Methods and Tools for Developing Ontology-Based Data Management Solutions

Ontology Modeling Patterns

Domenico Lembo, Valerio Santarelli, Domenico Fabio Savo

SEMANTICS 18
Vienna, Austria, September 11, 2018
Overview

1. Introduction

2. Modeling the notion of role

3. Modeling characteristics that may change over time
Ontology patterns

- Ontology languages provide the means to describe the domain of interest at a very high level of accuracy.
  - Ontologies may become large and complex.

- Ontology modeling (or design) patterns can be used to model situations that may recur frequently in several domains (see, e.g., [BS05, Gan05, HGJ+16, ACC+14], or the web site http://ontologydesignpatterns.org).
  - An ontology pattern is typically very rich and general (to meet the needs of every domain).
  - Ontology patterns need to be adapted to the specific domain.
  - The use of ontology patterns promotes a modular approach to modeling which can be extremely effective in practice.
  - It has to be formalized in a standard language (i.e., OWL!)
Overview

1. Introduction

2. Modeling the notion of role

3. Modeling characteristics that may change over time
The notion of role

- In many situations, we tend to identify the agent with the role he/she is playing
  - e.g. Person vs. Director (of a department)
- In such situations, properties characterizing the agent might be perceived as if they were characterizing the role
  - e.g., we refer to the name and the Social Security Number of a director, while the latter are properties characterizing the person who plays the role of director
  - We thus do not distinguish between the above properties and the properties the agent may have only because he/she is playing a role (e.g., the year in which he/she has been elected, or the department he/she directs)
- This identification of the agent with the role he/she is playing may be adequate in some simple scenarios, but in general the situation is far more complex.
- Several modeling options of the notion of role are possible
  - we will investigate which is the most appropriate depending on the situation we need to model
First option: using a single class

- We use a class to represent the role and define all properties characterizing either the agent or the role as properties of this class (i.e., such properties are typed on this class). That is, the class representing the agent is not represented and “collapses” into the class representing the role. Thus we do not distinguish between the properties of the agent and the properties of the role.

To ask for the SSN and the election year of directors, we use the query

```
Select ?d ?SSN ?elecYear Where {
  ?d a Director.
  ?d SSN ?SSN.
  ?d electionYear ?elecYear.
}```
First option: using a single class

- We use a class to represent the role and define all properties characterizing either the agent or the role as properties of this class (i.e., such properties are typed on this class). That is, the class representing the agent is not represented and “collapses” into the class representing the role. Thus we do not distinguish between the properties of the agent and the properties of the role.

**Problem:** How can we model properties that characterize agents who do not play the role that is modeled through the class?

**When is this modeling pattern correct?**
- each time we do not need to predicate on agents who do not play the role we model.
Second option: two classes connected by an ISA relation

- we use two distinct classes to represent the role and the agent, and define the former as a subclass of the latter

![Diagram of classes and relationships]

- To ask for SSN and election year of directors we can still use the query of the previous example

- Problem: What happens if an (instance of the) agent plays the “same” role more than once?
  - If the same person directs, e.g., two departments, he/she should be denoted by two distinct instances $d_1, d_2$ of the class Director, but then $d_1, d_2$ would be also two distinct instances of the class Person!

- When is this modeling pattern correct?
  - each time the agent can play at most once the role
we use two distinct classes to represent the role and the agent, and connect them through an object property that, to every instance of the class representing the role, associates exactly an instance of the class representing the agent, i.e., plays_role is inverse functional and Director has a mandatory participation in its range (similar to [BHJ+16])
According to this pattern:
- the agent can play several instances of the “same” role
- we can predicate on both the agent and the role she/he is playing
- However we lose the inheritance between the agent and the role, thus, for instance, a director no longer has a SSN: in fact, the SSN of a director is the SSN of the person playing that director role.
Example of Instances:

**DL syntax**
- Person(John)
- Director(JohnDir)
- plays_role(John,JohnDir)
- directs(JohnDir,CSDep)
- name(John, “John”),
- electionYear(JohnDir,2015)

**OWL functional syntax (no IRI, for simplicity)**
- ClassAssertion(Person John)
- ClassAssertion(Director JohnDir)
- ObjectPropertyAssertion(plays_role John JohnDir)
- ObjectPropertyAssertion(directs JohnDir CSDep)
- DataPropertyAssertion(name John “John”)
- DataPropertyAssertion(electionYear JohnDir 2015)
To ask for the SSN and the election year of directors, we use the query

```
Select ?d ?SSN ?elecYear Where {
?p a Person.
?p SSN ?SSN.
?p plays_role ?d
?d a Director.
?d electionYear ?elecYear.}
```
Third option: two classes connected by an object property

Problem: How can we model properties that characterize the role without an agent playing it? E.g., how can we specify that the email of the director of the computer science department at Sapienza is CS_DeptDirector@uniroma1.it?
Fourth option: three classes and two object properties

- we use a class to represent the agent, a class to represent the agent playing the role, and a class to represent the role played by the agent (Note: in our example Director has so far represented the agent playing the role; now it represents the role itself, whereas Directing_person represents the agent playing the role).

![Diagram](image)
Fourth option: three classes and two object properties

Example of Instances:
Person(John)
Directing_person(JohnDir)
plays_role(John, JohnDir)
Director(CSDepDir)
has_directing_person(CSDepDir, JohnDir)
directs(CSDepDir, CSDep)
name(John, “John”),
electionYear(JohnDir, 2015)
email(CSDepDir, CSDepDir@myUniv.org)
Query that asks for the SSN, the election year and the email of directors:

Select ?d ?SSN ?elecYear ?email Where {
  ?p a Person.
  ?p SSN ?SSN.
  ?p plays_role ?dp
  ?dp a Directing_person.
  ?dp electionYear ?elecYear.
  ?d has_directing_person ?dp.
  ?d a Director.
  ?d email ?email}
Fourth option: three classes and two object properties

Below we give the general (skeleton) ontology for this pattern: 

```
Playing_agent --|exists|---|--plays_role|--exists|---|Agent

|exists|---|has_playing_agent|

|exists|---|Role|
```
Overview

1 Introduction

2 Modeling the notion of role

3 Modeling characteristics that may change over time
Modeling characteristics that may change over time

- In many situations we are interested in modeling characteristics that may change over time
  - e.g. we might be interested in the following properties of persons
    - the year and city of birth – these are properties that do not change
    - the residential address and the conjugality – these are properties that may 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
    - From now on, we call **evolving properties** those properties of individuals that evolve and whose changes are of interest within the domain

We aim at modeling evolving properties in a standard ontology language like OWL2, and more precisely in an OWL2 fragment that is suited for OBDA, like OWL2QL or DL-Lite. Thus we do not resort to temporal logic formalisms (see, e.g., [AKK⁺ 17, AKRZ10]), that allow to model even more complicated scenarios involving temporal data, but go beyond the expressiveness of OWL
Modeling states of individuals

- In order to model evolving properties we resort to the notion of **state of an individual**, relative to a certain (sub)set of its evolving properties.

- A state gathers together such set of properties and is characterized by a **validity period**, i.e., a portion of time during which such properties do not change (or for which we do not record the changes). In other terms, a state represents a “snapshot” (valid for a certain amount of time) of the individual’s evolving properties that the state is modeling.

  - A state is therefore characterized by a set of properties and the period of time it refers to, i.e., a state has a starting and (optionally) an ending point, which (depending on the specific modeling needs) can be dates (e.g., `xsd:dates`), timestamps (e.g., `xsd:dateTime`), etc.

**Remark:** our aim is to model evolving properties, and not to represent the time (e.g., how to deal with time encodings or temporal ordering relationships). For these latter aspects we refer, for instance, to the OWL-time ontology (https://www.w3.org/TR/owl-time/), whose temporal concepts can be smoothly imported in our pattern.
Modeling options

Suppose that for the class $C$ we are interested in the evolving properties $p_1, p_2, \ldots, p_n$. Several options are possible for modeling $p_1, p_2, \ldots, p_n$, which differ for the number of “state types” one wants to adopt. The two extreme cases are:

1. **Use one single class**, called, e.g., $\text{State\_of\_C}$, that denote all states of each instance of the class $C$. In this case, $p_1, p_2, \ldots, p_n$ are all typed on the class $\text{State\_of\_C}$, and each change of a property gives rise to a new instance of $\text{State\_of\_C}$
   
   **Pros**: By accessing an instance of $\text{State\_of\_C}$ we get all properties and their value at a certain point in time
   
   **Cons**: Every time a property changes, and thus a new state is generated, it is necessary to instantiate also all the other properties, even if they did not change

2. **Use one state class**, called, e.g., $\text{State\_i\_of\_C}$, for each property $p_i$ to denote the state of instances of $C$ relative only to the changes of the property $p_i$
A simple modeling pattern for evolving aspects

Notice that the data property to is not mandatory, since current states do not have a final timestamp

Example of Instances

Person(John) has_state(John,JoS1) height(John,175)
has_state(John,JoS2) height(John,175)
has_state(John,JoS3) height(John,175)
dateOfBirth(John, 13/11/1987) from(JoS1,13/11/2010) weight(John,70)
from(JoS2,13/11/2011) weight(John,75)
from(JoS3,13/11/2012) weight(John,80)
to(John,13/11/2011) to(John,13/11/2012) to(John,13/11/2013)
A simple modeling pattern for evolving aspects

![Diagram showing John's changes over time with dates for weight and height]

- **13/11/10**:
  - `JoS1` has `has_state` with `weight: 70`
  - `dateOfBirth: 13/11/1987`

- **13/11/11**:
  - `JoS2` has `has_state` with `weight: 75`

- **13/11/12**:
  - `JoS3` has `has_state` with `weight: 80`

- **13/11/13**:
  - `height: 175`

---

Domenico Lembo

MT4OBDM – Ontology Patterns

SEMANȚICS 2018
A simple modeling pattern for evolving aspects

Give me height and weight of persons at 12/7/2011 (only terminated states)

Select ?p ?h ?w where {
?p a Person.
?p hasState ?s.
?s from ?t0.
?s to ?t1.
?s height ?h.
?s weight ?w.
FILTER(?t0 <=’12/7/2011’^^xsd:date & & ?t1 >’12/7/2011’^^xsd:date) }
Several possible modeling options

Suppose that for the class \( C \) we are interested in the evolving properties \( p_1, p_2, \ldots, p_n \). Several options are possible for modeling \( p_1, p_2, \ldots, p_n \), which differ for the number “state types” we want to adopt. The two extreme cases are:

- **Use one single class**, called, e.g., State_of_\( C \), to denote all states of each instance of the class \( C \). In this case, \( p_1, p_2, \ldots, p_n \) are all typed on the class State_of_\( C \), and each time a property changes, a new instance of State_of_\( C \) has to be considered.

- **Use one state class**, called, e.g., State_of_\( C \) for \( p_i \), for each property \( p_i \) to denote the state of instances of \( C \) relative only to the changes of the property \( p_i \).

**Pros:** No need to replicate in a state properties that do not change.

**Cons:** To get the value of all properties at a certain point in time \( t \) we need to access all the states that are valid at \( t \).
Another simple modeling pattern for evolving aspects

Example of Instances:
Person(John)
has_state(John, JoSW1)
weight( JoSW1, 70)
has_state(John, JoSW2)
weight( JoSW2, 75)
has_state(John, JoSW3)
weight( JoSW3, 80)
has_state(John, JoSH)
height( JoSH, 175)
Another simple modeling pattern for evolving aspects

John

JoSW1

JoSW2

JoSW3

JoSH

John

dateOfBirth: 13/11/1987

weight: 70

weight: 75

weight: 80

13/11/10

13/11/11

13/11/12

13/11/13

has_state

height: 175

has_state

has_state

Domenico Lembo

MT4OBDM – Ontology Patterns

SEMANTICS 2018
Another simple modeling pattern for evolving aspects

Give me height and weight of persons at 12/7/2011 (only terminated states)

Select ?p ?h ?w where {
  ?p a Person.
  ?p hasState ?s1.
  ?p hasState ?s2.
  ?s1 from ?t0.
  ?s1 to ?t1.
  ?s1 height ?h.
  ?s2 from ?d0.
  ?s2 to ?d1.
  ?s2 weight ?w.
  FILTER(?t0 <='12/7/2011'^^xsd:date && ?t1 >'12/7/2011'^^xsd:date
     && ?d0 <='12/7/2011'^^xsd:date && ?d1 >'12/7/2011'^^xsd:date)
Example: the ship domain

Suppose we are interested in modeling the following aspects of ships:

- their identity, i.e., the IMO (International Maritime Organization) number, the name, the type (cargo, ferry, etc.), and the owner;
- their movements, i.e., their spatial position.

All the above mentioned properties, except the IMO number, evolve, however we are not interested in modeling the evolution of the type, but want to consider only the current type.  

\( \Rightarrow \) the evolving properties are the ship name, owner and position.

Also, suppose that, for each ship:

- the name and 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: The ship domain
Guidelines to model evolving properties

In order to choose the most appropriate modeling pattern, we can adopt the following “guidelines”:

1. properties that change exactly at the same time should be represented by the same state class (e.g., longitude and latitude)
2. properties that are semantically related, and hence are often accessed together should be represented by the same state class (e.g., person email and telephone number)
3. Do not put too much properties on the same state class: properties that vary with different frequencies may produce a lot of states that replicate properties having the same value of the previous state (as for the height-weight example)

Important: one has to find the right trade-off between the various criteria to get the “best modeling”
In order to satisfy the information needs of users and simplify the queries over the ontology, the patterns for evolving properties can be enriched with the following elements and set of axioms:

- The object property `has_successor_state_of_C`, connecting two consecutive states
  - the domain and range will be typed over `State_of_C`
  - `has_successor_state_of_C` and its inverse are both functional

- The class `Initial_state_of_C` whose instances are the states describing the first values of the evolving properties of each instance of `C`
  - `Initial_state_of_C` is disjoint from the set of states that are successors of other states
  - every object having a state must be connected to exactly one instance of `Initial_state_of_C`
The class `Current_state_of_C` whose instances are the states describing the current values of the evolving properties of each instance of C

- `Current_state_of_C` is disjoint from the states that have a successor
- `Current_state_of_C` is disjoint from the states that have a final timestamp (i.e., the data properties cannot be instantiated on current states)
- every object having a state must be connected to exactly one instance of `Current_state_of_C`
A general modeling pattern for evolving aspects

exists (0,1)
exists
not
and
exists
State_of_C
from exists
to exists
has_successor_state
not
exists
Current_state_of_C
Initial_state_of_C
exists
exists
has_state
C
(0,1)
exists
Current_state_of_C
Initial_state_of_C
State_of_C
C
has_state
(0,1)
[ACC\textsuperscript{+}14] Natalia Antonioli, Francesco Castanò, Spartaco Coletta, Stefano Grossi, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Emanuela Virardi, and Patrizia Castracane.

Ontology-based data management for the italian public debt.


Ontology-Mediated Query Answering over Temporal Data: A Survey (Invited Talk).


Past and future of dl-lite.

In *Proc. of the 24th AAAI Conf. on Artificial Intelligence (AAAI 2010)*, 2010.
Considerations regarding ontology design patterns. 

[BS05] Eva Blomqvist and Kurt Sandkuhl.
Patterns in ontology engineering: Classification of ontology patterns.

Ontology design patterns for semantic web content.