Dealing with inconsistencies in Linked Data Mashups

Eveline R. Sacramento, Marco A. Casanova, Karin K. Breitman, Antonio L. Furtado
Department of Informatics – PUC-Rio
Rio de Janeiro, RJ – Brazil
{esacramento, casanova, karin, furtado}@inf.puc-rio.br

José Antonio F. de Macêdo, Vânia M.P. Vidal
Department of Computing - UFC
Fortaleza, CE – Brazil
{jose.macedo, vvidal}@lia.ufc.br

ABSTRACT
Data mashups constructed from independent sources may contain inconsistencies, puzzling the user that observes the data. This paper formalizes the notion of consistent data mashups and introduces a heuristic procedure to compute such mashups.

Categories and Subject Descriptors
H. Information Systems **[H.m. Miscellaneous]**: Databases

General Terms
Design,Verification.

Keywords
Data mashup, constraint verification, Linked Data, inconsistency

1. INTRODUCTION
The term Linked Data refers to a set of best practices for publishing and connecting structured data on the Web [3]. From the user's perspective, the main goal of Linked Data is the provision of integrated access to data from a wide range of distributed and heterogeneous data sources [4]. However, applications accessing a Linked Data corpus from different sources may face challenges [8] since the combined data may be inconsistent, inaccurate, incomplete, or stale [7]. In this paper, we investigate the problem of constructing consistent data mashups in the context of Linked Data.

In more detail, consider a *Linked Data mashup service* that covers a given domain, defined by a *domain ontology* and a set of the Linked Data sources, modeled by *application ontologies*. We consider only one domain ontology for simplicity. We assume that: (1) the application ontology vocabularies are subsets of that of the domain ontology; (2) the Linked Data mashup service has access to the vocabularies of the application ontologies (but not to their constraints); (3) the Linked Data mashup service has access to the vocabulary and constraints of the domain ontology. These assumptions are consistent with the current Linked Data practice, which promotes: (1) reuse of known vocabularies to define a Linked Data source; (2) adoption of a VoiD document to indicate the vocabularies – but not the constraints – that a Linked Data source uses; (3) adoption of repositories that provide access to the full definition – vocabulary and constraints – of commonly used domain ontologies.

We cannot assume, however, that the data retrieved from different Linked Data sources is consistent with the constraints of the domain ontology, for two reasons. First, we have no guarantee that each Linked Data source returns consistent data; in fact, we do not even know what constraints the Linked Data source respects. Second, even if each Linked Data source returned data which is consistent with the domain ontology constraints, the combined data might be inconsistent. In view of these observations, the Linked Data mashup service must always analyze the data coming from the Linked Data sources to identify and isolate inconsistent data.

The contribution of the paper is a heuristic procedure to compute consistent data mashups, when the data sources return positive assertions, including equalities. The formalization is coherent with the current Linked Data best practices and is based on DL-Lite core with arbitrary number restrictions [1] and on the notion of open ontology fragments [6]. The heuristic procedure explores the constraint graph [5] of the data mashup specification to construct a consistent data mashup.

The paper is organized as follows. Section 2 further discusses the question of consistency in data mashups. Section 3 briefly reviews DL-Lite core, constraint graphs and open ontology fragments. Section 4 formalizes the notion of data mashup and discusses how to compute consistent data mashups. Finally, Section 5 contains the main conclusions.

2. CONSISTENCY IN DATA MASHUPS
This section illustrates our problem with the help of an example. We adopt the *Music Ontology* [9], which provides concepts and properties for describing artists, albums, tracks, performances, arrangements, etc. It is used by several Linked Data sources, including MusicBrainz and BBC Music. The *Music Ontology* uses terms from the *Friend of a Friend* and the XML Schema vocabularies. We use "mo:", "foaf:" and "xsd:" to refer to these vocabularies. Figure 1 shows the class hierarchies of the *Music Ontology* rooted at classes foaf:Agent and foaf:Person. We take the domain ontology to be this part of the *Music Ontology*, which we call the *Agent-Person Ontology* (***APO***). We also consider that ***APO*** has a constraint which says that *each person has at most one name*.

Let μ_1 be a data source about music artists and groups and μ_2 be a data source about music contracts, whose designs are both based on the ***APO*** domain ontology. Consider now a data mashup

IDEAS12 2012, August 8-10, Prague [Czech Republic]
Editors: Bipin C. Desai, Jaroslav Pokorny, Jorge Bernardino Copyright ©2012 ACM 978-1-4503-1234-9/12/08 $15.00.

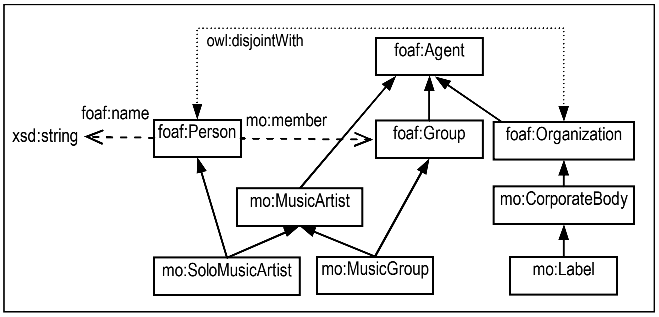

Figure 1. The class hierarchies of *APO*.

for music data, modeled according to ***APO***, and using data from μ_1 and μ_2. First, the user specifies the mashup he wants by selecting classes and properties from the vocabulary of ***APO***. Suppose that he selects mo:MusicArtist, mo:SoloMusicArtist, mo:Label, foaf:name, xsd:string. For these terms, one may derive the following constraints from ***APO***:

- mo:SoloMusicArtist and mo:Label are disjoint classes
- mo:SoloMusicArtist is a subclass of mo:MusicArtist
- each solo music artist has at most one name

Next, data is retrieved from μ_1 and μ_2, based on the selected classes and properties. Suppose that the retrieved data is expressed as the set of assertions shown in Table 1. For example, S1.2 indicates that "URI4" denotes a solo music artist and S1.4 says that "URI4" and "URI6" denote the same individual.

Table 1. Assertions expressing data returned from μ_1 and μ_2.

#	A_1 - *Assertions from* μ_1	A_2 - *Assertions from* μ_2	#
S1.1	mo:SoloMusicArtist("URI5")	mo:Label("URI7")	S2.1
S1.2	mo:SoloMusicArtist("URI4")	mo:SoloMusicArtist("URI6")	S2.2
S1.3	foaf:name("URI4", "Janis Joplin")	foaf:name("URI6", "Janis Lyn Joplin")	S2.3
S1.4	"URI4" owl:sameAs "URI6"	"URI7" owl:sameAs "URI5"	S2.4

We have the following inconsistencies: S1.1 and S1.2 violate the constraint saying that mo:SoloMusicArtist is a subclass of mo:MusicArtist (the assertions mo:MusicArtist("URI4") and mo:MusicArtist("URI5") are missing); S1.1, S2.1 and S2.4 violate the constraint saying that mo:SoloMusicArtist and mo:Label are disjoint classes; S1.2, S1.3, S1.4 and S2.3 indicate a solo music artist, identified by "URI4" and "URI6", with different names, violating the constraint that says that each solo music artist has at most one name.

So, we must address two questions. The first question refers to which constraints must hold for a data mashup specification, which are not the constraints of the domain ontology, but those that are logical consequences of such constraints and that involve only the classes and properties of the data mashup. The second question refers to how to analyze data coming from different sources to identify and isolate conflicting data.

3. A FORMAL FRAMEWORK

3.1 DL-Lite Core with Number Restrictions

We adopt DL-Lite core with arbitrary number restrictions [1], denoted $DL\text{-}Lite_{core}^{\mathcal{N}}$, a DL-Lite dialect which is useful for conceptual modeling. A *language* $\mathcal{L}$ in the $DL\text{-}Lite_{core}^{\mathcal{N}}$ dialect is characterized by a *vocabulary* V, consisting of a set of *object names*, a set of *atomic concepts*, a set of *atomic roles*, and the *bottom concept* $\bot$. The sets of *basic concept descriptions*, *concept descriptions* and *role descriptions* of $\mathcal{L}$ are defined as:

- If P is an atomic role, then P and P^- (*inverse role*) are role descriptions
- If u is an atomic concept or the bottom concept, and p is a role description, then u and $(\geq n\ p)$ (*at-least restriction*, where n is a positive integer) are basic concept descriptions and also concept descriptions
- If u is a concept description, then $\neg u$ (*negated concept*) is a concept description

An *inclusion* of $\mathcal{L}$ (or in V) is an expression of one of the forms $u \sqsubseteq v$ or $u \sqsubseteq \neg v$, where u and v are basic concept descriptions. An *assertion* of $\mathcal{L}$ (or in V) is an expression of one of the forms $C(a)$, $\neg C(a)$, $P(a,b)$, $\neg P(a,b)$, $(a \approx b)$ and $\neg(a \approx b)$, where C is an atomic concept, P is an atomic role and a and b are object names. We also say that $(a \approx b)$ and $\neg(a \approx b)$ are an *equality* and an *inequality*, respectively. A *formula* of $\mathcal{L}$ (or in V) is an inclusion or an assertion of $\mathcal{L}$.

An *interpretation* $\boldsymbol{s}$ for $\mathcal{L}$ consists of a nonempty set Δ^s, the *domain* of $\boldsymbol{s}$, and an *interpretation function*, also denoted $\boldsymbol{s}$, with the usual definition [1]. We use $\boldsymbol{s}(u)$ to indicate the value that $\boldsymbol{s}$ assigns to an expression u of $\mathcal{L}$. We say that $\boldsymbol{s}$ *satisfies* a formula σ of $\mathcal{L}$ or that $\boldsymbol{s}$ is a *model* of σ, denoted $\boldsymbol{s} \vDash \sigma$, iff

$\boldsymbol{s}(u) \subseteq \boldsymbol{s}(v)$	if σ is of the form $u \sqsubseteq v$
$\boldsymbol{s}(u) \subseteq \boldsymbol{s}(\neg v)$	if σ is of the form $u \sqsubseteq \neg v$
$\boldsymbol{s}(a) \in \boldsymbol{s}(C)$	if σ is of the form $C(a)$
$(\boldsymbol{s}(a), \boldsymbol{s}(b)) \in \boldsymbol{s}(P)$	if σ is of the form $P(a,b)$
$\boldsymbol{s}(a) = \boldsymbol{s}(b)$	if σ is of the form $(a \approx b)$
$\boldsymbol{s} \nvDash \theta$	if σ is of the form $\neg\theta$

Let Σ be a set of formulas of $\mathcal{L}$. We say that: $\boldsymbol{s}$ *satisfies* Σ or that $\boldsymbol{s}$ is a *model* of Σ, denoted $\boldsymbol{s} \vDash \Sigma$, iff $\boldsymbol{s}$ satisfies all formulas in Σ; Σ *logically implies* σ, denoted $\Sigma \vDash \sigma$, iff any model of Σ satisfies σ; Σ is *satisfiable* or *consistent* iff there is a model of Σ.

We say that a set of assertions A *induces a model* of Σ iff the interpretation s such that $a \in s(C)$ iff $C(a) \in A$ and $(a,b) \in s(P)$ iff $P(a,b) \in A$, for each atomic concept C and atomic role P, is a model of Σ. We abbreviate: "$\neg\bot$" as "$\top$" (*universal concept*), "$(\geq 1\ p)$" as "$\exists p$" (*existential quantification*), "$\neg(\geq n+1\ p)$" as "$(\leq n\ p)$" (*at-most restriction*) and "$u \sqsubseteq \neg v$" as "$u \mid v$" (*disjunction*). By an *unabbreviated* expression we mean an expression that does not use such abbreviations.

3.2 Ontologies and Knowledge Bases

We work with several notions built upon DL-Lite core with arbitrary number restrictions, defined as follows.

Definition 1:

(a) An *ontology* is a pair $\boldsymbol{O}=(V,\Sigma)$ such that
  (i) V is a finite alphabet, the *vocabulary* of $\boldsymbol{O}$, whose atomic concepts and atomic roles are called the *classes* and *properties* of $\boldsymbol{O}$, respectively, and
  (ii) Σ is a finite set of inclusions in V, the *constraints* of $\boldsymbol{O}$.

(b) A *knowledge base* is a triple $\boldsymbol{KB}=(V,\Sigma,A)$ such that
  (i) (V,Σ) is an ontology, and
  (ii) A is a finite set of assertions in V.

(c) A *data source* is a pair $\boldsymbol{DS}=(V,A)$ such that
  (i) V is a finite alphabet, and
  (ii) A is a finite set of assertions in V. □

Note that we allow equality and inequality assertions to occur as assertions of a knowledge base or of a data source (to capture owl:sameAs and owl:differentFrom OWL properties). Example 1 illustrates the concept of ontology.

Example 1: Recall that, in the example of Section 2, we adopted as domain ontology the *Agent-Person Ontology*, which corresponds to the part of *Music Ontology* depicted in Figure 1. This ontology is formalized as ***APO*** *= (V_{APO}, Σ_{APO})*, where

V_{APO} = { foaf:Agent, foaf:Person, foaf:Group, foaf:Organization, mo:MusicArtist, mo:CorporateBody, mo:SoloMusicArtist, mo:MusicGroup, mo:Label, mo:member_of, foaf:name, xsd:string }

and Σ_{APO} is the set of constraints shown in Table 2.

Table 2. Constraints of *APO* (unabbreviated form).

Constraint	Informal specification
(≥1 foaf:name) ⊑ foaf:Person	The domain of foaf:name is foaf:Person
(≥1 foaf:name⁻) ⊑ xsd:string	The range of foaf:name is xsd:string
(≥1mo:member_of) ⊑ foaf:Person	The domain of mo:member_of is foaf:Person
(≥1mo:member_of⁻)⊑ foaf:Group	The range of mo:member_of is foaf:Group
mo:MusicArtist ⊑ foaf:Agent	mo:MusicArtist is a subset of foaf:Agent
foaf:Group ⊑ foaf:Agent	foaf:Group is a subset of foaf:Agent
foaf:Organization ⊑ foaf:Agent	foaf:Organization is a subset of foaf:Agent
mo:SoloMusicArtist ⊑ foaf:Person	mo:SoloMusicArtist is a subset of foaf:Person
mo:SoloMusicArtist⊑ mo:MusicArtist	mo:SoloMusicArtist is a subset of mo:MusicArtist
mo:MusicGroup ⊑ mo:MusicArtist	mo:MusicGroup is a subset of mo:MusicArtist
mo:MusicGroup ⊑ foaf:Group	mo:MusicGroup is a subset of foaf:Group
mo:CorporateBody ⊑ foaf:Organization	mo:CorporateBody is a subset of foaf:Organization
mo:Label ⊑ mo:CorporateBody	mo:Label is a subset of mo:CorporateBody
foaf:Person ⊑ ¬foaf:Organization	foaf:Person and foaf:Organization are disjoint
foaf:Person ⊑ ¬(≥2 foaf:name)	Each person has at most one name

3.3 Constraint Graphs

The notion of constraint graph captures the structure of sets of constraints and is fundamental to construct the constraints of a data mashup specification. We introduce this notion with the help of an example and refer the reader to [5] for the details.

Note that the nodes of a constraint graph *G* are labeled with expressions and their complements. We say that the *complement* of a basic concept description *e* is ¬*e*, and vice-versa. If *c* is a concept description, then $\overline{c}$ denotes the complement of *c*. We say that node *u* is a *⊥-node* of *G* iff there are paths from node *u* to nodes *v* and $\overline{v}$, for some expression *v*. If node *u* is a ⊥-node then we say that node $\overline{u}$ is a *⊤-node*.

Example 2: Consider the set of constraints Σ_{APO}, shown in Table 2. Figure 2 depicts the graph $G(\Sigma_{APO})$ that represents Σ_{APO}, which is constructed as follows. For each inclusion *u* ⊑ *v* in Σ_{APO}, there are nodes in $G(\Sigma_{APO})$ labeled with *u*, $\overline{u}$, *v* and $\overline{v}$, and arcs from node *u* to node *v* and from node $\overline{v}$ to node $\overline{u}$. For example, the constraint foaf:Person ⊑ ¬foaf:Organization generates two arcs: an arc from node foaf:Person to node ¬foaf:Organization and an arc from node foaf:Organization to node ¬foaf:Person.

$G(\Sigma_{APO})$ is such that, if there is a path from node *u* to node *v*, then Σ_{APO} logically implies *u* ⊑ *v*. For example, since there is a path from node mo:CorporateBody to node ¬(≥2 foaf:name), Σ_{APO} logically implies mo:CorporateBody ⊑ ¬(≥2 foaf:name). Note that there is a path from node (≥2 foaf:name) to nodes foaf:Person and ¬foaf:Person. Hence, we have that Σ_{APO} logically implies (≥2 foaf:name) ⊑ foaf:Person and (≥2 foaf:name) ⊑ ¬foaf:Person. Thus, Σ_{APO} logically implies (≥2 foaf:name) ⊑ ⊥, that is, node (≥2 foaf:name) is a ⊥-node of $G(\Sigma_{APO})$.

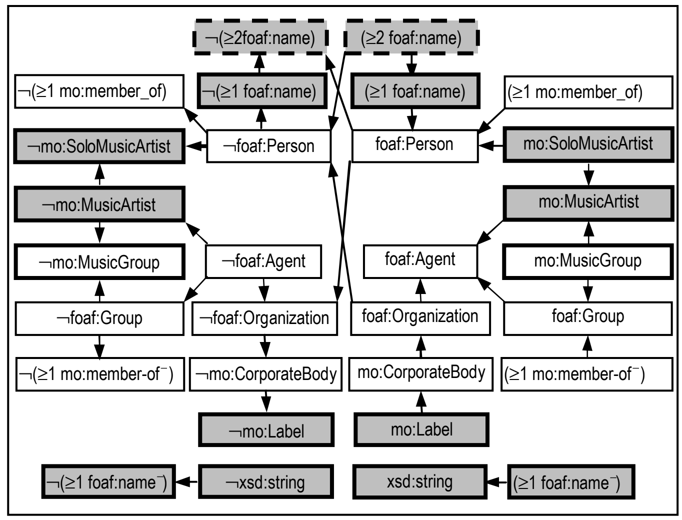

Figure 2. Graph $G(\Sigma_{APO})$ representing the constraints of *APO*.

3.4 Open Ontology Fragments

After the user selects classes and properties from the domain ontology, the mashup service must compute a set of constraints that captures their semantics. More precisely, if *W* is an alphabet, let Σ / W denote the set of formulas that use only classes and properties in *W* and that are logically implied by Σ.

Definition 2: Let ***O*** = (V_O, Σ_O) and ***F*** = (V_F, Σ_F) be two ontologies. Then, ***F*** is an *open ontology fragment* of ***O*** iff

(i) All classes and properties in V_F occur in V_O, and

(ii) Σ_F is tautologically equivalent to Σ_O / V_F. □

The next example illustrates how to generate Σ_F so that the second requirement is satisfied, using the graph representing Σ_O.

Example 3: Recall that, in the example of Section 2, the mashup is formalized as the ontology $\boldsymbol{M_0}=(V_0,\Sigma_0)$, where

V_0 = { mo:MusicArtist, mo:SoloMusicArtist, mo:Label, foaf:name, xsd:string }.

We may compute the constraints in Σ_0, shown in Table 3, as follows. First mark the nodes of the constraint graph of Σ_{APO} labeled with expressions that use only symbols in V_0 (in shaded boxes in Figure 2). Among such nodes, detect which ones are ⊥-nodes and ⊤-nodes (in dashed border lines in Figure 2). Construct the constraints in Σ_0 as follows. Let *Q* be a marked ⊥-node and *u* be an expression which labels *Q* and which uses only symbols in V_0. Add a constraint of the form *u* ⊑ ⊥ to Σ_0, as in line 1 of Table 3. Let M and N be two marked nodes, which are not a $\bot$-node or a $\top$-node, such that there is a path from M to N. Let u be a positive expression and v be an expression which label M and N, respectively, and which use only symbols in V_0. Add a constraint of the form $u \sqsubseteq v$ to Σ_0, as in lines 2, 3, 4 and 5 of Table 3. However, if $\overline{v} \sqsubseteq \overline{u}$ is in Σ_0, do not add $u \sqsubseteq v$ to Σ_0.

Table 3. Constraints of M_0 (unabbreviated form).

#	Constraint	Informal specification
1	($\geq$2 foaf:name) $\sqsubseteq \bot$	No individual has more than one name
2	mo:Label $\sqsubseteq \neg$($\geq$1 foaf:name)	Individuals in mo:Label have no name
3	($\geq$1 foaf:name$^-$) $\sqsubseteq$ xsd:string	The range of foaf:name is xsd:string
4	mo:SoloMusicArtist $\sqsubseteq \neg$mo:Label	mo:SoloMusicArtist and mo:Label are disjoint
5	mo:SoloMusicArtist $\sqsubseteq$ mo:MusicArtist	mo:SoloMusicArtist is a subclass of mo:MusicArtist

4. DATA MASHUPS

4.1 A Conceptual Framework for Mashups

A data mashup is formalized as follows.

Definition 3: Let $\boldsymbol{DO}=(V_{DO},\Sigma_{DO})$ be the domain ontology.

(a) We say that $\Phi=((V,\Sigma,A),(A_1,\ldots,A_n))$ is a *data mashup* of $\boldsymbol{DO}$ iff

   (i) $\boldsymbol{a}=(A_1,\ldots,A_n)$ is a finite list of finite sets of assertions whose atomic concepts and atomic roles occur in V;

   (ii) $\boldsymbol{KB}=(V,\Sigma,A)$ is a knowledge base such that

      a. (V,Σ) is an open ontology fragment of $\boldsymbol{DO}$, where V is called the *mashup vocabulary* and Σ is called the set of *mashup constraints*.

      b. A is a finite set of assertions whose atomic concepts and atomic roles occur in V; furthermore, there is a set of assertions $\boldsymbol{B} \subseteq A_1 \cup\ldots\cup A_n$ such that $\Sigma \cup \boldsymbol{B}$ is satisfiable and $\Sigma \cup \boldsymbol{B}$ logically implies A.

(b) We say that Φ is a *positive data mashup with equalities* of $\boldsymbol{DO}$ iff $\boldsymbol{a}$ is a finite list of finite sets of positive assertions, possibly including equalities. □

The ontology (V,Σ) is a conceptual model of what the user observes. The vocabulary V represents the classes and properties in V_{DO} that the user selected; the set of constraints Σ is computed by the mashup service, based on V and Σ_{DO}, in such a way that (V,Σ) is an open ontology fragment of $\boldsymbol{DO}$, in order to capture what constraints of the domain ontology apply to the classes and properties the user selected.

For $i=1,\ldots,n$, the set A_i models the data obtained from the i^{th} data source to populate the classes and properties in V. Note that $\Sigma \cup A_1 \cup\ldots\cup A_n$ may not be satisfiable, as discussed in Section 2.

The set of assertions A represents the data that the user observes. We take A as a logical consequence of Σ and a subset $\boldsymbol{B}$ of $A_1 \cup\ldots\cup A_n$, provided that $\Sigma \cup \boldsymbol{B}$ is satisfiable. Note that A is therefore a logical consequence of $\Sigma \cup A_1 \cup\ldots\cup A_n$ using a simple paraconsistent notion of logical implication to account for inconsistencies.

4.2 Overview

Consider the following problem, which we call *MASHUP*:

Instance: An ontology (V,Σ), built upon DL-Lite core with arbitrary number restrictions, and a list $A_1,\ldots,A_n$ of finite sets of assertions in V.

Question: What is the largest set of assertions A whose atomic concepts and atomic roles occur in V such that there is a set of assertions $\boldsymbol{B} \subseteq A_1 \cup\ldots\cup A_n$ such that $\Sigma \cup \boldsymbol{B}$ is satisfiable and $\Sigma \cup \boldsymbol{B}$ logically implies A?

We can prove that *MASHUP* is NP-Complete by a reduction of the satisfiability problem of $DL\text{-}Lite_{core}^{\mathcal{N}}$ knowledge bases with equality and inequality constraints [1]. In view of this result, we present a heuristic procedure that computes consistent mashups (not necessarily maximal) in polynomial time for sets of positive assertions. The heuristic procedure has three stages and explores a strategy to minimize the cost of consistency checking. The stages are implemented by the procedures **MashupAnalysis**, **ConsistentLocalData** and **ConsistentMashupData**.

During the first stage, the **MashupAnalysis** procedure computes the set Σ of mashup constraints so that (V,Σ) is an open fragment of $\boldsymbol{DO}$. At the second stage, each wrapper service separately invokes the **ConsistentLocalData** procedure to preprocess the assertions obtained from the data source the wrapper encapsulates to avoid inconsistencies w.r.t. Σ. In the third stage, the **ConsistentMashupData** procedure analyses the combined data passed by the wrappers to create a final set of assertions that is consistent with Σ.

4.3 Consistency Services at the Wrapper

Let $\boldsymbol{DO}=(V_{DO},\Sigma_{DO})$ be the domain ontology, V be the mashup vocabulary and Σ be the mashup constraints. For $i=1,\ldots,n$, the wrapper of the i^{th} data source first obtains a finite set A_i of positive assertions in V, including equalities. The wrapper calls the **ConsistentLocalData** procedure (see Figure 3) to compute the *completion of* A_i *w.r.t.* Σ, denoted $comp[A_i,\Sigma]$, defined as: (i) the smallest finite set that contains A_i; (ii) uses only individuals that occur in A_i; and (iii) induces a finite model of Σ.

Condition (i) corresponds to the goal that $comp[A_i,\Sigma]$ should expand A_i in the least possible way. Condition (ii) reflects the idea that $comp[A_i,\Sigma]$ should not artificially introduce new individuals (created perhaps with the help of Skolem functions). Finally, condition (iii) captures the fact that we need to construct, and pass to the user, a finite set of assertions that represent data consistent with Σ. It is not always possible to compute $comp[A_i,\Sigma]$ satisfying all three conditions (this discussion is outside the scope of this paper).

As an example, returning to Section 2, since mo:SoloMusicArtist is a subclass of mo:MusicArtist, the **ConsistentLocalData** procedure creates an assertion of the form mo:MusicArtist(a), if a wrapper obtains mo:SoloMusicArtist(a) from its data source.

In the presence of equalities, we also have to consider equivalence classes of object names. In Table 1, the equality S1.4 (expressed using owl:sameAs) indicates that “URI4” and “URI6” are equivalent, and S2.4 that “URI7” and “URI5” are equivalent. We use $[o]$ to denote the equivalence class an object name o belongs to.

ConsistentLocalData (A_i , Σ ; $comp[A_i,\Sigma]$)
begin
initialize T with a normalization of A_i;
mark all assertions in T as unprocessed;
for each unprocessed assertion $P(a,b)$ in T **do** // role assertions
begin add assertions of the form $(\geq 1\ P)(a)$ and $(\geq 1\ P^-)(b)$ to T, marked as unprocessed;
mark $P(a,b)$ as processed
end
while there is an unprocessed assertion $e(a)$ in T **do**
// class assertions
begin let M be the node of $G(\Sigma)$ labeled with e;
for each other expression f that also labels M **do**
// Σ logically implies $e \equiv f$
add an assertion $f(a)$ to T, if not already in T, marked as unprocessed;
for each node N such that (M,N) is an arc of $G(\Sigma)$ **do**
// Σ logically implies $e \sqsubseteq f$
for each expression g that labels N **do**
add an assertion $g(a)$ to T, if not already in T, marked as unprocessed;
mark $e(a)$ as processed
end
$comp[A_i,\Sigma] = T \cup \{ (a \approx b) \mid (a \approx b)$ is an equality in $A_i \}$;
return $comp[A_i,\Sigma]$
end

Figure 3. Procedure ConsistentLocalData.

Let A be a set of assertions, including equalities. The *set of equivalence classes of* A, denoted $[A]$, is the set of equivalence classes of the object names that occur in A induced by the equalities in A. A *normalization function* for A is a function $\boldsymbol{n}$ that maps each equivalence class $\boldsymbol{E}$ in $[A]$ to an individual in $\boldsymbol{E}$. The *normalization* of A w.r.t. $\boldsymbol{n}$ is the set of assertions, excluding equalities, obtained by replacing each occurrence of an object name a (of an equivalence class $[a]$) in an assertion in A (excluding equalities) by $\boldsymbol{n}([a])$.

4.4 Computing the Final Mashup

Now we describe the **ConsistentMashupData** procedure that implements a greedy strategy based on an ordering of the assertions, induced by an ordering of the data sources and, within the same data source, induced by an ordering of the symbols in the alphabet, computed from the structure of the constraint graph (equalities have precedence over the other assertions).

Before **ConsistentMashupData** is called, the **Prepare** procedure orders the symbols in V as follows. It constructs the constraint graph $G(\Sigma)$ and creates a topological sort $\boldsymbol{M}$ of the nodes in $G(\Sigma)$ labeled with positive expressions, from sinks to sources, excluding the $\bot$-nodes and $\top$-nodes. Then, **Prepare** sorts the symbols in V (excluding symbols that represent XML data types), creating a list $\boldsymbol{U}$ which is *coherent* with $\boldsymbol{M}$, that is, if M appears before N in $\boldsymbol{M}$, then all symbols that appear in expressions that label M occur in $\boldsymbol{U}$ before all symbols that appear in expressions that label N, but not in expressions that label M. The ordering of the symbols in $\boldsymbol{U}$ coherently with a topological sort of the nodes of the constraint graph helps healing inconsistencies in much the same way as the **ConsistentLocalData** procedure does.

ConsistentMashupData (in Figure 4) receives as input $\boldsymbol{U}$, Σ and $\boldsymbol{a} = (A_1,\ldots,A_n)$. It outputs $\boldsymbol{U}$, perhaps with new object names, and a finite set of assertions A in V such that $\Sigma \cup \boldsymbol{B}$ implies A, for some $\boldsymbol{B} \subseteq A_1 \cup\ldots\cup A_n$ such that $\Sigma \cup \boldsymbol{B}$ is satisfiable.

ConsistentDataMashup ($\boldsymbol{U}$, Σ, $\boldsymbol{a}$; $\boldsymbol{U}$, A)
input: $\boldsymbol{U}$ – a list of the symbols in V_0, where $V_0 \subseteq V_{DO}$
Σ – a set of constraints
$\boldsymbol{a} = (A_1,\ldots,A_n)$ – a list of finite sets of positive assertions in V_0
output: $\boldsymbol{U}$ – the original list $\boldsymbol{U}$ perhaps with new object names
A – a set of assertions that represents the mashup
begin // construct a model $\boldsymbol{s}$ of Σ and
ConstructModel ($\boldsymbol{U}$, Σ, $\boldsymbol{a}$; $\boldsymbol{s}$, $\boldsymbol{e}$) // a set of equivalence classes $\boldsymbol{e}$
ConstructMashup ($\boldsymbol{s}$, $\boldsymbol{e}$, $\boldsymbol{U}$; $\boldsymbol{U}$, A) // construct A and adjust $\boldsymbol{U}$
return $\boldsymbol{U}$, A
end

Figure 4. Procedure ConsistentDataMashup.

ConstructModel ($\boldsymbol{U}$, Σ, $\boldsymbol{a}$; $\boldsymbol{s}$, $\boldsymbol{e}$)
// $\boldsymbol{U}$, Σ and $\boldsymbol{a} = (A_1,\ldots,A_n)$ as in **ConsistentDataMashup**
begin // $\boldsymbol{s}$ is a model of Σ and $\boldsymbol{e}$ is a set of equivalence classes
$\boldsymbol{e} = \{\{a\} \mid a$ is an individual that occurs in $A_1,\ldots,A_n\}$ // initialize $\boldsymbol{e}$
for each atomic concept or atomic role v in $\boldsymbol{U}$ **do** $\boldsymbol{s}(v) = \varnothing$; // initialize $\boldsymbol{s}$
for each $i=1$ **to** n **do**
begin ProcessEqualities (Σ, A_i , $\boldsymbol{s}$, $\boldsymbol{e}$; $\boldsymbol{s}$, $\boldsymbol{e}$);
Let B_i contain all assertions in A_i which are not equalities;
ProcessAssertions ($\boldsymbol{U}$, Σ , B_i , $\boldsymbol{s}$, $\boldsymbol{e}$; $\boldsymbol{s}$)
end
return $\boldsymbol{s}$, $\boldsymbol{e}$
end

Figure 5. Procedure ConstructModel.

ProcessEqualities (Σ , A_i , $\boldsymbol{s}$, $\boldsymbol{e}$; $\boldsymbol{s}$, $\boldsymbol{e}$)
// Σ and A_i are as in **ConsistentDataMashup**
begin // $\boldsymbol{s}$ is a model of Σ and $\boldsymbol{e}$ is a set of equivalence classes
for each equality $(a \approx b)$ in A_i **do**
begin change $\boldsymbol{e}$ so that a and b become members of the same equivalence class;
change $\boldsymbol{s}$ to accommodate the new version of $\boldsymbol{e}$;
if $\boldsymbol{s}$ is still a model of Σ // a simple test
then commit the changes to $\boldsymbol{s}$ and $\boldsymbol{e}$ // accept $(a \approx b)$
else reject the changes to $\boldsymbol{s}$ and $\boldsymbol{e}$ // ignore $(a \approx b)$
end
return $\boldsymbol{s}$, $\boldsymbol{e}$
end

Figure 6. Procedure ProcessEqualities.

ConsistentMashupData first calls **ConstructModel** (in Figure 5), which calls **ProcessEqualities** and **ProcessAssertions** (in Figures 6 and 7) to construct a set $\boldsymbol{e}$ of equivalence classes of object names, using the equalities in A_i, and a model $\boldsymbol{s}$ of Σ, using the other assertions in A_i, for each $i=1,\ldots,n$. In each iteration, these two procedures check if $\boldsymbol{s}$ remains a model of Σ by testing, for each constraint $e \sqsubseteq f$ of Σ, if $\boldsymbol{s}(e) \subseteq \boldsymbol{s}(f)$. Finally, **ConsistentMashupData** calls **ConstructMashup** (in Figure 8) to create the set of assertions A of the mashup.

Note that $\Sigma \cup A$ is satisfiable since $\boldsymbol{s}$ is constructed as a model of Σ and since A is the set of (positive) assertions that represent $\boldsymbol{s}$ and $\boldsymbol{e}$. Furthermore, let $\boldsymbol{B}$ be the set of assertions actually used to construct $\boldsymbol{s}$ and $\boldsymbol{e}$ in the procedure. $\Sigma \cup \boldsymbol{B}$ logically implies A, by the construction of $\boldsymbol{s}$ and $\boldsymbol{e}$ in the procedure.

We illustrate how **ConsistentMashupData** operates using the example in Section 2. Recall that ***APO*** is the domain ontology and that the mashup vocabulary is V = { foaf:name, xsd:string, mo:Label, mo:MusicArtist, mo:SoloMusicArtist }. Also recall from Example 3 that the mashup constraints are:

Σ = { ($\geq$2 foaf:name) $\sqsubseteq \bot$, mo:Label $\sqsubseteq \neg$($\geq$1 foaf:name), ($\geq$1 foaf:name$^-$) $\sqsubseteq$ xsd:string, mo:SoloMusicArtist $\sqsubseteq \neg$mo:Label, mo:SoloMusicArtist $\sqsubseteq$ mo:MusicArtist }

```
ProcessAssertions (U , Σ , B_i , s , e ; s)
        // U=(v_1,…,v_m) ; Σ as in ConsistentDataMashup
begin   // B_i are the positive assertions in A_i, except equalities
  for k=1 to m do    // s is a model of Σ and e is a set of equiv. classes
    for each assertion σ in B_i about v_k do   // expand s
      begin                          // σ is a positive class assertion
        if v_k is an atomic concept C and σ is of the form C(a)
          then add [a] to s(C)       // tentatively add [a] to s(C)
                                     // σ is a positive role assertion
        if v_k is an atomic role P and σ is of the form P(a,b)
          then add ([a],[b]) to s(P); // tentatively add ([a], [b]) to s(P)
        if s is still a model of Σ          // a simple test
          then commit the changes to s   // accept changes to s
          else   reject the changes to s   // reject changes to s
      end
  return s
end
```

Figure 7. Procedure ProcessAssertions.

```
ConstructMashup (s , e , U ; U , A)
        //  s is a model of Σ and e is a set of equiv. classes
begin  Initialize A = ∅;  //  U and A are as in ConsistentDataMashup
   add equalities to A to express the equivalence classes in e;
   // add other assertions in A
   for each atomic concept C in U do   // one individual per equiv. class
      for each equivalence class {a_1,..,a_r} in s(C) do
          begin add C(a_1) to A and
                add a_1,..,a_r to U, if not already in U end;
   for each atomic role P in U do  // one individual per equiv. class
      for each pair of equivalence classes ({a_1,..,a_r},{b_1,..,b_s}) in s(P) do
          begin add P(a_1,b_1) to A and
                add a_1,..,a_r,b_1,..,b_s to U, if not already in U end;
   return U , A
end
```

Figure 8. Procedure ConstructMashup.

Prepare computes $G(\Sigma)$ (see Fig. 2. Next, **Prepare** creates a topological sort $\boldsymbol{M}$ of the nodes in $G(\Sigma)$ labeled with positive expressions. Finally, **Prepare** sorts the symbols in V, creating a list $\boldsymbol{U}$, which is coherent with $\boldsymbol{M}$. Assume that $\boldsymbol{U}$ is

$\boldsymbol{U}$ = (mo:MusicArtist, mo:SoloMusicArtist, mo:Label, foaf:name).

Assume that B_1 and B_2, shown in Tables 1 and 4, are the sets of assertions passed by the wrappers of the data sources μ_1 and μ_2 (the assertions in Table 4 are derived by the wrappers from those in Table 1). Also assume that μ_1 has precedence over μ_2.

Table 4. Assertions expressing data returned from μ_1 and μ_2.

#	*Assert. derived from μ_1*	*Assert. derived from μ_2*	#
S1.5	mo:MusicArtist("URI5")	mo:MusicArtist("URI6")	S2.5
S1.6	mo:MusicArtist("URI4")		

ConsistentMashupData calls **ConstructModel**, which processes the assertions in B_1 before those in B_2 and selects the symbols in the order in which they appear in $\boldsymbol{U}$. **ConstructModel** first calls **ProcessEqualities** to process the equalities in B_1, creating the equivalence classes {"URI4","URI6"}, {"URI5"} and {"Janis Joplin"}. Then, it calls **ProcessAssertions** to process the other assertions in B_1, resulting in

s(mo:MusicArtist)=s(mo:SoloMusicArtist)={{"URI4","URI6"},{"URI5"}
s(mo:Label) = ∅
s(foaf:name) = {({"URI4","URI6"},{"Janis Joplin"})}.

Next, **ConstructModel** calls **ProcessEqualities** to process the equalities in B_2. To process S2.4, **ProcessEqualities** tentatively changes {"URI5"} to {"URI5","URI7"} and s(mo:MusicArtist) and s(mo:SoloMusicArtist) to {{"URI4","URI6"},{"URI5","URI7"}}. Since s remains a model of Σ_0, **ProcessEqualities** commits these changes. Then, **ConstructModel** calls **ProcessAssertions** to process the other assertions in B_2. The interpretation

s(mo:MusicArtist) = s(mo:SoloMusicArtist)
= {{"URI4","URI6"},{"URI5","URI7"}}

remains unchanged after processing S2.2. The interpretations

s(mo:Label)=∅ and
s(foaf:name)={({"URI4","URI6"}, {"Janis Joplin"})}

also remain unchanged, since S2.1 and S2.3 cannot be considered without leading to consistency violations. Indeed, adding {"URI5","URI7"} to s(mo:Label) would violate

mo:SoloMusicArtist⊑¬mo:Label

and adding ({"URI4","URI6"},{"Janis Lyn Joplin"}) to s(foaf:name) would violate

(≥2 foaf:name) ⊑ ⊥

Finally, **ConsistentMashupData** calls **ContructMashup**, which uses s and e to create the final set of assertions A. For example, the equivalence classes {"URI4","URI6"} and {"URI5","URI7"} generate

"URI4" owl:sameAs "URI6" and "URI5" owl:sameAs "URI7"

and s(mo:SoloMusicArtist)={{"URI4","URI6"},{"URI5","URI7"}} induces
mo:SoloMusicArtist("URI4") and mo:SoloMusicArtist("URI5").

5. CONCLUSIONS

We investigated the problem of creating data mashups from potentially inconsistent sources. We first formalized the notion of data mashups and then described a heuristic procedure to compute consistent data mashups. As for current work, we are implementing a data mashup service that includes the approach.

6. REFERENCES

[1] Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M. The DL-Lite family and relations. *J. of Artificial Intelligence Research* 36, 1–69.

[2] Bizer, C., Schultz, A. The R2R Framework: Publishing and Discovering Mappings on the Web. First Int'l. Workshop on Consuming Linked Data (COLD 2010) (Nov. 2010).

[3] Bizer, C., Cyganiak, R., Heath, T. How to Publish Linked Data on the Web. (http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/) (2007).

[4] Bizer, C., Heath, T., Berners-Lee, T. Linked Data – The Story So Far. In: Heath, T., Hepp, M., and Bizer, C. (eds.). *Int'l. J. on Semantic Web and Information Systems* (2009).

[5] Casanova, M.A., Lauschner, T., Leme, L. A. P. P., Breitman, K. K., Furtado, A. L., Vidal, V. M. P. Revising the Constraints of Lightweight Mediated Schemas. *DKE* v.69 (2010) , 1274–1301.

[6] Casanova, M.A., Breitman, K. K., Furtado, A. L., Vidal, V. M. P., Macêdo, J.A.F. The Role of Constraints in Linked Data. In: Proc. ODBASE 2011, 781–799.

[7] Fan, W., Geerts, F. and Xibei, J. A Revival of Integrity Constraints for Data Cleaning. In: Proc. VLDB´08, 24–30.

[8] Hogan, A. Integrating Linked Data through RDFS and OWL: Some Lessons Learnt. In: Proc. 5th International Conference on Web Reasoning and Rule Systems (2011).

[9] Raimond, Y., Giasson, F. Music Ontology Specication. (Nov. 2010).