E. Annex: Conformance of UML class and object diagrams with DOL

(Informative)

This informative annex demonstrates conformance of a subset of UML class and object diagrams with DOL by defining an institution for both. The subset is restricted to the static aspects of class diagrams; that is, change of state is ignored. This means that all operations are query operations.

The institution of UML class and object diagrams is defined using a translation of UML class diagrams to Common Logic, following the fUML specification and [70].

E.1. Preliminaries

The axioms for primitive types are imported from the fUML specification, section 10.3.1: Booleans, numbers, sequences and strings. These axiomatize (among others) predicates corresponding to primitive types, e.g. `buml:Boolean`, `form:Number`, `form:NaturalNumber`, `buml:Integer`, `form:Sequence`, `form:Character`, and `buml:String`.

The following infrastructure, consisting off a number of predicates axiomatized in Common Logic, provides a foundation for an institution for UML class diagrams described in the later sections of this Annex.

```
logic CLIF

oms pairs =
  (forall (x y) (= (form:first (form:pair x y)) x))
  (forall (x y) (= (form:second (form:pair x y)) y))
  (forall (x y) (form:Pair (form:pair x y)))
  (forall (p) (if (form:Pair p)
                  (= (form:pair (form:first p) (form:second p)) p)))
end

oms sequences =
fuml:sequences.clif and pairs
then
  // fuml:sequence - membership of an element in a sequence
  (forall (x s)
      (if (form:sequence-member x s)
          (form:Sequence s)))

  (forall (x s)
      (iff (form:sequence-member x s)
           (exists (pt)
               (and (form:in-sequence s pt)
                    (form:in-position pt x)) )))

  // selection of elements
  (forall (o) (= (form:select1 o form:empty-sequence) form:empty-sequence))
  (forall (o y s)
          (= (form:select1 o (form:sequence-insert (form:pair o y) s))
             (form:sequence-insert y (form:select1 o s))))
  (forall (o x y s)
          (if (not (= x o))
              (= (form:select1 o (form:sequence-insert (form:pair x y) s))
                 (form:select1 o s))))
  (forall (o) (= (form:select2 o form:empty-sequence) form:empty-sequence))
  (forall (o x s)
```

```
          (= (form:select2 o (form:sequence-insert (form:pair x o) s))
             (form:sequence-insert x (form:select2 o s))))
  (forall (o x y s)
          (if (not (= y o))
              (= (form:select2 o (form:sequence-insert (form:pair x y) s))
                 (form:select2 o s))))

  (forall (i s)
          (= (form:n-select form:empty-sequence i s)
             form:empty-sequence))
  (forall (a i s t x)
          (if (= (insert-i i x t) s)
              (= (form:n-select (form:sequence-insert s a) i t)
                 (form:sequence-insert s (form:n-select a i t)))))
  (forall (a i s t)
          (if (not (exists (x) (= (insert-i i x t) s)))
              (= (form:n-select (form:sequence-insert s a) i t)
                 (form:n-select a i t))))

  // insert element at i-th position
  (forall (x s)
          (= (insert-i form:0 x s) (form:sequence-insert x s)))
  (forall (i j x y s)
          (if (form:add-one i j)
              (= (insert-i j x (form:sequence-insert y s))
                 (form:sequence-insert y (insert-i i x s)))))
end

oms sequences-insert =
sequences then
  // insertion of elements
  (forall (x s1 s2)
    // inserting an element means...
    (if (= (form:sequence-insert x s1) s2)
        (and (form:Sequence s1)
             (form:Sequence s2)
             // the new element is at the first position
             (form:in-position-count s2 form:1 x)
             // and all other elements are shifted by one
             (forall (n1 n2 y)
               (if (form:add-one n1 n2)
                   (iff (form:in-position-count s1 n1 y)
                        (form:in-position-count s2 n2 y)))))))
  // synonym
 (forall (s) (= (form:sequence-length s) (form:sequence-size s)))
end

oms ordered-sets =
sequences with
  form:Sequence |-> form:Ordered-Set,
  form:empty-sequence |-> form:empty-ordered-set,
  form:sequence-length |-> form:ordered-set-size,
  form:same-sequence |-> form:same-ordered-set,
  form:sequence-member |-> form:ordered-set-member,
  form:in-sequence |-> form:in-ordered-set,
  form:before-in-sequence |-> form:before-in-ordered-set,
  form:position-count |-> form:ordered-set-position-count,
  form:in-position-count |-> form:in-ordered-set-position-count
then
  //Different positions contain different elements
  (forall (s x1 x2 n1 n2)
            (if (and (form:in-ordered-set-position-count s n1 x1)
```

```
                    (form:in-ordered-set-position-count s n2 x2)
                    (= x1 x2))
                (= n1 n2)))
  // insertion of elements
  (forall (x s1 s2)
    (if (= (form:ordered-set-insert x s1) s2)
        (and (form:Ordererd-Set s1)
             (form:Ordererd-Set s2)
  // no element can be inserted twice
  (forall (x s)
    (if (from:ordered-set-member x s)
        (= (form:ordered-set-insert x s) s)))
  // inserting a new element
  (forall (x s)
    (if (not (from:ordered-set-member x s1))
        (exists (s2)
          (and (= (form:ordered-set-insert x s1) s2)
               // the new element is at the first position
               (form:in-ordered-set-position-count s2 form:1 x)
               // and all other elements are shifted by one
               (forall (n1 n2 y)
                 (if (form:add-one n1 n2)
                     (iff (form:in-ordered-set-position-count s1 n1 y)
                          (form:in-ordered-set-position-count s2 n2 y)))))))
end

oms sets =
//An empty set has no members.
(forall (s)
        (if (form:empty-set s)
            (form:Set s)))
(forall (s)
        (if (form:Set s)
            (iff (form:empty-set s)
                 (not (exists (x)
                              (form:set-member x s))))))
//Size of sets
(forall (s n)
        (if (form:set-size s n)
            (and (form:Set s)
                 (buml:UnlimitedNatural n))))
(= (form:set-size form:empty-set) form:0)
(forall (x s)
        (if (not (form:set-member x s))
            (exists (n)
              (and (form:add-one (form:set-size s) n)
                   (= (form:set-size (form:set-insert x s))
                      n)))))

//The same-set relation is true for sets that have the same members.
// but: why not replace same-set with = ?
(forall (s1 s2)
        (if (form:same-set s1 s2)
            (and (form:Set s1)
                 (form:Set s2))))
(forall (s1 s2)
        (iff (form:same-set s1 s2)
             (forall (x)
                     (iff (form:set-member x s1)
                          (form:set-member x s2)))))
//Insertion of elements into sets and set membership
(forall (x s)
```

```
        (if (form:Set s)
            (form:Set (form:set-insert x s))))
(forall (x y s)
        (iff (form:set-member x (form:set-insert y s))
             (or (= x y)
                 (form:set-member x s))))
end

oms bags =
//An empty bag has no members.
(forall (s)
        (if (form:empty-bag s)
            (form:Bag s)))
(forall (s)
        (if (form:Bag s)
            (iff (form:empty-bag s)
                 (not (exists (x)
                              (form:bag-member x s))))))
//Size of bags
(forall (s n)
        (if (form:bag-size s n)
            (and (form:Bag s)
                 (buml:UnlimitedNatural n))))
(= (form:bag-size form:empty-bag) form:0)
(forall (x s)
        (exists (n)
            (and (form:add-one (form:bag-size s) n)
                 (= (form:bag-size (form:bag-insert x s))
                    n))))

//The same-bag relation is true for bags that have the same members.
(forall (s1 s2)
        (if (form:same-bag s1 s2)
            (and (form:Bag s1)
                 (form:Bag s2))))
(forall (s1 s2)
        (iff (form:same-bag s1 s2)
             (forall (x)
                     (iff (form:bag-member-count x s1)
                          (form:bag-member-count x s2)))))
//Insertion of elements into bags and bag membership
(forall (x s)
        (if (form:Bag s)
            (form:Bag (form:bag-insert x s))))
(forall (x y s)
        (iff (form:bag-member x (form:bag-insert y s))
             (or (= x y)
                 (form:bag-member x s))))
//Member count
(forall (x s)
        (if (form:Bag s)
            (buml:UnlimitedNatural (form:bag-member-count x s))))
(= (form:bag-member-count form:empty-bag) form:0)
(forall (x s)
        (exists (n)
           (and (form:add-one (form:bag-member-count x s) n)
                (= (form:bag-member-count x (form:bag-insert x s))
                   n))))
(forall (x y s)
        (if (not (= x y))
            (= (form:bag-member-count x (form:bag-insert y s))
               (form:bag-member-count x s))))
```

```
end

oms collection-types =
  sequences-insert and ordered-sets and sets and bags
then
//bag to set
(forall (b)
        (if (form:Bag s)
            (form:Set (form:bag2set b))))
(= (form:bag2set form:empty-bag) form:empty-set)
(forall (x b)
        (if (form:Bag b)
            (= (form:bag2set (form:set-insert x b))
               (form:bag-insert x (form:bag2set b)))))

//sequence to ordered set
(forall (s)
        (if (form:Sequence s)
            (form:Ordered-Set (form:seq2ordset s))))
(= (form:seq2ordset form:empty-sequence) form:empty-ordered-set)
(forall (x s)
        (if (form:Sequence s)
            (= (form:seq2ordset (form:sequence-insert x s))
               (form:ordered-set-insert x (form:seq2ordset s)))))

//sequence to bag
(forall (s)
        (if (form:Sequence s)
            (form:Bag (form:seq2bag s))))
(= (form:seq2bag form:empty-sequence) form:empty-bag)
(forall (x s)
        (if (form:Sequence s)
            (= (form:seq2bag (form:sequence-insert x s))
               (form:bag-insert x (form:seq2bag s)))))

//ordered-set to set
(forall (b)
        (if (form:Ordered-Set s)
            (form:Set (form:ordset2set b))))
(= (form:ordset2set form:empty-ordered-set) form:empty-set)
(forall (x b)
        (if (form:Ordered-Set b)
            (= (form:ordset2set (form:set-insert x b))
               (form:ordered-set-insert x (form:ordset2set b)))))

//sequence to set
(forall (s)
        (if (form:Sequence s)
            (form:Set (form:seq2set s))))
(forall (s) (= (form:seq2set s) (form:ordset2set (form:seq2ordset s))))

// leq
(forall (x y)
   (iff (buml:leq x y)
        (or (= x y)
            (buml:less-than x y))))
end


oms uml-cd-preliminaries =
  collection-types and pairs
end
```

E.2. Signatures

Class/data type hierarchies. A *class/data type hierarchy* $(C, \leq_C)$ is given by a partial order where the set C contains the *class/data type names*, which are closed w.r.t. the *built-in data types* Boolean, UnlimitedNatural, Integer, Real, and String, i.e., $\{\mathsf{Boolean}, \mathsf{UnlimitedNatural}, \mathsf{Integer}, \mathsf{Real}, \mathsf{String}\} \subseteq C$; and the partial ordering relation $\leq_C$ represents a *generalization relation* on C, where c_1 is a *sub-class/data type* of c_2 if $c_1 \leq_C c_2$.

A *class/data type hierarchy map* $\gamma : (C, \leq_C) \to (D, \leq_D)$ is given by a monotone map from $(C, \leq_C)$ to $(D, \leq_D)$, i.e., $\gamma(c) \leq_D \gamma(c')$ if $c \leq_C c'$, such that $\gamma(c) = c$ for all $c \in \{\mathsf{Boolean}, \mathsf{UnlimitedNatural}, \mathsf{Integer}, \mathsf{Real}, \mathsf{String}\}$.

The *collection type constructors* OrderedSet, Set, Sequence, and Bag are used for representing the meta-attributes "ordered" and "unique" of MultiplicityElement according to the following table:[1]

	ordered	not ordered
unique	OrderedSet	Set
not unique	Sequence	Bag

The default is "not ordered" and "unique".[2]

For a class/data type $c \in C$ of a class/data type-hierarchy $(C, \leq_C)$ and a collection type constructor

$$\tau \in \{\mathsf{OrderedSet}, \mathsf{Set}, \mathsf{Sequence}, \mathsf{Bag}\},$$

the expression $\tau[c]$ denotes the induced *collection type*.

Let $(C, \leq_C)$ be a class/data type hierarchy.

- An *attribute declaration* over $(C, \leq_C)$ is of the form $c.p : \tau[c']$ with $c, c' \in C$, τ a collection type constructor, and p an *attribute name*. (Attributes and association member ends are distinguished due to their different uses. In UML, both are of class Property.)
- A *query operation declaration* over $(C, \leq_C)$ is of the form $c.q(x_1 : \tau_1[c_1], \ldots, x_r : \tau_r[c_r]) : \tau[c']$ with $c, c_1, \ldots, c_r, c' \in C$, τ a collection type constructor, o an *operation name*, and $x_1, \ldots, x_r$ *parameter names*.
- An *association declaration* over $(C, \leq_C)$ is of the form $a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r])$ with $r \geq 2$, $c_1, \ldots, c_r \in C$, $\tau_1, \ldots, \tau_r$ classifier annotations, a an *association name*, and $p_1, \ldots, p_r$ *member end names*.[3] An association declaration $\mathbf{a} = a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r])$ *yields* the *property declarations* $\mathbf{a}.p_i : \tau_i[c_i]$ for $1 \leq i \leq r$. An association declaration is *binary* if $r = 2$.[4]
- A *composition declaration* over $(C, \leq_C)$ is of the form $m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2])$ with $c_1, c_2 \in C$, τ_2 a collection type constructor, m a *composition name*, and p_1, p_2 *member end names*. A composition declaration $\mathbf{m} = m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2])$ *yields* the property declarations $\mathbf{m}.p_1 : \mathsf{Set}[c_1]$ and $\mathbf{m}.p_2 : \tau_2[c_2]$.

In UML, each Property may have AggregationKind composite. However, such an aggregation kind has no semantic meaning when the property is not a member end of an association: the UML Superstructure Specification 2.4.1 does not mention the aggregation kind in the description of the semantics of Property, and UML 2.5 explains the use of aggregations for Property as "to model circumstances in which *one instance* is used to group together a set of instances" (p. 112, our emphasis). Moreover, composite properties, i.e., properties with aggregation kind composite can only be member ends of binary associations (UML Superstructure Specification 2.4.1, p. 37; UML 2.5, p. 228) and their multiplicity must not exceed one (UML Superstructure Specification 2.4.1, p. 126; UML 2.5, p. 155). Thus, composition declarations are distinguished from general association declarations.

Class/data type nets (Signatures). A *class/data type net* $\Sigma = ((C, \leq_C), P, O, A, M)$ comprises a class/data type hierarchy $(C, \leq_C)$ and a set P of attribute declarations, a set O of operation declarations, a set A of association declarations over $(C, \leq_C)$, and a set M of composition declarations over $(C, \leq_C)$, such that the following properties are satisfied:

- attribute names are unique along the generalization relation: if $c_1.p_1 : \tau_1[c_1']$ and $c_2.p_2 : \tau_2[c_2']$ are different property declarations in P and $c_1 \leq_C c_2$, then $p_1 \neq p_2$;
- association and composition names are unique: if d_1 and d_2 are the names of two different association or composition declarations in $M \cup A$, then $d_1 \neq d_2$;
- member end names are unique: if $p_1, \ldots, p_r$ are the member end names of an association declaration in A or a composition declaration in M, then $p_i \neq p_j$ for $1 \leq i \neq j \leq r$;[5]

[1] Cf. UML Superstructure Specification 2.4.1, p. 128; UML 2.5, p. 27.

[2] UML Superstructure Specification 2.4.1, p. 96; there does not seem to be default in UML 2.5.

[3] The member ends are ordered according to the UML Superstructure Specification 2.4.1, p. 29; UML 2.5, p. 206; hence they are represented in a tuple-like notation.

[4] Only binary association may show member ends that are properties not owned by the association (UML Superstructure Specification 2.4.1, p. 37; UML 2.5, p. 228). The property declarations induced by a more than binary association result in a query operation.

[5] In UML, member end names need not be unique. However, for (1) a simpler handling of selecting a particular member end in the sentences and avoiding the use of number selectors, and (2) making the notion of member ends "owned" by a class/data type, this constraint is added. An association declaration violating this uniqueness constraints can easily be transformed into an association declaration satisfying it by decorating member end names with the numbers $1, \ldots, r$.

- the type of a member end[6] owned by a class/data type coincides with its declarations as attribute: a property declaration $\mathbf{a}.p_i : \tau_i[c_i]$ yielded by a binary association $\mathbf{a} = a(p_1 : \tau_1[c_1], p_2 : \tau_2[c_2])$ is *owned by* $c_0 \in C$, if $c_{3-i} \leq_C c_0$ and there is an attribute declaration $c_0.p_i : \tau_i[c_i] \in P$; and similarly for property declarations yielded by composition declarations. (Note that by the uniqueness of attribute names along the generalization hierarchy only a single attribute with name p_i may exist.)

A *class/data type net morphism* $\sigma = (\gamma, \varphi, \alpha, \mu) : \Sigma = ((C, \leq_C), P, A, M) \to \mathrm{T} = ((D, \leq_D), Q, B, N)$ is given by

- a class/data type hierarchy map $\gamma : (C, \leq_C) \to (D, \leq_D)$;
- an attribute declaration map $\varphi : P \to Q$ such that if $\varphi(c.p : \tau[c']) = d.q : \tau'[d'] \in Q$, then $d = \gamma(c)$, $d' = \gamma(c')$, and $\tau = \tau'$;
- a query operation declaration map $\rho : O \to R$ such that if $\rho(c.q(x_1 : \tau_1[c_1], \ldots, x_r : \tau_r[c_r]) : \tau[c']) = d.r(x_1 : \tau'_1[d_1], \ldots, x_r : \tau'_r[d_r]) : \tau[d'] \in R$, then $d = \gamma(c)$, $d_i = \gamma(c_i)$, $d' = \gamma(c')$, $\tau'_i = \tau_i$ and $\tau = \tau'$;
- an association declaration map $\alpha : A \to B$ such that if $\alpha(a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r])) = b(q_1 : \tau'_1[d_1], \ldots, q_s : \tau'_s[d_s]) \in B$, then $r = s$ and $d_i = \gamma(c_i)$ and $\tau_i = \tau'_i$ for $1 \leq i \leq r$, and member ends owned by the association are mapped into owned member ends;
- a composition declaration map $\mu : M \to N$ such that if $\mu(m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2])) = n(q_1 : \mathsf{Set}[d_1], \blacklozenge q_2 : \tau'_2[d_2]) \in N$, then $d_1 = \gamma(c_1)$, $d_2 = \gamma(c_2)$, and $\tau_2 = \tau'_2$, and member ends owned by the composition are mapped into owned member ends.

Class/data type nets as objects and class/data type net morphisms as morphisms form the category of *class/data type nets*, denoted by Cl.

For the example in Fig. E.1 the class/data type net is

Classes/data types: Net, Station, Line, Connector, Unit, Track, Point, Linear, Boolean, UnlimitedNatural, Integer, Real, String

Generalizations: Point $\leq$ Unit, Linear $\leq$ Unit

Properties: Line.linear : Set[Boolean], Track.linear : Set[Boolean], Net.station : Set[Station], Net.line : Set[Line], Station.net : Set[Net], Station.unit : Set[Unit], Station.track : Set[Track], Line.net : Set[Net], Line.linear : Set[Linear], Connector.unit : Set[Unit], Unit.station : Set[Station], Unit.connector : Set[Connector], Track.station : Set[Station], Track.linear : Set[Linear], Linear.track : Set[Track], Linear.line : Set[Line]

Associations: l2l(line : Set[Line], linear : Set[Linear]), l2t(linear : Set[Linear], track : Set[Track]), c2u(connector : Set[Connector], unit : Set[Unit])

Compositions: n2s(net : Set[Net], ◆station : Set[Station]), n2l(net : Set[Net], ◆line : Set[Line]), s2u(station : Set[Station], ◆unit : Set[Unit]), s2t(station : Set[Station], ◆track : Set[Track])

Here all member ends are owned by class/data types.

E.3. Models

As stated above, models (in the sense of the term model defined in clause 4) of UML class diagrams are obtained via a translation to Common Logic.

For a classifier net $\Sigma = ((C, \leq_C), K, P, M, A)$, a Common Logic theory $CL(\Sigma)$ is defined consisting of:

- for $c \in C$, a predicate[7] $\mathsf{CL}(c)$, such that
  - $\mathsf{CL}(\mathsf{Boolean}) =$ `buml:Boolean`,
  - $\mathsf{CL}(\mathsf{String}) =$ `buml:String`,
  - $\mathsf{CL}(\mathsf{Integer}) =$ `buml:Integer`,
  - $\mathsf{CL}(\mathsf{UnlimitedNatural}) =$ `form:NaturalNumber`,
  - $\mathsf{CL}(\mathsf{Real}) =$ `buml:Real`,

[6] All member ends are instances of Property; UML Superstructure Specification 2.4.1, p. 36; UML 2.5, p. 206.
[7] Strictly speaking, this is just a name.

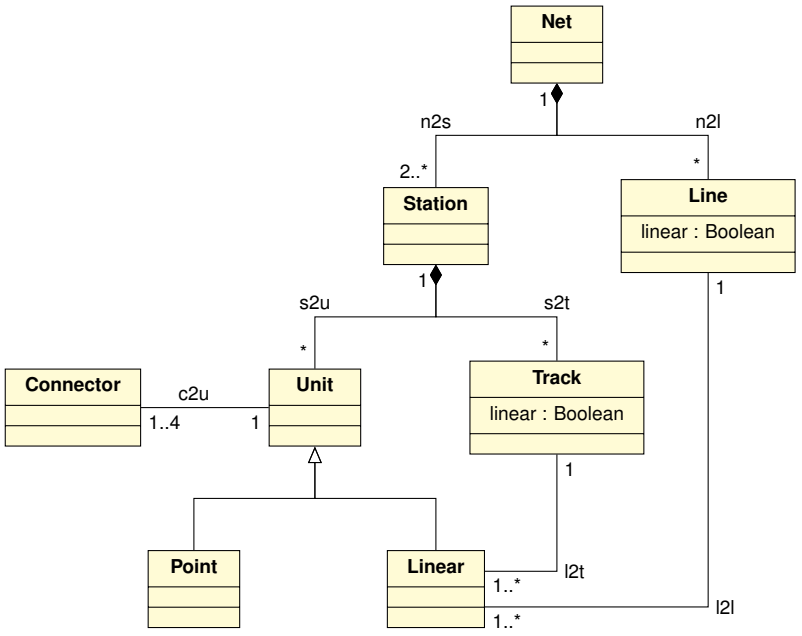

Figure E.1.: Sample UML class diagram.

  - CL(c) = c, if c is an enumeration type with values $k_1, \ldots, k_n$. In this case, additionally, the Common Logic theory is augmented by `(not (=` k_i $\cdots$ k_j`))` for $i \neq j$ and `(forall (x) (if (c x) (or (= x` k_1`)` $\cdots$ `(= x` k_n`))))`,
  - CL(List[c]) = `form:Sequence`,
  - CL(Set[c]) = `form:Set`,
  - CL(OrderedSet[c]) = `form:OrderedSet`,
  - CL(Bag[c]) = `form:Bag`,
  - CL(c) = c, if c a class name which is not one of the above.
- for each relation $c_1 \leq_C c_2$, an axiom `(forall (x) (if (`C_1 `x) (`C_2 `x)))`, where $C_1 = \mathsf{CL}(c_1)$, $C_2 = \mathsf{CL}(c_2)$,
- CL maps each attribute declaration $c.p : \tau[c'] \in P$ to a predicate CL($c.p$) and axioms stating type-correctness and functionality:
  - `(forall (x y) (if (c.p x y) (c x)))`
  - `(forall (x y) (if (c.p x y) (`$\tau[c']$ `y)))` [8]
  - `(forall (x)`
    `(if (c x) (exists (y) (c.p x y))))`
  - `(forall (x y z)`
    `(if (and (c.p x y) (c.p x z))`
    `(= y z)))`
- CL maps each query operation declaration $c.q(x_1 : \tau_1[c_1], \ldots, x_r : \tau_n[c_r]) : \tau[c'] \in O$ to a predicate CL($c.q$) and axioms stating type-correctness and functionality:
  - `(forall (x` x_1 x_2 $\cdots$ x_n `y) (if (c.q x` x_1 x_2 $\cdots$ x_n `y) (`c x`)))`
  - `(forall (x` x_1 x_2 $\cdots$ x_n `y) (if (c.q x` x_1 x_2 $\cdots$ x_n `y) (`$\tau_i[c_i]$ x_i`)))` for each $i = 1 \ldots n$,[9]
  - `(forall (x` x_1 x_2 $\cdots$ x_n `y) (if (c.q x` x_1 x_2 $\cdots$ x_n `y) (`$\tau[c']$ y`)))`
  - `(forall (x` x_1 x_2 $\cdots$ x_n `y z)`
    `(if (and (c.q x` x_1 x_2 $\cdots$ x_n `y) (c.q x` x_1 x_2 $\cdots$ x_n `z))`
    `(= y z)))`

  Query operations are modeled as partial functions: they may be undefined for certain arguments due to violation of multiplicity constraints.

[8] `(`$\tau[c]$ `x)` is an abbreviation of either (if τ is present)
`(and (`τ `x) (forall (m) (if (from:`τ`-member m x) (c' m))))`.
or (if τ is omitted) just `(c x)`.

[9] Note that the $\cdots$ here is meta notation, not a sequence marker.

- CL maps each composition declaration $m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]) \in M$ to a constant $\mathsf{CL}(m)$ and axioms stating that $\mathsf{CL}(m)$ is a finite binary relation represented as a sequence of pairs of the correct type:

```
(from:Sequence m)
(forall (p) (if (form:sequence-member p m)
                (and (form:Pair p) (c1 (form:first p)) (c2 (form:second p))))
```

  In case τ_2 is not present or $\tau_2 = \mathsf{Set}$, this is simplified to a binary relation directly represented as a binary predicate:
  `(forall (x y) (if (m x y) (and (`c_1` x) (`c_2` y))))`

- for any pair of composition declarations $m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]), m'(p_1' : \mathsf{Set}[c_1'], \blacklozenge p_2' : \tau_2'[c_2']) \in M$, an axiom stating "each instance has at most one owner":

```
(forall (o o' i)
        (if (and (form:sequence-member (form:pair o i) m)
                 (form:sequence-member (form:pair o' i) m'))
            (= o o')))
```

  In case `m` is represented in the simplified way, `(form:sequence-member (form:pair o i) m)` is replaced by `(m o i)`, and analogously for `m'`.

- CL maps each association declaration $a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r]) \in A$ to a predicate $\mathsf{CL}(a)$ and axioms stating that $\mathsf{CL}(a)$ is a finite relation represented as a sequence of tuples of the correct types (the latter again being represented as sequences)[10]:

```
(from:Sequence a)
(forall (t) (if (form:sequence-member t a)
         (exists (x_1 ··· x_r)
                 (and (c_1 x_1) ··· (c_r x_r)
(= t (form:sequence-insert x_1 (··· (form:sequence-insert
                 x_r form:empty-sequence)))))))
```

  In case that all the τ_i are omitted (or, equivalently, equal to Set), the representation is simplified to an n-ary predicate:
  `(forall (`x_1 x_2 ··· x_n`) (if (a `x_1 x_2 ··· x_n`) (and (`c_1 x_1`) ··· (`c_n x_n`))))`

- the interpretation of a member end of a binary association declaration owned by a class/data type coincides with the interpretation of the attribute: if for $i \in \{1, 2\}$, $\mathbf{a}.p_i : \tau_i[c_i]$ for $\mathbf{a} = a(p_1 : \tau_1[c_1], p_2 : \tau_2[c_2]) \in A$ is owned by $c \in C$ with $c.p_i : \tau_i[c_i] \in P$, then

```
(forall (o s)
    (if (c.p o s) (= s (form:seq2τ_i (form:selecti o a)))))
```

  If $\mathbf{a}$ is represented in simplified form, then instead the following is used

```
(forall (o s)
    (if (c.p o s) (forall (x) (iff (member x s) (a o x)))))
```

- the interpretation of a member end of a composition declaration owned by a class/data type coincides with the interpretation of the attribute: if for $i \in \{1, 2\}$, $\mathbf{m}.p : \tau_i[c_i]$ for $\mathbf{m} = m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]) \in M$ is owned by $c \in C$ with $c.p : \tau_i[c_i] \in P$, then `(forall (o s)`

```
    (if (c.p o s) (= s (form:seq2τ_i (form:selecti o m)))))
```

  Again, if $\mathbf{m}$ is represented in simplified form, then the following is used

```
(forall (o s)
    (if (c.p o s) (forall (x) (iff (member x s) (m o x)))))
```

It is straightforward to extend CL from signatures to signature morphisms.

Models. A Σ-model of the UML class diagram institution is just a $\mathsf{CL}(\Sigma)$-model in Common Logic. That is, the UML class diagram institution inherits models from Common Logic. Moreover, model reducts are inherited as well, using the action of CL on signature morphisms.

[10] Ignoring the annotations τ_i in the interpretation of an association is intentional, see OMG UML version 2.5 (ptc/2013-09-05) in section 11.5.3: "When one or more ends of the Association have isUnique =false, it is possible to have several links associating the same set of instances. In such a case, links carry an additional identifier apart from their end values. When one or more ends of the Association are ordered, links carry ordering information in addition to their end values." Similarly in UML Superstructure Specification 2.4.1, p. 37. The additional information required for links is covered by using sequences of tuples.

E.4. Sentences

The set of *multiplicity formulae Frm* is given by the following grammar:

$$
\begin{aligned}
\mathit{Frm} &::= \mathit{NumLiteral} \leq \mathit{FunExpr} \\
&\mid \mathit{FunExpr} \leq \mathit{NumLiteral} \\
\mathit{FunExpr} &::= \#\ \mathit{Attribute} \\
&\mid \#\ \mathit{Association}\ .\ \mathit{End} \\
&\mid \#\ \mathit{Composition}\ .\ \mathit{End} \\
&\mid \#\ \mathit{Operation}\ .\ \mathit{Param} \\
\mathit{Attribute} &::= \mathit{Classifier}\ .\ \mathit{End}{:}\mathit{Type} \\
\mathit{Association} &::= \mathit{Name}\ (\ \mathit{End} : \mathit{Type}(\ ,\ \mathit{End} : \mathit{Type})^*\ ) \\
\mathit{Composition} &::= \mathit{Name}\ (\ \mathit{End} : \mathsf{Set}\ [\ \mathit{Classifier}\ ], \blacklozenge \mathit{End} : \mathit{Type}\ ) \\
\mathit{Operation} &::= \mathit{Name}\ (\ (\ \mathit{NumLiteral} \leq \mathit{Param} \leq \mathit{NumLiteral}{:}\ \mathit{Type},\ )^*\ ) : \mathit{Type} \\
\mathit{Type} &::= \mathit{Annot}\ [\ \mathit{Classifier}\ ] \\
\mathit{Classifier} &::= \mathit{Name} \\
\mathit{End} &::= \mathit{Name} \\
\mathit{Param} &::= \mathit{Name} \\
\mathit{Annot} &::= \mathsf{OrderedSet} \mid \mathsf{Set} \mid \mathsf{Sequence} \mid \mathsf{Bag} \\
\mathit{NumLiteral} &::= 0 \mid 1 \mid \cdots
\end{aligned}
$$

where *Name* is a set of names and *NumLiteral* is assumed to be equipped with an appropriate function $[\![-]\!] : \mathit{NumLiteral} \to \mathbb{Z}$.

The set of Σ-*multiplicity constraints* $\mathit{Mult}(\Sigma)$ for a class/data type net Σ is given by the multiplicity formulae in *Frm* such that all mentioned elements of *Association* and *Composition* correspond to association declarations and composition declarations of Σ, respectively, and the member end name mentioned in the clauses of *FunExpr* occur in the mentioned association and composition, respectively.

The *translation* of a formula $\varphi \in \mathit{Mult}(\Sigma)$ along a class/data type net morphism σ, written as $\sigma(\varphi)$, is given by applying σ to associations, compositions, and member end names.

EXAMPLE For the example in Fig. E.1 there are the following multiplicity formulas:

$2 \leq \#\mathsf{n2s}(\mathsf{net} : \mathsf{Set}[\mathsf{Net}], \blacklozenge\mathsf{station} : \mathsf{Set}[\mathsf{Station}]).\mathsf{station}$

$\#\mathsf{n2s}(\mathsf{net} : \mathsf{Set}[\mathsf{Net}], \blacklozenge\mathsf{station} : \mathsf{Set}[\mathsf{Station}]).\mathsf{net} = 1$

$\#\mathsf{n2l}(\mathsf{net} : \mathsf{Set}[\mathsf{Net}], \blacklozenge\mathsf{line} : \mathsf{Set}[\mathsf{Line}]).\mathsf{net} = 1$

$\#\mathsf{s2u}(\mathsf{station} : \mathsf{Set}[\mathsf{Station}], \blacklozenge\mathsf{unit} : \mathsf{Set}[\mathsf{Unit}]).\mathsf{station} = 1$

$\#\mathsf{s2t}(\mathsf{station} : \mathsf{Set}[\mathsf{Station}], \blacklozenge\mathsf{track} : \mathsf{Set}[\mathsf{Track}]).\mathsf{station} = 1$

$1 \leq \#\mathsf{c2u}(\mathsf{connector} : \mathsf{Set}[\mathsf{Connector}], \mathsf{unit} : \mathsf{Set}[\mathsf{Unit}]).\mathsf{unit} \leq 4$

$\#\mathsf{c2u}(\mathsf{connector} : \mathsf{Set}[\mathsf{Connector}], \mathsf{unit} : \mathsf{Set}[\mathsf{Unit}]).\mathsf{connector} = 1$

$1 \leq \#\mathsf{l2t}(\mathsf{track} : \mathsf{Set}[\mathsf{Track}], \mathsf{linear} : \mathsf{Set}[\mathsf{Linear}]).\mathsf{track}$

$\#\mathsf{l2t}(\mathsf{track} : \mathsf{Set}[\mathsf{Track}], \mathsf{linear} : \mathsf{Set}[\mathsf{Linear}]).\mathsf{linear} = 1$

$1 \leq \#\mathsf{l2t}(\mathsf{line} : \mathsf{Set}[\mathsf{Line}], \mathsf{linear} : \mathsf{Set}[\mathsf{Linear}]).\mathsf{line}$

$\#\mathsf{l2l}(\mathsf{line} : \mathsf{Set}[\mathsf{Line}], \mathsf{linear} : \mathsf{Set}[\mathsf{Linear}]).\mathsf{linear} = 1$

"$x = y$" is an abbreviation for the two inequations "$x \leq y$" and "$y \leq x$". "$x \leq y \leq z$" is an abbreviation for the two inequations "$x \leq y$" and "$y \leq z$".

E.5. Satisfaction Relation

The satisfaction relation is inherited from Common Logic, using a translation $\mathsf{CL}(\_)$ of multiplicity formulas to Common Logic. That is, given a UML class and object diagram Σ, a multiplicity formula φ and a Σ-model M (the latter amounts to a $\mathsf{CL}(\Sigma)$-model M in Common Logic):

$$M \models_\Sigma \varphi \text{ iff } M \models_{\mathsf{CL}(\Sigma)} \mathsf{CL}(\varphi)$$

The translation of multiplicity formulas to Common Logic is as follows:

- $\mathsf{CL}(\ell \leq \#c.p : \tau[c']) =$

```
(forall (x y n)
    (if (and (c.p x y) (form:τ-size y n)) (buml:leq [[ℓ]] n))
```

- $\mathsf{CL}(\ell \leq \#a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r]).p_i =$
```
(forall (x_1 ··· x_{i-1} x_{i+1} ··· x_r)
     (if (and (c_1 x_1) ··· (c_{i-1} x_{i-1}) (c_{i+1} x_{i+1}) ··· (c_r x_r)
              (form:sequence-size
                   (form:n-select a i [x_1 ··· x_{i-1} x_{i+1} ··· x_r]) n))
         (buml:leq ⟦ℓ⟧ n)))
```
  If a is represented in simplified form, the following is used instead:
  $\mathsf{CL}(\ell \leq \#a(p_1 : \tau_1[c_1], \ldots, p_r : \tau_r[c_r]).p_i =$
```
(forall (x_1 ··· x_{i-1} x_{i+1} ··· x_r)
     (if (and (c_1 x_1) ··· (c_{i-1} x_{i-1}) (c_{i+1} x_{i+1}) ··· (c_r x_r))
         (exists (y_1 ··· y_⟦ℓ⟧)
            (and (not (= (y_1 y_2))) ··· (not (= (y_⟦ℓ⟧-1 y_⟦ℓ⟧)))
                 (a x_1 ··· x_{i-1} y_1 x_{i+1} ··· x_r)
                 ···
                 (a x_1 ··· x_{i-1} y_⟦ℓ⟧ x_{i+1} ··· x_r) ))))
```
- $\mathsf{CL}(\ell \leq \#m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]).p_i) =$
```
(forall (x)
     (if (and (c_{3-i} x) (form:τ-size (form:selecti x m) n))
         (buml:leq ⟦ℓ⟧ n))
```
  If m is represented in simplified form, the following is used instead:
  $\mathsf{CL}(\ell \leq \#m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]).p_1) =$
```
(forall (x)
     (if (c_2 x)
         (exists (y_1 ··· y_⟦ℓ⟧)
            (and (not (= (y_1 y_2))) ··· (not (= (y_⟦ℓ⟧-1 y_⟦ℓ⟧)))
                 (m y_1 x)
                 ···
                 (m y_⟦ℓ⟧ x) ))))
```
  $\mathsf{CL}(\ell \leq \#m(p_1 : \mathsf{Set}[c_1], \blacklozenge p_2 : \tau_2[c_2]).p_2) =$
```
(forall (x)
     (if (c_1 x)
         (exists (y_1 ··· y_⟦ℓ⟧)
            (and (not (= (y_1 y_2))) ··· (not (= (y_⟦ℓ⟧-1 y_⟦ℓ⟧)))
                 (m x y_1)
                 ···
                 (m x y_⟦ℓ⟧) ))))
```
- $\mathsf{CL}(\ell \leq \#c.q(\ell_1 \leq f_1 \leq \ell_1' : \tau_1[c_1], \ldots, \ell_k \leq f_k \leq \ell_k' : \tau_k[c_k]) : \tau[c']) =$
```
(forall (x x_1 x_2 ··· x_n)
     (if (and (c.q x x_1 x_2 ··· x_n y)
              (form:τ-size x_1 n_1)
              ...
              (form:τ-size x_k n_k)
              (form:τ-size y n)
              (buml:leq ⟦ℓ_1⟧ n_1)
              (buml:leq n_1 ⟦ℓ_1'⟧)
              ...
              (buml:leq ⟦ℓ_k⟧ n_k)
              (buml:leq n_k ⟦ℓ_k'⟧) )
         (buml:leq ⟦ℓ⟧ n)))
```

where $[\![-]\!] : \mathit{NumLit} \to \mathbb{Z}$ maps a numerical literal to an integer, and $[x_1 \cdots x_n]$ abbreviates `(form:sequence-insert` x_1 ··· `(form:sequence-insert` x_n `form:empty-sequence))`. The translation for $\mathit{FunExpr} \leq \mathit{NumLiteral}$ is analogous. In case of simplified representation, the existence of $[\![\ell]\!]$ distinct individuals would be replaced with a statement expressing that if $[\![\ell]\!] + 1$ individuals have the specified property, at least two of them must be equal.