Ontology-based Data Management

Maurizio Lenzerini

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

SAPIENZA
UNIVERSITÀ DI ROMA

Part II

Seminars in Advanced Topics in Computer Science Engineering
April 27 - May 4, 2018

The course

- Part I
  - Ontology-based data management: The framework
  - Queries in OBDM
  - The nature of query answering in OBDM
- Part II
  - Ontology languages
  - Modeling the domain through the ontology
  - Modeling the mapping with the data sources
- Part III
  - Algorithms for query answering
  - Beyond classical first-order queries

Outline

1. Ontology languages
2. Modeling the domain through the ontology
3. Modeling the mapping with the data sources

Complexity of conjunctive query answering in DLs

	Combined complexity	Data complexity
Plain databases	NP-complete	in LogSpace (1)
OWL 2	?	coNP-hard (2)

(1) Going beyond probably means not scaling with the data.
(2) Already for a TBox with a single disjunction (see example above).

Questions

- Can we find interesting DLs for which the query answering problem can be solved efficiently (in LogSpace wrt data complexity)?
- If yes, can we leverage relational database technology for query answering in OBDM?

A very popular logic: *DL-Lite*$_{A,id}$

DL-Lite$_{A,id}$ is the most expressive logic in the *DL-Lite* family

Expressions in *DL-Lite*$_{A,id}$:

$$B \longrightarrow A \mid \exists Q \mid \delta(U) \qquad E \longrightarrow \rho(U) \qquad C \longrightarrow B \mid \neg B$$
$$Q \longrightarrow P \mid P^- \qquad V \longrightarrow U \mid \neg U \qquad R \longrightarrow Q \mid \neg Q$$
$$T \longrightarrow \top_D \mid T_1 \mid \cdots \mid T_n$$

Assertions in *DL-Lite*$_{A,id}$:

$B \sqsubseteq C$	(*concept inclusion*)	$E \sqsubseteq T$	(*value-domain inclusion*)
$Q \sqsubseteq R$	(*role inclusion*)	$U \sqsubseteq V$	(*attribute inclusion*)
$(id\ B\ \pi_1, ..., \pi_n)$	(*identification assertions*)	(**funct** Q)	(*role functionality*)
(**funct** U)	(*attribute functionality*)		

In identification and functional assertions, roles and attributes cannot specialized, and each π_i denotes a *path* (with at least one path with length 1), which is an expression built according to the following syntax rule:

$$\pi \longrightarrow S \mid B? \mid \pi_1 \circ \pi_2$$

SAPIENZA Università di Roma

Semantics of $\textit{DL-Lite}_{A,id}$

Construct	Syntax	Example	Semantics
atomic conc.	A	Doctor	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
exist. restr.	$\exists Q$	$\exists$child$^-$	$\{d \mid \exists e. (d,e) \in Q^{\mathcal{I}}\}$
at. conc. neg.	$\neg A$	$\neg$Doctor	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
conc. neg.	$\neg \exists Q$	$\neg\exists$child	$\Delta^{\mathcal{I}} \setminus (\exists Q)^{\mathcal{I}}$
atomic role	P	child	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
inverse role	P^-	child$^-$	$\{(o,o') \mid (o',o) \in P^{\mathcal{I}}\}$
role negation	$\neg Q$	$\neg$manages	$(\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) \setminus Q^{\mathcal{I}}$
conc. incl.	$B \sqsubseteq C$	Father $\sqsubseteq \exists$child	$B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
role incl.	$Q \sqsubseteq R$	hasFather $\sqsubseteq$ child$^-$	$Q^{\mathcal{I}} \subseteq R^{\mathcal{I}}$
funct. asser.	(**funct** Q)	(**funct** succ)	$\forall d,e,e'.(d,e) \in Q^{\mathcal{I}} \wedge (d,e') \in Q^{\mathcal{I}} \rightarrow e = e'$
mem. asser.	$A(c)$	Father(bob)	$c^{\mathcal{I}} \in A^{\mathcal{I}}$
mem. asser.	$P(c_1, c_2)$	child(bob, ann)	$(c_1^{\mathcal{I}}, c_2^{\mathcal{I}}) \in P^{\mathcal{I}}$

$\textit{DL-Lite}_{A,id}$ (as all DLs of the *DL-Lite* family) adopts the Unique Name Assumption (UNA), i.e., different individuals denote different objects.

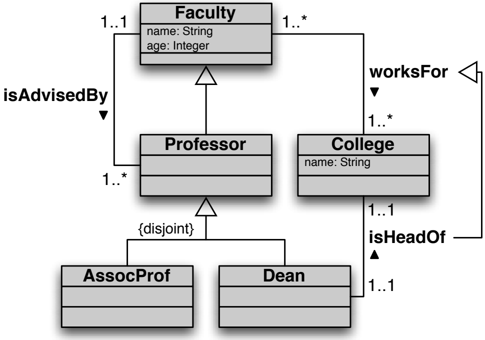
Capturing basic ontology constructs in *DL-Lite*$_{A,id}$

ISA between classes	$A_1 \sqsubseteq A_2$
Disjointness between classes	$A_1 \sqsubseteq \neg A_2$
Domain and range of properties	$\exists P \sqsubseteq A_1 \quad \exists P^- \sqsubseteq A_2$
Mandatory participation *(min card = 1)*	$A_1 \sqsubseteq \exists P \quad A_2 \sqsubseteq \exists P^-$
Functionality of relations *(max card = 1)*	$(\textbf{funct } P) \quad (\textbf{funct } P^-)$
ISA between properties	$Q_1 \sqsubseteq Q_2$
Disjointness between properties	$Q_1 \sqsubseteq \neg Q_2$

Note 1: *DL-Lite*$_{A,id}$ cannot capture completeness of a hierarchy. This would require disjunction (i.e., OR).

Note 2: *DL-Lite*$_{A,id}$ can be extended to capture also min cardinality constraints ($A \sqsubseteq \leq n\ Q$), max cardinality constraints ($A \sqsubseteq \geq n\ Q$) [Artale et al, JAIR 2009], n-ary relations, and denial assertions (not considered here for simplicity).

Example of *DL-Lite*$_{A,id}$ ontology



$$
\begin{array}{rcl}
\text{Professor} & \sqsubseteq & \text{Faculty} \\
\text{AssocProf} & \sqsubseteq & \text{Professor} \\
\text{Dean} & \sqsubseteq & \text{Professor} \\
\text{AssocProf} & \sqsubseteq & \neg\text{Dean} \\
\\
\text{Faculty} & \sqsubseteq & \exists \mathit{age} \\
\exists \mathit{age}^- & \sqsubseteq & \texttt{xsd:integer} \\
& (\textbf{funct}\ \mathit{age}) & \\
\\
\exists\text{worksFor} & \sqsubseteq & \text{Faculty} \\
\exists\text{worksFor}^- & \sqsubseteq & \text{College} \\
\text{Faculty} & \sqsubseteq & \exists\text{worksFor} \\
\text{College} & \sqsubseteq & \exists\text{worksFor}^- \\
\\
\exists\text{isHeadOf} & \sqsubseteq & \text{Dean} \\
\exists\text{isHeadOf}^- & \sqsubseteq & \text{College} \\
\text{Dean} & \sqsubseteq & \exists\text{isHeadOf} \\
\text{College} & \sqsubseteq & \exists\text{isHeadOf}^- \\
\text{isHeadOf} & \sqsubseteq & \text{worksFor} \\
& (\textbf{funct}\ \text{isHeadOf}) & \\
& (\textbf{funct}\ \text{isHeadOf}^-) & \\
& \vdots &
\end{array}
$$

Graphol

Graphol is a graphical language developed at Sapienza with the following key features:

- Looks similar to UML Class Diagrams and Entity-Relationship Diagrams
- Is a graphical counterpart of full OWL 2

Classes and Object Properties

- A class represents a set of objects (i.e., its instances) sharing common properties.
  - E.g., "Student", "Worker"

Graphol	OWL	Semantics
Student	`Class(Student)`	$\text{Student}^{\mathcal{I}} \subseteq \Delta_o^{\mathcal{I}}$

- An object property represents a binary relation between objects, i.e., a set of tuples (ordered pairs) of objects.
  - E.g., "enrolled" represents the set of tuples (pairs) such that the first component is the object that is *enrolled* nd the second component is the object in which it is enrolled.

Graphol	OWL	Semantics
enrolled	`ObjectProperty(enrolled)`	$\text{enrolled}^{\mathcal{I}} \subseteq \Delta_o^{\mathcal{I}} \times \Delta_o^{\mathcal{I}}$

Data Properties and Datatypes

- A data property represents a local property of a class, i.e., a property whose value depends only on the object itself and has no relationships with the other elements of the ontology.
  - E.g., "studentId" is a local property

Graphol	OWL	Semantics
studentId ○	`DataProperty(studentId)`	$\text{studentId}^{\mathcal{I}} \subseteq \Delta_o^{\mathcal{I}} \times \Delta_v$

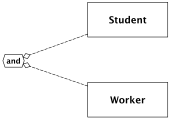
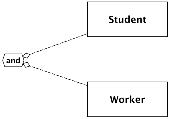
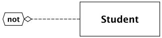
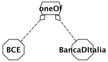
- A datatype represents a set of values (NOT objects!). Datatypes can be *predefined* ones in OWL or can be defined in the ontology itself (through `DatatypeDefinition` assertion).
  - E.g., "xsd:string", "xsd:interger", "rdfs:Literal" (predefined datatypes)
  - "StringOrInt"

Graphol	OWL	Semantics
xsd:string	(*predefined OWL datatype*)	$\texttt{xsd:string}^{\mathcal{I}} \subseteq \text{String}$
StringOrInt	`Datatype(StringOrInt)`	$\texttt{StringOrInt}^{\mathcal{I}} \subseteq \Delta_v^{\mathcal{I}}$

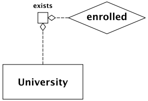
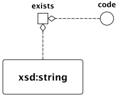
Complex Graphol expressions

- Starting from atomic expression by using OWL operators (e.g., `ObjectUnionOf`, `ObjectIntersectionOf`, `ObjectComplementOf`, etc.), we can build complex concept/role expressions.
- In Graphol each operator is identified by its name and characterized by the number and types of its input parameters
- In Graphol, to express that a (Graphol) expression is an input of an operator, we use a directed dashed edge ----------------◇ form the expression to the operator (the target is where the small diamond appears)

Complex Graphol Class expressions

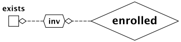
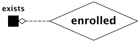
Graphol	OWL	Semantics
	`ClassUnionOf(Student Worker)`	$\mathrm{Student}^{\mathcal{I}} \cup \mathrm{Worker}^{\mathcal{I}}$
	`ClassIntersectionOf(Student Worker)`	$\mathrm{Student}^{\mathcal{I}} \cap \mathrm{Worker}^{\mathcal{I}}$
	`ObjectComplementOf(Student)`	$\Delta_o^{\mathcal{I}} \setminus \mathrm{Student}^{\mathcal{I}}$
	`ObjectOneOf(BCE BancaDItalia)`	$\{\mathrm{BCE}^{\mathcal{I}},\mathrm{BancaDItalia}^{\mathcal{I}}\}$

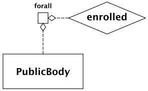
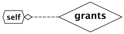
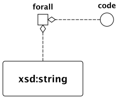
- Using the OWL operator `ObjectSomeValuesFrom` we can represent sets of object that form the domain of an object property e or data property

Graphol	OWL	Semantics
	`ObjectSomeValuesFrom(enrolled University)`	$\{e \mid \exists e'$ s.t. $(e, e') \in \text{enrolled}^{\mathcal{I}}, e' \in \text{University}^{\mathcal{I}}\}$
	`DataSomeValuesFrom(code rdfs:Literal)`	$\{e \mid \exists e' \in \Delta_v^{\mathcal{I}}$ s.t. $(e, e') \in \text{code}^{\mathcal{I}}\}$
	`DataSomeValuesFrom(code xsd:string)`	$\{e \mid \exists v$ s.t. $(e, v) \in \text{code}^{\mathcal{I}}$, v is a string$\}$

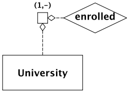
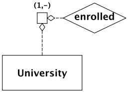
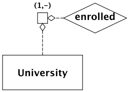
Class expression involving property range

- The range of ab object property is the domain of its inverse
  $\leadsto$ the range can be represented by combining `ObjectSomeValuesFrom` and `ObjectInverseOf`
- Graphol includes a convenient *shortcut* to denote the range

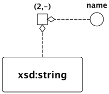
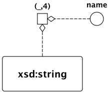
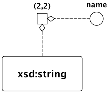
Graphol (1)	Graphol (2)	OWL	Semantics
		`ObjectSomeValuesFrom( ObjectInverseOf(enrolled) owl:Thing)`	$\{e \mid \exists e' \in \Delta_o^{\mathcal{I}}$ s.t. $(e', e) \in$ enrolled$^{\mathcal{I}}\}$

Graphol	OWL	Semantics
	`ObjectAllValuesFrom(enrolled Ente_pubblico)`	$\{e \mid \forall e'$ s.t. $(e, e') \in$ enrolled$^{\mathcal{I}} \rightarrow e' \in$ PublicBody$^{\mathcal{I}}\}$
	`ObjectHasSelf(grants)`	$\{e \mid (e, e) \in$ grants$^{\mathcal{I}}\}$
	`DataAllValuesFrom(code xsd:string)`	$\{e \mid \forall v$ s.t. $(e, v) \in$ code$^{\mathcal{I}} \rightarrow v$ is a string$\}$

Other class expressions (cont.)

Graphol	OWL	Semantics
	`ObjectMinCardinality(1 enrolled University)`	$\{e \mid$ s.t. $\mathsf{card}(\{e' \in \mathsf{University}^{\mathcal{I}} \mid (e,e') \in \mathsf{enrolled}^{\mathcal{I}}\}) \geq 1\}$
	`ObjectMaxCardinality(3 enrolled University)`	$\{e \mid$ s.t. $\mathsf{card}(\mid\{e' \in \mathsf{University}^{\mathcal{I}} \mid (e,e') \in \mathsf{enrolled}^{\mathcal{I}}\}) \leq 3\}$
	`ObjectExactCardinality(1 enrolled University)`	$\{e \mid$ s.t. $\mathsf{card}(\{e' \in \mathsf{University}^{\mathcal{I}} \mid (e,e') \in \mathsf{enrolled}^{\mathcal{I}}\}) = 1\}$

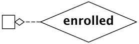
Other class expressions (cont.)

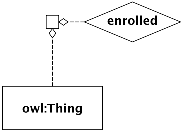
Graphol	OWL	Semantics
	`DataMinCardinality(2 name xsd:string)`	$\{e \mid$ s.t. card$(\{v$ string $\mid (e,v) \in \mathsf{name}^{\mathcal{I}}\}) \geq 2\}$
	`DataMaxCardinality(4 name xsd:string)`	$\{e \mid$ s.t. card$(\{v$ string $\mid (e,v) \in \mathsf{name}^{\mathcal{I}}\}) \leq 4\}$
	`DataExactCardinality(2 name xsd:string)`	$\{e \mid$ s.t. card$(\{v$ string $\mid (e,v) \in \mathsf{name}^{\mathcal{I}}\}) = 2\}$

Observation on the Graphol **exists** operator

- The label **exists** which represent *projection* on the domain or range can be omitted. Hence: 

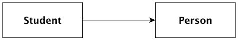
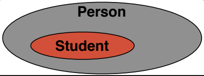
cab be written also as 
- When the operator **exists** gets as input either `owl:Thing` o `rdfs:Literal`, we typically omit it. Hence the following are equivalent:



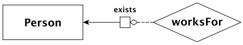
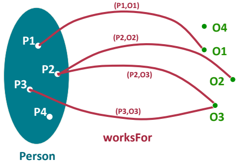
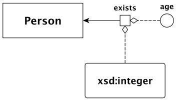
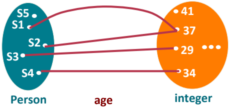


Inclusion assertions

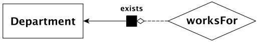
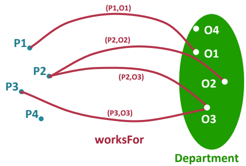
- In Graphol we represent classi inclusions (or ISA) assertions, by linking the subclass to the superclass with an oriented edge as follows:

Graphol	OWL	Semantics
	`SubClassOf(Student Person)`	

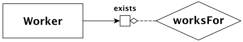
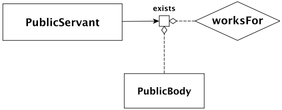
Inclusion assertions: object/data property domain typing

Graphol	OWL	Semantics
	`SubClassOf( ObjectSomeValuesFrom( worksFor owl:Thing) Person)`	
	`SubClassOf( ObjectSomeValuesFrom( age xsd:integer) Person)`	

Inclusion assertions: object/data property range typing

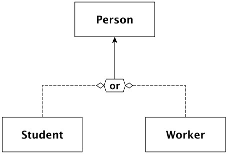
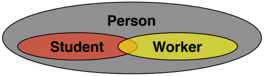
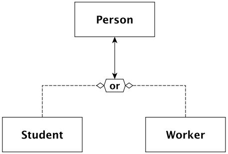
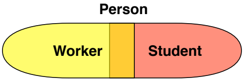
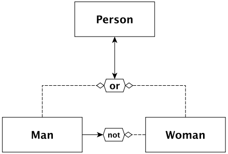
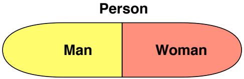
Graphol	OWL	Semantics
	`SubClassOf( ObjectSomeValuesFrom( ObjectInverseOf( worksFor) owl:Thing) Department)`	

Inclusion assertions: mandatory participation (unqualified/qualified)

Graphol	OWL
	SubClassOf(Worker ObjectSomeValues(worksFor owl:Thing))
	SubClassOf(PublicServant ObjectSomeValues(worksForPublicBody))

Generalizations (more advanced forms of ISAs)

Using inclusion assertions we can easily represent generalizations.

Graphol	OWL	Semantics
	SubClassOf(Student Person) SubClassOf(Worker Person)	
	SubClassOf(Studente Persona) SubClassOf(Worker Person) SubClassOf(Person ObjectUnionOf(Student Person))	
	SubClassOf(Man ObjectComplementOf(Woman))	

Outline

Two methodological aspects

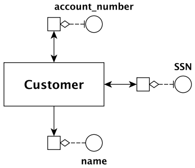
- The notion of role
- Modeling evolving properties of objects

The notion of role

- In many situations, we tend to identify the agent or actor with the role he/she play
  - e.g. Person vs. Customer
- In all these situations, properties characterizing in fact the actor are perceived as if they were characterizing the role
  - e.g., we refer to the name and the Social Security Number of a customer, while the latter are properties characterizing the persons who play the role of customers
- Several modeling options of the notion of role are possible
  $\leadsto$ we will investigate which is the most appropriate depending on the situation we need to model

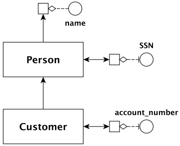
First option: using a single class

- associating the properties of the actor to the object representing the role



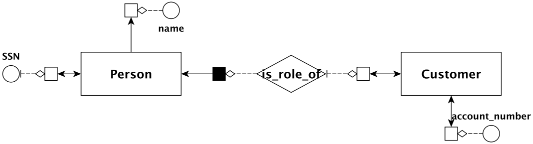
- Problem: How can we model properties that characterize actors who do not play the role that is modeled?
  - instances of Customer represent customers!
- When is this modeling pattern correct?
  - each time we do not need to predicate on actors who do not play the role we model

Second option: two classes connected by an ISA relation



- Problem: The bank account number typically identifies the customer within the bank.
  What happens if a person has two bank accounts?
  - on one side we would like to model that there cannot exist two persons with the same SSN, on the other side, this can happen if a customer has two bank accounts
- When is this modeling pattern correct?
  - each time the actor can play at most once the role we want to model

Third option: two classes connected by an object property

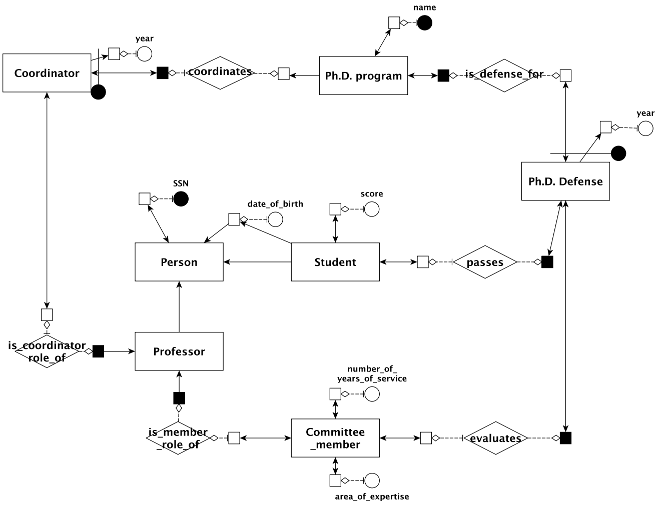


- This is the most general pattern modeling both actors and roles:
  - the actor can play several instances of the same role
  - we can predicate on both the actor and the role
- Problem: it is not possible to model in OWL that the SSN of a customer has to coincide with the SSN of the person playing the role of the customer

Example about modeling actors and roles

A university offers several Ph.D. programs, each one characterized by a name. For each of them, each year, we are interested in modeling: the professors that were members of the final Ph.D. defense (one per year) and the students that passed the defense, together with the score they obtained (we assume that there is only one defense per stufent). Each professor is identified by her SSN and has a date of birth. Each member of a Ph.D. committee is characterized by the number of years of service and the area of expertise. Each student that passed the defense is characterized by the SSN and the date of birth. Each Ph.D. programs is characterized by the professor who coordinated the Ph.D. in the various years.

The ontology



When does one need to model evolving aspects?

- In many situations we are interested in modeling objects and relations that evolve
  - e.g. we might be interested in the following properties of persons
    - the SSN and the biological parents - these are properties that do not evolve
    - the residential address and the conjugality - these are properties that evolve over time
- Important: the fact that a property evolves does not imply that we have to model its changes
  - e.g. we might be interested in modeling the changes of the residential address while we might be interested in modeling only the current conjugality

- By definition, the domain objects represented within an ontology are time-independent, in the sense that they do not change their identity over time
- It can happen that we are interested in modeling some properties of such objects that evolve over time, which we call evolving properties
  $\leadsto$ we resort to the notion of states of an object, representing a "snapshot" of a certain subset of its evolving properties during a certain period, called validity period
  $\leadsto$ a state is therefore characterized by the corresponding object, the set of properties it represents and the period of time it refers to

How to model object states

- Identify the evolving properties to be modeled
- Identify the temporal granularity needed to model the changes of each evolving property
  - e.g., the age of a person evolves every year (the day of her/his birthday)
  - e.g., the conjugality of a person might never change or change twice per year! (each time she/he gets divorced or married)

  $\rightsquigarrow$ Important: the temporal granularity depends on the occurrence of some event that triggers the change of state
- Choose the descriptive granularity we want to model through a state
  - e.g., the age and the conjugality may be represented by the same state which would change as soon as one event occurs which triggers the change either of the age or of the conjugality

$\Rightarrow$ Depending on the descriptive granularity, one or more classes are needed to model a snapshot of the object at any point of its life, each with its own validity period

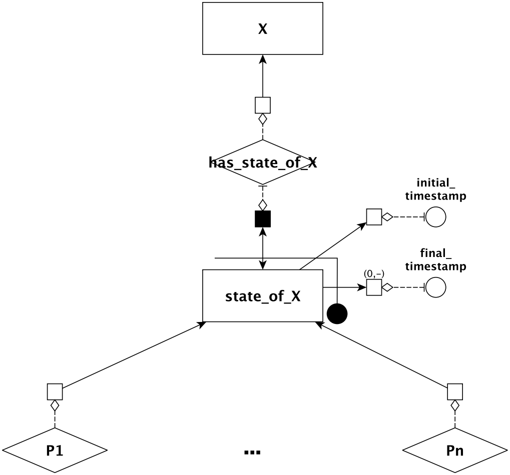
Descriptive granularity and states

Suppose that we are interested in the evolving properties $P_1, P_2, \ldots, P_n$ of the objects of a class `X`.

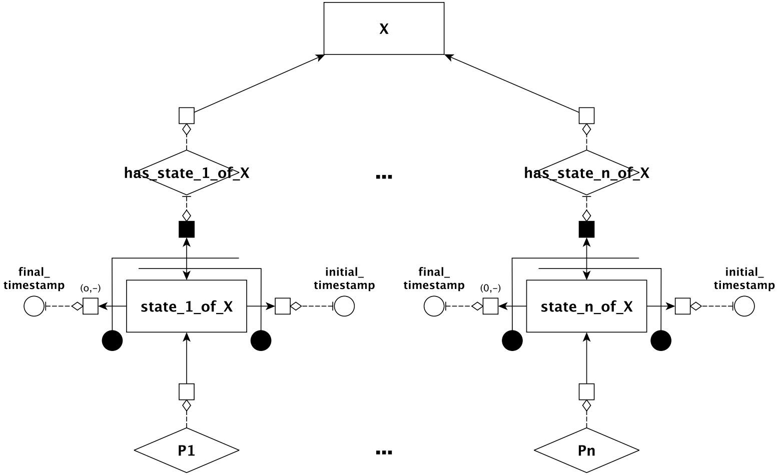
Several options are possible for the choice of the descriptive granularity. Two extremes:

- if we decide to model through a single state the values of all P_i's at any time t of the life of each instance o of `X`, a single class `State_of_X` is sufficient, since it represents the whole set of values of every P_i that characterizes o at t
- if we decide to model through different states the value of each P_i at any time t of the life of o, n classes `State_i_of_X` are necessary, for $i = 1, \ldots, n$, each representing the value of P_i at t.

A simple modeling pattern for evolving aspects



Another simple modeling pattern for evolving aspects



Example: modeling the domain of vessels carrying Petroleum

Suppose we are interested in modeling the following aspects of vessels

- their identity, i.e., the IMO (International Maritime Organization) number, the name, the current state of operation and the owner
- their movements, i.e., their spatial position

All above mentioned properties, but the IMO number, are properties that evolve, however, as for the state of operation, we are not interested in modeling its evolution but only the current state of operation

$\leadsto$ the evolving properties are the vessel name, owner and position

Also, as for the temporal granularity of each of them, for each vessel:

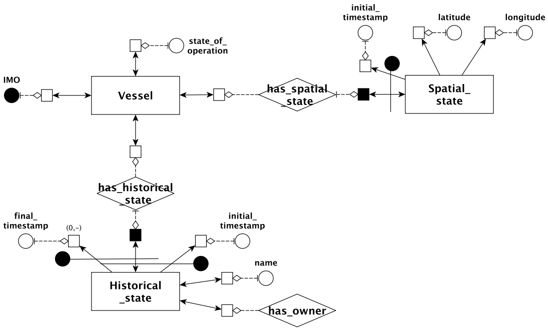
- the name and of the owner typically evolve with a low frequency
- the spatial position typically evolves every 3 minutes (since the GPS provider provides such information every 3 minutes)

Example: modeling the domain of vessels carrying Petroleum (Cont'd)

As for the descriptive granularity, given the temporal granularity described above, we choose to model the set of evolving properties of a vessel throw two classes representing the object states, such that at each time the snapshot of a vessel can be obtained by merging the instance of each class that valid at that time

- one class to represent the evolving properties name and owner
- one class to represent the position of a vessel

Example: modeling the domain of vessels carrying Petroleum (Cont'd)

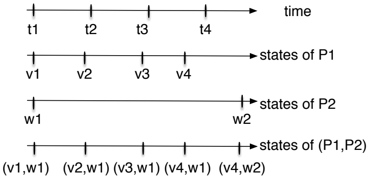


Guidelines for making the right choices to model evolving properties

- In order to choose the most appropriate descriptive granularity, we can adopt the following "guidelines":
  1. properties evolving at the same time should be represented by the same state (e.g., longitude and latitude)
  2. properties that are semantically related, and hence are often accessed together should be represented by the same state (e.g., a person address and telephone)
  3. the longer is the validity period of a state the better: hence, one should not represent by the same state properties that evolve with very different frequencies
- Important: While the first guideline can be always followed, the other guidelines might lead to different choices $\leadsto$ one has to face a trade-off to get the "best modeling"

SAPIENZA UNIVERSITÀ DI ROMA

Tradeoff between guidelines 1 and 2



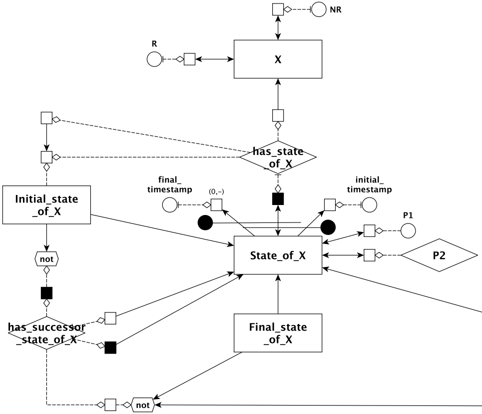
Pattern enrichment

In order to satisfy the information needs of users and simplify the queries over the ontology, the general pattern proposed can be enriched with the following elements and set of axioms:

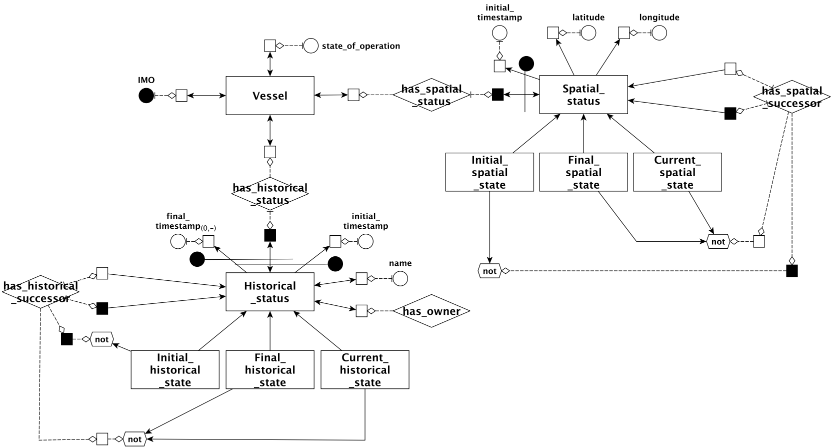
- The object property `has_successor_state_of_X`, connecting two consecutive states
  - the domain and range will be typed over `State_of_X`
  - `has_successor_state_of_X` and its inverse are both functional
- the class `Initial_state_of_X` whose instances are the states describing the first values of the evolving properties of each instance of X
  - `Initial_state_of_X` is disjoint from the set of states that have a successor
  - every object having a state must be connected to exactly one instance of `Initial_state_of_X`

Pattern enrichment (Cont'd)

- The class Final_state_of_X whose instances are the states describing the last values of the evolving properties of each instance of X, i.e. the values at the time an object stops evolving or being of interest
  - Final_state_of_X is disjoint from the states that have a successor
- The class Current_state_of_X whose instances are the states describing the current values of the evolving properties of each instance of X
  - Current_state_of_X is disjoint from the states that have a successor
  - Current_state_of_X is disjoint from the states that have a final timestamp
  - every object having a state must be connected to exactly one instance of Current_state_of_X


NR
R
X
has_state
_of_X
final_
timestamp
(0,-)
initial_
timestamp
final_
timestamp
Initial_state
_of_X
P1
State_of_X
P2
not
not
has_successor
_state_of_X
Final_state
_of_X
Current_state
_of_X
not

Example: modeling the domain of vessels carrying Petroleum (Cont'd)



Observations

The general pattern can be simplified in several ways and need to be adapted in every scenario, depending on the features of the domain itself as well as on the information needs of users

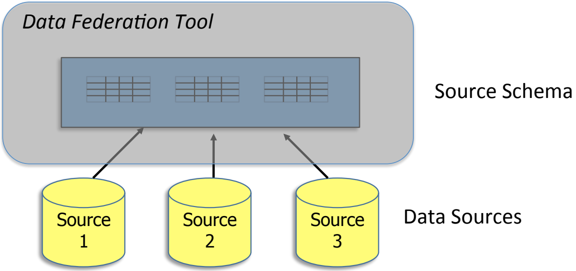
Examples of simplifications are the following:

- the classes `Initial_state_of_X` and `Final_state_of_X` might not be necessary
- the states might follow one another with no gaps, in which case the final timestamp might not be necessary
- the class representing the evolving objects `X` might represent as well the current values of some/all evolving properties
  - the class `Current_state_of_X`
  - the classes representing the states would then represent only states been passed

Outline

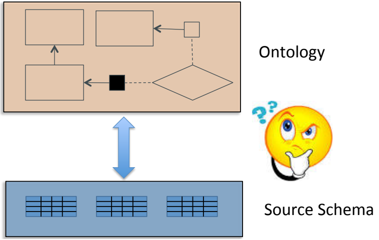
Data Sources

- In OBDA, data reside in autonomous data sources, typically pre-existing the ontology.
- Data sources are seen as a unique relational database that constitutes the Source component of an OBDA specification.
- Off-the-shelf Data Federation/Virtualization tools can be used to wrap multiple, possibly non-relational, sources, and present them as they were structured according to a single relational schema.



The problem

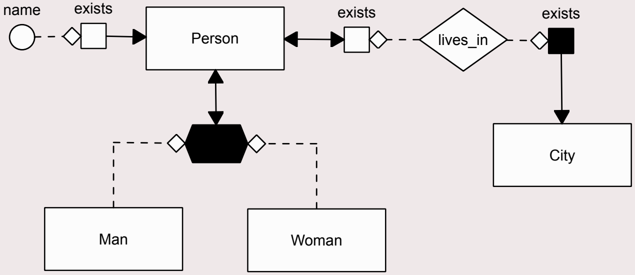
- Problem: How do we relate the ontology with the source schema?



- Main Design Challenges
  - Different representation languages, i.e., a DL TBox vs. a relational schema.
  - Different modeling: Data sources serve applications, and thus typically their structure does not directly reflect the abstract conceptualization given by the ontology,

Example

Ontology



Source Schema

RomanCitizens		
SSN	Name	Gender

Mapping relational schemas to ontologies: impedance mismatch

- Two different data models are used (relational databases vs. ontologies)
  - In relational databases, information is represented in forms of tuples of values.
  - In ontologies information is represented using both individuals (denoting objects of the domain) and values (as fillers of individuals's attributes) ...
- Solution: We need constructors to create individuals of the ontology out of tuples of values in the database.

  Note: from a formal point of view, such constructors can be simply Skolem functions!

Mapping in OBDA: formal definition

A Mapping in OBDA is a set of assertions having the following forms

$$\begin{array}{lcl} \Phi(\vec{x}) & \leadsto & C(\mathsf{f}(\vec{x})) \\ \Phi(\vec{x}) & \leadsto & R(\mathsf{f}_1(\vec{x}_1), \mathsf{f}_2(\vec{x}_2)) \\ \Phi(\vec{x}) & \leadsto & A(\mathsf{f}(\vec{x}_1), \vec{x}_2) \end{array}$$

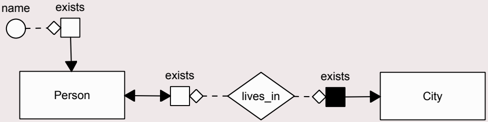
where:

- $\Phi(\vec{x})$ is an arbitrary SQL query over the source schema, returning attributes $\vec{x}$
- C is an atomic concept, R is atomic role, and A is an attribute
- f, f_1, f_2 are function symbols
- $\vec{x}_1$ and $\vec{x}_2$, possibly overlapping, contains only variables in $\vec{x}$

The left-hand side of a mapping assertion is called body, whereas the right-hand side is called head.

Example

Ontology



Source Data

RomanCitizens		
SSN	*Name*	*Gender*
1234	Marco	M
...	...	...

Mapping

`SELECT SSN FROM RomanCitizens` $\rightsquigarrow$ Person(**pers**(SSN))

`SELECT SSN, 'Rome' AS City FROM RomanCitizens` $\rightsquigarrow$ lives_in(**pers**(SSN),**ct**(City))

`SELECT SSN, Name FROM RomanCitizens` $\rightsquigarrow$ name(**pers**(SSN),Name)

Semantics

Def.: Semantics of mapping

Given an OBDA specification $\langle \mathcal{O}, \mathcal{S}, \mathcal{M} \rangle$, we say that a FOL interpretation $\mathcal{I}$ satisfies $\Phi(\vec{x}) \leadsto C(\mathsf{f}(\vec{x}))$ if

$$\forall \vec{t} \in \mathsf{eval}(\Phi(\vec{x}), \mathcal{S}),\ \mathcal{I} \models C(\mathsf{f}(\vec{t}))$$

Analogously for the other forms of mapping assertions.

$\mathcal{I}$ satisfies $\mathcal{M}$ wrt D if $\mathcal{I}$ satisfies all assertions in $\mathcal{M}$ wrt D.

Def.: Semantics of OBDA specification

$\mathcal{I}$ is a model of $\langle \mathcal{O}, \mathcal{S}, \mathcal{M} \rangle$ wrt D if:

- $\mathcal{I}$ is a model of $\mathcal{O}$
- $\mathcal{I}$ satisfies $\mathcal{M}$ wrt D

Semantics

Def.: Semantics of mapping

Given an OBDA specification $\langle \mathcal{O}, \mathcal{S}, \mathcal{M} \rangle$, we say that a FOL interpretation $\mathcal{I}$ satisfies $\Phi(\vec{x}) \leadsto C(\mathsf{f}(\vec{x}))$ if

$$\forall \vec{t} \in \mathsf{eval}(\Phi(\vec{x}), \mathcal{S}),\ \mathcal{I} \models C(\mathsf{f}(\vec{t}))$$

Analogously for the other forms of mapping assertions.

$\mathcal{I}$ satisfies $\mathcal{M}$ wrt D if $\mathcal{I}$ satisfies all assertions in $\mathcal{M}$ wrt D.

Def.: Semantics of OBDA specification

$\mathcal{I}$ is a model of $\langle \mathcal{O}, \mathcal{S}, \mathcal{M} \rangle$ wrt D if:

- $\mathcal{I}$ is a model of $\mathcal{O}$
- $\mathcal{I}$ satisfies $\mathcal{M}$ wrt D

"Generalized" GAV Mapping

- In OBDA we can even use mapping assertions that present a conjunction of atoms in their head and using as variables only those returned by the query in the body. This form of mapping is often called Generalized GAV.
- It is easy to see that a Generalized GAV mapping can be transformed into a GAV one (roughly, it is sufficient to "split" a mapping assertion with n atoms in the head into n mapping assertions with the same body and one single atom in the head)

Example

```
SELECT SSN, Name, 'Rome' AS City
FROM RomanCitizens
```

$\leadsto$ Person(**pers**(SSN)),
lives_in(**pers**(SSN),**ct**(City)),
name(**pers**(SSN),Name)

R2RML syntax for mappings

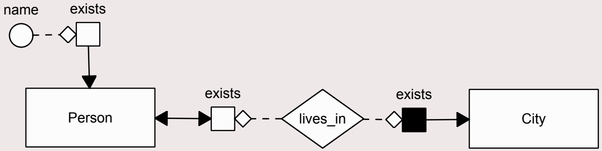
- The head of a mapping assertion is an RDF triple, where (object-)terms of the form f($\vec{x}$) are specified as IRI templates, i.e., format strings that reference names of variables in the SQL query by enclosing them in curly braces.

Example

Person(**pers**(SSN))	$\rightarrow$	:**pers**({SSN}) a :Person .
lives_in(**pers**(SSN),**ct**(City))	$\rightarrow$	:**pers**({SSN}) :lives_in :**ct**({City}) .
name(**pers**(SSN),Name)	$\rightarrow$	:**pers**({SSN}) :name {Name} .

Example – Ontology Rewriting

Ontology



User Query

Select $x
Where { $x a :Person }

Ontology Rewriting

Select $x
Where {
{ $x a :Person }
UNION
{ $x :lives_in $ndv1 }
UNION
{ $x :name $ndv2 }
}

Example – Mapping Rewriting

Mapping

```
SELECT SSN, Name, 'Rome' AS City
FROM RomanCitizens
```

$\rightsquigarrow$:**pers**({SSN}) :lives_in :**ct**({City}) .

Ontology Rewriting

```
Select $x
Where {
  { $x a :Person }
  UNION
  { $x :lives_in $ndv1 }
  UNION
  { $x :name $ndv2 }
}
```

Mapping Rewriting *(final rewriting)*

```
SELECT CONCACT(CONCAT('pers(',V.SSN),')')
FROM (SELECT SSN, Name, 'Rome' AS City
      FROM RomanCitizens) AS V
```

Further Example on Query Rewriting

Consider now a source schema with the relations `TabPers(SSN,Name)` and `TabRes(SSN,CityCode)`, and the following mapping

Mapping

`SELECT SSN, Name FROM TabPers`	⇝	:**pers**({SSN}) :name {Name} .
`SELECT SSN, CityCode FROM TabRes`	⇝	:**pers**({SSN}) :lives_in :**ct**({CityCode}) .

User Query

Select $x
Where {
  $y :name $x .
  $y :lives_in **ct**('RM')
}

Final rewriting

```
SELECT V1.Name
FROM (SELECT SSN,Name FROM TabPers) AS V1,
     (SELECT SSN,CityCode FROM TabRes) AS V2
WHERE V1.SSN=V2.SSN
      AND V2.CITYCODE='RM'
```

Some critical issues in mapping specification

We now discuss some main issues that a designer have to deal with when specifying mappings. The following list is definitely not complete but contains crucial aspects that necessarily need to be addressed.

- Define constructors, i.e., the functions used to construct individuals, which means deciding both the function symbols and their arguments.
- Select which ontology predicates to map.

In the following we assume to deal with *DL-Lite* ontologies.

Notice that *DL-Lite* adopts the UNA: different individuals denote different objects of the domain.

Principles on how to construct individuals

- An aspect a designer should take care is to avoid specifying mapping assertions in such a way that different domain objects are denoted by the same individual.

Example of wrong mapping

Let'us assume to have in the source schema two different tables representing disjoint sets of persons coming from different databases:

TabPers1		
ID	*SSN*	*Gender*
1	123	M
...	...	...

TabPers2		
ID	*SSN*	*Gender*
1	456	F
...	...	...

and the following mapping assertions

`SELECT ID FROM TabPers1` $\leadsto$:**pers**({ID}) a :Person .

`SELECT ID FROM TabPers2` $\leadsto$:**pers**({ID}) a :Person .

Principles on how to construct individuals

Possible solution 1 - use different constructors

TabPers1		
ID	*SSN*	*Gender*
1	123	M

TabPers2		
ID	*SSN*	*Gender*
1	456	F

`SELECT ID FROM TabPers1` $\leadsto$:**pers1**({ID}) a :Person .

`SELECT ID FROM TabPers2` $\leadsto$:**pers2**({ID}) a :Person .

Possible solution 2 - use a business identifier

TabPers1		
ID	*SSN*	*Gender*
1	123	M

TabPers2		
ID	*SSN*	*Gender*
1	456	F

`SELECT SSN FROM TabPers1` $\leadsto$:**pers**({SSN}) a :Person .

`SELECT SSN FROM TabPers2` $\leadsto$:**pers**({SSN}) a :Person .

Principles on how to construct individuals

- Since *DL-Lite* adopts the UNA, the mapping has also to guarantee that an object of the domain is always denoted with the same individual.

Example of wrong mapping

TabPers		
ID	*SSN*	*Occupation*
1	123	'stud'
...	...	...

`SELECT ID FROM TabPers WHERE Occupation='stud'` $\rightsquigarrow$:**stud**({ID}) a :Student .

`SELECT ID FROM TabPers` $\rightsquigarrow$:**pers**({ID}) a :Person .

Principles on how to construct individuals

Possible solution - use the same constructor

TabPers		
ID	*SSN*	*Occupation*
1	123	'stud'

`SELECT ID FROM TabPers WHERE Occupation='stud'` $\leadsto$:**pers**({ID}) a :Student .

`SELECT ID FROM TabPers` $\leadsto$:**pers**({ID}) a :Person .

Principles on how to construct individuals

- Generally speaking, constructing individuals from the values retrieved at the data sources is a very complex activity, for which no consolidated methods exists.
- Solving this task requires understanding how objects are identified in the domain, and finding out the identifier at the sources (e.g. for a person, her SSN).
- We have to guarantee that (i) different domain objects are not denoted with the same individual and also that (ii) a domain object is never denoted with different individuals (due to the UNA).
- data matching methods need often to be adopted to find out different representations of the same object [**?**].

SAPIENZA Università di Roma

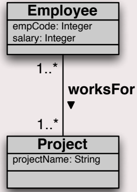
Principles on how to construct individuals

- Consider the simplified scenario in which for every object we can retrieve from the sources the values that identify it, and that such identification is uniform over all the source tables (e.g., a person is always identified by her SSN). We may proceed as follows:
  1. Let MGC (Most General Concepts) be the set of all ontology concepts that are subsumed only by owl:Thing (the Top concept all individuals are instance of);
  2. For each concept C in MGC, find out how instances of C are identified in the sources and define a constructor based on such identifier;
  3. Use the same constructor for all concepts subsumed by C
- Comments: (a) if there are equivalent concepts, put only a representative one in MGC (b) for objects that are instance of more than one concept in MGC, select one constructor among the possible one (typically, concepts in MGC are in fact all pair-wise disjoint) (c) define additional constructors for objects that are not instance of any concept in MGC (typically, there are no such objects, since MGC is a partition of owl:Thing).

Principles on how to construct individuals

Ontology $\mathcal{O}$ (TBox)

$\mathsf{Employee} \sqsubseteq \exists \mathsf{worksFor}$
$\mathsf{Employee} \sqsubseteq \exists \mathsf{empCode}$
$\mathsf{Employee} \sqsubseteq \exists \mathsf{salary}$
$\mathsf{Project} \sqsubseteq \exists \mathsf{worksFor}^-$
$\mathsf{Project} \sqsubseteq \exists \mathsf{projectName}$
$\exists \mathsf{worksFor} \sqsubseteq \mathsf{Employee}$
$\exists \mathsf{worksFor}^- \sqsubseteq \mathsf{Project}$



Federated schema of the DB $\mathcal{S}$

D_1[*SSN*: String, *PrName*: String]
Employees and Projects they work for

D_2[*Code*: String, *Salary*: Int]
Employee's Code with salary

D_3[*Code*: String, *SSN*: String]
Employee's Code with SSN

...

Mapping $\mathcal{M}$

M_1: `SELECT SSN,PrName FROM D1` $\leadsto$ V_1(SSN,PrName) $\leadsto$ Employee(**pers**(*SSN*)), Project(**proj**(*PrName*)), projectName(**proj**(*PrName*), *PrName*), workFor(**pers**(*SSN*), **proj**(*PrName*))

M_2: `SELECT SSN,Salary FROM D2, D3 WHERE D2.Code = D3.Code` $\leadsto$ V_2(SSN,Salary) $\leadsto$ Employee(**pers**(*SSN*)), salary(**pers**(*SSN*), *Salary*)

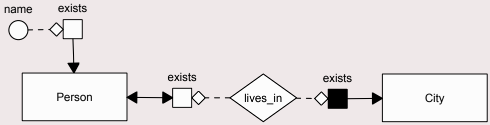
Select which ontology predicates to map

- Let us consider the (extreme) case where the ontology is empty, i.e., it has no axioms and thus it is simply a set of predicates.
- In this case we have to map every ontology predicate. Indeed, the ontology cannot infer new facts besides those directly constructed through the mapping.
- The most interesting case, however is the one of non-empty ontologies.
- In this case, we can exploit inclusions in the ontology to reduce the number of mapping assertions to write. Intuitively, we avoid to write assertions that are implied by the OBDA specification.
- Furthermore, we have to avoid writing mappings that are *intensionally inconsistent*, i.e., such there are no source instances for which the specification has a model (e.g., two disjoint concepts mapped to the same query, using the same constructor) [**?**].

Example

A role mapping assertions together with the typing of the role domain over a concept C implies a concept mapping assertion for C.

Ontology



Source Data

RomanCitizens		
SSN	*Name*	*Gender*
1234	Marco	M
...	...	...

Mapping

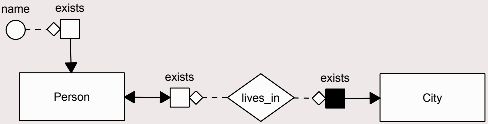
```
SELECT SSN                          ⇝ :pers({SSN}) a :Person
FROM RomanCitizens

SELECT SSN, 'Rome' AS City          ⇝ :pers({SSN}) :lives_in :ct({City})
FROM RomanCitizens

SELECT SSN, Name                    ⇝ :pers({SSN}) :Name {Name}
FROM RomanCitizens
```

Example

Ontology



Source Data

RomanCitizens		
SSN	*Name*	*Gender*
1234	Marco	M
...	...	...

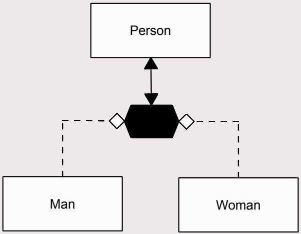
Mapping

`SELECT SSN, 'Rome' AS City` $\leadsto$:**pers**({SSN}) :lives_in :**ct**({City})
`FROM RomanCitizens`

`SELECT SSN, Name` $\leadsto$:**pers**({SSN}) :Name {Name}
`FROM RomanCitizens`

Example

Ontology



Source Data

RomanCitizens		
SSN	*Name*	*Gender*
1234	Marco	M
...	...	...

If we know that 'F' and 'M' are the only allowed values for Gender, and that it is not not nullable, we can avoid to write a mapping per Person.

Mapping

`SELECT SSN FROM RomanCitizens` $\leadsto$:**pers**(SSN) a :Man
`WHERE Gender='M'`

`SELECT SSN FROM RomanCitizens` $\leadsto$:**pers**(SSN) a :Woman
`WHERE Gender='F'`