A Dataflow Platform for Applications based on Linked Data

Paolo Bottoni

Department of Computer Science, Sapienza, University of Rome, Rome, Italy
E-mail: bottoni@di.uniroma1.it

Miguel Ceriani

Department of Computer Science, Sapienza, University of Rome, Rome, Italy
E-mail: ceriani@di.uniroma1.it

Abstract: Modern software applications increasingly benefit from accessing the multifarious and heterogeneous Web of Data, thanks to the use of Web APIs and Linked Data principles. In previous work the authors proposed a platform to develop applications consuming Linked Data in a declarative and modular way. This paper describes in detail the functional language the platform gives access to, which is based on SPARQL (the standard query language for Linked Data) and on the dataflow paradigm. The language features interactive and meta-programming capabilities so that complex modules/applications can be developed. By adopting a declarative style, it favours the development of modules that can be reused in various specific execution contexts.

Keywords: Linked Data; Semantic Web; SPARQL; RDF; dataflow; declarative programming.

1 Introduction

Modern software applications increasingly rely on the consumption of multiple and heterogeneous sources of data, which are often online. This trend challenges the classical model of client/server applications, typically based on a centralized database and an enclosed set of modules representing the logic of the system.

Two complementary paradigms are changing the organization and distribution of data: (1) the ubiquity of *Web APIs* as the primary means to access a distributed and mutating landscape of data-sources; (2) the slow, but growing, adoption of *Semantic Web* technologies and principles to represent, in a non-centralized way, the connections and relationships among different data sources, collectively composing the Linked Data.

For software logic, there is a trend in modularization and distribution at the Web scale, too. Modern package managers, like `npm` or `Maven`, simplify the process of using external libraries, favouring the progressive atomization of modules that tend to offer minimal sets of functionalities. The role of the developer is then increasingly concerned with choosing and composing existing modules, rather than developing new solutions based on monolithic languages and standard libraries.

Still, reuse of logic across different contexts is hindered by ties to specific execution paradigms. For example, in imperative programming, a module written for synchronous use must be rewritten to be used in an asynchronous way. This paper, arguing that reuse of application logic may be facilitated by the use of declarative forms of programming, describes a development platform, the Semantic Web Open dataflow System (SWOWS), with an underlying dataflow computational model making use of existing Semantic Web standards, which allows distribution and integration of the application logic, while providing easy and transparent access to Linked Data. Data and logic can thus be seen as SWOWS modules which can interoperate and be integrated in several ways.

SWOWS uses a specific language, called **DfPL**, for Dataflow Programming Language, equipped with two equivalent forms of syntax: one which is expressed in RDF and is based on a specific ontology, and one which uses visual primitives and constructs to express a computation. Previous works introduced the first versions of the language and of the platform [Bottoni et al., 2013, Bottoni and Ceriani, 2014a], then extended them to allow for dynamic programming [Bottoni and Ceriani, 2014b]. This paper provides details on this extension, concerning the definition of both the ontology and the visual components.

The rest of the paper is organised as follows. After discussing some technological background in Sect. 2, Sect. 3 presents a motivating example. Related work is discussed in Sect. 4, while Sect. 5 presents the language, the semantics of which are specified in Sect. 6. Sect. 7 describes the proposed platform and its user interface, with hints on its implementation given in Sect. 8. Finally, Sect. 9 discusses conclusions and future work.

Copyright © 201X Inderscience Enterprises Ltd.

2 Technological Background

The DfPL language and the SWOWS platform are based on existing Web and Semantic Web standards and technologies. In this section the main adopted technologies are briefly described.

2.1 RDF

The Resource Description Framework (RDF) [Klyne et al., 2004] is a model for structuring data on the Web, designed as basic layer for the Semantic Web [Berners-Lee et al., 2001]. In the RDF data model, knowledge is represented via statements about resources, where a resource is an abstraction of any piece of information to be modeled. An *RDF statement* (also called a triple) is composed of *subject* (a resource), *predicate* (specified by a resource as well) and *object* (a resource or a literal, i.e. a value from a basic type). An *RDF graph* is therefore a set of triples. Resources in a graph can be identified by a URI if they have meaning outside of that graph, or by a local identifier otherwise (in which case they are called blank nodes). The resources used to specify predicates are called properties. A resource may have one or more types, specified by the predefined property `rdf:type`. An *RDF dataset* is a set of graphs, each one associated with a different name (a URI), plus a default graph without a name. RDF is used through the platform to represent any kind of information and its transformations.

2.2 SPARQL

SPARQL is the standard query language for RDF datasets [Harris et al., 2013]. It has a relational algebra semantics, analogous to those of traditional relational languages such as SQL. A SPARQL `Construct` takes as input an RDF dataset and produces an RDF graph. While the SPARQL Query Language is "read-only", the SPARQL Update standard [Schenk et al., 2013] enables updates of an *RDF graph store*, the "modifiable" version of an RDF Dataset. A SPARQL Update *request* is composed of SPARQL Update *operations*, each describing a specific type of update on the RDF graph store. The current version of the standard is SPARQL 1.1, but much of the existing work in literature refers to SPARQL 1.0 [Prud'hommeaux and Seaborne, 2008]. SPARQL 1.1 algebra offers an expanded set of operators, effectively allowing the expression of a new class of queries. SPARQL is the basis for DfPL, which define Linked Data applications through a network of SPARQL queries and SPARQL Update requests.

2.3 Rich Web Clients

The ubiquity of Web browsers and Web document formats across a range of platforms and devices drives developers to choose to build applications on the Web and its standards. From the point of view of the requirements for browser, there has been a dramatic change from the first days of the Web. Nowadays, a browser is an interface to an ever growing set of client capabilities. All modern browsers natively support the Scalable Vector Graphics (SVG) standard [Dahlström et al., 2011], an XML-based language representing mixed vector and raster content. Together with the long established Document Object Model (DOM) Events [Pixley, 2000, Kacmarcik et al., 2000] and JavaScript support, it allows the realisation of complete interactive visualisation applications. Indeed, JavaScript libraries for interactive data visualization are proliferating, from standard visualisations [Google, 2010, Belmonte, 2011, Bostock et al., 2011] to specialized visualisations for specific domains [Smits and Ouverney, 2010], leveraging especially SVG technology.

3 An Example Application

As a concrete example, the general design of an example Web application, PoIShow, is discussed. PoIShow will present to users points of interest (POIs) close to their current location. The data for these POIs may be taken from different data sources containing information on geo-located places. In the case of multiple data sources, the data could be integrated either by materializing a local view or by requesting the resources from the sources on demand. For the sake of simplicity, it is assumed that data come from a single source: Wikidata, a collaboratively edited knowledge base operated by the Wikimedia Foundation (sort of "the Wikipedia of the Data") for which a public SPARQL endpoint is available, although a private replica of the relevant part of Wikidata could be used as well.

The Graphical User Interface (GUI) of PoIShow will provide a presentation of the POIs dynamically found in an area centred on the user, for example in the form of a simple list with a short description of each POI, or of a map with icons corresponding to the POIs in the corresponding positions. Upon activation, and whenever the user position changes, PoIShow will need to update the POI list/map by executing the corresponding query on the Wikidata SPARQL endpoint.

Using established methods, PoIShow will typically consist of three main components: *client-side*, a single page HTML+CSS+JavaScript application interacting with the back-end to get data; *front-end*, the server-side application that generates the client-side code to be sent to the client; *back-end*, the server-side application that gives access to data, by relying requests as queries to the SPARQL endpoint. It should be noted that while the client-side is certainly needed, each of the other two components could possibly be dispensed with: the front-end could be absent if no logic is needed to generate the client-side, i.e. all the Web resources are static; the back-end could be avoided by accessing directly the SPARQL endpoint from the client-side.

In any case the application logic will have to guarantee the following activities: 1. build the query,

including binding some latitude/longitude variables with the location of the user; 2. run the query against the endpoint and get the results (the POIs); 3. update the client-side view based on the loaded POIs; 4. upon changes in the user's location, reiterate the procedure.

The essential aspect of this logic is that a new query is dynamically created each time, so that a suitable mechanism is needed.

To that, it should be added that different categories of POIs can be defined, corresponding to different filters on the geo-localised data available from Wikidata. Finding a way to represent these filters, and devising a mechanism to build the dynamic query according to both the used filters and the user position, adds another layer of complexity to the application.

In DfPL, queries and transformations are represented in RDF, and hence first class citizenships, like any other RDF graph of data. An application such as PoIShow can be programmed and modified incrementally in a natural way, by leveraging functionality, modularity, and meta-programming capabilities of DfPL.

4 Related Work

This section reports on usage of declarative languages and paradigms in the context of applications that consume Linked Data.

4.1 View Definition Languages

Several languages have been proposed for the definition of SPARQL views, in a way analogous to SQL views. A SPARQL view is a graph intensionally defined through a SPARQL `Construct` query; the input dataset of the query can be composed by both "real" graphs (extensionally defined) and views.

RVL [Magkanaraki et al., 2003] is an early effort, using an imperative language for defining views based on an independently defined query language (RQL [Karvounarakis et al., 2002]). Views are associated with "namespaces" (identified through unique URIs), representing virtual RDF/S datasets. Namespaces can be both extensionally and intensionally defined, as RQL queries of other namespaces. The query language offers some relations and operators to deal with RDF/S concepts (`rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain, rdfs:range`).

vSPARQL [Shaw et al., 2011] is an extension of the SPARQL 1.0 grammar allowing named views defined with `Construct` queries and reusable in other queries.

Schenk and Staab, working on *Networked Graphs* [Schenk and Staab, 2008], propose an RDF-based syntax to specify views as graphs resulting from the execution of SPARQL 1.0 `Construct` queries on explicitly identified graphs and other views. Although powerful enough to define read-only applications (possibly together with visualization tools described below), network of views do not easily model interactive applications. In particular, they face the problem of how to represent events and time-dependent information, including the application state.

4.2 Pipeline Languages and Linked Data Visualization Tools

Two pipeline languages have been proposed for the (visual) specification of RDF transformations, namely DERI Pipes [Le-Phuoc et al., 2009] and SPARQLMotion [Knublauch et al., 2010]. They offer a set of basic operators on RDF graphs to build the pipelines, which are then typically executed in the context of a batch or Web application (within a Web site or a Web service), in response to GET or POST requests. Both languages are endowed with a graphical environment to create the pipelines using the available operators (free for DERI Pipes, in a commercial software for SPARQLMotion).

In DERI Pipes, pipelines are defined in XML, with SPARQL queries in textual form. The set of operators includes one that loads RDF graphs from files or as result of a query to a SPARQL End Point (FETCH), one that transforms RDF graphs using SPARQL `Construct` queries (CONSTRUCT) and one that merges a set of RDF graphs into one (MIX). The operators have no side-effects and the pipelines are stateless.

In SPARQLMotion, both pipelines and queries are represented in RDF (for queries using the SPIN-SPARQL syntax [Fürber and Hepp, 2010]). SPARQLMotion has operators similar to DERI Pipes together with a number of operators converting to and from a number of data formats (XML among them) and RDF. The main difference with DERI Pipes is the presence of operators with side-effects, e.g. to save to file, update the *active* RDF Graph, or send emails.

Visualbox [Graves, 2013] and Callimachus [Battle et al., 2012] have been explicitly developed for linked data visualisation. In their two-step model/view approach, SPARQL queries select data and a template language generates the (XML-based) visualisation.

SPARQL Web Pages (SWP) is an RDF-based framework (to be used with SPARQL Motion or on its own) to describe and render HTML+SVG visualisations of linked data. HTML and SVG are mapped to two corresponding vocabularies and, together with the UISPIN Core Vocabulary, allow the association of a RDF resource with the description of its visualisation. The description may be also statically associated with a class of resources, with each specific resource mapping defined through a SPARQL query (in SPIN-SPARQL syntax).

In all these proposals, the execution model corresponds to the management of a single HTTP request, as with typical application server technologies like Java Servlet or PHP. Persistence and logical relationships between requests and client state must be managed explicitly (e.g. saving/loading data related to a session and encoding parameters in requests).

4.3 RDF-Based Transformation Languages

Besides Networked Graphs and SPARQLMotion, other proposals for RDF-based declarative languages operating on RDF include the functional and stack-oriented Ripple language [Shinavier, 2007], which takes a resource-centric approach (in contrast with the SPARQL relational approach) and a language to apply Algebraic Graph Transformations to RDF Graphs [Braatz and Brandt, 2010]. These otherwise interesting alternative approaches are impervious from the point of view of the developer, because of the need to learn from scratch a non-standard language.

4.4 Dynamic Query Generation

Generation/manipulation of queries at runtime is widely used both in SQL and in SPARQL. Several systems (for a SPARQL example, see Jena [McBride, 2002]) provide basic support in the form of *parameterised queries*, in which some *parameters* are bound (via some external mechanism) to actual (scalar) values at execution time. This mechanism is not sufficient when the structure of the query must be changed dynamically (e.g. for multiple or complex search criteria and/or ordering rules). Generic dynamic query generation is usually achieved through string-based, semantically-unaware manipulation, or through a programmatic interface offered by the host language. There have been efforts to represent queries using semantically rich structures not tied to a specific host language [Van den Bussche et al., 2005], e.g. the already mentioned SPIN-SPARQL [Fürber and Hepp, 2010] vocabulary for SPARQL. While this vocabulary potentially allows dynamic query generation, its use for this purpose is documented only for a specific case of query rewriting [Follenfant et al., 2012] (SPIN-SPARQL vocabulary is as a fact widely used, but mainly as a way to attach SPARQL rules to RDF resources).

5 Language Description

This section describes the Dataflow Pipeline Language (DfPL), a declarative language for RDF-based applications.

As DfPL is aimed at creating applications and modules that can be shared and reused, any module should be as independent and self-contained as possible. Hence, DfPL is built using side-effect-free operators.

While languages considered in Sect. 4 are either stateless or have operators with side-effects that modify the system state, e.g. updating some RDF Store, DfPL offers special operators (like the *simple graph store*) to define a portion of the application state. In other words, the state is distributed and bound directly to the related functional module.

In DfPL the basic functional module is the *Dataflow*. A Dataflow defines how to build an output RDF dataset from an input one, through a configuration RDF graph, called the *dataflow graph*. The content of the output dataset is defined as the result of the cascaded application of graph operators to the input dataset.

5.1 Dataflow Ontology

The syntax for specifying a dataflow is based on a suitably defined ontology, with namespace `http://www.swows.org/2012/06/dataflow`, denoted in this paper by the prefix `df:`. The ontology reuses classes and properties from the *SPIN SPARQL Syntax* [Fürber and Hepp, 2010] (`http://spinrdf.org/sp`, prefix `sp:`).

The classes defined by the ontology are described in Table 1, which also shows the hierarchy relations between classes – represented with `rdfs:subClassOf`. Table 2 shows their properties, presenting for each property its domain, along with the range and cardinality restrictions associated with it – domain and range are represented with RDFS, while restrictions on the corresponding domain are represented with OWL. A dataflow is described using a resource of type `df:Dataflow`. The property `df:hasComponent` associates a dataflow resource with each of its components.

A resource corresponding to a component of the dataflow must have as type one of the (disjoint) subclasses of the class `df:DataflowComponent`. Of special interest are the resources of class `df:Transformer`. They read the input RDF dataset from another component and apply a transformation on it, defined via a resource of type `df:DatasetTransformation`. This resource can be in the same RDF graph as the dataflow, connected with the property `df:inlineConfig`, or in an RDF graph dynamically generated by another component, connected with the property `df:config`. In the latter case the property `df:configRoot` is used to identify the dataset transformation resource in the RDF graph. The `df:DatasetTransformation` class is subclassed to provide four kinds of transformation. Three of the subclasses, `sp:Construct`, `sp:Select`, and `sp:Update`, are SPIN SPARQL classes used to describe a query or an update request in the SPIN SPARQL Syntax. They are interpreted so that the request is executed against the input RDF dataset and the output is defined in the following table:

Operation	Output
`CONSTRUCT` query	query result
`SELECT` query	query result represented as RDF graph
update request	the input RDF dataset after applying the update request

Finally, `df:DatasetTransformation` is subclassed also by `df:Dataflow`, used to describe the transformation through a dataflow, following the syntax described in the present section. By using a dataflow as an RDF dataset transformation, it becomes possible to build a dataflow composing other (sub)dataflows.

The `df:Transitioner` is the dataflow component dealing with the internal state of the dataflow. It maintains its output until the input changes, at which point the input is propagated to the output but with a new timestamp – meaning that a transition has occurred.

Table 1 Classes of the dataflow ontology (only direct relationships are shown).

Class Name	Description	has super-classes
`df:GraphProducer`	has an RDF graph as output	
`df:GraphConsumer`	has an RDF graph as input	
`df:DatasetProducer`	has an RDF dataset as output	
`df:DatasetConsumer`	has an RDF dataset as input	
`df:DatasetTransformation`	a transformation from an RDF dataset to another	

`df:Dataflow`	a dataflow, described through its components	`df:DatasetTransformation`
		`df:DatasetConsumer`
		`df:DatasetProducer`

`df:DataflowComponent`	a component of a dataflow	
`df:GraphNamer`	gives a name to a graph for building an RDF dataset	`df:DataflowComponent` `df:GraphConsumer`
`df:NamedGraphSelector`	selects a named graph from an RDF dataset	`df:DataflowComponent` `df:GraphProducer`
`df:DefaultGraphSelector`	selects the default graph from an RDF dataset	`df:DataflowComponent` `df:GraphProducer`
`df:ConstantTriple`	generates an RDF graph consisting of a single triple	`df:DataflowComponent` `df:GraphProducer`
`df:Merger`	applies the *RDF union* operation to all the input RDF graphs	`df:DataflowComponent` `df:GraphProducer`

`df:Transformer`	applies a `df:DatasetTransformation` on the input RDF dataset, obtaining another RDF dataset in output	`df:DataflowComponent`
		`df:DatasetProducer`
		`df:DatasetConsumer`

`df:Transitioner`	if the input is changed, copy it to the output performing a transition to a new state	`df:DataflowComponent`
		`df:GraphProducer`
		`df:GraphConsumer`

`df:SingleGraphStore`	based on the effect of a `df:DatasetTransformation` on the default graph, performs a transition on the state of an initially empty local graph; it is just syntactic sugar on top of a `df:Transformer` and a `df:Transitioner`	`df:DataflowComponent`
		`df:GraphProducer`
		`df:DatasetConsumer`

`df:NumericRange`		`df:DataflowComponent` `df:GraphProducer`
`sp:Construct`	describes a SPARQL `CONSTRUCT` query	`df:DatasetTransformation`
`sp:Select`	describes a SPARQL `SELECT` query	`df:DatasetTransformation`
`sp:Update`	describes a SPARQL update request	`df:DatasetTransformation`

Table 2 Properties of the dataflow ontology (only direct relationships are shown).

Property Name	Description	Domain	Range & Cardinality	
`df:hasComponent`	component of dataflow	`df:Dataflow`	`df:DataflowComponent`	0–∞
`df:input`	input RDF graph	`df:GraphConsumer` `df:Merger`	`df:GraphProducer` `df:GraphProducer`	1 0–∞
`df:defaultInput`	default graph for input RDF dataset	`df:DatasetConsumer`	`df:GraphProducer`	0–1
`df:namedInput`	named graph for input RDF dataset	`df:DatasetConsumer`	`df:GraphNamer`	0–∞
`df:name`	name of named graph	`df:GraphNamer` `df:NamedGraphSelector`	`xsd:anyURI` `xsd:anyURI`	1 1
`df:datasetProducer`	associated RDF dataset producer	`df:DefaultGraphSelector` `df:NamedGraphSelector`	`df:DatasetProducer` `df:DatasetProducer`	1 1
`df:subject`	subject of triple	`df:ConstantTriple`		1
`df:predicate`	predicate of triple	`df:ConstantTriple`		1
`df:object`	object of triple	`df:ConstantTriple`		1
`df:config`	input from which configuration graph is read	`df:Transformer` `df:FixpointFinder`	`df:GraphProducer` `df:GraphProducer`	0–1 0–1
`df:configRoot`	relevant resource in configuration graph, must be of type `df:DatasetTransformation`	`df:Transformer` `df:FixpointFinder`	`xsd:anyURI` `xsd:anyURI`	0–1 0–1
`df:inlineConfig`	configuration resource, directly in current graph	`df:Transformer` `df:FixpointFinder`	`df:DatasetTransformation` `df:DatasetTransformation`	0–1 0–1

The aforementioned `df:SingleGraphStore` is defined as syntactic sugar on top of the `df:Transformer` and `df:Transitioner` components. Specifically, a `df:SingleGraphStore` can be represented by a `df:Transformer`, whose output default graph is sent to a `df:Transitioner` and then feedbacked as its input default graph. So, for example, each occurrence of

```
?s a df:SingleGraphStore ;
   df:inlineConfig ?c ;
   df:namedInput ?i .
```

can be interpreted as replaced by

```
?s a df:Transitioner ;
   df:input [
     a df:DefaultGraphSelector ;
     df:datasetProducer [
       a df:Transformer ;
       df:inlineConfig ?c ;
       df:defaultInput ?s ;
       df:namedInput ?i ] ] .
```

The classes `df:(Graph/Dataset)(Producer/Consumer)` represent different roles in a dataflow and are subclassed by the specific component classes. In particular, a `df:Dataflow` resource is in turn both a `df:DatasetProducer` and a `df:DatasetConsumer` for its internal components. It produces the input RDF dataset to be transformed and consumes the output RDF dataset.

The dataflow obtained by excluding `df:Transitioner` components (hence `df:SingleGraphStore` components) is required to be acyclic. This requirement will affect the language semantics, in the sense that no recursive computations can be made without triggering a state transition event.

5.2 XML DOM Elements

As many of the intended applications of DfPL rely on XML formats, RDF graphs are used also to represent XML documents. Hence, an ontology to describe XML documents in RDF has been defined (namespace `http://www.swows.org/2013/07/xml-dom` and prefix `xml:`), based on the XML Document Object Model (DOM) [Apparao et al., 1998].

The ontology offers classes such as `xml:Element`, `xml:Attr`, `xml:Text` for representing the basic XML node types. The engine provides also some methods to avoid explicitly subclassing the known elements, so that they can be used directly.

The hierarchical relation between DOM nodes is represented using the property `xml:hasChild`. In case the relative order of the children in the DOM tree is important, there are two ways to express it: 1) using the `xml:orderKey` property on each child one wants to order, or 2) using the `xml:childrenOrderedBy` property on the parent. The `xml:childrenOrderedBy` property takes as value the name of the property that will be used to order the children. In both cases, elements are ordered following the semantics of the operator `<` in SPARQL; the order can be reversed to descending, by using the `xml:childrenOrderType` property with the value `xml:Descending` on the parent. This allows SPARQL developers to flexibly define orders, instead of being forced to indicate the exact index of a child or to create a linked list of children. Moreover, in this model, a single RDF node can be used as child of different parents, and thus converted into different XML DOM nodes (this can be useful if the same DOM subtree is needed in different places of a document) that can be even ordered in different ways (depending on the properties of each parent). If an element uses an `id` attribute, this corresponds, in the RDF model, to a resource with the URI for the fragment identifier syntax in XML. This allows easy reference from SPARQL queries to specific elements inside a XML document.

5.3 DOM Events

In order to bridge SWOWS to existing event management infrastructures, another ontology has been defined to represent DOM Events [Pixley, 2000], from generic to specific ones, e.g. mouse events, mimicking the interfaces of the W3C DOM standard. In the output, each node of type `xml:Element` can have one or more values for the property `xml:listenedEventType`, describing the event types that the application needs to listen to.

5.4 Application Context

A dataflow can be designed just for reuse by other dataflows. If a dataflow has to be executed as a complete application (i.e. it is a *top-level* dataflow), its inputs and outputs will depend on its context of use. The engine running the dataflow may provide adapters for different contexts. Here the context of a GUI application is considered.

For the developer of a GUI application, the output is an image to be drawn in the application window while the input is given by events generated by the system as a result of user interaction.

The image is represented using HTML and SVG [Dahlström et al., 2011]. The incoming events are associated with RDF nodes corresponding to target HTML and SVG elements.

The default graph of the input dataset contains information about the latest event (if one has been fired and it must still be used). Of the output dataset, also the default graph is considered: it is the HTML+SVG document, defined through the XML DOM ontology (see 5.2). The running engine has to generate and update the screen graphics as this graph changes.

6 Language Semantics

This section describes the intended language semantics for DfPL. An RDF time model is introduced, such that dynamic RDF data sources may be considered, instead of static RDF graphs. The semantics is first presented for the stateless fragment and later extended to the full language by adding a representation of state transitions.

6.1 Time Model

In the standard terminology, an *RDF source* [Cyganiak et al., 2014] is a mutable (in time) source of RDF graphs; a *snapshot* of the state of an RDF source is an RDF graph. The input dataset of a dataflow is a set of named RDF sources plus a default RDF source; we can say that this dataflow is "applied" to a set of RDF sources. The result (represented by the output dataset) is another set of named RDF sources.

Given a set T representing *the time* – the set of *instants* one is interested in – an *RDF graph source* G_S is defined as a function

$$\begin{aligned} G_S : T &\to \mathcal{G} \\ t &\mapsto G \end{aligned}$$

where G is the snapshot of G_S at time t.

Similarly an *RDF dataset source* D_S is defined as

$$\begin{aligned} D_S : T &\to \mathcal{D} \\ t &\mapsto D \end{aligned}$$

where D is the snapshot of D_S at time t.

An explicit characterization of instants and of the set T is avoided by design. In a simple application, instants can be timestamps generated by the local processor, while in a more complex scenario, e.g. of a distributed system and with time order critical, a centralized service can generate representations for those instants. Even with an explicit representation for the instants, the set T is not necessarily fixed or uniquely defined.

The model of time that has been chosen is linear and discrete, as typical for user interface event management and data management models. Linearity means that there is a total – chronological – order relation, $<$, defined on T so that any two distinct instants can be compared. Discreteness means that the variation of content of a data source happens in a finite or countably infinite set of instants. Two specific cases of RDF graph sources are considered: *RDF graph stream (Rgs)* and *temporal RDF graph (tRg)*, which can be both defined as an ordered sequence G_S of pairs: $G_S = ((t_1, G_1), (t_2, G_2), \dots)$, where $t_i \in T$ is a instant in time such that $t_i < t_{i+1}$ and G_i is the snapshot at time t_i, but with two different interpretations:

- for an Rgs, G_i is associated only with the instant t_i and no content is associated with the instants that are not in the sequence (in general $\{t_i\} \subset T$);
- for $atRg$, G_i is associated with the open time interval $[t_i, t_{i+1})$ – that is, valid until it changes – so that every $t \in T$ has some associated content.

Streams are used to model *events*, for which content has meaning only while they are handled (for example, for the application described in Section 3, they would be user interface events). Temporal RDF graphs, instead, represent content that changes in time, typically depicting a state, internal or external to the application (such as the current location in the example application). Similar definitions can be applied on RDF datasets to define *temporal RDF datasets* and *RDF dataset streams*.

The set T depends from the context in which the source is used. It should also be noted that this model of time is used in the language to coherently define content modifications, event streams, and state transitions. Hence the obvious limitation is that there is no way to express the content of a source at an instant t_1 as depending from contents of any resource at any instant t_2 in the future ($t_1 < t_2$).

6.2 Stateless Semantics

By excluding `df:Transitioner` components, one obtains the stateless (combinatorial) and acyclic fragment of the language. A stateless dataflow calculates the state of the output RDF sources as a function of the state of the input RDF sources. The semantics of these calculation are given directly by the execution of the corresponding SPARQL queries and the other stateless operators.

6.3 Semantics of Stateful Dataflows

The semantic interpretation for stateless dataflows with internal stateless operators is easily extended to the framework of RDF graph (dataset) sources. For all the stateless components the output sequence of instants is the (ordered) union of the input sequences; the snapshot of the output at a certain instant can be computed from the snapshot of the inputs at the same instant.

A stateful dataflow is one containing at least an instance of the `df:Transitioner` component. To interpret such a dataflow, one can assume that all the `df:Transitioner` components are removed from the dataflow, considering then each input (output) of a `df:Transitioner` as an output (input) of the dataflow. Whenever the input of the `df:Transitioner` changes, a new instant is added to the sequence, meaning an action has been executed.

Note that – due to state transitions – the output sequence may contain more instants than the input. The implementation may keep track of the last produced output dataset to avoid generating the pair (t, D') if D' is not changed. In this case the output sequence may contain fewer instants than the input.

It should be noted that the execution of the step function is considered exclusive – it cannot be called simultaneously by different processes for the same dataflow instance. This means that if each event is associated with a distinct time instant, the engine must fully handle an event before considering the following one. In practice – as some parts of a dataflow can be independent from some inputs or certain changes – the implementation may handle events in parallel as long as the semantics are preserved.

7 The SWOWS Platform

The proposed platform allows users to design and execute linked data applications using DfPL. Users create pipelines in a modular way via a visual representation provided by a Web-based *editor* (see Sect. 7.5), which in turn interacts with a *pipeline repository*. At the back-end, the *dataflow engine* executes the pipeline (after downloading the corresponding RDF Graph from the repository) on a possibly separate server, offering a Web-based interface as well. An edited pipeline can be saved into a directly controlled repository, from which it can be extracted to be used "as is" or to be incorporated and reused in other pipelines.

7.1 The Repository

The pipeline repository is embedded in the Callimachus web server [Battle et al., 2012], a Content Management System based on linked data. Callimachus contains an OpenRDF Sesame triple store, maintaining data that represents the content structure. The actual content of each uploaded file is stored in blobs (in the case of RDF files both as blobs and as triples). Any content that is managed by Callimachus is an instance of specific OWL class such as *Page*, *Folder*, *File*, *User* (full ontology at `http://callimachusproject.org/rdf/2009/framework`).

The Web interface of Callimachus exposes the content of a server instance in three ways:

- Web pages for view, creation, editing of resources associated via templates to each class;
- REST API for programmatic access to resources, also dependent on their class;
- direct SPARQL access to the triple store.

Callimachus applications are composed of instances of existing classes, possibly new classes and supporting resources and code. For the pipeline repository, a Callimachus application has been developed: it contains the new class *SWOWS Pipeline*, associated with the pipeline editor code. A SWOWS Pipeline is represented in Callimachus using DfPL and the vocabulary described in the following section.

7.2 Pipeline layout representation

The pipeline is saved for execution using DfPL which, however, does not directly represent how the pipeline is visually manipulated in the editor. Hence, a different vocabulary to represent the pipeline visual aspects is added, with a one-to-one matching between visual components and RDF representations. This also allows a decoupling of the editor from the engine: the same editor could generate executable programs according to other syntaxes (e.g. SPARQLMotion syntax [Knublauch et al., 2010]), while different kinds of editor could generate dataflows. The ontology for this visual vocabulary is an extension of: (1) the `myExperiment` ontology [Newman et al., 2009] for dataflow components; (2) the Dublin Core meta-data ontology; and (3) the GRAPHIC module of Visualization Ontology (VISO) for describing data visualizations [Polowinski and Voigt, 2013].

7.3 The Execution Engine

The *dataflow engine* is the software component that executes DfPL pipelines. It downloads the pipeline description and other file sources through URLs and, during execution, it can connect to SPARQL endpoints to get data on demand. The engine is neutral with respect to the specific inputs and outputs used by a dataflow, which are defined by the execution context.

7.4 Web Interface of the Engine

The Web interface of the dataflow engine consists of a service that manages a pool of running dataflows, identified by the dataflow graph URI and a unique code. Currently, two operations are available:

- creation of a new instance of a dataflow, returning the initial Web page together with added JavaScript code to support interaction;
- pushing a user interface event to the input of an existing instance of a dataflow and returning the changes to be applied client side.

The content of the Web page sent to the client can be any XML document represented by the vocabulary described in Sect. 5.2, typically HTML and/or SVG. The added JavaScript code manages the user interface events and sends them to the server as RDF graphs. The server replies with a piece of JavaScript code that, when executed, changes the XML DOM as needed.

7.5 Editor User Interface

The editor is contained in a Web page (see Figure 1) providing tools to create and modify dataflows saved as RDF graphs on a Graph Store [Ogbuji, 2013]. The dataflows are built visually as pipelines of components that loosely correspond to the components in the dataflow ontology.

The user interface of the editor is composed of:

- a *component panel* (left), with the available components;
- the *editor area* (center), where the pipeline is built by dragging, linking, and configuring the desired components selected from the panel;
- the *command panel* (upper right), containing buttons for operations related to the whole pipeline, e.g. saving it or executing it;

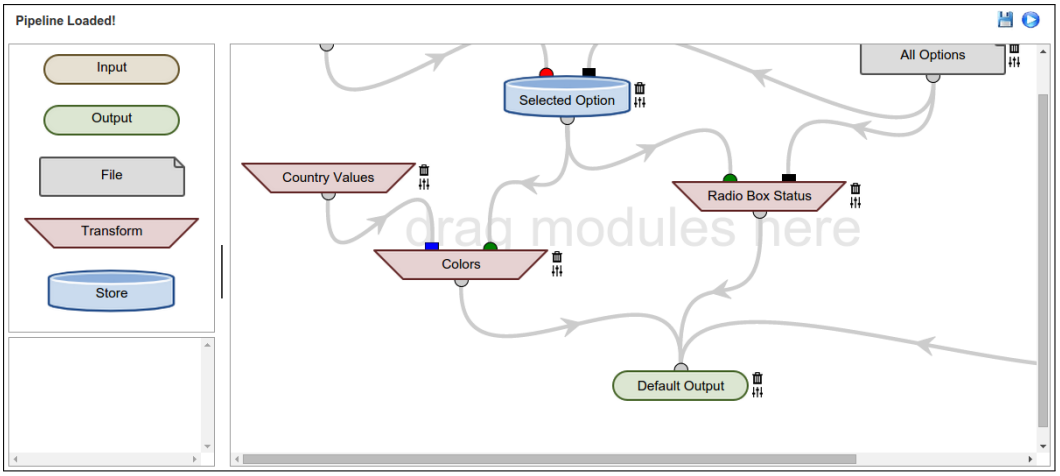

Figure 1 A screenshot of the Web-based editor.

- the *helper area* (bottom-left corner), for contextual help on components.

The components are: 1. the *default input graph* and the *named input graphs*, corresponding respectively to the default graph and the named graphs of the input RDF dataset; 2. the *default output graph* and the *named output graphs*, corresponding respectively to the default graph and the named graphs of the output RDF dataset; 3. the *transform processor*, for the `df:Transformer` component; 4. the *single graph store*, for the `df:SingleGraphStore` component; and 5. *file data sources*, RDF graphs generated by loading local or remote files (serialized in one of the standard RDF formats). Apart form file data sources, data from outside the pipeline can be accessed through *SPARQL Federated Queries* [Prud'hommeaux and Buil-Aranda, 2013], used in transform processors or simple graph stores.

8 Implementation

The editor is a rich Web application with its client side logic coded in HTML+CSS+JavaScript. The pipeline repository is an instance of Graph Store that must be located in the same host as the editor. The dataflow engine is a Java based (using Apache Jena [McBride, 2002]) Web application maintaining the state of each running pipeline instance; when a new instance is launched (e.g., from the editor) the engine initialises the pipeline and returns its output to the client, along with JavaScript code to report handled events back to the server; each time an event is fired on the client, the dataflow engine is notified and answers with the changes to be executed on the client content. On the client side, any *modern* browser supporting JavaScript can use both the editor and the generated application. The software is free and available at `http://www.swows.org/`.

9 Conclusions and Future Work

The DfPL language, and the asssociated interactive platform, SWOWS, for visual composition and execution of sentences, are designed to build data manipulation and visualisation applications through a declarative approach, based on Semantic Web technologies as RDF and SPARQL. The language is novel in being fully based on SPARQL query language, while at the same time providing support for interactive applications and meta-programming techniques. While the language was already introduced in previous work, in this paper the RDF-syntax is fully specified and the semantics described.

References

V. Apparao, S. Byrne, M. Champion, S. Isaacs, et al. Document Object Model (DOM) Level 1 Specification - Version 1.0. W3C Recommendation, 1998.

S. Battle, D. Wood, J. Leigh, and L. Ruth. The Callimachus Project: RDFa as a Web Template Language. In *Proc. COLD*, 2012.

N. G. Belmonte. JavaScript InfoVis Toolkit. philogb.github.io/jit, 2011.

T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics*, 17:2301–9, 2011.

P. Bottoni and M. Ceriani. A Dataflow Platform for In-silico Experiments Based on Linked Data. In *Proc. DNIS 2014*, pages 112–131, 2014a.

P. Bottoni and M. Ceriani. SWOWS and dynamic queries to build browsing applications on linked data. *J. Vis. Lang. Comput.*, 25(6):738–744, 2014b.

P. Bottoni, M. Ceriani, and S. Valentini. A user interface to build interactive visualizations for the Semantic Web. In *ISWC 2013 (Posters & Demos)*, pages 165–168, 2013.

B. Braatz and C. Brandt. How to Modify on the Semantic Web? In *Current Trends in Web Engineering*, pages 187–198. Springer Berlin / Heidelberg, 2010.

R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, 2014.

E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C Recommendation, 2011.

C. Follenfant, O. Corby, F. Gandon, D. Trastour, et al. RDF modelling and SPARQL processing of SQL abstract syntax trees. In *Proc. PSW'12*, 2012.

C. Fürber and M. Hepp. Using SPARQL and SPIN for data quality management on the semantic web. In *Business Information Systems*, pages 35–46. Springer, 2010.

Google. Google charts, 2010.

A. Graves. Creation of visualizations based on linked data. In *Proc. WIMS'13*, page 41. ACM, 2013.

S. Harris et al. SPARQL 1.1 Query Language. W3C Recommendation, 2013.

G. Kacmarcik, T. Leithead, J. Rossi, D. Schepers, B. Höhrmann, P. Le Hégaret, and T. Pixley. Document Object Model (DOM) Level 3 Events Specification. W3C Recommendation, 2000.

G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *Proc. WWW2002*, pages 592–603. ACM, 2002.

G. Klyne, J. J. Carroll, and B. McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.

H. Knublauch et al. SPARQLMotion Specifications, 2010. sparqlmotion.org.

D. Le-Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid Prototyping of Semantic Mash-Ups through Semantic Web Pipes. In *Proc. WWW '09*, pages 581–590. ACM, 2009.

D. Newman, S. Bechhofer, and D. De Roure. myExperiment: An ontology for e-Research. In *Proc.*

A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web through RVL Lenses. In *Proc. ISWC 2003*, pages 96–112. Springer, 2003.

B. McBride. Jena: a semantic Web toolkit. *Internet Computing, IEEE*, 6(6):55–59, Nov/Dec 2002.

SWASD at ISWC-2009, volume 523. CEUR-WS, 2009.

C. Ogbuji. SPARQL 1.1 Graph Store HTTP Protocol. W3C Recommendation, 2013.

T. Pixley. Document object model (DOM) level 2 events specification. W3C Recommendation, 2000.

J. Polowinski and M. Voigt. VISO: a shared, formal knowledge base as a foundation for semi-automatic InfoVis systems. In *Proc. CHI'13*, pages 1791–1796. ACM, 2013.

E. Prud'hommeaux and C. Buil-Aranda. SPARQL 1.1 Federated Query. W3C Recommendation, 2013.

E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *Proc. WWW '08*, pages 585–594. ACM, 2008.

S. Schenk, P. Gearon, et al. SPARQL 1.1 Update. W3C Recommendation, 2013.

M. Shaw, L. T. Detwiler, N. Noy, J. Brinkley, and D. Suciu. vSPARQL: A view definition language for the semantic web. *Journal of Biomedical Informatics*, 44(1):102–117, 2011. Ontologies for Clinical and Translational Research.

J. Shinavier. Functional programs as linked data. In *3rd Workshop on Scripting for the Semantic Web*, 2007.

S. A. Smits and C. C. Ouverney. jsPhyloSVG: A Javascript Library for Visualizing Interactive and Vector-Based Phylogenetic Trees on the Web. *PloS one*, 5(8):e12267, 2010.

J. Van den Bussche, S. Vansummeren, and G. Vossen. Towards practical meta-querying. *Information Systems*, 30(4):317–332, 2005.