Integrating XML Data Sources using RDF/S Schemas: The ICS-FORTH Semantic Web Integration Middleware (SWIM)

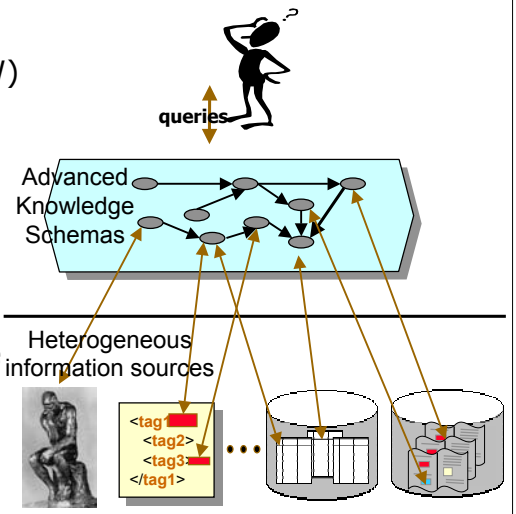
V. Christophides I. Koffina G. Serfiotis
Computer Science Department, University of Crete
and Institute for Computer Science - FORTH
Vassilika Vouton, P.O.Box 1385
GR 711 10 Heraklion, Crete, Greece
in collaboration with

V. Tannen
Computer and Information Science Department, UPenn
200 South 33rd Street,
Philadelphia, Pennsylvania, USA

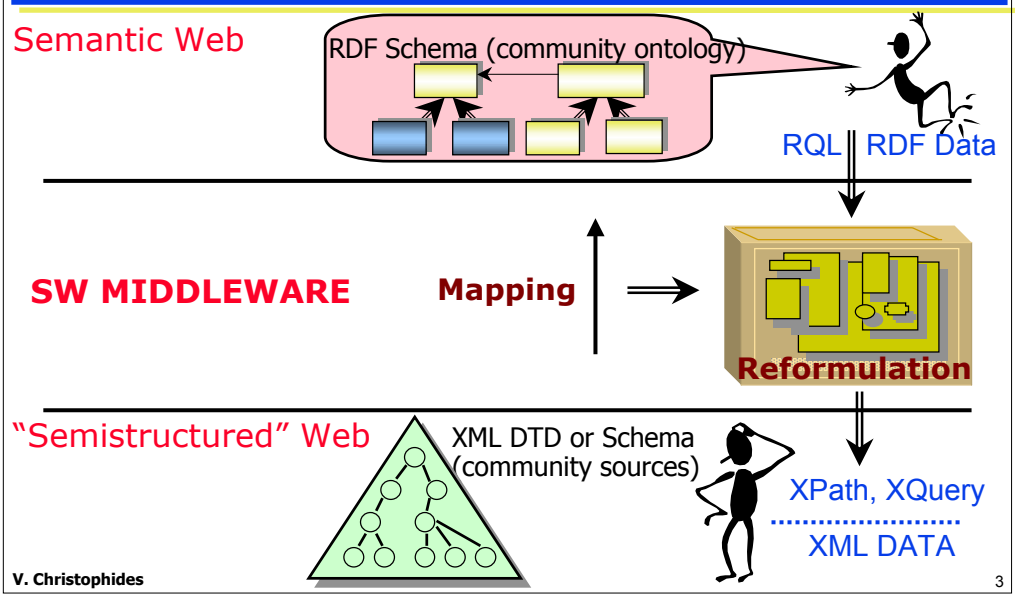
A. Deutsch
Department of Computer Science & Engineering, UCSD
9500 Gilman Drive
La Jolla, CA 92093, USA

Need for Data Integration on the Semantic Web

- The bulk of existing Web data is not yet in RDF/S (or any other form suitable for the SW)
  - Data physically stored in relational DBs and/or published as virtual XML
- SW applications require viewing data as virtual RDF
  - valid instances of domain or application-specific RDF/S schemas
- Need the ability to manipulate data with high-level query or view languages (RQL, RVL)
- How to do it?
  - republish XML as RDF
  - publish relational data as RDF
  - do both



Motivation: Republish XML as RDF

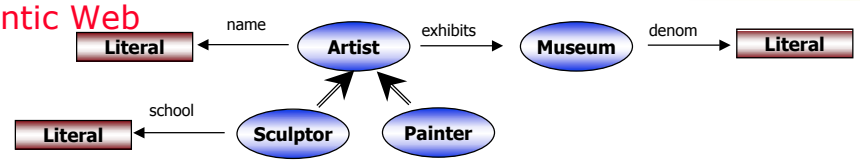


Semantic Web Integration Middleware (SWIM)

- Practical concerns:
  - XML publishing systems often provide an XML query interface
    - SW middleware can function as an alternative to the XML publishing systems;
    - SW middleware provides direct access to underlying DBMSs
  - SW middleware may also be required to integrate DBMS data with data in native XML storage
- SW middleware data integration services:
  - Specify mappings: XML→ RDF, RDB → RDF
  - Verify conformance to the semantics of employed schemas
  - Reformulate queries (i.e., compose RQL queries with mappings to produce XML or RDB queries)
  - Provide further abstractions of RDF data/schemas (RVL views)
  - Compose queries with views

Motivating Example

Semantic Web



"Semistructured" Web

```
<!DOCTYPE Museums [
<!ELEMENT Museums (Museum*)>
<!ELEMENT Museum (name, address?,
                  Collection*)>
<!ELEMENT Collection (Artifact*)>
<!ATTLIST Collection kind
          (painting|sculpture) "painting">
<!ELEMENT name (#PCDATA)>
```

```
<!ELEMENT address (#PCDATA)>
<!ELEMENT Artifact (title, Artist)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT Artist (school?)>
<!ATTLIST Artist name ID #IMPLIED>
<!ATTLIST Artist ref IDREF #IMPLIED>
<!ELEMENT school (#PCDATA)>
]>
```

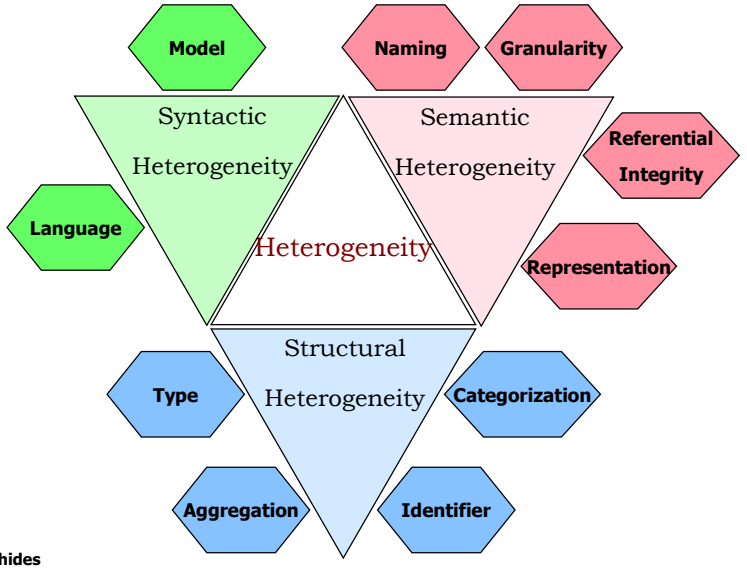

Motivating Example

```
<Museums>
 <Museum>
   <Name>Louvre</Name>
   <Collection kind="Sculpture">
     <Artifact>
       <title>Equestrian Statue</title>
       <Artist name="Leonardo Da Vinci"/>
     </Artifact>
     <Artifact>
       <title>Madonna with two
                          angels</title>
       <Artist name="Giovanni di Agostino"/>
     </Artifact>
   </Collection>

    <Collection kind="Painting">
      <Artifact>
        <title>La Joconde</title>
        <Artist ref="#Leonardo Da Vinci"/>
      </Artifact>
   </Collection>
</Museum>
...
</Museums>
```

```
<RDF>
  <Sculptor rdf:about="#Vinci">
     <name>Leonardo Da Vinci </name>
     <exhibits rdf:about ="#Louvre"/>
</Sculptor>
<Sculptor rdf:about="#Agostino">
     <name>Giovanni di
                    Agostino</name>
     <exhibits rdf:about ="#Louvre"/>
</Sculptor>
<Painter rdf:about="#Vinci">
     <name>Leonardo Da Vinci </name>
     <exhibits rdf:about ="#Louvre"/>
</Painter>
<Museum rdf:about="#Louvre">
     <denom>Louvre</denom>
</Museum>
</RDF>
```

Various Forms of Heterogeneity



Syntactic/Structural/Semantic Discrepancies

- Data Model
  - XML labeled tree (labels at nodes) vs RDF labeled graph (labels at nodes and edges)
  - Nesting vs subsumption
  - Tight vs loose coupling with document physical support
- Categorization
  - Unlike our XML DTD, our RDF schema represents *Painter* and *Sculptor* as subclasses of *Artist*
- Type
  - In our RDF schema we have *Sculptors* (*Painters*), while in our XML DTD there are collections of kind "Sculpture" ("Painting")
- Representation
  - In our XML DTD *Artifacts* contain information about their *Artists*, while our RDF schema *Artists* exhibit in a *Museum*
- Name
  - XML Museum *name* vs *denom*ination

SWIM Logic Framework (SWLF)

- Several reasons for representing RQL queries/views using a Datalog framework along with constraints:
  - Reducing the RDF→XML query reformulation problem to the relational equivalent exploits background theory on relational query optimization
    - query containment & minimization,
    - query composition,
    - query rewriting using views

    are solvable for a fairly large class of queries in the presence of certain classes of constraints
  - A robust formalism to rely on: union of conjunctive queries (linear Datalog) of the form $q(\vec{\chi}) \leftarrow \bigvee_{i=1}^{l} \phi_i(\vec{\chi}, \vec{\psi}_i)$ interpreted as:
    - $\bigvee_{i=1} \exists \vec{\psi}_i \phi_i(\vec{\chi}, \vec{\psi}_i)$

SWIM Logic Framework (SWLF)

- The constraints considered in SWIM are disjunctive embedded dependencies (DEDs) of the form:

$$\forall \vec{\chi} \left[\phi(\vec{\chi}) \rightarrow \bigvee_{i=1}^{l} \exists \vec{\psi}_i \phi_i'(\vec{\chi}, \vec{\psi}_i) \right]$$

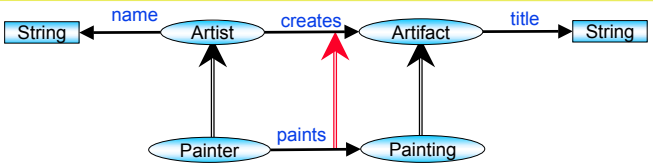
  - DEDs restrictions:
    - No negation
    - No recursion

- Our first order logic (SWLF) representation for RDF/S schemas and resource descriptions consists of:
  - predicates capturing RDF/S schema classes and properties
  - a set of constraints (DEDs) capturing RDF/S semantics

RDF/S Representation in SWLF

- RDF/S schemas and resource descriptions are represented by the following predicates:
  - CLASS(name: Class): All the RDF/S schema classes (*name*)
  - C_SUB(subC:Class, class:Class): Class *subC* is a subclass of *class*
  - PROP(subject:Class, predicate:Property, object:Class): All the RDF/S schema properties (*predicate*) with domain (*subject*) and range (*object*) restrictions
  - P_SUB(subP:Property, prop:Property): Property *subP* is a subproperty of *prop*
  - C_EXT(class:Class, inst:Resource): Resource *inst* belongs to extent of *class*
  - P_EXT(subject:Resource, predicate:Property, object:Resource): Resources *subject* and *object* belong to the extent of property *predicate*

From RDF/S Schemas to SWLF: Example



- Some of the relations for the RDF schema above:

CLASS

Artist
Artifact
Painter
Painting

PROP

Artist	Name	String
Artifact	Title	String
Artist	Creates	Artifact
Painter	Paints	Painting

P_SUB

paints	creates

C_SUB

Painter	Artist

General RDF Constraints

- Basic constraints:
  1. every resource in the extent of a class implies the existence of the corresponding class

     $\forall c, x\ (\ C_EXT(c, x) \rightarrow CLASS(c)\ )$
  2. every statement in the extent of a property implies the existence of the corresponding property

     $\forall x, p, y\ (\ P_EXT(x, p, y) \rightarrow \exists c, d\ PROP(c, p, d)\ )$
  3. every subclass of a class is also a class

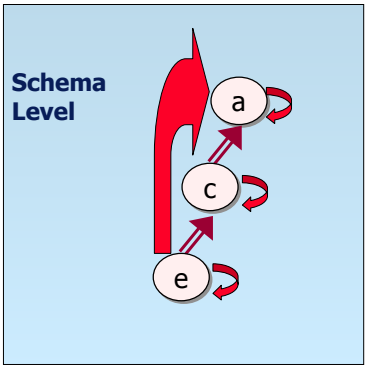
     $\forall c_1, c_2\ (\ C_SUB(c_1, c_2) \rightarrow CLASS(c_1) \wedge CLASS(c_2)\ )$
  4. every sub-property of a property is also a property

     $\forall p, q\ (\ P_SUB(p, q) \rightarrow \exists a, b, c, d\ (\ PROP(a, p, b) \wedge PROP(c, q, d)\ )\ )$
  5. the domain & range of every property is a class

     $\forall c_1, c_2\ (\ PROP(c_1, p, c_2) \rightarrow CLASS(c_1) \wedge CLASS(c_2)\ )$
  6. the domain & range of every property is unique

     $\forall a_1, a_2, p, b_1, b_2\ (\ PROP(a_1, p, b_1) \wedge PROP(a_2, p, b_2) \rightarrow a_1 = a_2 \wedge b_1 = b_2\ )$

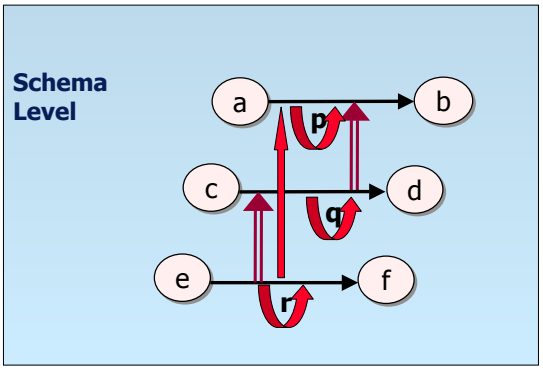
General RDF Constraints



Every class is a subclass of itself and the subclass relationship is transitive

- SUB constraints
  1. $\forall c\ (\ CLASS(c) \rightarrow C_SUB(c, c)\ )$ (subclass reflexivity)
  2. $\forall c_1, c_2, c_3\ (\ C_SUB(c_1, c_2) \wedge C_SUB(c_2, c_3) \rightarrow C_SUB(c_1, c_3)\ )$ (subclass transitivity)

General RDF Constraints

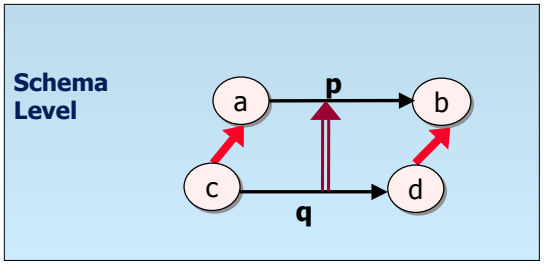


Every property is a sub-property of itself and the sub-property relationship is transitive

❸ $\forall a, p, b\ (\ PROP(a, p, b) \rightarrow P_SUB(p, p)\ )$ (subproperty reflexivity)

❹ $\forall p_1, p_2, p_3\ (\ P_SUB(p_1, p_2) \wedge P_SUB(p_2, p_3) \rightarrow P_SUB(p_1, p_3)\ )$ (subproperty transitivity)

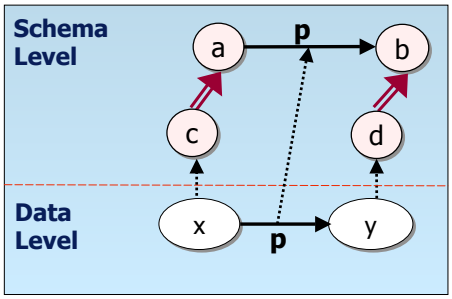
General RDF Constraints



In a valid RDF description schema the domain (range) of every sub-property is subsumed by the domain (range) of its super-property

- Domain-range constraints
  - ❶ $\forall a, p, b, c, q, d\ (\ P_SUB(q, p) \wedge PROP(a, p, b) \wedge PROP(c, q, d) \rightarrow C_SUB(c, a) \wedge C_SUB(d, b)\ )$ (subproperty-subclass compatibility)

General RDF Constraints



In a valid RDF description base the subject/object resources in every statement are instances of the property's domain/range classes (either direct or a subclass' instances)

❷ $\forall a, p, b, x, y$ ($PROP(a, p, b) \wedge P_EXT(x, p, y) \rightarrow \exists c, d$ ($C_SUB(c, a) \wedge C_SUB(d, b) \wedge C_EXT(c, x) \wedge C_EXT(d, y)$))
(property-class extent compatibility)

SWLF Semantics vs. RDF-MT

- RDF-MT does not consider different sorts of resources
  - SWLF distinguish between classes, properties and their corresponding instances
- Additional SWLF Constraints
  - SWLF 6th basic constraint stating that each property has exactly one domain & range
  - SWLF 1st domain-range constraint stating that the domain (range) of every sub-property is subsumed by the domain (range) of its super-property
- These additional SWLF constraints degrease the complexity of RQL/RVL query containment and minimization algorithms
  - result to optimal query reformulations

XML Representation in SWLF

- Semi-structured XML data are represented by the following basic predicates:
  - T(e: int, l: String): e is the integer that identifies a node (in the XML tree) with label l.
  - C(s: int, t: int): there is a parent – child relation between nodes with identifiers s, t respectively (s is the father)
  - D(s: int, t: int): there is an ancestor – descendant relation between nodes with identifiers s, t respectively (s is the ancestor)
  - A(e:int, n:String, v:String): there is an attribute with identifier e, name n and value v.
  - Txt(e: int, str: string): node with identifier e has as content the string str
  - E(s: int, t: int, l:label): there is a parent – child relation between nodes with identifiers s and t and additionally node with identifier t has a label l

From XML Data to SWLF: Example

myXMLroot

0

T

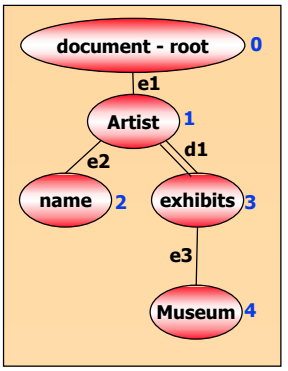
D

s	t
1	3

e	l
3	exhibits

E

s	t	l
0	1	Artist
1	2	name
3	4	Museum



General XML Constraints

- Basic Constraints
  - ❶every y that is a child of x implies that y is a descendant of x
    $\forall x,y\ [C(x, y) \rightarrow D(x, y)]$ or $\forall x,y,l\ [E(x, y, l) \rightarrow D(x, y)]$ (base)
  - ❷every y that is a descendant of x and every z is a descendant of y implies that z is a descendant of x
    $\forall x,y,z\ [D(x, y) \wedge D(y, z) \rightarrow D(x, z)]$ (transitivity)
  - ❸every node is a descendant of itself
    $\forall x,l\ [T(x, l) \rightarrow D(x, x)]$ or $\forall x,y,l\ [E(y, x, l) \rightarrow D(x, x)]$ (reflexivity)
  - ❹every x that is a descendant of y and y is a descendant of x then x and y are the same node
    $\forall x,y\ [D(x, y) \wedge D(y, x)] \rightarrow x=y]$ (anti-symmetry)
  - ❺every x that is child of y and y is child of x then x and y are the same node
    $\forall x,y,z\ [C(x, z) \wedge C(y, z) \rightarrow x=y]$ or $\forall x,y,z,l\ [E(x, z, l) \wedge E(y, z, l) \rightarrow x=y]$ (one parent)

General XML Constraints

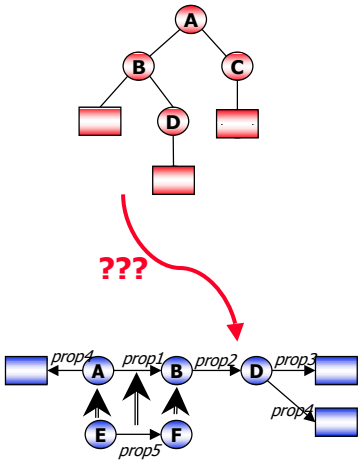
- ❻every y that is a child of x implies that both x, y are nodes
  $\forall x,y\ [C(x, y) \rightarrow \exists z,k,l1,l2\ [[T(x, l1) \vee E(z, x, l1)] \wedge [T(y, l2) \vee E(k, y, l2)]]]$ (nodes)
- ❼every y that is descendant of x implies that x, y are nodes
  $\forall x,y\ [D(x, y) \rightarrow \exists z,k,l1,l2\ [[T(x, l1) \vee E(z, x, l1)] \wedge [T(y, l2) \vee E(k, y, l2)]]]$ (nodes)
- ❽every node has at most one tag
  $\forall x,l1,l2\ [T(x, l1) \wedge T(x, l2) \rightarrow l1=l2]$ or $\forall x,y,l1,l2\ [E(y, x, l1) \wedge E(y, x, l2) \rightarrow l1=l2]$ or $\forall x,y,l1,l2\ [E(y, x, l1) \wedge T(x, l2) \rightarrow l1=l2]$ (one tag)
- ❾every attribute has at most one value
  $\forall x,n,v1,v2\ [A(x, n, v1) \wedge A(x, n, v2) \rightarrow v1=v2]$ (one attribute)
- ❿there exist only one root node
  $\forall x,y\ [Root(x) \wedge Root(y) \rightarrow x=y]$ (one root)

General XML Constraints

1. every y that is root and it is descendant of x then x is root too
   $\forall x,y\ [D(x, y) \wedge Root(y) \rightarrow Root(x)]$ (top root)
2. every z that is descendant of both x and y then x and y have a descendant relationship or are the same node
   $\forall x,y,z\ [D(x, z) \wedge D(y, z) \rightarrow x=y \vee D(x, y) \vee D(y,x)]$ (linear)
3. every y that is a child of x and there is a node z between them then z is the same node with either x or y
   $\forall x,y,z\ [C(x, y) \wedge D(x, z) \wedge D(z, y) \rightarrow x=z \vee y=z]$ or $\forall x,y,z,l\ [E(x, y, l) \wedge D(x,z) \wedge D(z,y) \rightarrow x=z \vee y=z]$ (choice)

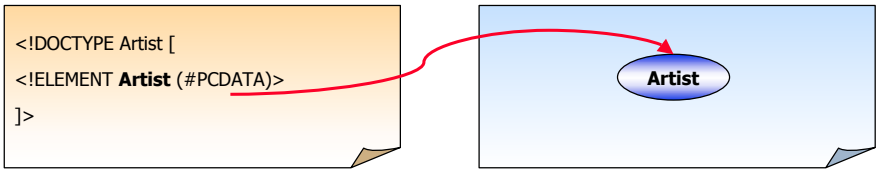
SWIM Mapping Rules

- SWIM rules map XML fragments to RDF/S classes & properties:
  $$\phi_{RPred}(\overline{\chi}) :- \phi_{XPred}(\overline{\chi}')$$
  - $\phi_{RPred}(\overline{x})$: conjunction of SWLF RDF/S predicates (C_EXT, P_EXT)
  - $\phi_{XPred}(\overline{x}')$: conjunction of SWLF XML predicates (Root, T, C, D, E, Txt)
  - $\overline{x} \subseteq \overline{x}'$ are variables
- SWIM adopts the GLAV approach by representing mappings in terms of constraints of the following form:
  - $\forall \overline{\chi} \left[\phi_{RPred}(\overline{\chi}) \rightarrow \bigvee_{i=1}^{l} \exists \overline{\psi}_i \phi_{XPred\ i}'(\overline{\chi}, \overline{\psi}_i) \right]$ GAV
  - $\forall \overline{\chi} \left[\phi_{XPred}(\overline{\chi}) \rightarrow \bigvee_{i=1}^{l} \exists \overline{\psi}_i \phi_{RPred\ i}'(\overline{\chi}, \overline{\psi}_i) \right]$ LAV



SWIM Mapping Examples

- XML tags correspond to RDF/S class (unique URIs for resource instances) or literal nodes

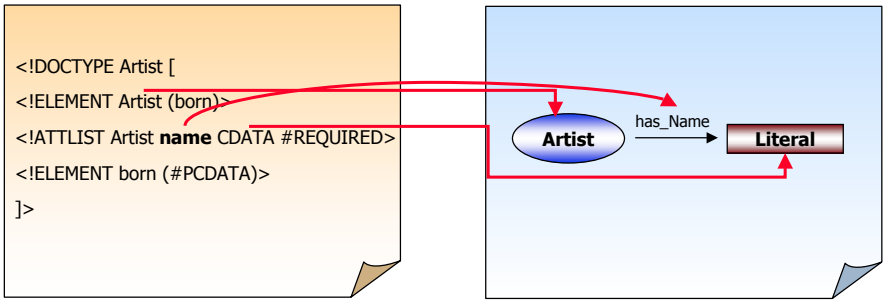


- Mapping an XML node to an RDF/S class node

  C_EXT(Artist, x) :- {//Artist}(x).
  - ∀ce in C_EXT ∃a,s,x Root (a), ∃D(a,s), ∃T(s,"Artist"), ∃Txt(s,x), ce.class="Artist", ce.inst=x.
  - ∀a,s,x Root (a), D(a,s), T(s,"Artist"), Txt(s,x) ∃ce in C_EXT ce.class="Artist", ce.inst=x.

SWIM Mapping Examples

- XML attributes (atomic types) or textual elements are mapped to properties with range of type literal



- Mapping an XML path to an RDF/S property edge

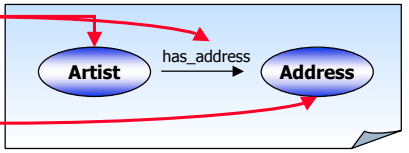
  C_EXT(Artist, x):- {//Artist}(k), {./@name}(k, x).

  P_EXT(x, has_Name, y):- {//Artist}(k), {./@name}(k, y), x=y.

SWIM Mapping Examples

- XML complex elements are additionally mapped to RDF properties with range class

```
<!DOCTYPE Artist [
<!ELEMENT Artist (born, address?)>
<!ATTLIST Artist name CDATA #REQUIRED>
<!ELEMENT born (#PCDATA)>
<!ELEMENT address (city)>
<!ATTLIST address street CDATA #REQUIRED>
<!ELEMENT city (#PCDATA)>
]>
```



- Mappings:

C_EXT(Artist, x):- {//Artist}(k), {./@name}(k, x).

C_EXT(Address, x):- {//address}(k), {./@street}(k, x).

P_EXT(x, has_address, y):- {//Artist}(k), {./@name}(k, x), {./address/@street}(k, y).

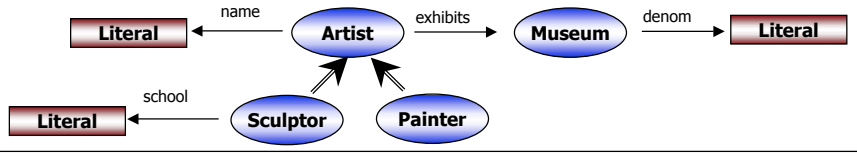
XPath Fragment employed in SWIM Mappings

- SWIM mappings from XML to RDF/S rely on XPath 2.0
- We employ the following fragment of XPath 2.0:
  - Contains all navigational axes except axes for those about siblings (following, preceding, …)(document order is disregarded)
  - Contains wildcard * only for attributes (no to:a/*/b, but yes to:a/@*) (higher complexity)
  - Accepts abbreviated syntax (a/b instead of a/child::b)
- Additionally we are able to employ non interpreted functions (e.g. for handling strings like concat, split) in order to establish more expressive mappings (e.g., involving complex keys)

RQL Query Reformulation to XML Data Sources

- Basic steps:
  - RQL query patterns are translated into a SWLF conjunctive query
  - SWLF query is chased in order to be composed with the RDF/S schema information and constraints
  - SWLF query is backchased in order to be minimized in terms of the P_EXT and C_EXT predicates
  - Further chasing is used in order to compose the minimized SWLF query with the mappings
  - Then the resulting query is minimized once more with the help of additional constraints (if any) from the data sources
  - The SWLF reformulated query is finally translated into XPath or XQuery in order to be executed by the XML data sources

XML vs. RDF Schema




```
<!DOCTYPE Museums [
<!ELEMENT Museums (Museum*)>
<!ELEMENT Museum (name, address?,
                  Collection*)>
<!ELEMENT Collection (Artifact*)>
<!ATTLIST Collection kind
            (painting|sculpture) "painting">
<!ELEMENT name (#PCDATA)>
```

```
<!ELEMENT address (#PCDATA)>
<!ELEMENT Artifact (title, Artist)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT Artist (school?)>
<!ATTLIST Artist name ID #IMPLIED>
<!ATTLIST Artist ref IDREF #IMPLIED>
<!ELEMENT school (#PCDATA)>
]>
```

XML vs. RDF Data

```
<Art>
 <Museum>
   <Name>Louvre</Name>
   <Collection kind="Sculpture">
     <Artifact>
       <title>Equestrian Statue</title>
       <Artist name="Leonardo Da Vinci "/>
     </Artifact>
     <Artifact>
       <title>Madonna with two
                          angels</title>
       <Artist name="Giovanni di Agostino"/>
     </Artifact>
   </Collection>

    <Collection kind="Painting">
      <Artifact>
        <title>La Joconde</title>
        <Artist ref="#Leonardo Da Vinci "/>
      </Artifact>
   </Collection>
</Museum>
...
</Art>
```

```
<RDF>
  <Sculptor rdf:about="#Vinci">
     <name>Leonardo Da Vinci </name>
     <exhibits rdf:about ="#Louvre"/>
</Sculptor>
<Sculptor rdf:about="#Agostino">
     <name>Giovanni di
                        Agostino</name>
     <exhibits rdf:about ="#Louvre"/>
</Sculptor>
<Painter rdf:about="#Vinci">
     <name>Leonardo Da Vinci </name>
     <exhibits rdf:about ="#Louvre"/>
</Painter>
<Museum rdf:about="#Louvre">
     <denom>Louvre</denom>
</Museum>
</RDF>
```


RDF/S Facts & XML -> RDF/S Mappings

- FACTS

```
CLASS(Artist).
CLASS(Painter).
CLASS(Sculptor).
CLASS(Museum).
C_SUB(Painter, Artist).
C_SUB(Sculptor, Artist).
PROP(Artist,exhibits,Museum).
PROP(Artist,name, String).
PROP(Museum,denom,String).
PROP(Sculptor,school,String).
```

- MAPPINGS

```
C_EXT(Artist,x) :- {//Collection}(k),
   {./@kind}(k,s), {.//Artist/@name}(k,x),
   s="Scultrure".
C_EXT(Artist,x) :- {//Collection}(k),
   {./@kind}(k,s), {.//Artist/@name}(k,x),
   s="Painting".
C_EXT(Sculptor,x) :- {//Collection}(k),
   {./@kind}(k,s), {.//Artist/@name}(k,x),
   s="Scultrure".
C_EXT(Painter,x) :- {//Collection}(k),
   {./@kind}(k,s), {.//Artist/@name}(k,x),
   s="Painting".
C_EXT(Museum,x) :- {//Museum}(k),
   {./name}(k,x).
P_EXT(x,exhibits,y) :- {//Museum}(k),
   {./name}(k,y),
   {./Collection//Artist/@name}(k,x).
P_EXT(x,exhibits,y) :- {//Museum}(k),
   {./name}(k,y),
   {./Collection//Artist/@ref}(k,x).
P_EXT(x,denom,y) :- {//Museum}(m),
   {./name}(m,y), x=y.
```

A Full Example

- Given our example RDF schema we pose the following RQL query:

  Find artists exhibiting in Louvre

  SELECT X
  FROM {X}exhibits{Y}, {Y}denom{Z}
  WHERE Z = "Louvre"

- The class and property patterns of this RQL query will be translated:

  ans(x):- P_SUB(A, exhibits), P_EXT(x, A, y),
  P_SUB(B, denom), P_EXT(y, B, z),
  z="Louvre".

A Full Example

- Chase starts…
- Taking into consideration the facts about subproperties and subclasses:

  ans(x):-P_SUB(A, exhibits), P_EXT(x, A, y),
  P_SUB(B, denom), P_EXT(y, B, z),
  A = exhibits, B=denom, z="Louvre".

∀x, p, y (P_EXT(x, p, y) → ∃c, d PROP(c, p, d)) ❺

- Eliminating some trivial predicates and applying the property domain/range constraint:

  ans(x):- P_EXT(x, A, y), P_EXT(y, B, z),
  PROP(s, A, t), PROP(m, B, k), A = exhibits,
  B=denom, z="Louvre".

A Full Example

- Due to facts about properties:
  ans(x):- P_EXT(x, A, y),P_EXT(y, B, z)
  PROP(s, A, t), PROP(m, B, k),
  A = exhibits, B=denom,
  s= Artist, t = Museum, m=Museu
  k = String, z="Louvre".

∀a, p, b, x, y PROP(a, p, b) ∧ P_EXT(x, p, y) → ∃c, d C_SUB(c, a) ∧ C_SUB(d, b) ∧ C_EXT(c, x) ∧ C_EXT(d, y)

11

- Apply the property instance domain/ra constraint:
  ans(x):- P_EXT(x, A, y), P_EXT(y, B, z),
  PROP(s, A, t), PROP(m, B, k),
  C_SUB(a, s), C_EXT(a, x), C_SUB(b, t), C_EXT(b, y),
  C_SUB(c, m), C_EXT(c, y), C_SUB(d, k), C_EXT(d, z),
  A = exhibits, B=denom,
  s= Artist, t = Museum, m = Museum, k = String, z="Louvre".

A Full Example

- Using the facts about subclasses of class Artist we get the following:

  ans(x):- P_EXT(x, exhibits, y), P_EXT(y, denom, z),
  C_EXT(Artist, x), C_EXT(Museum, y), z="Louvre".
  U
  ans(x):- P_EXT(x, exhibits, y), P_EXT(y, denom, z),
  C_EXT(Painter, x), C_EXT(Museum, y), z="Louvre".
  U
  ans(x):- P_EXT(x, exhibits, y), P_EXT(y, denom, z),
  C_EXT(Sculptor, x), C_EXT(Museum, y), z="Louvre".

- At this point chase has finished and the result of the backchase is given in an abbreviated form

A Full Example

- Further chasing is used in order to compose the minimized SWLF query with the mappings

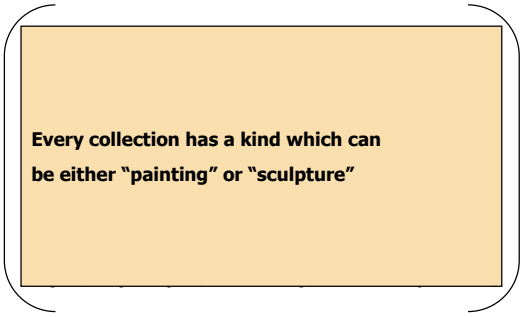
ans(x):-{//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre"
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x),
{//Collection}(n), {./@kind}(n,q), {.//Artist/@name}(n,x), q=Sculpture.
U
...
{//Collection}(n), {./@kind}(n,q), {.//Artist/@name}(n,x), q=Painting.
U
ans(x):-{//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre"
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@ref}(s,x).
{//Collection}(n), {./@kind}(n,q), {.//Artist/@name}(n,x), q=Sculpture.
U
...
{//Collection}(n), {./@kind}(n,q), {.//Artist/@name}(n,x), q=Painting.

Towards More Efficient Reformulations

- Until now the RQL query has been reformulated with the use of general constraints that are part of the semantics of both the RDF & XML data models
- However, using additional constraints from the XML data sources we are able to perform more efficient reformulation
  - XML, for example, gives us the ability to specify keys and keyrefs, as well as value enumerations for attributes
- Such additional constraints can improve our reformulations by eliminating redundant queries (previous example) or by simplifying the generated queries ("cutting" some predicates)

A Full Example (cont.)

- The fact that attribute kind of the Collection is an enumerated type can be encoded in SWIM as a constraint:



A Full Example

- Further backchasing with the help of the constraint defined previously

- ans(x):- {//Museum}(k), {./name}(k,y),
  {//Museum}(m), {./name}(m,y),z=y
  {//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x),
  {//Collection}(n), {.//Artist/@name}(n,x), z="Louvre".
  ∪
  ans(x):- {//Museum}(k), {./name}(k,y),
  {//Museum}(m), {./name}(m,y),z=y
  {//Museum}(s), {./name}(s,y), {./Collection//Artist/@ref}(s,x).
  {//Collection}(n), {.//Artist/@name}(n,x) , z="Louvre".

A Full Example (cont.)

- Constraints for elements or attributes defined as key or keyref in XML Schema

```
<key name="name">
   <selector xpath="//Museum"/>
   <field xpath="name"/>
</key>
```

```
<key name="aname">
   <selector xpath="//Artist"/>
   <field xpath="@name"/>
</key>
```

```
<keyref name="refer"
        refer="aname">
   <selector xpath=
              "//Collection//Artist"/>
   <field xpath="ref"/>
</keyref>
```


Taking advantage of XML Constraints

- The definition of key constraints is encoded in SWIM as follows:

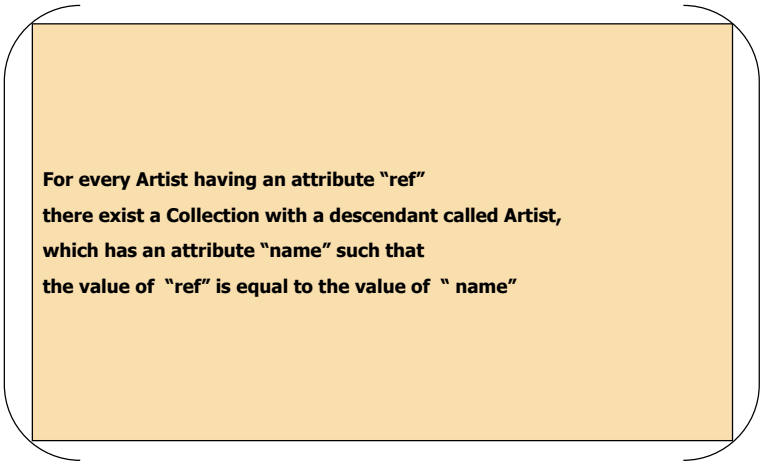
Every Museum has at least one "name"

For every Museum having two "name"s then these "name"s are equal

For every two Museums having the same "name" then these Museums are the same

Taking advantage of XML Constraints

- The definition of a keyref constraint is encoded in SWIM as follows:



A Full Example (cont.)

- The previous query was reformulated into:

```
ans(x):- {//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre",
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x),
{//Collection}(n), {.//Artist/@name}(n,x).
U
ans(x):- {//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre",
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@ref}(s,x).
{//Collection}(n), {.//Artist/@name}(n,x).
```

- Keyref constraint will add to the second query the following:

```
{//Collection}(w), {.//Artist/@name}(w,o), o=x.
```

A Full Example (cont.)

- The previous query was reformulated into (with keys):

ans(x):- {//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre",
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x),
{//Collection}(n), {.//Artist/@name}(n,x).
∪
ans(x):- {//Museum}(k), {./name}(k,y),
{//Museum}(m), {./name}(m,y),z=y, z="Louvre",
{//Museum}(s), {./name}(s,y) {./Collection//Artist/@ref}(s,x),
{//Collection}(n), {.//Artist/@name}(n,x),
{//Collection}(w), {.//Artist/@name}(w,o), o=x.

A Full Example (cont.)

- The previous query was reformulated into:

ans(x):- z=y, z="Louvre",
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x).
∪
ans(x):- z=y, z="Louvre",
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@ref}(s,x),
{//Collection}(n), {.//Artist/@name}(n,x)

- And finally the first query is a superset of the second one. So…

ans(x):-
{//Museum}(s), {./name}(s,y), {./Collection//Artist/@name}(s,x),
y="Louvre".

Translating SWIM Queries into XPath/XQuery

- The previous query should be translated into a query language that is appropriate for retrieving data from XML sources
- Two target XML query languages:
  - XPath (for some RQL queries)
  - XQuery (for all RQL queries)
- *XPath:*

```
//Museum[./name="Louvre"]//Artist/@name

<name>Giovanni di Agostino</name>
<name>Leonardo Da Vinci</name>
```


XPath Translation

- XPath returns nodes, so the result is not an XML document and of course not an RDF description (as desired)
- XPath 1.0 is not able to project over two node values
  - i.e., return both first and last name of an Artist
- XPath 1.0 does not support explicit quantification
  - i.e., to concatenate first and last name of each Artist
- XPath 2.0 is able to express more complex queries due to the fact that can use for, let ,… and variable bindings
- So …

  XPath 1.0 only for simple RQL queries (projections on one variable) and simple XML constraints (no complex keys)

XQuery Translation

- *XQuery:*

```
<RDF>
{
 <Bag>
   { for $var0 in document("art.xml")
     for $var1 in $var0//Museum
     for $var2 in $var1/name
     for $var3 in $var1//Artist
     for $var4 in $var3/@name
     where ($var2/text()="Louvre")
     return
         <li>{$var4/text()}</li>}
 </Bag>
 }
</RDF>
```


XQuery Translation

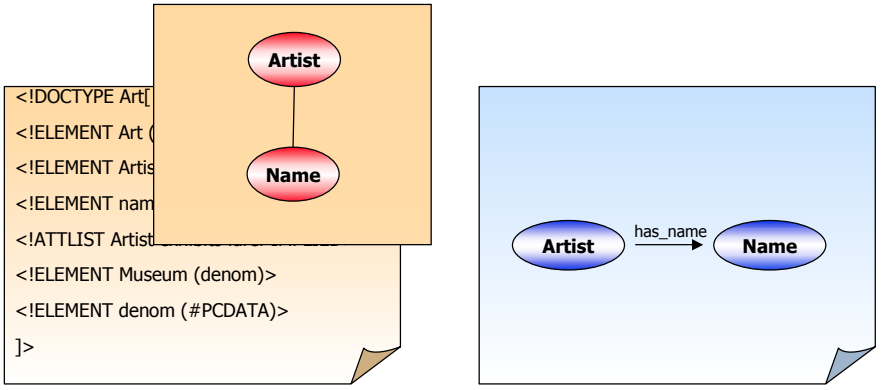
- The result of this query will be:

```
<RDF>
    <Bag>
        <li>Giovanni di Agostino</li>
        <li>Leonardo Da Vinci</li>
        …
    </Bag>
</RDF>
```

- XQuery 1.0 is used for the above transformation
- XQuery has the ability to construct elements and produce a well-formed or valid XML document
  - With appropriate tag selection in the return clause we can return a valid RDF description according to the employed RDF/S schema

Expressiveness of SWIM Mappings

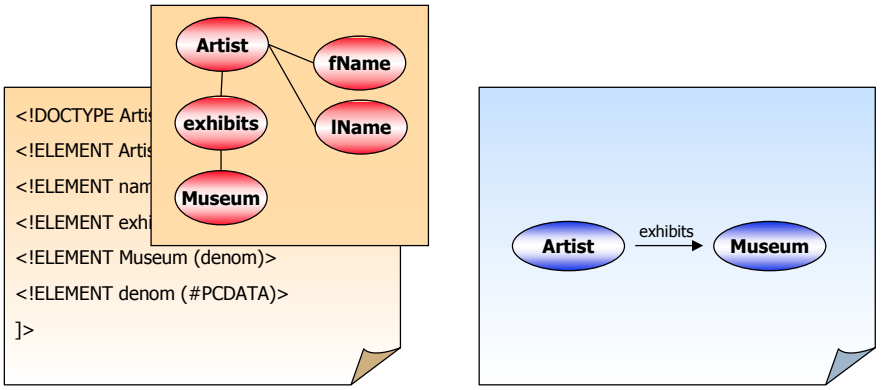
- Mappings are specified through simple or complex XPath or RQL path expressions:



- Mapping an XML path to an RDF/S property

  P_EXT(x, has_Name, y):- {//Artist}(x), {./Name}(x, y)

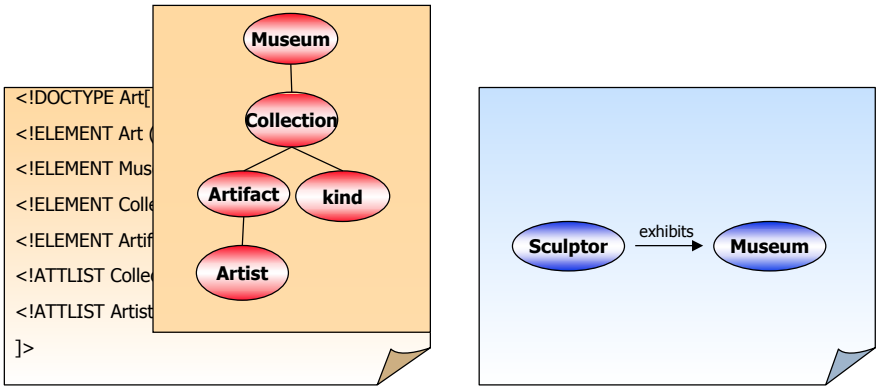
Expressiveness of SWIM Mappings



- Mapping an XML subpath to an RDF/S property

  P_EXT(x, exhibits, y):- {//Artist}(k), {./fName}(k, a), {./lName}(k, b), {./exhibits/Museum/denom}(k, y) | concat(x, a, b).

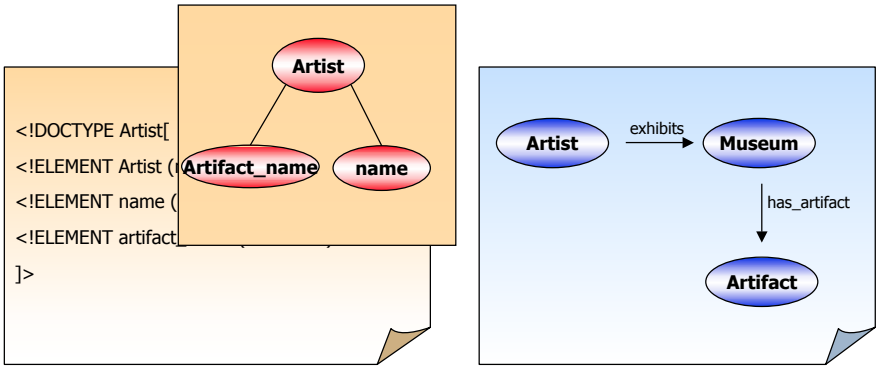
Expressiveness of SWIM Mappings



- Mapping an XML tree to an RDF/S property

  P_EXT(x, exhibits, y):- {//Museum}(a), {./Collection}(a,k), {./type}(k, s), {.//Artist/@id}(k, x), {./name}(a,y), s = "Sculptures".

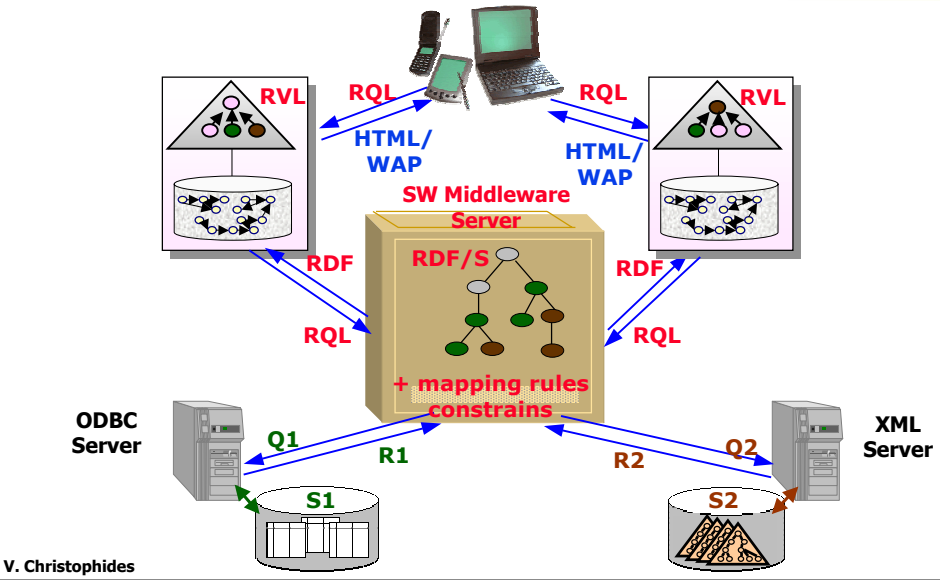
Expressiveness of SWIM Mappings



- Mapping an XML tree to an RDF/S path

  P_EXT(x, exhibits, y), P_EXT(y, has_artifact,z):- {//Artist}(k), {./name}(k,x), {./artifact_name}(k,y).

The ICS-FORTH SWIM Architecture



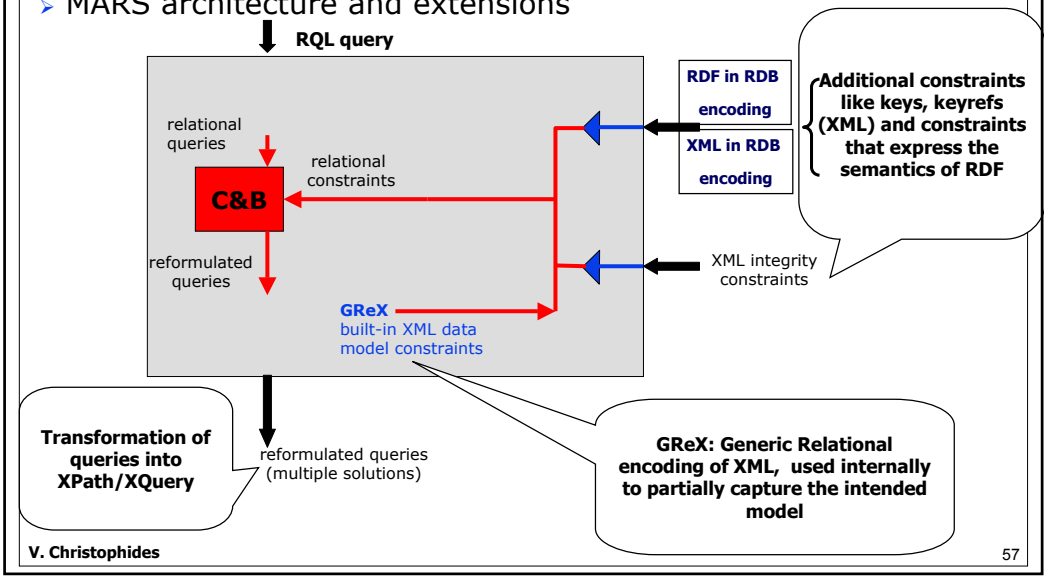
SWIM Implementation using MARS

- The MARS system (UPenn&UCSD) was originally designed for query optimization/reformulation over (object) relational databases:
  - Checking for query containment,
  - Checking query equivalence,
  - Minimizing queries
- While early approaches on reformulation considered
  - either materialized views or integrity constraints

  the Chase & Backchase algorithm implemented in MARS was the first that allowed proving completeness theorems considering both cases
- The key to its success was the ability to treat mappings between published & proprietary schemas defined using both the Global-As-View and Local-As-View integration approaches (GLAV)

SWIM Implementation using MARS

- MARS architecture and extensions



Related Work

- STYX [Fundulaki 03 INRIA]
  - Integrates XML sources through extended ER schemas
  - Maps ER entities and (binary) relationships to XPath expressions (only attribute and child axes)
  - LAV approach
- PEPSINT [Cruz 04 Univ. of Illinois]
  - Integrates XML sources through RDF/S schemas
  - XML DTDs are first converted to an RDF/S schema and then these source schemas are mapped to a global one (RDF/S to RDF/S mappings of concepts and properties)
  - GAV approach
- Integration of XML sources with OWL [Lethi 04 IPSI]
  - Integrates XML sources through OWL global schemas
  - Maps XML Schema constructs to OWL concepts and properties (OWL union/intersection of concepts and properties, equivalence of classes and properties, inverse properties)
  - GAV approach

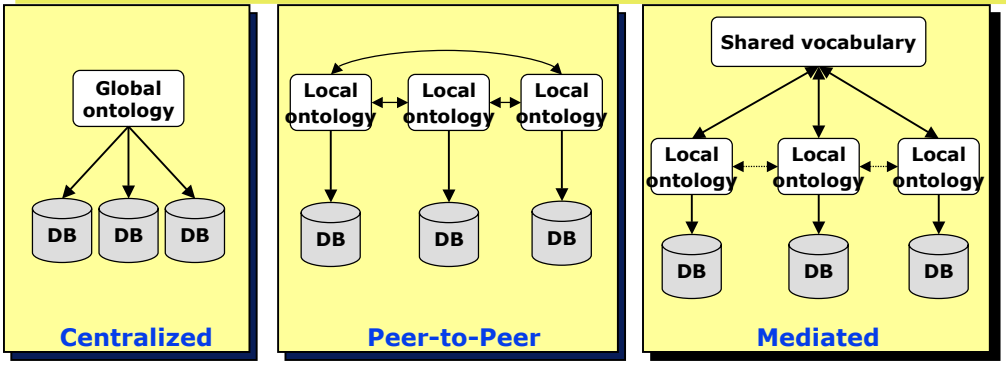
Summary and Future Work

- SWIM relies on state-of-the art relational database theory for SW data integration applications
  - XML and RDF/S data models
    - constructs are represented by first order predicates
    - semantics is captured by constraints (DED)
  - Sound and complete composition and reformulation of RQL queries (or RVL views) using XML→ RDF (or RDB → RDF) mappings
    - Chase&Backchase Algorithm
- Extend SWIM functionality with
  - More expressive SW ontology constraints (e.g., OWL inverse properties, cardinality constraints)
  - More expressive RQL queries (e.g., schema querying)
  - Generate entire translation programs for SW Data Warehouses

Three Ways to Overcome Semantic Heterogeneity

1. Standardization: agree on common user-defined schemas/ ontologies
   - great if no pre-existing applications
   - great if power player enforces it
2. Translation: create mappings among different schemas/ ontologies
   - requires human interpretation and machine reasoning
   - mappings can be difficult, expensive to establish
3. Annotation: create relationships to agreed upon conceptualizations
   - requires human interpretation and machine reasoning
   - annotation can be difficult, expensive to establish
   - reasoning over the conceptualization can provide added value

Ontology Integration vs…

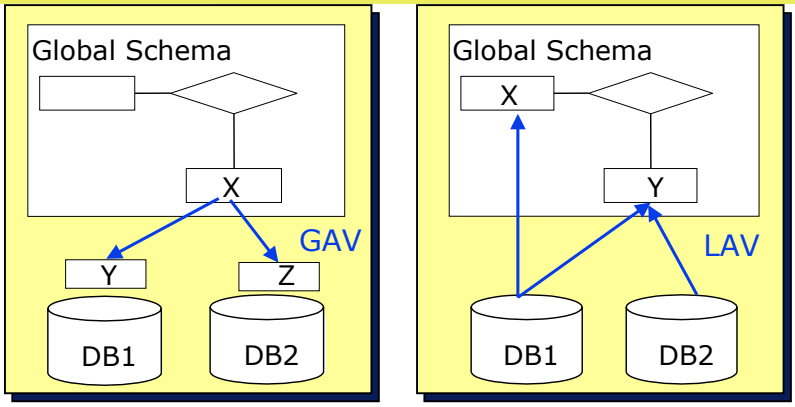


- Two mappings => Two translations
  - Mapping between ontology and the information they describe
  - Mapping between ontologies
- Semantic relationships (e.g., equivalence) to specify translation of concept & property definitions between ontologies

Comparing Ontology Integration Approaches

	Single Ontology Approaches	Multiple Ontology Approaches	Hybrid Ontology Approaches
Implementation effort	Straight-forward	Costly	Reasonable
Semantic heterogeneity	Similar view of a domain	Supports heterogeneous views	Supports heterogeneous views
Adding/ removing sources	Need for some adaptation in the global ontology	Providing a new source ontology; relating to other ontologies	Providing a new source ontology
Comparing multiple ontologies	--	Difficult because of the lack of a common vocabulary	Simple because ontologies use a common vocabulary

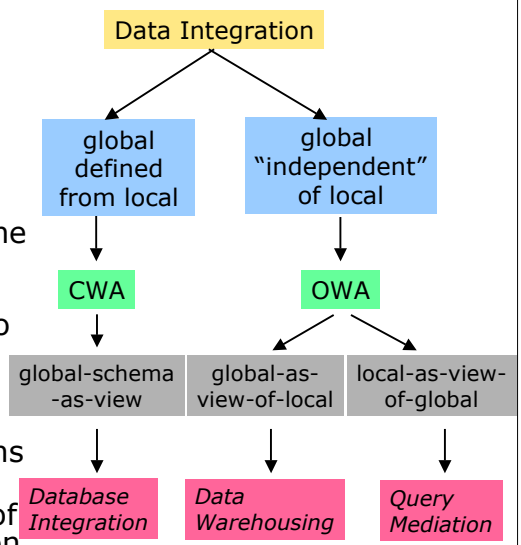
vs Data Integration



- One mapping => One translation
  - From local to global schema or vise versa
- Views (i.e., queries) to specify data translation from global to local schemas or vice versa

Comparing Data Integration Approaches

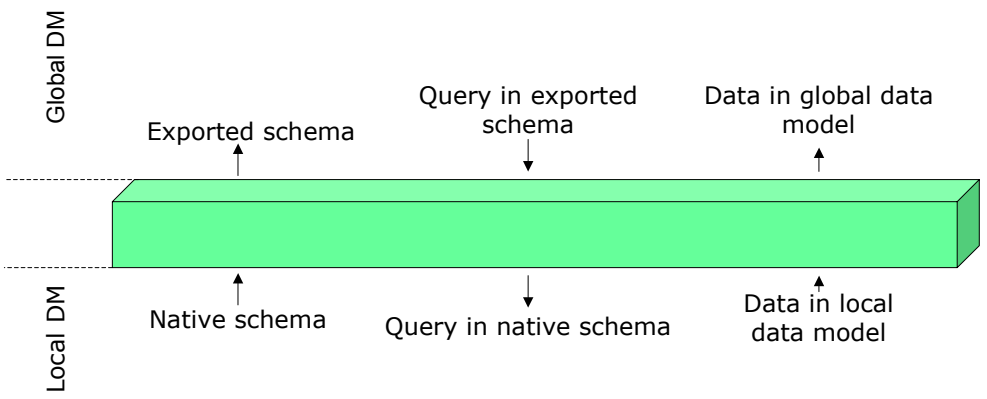
- Query answering
  - In GAV is simple (query unfolding, polynomial)
  - In LAV is more complex (answering queries with views NP-Complete)
- Source addition or removal
  - In GAV implies changes to the global schema specification
  - In LAV is simpler since new local source demands only to be described as view over the global schema
- Adding constraints to sources
  - LAV requires some extensions to the source descriptions
  - GAV requires modifications of the global schema description



GLAV

- In order to take advantage of both LAV and GAV, an other approach has been proposed: GLAV
- GLAV approach combines the expressive power of both LAV and GAV, allowing flexible schema definitions independent of the particular details of the sources
- In GLAV we can find both the previous expressions:
  $loc_i(X) \rightarrow gl_1(X_1), gl_2(X_2), \ldots, gl_k(X_k)$ and
  $loc_1(X_1), loc_2(X_2), \ldots, loc_k(X_k) \rightarrow gl_j(X)$ or even
  $loc_1(X_1), loc_2(X_2), \ldots, loc_k(X_k) \rightarrow gl_1(X_1), gl_2(X_2), \ldots, gl_m(X_m)$

Translations Between Different Data Models



Complexity of View-based Query Answering

User Query

Source Descriptions

Sound	CQ	CQ$^{\neq}$	PQ	Datalog	FOL
CQ	*PTIME*	*coNP*	*PTIME*	*PTIME*	*undec.*
CQ$^{\neq}$	*PTIME*	*coNP*	*PTIME*	*PTIME*	*undec.*
PQ	*coNP*	*coNP*	*coNP*	*coNP*	*undec.*
Datalog	*coNP*	*undec.*	*coNP*	*undec.*	*undec.*
FOL	*undec.*	*undec.*	*undec.*	*undec.*	*undec.*
Exact	**CQ**	**CQ$^{\neq}$**	**PQ**	**Datalog**	**FOL**
CQ	*coNP*	*coNP*	*coNP*	*coNP*	*undec.*
CQ$^{\neq}$	*coNP*	*coNP*	*coNP*	*coNP*	*undec.*
PQ	*coNP*	*coNP*	*coNP*	*coNP*	*undec.*
Datalog	*undec.*	*undec.*	*undec.*	*undec.*	*undec.*
FOL	*undec.*	*undec.*	*undec.*	*undec.*	*undec.*

© Serge Abiteboul, Oliver Duschka 1998

Complexity of View-based Query Answering: Regular Path Queries

domain	views	Complexity		
		data	expression	combined
closed	all sound	*coNP*	*coNP*	*coNP*
	all exact	*coNP*	*coNP*	*coNP*
	arbitrary	*coNP*	*coNP*	*coNP*
open	all sound	*coNP*	*PSPACE*	*PSPACE*
	all exact	*coNP*	*PSPACE*	*PSPACE*
	arbitrary	*coNP*	*PSPACE*	*PSPACE*

© Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Moshe Y. Vardi 2000