CLASSIFICATION WITH SHACL RULES

HENRIETTE HARMSE

In my previous post, "Rule Execution with SHACL", we have looked at how SHACL rules can be utilized to make inferences. In this post we consider a more complex situation where SHACL rules are used to classify baked goods as vegan friendly or gluten free based on their ingredients.

1. Why use SHACL and not RDF/RDFS/OWL?

In my discussion I will only concentrate on the definition of vegan friendly baked goods since the translation to gluten free baked goods is similar. Gluten free baked goods are included to give a more representative example.

Essentially what we need to do is look at a baked good and determine whether it includes non-vegan friendly ingredients. If it includes no non-vegan friendly ingredients, we want to assume that it is a vegan friendly baked good. This kind of reasoning uses what is called **closed world reasoning**, i.e. when a fact does not follow from the data, it is assumed to be false. SHACL uses closed world reasoning and hence the reason for why it is a good fit for this problem.

RDF/RDFS/OWL uses **open world reasoning**, which means when a fact does not follow from data or schema, it cannot derive that the fact is necessarily false. Rather, it is both possible (1) that the fact holds but it is not captured in data (or schema), or (2) the fact does not hold. For this reason RDF/RDFS/OWL will only infer that a fact holds (or does not hold) if it explicitly stated in the data or can be derived from a combination of data and schema information. Hence, for this reason RDF/RDFS/OWL are not a good fit for this problem.

2. Baked Goods Data

Below are example baked goods RDF data:

```
bakery:hasIngredient rdf:type owl:ObjectProperty ;
    rdfs:domain bakery:BakeryGood ;
    rdfs:range bakery:Ingredient .

bakery:VeganFriendly rdf:type owl:Class .

bakery:NonVeganFriendly rdf:type owl:Class ;
    owl:disjointWith bakery:VeganFriendly .

bakery:GlutenFree rdf:type owl:Class .

bakery:NonGlutenFree rdf:type owl:Class ;
    owl:disjointWith bakery:GlutenFree .
```

Date: 15th March 2018.

```
bakery:Apple a bakery:VeganFriendly, bakery:GlutenFree .

bakery:Egg a bakery:NonVeganFriendly, bakery:GlutenFree .

bakery:Flour a bakery:VeganFriendly, bakery:NonGlutenFree .

bakery:AlmondFlour a bakery:VeganFriendly, bakery:GlutenFree .

bakery:RiceMilk a bakery:VeganFriendly, bakery:GlutenFree .

bakery:AppleTartA
    a bakery:BakedGood ;
    bakery:hasIngredient bakery:Apple, bakery:Egg, bakery:Flour .

bakery:AppleTartB
    a bakery:BakedGood ;
    bakery:hasIngredient bakery:Apple, bakery:RiceMilk,
        bakery:AlmondFlour .

bakery:AppleTartC
    a bakery:BakedGood ;
    bakery:hasIngredient bakery:Apple, bakery:RiceMilk, bakery:Flour .

bakery:AppleTartD
    a bakery:BakedGood ;
    bakery:hasIngredient bakery:Apple, bakery:Egg, bakery:AlmondFlour .
```

A couple of points are important w.r.t. the RDF data:

1. Note that we define both `VeganFriendly` and `NonVeganFriendly` ingredients to be able to identify ingredients completely. Importantly we state that `VeganFriendly` and `NonVeganFriendly` are disjoint so that we cannot inadvertantly state that an ingredient is both `VeganFriendly` and `NonVeganFriendly`.
2. We state that `AppleTartA-AppleTartD` are of type `BakedGood` so that when we specify our rules, we can state that the rules are applicable only to individuals of type `BakedGood`.
3. We enforce the domain and range for `bakery:hasIngredient` which results in whenever we say `bakery:a bakery:hasIngredient bakery:b`, the reasoner can infer that `bakery:a` is of type `bakery:BakedGood` and `bakery:b` is of type `bakery:Ingredient`.

3. Baked Good Rules

Now we define the shape of a baked good:

```
bakery:BakedGood
    a rdfs:Class, sh:NodeShape ;
    sh:property [
        sh:path bakery:hasIngredient ;
        sh:class bakery:Ingredient ;
        sh:nodeKind sh:IRI ;
        sh:minCount 1 ;
    ] ;
```

We state that `bakery:BakedGood a rdfs:Class` which is important to be able to apply rules to instances of `bakery:BakedGood`. We also state that `bakery:BakedGood a sh:NodeShape` which allows us to add shape and rule information to `bakery:BakedGood`. Note that our `bakery:BakedGood` shape state that a baked good has at least one property called `bakery:hasIngredient` with range `bakery:Ingredient`.

We now add a `bakery:NonVeganFriendly` shape

```
bakery:NonVeganFriendly
    a rdfs:Class, sh:NodeShape .
```

which we will use in the rule definition of `bakery:BakedGood`:

```
    sh:rule [
        a sh:TripleRule ;
        sh:subject sh:this ;
        sh:predicate rdf:type ;
        sh:object bakery:VeganBakedGood ;
        sh:condition bakery:BakedGood ;
        sh:condition [
            sh:property [
                    sh:path bakery:hasIngredient ;
                    sh:qualifiedValueShape
                        [ sh:class bakery:NonVeganFriendly ] ;
                    sh:qualifiedMaxCount 0 ;
            ] ;
        ] ;
    ] ;
    sh:rule [
        a sh:TripleRule ;
        sh:subject sh:this ;
        sh:predicate rdf:type ;
        sh:object bakery:NonVeganBakedGood ;
        sh:condition bakery:BakedGood ;
        sh:condition [
            sh:property [
                    sh:path bakery:hasIngredient ;
                    sh:qualifiedValueShape
                        [ sh:class bakery:NonVeganFriendly ] ;
                    sh:qualifiedMinCount 1 ;
            ] ;
        ] ;
    ] ;
```

We add two rules, one for identifying a `bakery:VeganBakedGood` and one for a `bakery:NonVeganBakedGood`. Note that these rules are of type `sh:TripleRule`, which will infer the existence of a new triple if the rule is triggered. The first rule states that the subject of this triple is `sh:this`, which refers to instances of our `bakery:BakedGood` class. The predicate is `rdf:type` and the object is `bakery:VeganBakedGood`. So if this rule is triggered it will infer that an instance of `bakery:BakedGood` is also an instance of type `bakery:VeganBakedGood`.

Both rules have two conditions which instances must adhere to before these rules will trigger. These rules will only apply to instances of `bakery:BakedGood` according to the first condition. The second condition of the rule for `bakery:VeganBakedGood` checks for `bakery:hasIngredient` properties of the shape `bakery:NonVeganFriendly`.

This ensures that the range of `bakery:hasIngredient` is of type `bakery:NonVeganFriendly`. If `bakery:hasIngredient` has a maximum count of 0, it will infer that this instance of `bakery:BakedGood` is of type `bakery:VeganBakedGood`. The rule for `bakery:NonVeganBakedGood` will also check for `bakery:hasIngredient` properties of the shape `bakery:NonVeganFriendly`, but with minimum count of 1 for which it will then infer that this instance is of type `bakery:NonVeganBakedGood`.

4. Jena SHACL Rule Execution Code

The Jena SHACL implementation provides command line scripts (`/bin/shaclinfer.sh` or `/bin/shaclinfer.bat`) which takes as arguments a data file and a shape file which can be used to do rule execution. However, for this specific example you have to write your own Java code. The reason being that the scripts creates a default model that has no reasoning support. In this section I provide the SHACL Jena code needed to do the classification of baked goods.

```
public static void main(String[] args) {
  try {
    Path path = Paths.get(".").toAbsolutePath().normalize();

    String data = "file:" + path.toFile().getAbsolutePath() +
        "/src/main/resources/bakery.ttl";
    String shape = "file:" + path.toFile().getAbsolutePath() +
        "/src/main/resources/bakeryRules.ttl";

    Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
    Model dataModel = JenaUtil.createDefaultModel();
    dataModel.read(data);
    InfModel infModel = ModelFactory.createInfModel(reasoner, dataModel);
    ValidityReport validity = infModel.validate();
    if (!validity.isValid()) {
      logger.trace("Conflicts");
      for (Iterator i = validity.getReports(); i.hasNext(); ) {
        logger.trace(" - " + i.next());
      }
    } else {
      Model shapeModel = JenaUtil.createDefaultModel();
      shapeModel.read(shape);
      Model inferenceModel = JenaUtil.createDefaultModel();
      inferenceModel = RuleUtil.executeRules(infModel, shapeModel,
          inferenceModel, null);
      String inferences = path.toFile().getAbsolutePath() +
          "/src/main/resources/inferences.ttl";
      File inferencesFile = new File(inferences);
      inferencesFile.createNewFile();
      OutputStream reportOutputStream = new FileOutputStream(inferencesFile);
      RDFDataMgr.write(reportOutputStream, inferenceModel, RDFFormat.TTL);
    }
  } catch (Throwable t) {
    logger.error(WTF_MARKER, t.getMessage(), t);
  }
}
```

5. Running the Code

Running the code will cause an `inferences.ttl` file to be written out to `$Project/src/main/resources/`. It contains the following output:

```
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

<http://bakery.com/ns#AppleTartC>
    a        <http://bakery.com/ns#NonGlutenFreeBakedGood> ,
        <http://bakery.com/ns#VeganBakedGood> .

<http://bakery.com/ns#AppleTartB>
    a   <http://bakery.com/ns#GlutenFreeBakedGood> ,
        <http://bakery.com/ns#VeganBakedGood> .

<http://bakery.com/ns#AppleTartA>
    a   <http://bakery.com/ns#NonGlutenFreeBakedGood> ,
        <http://bakery.com/ns#NonVeganBakedGood> .

<http://bakery.com/ns#AppleTartD>
    a   <http://bakery.com/ns#GlutenFreeBakedGood> ,
        <http://bakery.com/ns#NonVeganBakedGood> .
```

6. Conclusion

In this post I gave a brief overview of how SHACL can be used to do classification based on some property. This code example is available at shacl tutorial. This post was inspired by a question on Stack Overflow.

References