Methods and Tools for Developing Ontology-Based Data Management Solutions

Ontology Modeling Patterns

Domenico Lembo, Valerio Santarelli, Domenico Fabio Savo

SAPIENZA
UNIVERSITÀ DI ROMA

SEMANTICS 18
Vienna, Austria, September 11, 2018

Overview

Ontology patterns

- Ontology languages provide the means to describe the domain of interest at a very high level of accuracy
  $\rightsquigarrow$ 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!)

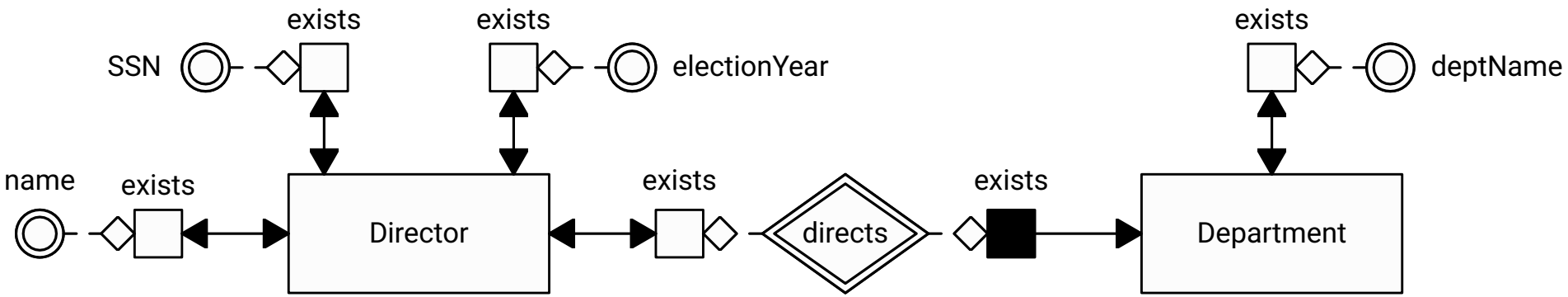
Overview

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
  $\rightsquigarrow$ 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.

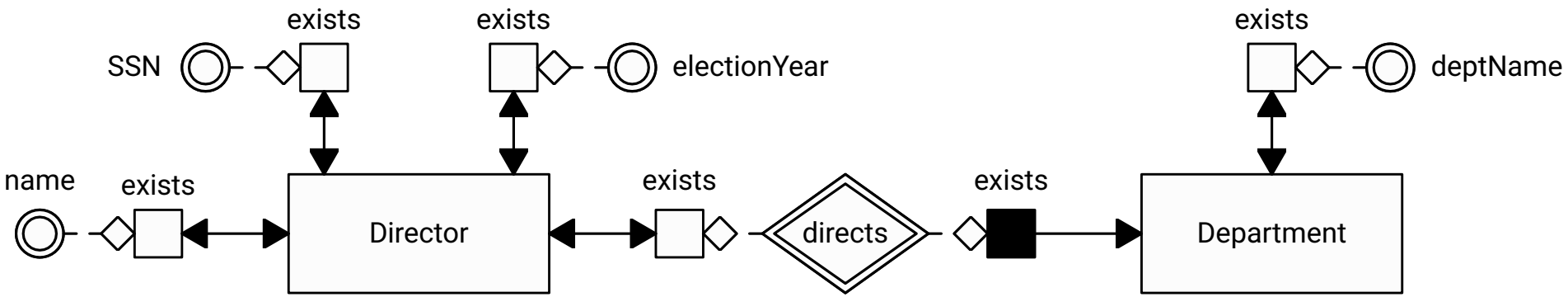


To ask for the SSN and the election year of directors, we use the query

```
Select ?d ?SSN ?elecYear Where {
?d a Director.
?d SSN ?SSN.
?d electionYear ?elecYear.}
```

First option: using a single class

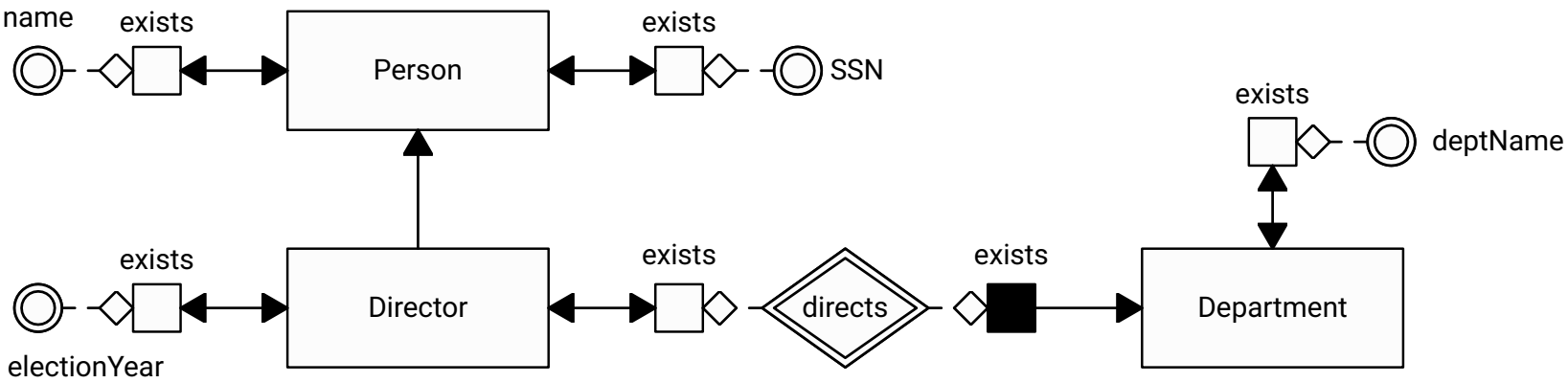
- We use a class to represent the role and define all properties characterizing either the agent or the role as properties of this class (i.e., such properties are typed on this class). That is, the class representing the agent is not represented and "collapses" into the class representing the role.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

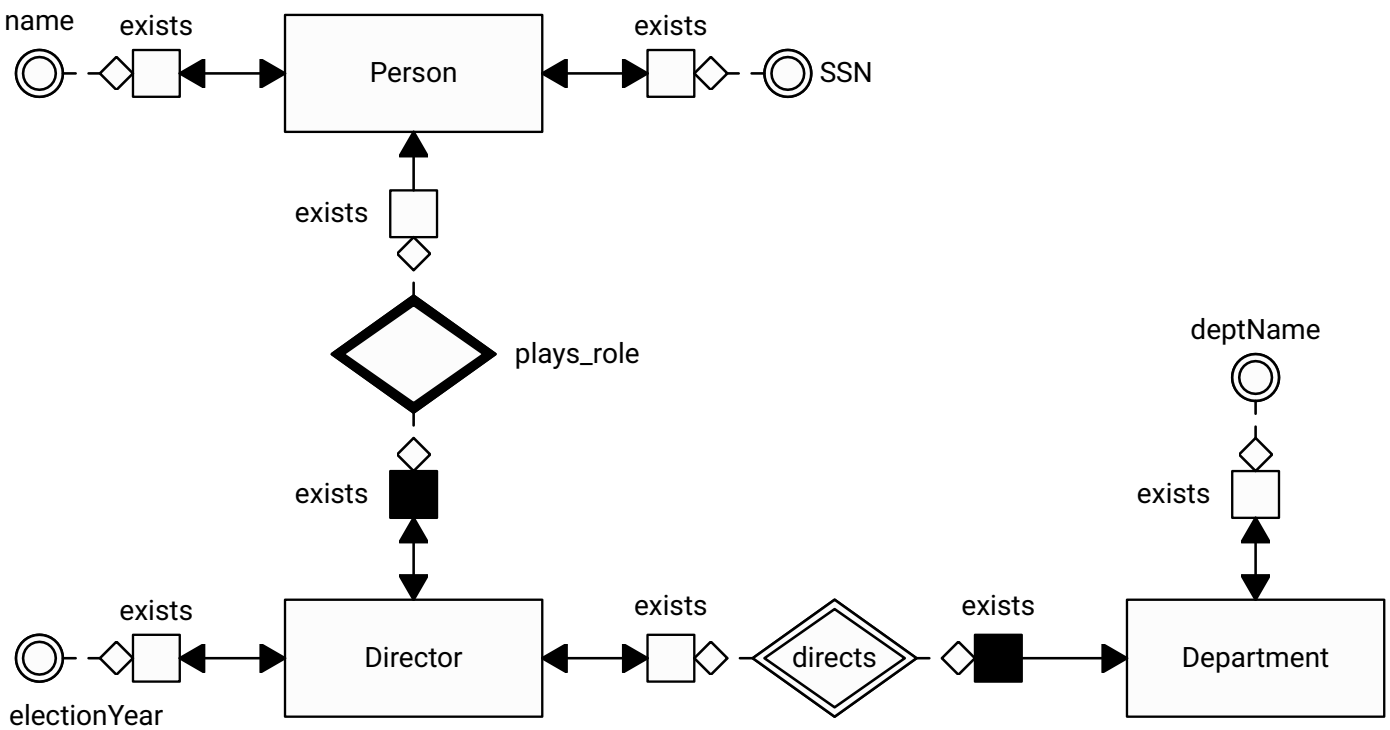
- we use two distinct classes to represent the role and the agent, and define the former as a subclass of the latter



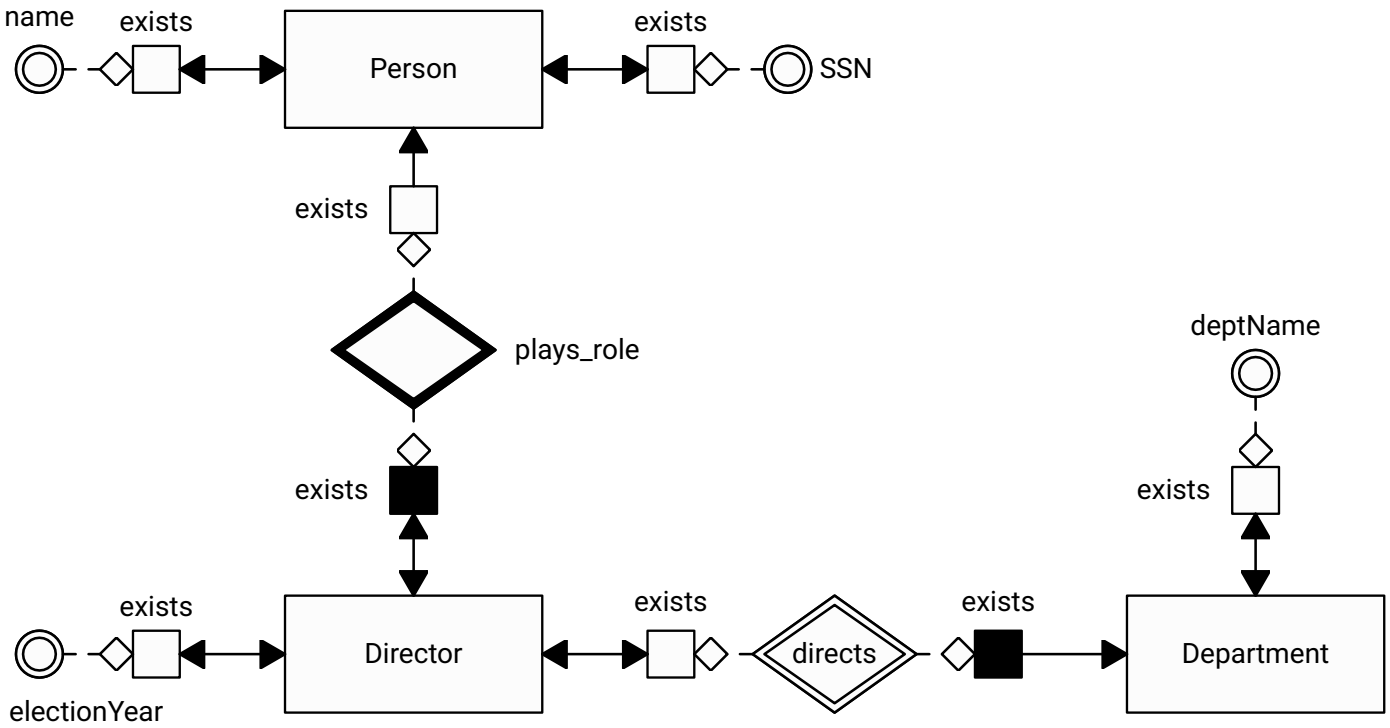
- To ask for SSN and election year of directors we can still use the query of the previous example
- Problem: What happens if an (instance of the) agent plays the "same" role more than once?
  - If the same person directs, e.g., two departments, he/she should be denoted by two distinct instances d_1, 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

Third option: two classes connected by an object property

- 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])

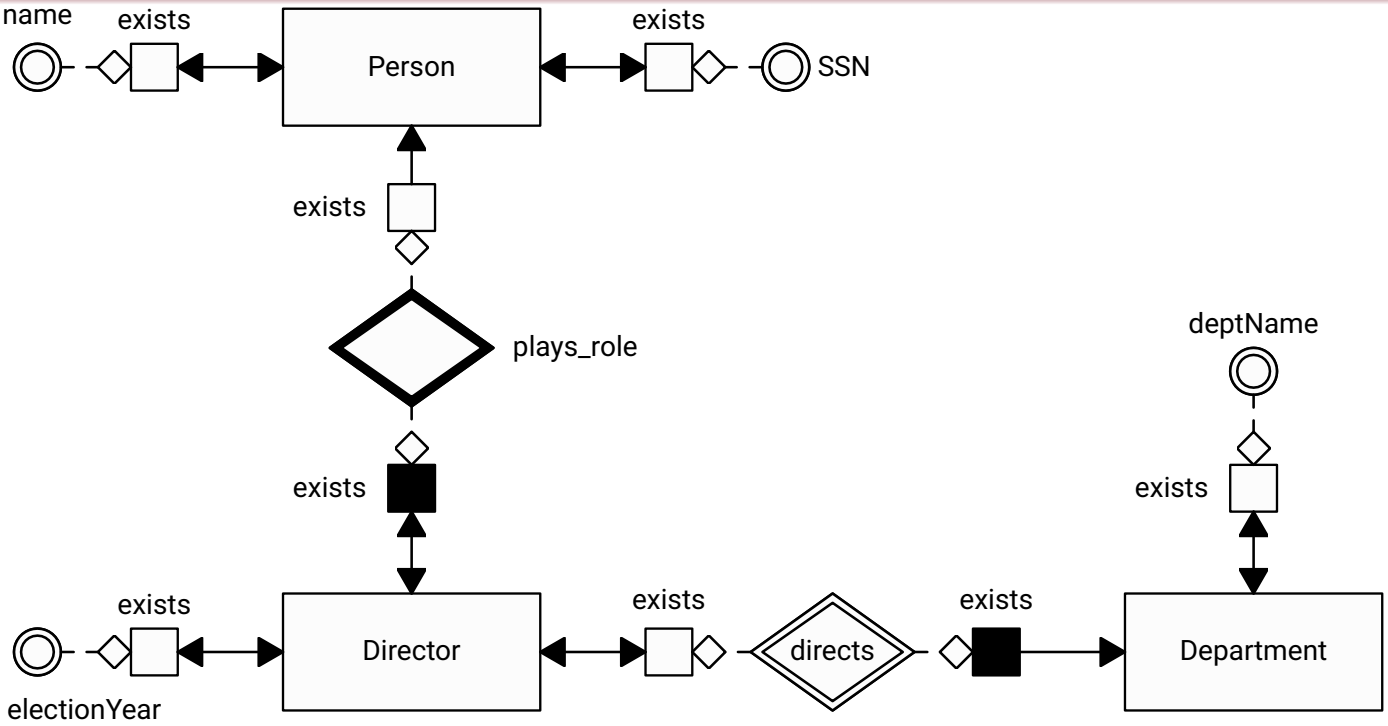


Third option: two classes connected by an object property



- 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.

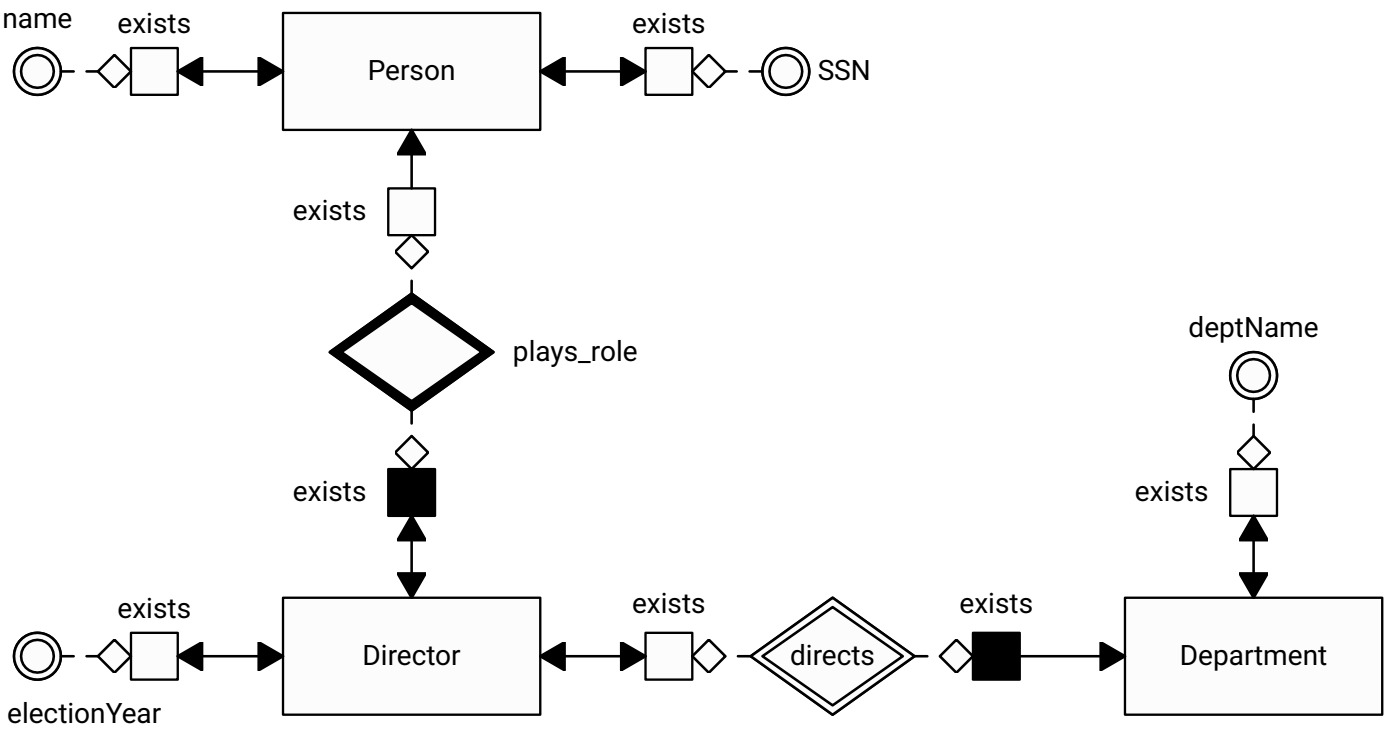
Third option: two classes connected by an object property



Example of Instances:

DL syntax	OWL functional syntax (*no IRI, for simplicity*)
Person(John)	ClassAssertion(Person John)
Director(JohnDir)	ClassAssertion(Director JohnDir)
plays_role(John,JohnDir)	ObjectPropertyAssertion(plays_role John JohnDir)
directs(JohnDir,CSDep)	ObjectPropertyAssertion(directs JohnDir CSDep)
name(John,"John"),	DataPropertyAssertion(name John "John")
electionYear(JohnDir,2015)	DataPropertyAssertion(electionYear JohnDir 2015)

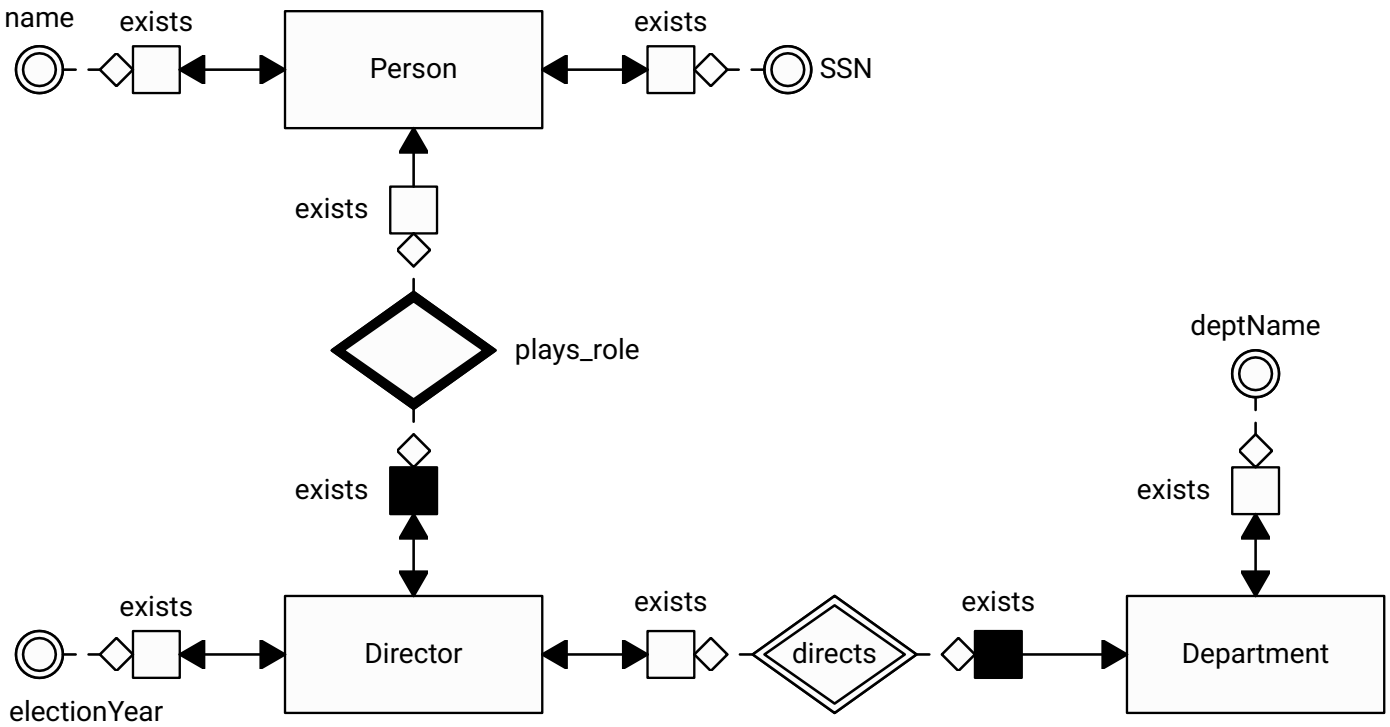
Third option: two classes connected by an object property



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.}
```

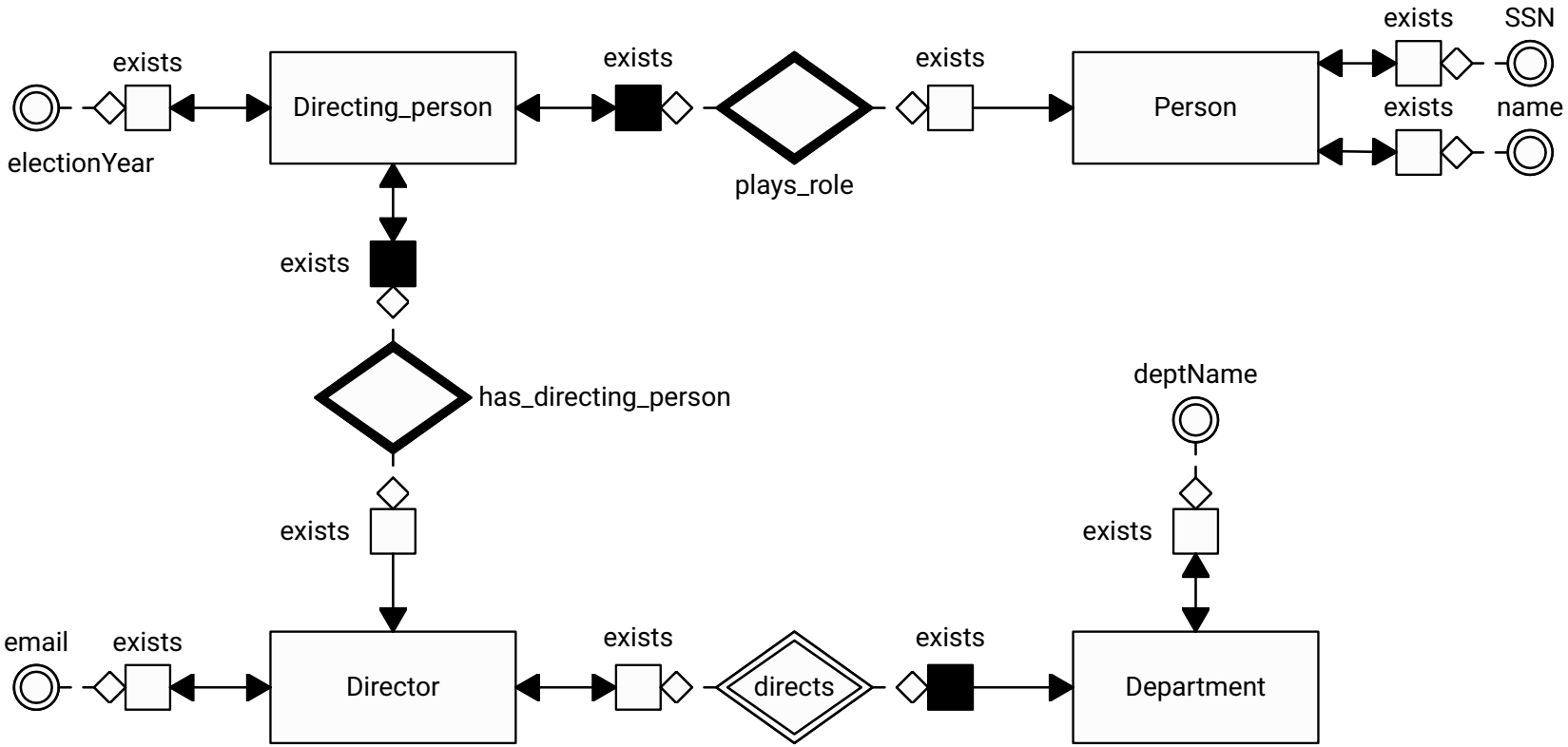
Third option: two classes connected by an object property



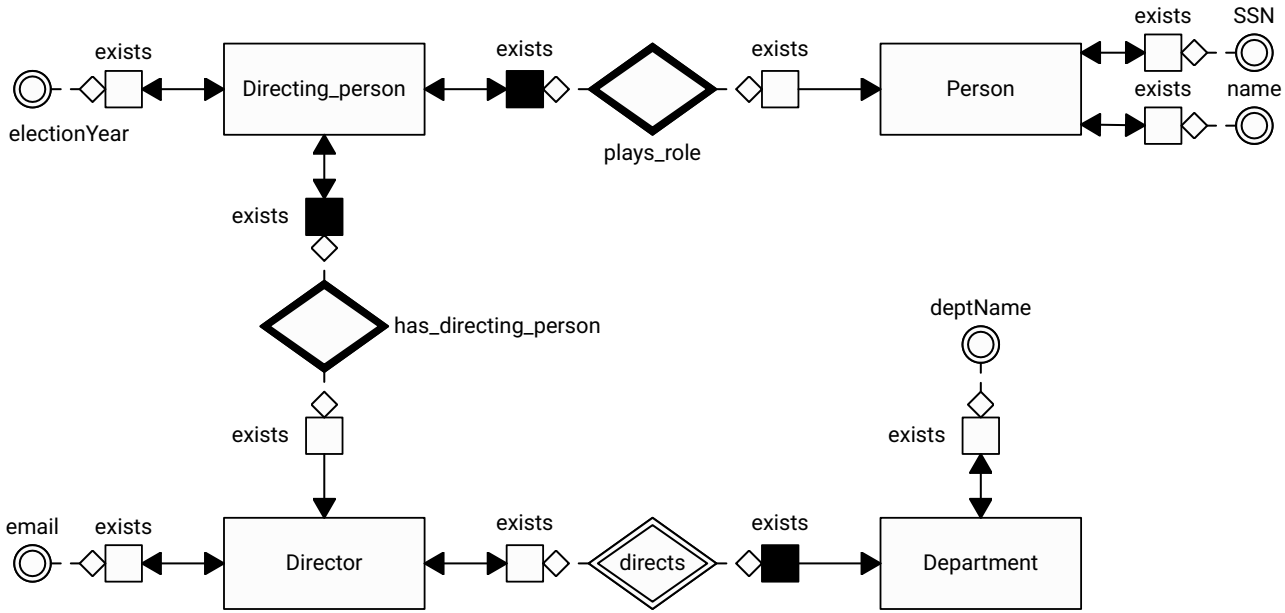
- Problem: How can we model properties that characterize the role without an agent playing it? E.g., how can we specify that the email of the director of the computer science department at Sapienza is `CS_DeptDirector@uniroma1.it` ?

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*).



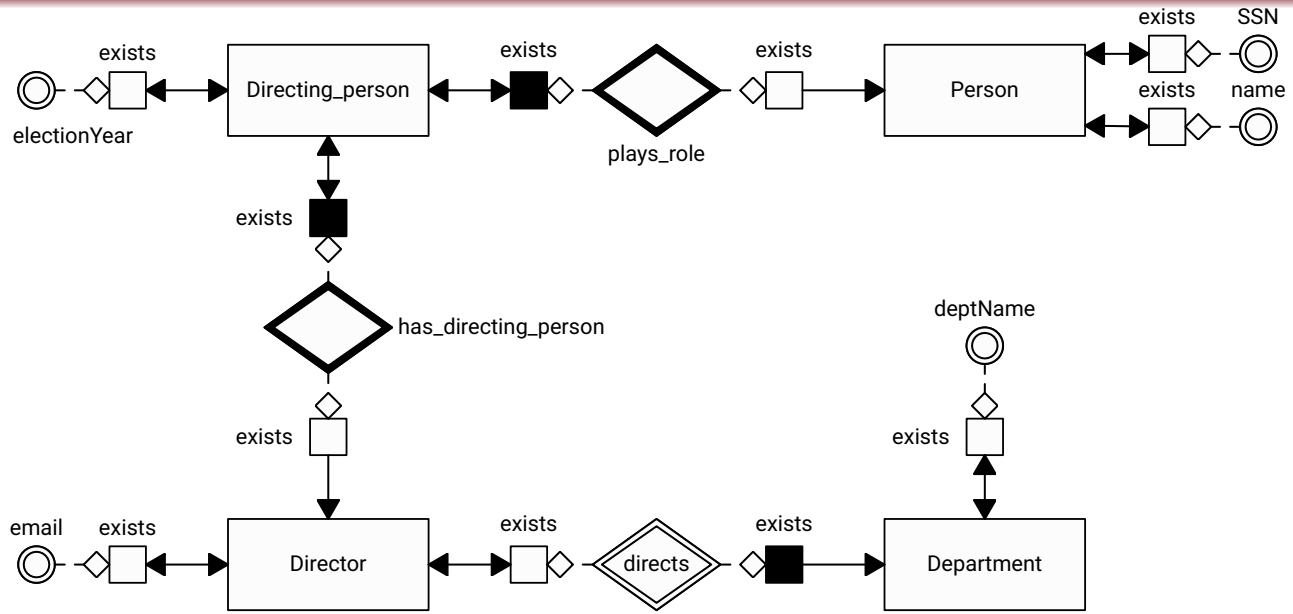
Fourth option: three classes and two object properties



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)

Fourth option: three classes and two object properties

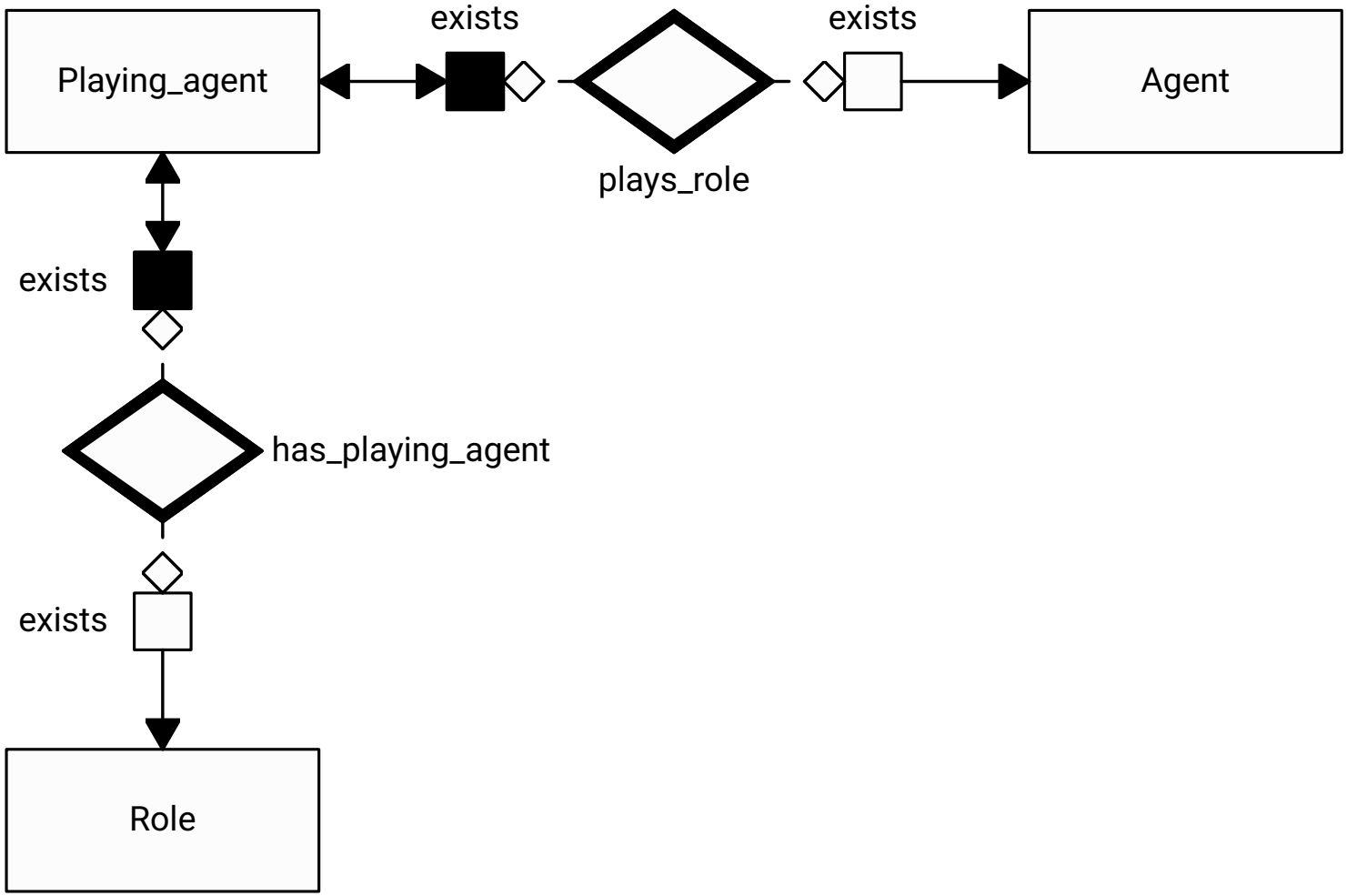


Query that asks for the SSN, the election year and the email of directors:

```
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



Overview

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

  $\rightsquigarrow$ 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 gather 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
  $\rightsquigarrow$ 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.

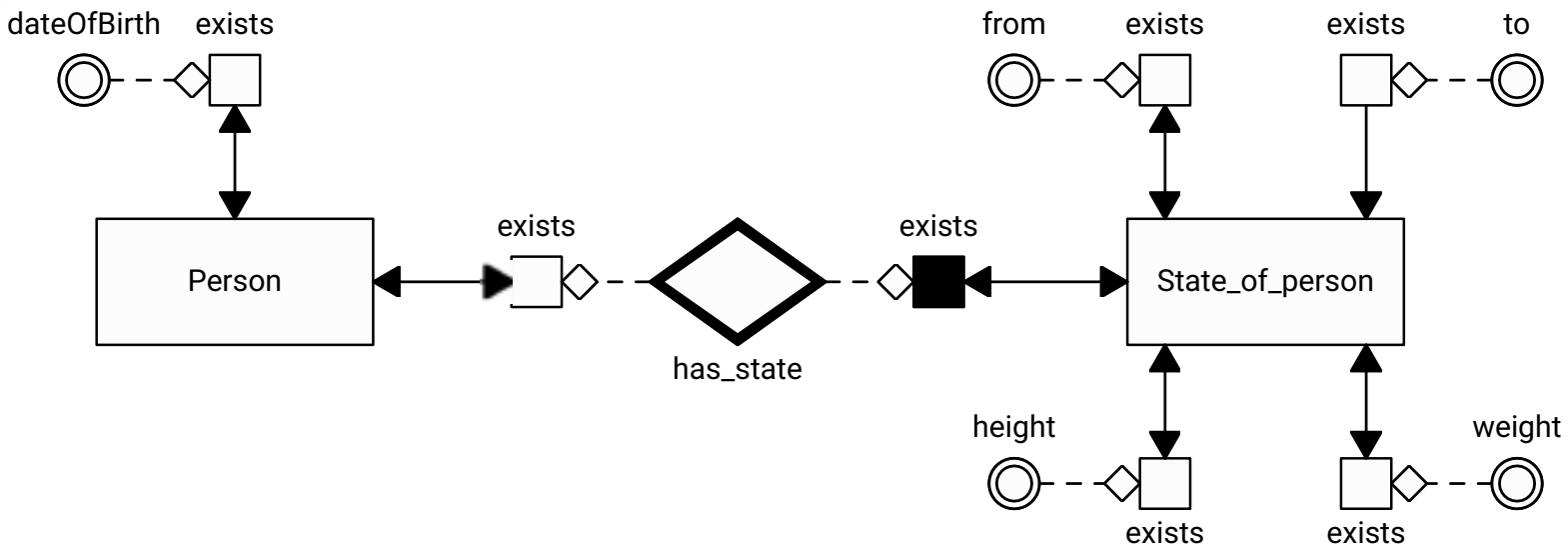
Remark: our aim is to model evolving properties, and not to represent the time (e.g., how to deal with time encodings or temporal ordering relationships). For these latter aspect we refer, for instance, to the OWL-time ontology (https://www.w3.org/TR/owl-time/), whose temporal concepts can be smoothly 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., 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 State_of_C, and each change of a property gives rise to a new instance of State_of_C
   Pros: By accessing an instance of 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., 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

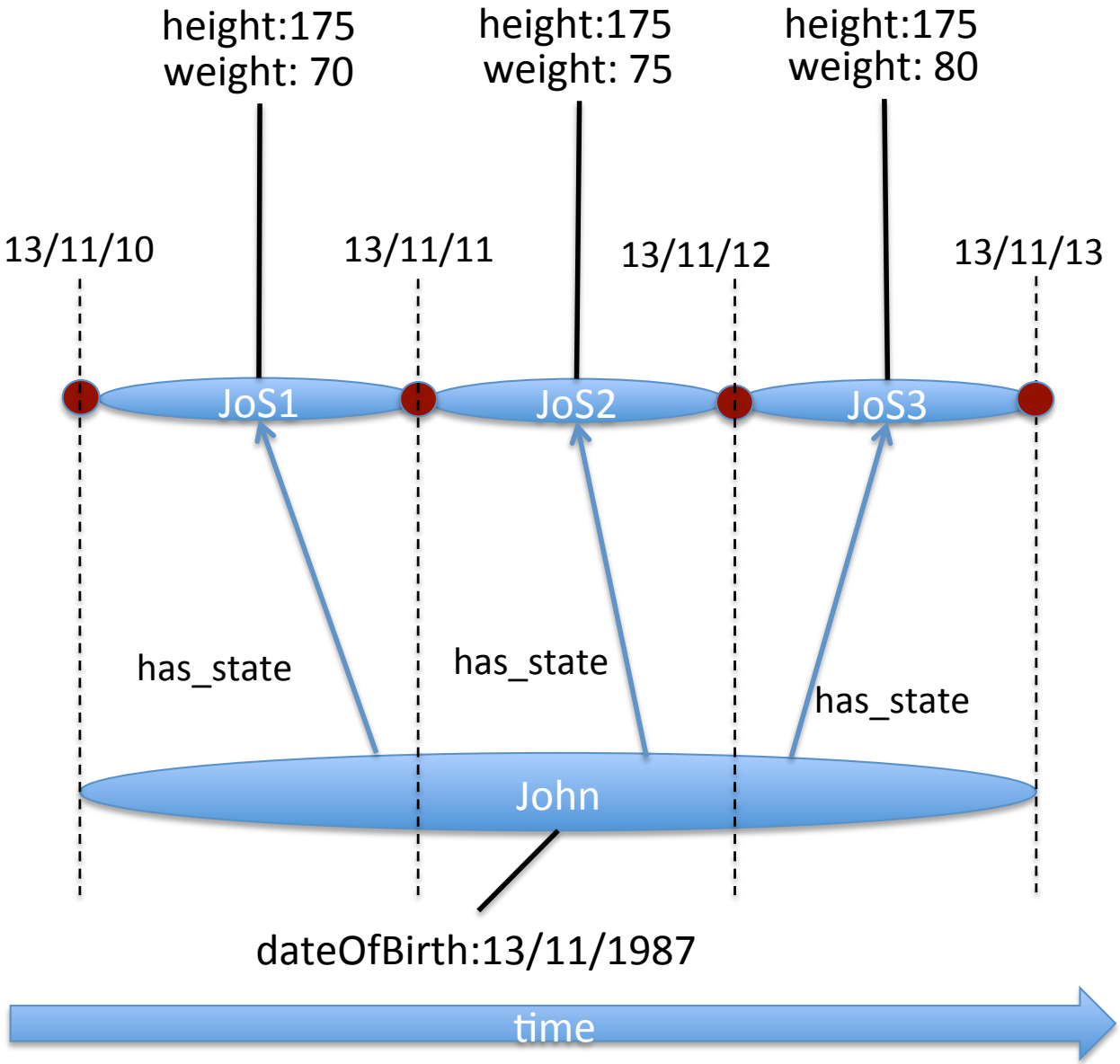


Notice that the data property to *is not mandatory, since current states do not have a final timestamp*

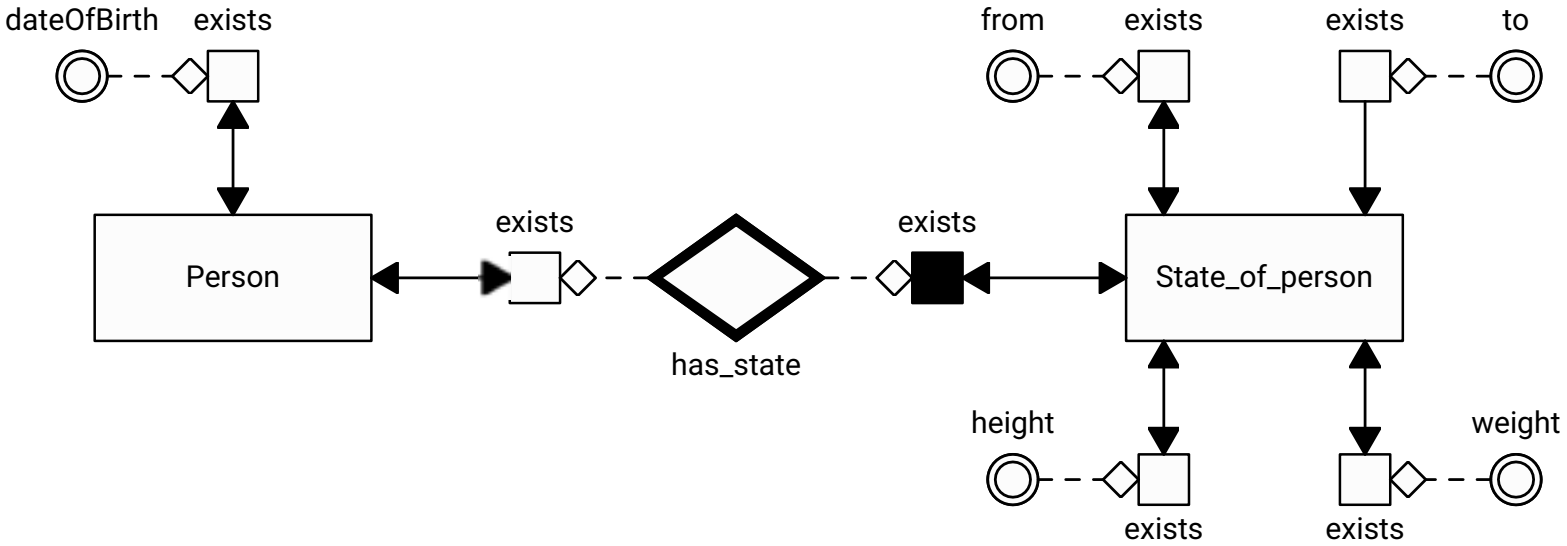
Example of Instances

Person(John)	dateOfBirth(John, 13/11/1987)	
has_state(John,JoS1)	from(JoS1,13/11/2010)	to(JoS1,13/11/2011)
height(JoS1,175)	weight(JoS1,70)	
has_state(John,JoS2)	from(JoS2,13/11/2011)	to(JoS2,13/11/2012)
height(JoS2,175)	weight(JoS2,75)	
has_state(John,JoS3)	from(JoS3,13/11/2012)	to(JoS3,13/11/2013)
height(JoS3,175)	weight(JoS3,80)	

A simple modeling pattern for evolving aspects



A simple modeling pattern for evolving aspects



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

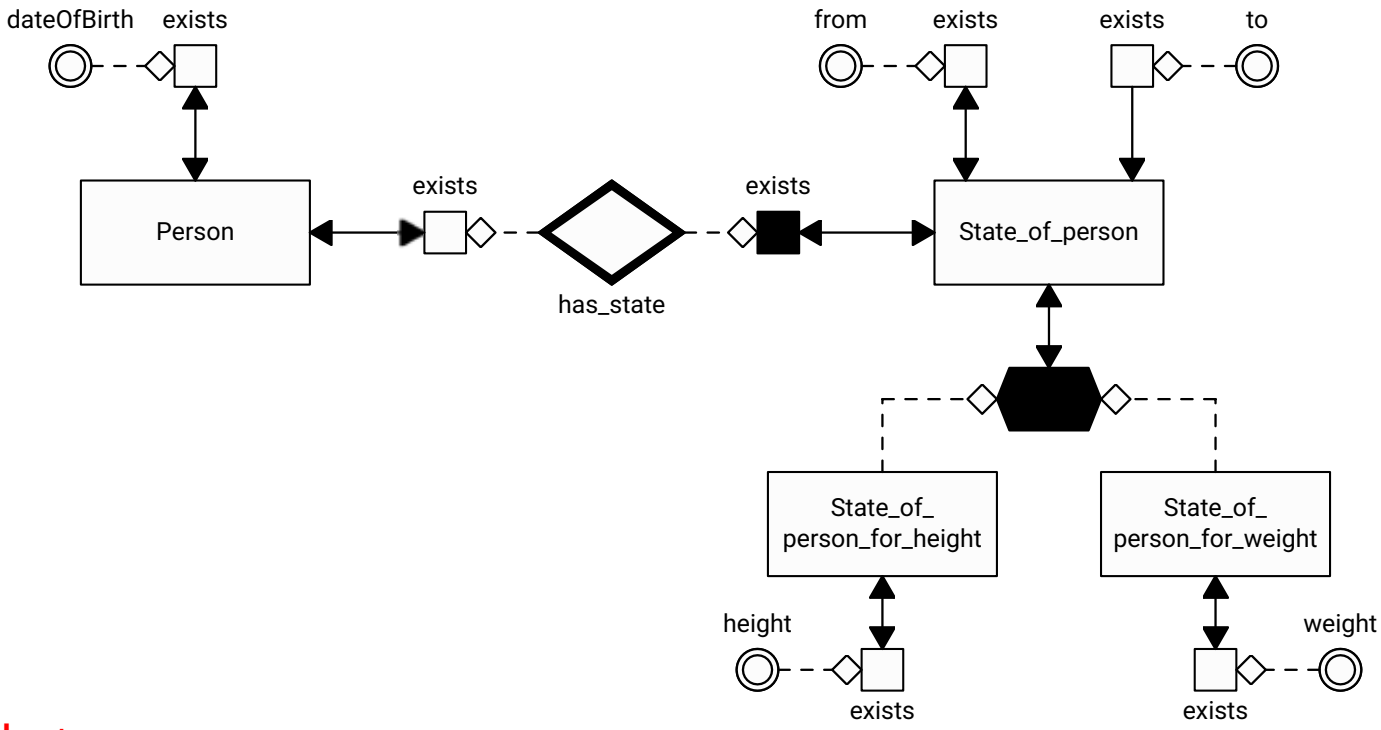
```
Select ?p ?h ?w where {
?p a Person.
?p hasState ?s.
?s from ?t0.
?s to ?t1.
?s height ?h.
?s weight ?w.
FILTER(?t0 <='12/7/2011'^^xsd:date && ?t1 >'12/7/2011'^^xsd:date)
}
```

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_pi, 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.

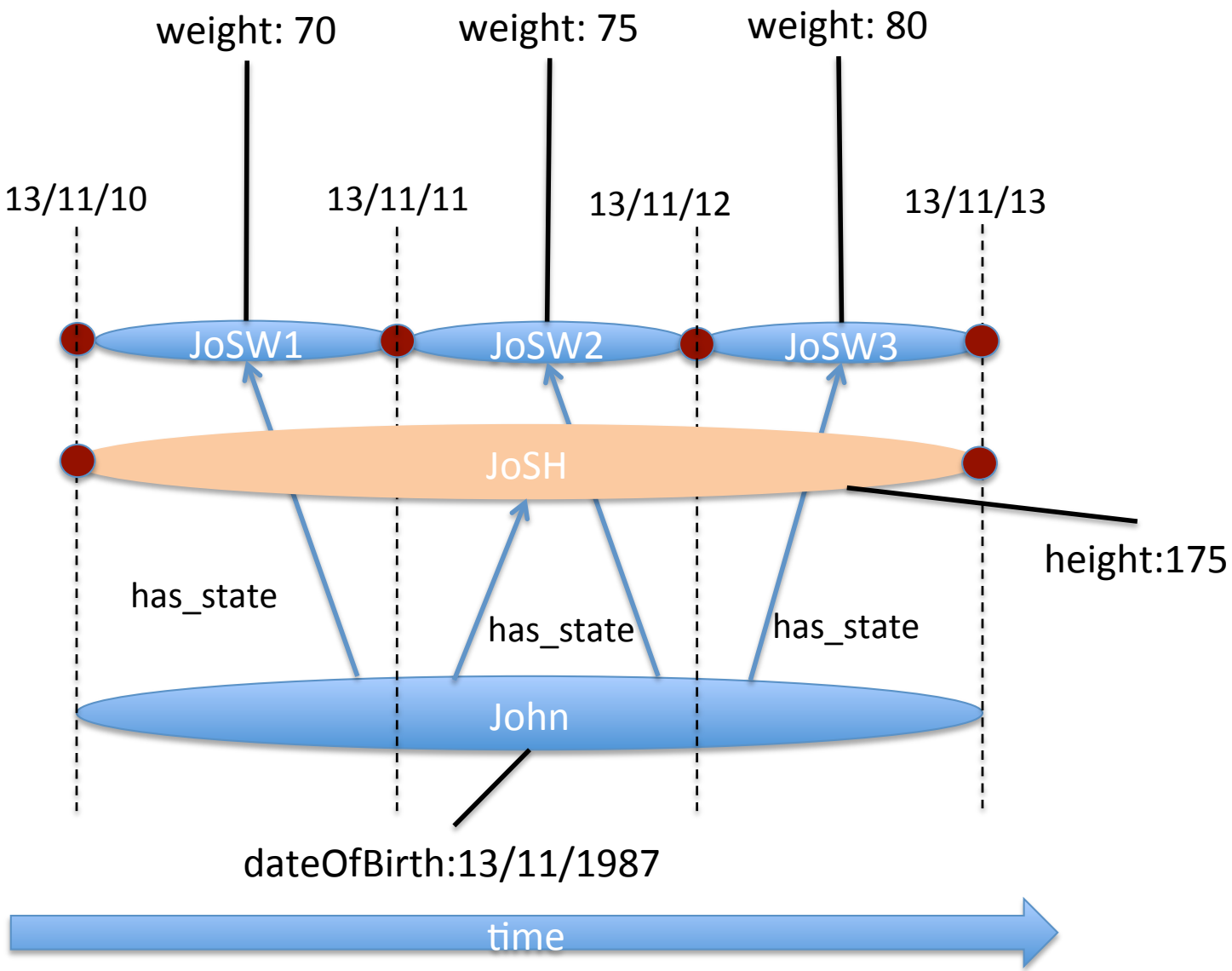
Another simple modeling pattern for evolving aspects



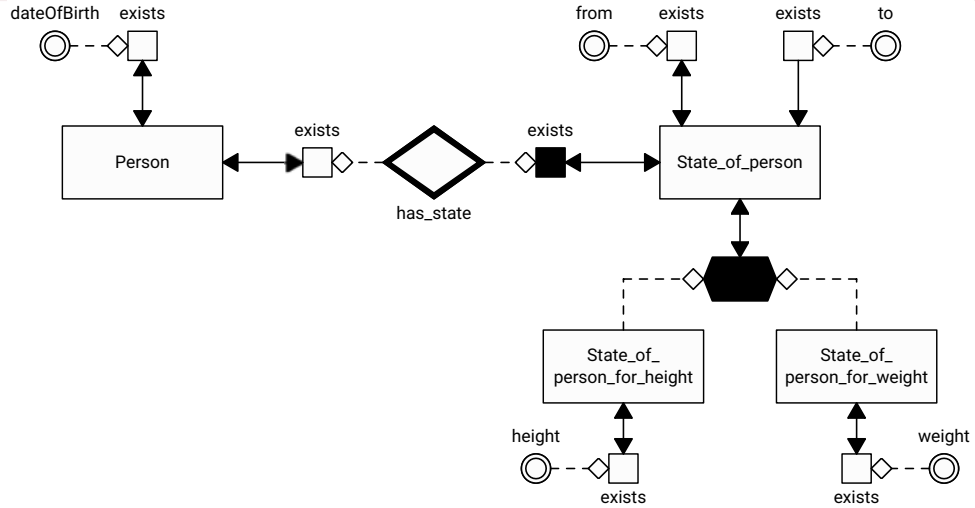
Example of Instances:

Person(John)	dateOfBirth(John, 13/11/1987)	
has_state(John,JoSW1)	from(JoSW1,13/11/2010)	to(JoS1,13/11/2011)
weight(JoSW1,70)	State_of_person_for_weight(JoSW1)	
has_state(John,JoSW2)	from(JoSW2,13/11/2011)	to(JoSW2,13/11/2012)
weight(JoSW2,75)	State_of_person_for_weight(JoSW2)	
has_state(John,JoSW3)	from(JoSW3,13/11/2012)	to(JoSW3,13/11/2013)
weight(JoSW3,80)	State_of_person_for_weight(JoSW3)	
has_state(John,JoSH)	from(JoSH,13/11/2010)	to(JoSH,13/11/2013)
height(JoSH,175)	State_of_person_for_height(JoSH)	

Another simple modeling pattern for evolving aspects



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

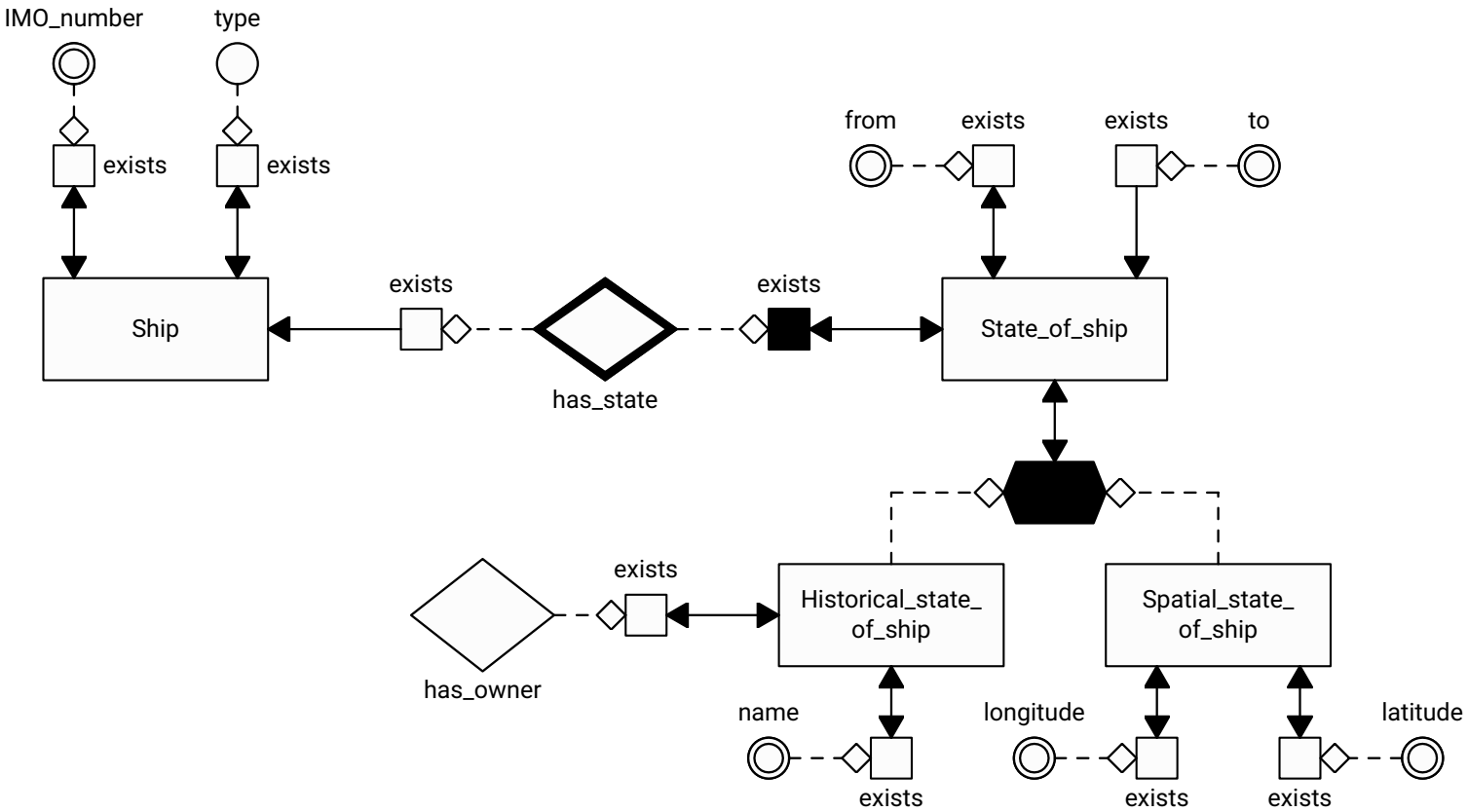
All the above mentioned properties, except the IMO number, evolve, however we are not interested in modeling the evolution of the type, but want to consider only the current type

$\rightsquigarrow$ 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"

Pattern enrichment

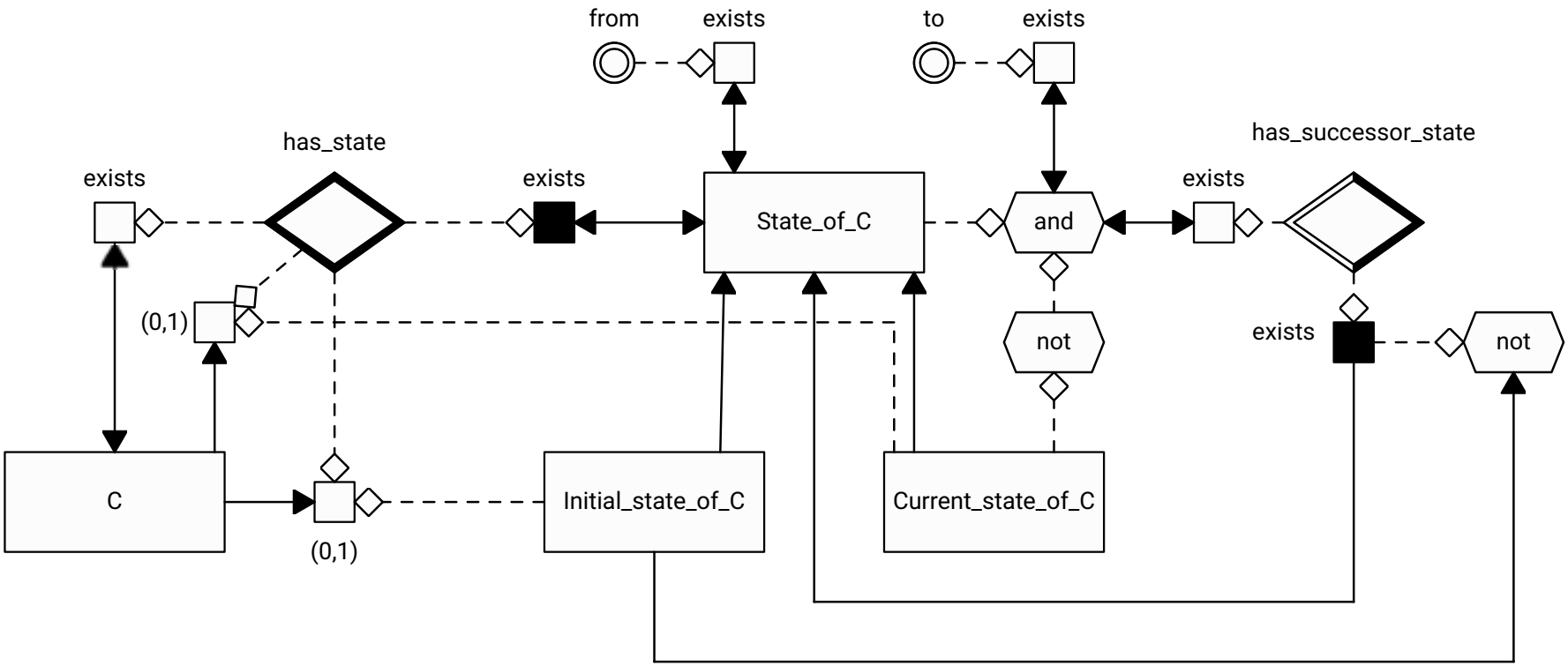
In order to satisfy the information needs of users and simplify the queries over the ontology, the patterns for evolving properties can be enriched with the following elements and set of axioms:

- 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`

Pattern enrichment (Cont'd)

- 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 properti to 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



References I

[ACC+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.

In *Proc. of the 8th Int. Conf. on Formal Ontology in Information Systems (FOIS 2014)*, pages 372–385, 2014.

[AKK+17] Alessandro Artale, Roman Kontchakov, Alisa Kovtunova, Vladislav Ryzhikov, Frank Wolter, and Michael Zakharyaschev.

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

In *24th Int. Symp. on Temporal Representation and Reasoning (TIME 2017)*, pages 1:1–1:37, 2017.

[AKRZ10] Alessandro Artale, Roman Kontchakov, Vladislav Ryzhikov, and Michael Zakharyaschev.

Past and future of dl-lite.

In *Proc. of the 24th AAAI Conf. on Artificial Intelligence (AAAI 2010)*, 2010.

References II

[BHJ+16] Eva Blomqvist, Pascal Hitzler, Krzysztof Janowicz, Adila Krisnadhi, Tom Narock, and Monika Solanki.

Considerations regarding ontology design patterns.

Semantic Web, 7(1):1–7, 2016.

[BS05] Eva Blomqvist and Kurt Sandkuhl.

Patterns in ontology engineering: Classification of ontology patterns.

In *Proc. of the 7th Int. Conf. on Enterprise Information Systems (ICEIS 20015)*, pages 413–416, 2005.

[Gan05] Aldo Gangemi.

Ontology design patterns for semantic web content.

In *Proc. of the 4th Int. Semantic Web Conf. (ISWC 2005)*, pages 262–276, 2005.

[HGJ+16] Pascal Hitzler, Aldo Gangemi, Krzysztof Janowicz, Adila Krisnadhi, and Valentina Presutti, editors.

Ontology Engineering with Ontology Design Patterns – Foundations and Applications, volume 25 of *Studies on the Semantic Web*.

IOS Press, 2016.