Vocabulary for Linked Data Visualization Model*

Jakub Klímek[1] and Jiří Helmich[2]

[1] Czech Technical University in Prague, Faculty of Information Technology
Thákurova 9, 160 00 Praha 6, Czech Republic
klimek@fit.cvut.cz
[2] Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
helmich@ksi.mff.cuni.cz

Abstract. There is already a vast amount of Linked Data on the web. What is missing is a convenient way of analyzing and visualizing the data that would benefit from the Linked Data principles. In our previous work we introduced the Linked Data Visualization Model (LDVM). It is a formal base that exploits the principles to ensure interoperability and compatibility of compliant components. In this paper we introduce a vocabulary for description of the components and an analytic and visualization pipeline composed of them. We demonstrate its viability on an example from the Czech Linked Open Data cloud.

Keywords: Linked Data, RDF, visualization, vocabulary

1 Introduction

Vast amount of data represented in a form of Linked Open Data (LOD) is now available on the Web. Unfortunately, not so many users are capable of using the data in a useful way yet. The data is represented in RDF and often uses commonly known vocabularies, which brings opportunities for data analysis and visualization that were not there before. However, the appropriate tools that would exploit these new benefits are still lacking.

Figure 1 shows datasets transformed to Linked Data by our research group over the past few years. The circles are the individual datasets and the edges mean there is a decent amount of links among entities of the two datasets. This gives the users some very rough ideas of what they can find in those datasets. Each dataset should also be described by its metadata, which gives more information about what is inside. However, the Linked Data principles offer more. For each of our datasets a SPARQL endpoint – an open endpoint to a database where everyone can place a structured query - is available. This in combination with commonly used Linked Data vocabularies means that anyone can simply ask whether a particular dataset contains interesting data. The obvious issue here is that non expert users do not know SPARQL so they do not know how to ask the right question. For example, if a user is interested in opening hours of a particular institution of public power, he could query the appropriate dataset that

* This work was partially supported by a grant from the European Union's 7th Framework Programme number 611358 provided for the project COMSODE

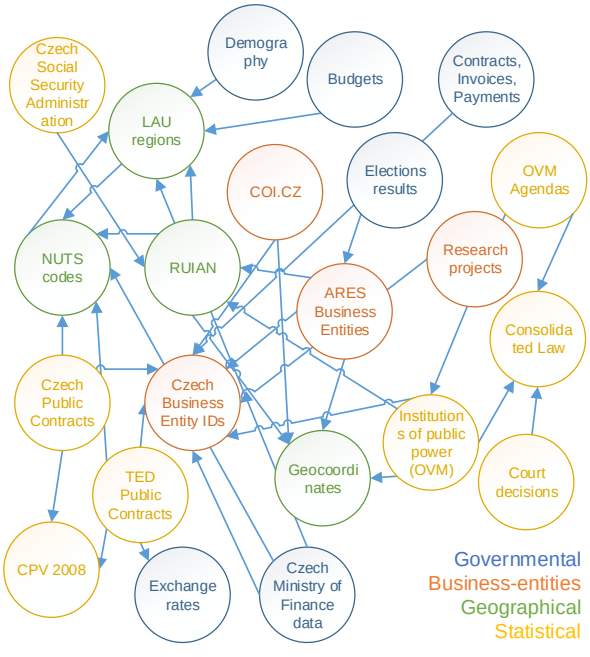

Fig. 1. Czech Linked Open Data Cloud

he sees in our Czech LOD cloud, if he knew how. This situation is somehow similar to programming and common algorithms. For every programming language there are libraries of algorithms implemented by experts who know how to do that, packaged for use by people who do not. Because on the Web of Data there are vocabularies that are de facto standards for representation of certain types of data such as opening hours of locations in general[3], if the data is in the dataset, it would be found by a general query suited for this, perhaps written by an expert. This means that a regular user could find our dataset of institutions of public power and assume that it uses the standard vocabulary. Then he could use the query from a library of queries suited for common tasks using the common vocabularies and execute this query on a dataset of his choosing. He would get the result, possibly even displayed in a user friendly way, again thanks to the standard vocabularies and all this without understanding SPARQL, RDF, Linked Data, etc.

Previously, we introduced the *Linked Data Visualization Model* (LDVM) [4], which allows users to create and reuse analytic and visualization components that leverage the Linked Data principles. We also showed a tool *Payola* [6] implementing the model and in [7] we demonstrated that expert users can prepare analyses and visualizations and allow non-experts to use them to get data from the LOD cloud[4].

[3] `http://www.heppnetz.de/ontologies/goodrelations/v1.html#OpeningHoursSpecification`
[4] `http://lod-cloud.net/`

In this paper we introduce the LDVM vocabulary, which allows LDVM implementations to store and exchange configuration of individual LDVM components as well as whole analytic and visualization pipelines in RDF and in compliance with the Linked Data principles. The vocabulary contains support for pipeline nesting so that complete pipelines created by experts can be wrapped as another component to be used in pipelines by non-experts. By publishing the vocabulary we also open our approach to Linked Data analysis and visualization so that anyone who is interested can easily create a reusable component or pipeline and share it with others. The technical benefits are easy sharing, open format easily adoptable by other implementations, easier management of configurations – all the configurations can be maintained by SPARQL queries and the possibility to configure the components and pipelines programmatically. In addition, there are all the generally known Linked Data benefits such as ability to better provide context through linking to other sources, better provenance tracking, etc.

This paper is structured as follows. In section 2 we briefly describe the principles of LDVM. In section 3 we introduce the LDVM vocabulary, which is the main contribution of this paper. In section 4 we show the usage of the vocabulary on examples. In section 5 we survey related work and in section 6 we conclude.

2 Linked Data Visualization Model

In our previous work we defined the Linked Data Visualization Model (LDVM), an abstract visualization process customized for the specifics of Linked Data. In short, LDVM allows users to create data visualization pipelines that consist of four stages: Source Data, Analytical Abstraction, Visualization Abstraction and View. The aim of LDVM is to provide means of creating reusable components at each stage that can be put together to create a pipeline even by non-expert users who do not know RDF. The idea is to let expert users to create the components by configuring generic ones with proper SPARQL queries and vocabulary transformations. In addition, the components are configured in a way that allows the LDVM implementation to automatically check whether two components are compatible or not. If two components are compatible, then the output of one can be connected to the input of the other in a meaningful way. With these components and the compatibility checking mechanism in place, the visualization pipelines can then be created by non-expert users.

2.1 Model Components

There are four stages of the visualization model populated by LDVM components. *Source Data* stage allows a user to define a custom transformation to prepare an arbitrary dataset for further stages, which require their input to be RDF. In this paper we only consider RDF data sources such as RDF files or SPARQL endpoints, e.g. DBPedia. The LDVM components at this stage are called *data sources*. The *Analytical Abstraction* stage enables the user to specify analytical operators that extract data to be processed from one or more data sources and then transform it to create the desired analysis. The transformation can also compute additional characteristics like aggregations. For example, we can query for resources of type `dbpedia-owl:City` and then compute the

number of cities in individual countries. The LDVM components at this stage are called *analyzers*. In the *Visualization Abstraction* stage of LDVM we need to prepare the data to be compatible with the desired visualization technique. We could have prepared the analytical abstraction in a way that is directly compatible with a visualizer. In that case, this step can be skipped. However, the typical use case for Visualization Abstraction is to facilitate reuse of existing analyzers and existing visualizers that work with similar data, only in different formats. For that we need to use a LDVM *transformer*. In *View* stage, data is passed to a *visualizer*, which creates a user-friendly visualization. The components, when connected together, create a analytic and visualization pipeline which, when executed, takes data from a source and transforms it to produce a visualization at the end. However, not every component can produce meaningful results from any input. Typically, each component is designed for a specific purpose, e.g. visualizing map data, and it does not make sense with other data. This means that only components that are somehow compatible can create a meaningful pipeline.

2.2 Component Compatibility

Now that we described the four basic types of LDVM components, let us take a look at the notion of their *compatibility*, which is a key feature of LDVM. We first introduced the idea of compatibility checking in [4] and then further refined it in [7]. However, as the implementation progressed we developed this feature even further.

The idea is based on the ability to check whether a component can work with the data it has on its input. We can check this using e.g. a SPARQL query, but we can do that only when the pipeline is already running, when we actually have the data to check. However, we want to use the component compatibility in design time to rule out component combinations that do not make any sense and to help the users to use the right components before they actually run the pipeline. Therefore, we need a way to check the compatibility without the actual data. For this, we use two constructs - an *input descriptor* and an *output data sample*. The input descriptor describes what is expected in the input data. For simplicity, let us use a set of SPARQL queries for the descriptor. A descriptor is bound to an input of its component.

In order to evaluate the descriptors in design time, we require that each LDVM component that produces data (data source, analyzer, transformer) also provides a static sample of the resulting data. For the data sample to be useful, it should be as small as possible, so that the input descriptors of other components execute as fast as possible. Also, it should contain the maximum amount of classes and properties whose instances can be produced by the component, making it as descriptive as possible. For example, when an analyzer transforms data about cities and their population, its output data sample will contain a representation of one city with all the properties that the component can possibly produce given it has all the inputs it needs. Note that, e.g. for data sources, it is also possible to implement the evaluation of descriptors over the output data sample as evaluation directly on the represented SPARQL endpoint.

Each LDVM component has a set of *features*, where each feature represents a part of the expected component functionality. A component feature can be either mandatory or optional. For example, a visualizer that displays points and their descriptions on a map can have 2 features. One feature represents the ability to display the points on a map.

This one will be mandatory, because without the points, the whole visualisation lacks purpose. The second feature will represent the ability to display a description for each point on the map. It will be optional, because when there is no data for the description, the visualization still makes sense - there are still points on a map. Whether a component feature can be used or not depends on whether there is the data needed for it on the input, therefore, each feature is described by a set of input descriptors.

We say that a feature of a component in a pipeline is *usable* when all queries in all descriptors are evaluated `true` on their respective inputs. A component is *compatible* with the mapping of outputs of other components to its inputs when all its mandatory features are usable. The usability of optional features can be further used to evaluate the expected quality of the output of the component. For simplicity, we do not elaborate on the output quality in this paper. The described mechanism of component compatibility can be used in design time for checking of validity of the visualization pipeline. It can also be used for suggestions of components that can be connected to a given component output. In addition, it can be used in run time for verification of the compatibility using the actual data that is passed through the pipeline. Finally, this concept can be also used for periodic checking of data source content, e.g. whether the data has changed its structure and therefore became unusable or requires pipeline change.

3 LDVM Vocabulary

In our current take on implementation of LDVM we aim to have individual components running as independent services that exchange only information needed to access the input and output data. Also we aim for easy configuration of individual components as well as easy configuration of the whole pipeline. In accordance with the Linked Data principles, we now use RDF as the format for storage and exchange of configuration so that any component that works with RDF can use LDVM components both individually and in a pipeline. For this purpose we have devised a vocabulary for LDVM, which is the main contribution of this paper. In Figure 2 there is a UML class diagram depicting the structure of the vocabulary. Boxes represent classes, edges represent object properties (links) and properties listed inside of the class boxes represent data properties. We chose the `ldvm`[5] prefix for the vocabulary, which is developed on GitHub[6].

3.1 Templates and Instances

There are blue and green classes. The blue classes belong to template level of the vocabulary and green classes belong to the instance level. The easiest way to imagine the division is to imagine a pipeline editor with a toolbox. In the toolbox, there are LDVM component templates with their default configuration. When a designer wants to use a LDVM component in a pipeline, he drags it onto the editor canvas, creating an instance. There can be multiple instances of the same LDVM component template in a single pipeline, each with configuration that overrides the default one. The template holds input descriptors and output data samples, which are used for the compatibility checking. The

[5] `http://linked.opendata.cz/ontology/ldvm/`
[6] `https://github.com/payola/ldvm`

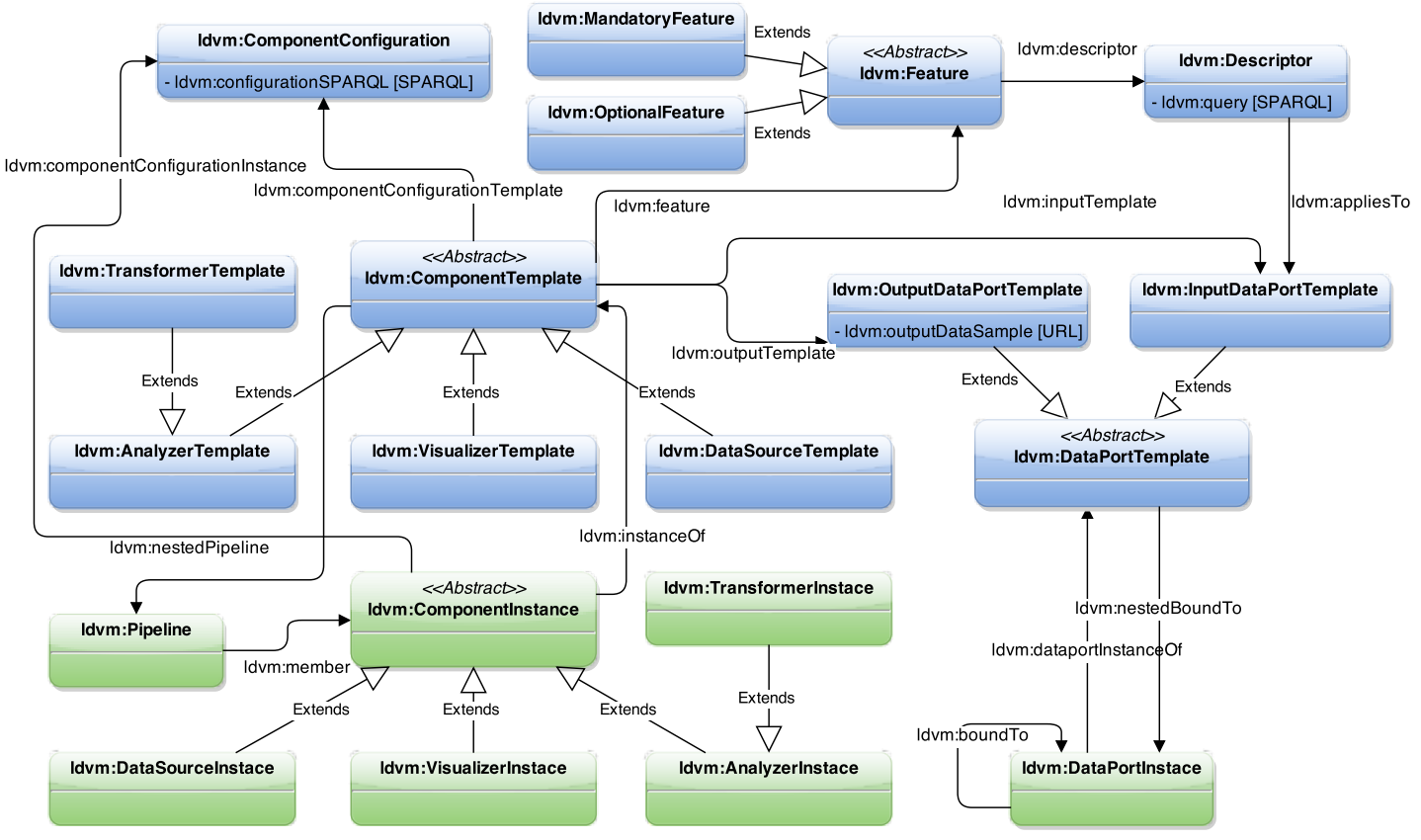

Fig. 2. LDVM Vocabulary

instance configuration and input and output mappings are then used for compatibility checking of a finished pipeline, which also depends on the content of the data sources. Also, they are used during pipeline execution to verify compatibility on the actual data. Each instance is connected to its template using the `ldvm:instanceOf` property.

3.2 Component Types

There are four basic component types as described in subsection 2.1 - data sources, analyzers, transformers and visualizers. They have their representation on both the template level - descendants of the `ldvm:ComponentTemplate` class - and instance levels - descendants of the `ldvm:ComponentInstance` class. From the implementation point of view, transformers are just analyzers with one input and one output, so the difference is purely semantic. This is why transformers are subclass of analyzers.

3.3 Data Ports

Components have input and output data ports. On the template level we distinguish the inputs and outputs of a component. To `ldvm:InputDataPortTemplate` the input descriptors of features can be applied. `ldvm:OutputDataPortTemplate` has the `ldvm:outputDataSample` links to the output data samples. Both are subclasses of `ldvm:DataPortTemplate`. The data ports are mapped to each other - output of one component to input of another - as instances of `ldvm:DataPortInstance` using the `ldvm:boundTo` property. This data port instance mapping forms the actual visualization pipeline, which can be then executed. Because data ports are not LDVM components, their instances are connected to their templates using a separate property `ldvm:dataPortInstanceOf`.

3.4 Features and Descriptors

On the template level, features and descriptors (see subsection 2.2) of a component are represented. Each component template can have multiple features connected using the `ldvm:feature` property. The features themselves - instances of either the `ldvm:MandatoryFeature` class or the `ldvm:OptionalFeature` class - can be described using standard Linked Data techniques and vocabularies such as `dcterms` and `skos`. Each feature can have descriptors, instances of `ldvm:Descriptor` connected using the `ldvm:descriptor` property. The descriptors have their actual SPARQL queries as literals connected using the `ldvm:query` property. In addition, the input data port templates to which the particular descriptor is applied are denoted using the `ldvm:appliesTo` property.

3.5 Configuration

Now that we have the LDVM components, we need to represent their configuration. On the template level, components have their default configuration connected using the `ldvm:componentConfigurationTemplate` property. On the instance level, components point to their configuration, when it is different from the default one, using the

`ldvm:componentConfigurationInstance` property. The configuration itself is the same whether it is on the template level or the instance level and therefore we do not distinguish the levels here and we only have one class `ldvm:ComponentConfiguration`.

The structure of the configuration of a LDVM component is completely dependent on what the component needs to function. It is also RDF data and it can use various vocabularies. It can be even linked to other datasets according to the Linked Data principles. Therefore it is not a trivial task to determine the boundaries of the configuration data in the RDF data graph in general. On the other hand, each component knows precisely what is expected in its configuration and in what format. This is why we need each component to provide a SPARQL query that can be used to obtain its configuration data so that the LDVM implementation can extract it. That SPARQL query is connected to every configuration using the mandatory `ldvm:configurationSPARQL` property.

3.6 Pipeline

Finally, the pipeline itself is represented by the `ldvm:Pipeline` class instance. It links to all the instances of LDVM components used in the pipeline.

3.7 Nested Pipelines

A key feature for collaboration of expert and non-expert users is pipeline nesting. An expert can create a pipeline that is potentially complex in number of components, their configuration and binding, but could be reused in other pipelines as a black box data source, analyzer or transformer. The intuitive way of achieving this goal is to let the expert to create the pipeline without a visualizer and potentially even without a data source. This pipeline would then create the black box with its own inputs represented by the missing input mappings of the inner pipeline and outputs represented by the outputs of the inner components to which no input is bound. However, there is one conceptual problem. This inner pipeline is made of component instances and we want to create a component template (reusable black box) out of it. For this, we need a property `ldvm:nestedPipeline` that indicates, that a pipeline is nested in the component template. In addition, we need to map the input data port templates of the new component template to be bound to the input data port instances of the components of the inner pipeline. Also, we need the output instances of the components of the inner pipeline to be bound to the output data port templates of the new component template. This is indicated by the `ldvm:nestedBoundTo` property.

4 Examples

In this section we will introduce examples of how actual templates and instances use LDVM. We use the Turtle RDF syntax[7] and due to space limitations we shorten full URLs and omit human readable labels in the data, which we otherwise recommend according to the "label everything" principle.

[7] `http://www.w3.org/TR/turtle/`

4.1 SPARQL Analyzer Template

In this section we show how a SPARQL analyzer component template uses LDVM vocabulary. See Listing 1.1.

```
1  a-sparql:SparqlAnalyzerConfiguration a rdfs:Class ;
2    rdfs:subClassOf ldvm:ComponentConfiguration .
3  a-sparql:query a rdf:Property ;
4    rdfs:domain a-sparql:SparqlAnalyzerConfiguration ;
5    rdfs:range xsd:string .
6  a-sparql-r:Configuration a a-sparql:SparqlAnalyzerConfiguration ;
7    a-sparql:query "CONSTRUCT {GRAPH ?g {?s ?p ?o}} WHERE {GRAPH ?g {?s ?p ?o}}" ;
8    ldvm:configurationSPARQL """
9      PREFIX a-sparql: <http://linked.opendata.cz/ontology/ldvm/analyzer/sparql/>
10
11     CONSTRUCT {
12       ?config a-sparql:query ?query;
13          dcterms:title ?title .
14     }
15     WHERE {
16       ?config a a a-sparql:SparqlAnalyzerConfiguration;
17       OPTIONAL {?config a-sparql:query ?query . }
18       OPTIONAL {?config dcterms:title ?title . }
19     }
20     """ .
21 a-sparql-r:Input a ldvm:InputDataPortTemplate .
22 a-sparql-r:Output a ldvm:OutputDataPortTemplate .
23 a-sparql-r:Descriptor a ldvm:Descriptor ;
24   ldvm:query """ASK {?s ?p ?o}""" ;
25   ldvm:appliesTo a-sparql-r:Input .
26 a-sparql-r:Feature a ldvm:MandatoryFeature ;
27   ldvm:descriptor a-sparql-r:Descriptor .
28 a-sparql-r:SparqlAnalyzerTemplate a ldvm:AnalyzerTemplate ;
29   ldvm:componentConfigurationTemplate a-sparql-r:Configuration ;
30   ldvm:inputTemplate a-sparql-r:Input ;
31   ldvm:outputTemplate a-sparql-r:Output ;
32   ldvm:feature a-sparql-r:Feature .
```

Listing 1.1. SPARQL Analyzer example

First, note that each LDVM component should define its own mini-vocabulary needed for its configuration. In the case of an analyzer that executes a SPARQL query, we need to configure the query. Therefore, we create a class representing the configuration of the SPARQL analyzer - see line 1 - a subclass of `ldvm:ComponentConfiguration`. Then we define the property to be used for the SPARQL query - see line 3 and we instantiate the the configuration as a default configuration - see line 6. Note the query that actually gets the whole configuration. This one would actually get every configuration of every SPARQL analyzer in the data. The LDVM implementation adds a special BIND clause that fixates the `?config` variable on the URI of the specific configuration. In addition, the component template has an input (line 21), output (line 22) and a mandatory feature (line 26) with its descriptor (line 23) that returns `true` whenever there is some RDF data on the input. Finally, we create the new component template itself on line 28.

4.2 Nested Pipeline Example

In this section we show how a nested pipeline instance can be wrapped into a new analyzer template. It is a simple pipeline of 3 analyzers where the first two take the input data from individual inputs, transform it and pass it to the third one. The third one merges the data and passes it to the output.

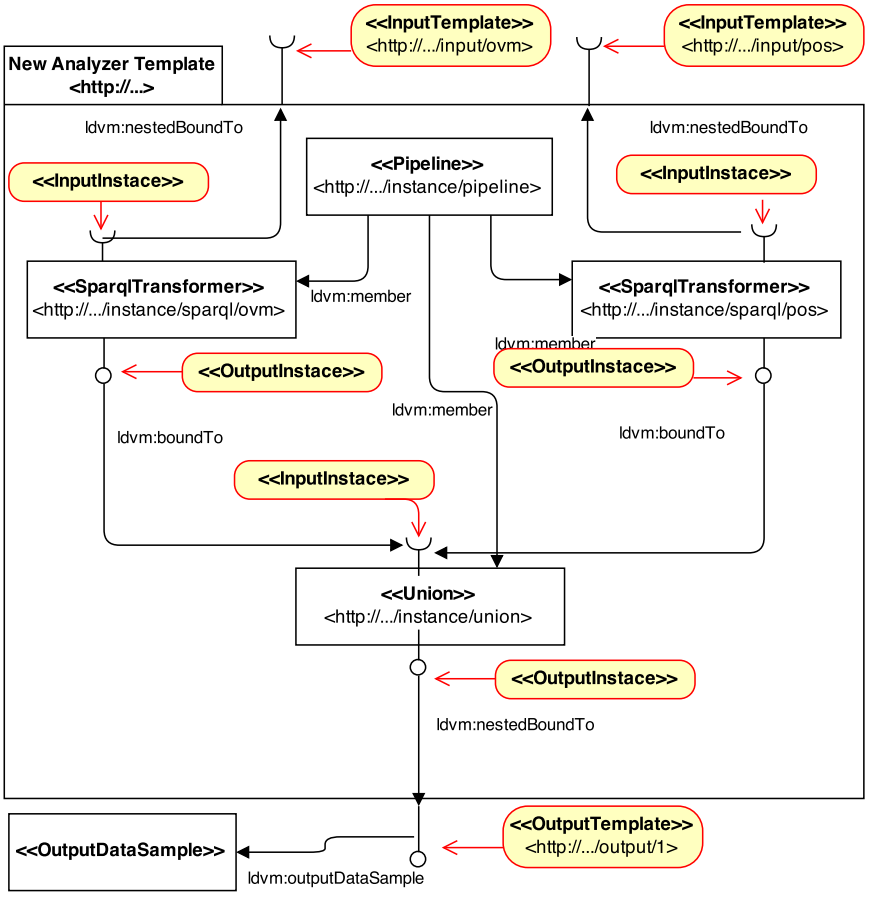

Fig. 3. Analyzer template containing nested pipeline

See Figure 3 where we chose a graphical representation rather than a textual one where boxes are entities, the class of the entities is written in bold and the actual URI of the entity is shortened. The new analyzer template has two inputs and one output and contains the nested pipeline. There is a link to a new output data sample from the output. There are two instances of the SPARQL analyzer template (see subsection 4.1),

which transform the data from the individual inputs, their inputs are bound to them using the `ldvm:nestedBoundTo` property. The third member of the pipeline has one input bound to the output of the SPARQL analyzers and one output bound to the output of the template itself.

At the same time, Figure 3 is an example of a very simple pipeline instance, which is the one nested in the new component template. What is missing due to lack of space is the instance configuration of a component, which can overwrite the one specified at template level. The configuration itself, however, looks the same at both levels and depends completely on the component being configured.

5 Related Work

The problem of Linked Data not being accessible to non-experts is well-known. With the LDVM Vocabulary we aim at an open web-services like environment that is independent of the specific implementation of the LDVM components. This of course requires proper definition of interfaces and the LDVM vocabulary is the base for that. However, the approaches so far usually aim at a closed browser environment that is able to analyze and visualize the Linked Data Cloud similarly to our first version of Payola [6]. They do not provide configuration and description using a reusable vocabulary. The approaches include *Hide the stack* [5], where the authors describe a browser meant for end-users, which is based on templates based on SPARQL queries. Also recent is *LDVizWiz* [1], which is a very LDVM-like approach to detecting categories of data in SPARQL endpoints and extracting basic information about entities in those categories. An lightweight application of LDVM in enterprise is described in LinDa [9]. Yet another similar approach that analyzes SPARQL endpoints to generate faceted browsers is *rdf:SynopsViz* [3]. In [2] the authors use their *LODeX* tool to summarize LOD datasets according to the vocabularies used. For more tools for Linked Data visualization see [7]. The most relevant related work to the specific topic of a vocabulary supporting Linked Data visualization is Fresnel - Display Vocabulary for RDF [8]. Fresnel specifies how a resource should be visually represented by Fresnel-compliant tools like LENA [8] and Longwell [9]. Therefore, Fresnel vocabulary could be perceived as a vocabulary for describing LDVM visualization abstraction. This is partly because the vocabulary was created before the Linked Data era and therefore focuses on visualizing RDF data without considering vocabularies and multiple sources.

6 Conclusions

In this paper we briefly described our Linked Data Visualization Model (LDVM) and proposed a Linked Data vocabulary for description of its components and their configuration. The vocabulary supports description of inputs and outputs of individual components, which allows LDVM implementations to check whether components are compatible with each other. In addition, the vocabulary supports creation of new LDVM compatible

[8] `https://code.google.com/p/lena/`

[9] `http://simile.mit.edu/issues/browse/LONGWELL`

component templates and representation of analytic and visualization pipelines based on those components. This support includes creation of component templates from pipeline instances, which facilitates cooperation between expert and non-expert users of LDVM implementations. Expert users can create complex pipelines and provide them as black box components to the non-experts who can then use them in their pipelines. We showed the vocabulary usage on an example of a component template and example of a nested pipeline. There are multiple advantages of representing the templates, their configuration and whole pipelines in RDF according to the LDVM vocabulary. For example, the data processed by the pipelines can be linked to the actual pipelines, the templates and pipelines can be easily manipulated by SPARQL queries and shared among users.

References

1. G. A. Atemezing and R. Troncy. Towards a linked-data based visualization wizard. In *ISWC 2014, 5th International Workshop on Consuming Linked Data (COLD 2014), 20 October 2014, Riva del Garda, Italy*, Riva Del Garda, ITALY, 10 2014.
2. F. Benedetti, S. Bergamaschi, and L. Po. Online Index Extraction from Linked Open Data Sources. In A. L. Gentile, Z. Zhang, C. d'Amato, and H. Paulheim, editors, *Proceedings of the 2nd International Workshop on Linked Data for Information Extraction (LD4IE)*, number 1267 in CEUR Workshop Proceedings, pages 9–20, Aachen, 2014.
3. N. Bikakis, M. Skourla, and G. Papastefanatos. rdf:SynopsViz – A Framework for Hierarchical Linked Data Visual Exploration and Analysis. In V. Presutti, E. Blomqvist, R. Troncy, H. Sack, I. Papadakis, and A. Tordai, editors, *The Semantic Web: ESWC 2014 Satellite Events*, Lecture Notes in Computer Science, pages 292–297. Springer International Publishing, 2014.
4. J. M. Brunetti, S. Auer, R. García, J. Klímek, and M. Nečaský. Formal Linked Data Visualization Model. In *Proceedings of the 15th International Conference on Information Integration and Web-based Applications & Services (IIWAS'13)*, pages 309–318, 2013.
5. A.-S. Dadzie, M. Rowe, and D. Petrelli. Hide the Stack: Toward Usable Linked Data. In G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, and J. Pan, editors, *The Semantic Web: Research and Applications*, volume 6643 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2011.
6. J. Klímek, J. Helmich, and M. Nečaský. Payola: Collaborative Linked Data Analysis and Visualization Framework. In *10th Extended Semantic Web Conference (ESWC 2013)*, pages 147–151. Springer, 2013.
7. J. Klímek, J. Helmich, and M. Nečaský. Application of the Linked Data Visualization Model on Real World Data from the Czech LOD Cloud. In C. Bizer, T. Heath, S. Auer, and T. Berners-Lee, editors, *Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, April 8, 2014.*, volume 1184 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.
8. E. Pietriga, C. Bizer, D. R. Karger, and R. Lee. Fresnel: A Browser-Independent Presentation Vocabulary for RDF. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2006.
9. K. Thellmann, F. Orlandi, and S. Auer. LinDA - Visualising and Exploring Linked Data. In *Proceedings of the Posters and Demos Track of 10th International Conference on Semantic Systems - SEMANTiCS2014*, Leipzig, Germany, 9 2014.