Rapid Prototyping of Semantic Mash-Ups through Semantic Web Pipes

Danh Le-Phuoc, Axel Polleres,
Manfred Hauswirth, Giovanni Tummarello
Digital Enterprise Research Institute,
National University of Ireland, Galway
Galway, Ireland

Christian Morbidoni
Univ. Politecnica delle Marche
Ancona, Italy

ABSTRACT

The use of RDF data published on the Web for applications is still a cumbersome and resource-intensive task due to the limited software support and the lack of standard programming paradigms to deal with everyday problems such as combination of RDF data from different sources, object identifier consolidation, ontology alignment and mediation, or plain querying and filtering tasks. In this paper we present Semantic Web Pipes that support fast implementation of Semantic data mash-ups while preserving desirable properties such as abstraction, encapsulation, component-orientation, code re-usability and maintainability which are common and well supported in other application areas.

Categories and Subject Descriptors:
D.1.1[Programming Techniques]: Applicative (Functional) Programming; H.4.0[Information Systems Applications]

General Terms: Algorithms, Design

Keywords: RDF, Pipes, Semantic Web, Mash-up

1. INTRODUCTION

Typical Semantic Web applications are data-intensive and require the combination and integration of RDF data from distributed data sources. The development of generic Web applications is well understood and supported by many traditional computer science domains, such as classical database applications. In current Web application development data integration and access are typically dealt with by fairly sophisticated abstractions and tools, supporting rapid application development and the generation of reusable and maintainable software components. The task of programming such applications has become the task of combining existing components from well-established component libraries, i.e., customizing and extending them for application-specific tasks. Typically, such applications are built built relying on a set of standard architectural styles which shall lower the number of bugs and ensure code that is easy to understand and maintain.

In contrast to that, data-intensive applications using RDF are currently mostly custom-built with limited support for reuse and standard functionalities are frequently re-implemented from scratch. Cumbersome, resource-intensive and error-prone tasks such as object identifier consolidation, ontology alignment and mediation ,or just plain querying and processing tasks are unnecessarily repeated in a lot of software

systems. While the use of powerful tools such as SPARQL processors, takes the edge off of some of the problems, a lot of classical software development problems remain. Also such applications are not yet built according to agreed architectural styles which is mainly a problem of use rather than existence of such styles. This problem though is well addressed in classical Web applications. For example, before the introduction of the standard 3-tier model for database-oriented Web applications and its support by application development frameworks, the situation was similar to a lot the situation that we see now with RDF-based applications.

In this paper we propose a flexible architectural style for the fast development of reliable data-intensive applications using RDF data. Architectural styles have been around for several decades and have been the subject of intensive research in other domains such as software engineering and databases. We base our work on the classical pipe abstraction and extend it to meet the requirements of (semantic) Web applications using RDF. The pipe concept lends itself naturally to the data-intensive tasks at hand by its intrinsic concept of decomposing an overall data-integration and processing task into a set of smaller steps which can be freely combined. This resembles a lot the decomposition of queries into smaller subqueries when optimizing and generating query plans. To some extent, pipes can be seen as materialized query plans defined by the application developer. Besides, the intrinsic encapsulation of core functionalities into small components, this paradigm is inherently well suited to parallel processing which is an additional benefit for high-throughput applications which can be put on parallel architectures or Grid environments.

The need for standard application development frameworks stems from the fact that an increasing amount of RDF data becomes available, for example, from widely used applications such as DBLP, DBpedia, blogs, wikis, forums, etc. that expose their content in different RDF-based formats such as SIOC [8] or FOAF [9], in the form of RDF/XML, or RDF statements embedded or extractable from HTML/XML pages by technologies such as GRDDL [12] or RDFa [1]. Despite the existence of standards and de-facto standards for publishing RDF, key problem in systems processing RDF are that the data (i) is fragmented, (ii) may be incomplete, incorrect or contradicting, (iii) partly follows ontologies, often with ontologies used wrongly or inconsistently, to name a few, and thus needs to be "sanitized" before it can be processed. A specifically cumbersome problem is the use of different identifiers denoting the same object which need to be unified. The processing steps required are often similar

across different applications and can in fact be encapsulated in customizable, reusable components which can be combined via a small set of base operators.

The pipes metaphor for the Web has been pioneered by Yahoo Pipes[1], which support the implementation of customized services and information streams through the combination of data sources (usually RSS feeds) with simple operators and processing constructs. Since the resulting Web pipes are themselves data sources, they can be reused and combined to form new pipes. Also, Web pipes are "live": they are computed on demand when requested via an HTTP invocation, and thus reflect an up-to-date state of the system (which can be detrimental as well in some scenarios where caching would be applicable).

Being mainly targeted to work with RSS feeds (item lists) though limits the applicability of Yahoo Pipes to the more general graph based data model of RDF. To this end, we come up with a pipes design, conceptual model and implementation that specifically targets graph based RDF data and allows the developer to quickly prototype (semantic) Web applications using RDF. Our system, called Semantic Web Pipes (SWP), emphasizes Semantic Web data and standards and enables the community-based generation of reusable Semantic Web processing components. By basing on common Web standards, pipes can be deployed on most common Web application servers. That means pipes can run on one machine or be distributed among an arbitrary number of nodes.

SWP offers specialized operators that can be arranged in a graphical Web editor to perform the most important data aggregation and transformation tasks without requiring programming skills or sophisticated knowledge about Semantic Web formats. This enables developers as well as end users to create, share and re-use semantic mash-ups.

1.1 Motivating Example

To give a concise overview of how SWP works, we sketch some of the main functionalities by giving a typical example of Semantic Web data aggregations from multiple sources.

We aim to aggregate data about Tim Berners-Lee from various sources on the Semantic Web (his FOAF file, DBLP, and DBPedia entries, etc.). However, we cannot simply merge these source, since all three sources use different identifiers for Tim. So we need to normalize the data before aggregation, for example by changing the URI used in DBPedia and DBLP to match with his self-chosen URI (the one used in his FOAF file). This job can be done by two SPARQL CONSTRUCT queries. For DBLP we query:

```
CONSTRUCT {<http://www.w3.org/People/Berners-Lee/card#i> ?p ?o.
          ?s2 ?p2 <http://www.w3.org/People/Berners-Lee/card#i>}
WHERE
{{<http://dblp.l3s.de/d2r/resource/authors/Tim_Berners-Lee> ?p ?o}
 UNION
{?s2 ?p2 <http://dblp.l3s.de/d2r/resource/authors/Tim_Berners-Lee>}}
```

For DBPedia we query:

```
CONSTRUCT {<http://www.w3.org/People/Berners-Lee/card#i> ?p ?o.
          ?s2 ?p2 <http://www.w3.org/People/Berners-Lee/card#i>}
WHERE
{{<http://dbpedia.org/resource/Tim_Berners-Lee> ?p ?o}
 UNION
 {?s2 ?p2 <http://dbpedia.org/resource/Tim_Berners-Lee>}}
```

The system should allow to encapsulate such queries in functional blocks (e.g. using the C-operator later described),

---

[1]http://pipes.yahoo.com/

which can then be connected to simple operators for fetching the data and to the output, obtaining a simple pipe that outputs the normalized RDF data. In Figure 1 we show a simple example of such a combination which solves our simple use case: URIs are normalized via the C-operators and then joined with Tim's FOAF file.

We observe that most of this example might be done in a single SPARQL query. However, besides the obvious advantage of modular design and reusability of basic building blocks as well as whole pipes for aggregations in more flexible ways, we will see in the course of this paper that the system offers many more useful features beyond the capabilities of single SPARQL queries. For instance, we allow the user to plug RDFS inference in between certain blocks (e.g., applying materialization of inferred triples for Tim's FOAF data, but not on the imported data); or, we allow to compute the input graphs for one SPARQL query by the output of another, i.e., dynamic addition resources to a pipe, etc.
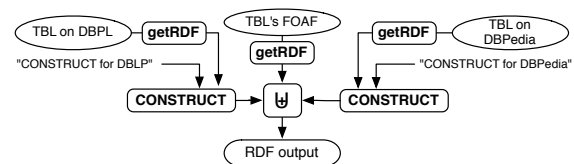

Figure 1: Workflow to aggregate Tim's SW data

In the remainder of this paper, we will first scetch the basic concepts and operators allowed in SWP (Section 2). Our implemented system that comprises an execution engine and enables the creation and modification of pipes in a graphical editor is described in Section 3: once saved in the editor, pipes are available at a stable URL and executable by users through a simple HTTP call. We qualitatively compare SWP against other available RDF Web application paradigma, namely single SPARQL queries and ad hoc Java scripting in Section 4. Related works, as well as an outlook to future work are presented in Sections 5 and 6.

2. BASIC CONCEPTS AND OPERATORS

The use cases we aim at addressing involve aggregation of data available on the Web (expressed in several RDF serialization formats, such as RDF/XML, RDFa, or microformats) as well as processing it in meaningful ways, that is, "filtering" and/or "transforming" the original RDF data. To this end Semantic Web Pipes introduce several base operators which can build up such a pipe that range from RDF extraction from existing Web content to SPARQL query processing or RDFS (and partial OWL) closure inference. In the following, we will introduce the basic definition of such operators and describe currently supported operators by example. We emphasize that the generic operator model of Semantic Web Pipes is extensible by new operators, some of which we will sketch at the end of this section.

A *Semantic Web pipe* as we define it implements a predefined workflow that, given a set of RDF sources (resolvable URLs), composes and processes them by means of pipelined special purpose operators. We do not aim at replacing complex workflow languages though, but rather promote a very reduced acyclic data processing model. More specifically, we only support two of the "classical" workflow patterns, namely split and merge.

All other base operators have exactly one output and one or more input parameters, each of which has a specific input

type. The set of base operators which we consider sufficient for most Semantic Web data aggregation tasks are shown in Fig. 2. All inputs in a pipe are either string text, RDF or other XML data. Parameters are named and may be instantiated either exactly once (*fixed parameter*) or arbitrary times (*variable arity parameter*) marked by "*" in the graphical representation in Fig. 2). As an example the SPARQL CONSTRUCT- and SELECT- operators may have exactly one default graph parameter (D), but an arbitrary number of named graph parameters ($N*$). Parameters can also either have single values or sequences as input marked by "?", an example being variable substitutions for the SPARQL CONSTRUCT- and SELECT- operators, see below. More details on the supported operator are given in Section 2.1. A *pipe* is a set of instances of the operators in Fig. 2, where:

- Each fixed parameter input and all variable arity parameter inputs are linked to either (i) quoted literals such as `"<?xml version="1.0" ?> ..."` for providing fixed string input (ii) a URL in angle brackets such as `<http://alice.example.org>` denoting a Web retrievable data source that contains data in the required input format, (iii) to the output of another pipe.

- All but one output (the "overall" output of the pipe) are linked inputs of other operators.

- Links between inputs and outputs are acyclic.

By following these constraints, each pipe can itself be used as an operator in another pipes. Apart from the graphical format implicitly given in Fig. 2, in Section 3 below, we will outline an XML format that allows to publish pipe "code" at an arbitrary URL to be re-usable as operators in other pipes. Furthermore, note that we do not constrain type mismatches between links of outputs of one operator or pipe to another operators: as described in Section 3, our system can handle several types of errors, e.g., timeouts while retrieving some of the sources or malformed source data, or ignoring non-RDF-convertible input, while still producing a valid output based on the remaining correct input data. The default behavior is to treat such unavailable/malformed inputs as empty, and likewise in certain cases, a wrong module input can cause the pipe processing to halt with an empty output.

For simplicity, we limit our model to pipes that do not contain cycles, leaving the study of cyclic pipes to future work.[2] We remark here that, as the current pipe engine is based on a XML tree model (see Section 3), such cycles are in fact not possible, at least within a single pipe.

2.1   Supported Operators

As mentioned above already, unlike fully-fledged workflow models, our current pipes model is a simple construction kit that consists of only two of the typical workflow constructs, namely split and merge, and a set of base *operators* which we will outline in the following section.

The Merge Operator: RDF Merge

This operator takes a arbitrary number of RDF graphs as inputs (variable arity parameter G*), expressed in RDF/XML, N3 [6] or Turtle [5] format, and produces an RDF graph that
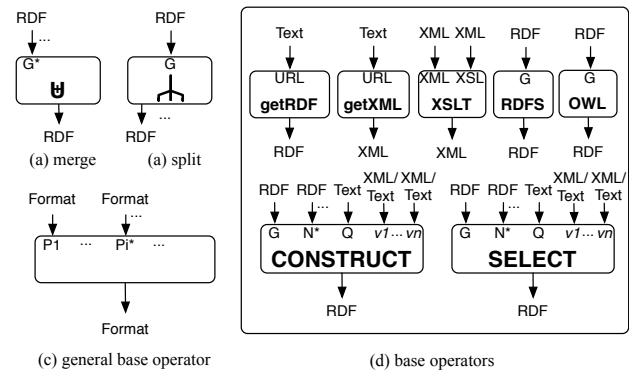

Figure 2: Semantic Web pipe operators.

is composed of the merge of its inputs. The standard implementation of the merge operator simply standardizes blank nodes apart, according to the RDF merge definition in [**?**].

The Split Operator

In order to enable reuse of outputs of other operators, we provide a split operator that allows to connect a single output to an arbitrary number of inputs of other operators. For an example, we refer to Fig. 4 below.

The getRDF *and* getXML *Operators*

These two operators take a text parameter URL as input which is supposed to retrieve a URL from the Web and convert it to RDF or XML respectively. Depending on the content type retrieved, for RDF extraction (getRDF) the following attempts are made: RDF/XML, Turtle, N3 can be consumed directly, HTML will be searched for RDFa or well-known microformats (XFN, hCal, hCard, hReview) for which a GRDDL extract is known. Likewise, explicitly mentioned GRDDL transforms will be executed. If none of these attempts succeeds an empty graph is returned. XML extraction (getXML) follows a simpler rule, retrieving HTML and tidying[3] it into XHTML, whereas other XML is left untouched, also possibly referenced XSL transformations are not executed, as this can be done explicitly by the XSLT operator covered next.

The XSLT *Operator*

This operator is useful for explicit execution of an XSL transformation on a particular XML input file. In the event that the XSL parameter is empty, the operator will attempt to execute an XSL transformation dereferenced in the XML input's prolog, otherwise the XML is passed through untouched. This operator is particularly useful when custom XML output formats are needed or when an input source in a custom XML format shall be transformed to RDF, e.g. when a GRDDL transform is not explicitly dereferenced.

### The SPARQL CONSTRUCT *and* SELECT *Operators*

The final and most essential operators in our framework are the SPARQL CONSTRUCT and SELECT operators which can help in aligning RDF data. Such queries can be used to extract only relevant information from a bigger graph as well as to 'align' two graphs, by smushing[4] identifiers or even performing basic ontology mapping operations [20].

The CONSTRUCT operator outputs the result of a construct query Q given as textual input performed on the SPARQL dataset [21] consisting of the default graph G and named graphs given in optional N* parameters. The input query allows full SPARQL CONSTRUCT syntax with the following differences for FROM/FROM NAMED clauses:

(i) the dataset may be implicitly given in the inputs by using the G and N* parameters. For illustration of this dataset generation, the CONSTRUCT operator corresponds to a native SPARQL CONSTRUCT query as follows:
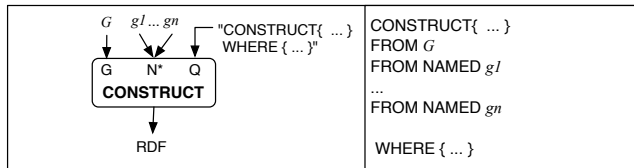
(ii) Variables linked via inputs may be used in FROM/FROM NAMED clauses, which we will explain in the following.

Variables occurring in the query in the CONSTRUCT, WHERE, FROM or FROM NAMED clause can be linked to inputs. This has the following effect: Depending on whether the input linked to variable v is a valid SPARQL result form [21] or not, either the query Q iterates over all result bindings for that very variable in that result form, replacing the occurrences of v in Q with respective result bindings for v in the input. This is handled by simply executing a respective XPath/Xquery extracting the respective sequence of bindings from the SPARQL result. If the same query result is linked (through the same split operator) to different variables input of the same subsequent query, then the iterations are as per solution, whereas variables bound to solution sequences of from different input queries are looped over nestedly executing the respective query building a Cartesian product. The results of these nested loops of CONSTRUCT queries are merged into the same output RDF graph for the CONSTRUCT operator.

The SELECT operator works analogously, but returning a SPARQL result format XML document. Again results of these nested loops are appended in the same SPARQL result format XML document.

This semantics allows us to stack SPARQL queries into each other. Thus pipes can emulate simple FOR-loops, without offering the full expressivity of procedural languages.

When a textual or other non-SPARQL result format input is linked to a variable input parameter simply all occurrences of the variable in the query Q are replaced with textual value as an RDF plain literal in the query.

We illustrate the use of input variables in the SPARQL query operator with a simple example in Fig. 3. This pipe extracts the `foaf:knows` relations in Tim Berner's Lee social "cloud" derived from RDF files linked in Tim Berners-Lee's own FOAF file via `rdfs:seeAlso` links. Such dynamic generation of the dataset or iteration over the results of one SPARQL query and processing them in another is not doable in "plain" SPARQL but easy in pipes.

As a last example for the use of operators SPARQL, let us turn to a pipe which uses the SPLIT operator, as shown in Fig. 4. This pipe first retrieves ISBN numbers of contributions in books by Tim Berners-Lee from DBLP by a respective SPARQL query. Then we assume to have RDF graphs of book prices for Amazon and Barnes & Noble where we search for these ISBN numbers and respective prices by further SPARQL queries. The results of the first query are
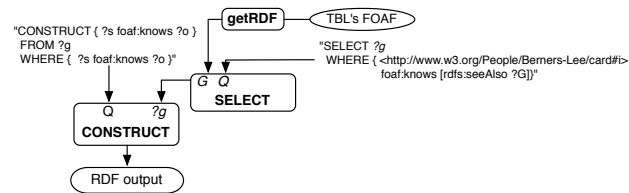

Figure 3: A pipe which iterates over results of one SPARQL operator and processes results in another.
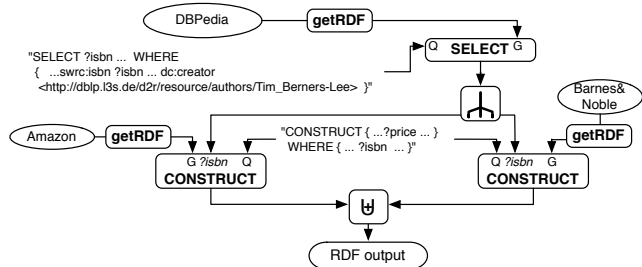

Figure 4: Tim's Books: Using the Split operator.

split to the inputs of the subsequent queries. Note that we used - for illustrative purposes - the same sketched query (constructing price information in RDF per ISBN) for both Amazon and Barnes & Noble in Fig. 4, but of course these queries could look completely different in practice. In fact, neither Amazon's nor and Barnes&Noble's data is currently available as one huge RDF graph, but it is in fact a more likely assumption that such data will only become available behind SPARQL endpoints in the future. In that case the separate queries could be pushed separately to those two endpoints, whereas a joint query on the merged data would need an intelligence in the query-optimizer to split off the distributed query (see e.g. [22]). Depending on how much information on the endpoints is available to the outside, it might be impossible for an automatic optimizer to come up with the right partition of the query. In such scenarios, a human who has experience with each of the endpoints may still be able to find and design an optimized query plan manually using the pipes paradigm.

The SELECT operator in conjunction with the XSLT-operator can easily produce RSS feeds or other XML dialect representations, thus acting as an adapter for non-Semantic Web applications.

We restrict ourselves, at the moment, to a standard SPARQL query engine (which is also what we currently deploy in our implementation). In previous work though, we have shown that "pure" SPARQL has several limitations which also affect our envisioned use cases. For instance, SPARQL does not allow complete mappings even between simple RDF vocabularies [20] such as FOAF and vCard. Conceptually, we could equally replace the existing SPARQL operators with more expressive extensions such as SPARQL++ [20] or XS-PARQL [2], which allow for aggregate functions or string manipulations when creating new RDF terms in CONSTRUCT queries, or for creating arbitrary XML directly out of RDF. Such extensions would also further minimize the necessity of intermediate applications of custom XSLT operators within pipes.

The RDFS and OWL Operators

These two operators basically perform materialization of the RDFS or OWL closure of the input graph by applying RDFS
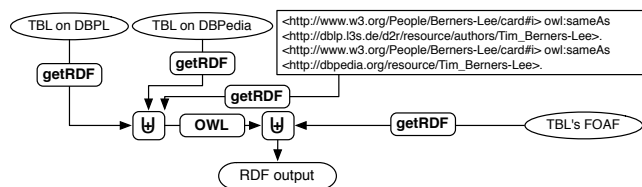
Figure 5: Aggregate Tim's data by OWL inference.

or partial OWL inference rules. Here, we mean finite materialization of entailed triples that can be done by simple forward chaining rules. Although this method will not infer e.g. all of the (infinitely many) axiomatic triples, in most cases a well enough approximation of RDFS which can be implemented by common rules engines, such as Jena Rules[5] or Datalog engines. Treating RDFS inference in pipes as a separate operator (decoupled from, e.g., SPARQL rules evaluation) is a pragmatic approach, as we do not fix at which part of the chain inference shall be made. Other RDFS or OWL fragments that can be approximated by materializing finitely evaluatable inference rules, like ρdf [18] on the lower end and the Horst's pD* rule set[24] on the upper end, could easily be plugged into our framework by similar operators, currently we support the latter by employing Jena OWL reasoner in the OWL operator.

The pipe in Fig. 5 for instance shows how a simple OWL inference operator can be used instead of SPARQL queries to achieve the same result as the pipe in Fig. 1. Note that this formulation reveals another strength of pipes: Unlike available SPARQL engines with partial OWL or RDFS reasoning support, where materialization always applies to all or none of the graphs mentioned in a query, in a pipe the designer can decide exactly where and on which intermediate graphs inference is done. We consider this as an advantage especially when dealing with large graphs, where one can first filter out relevant only triples in a SPARQL query and then apply inference only on a reduced dataset in a pipe.

Clearly, the base operators described so far are only a subset of the operators that can be thought useful in mashing up information on the Semantic Web. Nonetheless, the concept of an operator is general enough for more extensions in the future. One such useful operator might be a general wrapper for WSDL described Web service operations, which, being defined by inputs and outputs in specific XML formats, can be easily mapped to the operator metaphor we adopt here.

3. SYSTEM DESIGN & IMPLEMENTATION

This section describes and discusses the system design and its prototypal implementation which is available for experiments at `http://pipes.deri.org`.

3.1 System design

Obviously, our basic concepts and operators structure described above fits well into the popular pipes and filters pattern and architecture style [11, 23]. Another reason to choose the pipes and filters (or pipeline) structure is that it has been commonly used and well investigated in parallel programming environments such as [15, 14, 3], i.e. implementations of such modular pipes are parallelizable. The most widely known and used instances of pipes and filter architectures we refer to in our design are Unix shell scripts
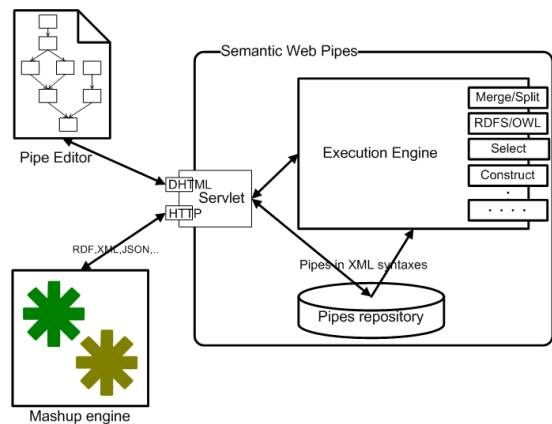

Figure 6: Semantic Web Pipes Architecture

and programs [4]. Similar to Unix, Semantic Web Pipes (SWP) support this architectural style by providing the notation in Section 2 for connecting operators as components (same as Unix processes) and by providing a run time execution engine for processing pipes.

We implemented the pipes and filter architecture style in SWP by adopting the architecture depicted in Fig. 6. As a typical property of *pipes and filter* systems mentioned in [23], this architecture enables the pipe designer to see overall input/output behavior of a pipe as a simple composition of the behaviors of the individual operators (*filters*).

Visually, a pipe is a workflow of connected operators. Hence, similar to scientific workflow editors such as Kepler, Taverna, etc[6], we provide a lightweight Web-based workflow editor called *pipe editor* (Fig. 7) for composing pipes by drag&drop and wiring operators inputs and outputs together, inspired by the simplified graphical representation chosen in Fig. 2. Using the pipe editor we aim at providing a programmable Web environment suitable for non-expert programmers. The designer does not need full programming skills to construct an arbitrary control flow of data he/she wishes to aggregate. Moreover, the Web-based WYSIWYG GUI enables designers to inspect the output of each single operator composing the pipe.

On top of that, an important aspect of the graphical Web editor is to foster reuse of previous pipe designs. Pipes once created can be published and then browsed, searched and re-used by other users. At any time, a new pipe can be derived as a modification of an existing one or an existing pipe can be included as a functional block in a new one: since the output of a published pipe is an HTTP-retrievable RDF model or XML file, simple pipes can work as sources for more complex pipes. Furthermore, as pipes can take parameters as input (parametric pipes are discussed next in Section 3.2), they can act within other pipes not only as data sources but as full featured, custom operators.

After visually designed, a pipe is serialized and stored in a custom XML format and can be later loaded into the editor or run by the server-side execution engine. SWP uses a simple database to store these XML representations of pipes in the *pipes repository*. When a pipe is invoked, the execution engine reads its specification from the repository and maps the XML representation to concrete invocations of the operators' implementations for execution. The engine fetches data from remote sources into an in-memory triple store,

---

[5] `http://jena.sourceforge.net/inference/`

[6] `http://kepler-project.org/`,`http://taverna.sourceforge.net/`

then executes the defined workflow. Each intermediate operator has its own RDF triples or XML infoset buffer where it can load data from input operators, execute SPARQL queries, materialize implicit triples of RDFS or OWL closures, filter and transform triples or XML trees, etc. SWP naturally supports concurrent execution because each operator can be implemented to run as a separate task and potentially be executed in a distributed fashion in parallel with other operators to improve scalability.

While it would be possible to store pipe descriptions themselves in RDF, our current ad hoc XML language is more terse and legible. If an RDF representation of pipes will be later needed, it can be possibly obtained via GRDDL.

A caching mechanism is deployed to avoid repeated fetching of remote resource and redundant execution of pipes when the input data has not changed between invocations of the pipe. Whenever content is fetched its hash value is calculated. When no changes in the hash value are detected the cached result is returned. Because it usually takes significant time to download data from remote resources, the cache engine also stores a parsed copy of the input RDF to a persistent triple store.

Circular invocations of the same pipe, which could create denial of service, can be easily detected within the same pipe engine, but not when different engines are involved.[7] Our solution for such cases relies on extra HTTP headers: whenever a model is fetched coming from an another pipe engine, an HTTP GET is performed putting an extra *PipeTTL* (time to live) header. The TTL number is decremented at each subsequent invocation. A pipe engine refuses to fetch more sources if the PipeTTL header is ≤ 1.

Finally, thanks to HTTP content negotiation, humans can use each Semantic Web Pipe directly through a convenient Web user interface. The pipe output format depends on the HTTP header sent in the request. For example, RDF-enabled software can retrieve machine-readable RDF data, while users are presented a richer graphical user interface to browse the pipe results.

3.2 Implementation

To describe our prototype implementation, we refer to the example pipe described in Section 1, that is available online at `http://pipes.deri.org:8080/pipes/pipes/?id=TBLonTheSW`. The pipe can be displayed and edited in the Web pipe editor, as shown in Fig. 7, wiring operators in the GUI without having to know any specific syntax, except for the SPARQL language that is used to configure some of the functional blocks.

The visual editor offers some more features to speed up the development process and help in keeping code consistent. First, the output of each operator can be connected only to those operators which allow a compatible data format as input. This type checking performed at design time in the editor ensures consistency of the pipeline, avoiding developers to establish wrong connections between functional blocks. Second, the data processed can at any step of the pipeline be inspected by executing sub-parts of the pipe, offering a useful tool for managing and debugging pipes, e.g. individuate an operator with empty results due to a semantically wrong query.

---

[7]Although cycles are disallowed within a pipe specification, such circular invocations could occur, by a pipe recursively invoking itself as an operator.
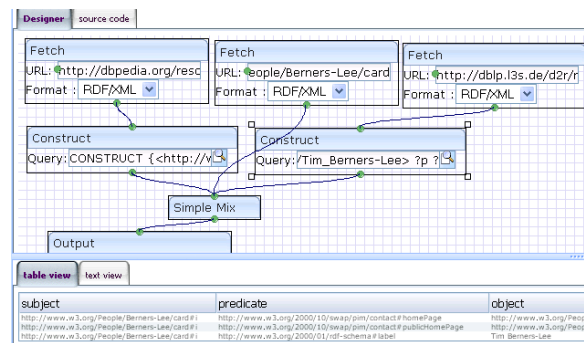

Figure 7: Pipe Editor with Tim's pipe

Pipes are represented and stored as XML documents, using an XML-based Domain Specific Language documented on the project Web site[8]. The current format is simple and easily extensible, thus enabling sharing and reuse of pipes among different system installations. While an in-depth discussion of the XML syntax is beyond the scope of this paper, we give some syntactic "teaser" of the XML generated to represent the example pipe in the following; here, `Q1` and `Q2` stand for the CONSTRUCT queries shown in Section 1.

```
<merge>
 <source><getRDF>
  <url>http://www.w3.org/People/Berners-Lee/card#i</url>
 </getRDF></source>
 <source><construct>
  <source><getRDF>
   <url>http://dblp.l3s.de/.../Tim_Berners-Lee</url>
  </getRDF></source>
  <query> <![CDATA[ Q1 ]]> </query>
 </construct></source>
 <source><construct>
  <source><getRDF>
   <url>http://dbpedia.org/.../Tim_Berners-Lee</url>
  </getRDF></source>
  <query> <![CDATA[ Q2 ]]> </query>
 </construct></source>
</merge>
```

In general, it can be useful to add parameters as input to a pipe, for example to build a pipe that grabs information about a city specified by a user or by an application using the pipe REST API. In our system a pipe can declare multiple parameters and take their values as input via HTTP GET requests. The following is typical example code of a parametric pipe. As soon as this pipe is stored into the pipe repository, it is reusable as a new operator.

```
<pipe>
<parameters> <parameter>
 <id>name</id><label>City Name</label><default>London</default>
 </parameter> </parameters>
<code>
 <merge>
  <source><getRDF>
   <url>..sindice.com...lookup?keyword=${name}..</url>
  </getRDF></source>
  <source><getRDF>
   <url>...geonames.org/search?name=${name}&...</url>
  </getRDF></source>
 </merge>
 </code>
</pipe>
```

The default output of most pipe operators is RDF. However, using RDF/XML in light weight, client-side Ajax applications would be associated with significant difficulties and performance problems. Hence, the our implementation supports RDF/JSON output for client side processing by Javascript. We also support other RDF formats such as Turtle/N3, TRIG, TRIX, NTRIPLES which might be more suitable to some applications than RDF/XML. Apart

---

[8]`http://pipes.deri.org/index.php/documentation`

from that, SWP allows user to visualize a pipe's output in a faceted browser by integrating SIMILE Exhibit [9], a Javascript library that lets users explore the RDF output interactively. Furthermore, users are able to specify parameters for pipes in the provided GUI and can directly check the results in this faceted browser. As a complement solution to other mash-up platforms which are using XML/RSS as input formats (see Section 5), SWP also provides RSS output to feed into such engines; we provide some example Yahoo's pipes which consume RSS and JSON data from SWP's pipes on the SWP homepage.

Our current execution engine not only supports the base operators from Figure 2 but also more advanced native operators such as patching RDF graphs [17], smushing URIs based on `owl:sameAs` relations.

We invite developers to contribute to the SWP open source project (`http://sourceforge.net/projects/semanticwebpipe`). SWP is not only aiming at Semantic Web developers for contributing new native operator implementations but also shall provide a framework to integrate off-the-self Semantic Web data processing components as visual ready-to-use operators. For example, we are implementing wrappers to encapsulate available "RDFizing" components and services (`http://simile.mit.edu/wiki/RDFizers`, `http://www.opencalais.com/`, etc) which are able to extract RDF data from popular formats such as CSV, BibTEX, plain text,etc. These operators shall enable users to access vast amount of data sources in their SWP mash-ups.

4. EVALUATION

In this section we preliminarily evaluate the SWP framework using three methodologies. First, we analyze how the framework applies to three practical data mash-up scenarios and a combination of metrics is used to compare SWP with other SW data aggregation solutions. We then proceed with a qualitative evaluation based on the cognitive dimensions notations (CDs), a methodology that has previously been used to evaluate related approaches. Finally, we discuss related aspects which are not covered by the previous two points.

4.1 Test Case Studies

We start by preliminarily evaluating the complexity of solutions to different use case problems when implemented using Semantic Web Pipes as opposed to directly using SPARQL, when possible, or implemented as custom Java programs.

For the Java paradigm we measure complexity using a measure of Line of Code equivalent (LOCE) which is obtained as sum of the number of lines of code in Java plus the LOCE of complex SPARQL queries. For deriving a LOCE equivalent for SPARQL queries, we make the assumption that each triple pattern element or query operator is meant to address a specific part of the data transformation, something that can be considered a lower bound to what a single line of code in a general purpose language could do. For example a query with two triple patterns joined over a common variable is considered to be 3 LOCE.

For SWP we measure the number of operators and provide a complexity measure also in LOCE. In this case, we calculate the pipes LOCE as 1 per block, plus one per each simple parameter changed in a block from the default, one per block to block interconnection, one per use of a pipe pa-

---

Table 1: **Complexities in LOCE. The numbers in parenthesis represent how many LOCE are consumed by SPARQL queries, if these are used inside the implementation.**

	Java implemetation	SPARQL	SWP
Case 1	174 (30)	48 (48)	44 (30)
Case 2	214 (49)	NA	63 (49)
Case 3	241 (43)	NA	72 (43)

rameter inside block plus the LOCE of any SPARQL queries possibly used inside pipe operators.

Let us note that while lines of code are a controversial measure of algorithmic complexity, in our context their use is limited to that of counting the number of atomic actions that a developer would have to make to reach a desired objective. With respect to this, these numbers help making qualitative, rather than quantitative evaluations of the framework's usability.

Case 1: "Tim Berners Lee on the Semantic Web"'
Plenty of data is available about persons like Tim Berners-Lee on the Semantic Web: in this case study we want to obtain a single RDF graph aggregated from his FOAF file with personal data, DBLP entries with bibliographic information and DBpedia data. This data cannot simply be merged since the URIs used in the original sources do not match. This case can be solved by all three compared approaches.

Case 2: Friend's publications
Given one's FOAF file and a conference name, it will use data.semanticweb.org to see which friends published a paper at that conference. The script will needs to resolve people's URIs to get the full name.

Case 3: SIOC Aggregation RSS feed
Given a SHA1 a user's email, the output is a list of messages that the user left on the internet in sites that expose SIOC [8] data. To do this the pipe needs to execute a Sindice [19] query which will return a set of documents likely containing the description of messages left by the user on possibly multiple Web sites. It will then aggregate them and compose an RSS feed.

The results of this implementation tests achieved by an experienced developer are reported in table 1.

Not surprisingly, Java required the highest number of LOCE by far when solving each of the scenarios. Each mash-up operation required several instructions to properly instantiate the frameworks/libraries which perform tasks like Web data fetching, RDF processing, querying etc. Iterating over results to perform further processing also required specific programming which added to the count. We notice that the LOCE of Java code are 3 to 4 times as many as the LOCE of the SPARQL queries which are embedded.

With respect to the pure SPARQL query programming, we notice that even in the case where this can be done in a single expression, these queries tend to be unnecessarily complex. As an example, let us have a look at following unified query which "emulates" the pipe of Figure 1.

Squeezing the independent query blocks in one large UNION query seems unintuitive compared to the clearly separated blocks in a pipe. Also a query engine needs to decompose these blocks automatically in its optimizer to achieve reasonable performance. We deem it to be more natural to write this query as a pipe – apart from saving LOCE – and pipes may be viewed as visualization of a concrete, manually

optimized "query plan". Other tasks, especially thus involving iterations over results, are not expressible in SPARQL alone. Although one could use a more complex language such as XSLT, XQuery or XSPARQL [2], all of which are Turing complete, to script those tasks, we deem again pipes to be easier to design, maintain and comprehend.

```
CONSTRUCT{
  <http://www.w3.org/People/Berners-Lee/card#i> ?p1 ?o1.
  ?s2 ?p2 <http://www.w3.org/People/Berners-Lee/card#i>.
  <http://www.w3.org/People/Berners-Lee/card#i> ?p3 ?o3.
  ?s4 ?p4 <http://www.w3.org/People/Berners-Lee/card#i>.
  ?s5 ?p5 ?p6}
FROM NAMED <http://dblp.l3s.de/d2r/.../Tim_Berners-Lee>
FROM NAMED <http://dbpedia.org/resource/Tim_Berners-Lee>
FROM NAMED <http://www.w3.org/People/Berners-Lee/card>
WHERE {
  {GRAPH <http://dblp.l3s.de/d2r/.../Tim_Berners-Lee> {
    {<http://dblp.l3s.de/d2r/.../Tim_Berners-Lee> ?p ?o}
    UNION
    {?s2 ?p2 <http://dblp.l3s.de/d2r/.../Tim_Berners-Lee>}}}
  UNION
  {GRAPH <http://dbpedia.org/resource/Tim_Berners-Lee> {
    {<http://dbpedia.org/resource/Tim_Berners-Lee> ?p3 ?o3}
    UNION
    ?s4 ?p4 <http://dbpedia.org/resource/Tim_Berners-Lee>}}}
  UNION
  {GRAPH <http://www.w3.org/People/Berners-Lee/card>
    { ?s5 ?p5 ?p6 }}
}
```

Using the Pipes paradigm, the tasks are completed by composing the available blocks. We observe that the low LOCE is an indication of the minimal overhead of the framework with respect to the problems. When SPARQL queries are used inside pipes (case 2 and 3), these take most of the complexity. In the worse case, a little more than half a LOCE is used for each LOCE of SPARQL used (case 3 with 43 SPARQL LOCE over a total of 72).

Finally, as more pipe operators are implemented, the tendency is that there will be lesser need to use SPARQL queries. For example we report that an implementation of case 1 can be done without SPARQL, but an additional URI smushing operator using only 18 LOCE.

The full details of this evaluation including the Java source code, the SPARQL queries and the corresponding Semantic Pipes are available online[10].

4.2 Cognitive Dimensions of Notations

In this section we apply the Cognitive Dimension of notations methodology (CDs) [7] to evaluate the usability of Semantic Web Pipes. CDs, which have been notably used to evaluate other Web mash-up approaches [13], is a subjective test composed by a set of terms and concepts which over time have established themselves as important by programmers. These concepts can be listed as follows (from Peyton Jones et al. [16]):

- **Abstraction gradient** What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
- **Consistency** When some of the language has been learnt, how much of the rest can be inferred?
- **Error-proneness** Does the design of the notation induce "careless mistakes"?
- **Hidden dependencies** Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
- **Premature commitment** Do programmers have to make decisions before they have the information they need?

---

[10] http://pipes.deri.org/index.php/evaluations

- **Progressive evaluation** Can a partially-complete program be executed to obtain feedback on "how am I doing"?
- **Role expressiveness** Can the reader see how each component of a program relates to the whole?
- **Viscosity** How much effort is required to perform a single change?
- **Visibility and Juxtaposability** Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

With respect to these dimensions, they derive the following dimension assessment:

Abstraction Gradient: Semantic Web Pipes provide basic building blocks which match the basic operations in Semantic data mash-ups, e.g. fetch, add, query, transform. When a pipe is completed, it can be encapsulated in a new single drag and drop operator and in general at HTTP level can be used by any software which fetches data as if it was a static source.

Consistency: Mash-ups are developed in pipes using the consistent paradigm of dragging/dropping/connecting/ configuring the blocks. Certain blocks, however, might have internal configurations which might include additional language complexity, e.g. writing SPARQL queries. Here, at least, we rely on existing SW language paradigms, which we try to simplify. Consistency is a key aspects of Semantic Web Pipes which puts most, possibly all, the instruments for semantic mash-up into a single, consistent, environment.

Error Proneness: Probably one of the most important advantages of the pipe paradigm is the ability to provide upfront step-by-step debugging; block compositions are guided preventing syntax mistakes. This lowers the entry barriers for experimentations and increases the framework learning curve.

Hidden Dependencies and Role Expressiveness: The connection between one block and another is graphically displayed such that all dependencies are explicit.

Premature Commitment: Currently, the commitments required are limited to the use of standards in data representation and transport. To be processed, data must be available either in RDF or in one the formats that are supported by the operators installed in the runtime (e.g. microformats, XML etc). For transport, pipes requires and provides HTTP as a way to access/return data. These limitations are due to the current runtime and can be overcome with by extending the framework with conversion operators to support more formats and protocols.

Progressive Evaluation: The visual pipe programming paradigm provides excellent support for progressive evaluation: each operator in the editor has a partial *run* button that shows the debug output of the pipe up to that block.

Role expressiveness, Visibility and Juxtaposability: The visual pipes programming mode emphasizes these features: roles are visualized by connections between operators, visibility is limited by screen size only (and can be aided by zoom functionality in the editor), code execution flows can be read both top down and bottom up.

Viscosity: Based on functional programming, pipes represent almost literally the principle of encapsulation and decoupling. A change in the format or location of one of a data

source, for example, should only require a modification at the respective getXML/getRDF operator at the beginning of the pipe.

4.3 Performance Issues

With respect to performance evaluation, exact performance results not to be very significative as they are mostly dependant on the implementation.

It is however interesting to discuss in general the model related aspects of the pipe execution performance and which optimizations can be applied.

By definition, with respect to a single input/single output scenario, e.g. a single, cold cached invocation of the execution framework, nothing can be more efficient than a software solution developed specifically for a specific problem at hand. With respect to this, a purely programmed solution would always be at least as fast as a solution composed by pipelining predefined operators.

Pipes however makes it easy and transparent to use many of the optimization opportunities of the *pipes and filters* processing model.

Firstly, the pipeline model allows execution of multiple branches that in parallel, e.g. by different threads and processor cores.

Also the purely functional model of each operator, or block composed by multiple operators, allow full or partial results to be cached, reused and shared across multiple execution threads. This caching happens both internally at execution level, a process also known as memoization, and externally: as the invocation happens using the REST model it would be straightforward to arbitrarily increase the aggregated performance of a Semantic Web Pipelining cluster simply by using multiple servlets and roundrobin Web request routing techniques.

We report that the current Semantic Web prototype, while more concerned with flexibility and functionalities than performance, does currently implement memoization and internal multithread parallelization, e.g. in the FOR loop operator where a thread pool is used to execute the inner FOR pipe branchs.

5. RELATED WORK

The term "pipeline" (or "pipe") is well known in Computer Science and denotes a chain of data processors where the output of each of these processors is sent as an input to the next one. The most famous implementation of this generic concept is within the UNIX console [4].

The pipeline paradigm has been successfully applied to XML transformation workflow definition and execution in projects like Apache Cocoon[11], and several languages, mostly based on XML themselves, have been used by vendors to represent such workflows[12]. A standardization effort is currently ongoing within the W3C XML Processing Model Working Group, which recently produced the XProc language specification [25]. XProc relies on XSD, XSLT, XPath and XQuery to perform several operations on XML documents, e.g. validation, aggregation, transformation and filtering, and provides classical programming language operators as if-then and for-each constructs. The XML language proposed in this paper bases similar ideas but on different technologies

(e.g. SPARQL, RDFS, OWL) for querying and modifying data, as RDF, although can be syntactically represented in XML, has an inherently different data model. While semantic processing is in general more expressive and powerful, we think that supporting pure XML processing, as we do with the XSLT-operator, is crucial for applying Semantic Web Pipes to real world scenarios.

The Yahoo Web Pipes framework, as mentioned earlier, was inspiring for our work. It provides a rich set of operators and uses RSS as main data input and exchange format among operational blocks, but lacks functionality to address our desired use cases. Other implementations of the same idea have appeared recently: DAMIA[13], from IBM, is a partially Web based commercial product for composing mash-ups of Web applications and data. Microsoft Popfly[14] has similar features and provides a rich user interface plus a number of predefined wrappers for a variety of different data Web sources as Facebook, Geonames, Google Maps.

A particularly interesting application is Intel MashMaker, recently released as a beta version and described in [13]. The basic idea is that of providing users with a graphical interface to create, while they browse the Web, functional programs where input data is taken from visited Web sites and the output is a sort of spreadsheet presenting the aggregated data. Although the idea is very interesting, the impression is that the software, explicitly targeted to non technical every-day users, is still too complex and requires considerable effort to create non-trivial mash-ups.

Concerning the Semantic Web world, the need for a cascade of operators to process RDF repositories is also addressed by the SIMILE Banach project. Banach operates inside the Sesame DB by leveraging its capabilities to have a pipelined stack of operators which can both process data and rewrite queries. Only few basic operators have been developed so far, but plenty of possibly useful operators are discussed in the Web site.[15] While some of these are already covered by the C-operator, others are not, but might be considered in future releases of our pipe engine.

Finally, TopBraid has recently developed a module (called SPARQLMotion[16]) of its Composer platform that implements an idea similar to the one discussed in this paper. It provides a visual canvas where RDF sources and transformation blocks, mostly based on the SPARQL language, can be arranged in pipelines to produce the desired output. If the functionalities provided seems comparable to the ones implemented in our system, unlike Semantic Web Pipes, the SPARQLMotion is a commercial, closed-source, desktop application based on Eclipse RCP. As confirmed by the previously mentioned projects, we believe that a Web based approach better fits the target scenario, being integrated in an environment (the Web browser) that the user knows well and uses every day, and fostering sharing and reuse of pipes and operators. On the other hand, we believe a valuable contribution of our project resides in its open-source nature, that makes it possible to extend the framework with new ad-hoc operators and provides architectural and implementation guidelines that can be of help for future research and development in this field by the Semantic Web community.

---

[11] http://cocoon.apache.org/

[12] http://www.orbeon.com/ops/doc/reference-xpl-pipelines, http://www.oracle.com/technology/tech/xml/xdkhome.html

[13] http://services.alphaworks.ibm.com/graduated/damia.html

[14] http://www.popfly.com

[15] http://simile.mit.edu/wiki/Banach

[16] http://www.topquadrant.com/sparqlmotion/

6. CONCLUSIONS AND FUTURE WORK

Semantic Web data is a general purpose representation which needs to be tailored to particular application requirements and needs. In the absence of better tools, these transformations are usually implemented in an ad-hoc manner using general purpose programming languages. The necessary solutions are often tedious to program, cumbersome to debug and hard to maintain. To remedy this situation, this paper presented Semantic Web Pipes (SWP), a conceptual framework for rapid prototyping of Semantic Web data mash-ups.

Semantic Web Pipes have the additional benefit that they can be implicitly used as access points for linked data without additional efforts. A Semantic Web Pipe consumes on-line data and each published pipe itself becomes a Semantic Web source that can be used for other mash-ups. Live execution on invocation guarantees that the data returned by a pipe reflects the latest state of the underlying Web data, a major advantage of dynamic data transformations and workflows as opposed to offline processing of data which would imply duplication and inconsistency, but at the cost of additional processing, which, however, is far outweighed by the benefits of SWP in our opinion.

Pipes can be stored, exported and reused, using a custom XML format and developers can use SWP mash-up capabilities inside their standalone Web applications by simply importing the pipe execution engine as a jar archive. SWP extends SPARQL by using workflows. We have illustrated the advantages of pipes compared with "pure" SPARQL and discussed several optimizations.

SWP are open source, which allows developers to extend the framework with new operators as needed. Since its original announcement SWP have received numerous contributions in terms of operators by third party developers. Moreover, on the project website, several mash-up demonstrations created by the user community are available in such diverse domains as health care & life sciences, social networking, online communities and more.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] B. Adida, M. Birbeck, S. McCarron, S. Pemberton (eds.). RDFa in XHTML: Syntax and Processing, Oct. 2008. W3C Rec.

[2] W. Akhtar, J. Kopecky, T. Krennwallner, A. Polleres. XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. ESWC2008, Tenerife, Spain, June 2008. Springer.

[3] G. R. Andrews. *Foundations of Parallel and Distributed Programming.* Addison-Wesley Longman Publishing,1999.

[4] M. J. Bach. *The design of the UNIX operating system.* Prentice-Hall, 1986.

[5] D. Beckett, T. Berners-Lee. Turtle - Terse RDF Triple Language. W3C Submission, Jan. 2008.

[6] T. Berners-Lee, D. Connolly. Notation3 (N3): A readable RDF syntax. W3C Submission, Jan. 2008.

[7] A. F. Blackwell, et al. Cognitive dimensions of notations: Design tools for cognitive technology. 4th Int'l Conf. on Cognitive Technology (CT'01), London, UK, 2001. Springer.

[8] U. Bojārs, et al. SIOC Core Ontology Specification. W3C Submission, June 2007.

[9] D. Brickley, L. Miller. FOAF Vocabulary Specification 0.91, Nov. 2007. http://xmlns.com/foaf/spec/.

[10] J. Carroll, C. Bizer, P. Hayes, P. Stickler Named graphs. Journal of Web Semantics 3(4):247–267, 2005.

[11] J. O. Coplien and D. C. Schmidt (eds.). *Pattern languages of program design.* ACM Press/Addison-Wesley Publishing, 1995.

[12] D. Connolly (ed.). Gleaning Resource Descriptions from Dialects of Languages (GRDDL), Sept. 2007. W3C Rec.

[13] R. Ennals, D. Gay. User-friendly functional programming for web mashups. 12th ACM SIGPLAN Int'l Conf. on Functional programming, 2007.

[14] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing, 1995.

[15] D. C. Hyde, G. M. Schneider, C. H. Nevison. *Laboratories for Parallel Computing.* Jones and Bartlett Publishers, Inc., USA, 1994.

[16] S. P. Jones, A. Blackwell, M. Burnett. A user-centred approach to functions in excel. ACM SIGPLAN ICFP'03, New York, NY, USA, 2003. ACM.

[17] C. Morbidoni, A. Polleres, and G. Tummarello. Who the FOAF knows Alice? RDF Revocation in DBin 2.0. 4th Italian Semantic Web Workshop (SWAP), 2007.

[18] S. Muñoz, J. Pérez, C. Gutierrez. Minimal deductive systems for RDF. ESWC 2007, Innsbruck, Austria, June 2007.

[19] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello. Sindice.com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies 3(1):37–52, 2008.

[20] A. Polleres, F. Scharffe, R. Schindlauer. SPARQL++ for mapping between RDF vocabularies. ODBASE 2007, Vilamoura, Algarve, Portugal, Nov. 2007.

[21] E. Prud'hommeaux, A. Seaborne (eds.). SPARQL Query Language for RDF, Jan. 2008. W3C Rec.

[22] B. Quilitz, U. Leser. Querying distributed RDF data sources with SPARQL. ESWC 2008, Tenerife, Spain, June 2008. Springer.

[23] M. Shaw, D. Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[24] H. J. ter Horst. Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary. Journal of Web Semantics, 3(2):79–115, 2005.

[25] N. Walsh, A. Milowski. Xproc: An xml pipeline language. W3C Working Draft, April 2008.