help organizations wishing to open their data in conformance to Semantic Web standards via LDPs.

2.2 LDP overview and current limitations of existing approaches

The LDP standard provides a model to organize data resources known as LDP resources and an interaction model to interact (read-write) with them. Two types of LDP resources exist: LDP RDF Sources (LDP-RS) and LDP Non-RDF Sources. These resources can be organized in LDP containers and as such are known as member resources. An LDP container is itself an LDP-RS. Three types of containers exist, but currently in our work, we only consider LDP Basic Containers (LDP-BC) in which members are limited to Web documents. Note that among current LDP implementations (discussed below), most support LDP-BCs and fewer support other types of containers. In this paper, we want facilitate the way LDP resources are organized in term of their IRIs, the content and organization in LDP containers and the content of LDP-RSs.

There are existing solutions for deploying linked data that do not conform with the LDP standard. Although we would like to take advantage of the homogeneous access mechanism of LDP, some non-LDP-conformant tools partially cover our requirements. Pubby[3], D2R Server [5], Virtuoso[4] and Triplify [2] are such examples. Triplify and D2R, have been designed to expose relational data as linked data and are focused on mappings. The final steps of publishing linked data only involves ensuring resources can be dereferenced with RDF. Virtuoso goes a step forward by doing the latter as well as providing a linked data interface to its triple store. Pubby can provide a linked data interface both to SPARQL endpoints and static RDF documents. Pubby and Virtuoso are the only tools for directly publishing RDF data as linked data in a highly automatized way. However, most design decisions are fixed and cannot be parameterized. Moreover, these solutions implement their own interpretation of the linked data principles, where data access and the content of RDF sources are neither standardized nor customizable. For instance, Pubby uses `DESCRIBE` queries to provide the content of RDF resources, which is a feature whose implementation varies from a SPARQL query engine to another. In summary, these tools offer the automatization that we require, but with little flexibility, and lacking the standard data access mechanism of LDP.

[3] `http://wifo5-03.informatik.uni-mannheim.de/pubby/` on 18 May 2017
[4] `https://virtuoso.openlinksw.com` on 19 July 2017

Concerning LDP implementations, they are mostly referenced in the standard conformance report[5] with the exception of Cavendish[6] which, to our knowledge, is the only one not referenced. We categorize them into *LDP resource management systems* (Cavendish, Callimachus, Carbon LDP, Fedora Commons, Apache Marmotta, Gold, rww-play, LDP.js) and *LDP frameworks* (Eclipse Lyo, LDP4j). LDP resource management systems can be seen as a repository for resources on top of which read and write operations conforming to the LDP standard are allowed. Currently, they do not satisfy our requirements because their deployment requires hardcoding most design decisions into appropriate write requests to an already deployed platform. On the other hand, LDP frameworks can be used to build custom applications which implement LDP interactions. While they are more flexible than management systems, they are not satisfying our requirements because their use involves much manual development. In summary, current LDP implementations are in their early stages as there is little to no support for automating the generation and deployment of LDPs from existing data, even if it is already in RDF.

Besides existing implementations, the current scientific literature about LDP is itself limited. To our knowledge four works [14, 15, 12, 16] have a focus on LDP. Among them, only [16] provides an approach for automatizing generations of LDPs from relational data using simple R2RML mapping (no SQL view, no multiple mappings to a class/property). While it minimally address the *Heterogeneity* requirement, it is rigid as it is not possible to customize the design of the output LDP. Apart from this, we find no other work that attemps to automatize the generation of LDPs.

2.3 Our Approach: The LDP Generation Workflow

In order to satisfy the requirements listed in Sec. 2.1, we provide an approach based on model-driven engineering that involves using models as first-class entities and transforming them into running systems by using generators or by dynamically interpreting the models at run-time [9]. Doing so enables separation of concerns thus guaranteeing higher reusability of systems' models [20].

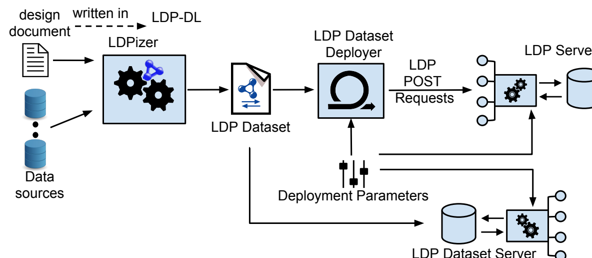

Fig. 1: General overview of our LDP Generation Workflow proposal

Fig. 1 shows a general overview of the approach. The LDP generation workflow includes two processes: LDPization and deployment. In the former process, the LDPizer consumes a design document written in our language, LDP-DL, that we use as a domain-specific language, a core component of model-driven engineering, to explicitly

[5] https://www.w3.org/2012/ldp/hg/tests/reports/ldp.html on 19 July 2017
[6] https://github.com/cavendish-ldp/cavendish on 07 July 2018

describe LDP design models. Doing so enables us to consider our *Reusability* requirement as the models are standalone, independent and separate from any implementation. The LDPizer interprets the model and exploits the data sources to generate what we call an LDP dataset (defined in Sec. 3), which is a structure to store LDP resources introduced to abstract ways from how current implementations store resources. The *Heterogeneity* requirement is handled by the possibility to specify mappings between non-RDF data to RDF graphs. The deployment process involves configuring the LDP and loading the LDP dataset into it. It can be done in two ways based on the nature of the LDP server. First, if the LDP server accepts `POST` requests, an LDP Dataset Deployer can generate and send such requests for each resource contained in the LDP dataset. Second, using an LDP server that can directly consume the LDP dataset and expose resources from it. Such a server can generate the content of requested resources at query time thus enable considering the *Hosting Constraint* requirement by avoiding the need for storing content of LDP resources. For now, our approach only requires the design document from which the entire LDP can be automatically generated (*Automatization* requirement).

In our approach, we exploit possibilities of model-driven engineering by performing model-to-model transformation when generating an LDP dataset from a design document in the LDPization process and by performing model-to-system transformation by generating an LDP from an LDP dataset in the deployment process.

3 LDP Design Language

In this section, we describe our language LDP-DL. We start with a general overview of its key concepts and provide its abstract syntax and formal semantics.

3.1 Illustrative Example

Fig. 2 shows an illustrative example that will be used throughout this section and later on. It comprises of an RDF graph that uses the DCAT vocabulary [13], shown in Fig. 2(a). The graph shows how the data appear on the original data portal from which we want to generate an LDP. Fig. 2(b) shows how we want to organize the data in the LDP, displaying the nesting of containers. In the DCAT vocabulary, data catalogs have datasets that are associated with themes and distributions. The organization of resources in the LDP in Fig. 2(b) uses a structure similar to DCAT where there are containers for describing catalogs that contains other containers for describing their datasets. The dataset containers in turn contain two containers for grouping non-containers that describe their distributions and themes. In this case, we want the resources identified under the namespace `ex:` to be described in an LDP available at the namespace `dex:`. On the LDP, we would like, e.g., that resources `dex:parking` and `dex:pJSON` be dereferenceable to obtain, respectively, the graphs shown in Fig. 2(c) and Fig. 2(d). Thus, `dex:parking` is an LDP resource that describes `ex:parking` with an graph that contains a subset of the original RDF graph in Fig. 2(a). What the design language must describe is how we can exploit the graph in Fig. 2(a) to generate the container hierarchy of Fig. 2(b), and make available the descriptions of the resources as RDF graphs found in Fig. 2(c) and Fig. 2(d).

3.2 Overview of the language

As mentioned in Sec. 2.2, in this paper, we restrict ourselves only to LDPs where all containers are basic and exclude non-RDF sources. From an abstract point of view, the

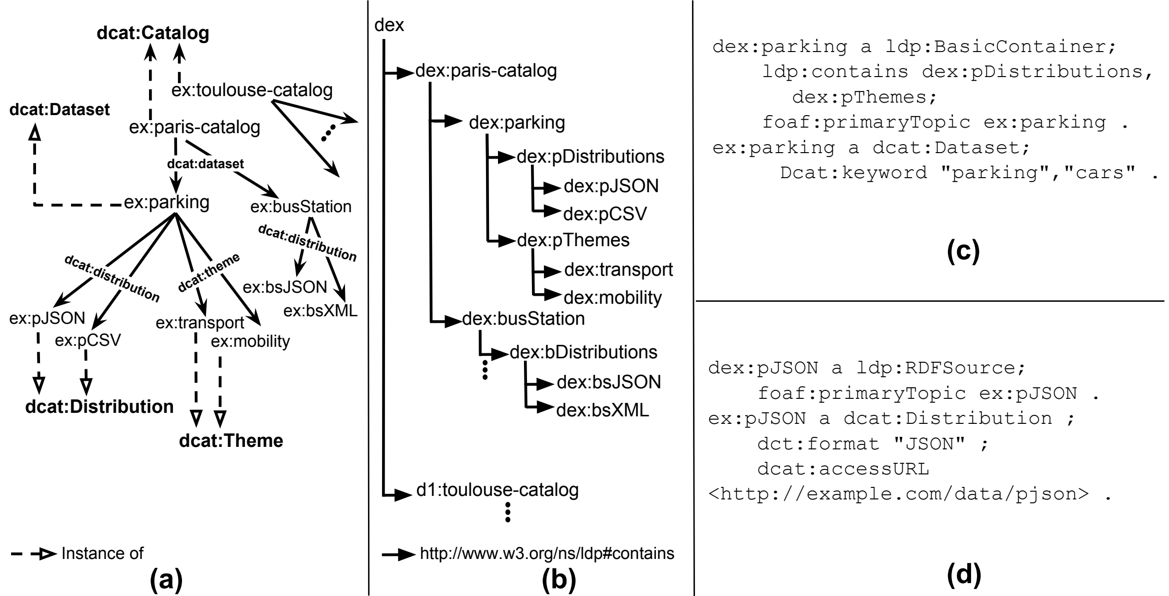

Fig. 2: Example of structure of an LDP with its data source and graphs

data in such an LDP can be described as an *LDP dataset*, a structure where each LDP resource is assigned a URL and has an associated RDF graph, and a set of members if it is a container. In it, pairs $(url, graph)$ representing a non-container, formalize the fact that accessing the URL on the LDP returns the graph, whereas triples $(url, graph, M)$ indicates that not only access to the URL returns the graph but the resource is a container whose members are in M. For example, in Fig. 2(b), `dex:parking` is the URL of a container associated with the graph in Fig. 2(c) and having members `dex:distributions` and `dex:themes`. Furthermore, `dex:pJSON` is the URL of a non-container in Fig. 2(b) with its graph in Fig. 2(d).

In a nutshell, LDP-DL provides constructs for describing the generation of an LDP dataset from existing data sources. In general, data sources may not be in RDF or may contain resources whose IRIs do not dereference to the LDP. Therefore, associated LDP resources within the LDP namespace may have to be generated to describe resources from the original data sources. For example, `dex:parking`, from Fig. 2(b), has been generated for the resource `ex:parking` from Fig. 2(a). `ex:parking` cannot be used directly as an LDP resource. Doing so may violate the LDP standard with respect to the lifecycle of the resource as the standard states that "a contained LDPR cannot be created (...) before its containing LDPC exists" [17,§2].[7] This is why in LDP-DL, to expose a resource from the data source via an LDP, a new LDP-RS is always generated to describe it. The resource for which an LDP-RS is generated is called the *related resource.* Thus, the related resource of the LDP-RS `dex:parking` is `ex:parking`. Let us note that an LDP-RS may not have a related resource, such as `dex:pDistributions` from Fig. 2(b). This is because it describes the set of distributions of `ex:parking`

[7] "LDPR" means LDP resource and "LDPC" means LDP container in the standard.

and such set is not itself identified as a proper resource in the data source. Fig. 3 shows an overview of the language in UML that we further describes in the next section

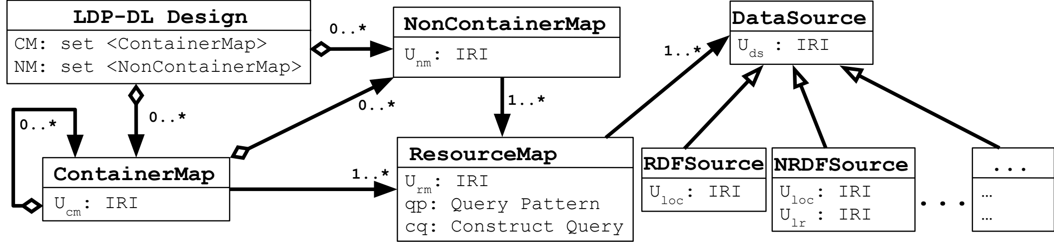

Fig. 3: Overview of the main constructs of LDP-DL in UML notation.

.

3.3 Abstract Syntax

Hereafter, we assume familiarity with the concepts of *IRIs*, *RDF graphs*, *named graphs*, *query variables*, *query patterns*, *construct queries*, *graph template*, *solution mappings* from RDF [7] and SPARQL [10]. We assume the existence of an infinite set $\mathcal{D}$ whose elements are *documents* and write **IRI** the set of all IRIs, **V** the set of query variables, $\mathcal{G}$ the set of all RDF graphs.

Following the diagram of Fig. 3, we can abstract a *design document* as a pair $\langle \mathbf{CM},\mathbf{NM} \rangle$, where **CM** is a set of `ContainerMap`s and **NM** is a set of `NonContainerMap`s. A `NonContainerMap` is a pair $\langle u_{\mathsf{nm}},\mathbf{RM} \rangle$ where u_{nm} is an IRI and **RM** is a set of `ResourceMap`s. A `ContainerMap` is a tuple $\langle u_{\mathsf{cm}},\mathbf{RM},\mathbf{CM},\mathbf{NM} \rangle$ where u_{cm} is an IRI, **RM** is a set of `ResourceMap`s, **CM** is a set of `ContainerMap`s, and **NM** is a set of `NonContainerMap`s. A `ResourceMap` is a tuple $\langle u_{\mathsf{rm}},\mathsf{qp},\mathsf{cq},\mathbf{DS} \rangle$ where u_{rm} is an IRI, qp is a SPARQL query pattern, cq is a `CONSTRUCT` query, and **DS** is a set of `DataSource`s. There are several ways of describing a `DataSource` that our concrete language covers (see details in the language specification [3]). Here, we only consider the cases of a pair $\langle u_{\mathsf{ds}},u_{\mathsf{loc}} \rangle$ or a triple $\langle u_{\mathsf{ds}},u_{\mathsf{loc}},u_{\mathsf{lr}} \rangle$ where u_{ds}, u_{loc} and u_{lr} are IRIs that respectively refer to a data source, its location and an RDF lifting rule.

As we can see, all components of a design document in LDP-DL have an IRI. Given a $*$`Map` or `DataSource` x, we refer to the IRI of x as $\mathsf{iri}(x)$. In a `ResourceMap`, qp is used to extract a set of related resources from `DataSource`s, and cq is used to generate the graph of the LDP-RSs associated with the related resources. In a `DataSource`, u_{loc} corresponds to the location of the source file, whereas u_{lr} is the location of what we call a *lifting rule*, used to generate an RDF graph from non-RDF data.

We assume the existence of an infinite set of variables $V_r = \{\rho,\nu,\pi_1,...,\pi_i,...\} \subseteq \mathbf{V}$ called the *reserved variables*, such that $\mathbf{V} \setminus V_r$ is infinite. `ResourceMap`s may use the reserved variables but these have a special semantics as explained in the next section. However, due to undesirable consequences, we forbid the use of variable ν in the `WHERE` clause of the `CONSTRUCT` query cq.

Fig. 4 shows a simple example of a design document[8] in the abstract syntax of the language. An arrow with the label *cm*, *nm* or *rm* indicates that the construct

[8] Design document in concrete syntax `https://tinyurl.com/y8n9cls2`

has a `ContainerMap`, `NonContainerMap` or `ResourceMap` in its **CM**, **NM** or **RM** respectively. Also, though not shown in the figure, in the **DS** of all `ResourceMap`s, there is a `DataSource` (`ex:ds`,`ex:paris`) which is actually the RDF graph in Fig. 2(a).

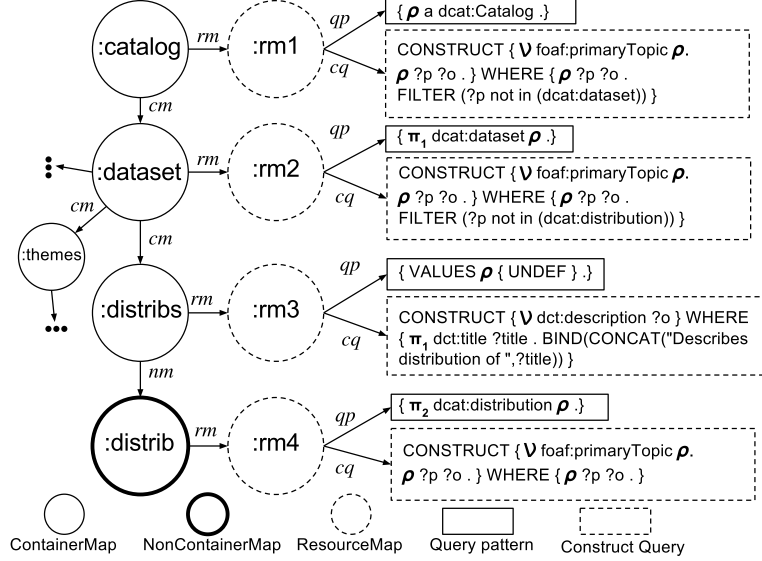

Fig. 4: Example of an LDP-DL document in the abstract syntax.

3.4 Overview of the Formal Semantics

The aim of the formal semantics is to associate an LDP dataset (as described in Sec. 3.2) to a design document. To this end, we define a notion of interpretation and a notion of satisfaction in a model-theoretic way. Several interpretations may satisfy a given design document, leading to different evaluations of it. This approach allows developers to implement the language in different ways, leading to different results, depending on whether they implement optional or alternative parts of the standard, yet have a non-ambiguous way to check the correctness of an implementation output.

LDP-DL interpretation. An LDP-DL interpretation determines which IRIs denote `ContainerMap`s, `NonContainerMap`s, `ResourceMap`s, `DataSource`s, or something else. Then, each `ContainerMap` (resp. `NonContainerMap`) is interpreted as a set of triples $(url,graph,M)$ (resp., a set of pairs $(url,graph)$) wrt a list of ancestors. A list of ancestors is a finite sequence of elements that can be IRIs or a special value $\epsilon \notin \mathbf{IRI}$ that indicates an absence of related resource. Formally, an *ancestor list* is an element of $\mathbf{IRI}^* = \bigcup_{n>=0} (\mathbf{IRI} \cup \{\epsilon\})^n$ and $\emptyset$ being the empty list $(\mathbf{IRI} \cup \{\epsilon\})^0$. We use the notation $\vec{p}$ to denote an ancestor list and use $\mathrm{len}(\vec{p})$ to denote the length of the list. Also $\vec{p}::r$ denotes appending element r to $\vec{p}$.

Definition 1 (LDP-DL Interpretation). *An* LDP-DL interpretation $\mathcal{I}$ *is a tuple* $\langle \Delta^{\mathcal{I}}, \mathcal{C}, \mathcal{N}, \mathcal{R}, \mathcal{S}, \cdot^{\mathcal{I}}, \mathcal{I}_{\mathcal{C}}, \mathcal{I}_{\mathcal{N}}, \mathcal{I}_{\mathcal{R}}, \mathcal{I}_{\mathcal{S}} \rangle$ *such that:*

- $\Delta^{\mathcal{I}}$ *is a non empty set (the* domain *of interpretation);*
- $\mathcal{C}$*,* $\mathcal{N}$*,* $\mathcal{R}$*,* $\mathcal{S}$ *are subsets of* $\Delta^{\mathcal{I}}$*;*
- $\cdot^{\mathcal{I}}:\mathbf{IRI}\to\Delta^{\mathcal{I}}$ *is the* interpretation function*;*
- $\mathcal{I}_{\mathcal{C}}:\mathcal{C}\times\mathbf{IRI}^*\to 2^{\mathbf{IRI}\times\mathcal{G}\times 2^{\mathbf{IRI}}}$*;*
- $\mathcal{I}_{\mathcal{N}}:\mathcal{N}\times\mathbf{IRI}^*\to 2^{\mathbf{IRI}\times\mathcal{G}}$*;*
- $\mathcal{I}_{\mathcal{R}}:\mathcal{R}\times\mathbf{IRI}^*\to 2^{\mathbf{IRI}\times\mathbf{IRI}\cup\{\epsilon\}\times\mathcal{G}}$ *such that* $(n,r_1,g_1)\in\mathcal{I}_{\mathcal{R}}(u_1,\overrightarrow{p_1})\wedge(n,r_2,g_2)\in\mathcal{I}_{\mathcal{R}}(u_2,\overrightarrow{p_2})\implies r_1=r_2\wedge g_1=g_2$ *(unicity constraint);*
- $\mathcal{I}_{\mathcal{S}}:\mathcal{S}\to\mathcal{G}$.

$\mathcal{C}$ (resp. $\mathcal{N}$, $\mathcal{R}$, $\mathcal{S}$) represents the container maps (resp., non container maps, resource maps, data sources) according to the interpretation. That is, if the interpretation function $\mathcal{I}$ maps an IRI to an element of $\mathcal{C}$, it means that this interpretation considers that the IRI is the name of a container map. For a given `ContainerMap` $cm\in\mathcal{C}$ and an ancestor list $\overrightarrow{p}$, $\langle n,g,M\rangle\in\mathcal{I}_{\mathcal{C}}(cm,\overrightarrow{p})$ means that, in the context of $\overrightarrow{p}$, cm must map the data sources to containers where n is the IRI of a container, g is the RDF graph obtained from dereferencing n, and M is the set of IRIs referring to the members of the container. Similarly, for a `NonContainerMap` $nm\in\mathcal{N}$, $\langle n,g\rangle\in\mathcal{I}_{\mathcal{N}}(nm,\overrightarrow{p})$ means that nm must map to resources where n is the IRI of a non-container LDP-RS that provides g when dereferenced. For a `DataSource` $ds\in\mathcal{S}$, $\mathcal{I}_{\mathcal{S}}(ds)$ is an RDF graph representing what can be obtained from the data source.

Informal description of the satisfaction. We describe satisfaction $\models$ relating interpretations to syntactic constructs that they validate. Due to space restriction, we only provide an overview of the definitions, that are formally given in our technical report [4] with more explanations. The rest of this section informally explains the semantics of LDP-DL constructs. To do so, we use Fig. 4, which is a design in LDP-DL for building the LDP[9] having the structure shown in Fig. 2(b), using the data source in Fig. 2(a).

In principle, a `DataSource` provides information to retrieve an RDF graph, using parameters that can take several forms. Here, we define only two forms of `DataSource`s, $ds=\langle u_{\mathsf{ds}},u_{\mathsf{loc}}\rangle$ that provides a URL to an RDF document directly, and $ds=\langle u_{\mathsf{ds}},u_{\mathsf{loc}},u_{\mathsf{lr}}\rangle$ that provides an additional URL to an arbitrary document with a transformation script to generate an RDF graph. We call such a script a *lifting rule* and can be seen as a function $lr:\mathcal{D}\to\mathcal{G}$. Our semantics is flexible enough to be extended to more complex such as for access rights, content negotiation etc. For example, the retrieval of the RDF graph in Fig. 2(a) could be described by a `DataSource` $ds_1=\langle u_{ds_1},u_{loc_1}\rangle$ where $\mathcal{I}_{\mathcal{S}}(u_{ds1}^{\mathcal{I}})$ is the RDF graph located at u_{loc_1}. Similarly, for a `DataSource` $ds_2=\langle u_{ds_2},u_{loc_2},u_{lr_2}\rangle$, $\mathcal{I}_{\mathcal{S}}(u_{ds1}^{\mathcal{I}})$ is the RDF graph obtained by executing the lifting rule found at u_{lr_2} on the document found at u_{loc_2}.

At the top level of the design document, the `ContainerMap` `:catalog` uses the `ResourceMap` `:rm1` to generate the top level containers. The `DataSource` used by `:rm1` is interpreted as the RDF graph of Fig. 2(a). At this level, `:rm1` is evaluated with an empty ancestor list. Using its query pattern, related resources extracted from the source are DCAT catalogs `ex:paris-catalog` and `ex:toulouse-catalog`. For each of them, an IRI is generated, namely `dex:paris-catalog` and `dex:toulouse-catalog`. Also, to satisfy the map `:rm1`, the RDF graph associated with the container IRI is obtained using its `CONSTRUCT` query, where

[9] `http://opensensingcity.emse.fr/ldpdfend/catalogs/ldp`

the variable ρ is bound to the related resource IRI, and ν to the IRI of the new LDP resource. For example, when doing so for dex:paris-catalog, ρ is bound to ex:paris-catalog, and ν to dex:paris-catalog. Finally, new containers generated from :catalog must define their members as well, and is thus satisfied only if its members correspond to the resources generated by their underlying ContainerMaps and NonContainerMaps (in this case, :dataset only).

The ContainerMap :dataset is used to generate members for containers generated from :catalog. Let us consider the case for dex:paris-catalog. Its related resource is ex:paris-catalog and its members must only have related resources that are DCAT datasets from this catalog. This is why the extraction of these resources is parameterized by the parent variable π_1 in the query pattern :rm2. π_1 is binded to the first element of the ancestor list which at this stage is ex:paris-catalog. The evaluation of :dataset generates two containers, dex:parking and dex:busStation, that are added to the members of dex:paris-catalog.

The map :distribs is used to generate members for containers generated by :dataset. Consider the case of doing so for dex:parking whose related resource is ex:parking. In this context, the aim of using :distribs is to generate a container to describe the set of distributions of ex:parking. Note that in the data source, there is no explicit resource to describe this set. This is why, in the ResourceMap :rm3, the query pattern returns a single result where ρ is unbound. Although the query pattern does not use any ancestor variable, it is evaluated using the ancestor list (ex:paris-catalog,ex:parking) and thus ancestor variables π_1 and π_2 are bound. The evaluation of :distribs in the context of dex:parking generates a single container dex:pDistributions. :distribs is satisfied when a single container is generated without a related resource.

Finally, the NonContainerMap :distrib is used to generate non-containers for each distribution of a DCAT dataset. Consider the case of doing so for dex:pDistributions with ancestor list (ex:paris-catalog,ex:parking,$\emptyset$). In this context, the proper related resource that must be used to extract the relevant distributions is associated with the grand parent container. This is why the query pattern of :rm4 uses π_2, bound to ex:parking, rather than π_1. Using the result (ex:pJSON and ex:pCSV) of this query pattern, two non-containers dex:pJSON and dex:pCSV in dex:pDistributions are generated using :distrib. In general, any ancestor's related resources can be referenced through the ancestor variables π_i simultaneously, even when they are unbound.

Evaluation of a design document using an interpretation. With an interpretation, we have a way of assigning an LDP dataset (as described in Sec. 3.2) to a design document, using the interpretations of the ContainerMaps and NonContainerMaps that appear in the document. We call this an *evaluation* of the document. Formally, it takes the form of a function that builds an LDP dataset given an LDP-DL interpretation and a document δ. We formalize the notion of LDP dataset as follows:

Definition 2 (LDP dataset). *An* LDP dataset *is a pair* $\langle \mathbf{NG},\mathbf{NC} \rangle$ *where* $\mathbf{NG}$ *is a set of named graphs and* $\mathbf{NC}$ *is a set of* named container*, that is a set of triples* $\langle n,g,M \rangle$ *such that* $n \in \mathbf{IRI}$ *(called the container name),* $g \in \mathcal{G}$ *and* $M \in 2^{\mathbf{IRI}}$*, and such that:*

- *no IRI appears more than once as a (graph or container) name;*

– *for all* $\langle n,g,M\rangle \in \mathbf{NC}$*, and for all* $u \in M$*, there exists a named graph or container having the name* u*.*

Having the notion of LDP dataset, the maps of the design document are evaluated wrt an ancestor list as follows:

Definition 3 (Evaluation of a map). *The* evaluation *of a* `ContainerMap` *or* `NonContainerMap` m *wrt an interpretation* $\mathcal{I}$ *and an ancestor list* $\vec{p}$ *s.t.* $\mathcal{I},\vec{p} \models m$ *is:*

$$[[m]]_{\mathcal{I}}^{\vec{p}} = \begin{cases} \mathcal{I}_{\mathcal{N}}(\mathsf{iri}(m)^{\mathcal{I}},\vec{p}), & \text{if } m \text{ is a } \texttt{NonContainerMap} \\ \\ \mathcal{I}_{\mathcal{C}}(\mathsf{iri}(m)^{\mathcal{I}},\vec{p}) \cup \bigcup\limits_{\substack{rm \in \mathbf{RM} \\ \langle n,r,g\rangle \in \mathcal{I}_{\mathcal{R}}(\mathsf{iri}(rm)^{\mathcal{I}},\vec{p}) \\ m' \in \mathbf{NM} \cup \mathbf{CM}}} [[m']]_{\mathcal{I}}^{\vec{p}::r}, & \text{if } m = \langle u_{\mathsf{cm}},\mathbf{RM},\mathbf{CM},\mathbf{NM}\rangle \text{ is a } \texttt{ContainerMap} \end{cases}$$

The evaluation of a map yields an LDP dataset. Indeed, the first condition of Def. 2 is satisfied because of the unicity constraint from Def. 1, and the second condition is satisfied because $\mathcal{I},\vec{p} \models m$. Now we can define the evaluation of a design document wrt an interpretation:

Definition 4 (Evaluation of a design document). *Let* $\mathcal{I}$ *be an interpretation and* $\delta = \langle \mathbf{CM},\mathbf{NM}\rangle$ *a design document. The* evaluation *of* δ *wrt* $\mathcal{I}$ *is*

$$[[\delta]]_{\mathcal{I}} = \bigcup_{m \in \mathbf{CM} \cup \mathbf{NM}} [[m]]_{\mathcal{I}}^{\emptyset}$$

In practice, an LDP-DL processor will not define an explicit interpretation, but will build an LDP dataset from a design document. Hence, we want to define a notion of conformity of an algorithm wrt the language specification given above. To this aim, we first provide a definition of a valid LDP dataset for a design document.

Definition 5 (Valid). *An LDP dataset* D *is* valid *wrt a design document* δ *if there exists an interpretation* $\mathcal{I}$ *that satisfies* δ*, such that* $[[\delta]]_{\mathcal{I}} = D$*.*

The validity of LDP dataset is an important property used in our implementation when generating and deploying LDPs. Finally, we can define the correctness of an algorithm that implements LDP-DL.

Definition 6 (Correct). *An algorithm that takes an LDP-DL document as input and returns an LDP dataset is correct if for all document* δ*, the output of the algorithm on* δ *is valid wrt* δ*.*

If an algorithm evaluates LDP-DL documents, the formal semantics given in [4] should be used to prove its correctness.

4 Implementation and Evaluation

This section describes the implementation of the tools participating in the LDP generation workflow described in Sec. 2.3. Then, we evaluate our approach with respect to our requirements outlined in Sec. 2.1 by performing experiments to deploy real datasets on LDPs. A detailed description of the tools and the experiments we conducted are available on GitHub[10].

[10] `https://github.com/noorbakerally/LDPDatasetExamples`

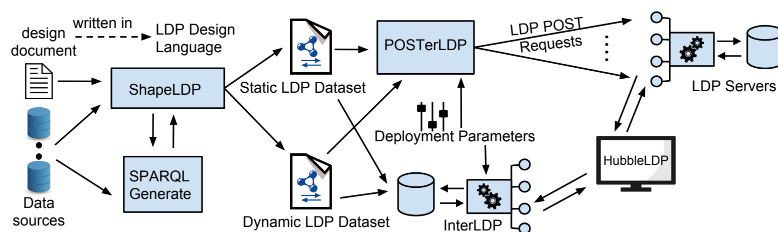

Fig. 5: Implementation of our LDP Generation Workflow proposal

4.1 Implementation of LDP Generation Workflow

ShapeLDP [11] is an LDPizer that interprets design documents and generate the LDP dataset from them. The concrete syntax in RDF is described in [3] and the algorithms we use are described in [4]. To enhance modularity, the design model may be split and written in different documents that are merged before processing. To exploit heterogeneous data sources, for now it can use lifting rules specified for `DataSource`s in SPARQL-Generate [11]. Future versions may consider other languages such as RML [8], XSPARQL [1] and others. ShapeLDP can process design documents in two modes: *static evaluation* and *dynamic evaluation*. In both modes, the output uses relative IRIs without an explicit base and act only as an intermediary document from which the LDP dataset should be constructed. In *static evaluation* (resp. *dynamic evaluation*), a *static LDP dataset* (resp. *dynamic LDP dataset*) is generated in TriG format [6]. The final LDP dataset is obtained from a static LDP dataset by adding a base IRI at deployment time. The result stays valid wrt to the sources as long as the evaluation of the query patterns and `CONSTRUCT` queries of all `ResourceMap`s from the design document do not change. The LDP dataset obtained from a dynamic LDP dataset stays valid wrt the sources as long as the evaluation of the quert patterns of all `ResourceMap`s do not change, but the result of `CONSTRUCT` queries may change from one request to the next. A *dynamic LDP dataset* is a structure somewhat similar to the LDP dataset that store all containers and non-containers and a `CONSTRUCT` query to generate their RDF graphs at request time. The `CONSTRUCT` query is obtained when evaluating `ResourceMap`s by using the bindings of the reserved variables.

InterLDP [12] is an LDP server which can directly consume an LDP dataset (static or dynamic) and expose resources from it. It was validated against the conformance tests of the LDP read interactions.[13] To cater for hosting constraints, it uses the dynamic LDP dataset to generate the RDF graph of the requested resource at query time.

POSTerLDP [14] is the implementation of the LDP deployer on existing implementations of the LDP standard. It consumes a static LDP dataset and deployment parameters: base

[11] `https://github.com/noorbakerally/ShapeLDP`
[12] `https://github.com/noorbakerally/InterLDP`
[13] The conformance report is available at `https://w3id.org/ldpdl/InterLDP/execution-report.html`.
[14] `https://github.com/noorbakerally/POSTerLDP`

URL of LDP server and optionally the username and password for basic authentication on the server. Currently, it deploys resources only on one server at a time but future versions may consider replication or partitioning schemes described in a particular deployment language. Moreover, it is independent of any particular LDP implementations. It generates standard LDP requests and thus, it is compatible with any LDP server. Finally, we provide a browser, *HubbleLDP*[15] which can be used to browse resources on an LDP and view their content. An instance of it is running at `http://bit.ly/2BGYl9X`.

4.2 Evaluation

We evaluate our approach by performing the following five experiments, that show the main requirements they satisfy.

Experiment 1: The input data sources correspond to 22 RDF datasets from city data portals, structured per the DCAT vocabulary. Five design documents have been defined and used to automatize the generation of 110 LDPs from the 22 datasets. This demonstrate the *Reusability* requirement through the use of the same design document on all input data sources. It also shows the flexibility as we are able to generate different platforms by applying different design documents on the same data source.

Experiment 2: The aim is to show the compatibility of our approach with existing implementations. To that aim, an LDP dataset is automatically generated from one input data source and the design document from Experiment 1. We use POSTerLDP to automatize its deployment over LDP servers that are instances of Apache Marmotta and Gold, both of them being referenced implementation. This experiment shows also the loose coupling of our approach as we are able to reproduce the two platforms with the same design on different LDP servers.

Experiment 3: To show the ability of our approach to consider the *Heterogeneity* requirement, a design document is used with a lifting rule to deal with original data sources, one in CSV and the other in JSON format. ShapeLDP uses the embedded SPARQL-Generate engine and the lifting rules to automatically generate the respective RDF data that is then deployed via an LDP.

Experiment 4: To evaluate satisfaction of the *Hosting Constraints* requirement, we use ShapeLDP in dynamic evaluation mode: a dynamic LDP dataset is generated first, from which we can deploy an LDP that offers dynamic results generated at request time from real-time data sources. Generating responses from the platform takes more time because their content are generated at query time.

Experiment 5: For now, we do not automatically generate design models, but we provide 2 generic design documents that organize data according to the class hierarchy of an ontology. We believe that this is a typical design that many data providers would choose, such that they would not have to rewrite the design document. In this experiment, we deploy LDPs using the generic designs from all data sources used in Experiment 1. For now, the generic designs can be reused on all RDFS/OWL vocabularies that do not have cycles in the class hierarchy.

The results of the above experiments and the requirements they fulfilled is shown in the Table.1.

[15] `https://github.com/noorbakerally/HubbleLDP`

	Requirements			
	Heterogeneity	*Hosting Constraints*	*Reusability*	*Automatization*
Experiment 1			✓	✓
Experiment 2				✓
Experiment 3	✓			✓
Experiment 4	✓	✓		✓
Experiment 5			✓	✓

Table 1: Summary of experiments and requirements fulfilled

Contribution to existing implementations: Our approach benefits both categories of LDP implementation described in Sec. 2.2. Using our tools, an LDP dataset can be generated from static, dynamic or heterogenous data sources from a custom-written design or a generic design we provided. In the case of LDP management system, it is possible to deploy the LDP dataset via an existing LDP implementation using POSTerLDP. With LDP frameworks, it is possible to develop complex applications using the LDP dataset (static or dynamic) and thus avoiding boilerplate code for retrieving data resources from their sources and transforming them into LDP resources.

Performance test: Finally, a simple performance test was carried on ShapeLDP using one design document in Experiment 1. Random DCAT datasets were generated with a maximum size of one million triples and were used as input data sources. We find that the performance is approximately linear[16]. However, more tests have to be made in this regard using different types of designs. The generation of the datasets, all scripts and the results are given on the GitHub page with all explanations.

5 Conclusion and Future Work

Linked Data Platforms can potentially ease the work of data consumers, but there is not much support from implementations to automate the generation and deployment of LDPs. Considering this, we proposed a language for describing parts of the design of an LDP. Such a description can be processed to generate and deploy LDPs from existing data sources regardless of the underlying implementation of the LDP server. We demonstrated the flexibility, effectiveness, and reusability of the approach to cope with heterogeneity and dynamicity of data sources.

For now, LDP-DL is restricted only to some design aspects. Yet, due to its flexibility and some original features, it can fulfill the requirements that we identified. We intend to consider other aspects in our future work. First, we want to support non-RDF sources and other types of LDP containers. Second, we want to generate LDPs that supports paging [19], which is a desired feature for large datasets. Third, we want to extend LDP-DL to allow the description of deployment design, security design, transaction model, etc. Our long term objective is to have a complete design language for LDPs. Finally, from a theoretical perspective, we want to analyze formal properties of the language, such as design compatibility, design containment, design merge, parallelizability, and so on, based on the formal semantics.

[16] https://github.com/noorbakerally/PerformanceTestShapeLDP

Acknowledgments This work is supported by grant ANR-14-CE24-0029 from *Agence Nationale de la Recherche* for project OpenSensingCity. We are thankful to the reviewers who helped very much improving this paper.

References

1. W. Akhtar, J. Kopeckỳ, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF worlds–and avoiding the XSLT pilgrimage. In *ESWC*, 2008.
2. S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th international conference WWW*. ACM, 2009.
3. N. Bakerally. LDP-DL: RDF Syntax and Mapping to Abstract Syntax. Technical report, Mines Saint-Étienne, 2018. `https://w3id.org/ldpdl`.
4. N. Bakerally, A. Zimmermann, and O. Boissier. LDP-DL: A language to define the design of Linked Data Platforms. Technical report, Mines Saint-Étienne, 2018. `http://w3id.org/ldpdl/technical_report.pdf`.
5. C. Bizer and R. Cyganiak. D2R server-publishing relational databases on the semantic web. In *Poster at the 5th ISWC*, volume 175, 2006.
6. G. Carothers and A. Seaborne. RDF 1.1 TriG, RDF Dataset Language, W3C Recommendation 25 February 2014. Technical report, W3C, 2014.
7. R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. Technical report, W3C, 2014.
8. A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW*, 2014.
9. R. B. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, 2007.
10. S. Harris and A. Seaborne. SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013. Technical report, W3C, 2013.
11. M. Lefrançois, A. Zimmermann, and N. Bakerally. A SPARQL extension for generating RDF from heterogeneous formats. In *ESWC*, 2017.
12. G. Loseto, S. Ieva, F. Gramegna, M. Ruta, F. Scioscia, and E. Sciascio. Linking the Web of Things: LDP-CoAP Mapping. In *ANT/SEIT Workshops*, 2016.
13. F. Maali and J. Erickson. Data Catalog Vocabulary (DCAT), W3C Recommendation 16 January 2014. Technical report, W3C, 2014.
14. N. Mihindukulasooriya, R. Garcia-Castro, and M. E. Gutiérrez. Linked data platform as a novel approach for enterprise application integration. In *COLD*, 2013.
15. N. Mihindukulasooriya, M. E. Gutiérrez, and R. García-Castro. A linked data platform adapter for the bugzilla issue tracker. In *ISWC Posters & Demo*, pages 89–92, 2014.
16. N. Mihindukulasooriya, F. Priyatna, Ó. Corcho, R. García-Castro, and M. E. Gutiérrez. morph-LDP: An R2RML-Based Linked Data Platform Implementation. In *ESWC Poster & Demo*, 2014.
17. S. Speicher, J. Arwe, and A. Malhotra. Linked Data Platform 1.0. Technical report, W3C, February 26 2015.
18. S. Speicher, J. Arwe, and A. Malhotra. Linked Data Platform 1.0, W3C Recommendation 26 February 2015. Technical report, W3C, 2015.
19. S. Speicher, J. Arwe, and A. Malhotra. Linked Data Platform Paging 1.0 W3C Working Group Note 30 June 2015. Technical report, W3C, 2015.
20. T. Stahl, M. Volter, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management.* Pitman, 2006.