1 Datastructures

`OntologyOperator` Keeps track of the `OWLOntology` object, the reasoner and the `OWLAxiomFlatteningTransformer`. Every operation done on the ontology (reasoning, flattening, other transformations) should be deligated by the `OntologyOperator` or an extension of it. The `OntologyOperator` objects are singleton w.r.t. a given ontology (uniquely associated with an `OWLOntology`). Create or retrieve an ontology operator with the static method:

```
OntologyOperator.getOntologyOperator(OWLOntology o);
```

The operator gives access to reasoner and flattening transformer, this usually directly invokes inference computation or transformation. The standard reasoning task when calling `getReasoner()` is computing the class hierarchy, when calling `getReasoner(true)`, class assertions are computed (implicitly computes class hierarchy). The functionality of the `OntologyOperator` is mainly used by the interpretation generators.

`FlatteningTransformer` Uses the `OWLVisitor` structure to flatten all axioms in the TBox and ABox. Introduces intermediary concept names and their definitions with a unique namespace in the following way:

- $\exists r.C \longrightarrow \exists r.ER_i$ and add $ER_i \equiv C$ to the TBox, if $C = \exists s.C'$,
- $\exists r.C \longrightarrow \exists r.CJ_j$ and add $CJ_j \equiv C$ to the TBox, if $C = \bigsqcap_x C_x$
- $\ldots \sqcap D \sqcap \ldots \longrightarrow \ldots \sqcap ER_i \sqcap \ldots$ and add $ER_i \equiv D$, if $D = \exists s.D'$

until every top-level concept description in an axiom is of the following form:

$$A \mid \bigsqcap_x A_x \mid \exists r.A$$

for $A, A_x \in N_C$ (including intermediary concept names). The indices i, j are incremented globally for every introduced intermediary. This way all existential restrictions in the TBox and ABox can be identified and iterated in order to create canonical models. This has the effect that some ids of domain nodes are intermediary concept names. In case you need the full concept description using only original concept names (no intermediaries), you can unravel exhaustively the TBox definitions of an intermediary. (Helper method `OntologyOperator.getDefinition(OWLClass c)` exists for that purpose).

`CanonicalInterpretation` The canonical interpretation can be named but bears no further function other than storing the `CanonicalDomain`.

`CanonicalDomain` Stores two maps of domain nodes

```
OWLNamedIndividual↦ DomainNode<OWLNamedIndividual>
OWLClassExpression↦ DomainNode<OWLClassExpression>
```

one with `OWLNamedIndividual` ids (i.e. ABox domain elements $d_a \in \Delta^{\mathcal{I}}, a \in N_I$) and one with `OWLClassExpression` ids (i.e. TBox domain elements $d_C \in \Delta^{\mathcal{I}}, C \in \mathfrak{C}(\mathcal{EL})$). You can iterate all domain nodes or retrieve any domain node by its `id` through the `CanonicalDomain`. All successor relations are stored within the domain nodes, thus representing a graph structure. Either iterate all domain nodes or pick a specific domain node and iterate through its successors.

`DomainNode<T>` Generic type describes the type of the `id` object for a `DomainNode`, simply for identification purposes of individual-DomainNodes and concept- DomainNodes. There are different ways to retrieve the successors of a domain node. You can list all successor roles (`getSuccessorRoles()`), get all domain node successors directly by a given role (`getSuccessors(OWLObjectProperty r)`) or compile a list of `RoleConnection` objects. These objects do not need to be stored but in cases where pointed interpretations are required (the domain node itself does not know what interpretation it belongs to) you can use `getSuccessorConnections(OWLObjectProperty r)` to create these object collections.

`RoleConnection` Stores the role-name and both connected domain nodes (from, to) aswell as a pointed interpretation object consisting of a given interpretation and the *to* domain node.

2 Initialization and Usage

Interpretation Generation The interpretation generation iterates all existential restrictions and requires reasoning w.r.t. their sub- or superclasses. This is why the flattening procedure simplifies the entire algorithm and allows to use the efficient `ElkReasoner`. There are two interpretation generators, the `IterativeKBInterpretationGenerator` and the `IterativeQTBoxInterpretationGenerator`. Both implement the `IInterpretationGenerator` interface and can therefore be used very generically. They have the following features in common:

- `generate(OWLOntology o)`: the actual generator-method that uses a given ontology and returns the final `CanonicalInterpretation`
- `setLogger(Logger log)`: there are several logging outputs, if you have a custom initialized logger, you can pass it here so that the logging of the generators behaves as you require. Call this setter with `null` in order to deactivate logging for the generator.
- `getClassRepresentation(OWLClassExpression expr)`: a helper method to get intermediary class expressions for reasoning purposes. Whenever it is not clear whether a handled class expression object is actually a concept description or just a concept name, this method will either return

the expression as it is (if it is a concept name) or return the concurrent intermediary class for the complex class expression.

You can use the `InterpretationGeneratorFactory` for easy creation of a generator. When calling the `create()` method with a concept description, it will automatically generate a QTBox generator as opposed to the default KB generator. An additional option for the KB generator is passing a set of individuals to the generate method[1]. This set is the base of the iterative unraveling of the canonical interpretation via successor relations. If no set is given explicitly, the set $sig(\mathcal{A}) \cap N_I$ is used.

Example:

```
IInterpretationGenerator gen = InterpretationGeneratorFactory.create();
CanonicalInterpretation i = gen.generate(ontology); // create KB model
System.out.println(i);
```

The debug-output of a canonical interpretation is pretty straight forward. It is split into the two sets of domain nodes and every domain node gets a line like this:

`d(id)` [list of instantiators] [list of successors]

On-The-Fly Normalization You have the option to create a generator with on-the-fly normalization. Usually the definition of a normal form (and the transformation to the normal form) of a canonical model relies on simulations between nodes. However when two domain nodes represent a concept description (rather than an individual), the subsumption relation between those concepts coincides with the simulation relation. Thus on-the-fly normalization omits to check for simulations after creating the model, however we can only normalize connections between those domain elements that represent a concept description. Here is an example for such a normalization step:

Example: Node d_A already has an r-successor to d_B and while iterating all superclasses of A, we find $A \sqsubseteq \exists r.C$. Usually we would create an r-successor from d_A to d_C, however if the TBox entails $C \sqsubseteq B$ we will not add this connection, since d_C will always behave as d_B and thus d_B would simulate d_C.

If the normalization is disabled, all nodes and connections are created without hesitation.

The generation algorithm This is just a brief description of how the iterative model generation works. First of all, the QTBox generator has a very general method that will unravel any domain node associated with a concept description by iterating all its superclasses. The generator starts by creating the domain node for the query concept and unraveling it. It keeps track of all freshly generated domain nodes and iteratively unravels those until no fresh domain nodes were created. This procedure always terminates for a finite TBox.

The KB model generation starts from the given set of individuals. It creates connections for all role assertion axioms that are present in the ABox. Then it

[1]Typecast necessary, this option is not represented through the common interface.

creates all connections starting from an individual domain element and going to a concept description domain element, i.e. create $d_a \xrightarrow{r} d_C$, if $KB \models (\exists r.C)(a)$. After all 1-step connections starting from individual domain elements are created, this set of reached domain nodes will be unraveled by the QTBox generator.

These iterative unraveling procedures ensure that the model consists of only those elements that are somehow reachable from a node of interest.