An Ontology for Describing Manufactured Objects and their Parts in a Factory

F. Sáenz-Pérez
GPD-DISIA-UCM
April 2012

1 Goals

- Implement an ontology following [AVS08] for a factory that manufactures objects from parts
- Represent concepts, relations, and algebraic properties attached to relations
- Enable constraint checking, both hard and soft. Hard constraint checking forbids inconsistent databases, whereas soft constraint checking allows them

2 Application Description

There are parts that are components of bigger parts or even the completed object, as, e.g., a spoke is a part of a wheel, and a wheel is a part of the completed object bike (respectively).

The basic concepts are parts and completed objects, as: "bike", "wheel", and "spoke".

The basic relation in this application is "component of", which should be attached the algebraic properties "irreflexive", "asymmetric" and "transitive".

3 Application Implementation

We use DES [Sae11] as a host system to implement the ontology, which supports strong constraints, constraint checking, and consistency checking. Datalog program examples cited in this paper can be found in **`factory.dl`** and **`route.dl`**.

3.1 Data Modelling

We focus on the part of the E-R model subpart depicted in Fig. 1 relating "Concepts", "Relations", "BinaryRelation", "HasAP", and "AlgebraicProperty"

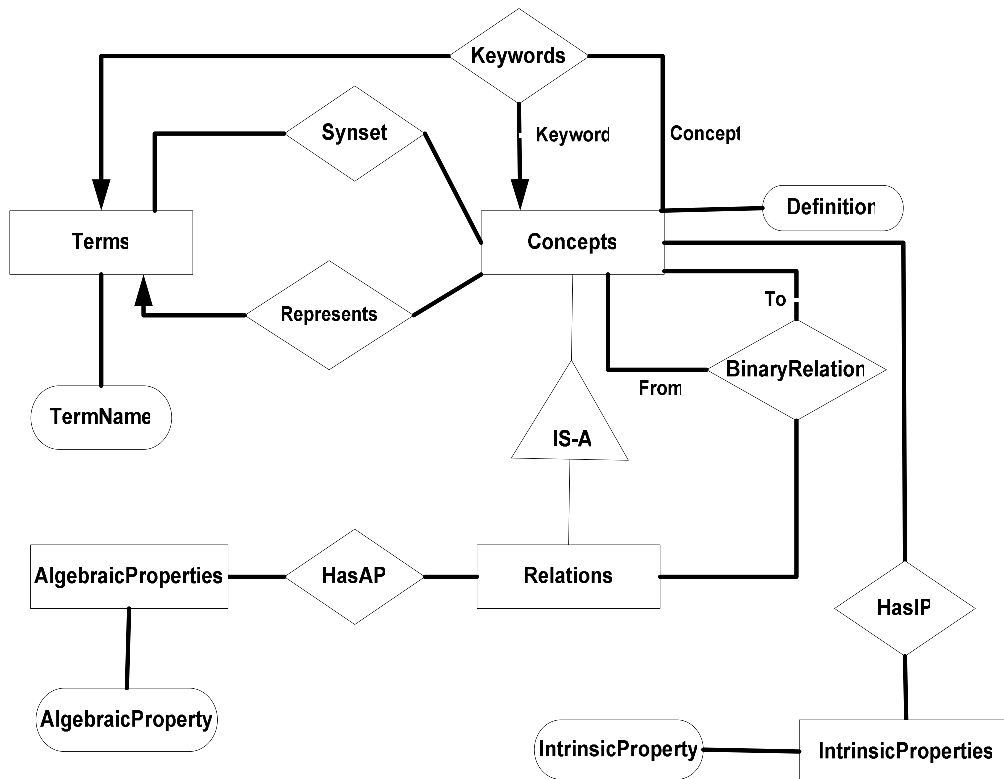

Fig. 1. Ontology Medatada E-R Model

Translating this conceptual model into the relational model, we get:

concepts(Type, <u>Id</u>, Arity)
Type ∈ {concept, relation}
Arity ≥ 0

binary_relation(<u>Id</u>, FromId, ToId)
binary_relation.FromId ∈ concepts.Id
binary_relation.ToId ∈ concepts.Id

algebraic_properties(Id)

has_algebraic_property(<u>BinaryRelation,AlgebraicProperty</u>)
has_algebraic_property.<u>BinaryRelation</u> ∈ binary_relation.Id
has_algebraic_property.<u>AlgebraicProperty</u> ∈ algebraic_properties.Id

The complete physical model can be examined in the file "**factory.dl**".

An instance of this ontology is:

```
% Concepts
concepts(concept, bike, 0).
concepts(concept, wheel, 0).
concepts(concept, spoke, 0).
concepts(concept, bearing, 0).
% Relations
concepts(relation, component_of, 2).
```

```
binary_relation(component_of, bearing, wheel).
binary_relation(component_of, spoke, wheel).
binary_relation(component_of, wheel, bike).

algebraic_properties(irreflexive).
algebraic_properties(asymmetric).
algebraic_properties(transitive).

has_algebraic_property(component_of, irreflexive).
has_algebraic_property(component_of, asymmetric).
has_algebraic_property(component_of, transitive).
```

3.2 Constraints

Algebraic properties can be seen as constraints on an database. These properties, when viewed as constraints, can be classified as exclusion constraints and inclusion constraints. An *exclusion* constraint forbids in an instance database those values which are not consistent w.r.t. this constraint. An *inclusion* constraint specifies all the values that must be in an instance database for it to be consistent. Examples of exclusion constraints include irreflexive and asymmetric, whereas transitive and symmetric are examples of inclusion constraints.

Constraints can also be classified as hard and soft. A *hard* constraint on a database is checked whenever any change related to the involved data sources for the constraint occurs. A *soft* constraint is only checked on-demand. So, in this case, it may be the case that the database becomes inconsistent w.r.t. to the soft constraint. However, they have been acknowledged as a need for certain situations. For instance, in relational databases the concept of *deferred* constraint is used to denote such soft constraints.

3.2.1 Stating Hard Constraints

Attaching an exclusion constraint to a relation implies to impose a strong constraint, which is checked with a Datalog integrity constraint. As an example:

```
:- irreflexive_violation(BinaryRelation, From, To).
```

where:

```
irreflexive_violation(BinaryRelation, From, From) :-
  has_algebraic_property(BinaryRelation, irreflexive),
  binary_relation(BinaryRelation, From, From).
```

That is, it cannot be the case to find a binary relation relating the same element with itself.

Following the example in Section 3.1, trying to assert an entry which makes the ontology inconsistent w.r.t. this algebraic property is rejected:

```
DES> /assert binary_relation(component_of, bearing, bearing).

Error: Integrity constraint violation.
       ic(BinaryRelation,From,To) :-
         irreflexive_violation(BinaryRelation,From,To).
       Offending values in database:
[ic(component_of,bearing,bearing)]
```

Attaching an inclusion constraint to a relation as a strong constraint implies in general to intensionally represent the meaning of the relation under such constraint. Otherwise, there are situations for which it is not possible to assert new data although it is possible to reach a consistent instance database eventually, after further assertions.

For example, considering the property "**transitive**", we can have the following instance database:

```
binary_relation(route, madrid, paris).
binary_relation(route, london, madrid).
binary_relation(route, london, paris).
```

And impose the property "**transitive**" as a hard constraint:

```
:- transitive_violation(BinaryRelation, From, To).
```

where:

```
transitive_violation(BinaryRelation, From, To) :-
  BinaryRelation = route,
  has_algebraic_property(BinaryRelation, transitive),
  count(binary_relation(route, _, _),RelationCardinality),
  count(transitive_route(BinaryRelation, _, _),TCCardinality),
  RelationCardinality \= TCCardinality,
  transitive_route(BinaryRelation, From, To),
  not(binary_relation(route, From, To)).

transitive_route(BinaryRelation, From, To) :-
  has_algebraic_property(BinaryRelation, transitive),
  binary_relation(BinaryRelation, From, To).
transitive_route(BinaryRelation, From, To) :-
  has_algebraic_property(BinaryRelation, transitive),
  binary_relation(BinaryRelation, From, Mid),
  transitive_route(BinaryRelation, Mid, To).
```

Then, it is neither possible to add:

```
binary_relation(route, paris, london).
```

nor:

```
binary_relation(route, madrid, london).
```

because the first tuple requires the second already asserted and the other way round (this example can be found in the file **route.dl**), as illustrated in the next system session:

```
DES> /assert binary_relation(route, paris, london).

Error: Integrity constraint violation.
       ic(BinaryRelation,From,To) :-
         transitive_violation(BinaryRelation,From,To).
       Offending values in database:
[ic(route,madrid,london),ic(route,london,london),ic(route,paris,
madrid),ic(route,paris,paris),ic(route,madrid,madrid)]
Error: Asserting rules due to integrity constraint violation.

DES> /assert binary_relation(route, madrid, london).

Error: Integrity constraint violation.
       ic(BinaryRelation,From,To) :-
         transitive_violation(BinaryRelation,From,To).
       Offending values in database:
[ic(route,london,london),ic(route,madrid,madrid)]
Error: Asserting rules due to integrity constraint violation.
```

Therefore, one possibility is to consider the property "**transitive**" as a soft constraint (c.f. next section) and another one is to consider that the meaning of a relation is extended with the intensional meaning derived by the inclusion property. This way, there is no need for checking consistency w.r.t. to inclusion properties at it is ensured by the definition of the relation.

Following this example about the property "**transitive**", we can extend the meaning of relations with this property attached as follows:

```
binary_relation(BinaryRelation, From, To) :-
  has_algebraic_property(BinaryRelation, transitive),
  binary_relation(BinaryRelation, From, Mid),
  binary_relation(BinaryRelation, Mid, To).
```

Now, any binary relation with attached transitive property, has its meaning extended with its transitive closure.

In the example, given the following explicit two tuples above for "route", already loaded in the database:

```
binary_relation(route,madrid,paris).
binary_relation(route,paris,london).
```

Then, if we ask for its meaning, we get:

```
DES> binary_relation(route,From,To)
{
```

```
  binary_relation(route,madrid,london),
  binary_relation(route,madrid,paris),
  binary_relation(route,paris,london)
}
Info: 3 tuples computed.
```

So that we see that the meaning of relations with the attached property transitive is automatically extended with its transitive closure.

Coming back to the factory example, we can see that incorrect tuples cannot be added in the presence of the transitive property. As "**component_of**" has three properties attached (i.e., properties "**irreflexive**", "**asymmetric**", and "**transitive**")), then it is not possible to assert an entry for "**component_of**" which implies a cycle in the directed graph that this relation forms. For instance:

```
DES> /assert binary_relation(component_of, bike, bearing).

Error: Integrity constraint violation.
       ic(BinaryRelation,From,To) :-
         acyclic_violation(BinaryRelation,From,To).
       Offending values in database:
[ic(component_of,bike,bike),ic(component_of,bearing,bearing),ic(
component_of,wheel,wheel)]
```

This shows that there are paths from "**bike**" to itself, and also for "**bearing**" and "**wheel**" (all of them form a cyclic path, which are not allowed because of the combination of the three properties).

3.2.2 Stating Soft Constraints.

If we omit the strong constraints, we can get inconsistent ontologies along authoring. Let's consider the following case:

```
% Concepts
concepts(concept, bike, 0).
concepts(concept, wheel, 0).
concepts(concept, spoke, 0).
concepts(concept, bearing, 0).
% Relations
concepts(relation, component_of, 2).

binary_relation(component_of, bearing, wheel).
binary_relation(component_of, spoke, wheel).
binary_relation(component_of, wheel, bike).
```

where the algebraic property "**irreflexive**" is not attached to "**component_of**".

Then, we can assert an incorrect entry w.r.t. "**irreflexive**":

```
DES> /assert binary_relation(component_of, bearing, bearing).
```

which is not rejected.

To test which data is inconsistent, we write:

```
DES> irreflexive_violation(BinaryRelation, From, To).

{
  irreflexive_violation(component_of,bearing,bearing)
}
Info: 1 tuple computed.
```

Then, what are soft constraints useful for? Along ontology development, one might allow for inconsistent databases for a number of reasons as, e.g.:

- Unknown data
- Not yet completed data

For the second one, we can think of the algebraic property "**transitive**" as introduced in the previous section. In this case, we require the meaning of relations to be explicit in our database. This means that if the following entries are in the database:

```
binary_relation(component_of, bearing, wheel).
binary_relation(component_of, spoke, wheel).
binary_relation(component_of, wheel, bike).
```

then, the following ones must also be in the database:

```
binary_relation(component_of, bearing, bike).
binary_relation(component_of, spoke, bike).
```

So, to test what are the missing tuples in a database because of the inclusion property "**transitive**", it is attached as a soft constraint, i.e., it is NOT imposed (here, it is shown only as a program remark, i.e., a line preceded by the symbol "**%**"):

```
%:- transitive_violation(BinaryRelation, From, To).
```

(If it is already imposed, you can remove it with:

```
DES> /drop_ic :- transitive_violation(BinaryRelation, From, To).
```

).

This allows us to incrementally build the ontology up to a point in which we think it is consistent and, therefore, to be sure of it. Then, we submit the following consistency check:

```
DES> transitive_violation(BinaryRelation, From, To)
```

```
{
  transitive_violation(component_of,bearing,bike),
  transitive_violation(component_of,spoke,bike)
}
Info: 2 tuples computed.
```

Here, we see that "**component_of**" is not consistent w.r.t. the algebraic property "**transitive**" because the entries shown are missing in the database. If they are asserted, the relation becomes a truly explicit transitive relation and therefore passes the consistency check:

```
DES> transitive_violation(BinaryRelation, From, To)

{
  transitive_violation(component_of,bearing,bike),
  transitive_violation(component_of,spoke,bike)
}
Info: 2 tuples computed.

DES> /assert binary_relation(component_of, bearing, bike)

DES> transitive_violation(BinaryRelation, From, To)

{
  transitive_violation(component_of,spoke,bike)
}
Info: 1 tuple computed.

DES> /assert binary_relation(component_of, spoke, bike)

DES> transitive_violation(BinaryRelation, From, To)

{
}
Info: 0 tuples computed.
```

References

[AVS08] F.J. Álvarez, A. Vaquero, and F. Sáenz-Pérez, "Conceptual Modeling of Ontology-based Linguistic Resources with a Focus on Semantic Relations", In Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC 2008), May, 2008.

[Sae11] F. Sáenz-Pérez, "DES: A Deductive Database System", ENTCS 271, pp. 63-78, 2011. des.sourceforge.net