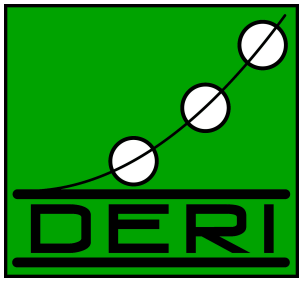



# Semantic Web Pipes

Christian Morbidoni Axel Polleres
Giovanni Tummarello Danh Le Phuoc

DERI Technical Report 2007-11-07
November 2007



**DERI Galway**
IDA Business Park
Lower Dangan
Galway, Ireland
http://www.deri.ie/

DERI Technical Report

DERI Technical Report 2007-11-07, November 2007

# Semantic Web Pipes

Christian Morbidoni[1] Axel Polleres[2] Giovanni Tummarello[3]
Danh Le Phuoc[4]

**Abstract.** This report presents Semantic Web pipes, a powerful paradigm to build RDF-based mashups. Semantic Web pipes work by fetching RDF models on the Web, operating on them, and producing an output which is itself accessible via a stable URL. We illustrate how Semantic Web pipes can solve use cases ranging from simple aggregation to complex collaborative editing and filtering of distributed RDF graphs. To this end, we introduce the concept of RDF revocations and describe a pipe operator that can apply such revocations. This operator enables Semantic Web pipes where agents cooperatively develop semantically structured knowledge, while still retaining and publishing their individual contributions and beliefs. We conclude with a description of two available implementations.

[1]SeMedia Group, Univ. Politecnica delle Marche,
Ancona, Italy. E-mail: christian@deit.univpm.it
[2]DERI, National University of Ireland,
Lower Dangan, Galway, Ireland. E-mail: axel.polleres@deri.org
[3]DERI, National University of Ireland,
Lower Dangan, Galway, Ireland. E-mail: giovanni.tummarello@deri.org
[4]DERI, National University of Ireland,
Lower Dangan, Galway, Ireland. E-mail: danle.phuoc@deri.org

**Acknowledgements**: This work has been supported by the European FP6 project inContext (IST-034718), by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131), and by the European project DISCOVERY(ECP-2005-CULT-038206).

Some preliminary results have been published in *Proceedings of the ISWC'07 Workshop on New forms of Reasoning for the Semantic Web*, Busan, Korea, November 11, 2007.

# 1 Introduction

Publishing RDF files on the Web allows an agent to share with the world a set of facts (statements) which are believed to be true. DBpedia [ABK+07],[1] for example, publishes a large collection of such statements by extracting them from the collective works of the Wikipedia community. FOAF [BM05] files are personal RDF models which are created by individuals to state facts about, typically, themselves. Nothing prevents, however, such data producers to state facts about any other entity mentioned on the Semantic Web or even to address new conceptual entities.

The sum of such statements, expressed in RDF triples and currently known to be HTTP retrievable, is now in the order of billions. The number of online sources providing RDF is now estimated in the order of tens on millions, rapidly increasing as more large databases are given RDF representations.

Unfortunately however, there is no clear and established model on how to use such amounts of information coming from many diverse sources. In general, such data is unlikely to be directly injected into an end application for multiple reasons. Just to name a few, sometimes the data might be fragmented or incomplete so that multiple sources needs to be joined to have a complete picture. Often the identifiers for the entities (URIs) on the Web or ontology terms are not consistently used across the sources, although they talk about essentially the same things. In other cases the information available from an external online source is believed to be just partially correct, so it should be "patched" before being used a specific end user or agent need. This means that, in general, getting information from the Web into one's own semantic client or system is very likely to require, or at least benefit, from a series of custom steps to be performed involving a number of external or internal sources.

To this end, this paper proposes a Software Pipeline metaphor for the Semantic Web.

## 1.1 Web Pipes and Semantic Web Pipes

Yahoo Web Pipes[2] are a recently introduced tool which already have made a big impact in Web 2.0 development. Using Web Pipes, customized services and information streams can be implemented by processing and combining Web sources using a cascade of simple operators. Such web sources are usually RSS feeds, but also other simple forms of data streams are supported. Creating new information streams and services happens by mashing up openly available information sources: there is no technical need for direct involvement of the original data producer except the agreement on the use of Web standard protocols such as HTTP and RSS.

Since Web pipes are themselves HTTP retrievable data sources, they can be reused and combined to form other pipes. Also, Web pipes are "live". They are computed on demand whenever the HTTP invocation happens, thus reflect the current status of the original data sources.

Likewise, live RDF data source processing would be highly desirable for Semantic Web use cases. We might, for example, want to use DBpedia knowledge about a topic, but yet mix it with the knowledge coming from certain other sites, correcting it by elimination of statements that we believe to be false, and so on. As DBpedia is a live dataset, processing it in a static manner, e.g. by downloading and performing modifications locally as a one-time procedure, is undesirable.

Instead, by describing such operations in terms of Semantic Web pipes, they could be performed transparently by simply fetching the pipe output location, and thus reflecting the most up to date data available online, dynamically from each source.

While Yahoo Web Pipes offer such capabilities, they are unfortunately unsuited to address Semantic Web use cases such as those we previously highlighted: generic RDF is completely unknown as datatype.

## 1.2 Use Cases

Let us outline some illustrative use cases for the emerging Semantic Web applications we envision. This shall lead us to the elicitation of a set of requirements for a Semantic Web pipes execution engine.

[1] `http://dbpedia.org/`
[2] `http://pipes.yahoo.com/`

**Example 1.** *(Aggregating Semantic Web data)*

Pipes should enable us to aggregate RDF data from various structured sources in a flexible manner. For instance, we know that there is data about Tim Berners-Lee on various sources on the Semantic Web (his FOAF file, DBLP, and DBPedia) which might be updated dynamically and adapted on the fly. We realize that we cannot simply merge this data, since all three sources use different identifiers for Tim. Since we prefer using his self-chosen identifier from Tim's FOAF file, we decide to filter the data from DBLP and DBPedia and change the identifiers used there, applying two SPARQL [PS07] queries before integrating these three sources. For DBLP (`http://dblp.l3s.de/d2r/resource/authors/Tim_Berners-Lee`) we thus 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>}
}
```

and from DBPedia (`http://dbpedia.org/resource/Tim_Berners-Lee`) 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>}}
```

◊

We observe that most of this example might be done in a single SPARQL query. However, Semantic Web pipes will allow to combine the building blocks for such aggregations in more flexible manners, for instance allowing to plug RDFS inference in between certain blocks (i.e., applying materialization of inferred triples for Tim's FOAF data, but not on the imported data, etc.).

**Example 2.** *(Filtering RDF Graphs based on personal preferences)*

As a more involved use case, let us try to aggregate information from the FOAF [BM05] files of our infamous friends' Alice, Bob and Charles, shown in Figure 1. Bob is stating that Charles knows Alice in his FOAF file. However, in Charles opinion, Alice has a questionable reputation and Charles, clearly, has no control on Bob's FOAF file. A minimal requirement on distributed metadata is the ability to counter such false statements, thus giving Charles a way to state in his own FOAF file a simple and unambiguous statement: "I don't know Alice", see last statement in Figure 1(c). Semantic Web pipes, as we are going to propose them in this paper, shall cater for such subjective revocations of RDF, and provide means to "merge" RDF sources involving revocations in a way that, depending on who I trust more, I either get the statement that Charles does or does not know Alice. Likewise, my friends should be able to specify their own views of the RDF world in a flexible manner by defining their own pipes, taking subjective revocations from different graphs published on the Web into account, according to their personal preferences. ◊

As we will see in Section 3, revocations as outlined above are hard to model with the ingredients that Semantic Web standards provided so far. For instance, SPARQL queries alone – as the only means to filter and aggregate data – are not really adequate for that task, especially, if we do not know beforehand which subjective statements one graph might revoke. Additionally, such revocations of RDF data might be useful for a second application scenario of Semantic Web pipes:

**Example 3.** *(Collaboratively publishing and patching RDF graphs)*

Alice and Bob work together on a software project and, although they have separate Web spaces to publish their RDF statements, they agree they want to essentially see the same things. Since they cannot

```
@prefix :  <http://ex.org/~alice#>
@prefix foaf:  <http://xmlns.com/foaf/0.1/>
@prefix rdf:  <http://www...rdf-syntax-ns#>
:me rdf:type foaf:Person;
    foaf:name "Alice".
:me foaf:knows <http://examp.org/~bob#me>.
...
```

(b) Alice's FOAF file

```
@prefix :  <http://examp.org/~bob#>
@prefix foaf:  <http://xmlns.com/foaf/0.1/>
:me foaf:name "Bob".
:me foaf:knows <http://ex.org/~alice#me>.
:me foaf:knows <http://ex.org/~charles#me>.
<http://ex.org/~charles#me> foaf:knows
        <http://ex.org/~alice#me>.
...
```

(b) Bob's FOAF file

```
@prefix :  <http://ex.org/~charles#>
@prefix foaf:  <http://xmlns.com/foaf/0.1/>
@prefix rdf:  <http://www...rdf-syntax-ns#>
:me rdf:type foaf:Person;
    foaf:name "Charles".
:me foaf:knows <http://examp.org/~bob#me>.
:me foaf:knows <http://ex.org/~alice#me>.
...
```

(c) Charles' FOAF file

Figure 1: Personal information in FOAF

(over-)write RDF mutually on each others Web spaces, they again use RDF revocations offered by Semantic Web pipes to mutually patch their information. As an example, assume Alice wants to state that Danh joins their joint project, so Alice puts the triple

```
:danh foaf:currentProject :ourProject.
```

Later on, Danh writes a mail to Bob that he is not available for the project, but that his friend Eva could help out. Since Bob cannot write into Alice's graph, he will simply publish information to "patch" Alice's graph by revocations, adding something like:

```
:eva foaf:currentProject :ourProject.
~~:danh foaf:currentProject :ourProject.~~.
```

The week after, Danh meets up Alice and she convinces him to rejoin the project, and also Eva stays in. However, adding `:danh foaf:currentProject :ourProject.` to her graph once again wouldn't change the common view of Alice and Bob, as Bob already revoked that statement. Rather, Alice will deploy another "patch" revoking Bob's revocation. ♢

Semantic Web pipes as described in this paper will also allow to state such mutual revocations and use a pipe giving them a common view on the collaborative "space" described by their both RDF graphs where revocations of several parties can be taken into account in no specific order of preference.

**Example 4.** ***(Smart integration with openly published RDF data)***

As the project goes on, Alice and her team realize that Semantic Web technologies are relevant for the project and wants to find skilled people in the project team on that topic. Alice finds a useful Web application (`openacademia.org`) that provides an RDF dump of publications in that area. So Alice decides to merge the pipe result from Example 3 with the publication data from `openacademia.org` to find out whether names of her project team appear there as authors by the following SPARQL CONSTRUCT query:[3]

```
CONSTRUCT { ?i foaf:maker ?p .}
WHERE{ ?p foaf:currentProject : ourProject; foaf:name ?n.
            ?i burst:publication [ swrc:author [foaf:name ?n] ] }
```

Obviously Alice wants to store this query to reinvoke it dynamically at later points in time, taking into account updates in her project team (by patches from herself or Bob) or new publications appearing on `openacademia.org`. ♢

Fortunately, Alice was pointed by her friend Bob to `http://pipes.deri.org` which will allow them to collaboratively manage, patch, and aggregate their RDF data in flexible enough ways to address all the outlined scenarios.

In the following, we will introduce the basic concept of Semantic Web pipes in Section 2 and introduce a small set of base operators which include simple merges of RDF graphs and executing SPARQL queries.

[3]For the namespace prefixes used here, see the RDF output format of `http://openacademia.org` and RDF vocabularies referenced therein.

Section 3 will introduce the novel concept of RDF revocations to collaboratively patch and filter out RDF and introduce an additional base operator to apply such revocations on RDF graphs. In Section 4, we will show you how the use cases outlined before can actually be modeled by Semantic Web pipes. Finally, Section 5.1 describes our current implementation. We conclude with an outlook to further improvements, related and ongoing work in Section 8.

## 2 Basic Concepts

Use cases such as the ones illustrated so far involve both aggregation of RDF data as well as processing it in meaningful ways, so to obtain RDF data that is a "filtering" and/or "transformation" of the original one. A *Semantic Web pipe* as we define it implements a predefined workflow that, given a set of RDF sources (resolvable URLs), processes them by means of special purpose operators. Unlike fully-fledged workflow models, our current pipes model is a simple construction kit that consists of linked *operators* as shown in Figure 2(a).

Each operator allows a set of unordered inputs in different specific formats (to make them distinguishable) as well as a list of optional, ordered inputs[4] (marked with dashed arrows, these may be of same formats), and exactly one output. Figure 2(b) shows a set of base operators which we implemented so far and which we will explain below.

A *pipe* is a set of instances of such operators, where:

- Both ordered and unordered inputs can be linked to either (i) quoted literals such as `"<?xml version="1.0" ?> ..."`, for providing fixed 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 format, (iii) to the output of another pipe, or (iv) one of the special input variables `$1`,...,`$n`.
- All unlinked inputs must have different formats.
- All but one outputs (the "overall" output of the pipe) are linked inputs of other operators.
- Links between inputs and outputs are acyclic.

Note that, by these constraints, each pipe can itself again be used as an operator.[5] Further, note that we do not constrain links between pipe operators by having the same formats only: as described in Section 5.1, the implementation is built so to be able to accommodate several kind of errors, e.g. sources in timeouts or malformed source data, while still producing a valid output. The default behavior would be 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.

Multiple outputs are not necessary, as they can always be emulated by connecting the same output to multiple inputs of different operators, obtaining the same behavior. Operators in our scenario usually process and produce RDF data, but other data formats are allowed, as for example SPARQL queries, XSLT or generic XML. Considering different formats other than RDF and providing means to transform between those, is important to provide means for integrating with a variety of applications and services (e.g. RSS feeds, keyword based search engines, or directly connecting to Yahoo Web pipes, which produces RSS as well). For simplicity, we limit our model to pipes that do not contain cycles, leaving the study of cyclic pipes to future works.[6] We remark here that, as the current pipe engine is based on a tree-based XML model (see Section 5.1), such cycles are in fact not possible, at least within a single pipe.

Let us now have a closer look at the basic operators that have been implemented so far and that we later demonstrate to be sufficient to address our target use cases. Clearly, these base operators are only a subset

[4] Drawing from the usual UNIX shell script syntax we denote the ordered inputs by the special input variables `$1`,...,`$n`.

[5] In Section 5.1 below, we will outline an XML format that will allow you to publish pipes on arbitrary URLs to re-use them as operators in other pipes.

[6] Such cycles would require conditional operators for termination conditions, etc., and would let us end up with a fully-fledged workflow language, which we do not intend to reinvent here.

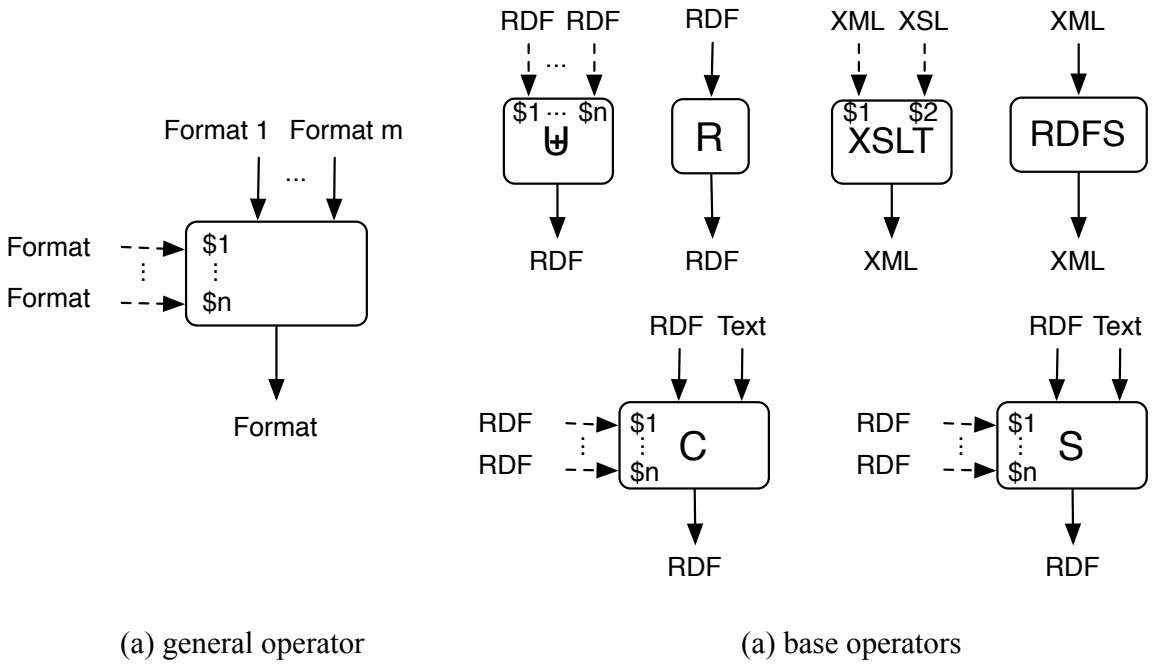



Figure 2: Semantic Web pipe operators.

of the operators that can be thought useful in mashing up information on the Semantic Web. Nonetheless, our notion of operator is general enough for more general extensions in the future.[7]

**The ⊎-Operator: RDF Merge**

This operator takes a list of RDF graphs as inputs, expressed in RDF/XML, N3 [BL98] or Turtle [Bec06] format, and produces an RDF graph that is composed by the merge of its inputs. The standard implementation of the ⊎-operator simply standardizes blank nodes apart, according to RDF merge definition in [Hay04], possibly generating non-lean graphs. In our implementation which is based on MSG decomposition [TMBGE07] we can also support an equivalent (simple entailment) merge operator which, while non lean graphs are still possible, at least avoids duplication of blank nodes for isomorphic subgraphs.

**The R-Operator: RDF Revocation**

Driven by our use cases, we have devised a methodology for expressing revocations of statements within RDF which will be detailed separately in Section 3 below. The Revocation operator takes as input an RDF graph and, if it contains revocations, applies them. The output is a filtered version of the input graph where both revoked triples and the revocation statements themselves are removed.

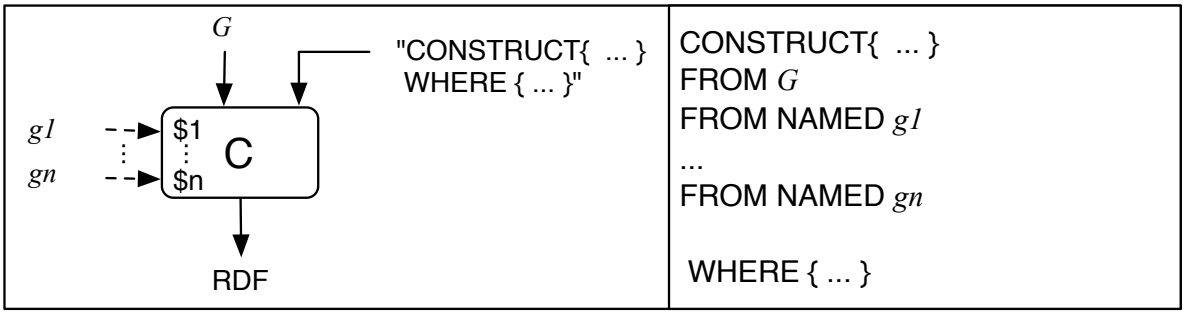
**The C- and S-Operators: CONSTRUCT and SELECT**

The use cases in Examples1+4 have shown how transformations of RDF graphs using SPARQL CONSTRUCT queries can help in aligning RDF data. Such queries can be used to extract only relevant informa-

[7] 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.

tion from a bigger graph as well as to 'align' two graphs, by smushing[8] identifiers or even performing basic ontology mapping operations [PSS07].

The C-operator outputs the result of a construct query given as textual input performed on the standard input RDF graphs as follows. The input query allows full SPARQL CONSTRUCT syntax with the following limitation: FROM and FROM NAMED clauses are not allowed, since the dataset is implicitly given in the inputs. Here, the single input RDF graph from the unordered inputs determines the default graph and the list of ordered input graphs denotes the set of named graphs in the dataset. The graphical C-operator corresponds to a query as follows:




Similarly, we define a SPARQL-Select operator that performs a SELECT query and outputs the result in the SPARQL-Result XML format. The S-operator in conjunction with the XSLT-operator, described in the followings, can easily produce RSS feeds or other XML dialect representation, thus acting as an adapter for non-Semantic Web applications.

**The RDFS-Operator: RDFS Materialization**

This operator basically performs materialization of the RDFS closure of the input graph by applying RDFS inference rules. Here, we mean finite materialization of entailed triples that can be done by simple forward chaining rules [dBH07, IMPT07]. Although this method will not infer 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[9]or Datalog engines [IMPT07]. 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 evaluable inference rules, like $\rho$df on the lower end and "OWL Horst"[tH05, KOM05] on the upper end, could easily be plugged into our framework by similar operators.

**The XSLT-Operator**

Finally, the XSLT-Adapter performs an XML transformation on a generic input XML document. Since both inputs have the same type, we use the ordered input list for this operator, $1 denoting the source XML document and $2 denoting the XSL transform, all other numbered parameters will be ignored. This operator is particularly handy when custom XML output formats are needed or when an input source in a custom XML format shall be transformed to RDF/XML. In a next step, we plan to extend this to an operator that executes nested GRDDL [(ed07] transforms on XML sources. Furthermore, we can easily adapt this operator to operate on HTML files[10] which might be handy to deploy wrappers for Web content.

[8] `http://esw.w3.org/topic/RdfSmushing`

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

[10] by detecting the format and automatically call TidyLib, `http://tidy.sourceforge.net/`, to obtain XSL processable XHTML in a preprocessing step.

# 3 Aggregating, Filtering, and Patching RDF Graphs

Before we turn to address our use cases and show how they can be modeled using pipes, we still miss a methodology to revoke RDF, as was needed for Examples 2 through 4. We observe here that neither RDF nor RDF Schema provide means to make statements such as "Charles doesn't `foaf:know` Alice", or "revoke the statement that `:danh`'s `foaf:currentProject` is `:ourProject`". The semantics of RDF(S) is purely monotonic and described in terms of positive inference rules, so even if Charles added instead a new statement

```
:me  myfoaf:doesntknow  <http://ex.org/~alice#me> .
```

he would not be able to state that statements with the property `myfoaf:doesntknow` should revoke[11] `foaf:knows` statements. Let us see what means the current Semantic Web language spectrum would provides us, in order to state such revocations.

**OWL** The falsehood of Charles knowing Alice can be expressed in OWL, however in a pretty contrived way, as follows (for the sake of brevity we use DL notation here, the reader might translate this to OWL syntax straightforwardly):

$$\{charles\} \in \forall \texttt{foaf:knows}.\neg\{alice\}$$

Reasoning with such statements firstly involves OWL reasoning with nominals, which most DL reasoners are not particularly good at, and secondly does not buy us too much, as the simple merge of this DL statement with the information in Bob's FOAF file would just generate a contradiction, invalidating all, even useful statements. Para-consistent reasoning techniques on top of OWL, such as for instance proposed in [HvHtT05] and related approaches, solve this problem of classical inference, but still require full OWL DL reasoning.

**N3** Tim Berners-Lee's Notation 3 (N3) [BL98] provides to some extent means to express what we are looking for by the ability to declare falsehood over reified statements which would be written as:

```
{ :me foaf:knows <http://ex.org/~alice#me> } a n3:falsehood .
```

Nonetheless, this solution is somewhat unsatisfactory, due to the lack of formal semantics for N3; N3's operational semantics is mainly defined in terms of its implementation `cwm`[12] only. Also, N3 would, from entailing `falsehood` run into similar inconsistency issues as the OWL version above.

**SPARQL** Finally, more along the pipes idea one could, as a naive solution, deploy an off-the-shelf SPARQL engine and filter Bob's FOAF file by a query, leaving just the clean statements. Imagine that Charles stores his unwanted statements in the RDF Web source `<http://ex.org/~charles/badstatements.rdf>`, then such a query filtering the information from merging Bob's and Charles' FOAF files could look as follows:

```
CONSTRUCT { ?S ?P ?O }
FROM <http://ex.org/~charles/foaf.rdf>
FROM <http://ex.org/~bob/foaf.rdf>
FROM NAMED <http://ex.org/~charles/badstatements.rdf>
WHERE { ?S ?P ?O .
        OPTIONAL { GRAPH <http://ex.org/~charles/badstatements.rdf>
                         { ?S1 ?P1 ?O1 . }
                   FILTER (?S1 = ?S && ?P1 = ?P && ?O1 = ?O &&) }
        FILTER ( !Bound(?S1) )  }
```

[11] In fact, we mean here "override" instead of simply contradicting in the pure logical sense.
[12] `http://www.w3.org/2000/10/swap/doc/cwm`

However, simply putting the bad information in a separate file is not a proper solution for the first scenario we outlined, as it is not clear in an open environment like the Web how, for instance, a Crawler stumbling over <http://ex.org/~charles/badstatements.rdf> should disambiguate this data from valid RDF information. Rather, we would need to reify the negative statements using for instance the N3 version outlined before, or the "native" RDF reification vocabulary[13] which would – besides blowing up metadata by unhandy reified statements – further complicate SPARQL querying of that data[14] to filter out the "good" data.

In the following, we will sketch a more practical solution to the problem, exploiting previous work on Minimum Self Contained Graphs (MSGs).

## 3.1 Revoking RDF Statements by MSG Hashes

Any RDF graph may be viewed as set of triples. Triple level processing of distributed RDF files, particularly identifying the same RDF graphs, is made very complex by the existence of blank nodes. For this reason, the RDFSync algorithm, which we presented in previous work, introduced the notion of Minimum Self Contained Graph (MSGs) [TMBGE07]. Simply said, an MSG is constructed starting from a triple and collecting, for each blank node in it, all the other triples attached to these until no more blank nodes are involved. Such "closure" makes sure that a graph can be recomposed at a different location simply by merging all the MSGs composing it, even if these are transferred one at a time. As MSGs are stand-alone RDF graphs, they can be processed with algorithms such as canonical serialization. We use an implementation of the algorithm described in [Car03], which is part of the RDFContextTools Java library[15] to obtain a canonical string representing the MSG and then we hash it to an appropriate number of bits to reasonably avoid collisions. This hash acts as a identifier for the MSG with the fundamental property of being content based, which implies that two remote peers would derive the same hash-ID for the same MSGs in their databases. A graph can be therefore treated as a set of digital hashes representing its constituent MSGs. In the context of the problem addressed in the present work, we use such digital hashes to refer to the MSG itself. I.e., the finest granularity at which we allow to revoke RDF statements is at the level of MSGs. The hash function we use for MSGs takes the form of a literal encoding the 16 bytes of the MD5 hash of the canonical graph serialization mentioned above. Stating that an MSG is false/revoked is therefore as easy as stating a single triple where the subject is a member of the class `pipes:MSGRevocation`,[16] the predicate is the designated predicate `pipes:md5hash` and the 16 bytes literal containing the MSG hash as an object, so the negative statement could be made directly within Charles' FOAF file as follows:

```
_:x a pipes:MSGRevocation .
_:x pipes:statedBy :me .
_:x pipes:date "2007-11-01T23:59:59-10:00"^^xsd:dateTime .
_:x pipes:revocationDescription
    "Who the FOAF said that I know Alice?
     I don't know her."^^xsd:string .
_:x pipes:involvedResource :me .
_:x pipes:revokesMSGHash
    "MD5HASH_OF_:ME_FOAF:KNOWS_ALICE"^^xsd:string .
```

Note that only the last triple stating the revoked MSG's hash is essential here, the other properties in the `pipes:` vocabulary are only for additional metadata (such as date, author, natural language decription, etc.) about the stated revocation and do not play a role for the semantics of such revocations.
Storing MSG hashes instead of reifying statements has (except being a concise representation saving considerable storage space) some other interesting implications: This solution works conceptually well as it takes

[13] Using `rdf:Statement`, `rdf:subject`, `rdf:predicate`, `rdf:object`

[14] Note that, in the FILTER, we exploit the admittedly awkward way to model set difference in SPARQL [Pol07] which as such might already not be considered intuitive unanimously.

[15] `http://www.dbin.org/RDFContextTools.php`

[16] We use the namespace `<http://pipes.deri.org/2007/10/ns#>` for the `pipes:` prefix. A simple RDFS vocabulary for stating RDF revocations and annotating pipes is Web-accessible there, where all supported properties are defined.

care even of cases where one wants to deny statements which involve blank nodes. This would not be possible using reification due to the arising ambiguity. Digital hashes over MSGs, which are agnostic about blank node IDs, avoid this problem.

A drawback of the solution to quasi "encode" the negative statements in MSG hashes – which in fact possibly turns out to be a feature in certain use cases[17] – is that the negated statements are not clearly "readable", e.g. by direct inspection of the RDF file. If, on the contrary, the denied statements should be made legible, one can explicitly state revoked statements using the `pipes:revokesStatement` property and using the standard RDF reification vocabulary. However, this has, as the original RDF Reification vocabulary, no extra semantics attached in our approach and serves as additional metadata only. For a more in-depth discussion of encodings of revocations by hashes and possible alternatives, see [MPT07].

**Semantics and Implementation of the R-Operator**

Now that we have explained RDF revocations in more detail, we are ready to define the semantics and implementation of the R-operator that was only informally outlined before. The semantics of revocations, as long as they only affect MSGs that do not contain revocation vocabulary themselves, is quite straightforward, i.e., given by the following algorithm:

```
RDFGraph revokeNaive( RDFGraph G )
 {
  out := empty graph;
  for ( m in [all MSGs in G] )
   {
     if ( md5hash(m) not in [values of
              pipes:revokesMSGHash property in G]
          && m is not a revocation itself )
        out := out + m;
   }
  return out;
 }
```

This algorithm intuitively filters out all MSGs which are object to revocations and drops all revocations, returning a "pure" (i.e., revocation-free) RDF graph.

However, let us reconsider Example 3 at this point.

**Example 5.** ***(Example 3 cont'd)***

Bob and Alice agree to use RDF revocations for deploying mutual patches on the merge of their respective RDF graphs, i.e. each of them incrementally writes into her/his own graph in order to revoke or add statements: In order to revoke the triple

```
:danh foaf:currentProject :ourProject.
```

with the MD5 hash *hash*1 from Alice's graph, Bob adds the triple

```
_:r1 pipes:revokesMSGHash "hash1"^^xsd:string.
```

with the MD5 hash *hash*2 to his graph. Subsequently, as soon as Alice wants to revoke that revocation in turn again, she simply adds

```
_:r2 pipes:revokesMSGHash "hash2"^^xsd:string.
```

[17] One wouldn't always want to make public which information she/he doubts.

to her own graph. ◊

As easily can be seen, the revocation operator as defined in the algorithm above would not necessarily do what it is supposed to do; namely, it would apply *all* revocations, ignoring such "chains" of revocations. Before we fix this, let us have a closer look on what such chains of revocations actually mean. We observe that, theoretically, even cyclic revocations are possible (although by md5 sums hard/almost impossible to compute):

$r1:$ `_:r1 pipes:revokesMSGHash "`*hash of r2*`"^^xsd:string.`
$r2:$ `_:r2 pipes:revokesMSGHash "`*hash of r1*`"^^xsd:string.`

In our current implementation, we consider such cycles in revocations are rather a sign of a hash collision and issue a warning to the user. Nonetheless, if we disregard collisions, cyclic revocations do not add anything to the picture, as shown by the following Proposition:

**Proposition 1.** *If there are no hash collisions, any revocation involved in a cycle of revocations only revokes other revocations.*

Based on this observation, we handle chains of revocations by adapting the algorithm above as follows: Initially, by procedure $createRDG(G)$ we create a directed graph, the so-called revocation dependency-graph (RDG), having as nodes the MSGs of the graph at hand, and we create an edge from $MSG_1$ to $MSG_2$ if $MSG_1$ contains a revocation statement, revoking $MSG_2$. Further we assume a procedure $applyPrimeRevocations()$ that returns the set of all MSGs without an incoming edge from $RDG$ and removes these and all adjacent MSGs from both $RDG$ and $G$. This procedure is simply applied until no more MSGs without an incoming edge are left:

```
RDFGraph revoke( RDFGraph G )
 {
  RDG := createRDG(G);
  while( applyPrimeRevocations() != empty set )
    {}
  r := all revocations left in G
  if ( r != empty set )
    { /* issue a warning, as cyclic revocations
         are suspicious for collisions  */ }
  return G - r;
 }
```

The last step only removes remaining cycles of revocation statements – those can be safelyignored following Proposition 1.[18]

Decomposing a graph into MSGs and calculating MSG hashes might sound computationally expensive if the graph is big, contains a large number of bnodes, and/or highly connected bnodes. The operations can however be performed efficiently if a proper index is put in place [TMBGE07], or by the use of an auxiliary property (`pipes:involvedResource`) that points to the URIs of the MSGs which are involved in the revocation (thus, we do bot need to decompose the entire graph to locate the wanted MSGs).

The current implementation exploits the `pipes:involvedResource` property, if present, as follows. since this property allows one to state involved RDF resources in the revoked graph, it on the one hand provides means to express what a revocation is about, but on the other hand also allows us to speed up the algorithm defining revocation semantics: When applying a revocation, we only need to calculate the hashes of those MSGs which – as a sufficient condition – contain at least one statement involving the values of `pipes:involvedResource` property for all revoked MSGs (in this case `<http://ex.org/\~charles#me>`), thus avoiding a complete graph decomposition. Similarly the `pipes:involvedResource` property can be used

[18] Still, a warning is issued, since we normally would not expect cyclic revocations to be deliberately made, but rather see them as a possible indication for a hash collision.

as weak form of hash collision dedection by checking – as a necessary condition – whether the involved resource appears at all in the revoked graph before applying a revocation. A way to speed up computation of revocations further might be to do a complete MSG decomposition once, when the graph is originally loaded, and to keep an index of MSG hashes to original triples locally cached. Such initial computational effort would however result in faster operations for repeated pipe calculation. Furthermore we notice that MSG decomposition might be needed anyway for other purposes, for example to perform remote RDF synchronization [TMBGE07].

# 4 Use Cases at Work

Getting back to our original goals, let us have a glimpse at the pipes addressing the use cases from Section 1.2.

Example 1 is easily modeled by the pipe shown in Figure 3. URIs are normalized via C-operators before joined with Tim's FOAF file. One might argue that this simple use case might easily be modeled in a single SPARQL query. However, unlikedoing the aggregation all-in-one in a single SPARQL query, the modular fashion pipes are built allows arbitrary more complex combinations. For instance, we additionally merge the FOAF ontology with Tim's FOAF file and do RDFS inference before merging the data with the other blocks.

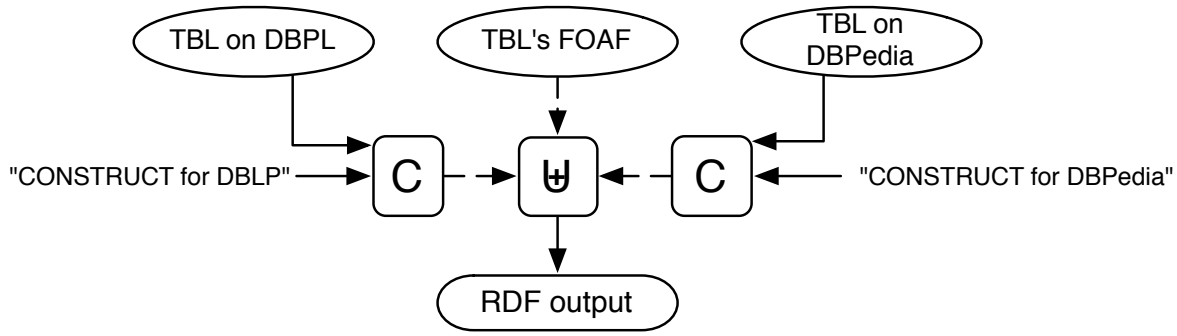
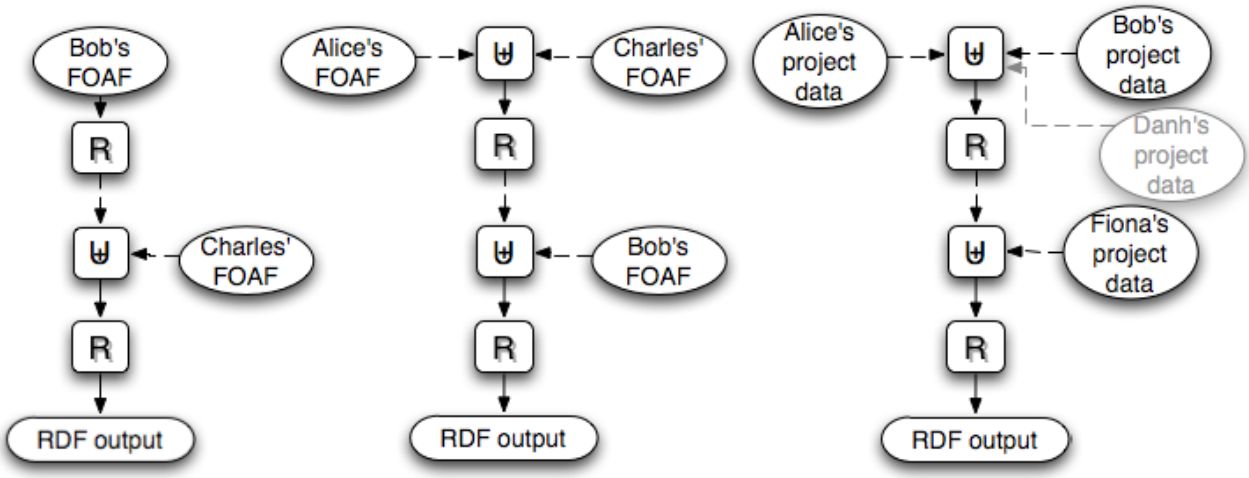



Figure 3: A pipe which combines Tim Berners-Lee's Foaf file, with his DBLP and DBPedia information.

As for modeling preferential "views" on RDF graphs , such as described inExample 2, let us take Charles's view of things. Since he considers Alice untrustworthy, he just wants to take Bob's and his own data into account, where revocations should be applied in the sense that his own graph is the most important one. Bob on the other hand, considers Alice's and Charles information equally important, but nonetheless prefers his own statements. Such preferences can be realized by chains of the $R$ and $\uplus$-operators, as shown in Figures 4(a)+(b), where in Charles' pipe the fact that he knows Alice is indeed revoked, unlike what happens in Bob's pipe.

Collaborative pipes like in the scenario of Example 3, can be modeled even easier, see Figure 5. The upper pipe shows Alice's, Bob's and Danh's collaborative view which is simply realized by applying an R-operator on the merge of their separate files. As long as all of them write in their data and patch in their own graphs, the joint pipe shows for all of them the same collaborative view. Alice reuses this pipe to realize the query outlined in Example 4. Preferential and collaborative pipes can also be mixed. Figure 4(c) shows Fiona's pipe: Fiona is Alice and Bob's supervisor and wants to have a view on their project. She is not taking part directly in the collaborative patching but by this pipe able to override their statements. As we see, the pipe for this use case looks almost the same as Bob's view on his friends, so the same pipe "shape" might serve different purposes.




(a) Charles' view on his friends (b) Bob's view on his friends (c) Fiona's view on Bob and Alice's project

Figure 4: Preferential views written as pipes.

**Updating Collaborations**

When Danh joined the project, Alice and Bob decided to let him participate in their collaborative "RDF space", which resulted in the upper pipe in Figure 5 which all of them use. However, Fiona did not notice that change and might get an incomplete view of the project state with her pipe. Someone stumbling over Alice's project data on the Web would have the same problem, not knowing where to get the complete picture. To avoid such situations, we provide a dedicated property in the Pipes RDFS vocabulary (see Footnote 16 above): `pipes:seeAlso`. Alice (and similarly Bob and Danh) uses this property to put a pointer to the pipe describing the "bigger picture" her partial RDF graph belongs to by adding the triple:

```
<http://ex.org/~alice/ourProject.rdf> pipes:seeAlso
<http://pipes.deri.org/pipes/?id=ourProject>
```

This property shall be used to indicate that an RDF graph is not to be viewed "alone" but as a part of a pipe. At the moment, we do not assign a fixed semantics to this property, but our pipe execution engine can issue a warning if it tries to use RDF graphs containing `pipes:seeAlso` triples out of one of their context pipe.

# 5 Implementations

We provide implementations covering two main aspects of using Semantic Web pipes: an online execution engine and AJAX editor, and the DBin 2.0 client to publish and cooperatively edit RDF data.

## 5.1 The Execution Engine

An implementation for Semantic Web pipe has been created and is available online at `http://pipes.deri.org`. The implementation is composed of an execution engine and by an AJAX based pipe editor.

Using pur pipe engine, online users can create pipes to fetch, mix, and process RDF files published on the Web. As the output of a pipe is an HTTP-retrievable RDF model or XML file, simple pipes can also

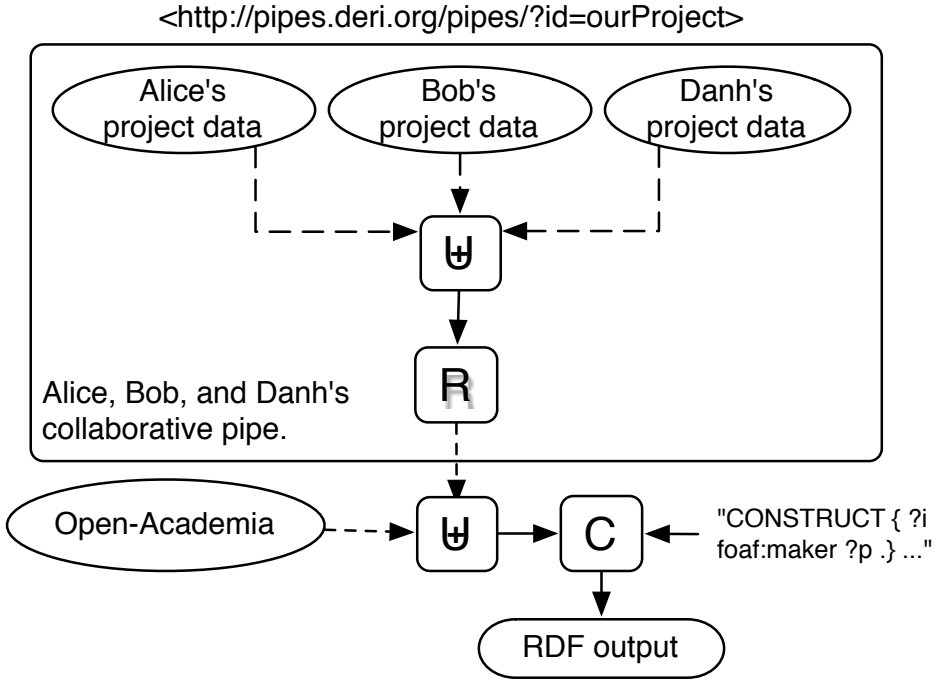



Figure 5: A pipe for collaborative editing/patching. Alice re-uses this pipe for her query.

work as inputs to more complex pipes. Pipes are written in a simple XML language.[19]For example mixing two RDF sources (the M-operaor) is written as follows:

```
<merge>
  <source url="http://www.w3.org/People/Berners-Lee/card"/>
  <source url="http://g1o.net/foaf.rdf"/>
</merge>
```

While it would be possible to implement pipe descriptions themselves in RDF, we decided against this option, and for an own XML format in favor of simplicity of both the implementation and editing: an ad hoc XML language can be much more terse and legible than an equivalent RDF representation. If an RDF representation of the structure of Semantic Web pipes will be requested at a later stage of development, it will be anyway possible to apply a GRDDL transformation of the current XML based pipe format.

The following XML implements the pipe from Example 1 shown in Figure 3.

```
<merge>
  <source url="http://www.w3.org/People/Berners-Lee/card#i"/>
  <construct>
    <source
     url="http://dblp.l3s.de/d2r/resource/authors/Tim_Berners-Lee"/>
    <query> <![CDATA[
    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}
```

[19]A grahical editor following the notation in Section 2 is currently under development

```
   UNION
 {?s2 ?p2 <http://dblp.l3s.de/d2r/resource/authors/Tim_Berners-Lee>}}
 ]]></query>
  </construct>
  <construct>
    <source url="http://dbpedia.org/resource/Tim_Berners-Lee"/>
  <query> <![CDATA[
   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>}}
 ]]></query>
  </construct>
</merge>
```

Our engine calculates the output of this pipe "live" when the pipe URL is fetched. The pipe URL for the dynamically generated output is composed of a common prefix (e.g., `http://pipes.deri.org/pipes/`) and a name chosen by the user who created the pipe.[20] HTTP-compliant caching is performed in the pipe execution engine to ensure that sources are not fetched more often than necessary. Since pipe components are functional elements, there is no reason to recompute a pipe output if the sources have not changed. For this reasons, whenever content is fetched it is always hashed to detect if it has actually changed. If this is not the case then the pipe, or parts of the pipe, will not be recalculated.

Circular invocations of instances of the same pipe could create denial of service situations. While one could detect circular pipe invocation within the same pipe engine (i.e. a pipe accessing another pipe produced by the same engine), this is less simple in case multiple pipe engines are involved. In this case, our solution relies on extra HTTP headers. The mechanism is simple but effective in most cases limiting the number of invocations of the same pipe: whenever the pipe engine fetches a model from another pipe engine, it will perform the HTTP GET putting an extra *PipeTTL* (Time To Live) header. This works similar to the IP mechanism to prevent routing loops on the internet: at each subsequent invocation of a pipe, the TTL number will be decremented and that number will be used when the engine performs further fetching from the internet to execute the pipe itself. A pipe engine will refuse to fetch more sources if its execution is invoked with a PipeTTL header of $\leq 1$.

Finally, the AJAX pipe editor that is currently provided, albeit simple, helps the user by providing inline operator documentation when inserting a component. Also, it presents a list of available pipes so to foster pipe reuse and composition. A simple debugging facility is also provided which highlights any execution error while executing the pipe. This faciity complements the normal runtime behavior which is very accommodating to network errors (e.g., using copies of previously available RDF files on network timeouts or simply proceed treating malformed input sources as empty where possible). To prevent unwanted deletion or editing, pipes can be password protected. Both execution engine and AJAX editor are available as open source software.

## 5.2 DBin 2.0: Cooperatively Editing RDF

Thanks to the R-operator, agents can achieve cooperative editing of Semantic data while each still publishing RDF independently on the Semantic Web, without requirements for shared infrastructures (e.g. a CVS server). To do this, however, agents must be able to create RDF revocations according to the semantics of the R-operator. This process is in general too complex to be realistically performed by hand editing one's own RDF file. This is due to the fact that statements that need to be inserted when the user wants to perform an action (e.g. adding a triple) vary according to the statements that are currently in the union

[20] For instance, the output of this simple example pipe is available at `http://pipes.deri.org/pipes/?id=simplemix`.

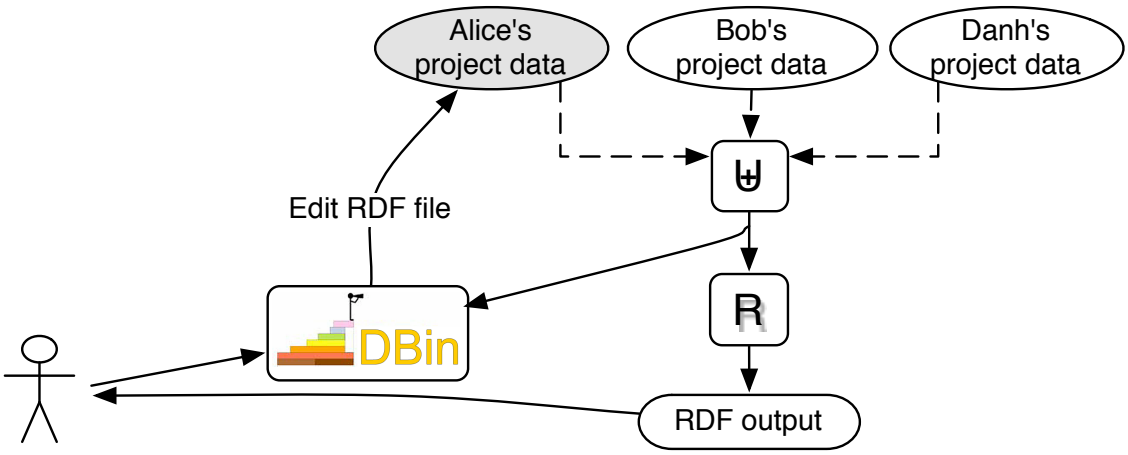



Figure 6: DBin can be used to edit a specific graph composing a pipe.

of the RDF models of all those participating in a collaborative space. For example, adding a triple to the common view might also require to add one or more revocations.

We have therefore implemented the full logic to operate on this scenario in the 2.0 version of the DBin Semantic Web client and authoring tool. While DBin 0.x [NMT06] was based on an ad hoc P2P infrastructure where information "flows" across peers, in DBin 2.0 information is instead shared and merged using locally driven Semantic Web pipes. DBin 2.0 therefore uses internally the same engine as highlighted in the previous section but on top of this it adds two fundamentals features that we describe in the following.

### Automatic Publishing of RDF Graphs on the Web

DBin 2.0 works in pair with a Web script (a rather minimalistic PHP file, very simple to host) to be able to publish RDF files on the Web. During the DBin 2.0 setup the user enters the URL by which the script can be executed (e.g., `http://ex.org/~alice/upload.php`) and a name for the online file where the the locally created RDF data will be stored (e.g. `ourProject.rdf`).

From now on we call this graph the '*base graph*' of the user. A base graph is univocally identified by its publishing URL, automatically assigned by the publishing script given the chosen base graph name (e.g. `http://ex.org/~alice/ourProject.rdf`).

### Collaboratively Editing RDF using the R-operator

Once DBin is given the list of the URLs of base graphs of the other users or agents involved in the cooperative editing (in our example case, Bob's and Danh's graphs), DBin 2.0 generates and applies the proper revocation chains in response to the user actions in the GUI.

The end result is cooperative editing of semantically structured knowlede, yet based on independent publishing of personal views (e.g. the individual FOAF or partial project files of the single participats).

This scenario is depicted in Figure 6. DBin uses the output from the Merge operator (before the R-operator) to calculate, if necessary, the revocations which are then added, by means of the online publishing script, to Alice's base graph. At the same time DBin calculates the revocations and presents Alice with the end result of the pipe.

With DBin 2.0 it is also possible to specify more complex pipes. For example, it is possible to shape a scenario where the cooperatively created knowledge can be overridden by yet another source, e.g., acting as a supervisor. This allows running the scenario illustrated in Figure 4(c) in which Fiona supervises the knowledge edited by Alice and Bob, while they retain the ability to state their own views of the world as

independently published in their base graphs, which can be directly accessed or possibly included in other (in this case, Fiona's) pipes.

Our experiences with using the beta version of DBin 2.0 show that the pipe layout structure needed for a basic cooperative scenario is in general simple (e.g., a single mix, with a revocation operator maybe involving RDFS inference) but many sources need to be imported (e.g., each of the RDFS or OWL ontologies needed to process data) and that the transform operators has to be used multiple times to cope at least with URI renaming issues.

DBin 2.0 developed as an open source project and is currently available in a beta version.

# 6 Related Works

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.[21]

Semantic Web pipes as described in this paper implement a similar concept, but allow, unlike UNIX pipes, to connect outputs to multiple inputs of other operators so that there can be multiple branches executed at the same time.

Similarly, cascaded XML transformations are sometimes referred to as XML pipelines and have been successfully employed in projects like Apache Cocoon.[22] Several languages to express XML pipelines have been proposed and used by software vendors,[23] and the World Wide Web Consortium is now working on a standardization of such a language [WMT07]. Although RDF can be written in XML, XML pipelines cannot be directly applied to RDF given the well known issues related to the differences between RDF and XML datamodels. However, XML processors, as the one we describe in the XSLT-operator, are useful even in a RDF centered scenario, e.g., as adapters for non-RDF applications.

The Yahoo Web Pipes framework, as mentioned earlier, was greatly inspiring our work, but lacks in functionality to address our desired use cases. Notably however, Yahoo pipes provides an easy to use and powerful Web based graphic composer for pipes.

The same idea and a similar UI is used by the Mac OS X built-in Automator,[24] which allows users to compose workflows of simple operations (send mail, open file, etc.).

Concerning the Semantic Web world, the need for a cascade of operators to process RDF repositories is also addressed in the SIMILE Banach project. Banach operates inside the Sesame DB by leveraging its capabilities to have a pipelined stack of operators (SAILS) which can both process data and rewrite queries. Only few basic operators has been developed so far, but plenty of possibly useful operators are discussed in the Web site.[25] While some of these are already covered by the C-operator, others are not, but might be considered in future releases of our engine.

The necessity to deal with the lack of absolute truth on the Web has been pointed out long ago already.[26] Several considerations about adding the possibility to express negative statements in RDF have been made in the literature [AADW05]. However, our novel concept of RDF revocations is tackling this issue from a different angle. Revocations neither correspond to classical negation, nor weak or strong negation as described in [AADW05]. Rather, our revocation vocabulary and the R-operator in pipes provide a purely operational, low-level tool to implement "user-defined, para-consistent reasoning" for RDF. A model-theoretic definition of our R-operator is still on our agenda, but ERDF programs from [AADW05] or for instance courteous logic programs [Gro97] might serve as starting points for a more logical formalization.

[21] http://www.linfo.org/pipe.html
[22] http://cocoon.apache.org/
[23] http://www.orbeon.com/ops/doc/reference-xpl-pipelines, http://www.oracle.com/technology/tech/xml/xdkhome.html
[24] http://automator.us/leopard/index.html
[25] http://simile.mit.edu/wiki/Banach
[26] http://www.w3.org/DesignIssues/

# 7 Future Works

First of all, SPARQL endpoints can be generated easily for each pipe. Following approaches such as Sesame's SAIL we will investigate optimizations where pipe components do not necessarily need to process all data to execute SPARQL queries. Instead, they could in certain cases rewrite the query or push parts of it forward to the original sources, in case they offer SPARQL endpoints, e.g., via mechanisms such as the Semantic Sitemap extentions[27] or well-known query plan optimization techniques from databases [Hal01].

As we mentioned, a number of additional operators can then be imagined to aid ontology and data alignment when SPARQL CONSTRUCT queries are inconvenient or do not have the required features. In previous works, 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 [PSS07] such as FOAF and vCard. We aim to replace the base C- and S-operators with more expressive extensions such as SPARQL++ [PSS07] which allows for the use of aggregates or string manipulations (to create new terms in CONSTRUCTs that do not appear in the source graphs).

Operators that batch-process sets of sources shall be introduced (commonly known as "FOR" loops). This can allow to address use cases such as merging of an a priori unknown number of sources (e.g. the results of a query to a Semantic Web search engine to find RDF sources).

Similarly, by our generic operator model, it is conceptually already foreseen to extend the current engine for "parametric pipes" which would inject in the XML pipe definition extra parameters as requested in the HTTP GET query string (using the $1,..., $n parameters explicitly in pipes defnitions, which our current XML syntax does not yet allow). This could be very useful to allow a pipe to have parametric behaviors and would allow pipes to act within other pipes not only as sources but as new, full featured, operators.

Also, it will be interesting to consider how to achieve interaction between advanced RSS feed processing tools like Yahoo Pipes and Semantic Web operators. The SPARQL SELECT operator, producing XML, together with XSLT transforms provides a base for this but more operators will have to be studied.

Many technical solutions can also be put in place to achieve scalability. These range from smart pipe execution strategies, advancing those explained in the previous sections, to others such as, for example, differential updates of the local copy of large remote RDF graphs [TMBGE07].

Finally, while Semantic Web pipes (like Web pipes and Unix pipes) are certainly a tool for expert users, it is undeniable that the overall engine will be much more useful once a visual pipe editor is availale. A grahical editor for our XML format following the notation in Section 2 is currently under development.

# 8 Conclusions

Publishing data online in RDF is an important enabler for data interoperability and reuse. Many problems, however, still become apparent when trying to aggregate and make intelligent use of such data. In this paper we introduced the Semantic Web pipes paradigm and showed how it can be readily applied to address such issues.

Semantic Web pipes were also shown to be a paradigm that can do more than data harmonization alone: they implement basic workflows which can be used to model data flow scenarios that also include collaborative aspects. Most importantly, Semantic Web pipes are based on the union of functional operators specific to Semantic Web with the HTTP REST paradigm. Such combination fosters clean implementations, and promotes reuse of data sources as well as pipes themselves.

In this context, we introduced "revocations" in RDF. Revocations are a practical solution to add negative statements to RDF without running into logical inconsistency. We illustrated how to implement a pipe operator (the R-operator) that can interpret such revocations and apply them. As a result, pipes which make use of the R-operator can be used to achieve web based collaborative editing of Semantic Web knowledge while still allowing individuals to state their perceived view of the world.

[27] http://sw.deri.org/2007/07/sitemapextension/

Finally, we described the implementations of the general pipe engine, of the R-operator and of the DBin 2.0 client. DBin 2.0 can be used to create RDF sources which are aware of the R-operator so to allow users to cooperatively edit a common RDF knowledge base.

## Acknowledgments

This work has been supported by the European FP6 project inContext (IST-034718), by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131), and by the European project DISCOVERY(ECP-2005-CULT-038206). We thank Thomas Krennwallner for useful comments on this report.

## References

[AADW05] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damásio, and Gerd Wagner. Stable model theory for extended RDF ontologies. In *4th Int.l Semantic eb Conf. (ISWC2005)*, 2005.

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *6th Int.l Semantic Web Conf.*, Busan, Korea, November 2007.

[Bec06] Dave Beckett. Turtle - Terse RDF Triple Language, April 2006. `http://www.dajobe.org/2004/01/turtle/`.

[BL98] Tim Berners-Lee. Notation 3, since 1998. `http://www.w3.org/DesignIssues/Notation3.html`.

[BM05] Dan Brickley and Libby Miller. FOAF Vocabulary Spec., July 2005. `http://xmlns.com/foaf/0.1/`.

[Car03] Jeremy J. Carroll. Signing rdf graphs. In *The Semantic Web - ISWC 2003, Second International Semantic Web Conference*, pages 369–384, Sanibel Island, FL, USA, October 2003.

[dBH07] Jos de Bruijn and Stijn Heymans. RDF and logic: Reasoning and extension. In *6th Int.l Workshop on Web Semantics (WebS 2007)*, Regensburg, Germany, 2007.

[(ed07] Dan Connolly (ed.). Gleaning Resource Descriptions from Dialects of Languages (GRDDL), July 2007. W3C Proposed Rec., `http://www.w3.org/TR/grddl/`.

[Gro97] Benjamin Grosof. Courteous logic programs: Prioritized conflict handling for rules. Technical Report RC20836, IBM Research, May 1997.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, December 2001.

[Hay04] Patrick Hayes. RDF semantics, February 2004. W3C Rec.

[HvHtT05] Zhisheng Huang, Frank van Harmelen, and Annette ten Teije. Reasoning with inconsistent ontologies. In *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, Edinburgh, Scotland, August 2005.

[IMPT07] Giovambattista Ianni, Alessandra Martello, Claudio Panetta, and Giorgio Terracina. Faithful and effective querying of RDF ontologies using DLV$^{\mathrm{DB}}$. In *4th Int.l Workshop on Answer Set Programming (ASP'07)*, Porto, Portugal, 2007.

[KOM05] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - a pragmatic semantic repository for OWL. In *Int.l Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2005)*, New York City, USA, 2005.

[MPT07] Christian Morbidoni, Axel Polleres, and Giovanni Tummarello. Who the FOAF knows Alice? A needed step towards Semantic Web Pipes. In *ISWC 2007 Workshop on New forms of Reasoning for the Semantic Web: Scaleable, Tolerant and Dynamic*, Busan, Korea, November 2007.

[NMT06] Michele Nucci, Christian Morbidoni, and Giovanni Tummarello. Enabling semantic web communities with dbin: an overview. In *ISWC2006 Semantic Web challenge*, Athens, GA, USA, 2006. Finalist.

[Pol07] Axel Polleres. From SPARQL to rules (and back). In *16th World Wide Web Conference (WWW2007)*, Banff, Canada, May 2007.

[PS07] Eric Prud'hommeaux and Andy Seaborne (eds.). SPARQL Query Language for RDF, June 2007. W3C Candidate Rec., `http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/`.

[PSS07] Axel Polleres, François Scharffe, and Roman Schindlauer. SPARQL++ for mapping between RDF vocabularies. In *6th Int.l Conf. on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, Vilamoura, Algarve, Portugal, November 2007.

[tH05] Herman 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), July 2005.

[TMBGE07] Giovanni Tummarello, Christian Morbidoni, Reto Bachmann-Gmur, and Orry Erling. RDFSync: efficient remote synchronization of RDF models. In *6th International Semantic Web Conference (ISWC 2007)*, Busan, Korea, 2007.

[WMT07] Norman Walsh, Alex Milowski, and Henry S. Thompson. Xproc: An xml pipeline language. Technical report, W3c, October 2007. Working Draft.