Concept Spaces:
Towards a Unified Semantic Model

Paul C. Brown

Version 0.5

Revised 09 Feb 2012

Principles

There are a significant number of semantic models around, and the SIMF effort seeks to provide an approach towards at least bridging between them. It is the conjecture of this paper that there are at least three complicating factors of models that could be removed. These complicating factors are embedded in virtually all existing semantic models (if there are exceptions, I am not specifically aware of them). These factors are:

- Relationships are modeled differently from the entities they relate
- Instances are modeled differently from their types
- Models at one meta-level are treated as being distinct from models at other meta-levels

It is the position of this paper that these three distinctions are not fundamental, and that a semantic model that is not burdened by these distinctions will greatly simplify the task of satisfying the SIMF requirements.

Relationships and Entities Are Just Concepts

The distinction between entities and relationships is not fundamental. Both are concepts, and the distinction relates to the relative roles that one plays with respect to the other. There is nothing that prevents the same concept from playing the role of an entity and also playing the role of a relation. For example, the concept of Customer can be an entity in one context and a relationship between a person and a company in another. If relationships and entities are represented differently, mapping their semantics between these contexts gets complicated.

More fundamentally, there are concepts that just cannot be expressed if this distinction is enforced. Consider the notion of analogy: A is to B as C is to D. Expressing this idea involves three relationships: one relating A to C, the other relating C to D, and the third relating the “A is to B” relation to the “C is to D” relation. This is a relation between relations.

Figure 1 illustrates this difficulty in the UML Class notation, which makes this distinction between entities (classes) and relationships (associations). This notation allows the expression of the original ABRelation and CDRealtion as well as the mapping of A onto C and B onto D. What it cannot express is the mapping of the ABRelation onto the CDRelation. Of course, if you add association classes to these two relations, you can now show an association between these classes, but that’s a bit of a hack.

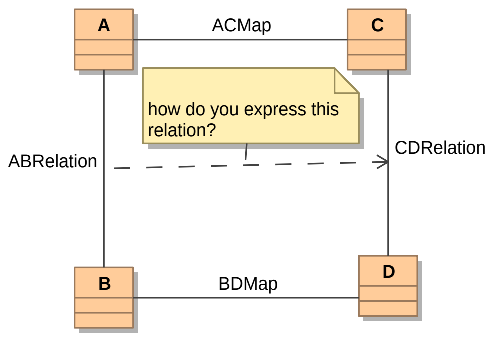

Figure 1: Representing a Relation in UML Class notation

These considerations lead us to the following principle:

> Unification Principle 1: A Relationship is a concept

Instances and Types

A type is a specification for a set or collection of entities. The idea of an instance is that it is an entity that conforms to the specification of the type. This does not necessarily imply that there is anything inherent about any given entity that makes it either a type or an instance. The notion of type characterizes refers to the *relative* role of one entity vs. another.

Consider the notion of a set of types. The members of this set are, on the one hand, instances of the abstract concept of type (hence their membership in the set), while on the other hand they are, themselves, types.

This consideration leads to the following principle:

> Unification Principle 2: A concept can be, simultaneously, both a type and an instance

Meta-Levels

It is common to treat the concepts and relationships associated with one level of abstraction as being somehow fundamentally different from those at another level of abstraction. However, maintaining such a distinction precludes a uniform representation of the elements of the different levels and mappings between them. Such representations are a requirement of SIMF. This leads to the following principle:

> Unification Principle 3: Meta-levels are just concepts, on a par with other concepts

Concept Spaces

We now define an abstract concept space that can be used to model concepts. It consists of a set of conceptual elements and three sets of binary mappings between conceptual elements: part-whole mappings, subclass mappings, and type-instance mappings (which turn out to be a subset of the subclass mappings).

Conceptual Element

A concept space provides a model for representing concepts as shown in Figure 2. A conceptual element is the building block from which we will shortly build the full representation of a concept. Principle 1 implies that it must be possible to have some structure to the conceptual element, here shown as parts. This structure allows a conceptual element to be an aggregate of other conceptual elements. Later we will see how a more conventional notion of relation can be constructed from this primitive.

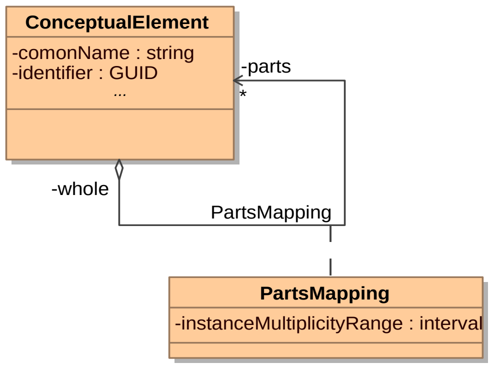

Figure 2: Concept Space Overview

The identifier of the core concept ensures a unique identity for each concept. The commonName is a human-readable label for the concept. It is present only for convenience and is not part of the formal semantics.

Associated with each part is a multiplicity range which indicates the allowed number of parts in the subclasses and instances of the conceptual element. This defaults to 0..*.[1]

A possible graphical notation is very close to the UML Class diagram notation (Figure 3). Notational differences (real, but minor) will emerge later on. Here we have four conceptual elements, X, A, B, and C. The structure of X has 0..n A's, exactly one B, and 1..n C's.

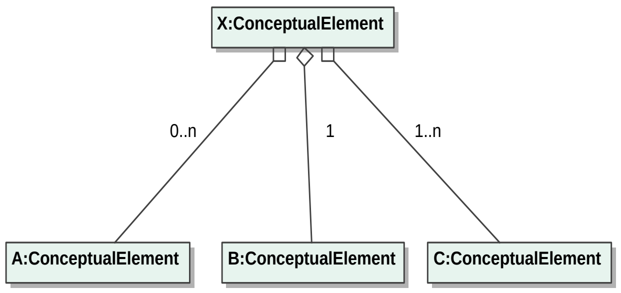

Figure 3: Graphical Notation for Part-Whole Structure

[1] Alternate representations of this constraint ought to be explored.

Is-A Mapping

One of the most common relationships between concepts is the is-a (subclass) relation. In the concept space, this mapping is defined as follows:

Every conceptual element is a subclass of itself.

A conceptual element may be a subclass of more than one element.

For a subclass of a conceptual element:

1. The subclass is mapped to the parent class (call this the subclass mapping)
2. For each part of the parent class there must be a corresponding part of the subclass with the additional constraint that there must be a subclass mapping between the parent part and the subclass part.
3. The allowed multiplicity of each subclass part must be a subinterval of the multiplicity of the corresponding parent's part.
4. Subclass can have additional parts

A graphical representation of the subclass relation is shown in Figure 4. The relation is here marked as <<abstract>>, since a refined version of this relation (without the <<abstract>> label) will be the one more commonly used in practice. This will be defined later.

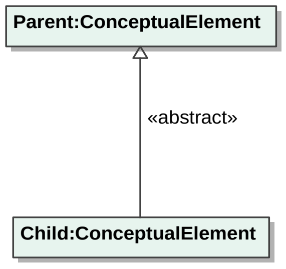

Figure 4: Abstract Subclass Notation

Formalization

Let a concept space (CS) be a tuple:[2]

$$CS = \{E, P, S\}$$

Such that:

E is a set of conceptual elements **e**.

P is a partial mapping E◊E indicating part-whole relationships. The expression e_1◊e_2 indicates that e_1 is a part of e_2. The full specification of the mapping is actually a triple {part, whole,

[2] There is likely a simpler formulation for the concept space from which this formulation can be derived.

interval}[3] where the interval defines the minimum and maximum number of parts allowed when this relationship is realized in a subclass or instance. Part relationships are not unique: an element can be a part of an arbitrary number of elements (including themselves).

S is a subclass mapping $E \triangleright E$ between conceptual elements. The expression $e_1 \triangleright e_2$ indicates that e_1 is a subclass of e_2. Subclass relationships are not unique: an element can be a subclass of an arbitrary number of other elements.

Every element is a subclass of itself:

$$\forall e_i \in E,\ e_i \triangleright e_i$$

There is a constraint on the subclass mapping with respect to the part-whole structure. If an element is a subclass of an element with parts, then the part elements of the subclass must themselves be subclasses of the corresponding parts of the parent class.

$$\forall e_a, e_b, e_x, e_y \in E, \text{ if } e_b \Diamond e_a \text{ then } e_x \triangleright e_a \text{ iff } \exists\ e_y \text{ such that } e_y \Diamond e_x \wedge e_y \triangleright e_b$$

There is a constraint on the intervals associated with the subclass's part mappings. The min of the subclassed mapping interval must be greater than or equal to that of the parent mapping interval, and the max must be less than or equal to the parent max.

In algebraic terms, the subclass relation $\triangleright$ is reflexive, anti-symmetric, and transitive. It defines a partial ordering on the set of conceptual elements.

The set operation of union is well defined in this space – it encompasses the notion of having multiple parent classes.

It appears as if the set operation of intersection is not well defined in this space. The part-whole mapping combined with the subclass constraint that all parts of the parent class must be present in the child class seems to prohibit the definition of a "bottom" or null-set element in the set of conceptual elements.

Stored Representation

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://simf.omg.org/ConceptSpace"
xmlns:tns="http://simf.omg.org/ConceptSpace" elementFormDefault="qualified">

    <simpleType name="ConceptURI">
      <restriction base="string"></restriction>
    </simpleType>

    <complexType name="Interval">
      <sequence>
            <element name="min" type="int"></element>
            <element name="max" type="int"></element>
```

[3] Alternate formulations of allowed instance multiplicity need to be explored.

```
      </sequence>
    </complexType>

    <complexType name="PartMapping">
      <sequence>
            <element name="part" type="tns:ConceptURI"></element>
            <element name="interval" type="tns:Interval"></element>
      </sequence>
    </complexType>

    <complexType name="ConceptElement">
      <sequence>
            <element name="qualifiedName" type="tns:ConceptURI"></element>
            <element name="commonName" type="string"></element>
            <element name="parts" type="tns:PartMapping"
maxOccurs="unbounded" minOccurs="0"></element>
      </sequence>
    </complexType>

    <complexType name="SubclassMapping">
      <sequence>
            <element name="parentClass" type="tns:ConceptURI"></element>
            <element name="subclass" type="tns:ConceptURI"></element>
      </sequence>
    </complexType>

  </schema>
```

A Populated Concept Space

Now that the idea of a concept space has been defined, it is time to create and populate one. Let CS_0 be the concept space we are defining. Where needed for clarification, subscripts will be used to indicate that the concept space parts belong to this particular concept space.

Concept

Now define the basic concept of a Concept in this space, which is a conceptual element with no parts:

$e_{Concept} \in E_0$

Graphically, we have:

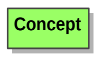

Figure 5: Concept in the Concept Space

This basic concept has no parts.

Role

Now we want to add some structure to this space. First, we define a Role, which will become a part of a concept called a Pattern. A Role is a subclass of Concept that has exactly one part, which is itself a Concept. The allowed instance multiplicity of the part is 1:

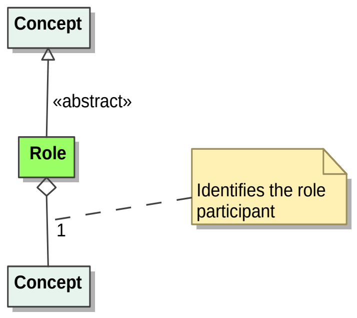

Figure 6: Role in the Concept Space

For convenience, we will refer to the concept that is part of the role as the *role participant*.

Pattern

Next we define our first complex concept, a Pattern (Figure 7). A pattern is a concept whose parts are Roles. For completeness, the sub-structure of a Role is shown as well. Thus a pattern has a number of Roles, with each Role having one participant which is a concept.

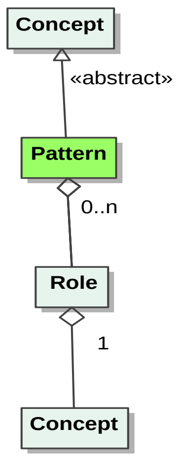

Figure 7: Pattern in the Concept Space

Note that because Patterns and Roles are concepts, they can be participants in Patterns.

Commentary: Because patterns identify the roles, it would be easy to associate logical statements with the patterns that amplify the relative roles. These could be constraints or computations.

Generalization Pattern

Next we define a pattern called Generalization using the concepts we have already defined (Figure 8). First, we define two subclasses of Role, the ParentClass and the ChildClass. Then we define Generalization as a pattern with two parts: ParentClass and ChildClass, each with an allowed multiplicity of one. For completeness, all of the abstract parent classes have been shown. As required by the concept space rules, the parts of the Generalization are abstract subclasses of Role. The use of subclassing and the use of two different subclasses (ParentClass and ChildClass) allows us to distinguish between the ends of the Generalization relation.

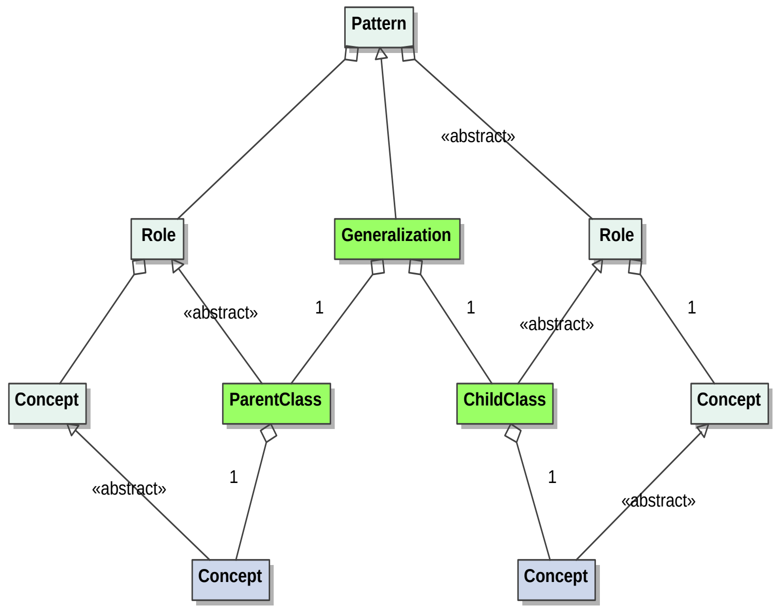

Figure 8: Generalization Relation Defined in the Concept Space

The Generalization relation is, itself, a pattern. We use it to declare that a generalization relation exists, for example, between two classes, X and Y (Figure 9). Here YisaX is a subclass of Generalization, and ParentX and ParentY are subclasses of ParentClass and ChildClass, respectively. If we add a further restriction that YisaX cannot be a child in any other subclass relation, then all the restrictions required to state that YisaX is an instance of Generalization have been met.

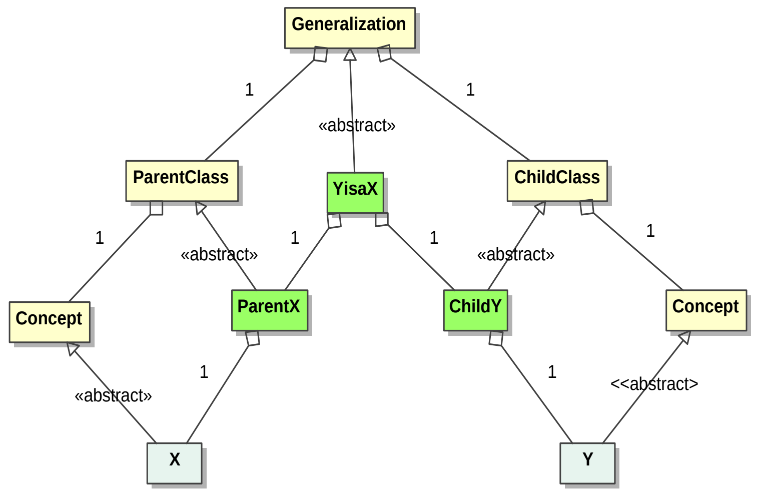

Figure 9: X is a Y Generalization

This pattern is instantiated frequently. In most cases, when we instantiate the Generalization relation, there is no need to see the details of the pattern structure. Therefore we define the shorthand graphic notation for instantiating generalization shown in Figure 10. Using this notation indicates the definition of the three concepts: YisaX, ParentX, and ChildY. If there is need to explicitly show these concepts (e.g. to define a mapping relation in which one of these concepts plays a role) then the full notation can be used).

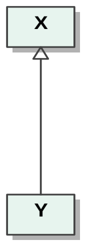

Figure 10: Shorthand Graphic Notation for Instantiating Generalization

Note that using this notation requires the use of an implicit naming scheme for the elements of the generalization instance.

Graphical Notation for Binary Associations

A binary association is just a pattern with named roles (Figure 11). Note that with the use of the Generalization notation (no <<abstract>> on the line) the parts of the pattern (recursively) do not have to be explicitly shown as they are implicit in the notation.

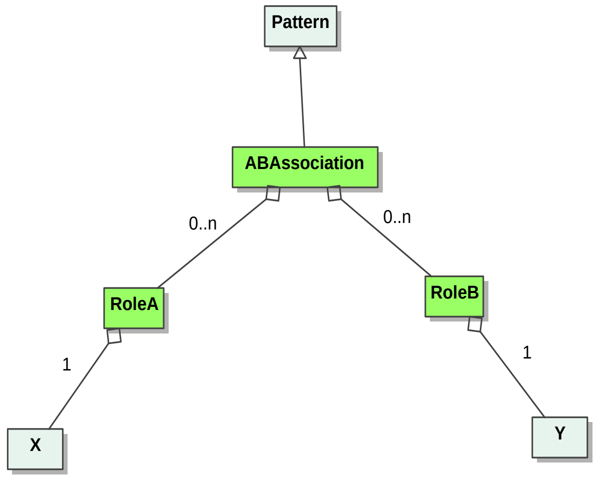

Figure 11: Example Binary Association

This use of patterns is so common that we give a shorthand graphical notation for it:

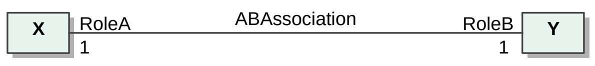

Figure 12: Graphical Shorthand for Binary Associations

Set

A set is just a subclass of Pattern as shown in Figure 13

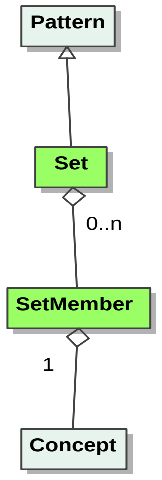

Figure 13: Set Definition

A PeopleSet (set of people) is shown in Figure 14.

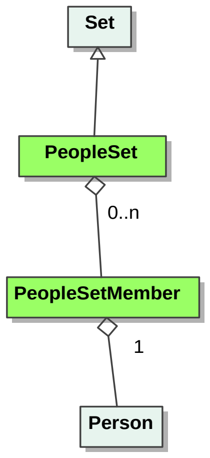

Figure 14: A PeopleSet

Ordered Set

To define an ordered set, we must first define an OrderingRelation, which is a kind of Pattern (Figure 15).

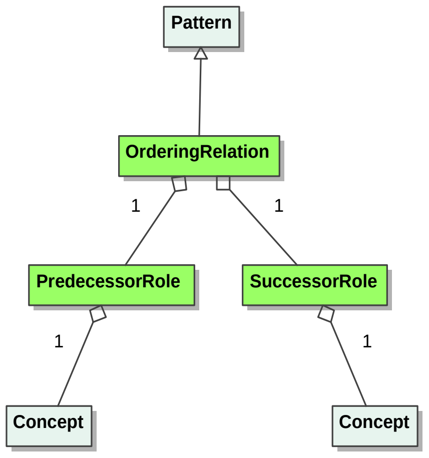

Figure 15: Ordering Relation

Now we define a subclass of ordering relation that is specific to sets (Figure 16).

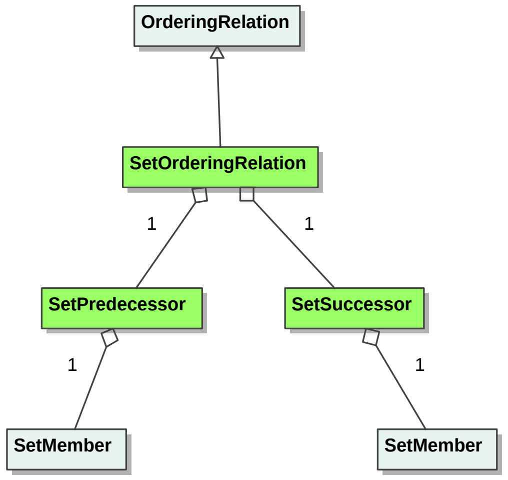

Figure 16: Set Ordering Relation

Finally, we define the ordered set as a subclass of Set with an additional part that defines the set ordering relations (Figure 17). Note that some additional constraints are required. One is that the referents for the SetPredecessor and SetSuccessor roles must be part of the ordered set that owns the SetOrderingRelation. The other is on the ordering relations themselves, regarding cycles, total or partial ordering, etc.

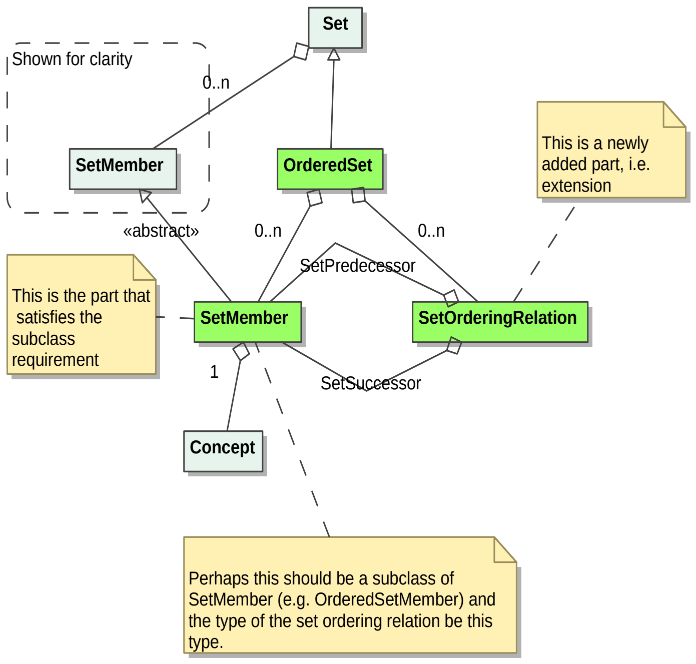
Set
Shown for clarity
0..n
SetMember
OrderedSet
This is a newly added part, i.e. extension
«abstract»
0..n
0..n
SetPredecessor
This is the part that satisfies the subclass requirement
SetMember
SetOrderingRelation
1
SetSuccessor
Concept
Perhaps this should be a subclass of SetMember (e.g. OrderedSetMember) and the type of the set ordering relation be this type.

Figure 17: Ordered Set

Ordered Pattern

First define a role ordering relation:

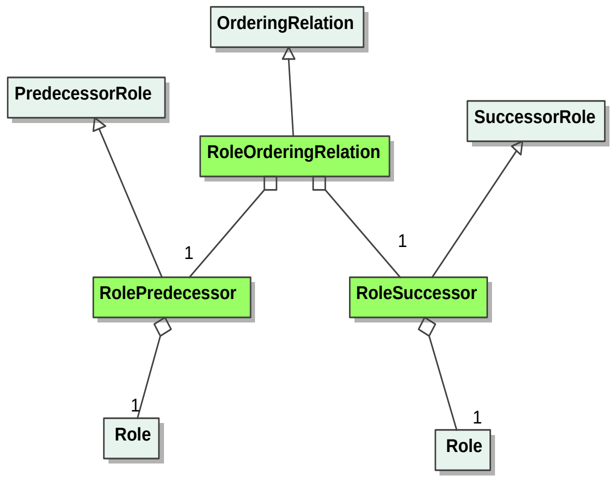

Figure 18: Role Ordering Relation

Now the ordered pattern

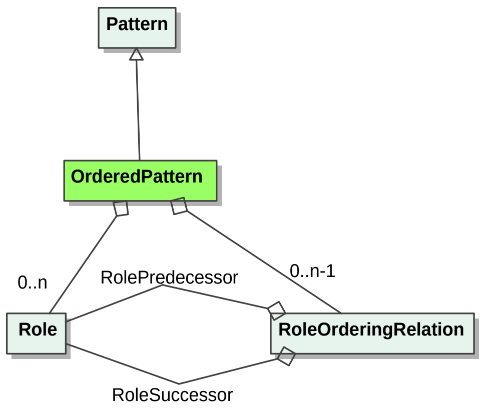

Figure 19: Ordered Pattern

Defining Concept Space Within a Concept Space

Conceptual Element Set

First we define the Conceptual Element Set:

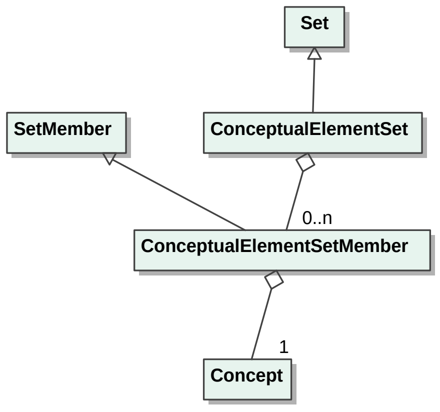

Figure 20: Conceptual Element Set

Interval

Next, we define the notion of an interval as a subclass of pattern:

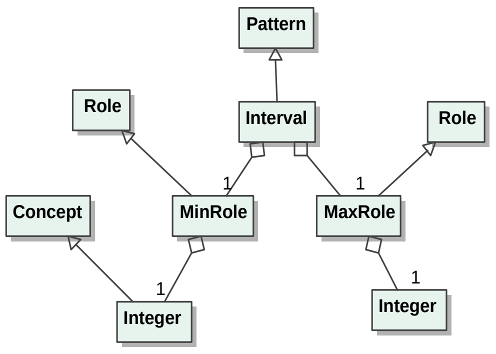

Figure 21: Interval as a Subclass of Pattern

Part Relation

Next, we define the part relation as a relationship between ConceptualElementSetMember roles.

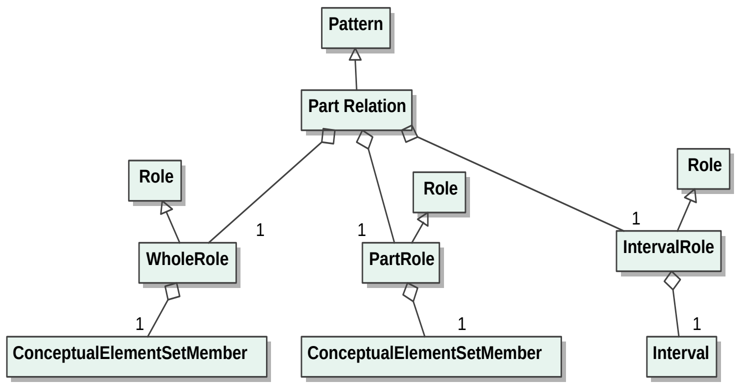

Figure 22: Part Relation

Part Relation Set

We also define the Part Relation Set:

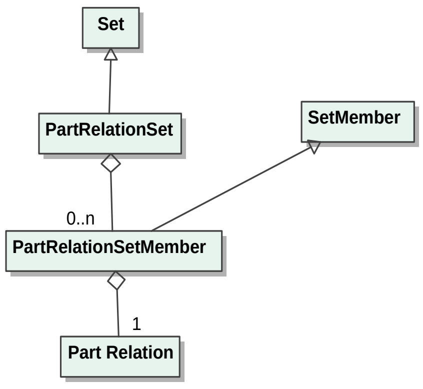

Figure 23: Part Relation Set

Parent-Child Relation

Next, we define a parent-child relation

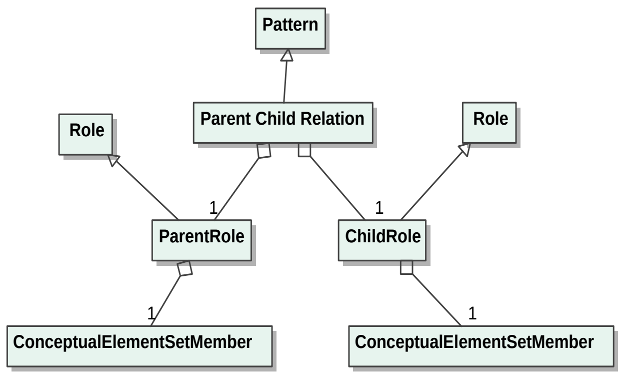

Figure 24: Parent-Child Relation

Parent-Child Relation Set

And the corresponding Parent-Child Relation Set:

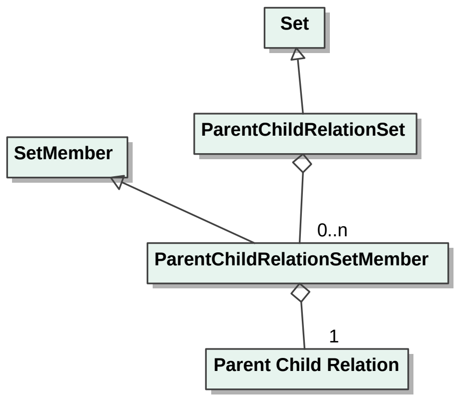

Figure 25: Parent-Child Relation Set

Concept Space

Finally, we assemble the finished Concept Space:

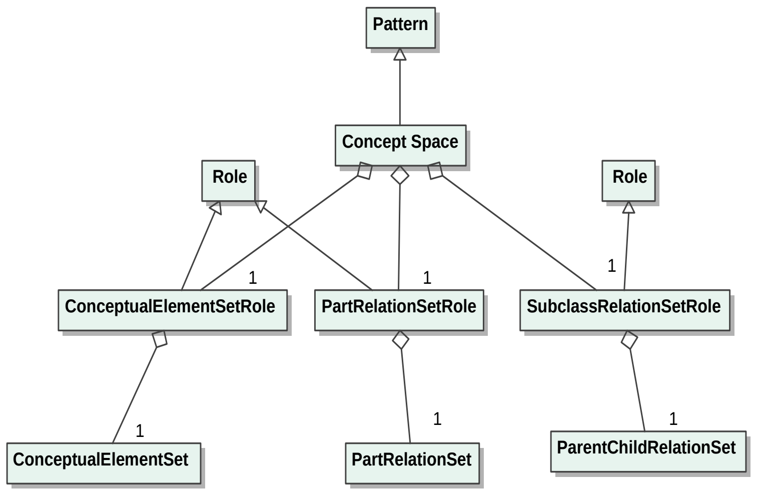

Figure 26: Concept Space Representation

Multiple Concept Spaces

The populated concept space described in the previous section forms an ordered set with the Concept being the root element of the ordering. It is expected (though not required) that most individual concept spaces will have this structure.

When we want to relate (map) concept spaces together, we can define a new concept space that is the union of the two concept spaces. Now we have a concept space with two sets of elements with two distinct root elements and no relations between them. However, by the rules of the concept space, we can define new concepts (e.g. mappings) that happen to have some parts from one of these sets and other parts from the other set. To make this simpler, however, we can also define a new abstract root concept and define both of the previous roots to be subclasses of this new root.