Technical Report LPT-2008-24

A Meta Model for the Design of Domain Ontologies

J. Morbach, A. Wiesner, W. Marquardt

July 2007

Enquiries should be addressed to:

RWTH Aachen University
Aachener Verfahrenstechnik
Process Systems Engineering
52056 Aachen

Tel.: +49 (0) 241 80 - 94668
Fax: +49 (0) 241 80 - 92326
E-Mail: secretary.pt@avt.rwth-aachen.de

Contents

List of Figures

1. Introduction

This document gives an informal specification of the *Meta Model*, an OWL-based ontology that has been designed to guide the development of the domain ontology OntoCAPE 2.0 (Morbach et al., 2007). By definition, a meta model is "a design framework, that describes the basic model elements and the relationships between the model elements as well as their semantics. This framework also defines rules for the use […] of model elements and relationships" (Ferstl & Sinz, 2001, p. 86). In accordance with this definition, the Meta Model introduces fundamental modeling concepts and design rules for the construction of the OntoCAPE ontology.

Although the Meta Model has been developed specifically for OntoCAPE, it is in fact domain-independent and can thus be reused to guide the construction of other OWL-based ontologies. Currently, the Meta Model constitutes the basic framework of three further ontologies, named *Document Model* (Morbach et al. 2008,), *Process Ontology* (Eggersmann et al. 2008), and *Decision Ontology* (Theißen and Marquardt 2008). ,. In the following, such ontologies which are derived from the Meta Model will be referred to as *target ontologies*.

Note that the term 'meta model' is used with two different meanings in the literature; for their differentiation, Atkinson & Kühne (2002) coined the terms *physical metalevel* and *logical metalevel*: The *physical metalevel* defines the concepts and mechanisms of the modeling language, whereas the *logical metalevel* guides the development of the target ontology. According to this definition, the Meta Model described herein corresponds to the *logical metalevel*; the *physical metalevel* is given by the formal definition of the OWL language (OWL, 2002).

The Meta Model is defined on top of the target ontology. As shown in Fig. 1, it is partitioned into the partial models[1] **fundamental_concepts**, **mereology**, **topology**, and **data_structures**. While both **mereology** and **topology** contain only a single ontology module[2], **data_structures** comprises five: *array*, *linked_list*, *multiset*, *binary_tree*, and *loop*. The module *meta_model* includes[3] all these ontology modules, thus assembling the ontological definitions of the Meta Model. The module *meta_model* is, in turn, included by the top-level module of the target ontology (shown here is the module *system*, which resides on the Upper Level of OntoCAPE). That way, the concepts defined in the Meta Model are available in the target ontology.

The Meta Model is not a genuine part of the target ontology. Rather, its function is (a) to explicitly represent the underlying design principles and (b) to establish some common standards for the design and organization of the target ontology. Thus, the Meta Model supports ontology engineering and ensures a consistent modeling style across the target ontology. These goals are achieved by means of two different mechanisms: the introduction of *fundamental concepts*, and the definition of *design patterns*..:

Fundamental concepts are fundamental classes and relations from which all the root terms of the target ontology can be derived (either directly or indirectly). By linking a root term of the target ontology to a fundamental concept, its role within the ontology is characterized. That way, a user or a software program is advised how to properly treat that particular root term and the classes or relations derived from it: For instance, all classes in the target ontology that are derived from the fundamental concept '*relation class*' are auxiliary constructs for the representation of n-ary relations. Since instances of such classes do not need to be given meaningful names (cf. Noy and Rector, 2006), a user or an

[1] Ontology modules assemble a number of classes that cover a common topic as well as the relations describing the interactions between the classes and the constraints defined on them.

[2] Ontology modules that address closely related topics are grouped into partial models. The partial models constitute a coarse categorization of the domain.

[3] Inclusion means that if module A includes module B, the ontological assertions provided by B are included in A. Inclusion is transitive, that is, if module B includes another module C, the ontological assertions specified in C will also be valid in A.

intelligent software program can conclude that such instances can be labeled automatically, according to some standardized naming convention.

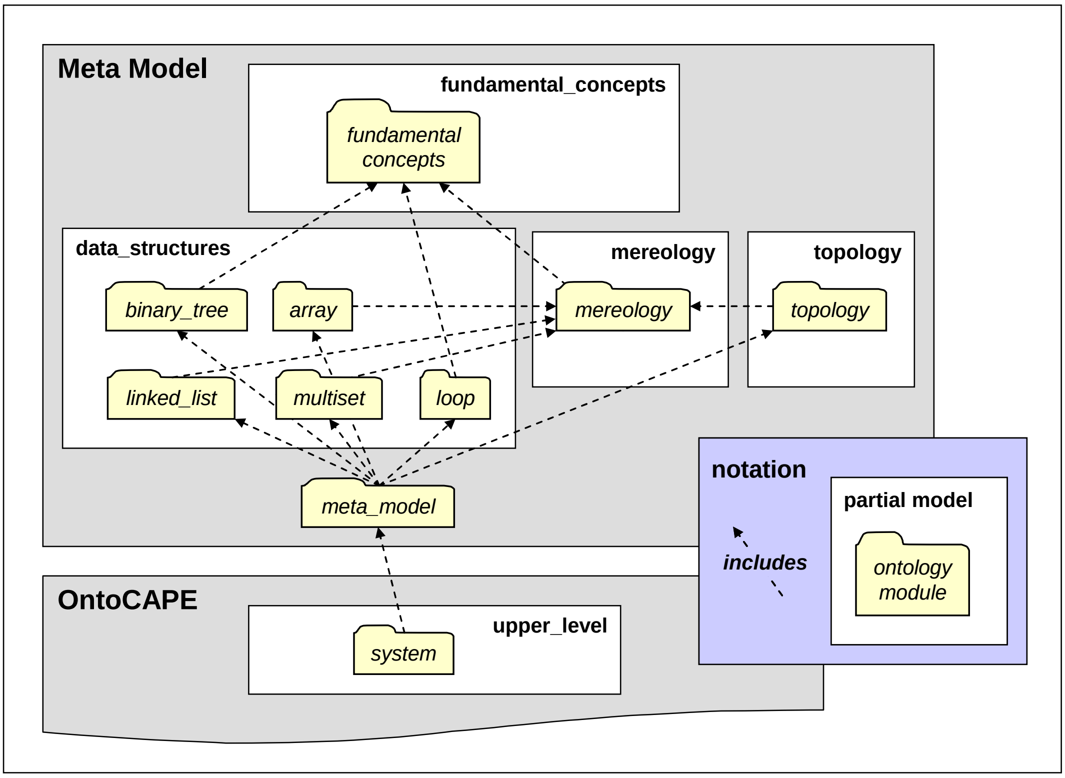

Fig. 1: Relations between the ontology modules of the Meta Model and those of OntoCAPE

Conceptually, the linkage between the ontological terms of the Meta Model and those of the target ontology should be established by means of instantiation. However, while the OWL modeling language supports such metamodeling (i.e., instantiation across multiple levels) in principle, it is at the cost of loosing scalability and compatibility with DL reasoners (Smith et al., 2004). Therefore, it is not advisable to interlink the Meta Model and the target ontology via instantiation. Hence, the linkage between OntoCAPE and the Meta Model is currently realized via specialization.

A design pattern[4] is a template formed by a set of classes, interconnecting relations, and constraining axioms; it establishes a best-practice solution to a recurring design problem. That way, patterns encourage a consistent, uniform design throughout the target ontology. A typical example is the representation of mereologic relations (part-whole relations): A design pattern defines a standard way of modeling this relation type, which is adopted by all ontology modules of the target ontology.

It is worthwhile noting that the design patterns of the Meta Model are implementation-dependent; that is, they constitute a best-practice solution only for an ontology that is *represented in OWL* and *processed by a customary DL reasoner*. For instance, the abovementioned mereology pattern states how to implement part-whole relations in OWL such that they efficiently scale for large amounts of instance data. Yet if the part-whole relations were implemented in a different modeling language, or if the ontology was processed by a non-standard reasoner, the mereology pattern might not constitute the best possible solution.

To apply a design pattern in the target ontology, we have adopted a rather pragmatic approach that was suggested by Clark et al. (2000): The classes, relations, and axioms that constitute the design pattern in the Meta Model are simply redefined in the target ontology. Practically, this is realized by (1) copying

---

[4] Design patterns are popular in software engineering (e.g., Gamma et al. 1995), where they specify general solutions for recurring problems. In ontology engineering, the term 'knowledge pattern' (Clark et al. 2000) is sometimes used instead.

the axiomatic definitions of the design pattern into the target ontology and (2) renaming the non-logical symbols within these expressions (i.e., the classes and relations); additionally, the duplicated classes and relations may be linked to their respective originals in the Meta Model, but this is not mandatory (cf. the discussion in the subsequent paragraph). The advantage of this approach is its flexibility: Often, only a selected part of a theory is to be transferred (i.e., there may be symbols in the pattern that have no counterpart in the target ontology) – either because only the transferred part is needed in the target ontology, or because the omitted part is to be implemented differently from the Meta Model. For this purpose, the transfer of the design pattern via rigorous specialization (or instantiation) would not be flexible enough, as it would call for copying the entire pattern in an "all-or-nothing" fashion. By contrast, the selected approach allows for deviations and variants. Clark et al. (2000) stress that this is architecturally significant, as well, since the approach supports a better modularization of the target ontology.

While the Meta Model has proven to be highly useful during the design of the target ontology and its refinement to a knowledge base, it becomes less relevant once the refined ontology is actually used as a knowledge base of some application; in some cases it might even be harmful, as the additional, abstract concepts of the Meta Model could confuse the user. Thus, the interconnectivity between target ontology and the Meta Model should be kept at a minimum, such that the two ontologies can be separated easily if desired. Therefore, the classes and relations defined in Meta Model are not to be used directly within the target ontology; rather, they are redefined by copying the respective concepts in the target ontology, as explained above. The duplicates may subsequently be linked to the originals in the Meta Model[5]. That way, only the links to the Meta Model need to be disconnected if a stand-alone usage of the target ontology is desired. Particularly for relations, the principle of overloading[6] is often applied: that is, the relation in the target ontology receives the same name as the original relation in the Meta Model. That way, a relation with the same name can be implemented in different ontology modules, however each time possibly with a different range and domain, and thus with a different semantics.

The remainder of this document is organized as follows: each ontology module is first described in natural language and by means of UML-like class diagrams, which show the main interrelations between classes and relations, including their hierarchical organization. Some application examples may be provided. Subsequently, the usage of the ontology module is explained, and some competency questions[7] are presented that characterize the functionality of the ontology module. Finally, the individual concepts (classes and relations) of the respective ontology module are described in natural language (see Appendix A for a description of the applied documentation format).

The Meta Model is completely implemented in OWL. The ontology modules are realized through namespaces, each of which is stored in a single OWL file. The partial models are implemented as directories. A directory may contain some additional OWL files, the names of which start with the prefix "*example_*"; these files illustrate the usage of the Meta Model by means of exemplary applications.

---

[5] Linking a duplicate to the original through specialization has proven valuable during ontology design, since it allows checking the consistency of the duplicate against the original by means of a reasoner.

[6] The idea of overloading originates from computer science; originally, it means that multiple functions, taking different types of input, can be defined with the same name.

[7] The formulation of competency questions forms part of the methodology for ontology engineering that was first suggested by Grüninger and Fox (1995) and later explicated in detail by Uschold and Grüninger (1996). Informal competency questions are questions in natural language that specify the requirements for the ontology to be developed, thus determining its scope. Once the ontology is implemented in a formal language, the competency questions are formalized in a machine-interpretable language such that they can be evaluated by a reasoner. By running the formal competency questions against the ontology (or rather against a set of test data instantiated from the ontology), it can be verified that the ontology complies with the specifications.

The reasoner RacerPro (Racer Systems, 2006) has been used to validate the consistency of the Meta Model.

Notation Conventions

Classes and relations of the Meta Model are named according to the CamelCase[8] naming convention: UpperCamelCase notation is used to denote identifiers of classes, while relation identifiers are represented in lowerCamelCase notation. No particular naming convention is followed for identifiers of individuals (i.e., instances of classes).

In this document, class identifiers are highlighted by *italicized sans-serif font*; for better readability, the UpperCamelCase notation is not applied in the text, but the individual words that constitute the class identifiers are written separately and in lowercase (e.g., *class identifier*). If relations are explicitly referred to in the text, they are written in lowerCamelCase notation and are additionally highlighted by sans-serif font. Individuals are accentuated by **bold sans-serif font**. Partial models are denoted **bold serif font**, *italicized serif font* refers to ontology modules.

In figures, a graphical notation in the style of UML class diagrams is used; the basic elements are depicted in Fig. 2. Grey shaded boxes represent *classes*, white boxes represent *individuals*. *Attributes* are denoted by grey shaded boxes with dashed boundary lines, *attribute values* by white boxes with dashed boundary lines. *Specialization* is depicted through a solid line with a solid arrowhead that points from the subclass to the superclass. A dashed line with an open arrowhead denotes *instantiation*. *Binary relations* are depicted though solid lines. Three basic relation types are distinguished: a line with one open arrowhead represents a *unidirectional* relation; a line with two open arrowheads represents a *symmetric* relation; a line without any arrowheads represents a bidirectional relation[9]. Finally, graphic elements for two special types of relation are introduced: an *aggregation* relation is depicted through a line with a white diamond-shaped arrowhead pointing towards the aggregate class. Similarly, a black diamond-shaped arrowhead indicates a *composition* relation.

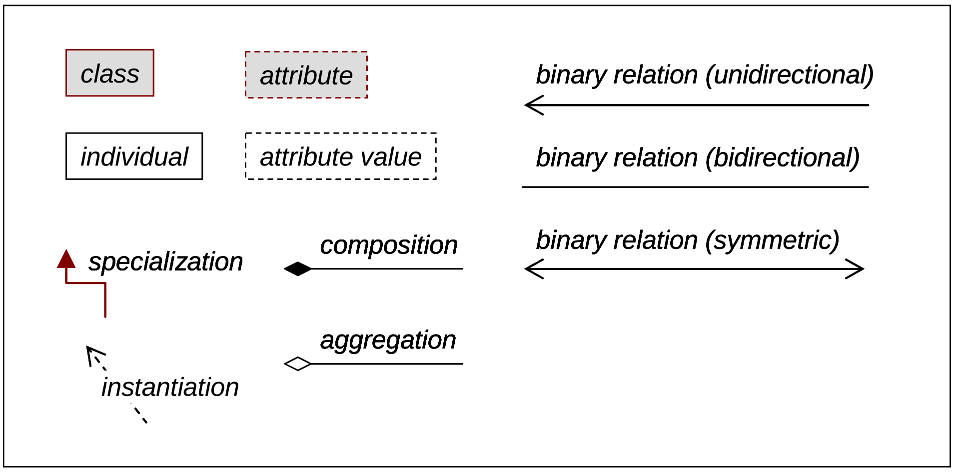

Fig. 2: Basic elements of graphical notation

[8] CamelCase is the practice of writing compound words joined without spaces; each word is capitalized within the compound. While the UpperCamelCase notation also capitalizes the initial letter of the compound, the lowerCamelCase notation leaves the first letter in lowercase.

[9] In OWL, a bidirectional relation is modeled through a unidirectional relation and its inverse.

2. Fundamental concepts

The ontology module *fundamental concepts* forms the basis of the Meta Model. It introduces *meta root concepts* and their refinements. A *root concept* is a class or a relation without ancestors. Accordingly, *meta root concepts* are the root classes and relations in the Meta Model. They form the topmost layer of the concept hierarchy; all other classes and relations – in the Meta Model as well as in the target ontology – can be derived from the meta root terms by specialization. As can be seen from Fig. 3, three root classes are defined in the Meta Model: *object*, *relation class*, and *feature space*.

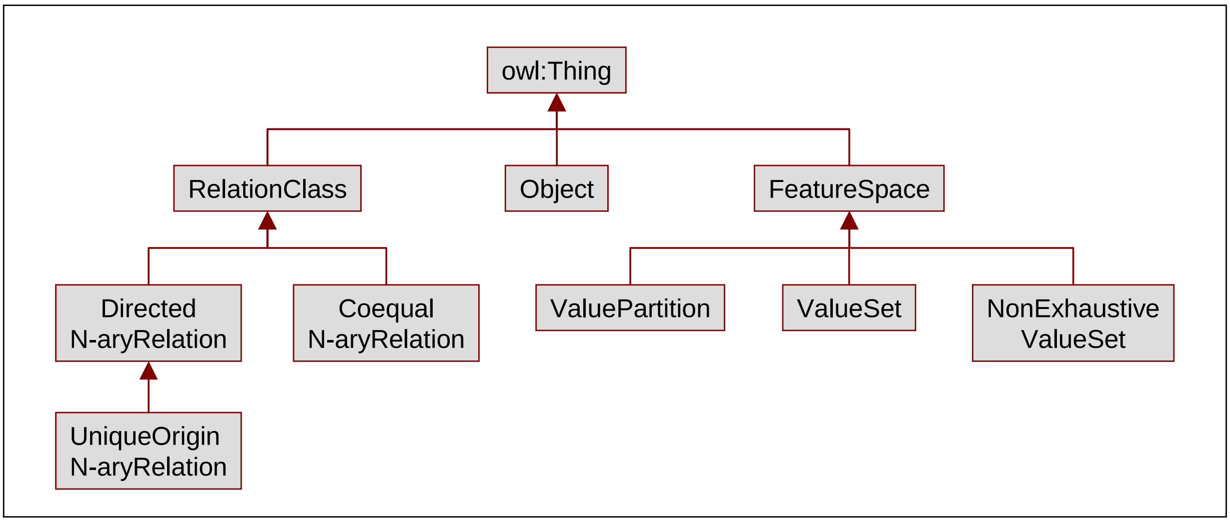

Fig. 3: Fundamental classes

Object is a generic class that subsumes all "self-standing" (Rector, 2003) entities – whether physical or abstract – that exist in an application domain. In conjunction with the *object* class, the root relation interObjectRelation is introduced, which subsumes all types of binary relations that exist between *objects*.

An *object* can be characterized by means of descriptive features. A *feature space* defines the range of values that a feature can take (Rector, 2005). Three specializations of *feature space* are distinguished, which reflect different ways to define the values of a particular feature: A *value partition* describes the feature values by partitioning a class into disjoint subclasses. In contrast, a *value set* defines the values as an enumeration of individuals. While a *value set* has a fixed number of individuals, the number of individuals is not predetermined in a *non-exhaustive value set*.

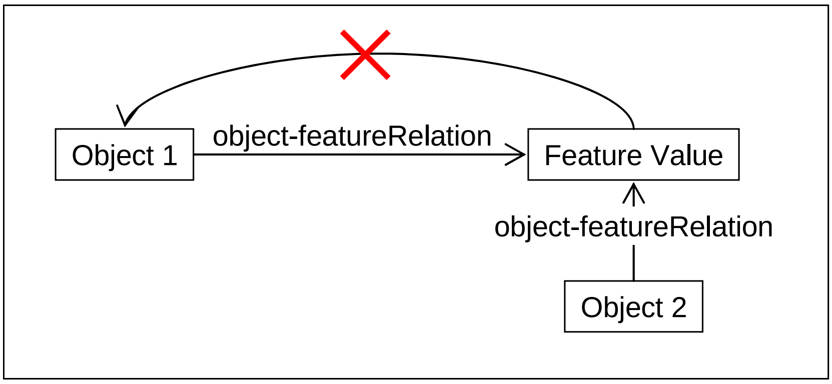

Fig. 4: A feature value may be assigned to different objects

A feature value (i.e., an instance of *feature space*) can be assigned to an *object* via the unidirectional object-featureRelation. Feature values are independent of a particular *object*; thus, a feature value may be assigned to different *objects*, as indicated in Fig. 4. Relations that refer from a feature value to an *object* are not permitted – such a relation would imply that the individual represents the feature of a particular *object*, i.e., the feature value would loose its independence. Therefore, a feature value cannot be the origin of an unidirectional object-feature relation, and there must not be any bidirectional relations between *object* and *feature space*, either.

The OWL language merely provides language primitives for *binary* relations; there is no predefined language element for an *n-ary* relation that could link three or more individuals. Also, binary relations cannot be characterized through attributes. To overcome these limitations, the concept of a *relation class* is introduced. A *relation class* may be used to

- represent n-ary relations between individuals of type *object* or *feature space*, and/or
- characterize a relation between two or more individuals by some additional attribute.

These two application cases are depicted in Fig. 5.

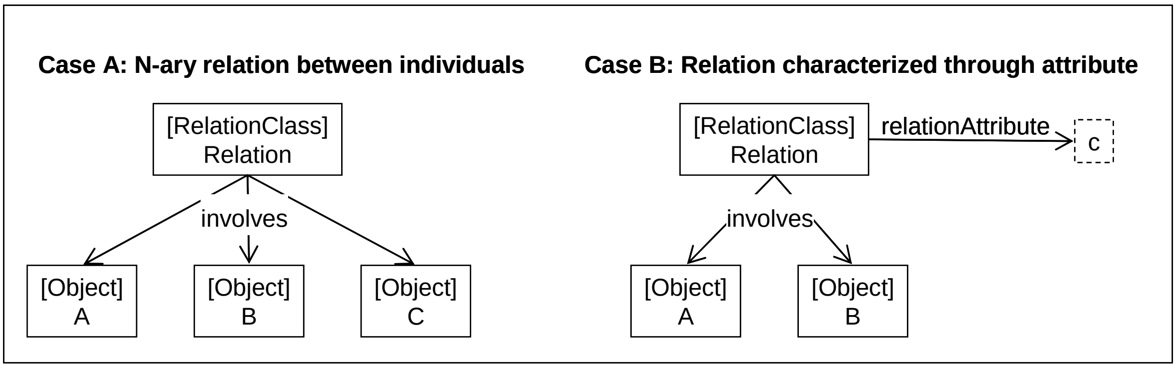

Fig. 5: Application cases for a *relation class*

Fig. 6 shows the design pattern that defines a *relation class*. A *relation class* involves at least one *object* and at least one other individual of type *object* or *feature space*. Moreover, it may be characterized by some relationAttributes. The *objects* involved in the n-ary relation can be explicitly identified via the inverse relations involvesObject and isInvovledInN-aryRelation.

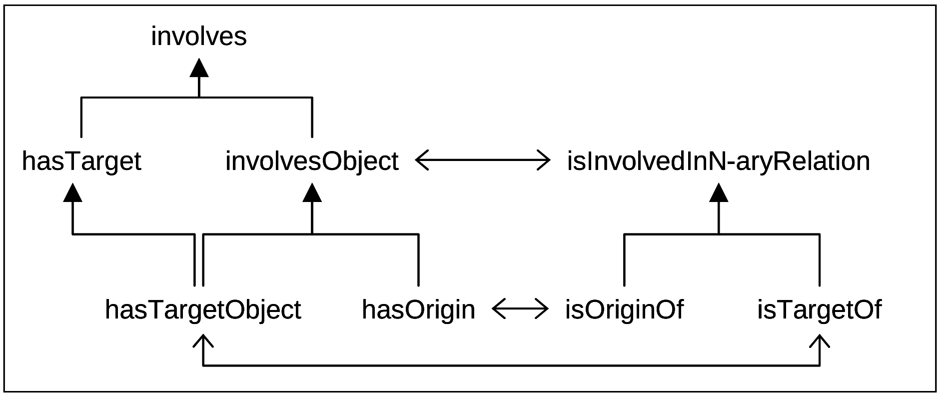

Fig. 6: Design pattern for a *relation class*

Two specializations of *relation class* are introduced:

- A *directed n-ary relation* describes an n-ary relation among some individuals of type *object* or *feature space* where at least one *object* is distinguished as the origin of the relation.
- By contrast, a *coequal n-ary relation* describes an n-ary relation among some individuals where none of the individuals involved in the relation stands out as the origin of the relation.

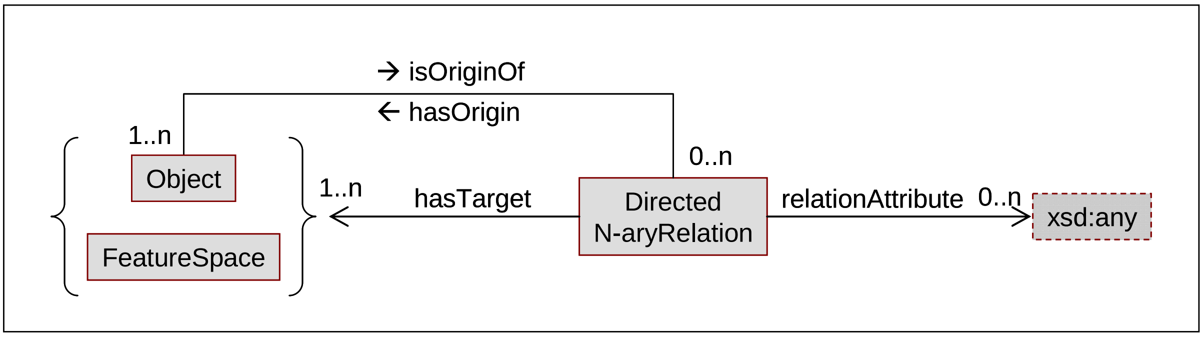

Fig. 7: Directed n-ary relation

The origin of a *directed n-ary relation* is identified by means of the inverse relations hasOrigin and isOriginOf (Fig. 7). The other involved individuals are denoted as targets of the n-ary relation through the hasTarget relation. The target objects can be explicitly identified via the inverse relations hasTargetObject and isTargetOf. The specialization hierarchy of these relations is displayed in Fig. 8.

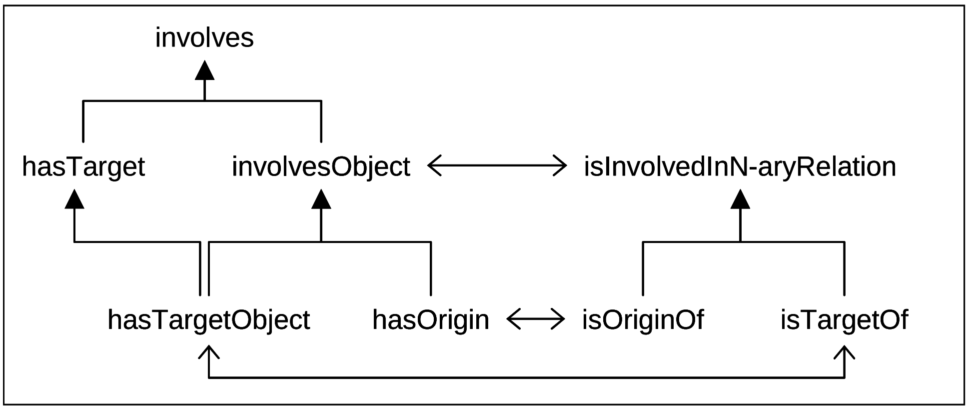

Fig. 8: Specialization hierarchy of the relations for the class *directed n-ary relation*

Fig. 9 gives an application sample of a *directed n-ary relation*. As can be seen in the figure, a *directed n-ary relation* may have more than one relation origin. The class *unique origin n-ary relation* is introduced to denote the special case of a *directed n-ary relation* which has exactly one relation origin.

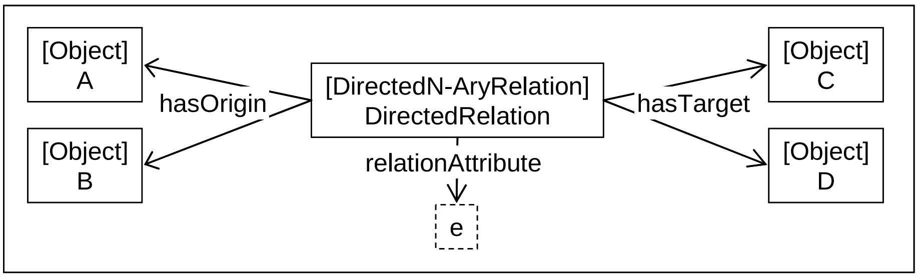

Fig. 9: Application sample of a *directed n-ary relation*

Usage

The fundamental concepts introduced above are not intended to be used (i.e., instantiated) directly; rather, they serve the purpose of (a) organizing the derived classes and relations, and (b) characterizing their role within the ontology. By means of the latter, a user or a software program is advised how to properly treat that particular concept. For example, classes that are derived from the *relation class* are obviously auxiliary constructs for the representation of n-ary relations. Consequently, instances of such classes do not need to be given meaningful names (cf. Noy and Rector, 2006). Instead, they may be labeled according to some standardized naming convention (a possible naming convention would be to use the identifier of the *relation class* and append an underscore (“_”), followed by a unique number, i.e., <identifier of relation class>_<unique number>). Thus, each time a class is identified as a specialization of *relation class*, the user (or an intelligent software program) can conclude that the instance should be labeled automatically, following the chosen naming convention.

Concept Descriptions

The individual concepts are described below.

Classes

Coequal n-ary relation

Description

Coequal n-ary relation is a *relation class* that describes an n-ary relation among three or more individuals or dataype values. None of the individuals involved in the relation stands out as the origin (or owner) of the relation.

Relations

- *Coequal n-ary relation* is a subclass of *relation class*
- A *coequal n-ary relation* has no origin.

Directed n-ary relation

Directed n-ary relation is a *relation class* that describes an n-ary relation among three or more individuals or attribute values. Some of the individuals involved in the n-ary relation are distinguished from the others in that they are origins of the relation.

Definition

A *relation class* that has at least one *object* as origin.

Relations

- *Directed n-ary relation* is a subclass of *relation class*.
- A *directed n-ary relation* has at least one *object* as origin.
- A *directed n-ary relation* has at least one target of type *object* or *feature space*.
- The target of a *directed n-ary relation* can only be of type *object* or *feature space*.
- The origin of a *directed n-ary relation* can only be an *object*.

Feature space

Description

An *object* can be characterized by means of descriptive features (qualities, characteristics). There are various ways how to model the values of such features, for example by representing the values as partitions of a classes or as enumerations of individuals – see (Rector, 2005) for a detailed discussion of this issue. A feature space defines the range of values that a particular feature can take. The meta root term *feature space* subsumes the different ways to define such a feature space.

Definition

A *feature space* is either a *value set* or a *value partition* or a *non-exhaustive value set*.

Non-exhaustive value set

Description

A *non-exhaustive value set* is a *feature space* that represents its possible values through individuals. These individuals, which are typically declared to be all different from each other, are instances of the *non-exhaustive value set*. Note that, in contrast to a *value set*, this class is not defined by an (exhaustive) enumeration of its instances. Thus, the number of individuals may change at run time.

Object

Description

Object is a meta root term that subsumes all the self-standing (Rector, 2003) entities – whether physical or abstract – that exist in an application domain.

Relations

- An *object* may be involved in an n-ary relation with a *relation class*

Relation class

Description

The OWL language merely provides language primitives to establish *binary* relations between two individuals or between an individual and an attribute value. To create an *n-ary* relation that links three or more individuals or attribute values, an auxiliary *relation class* needs to be introduced, which acts as an intermediate node. *Relation class* is a meta root term that subsumes the different types of n-ary relations that can be defined (Noy and Rector, 2006).

Definition

A *relation class* is either a *directed n-ary relation* or a *coequal n-ary relation*.

Relations

- A *relation class* involves at least one *object*.
- A *relation class* involves EITHER at least two individuals of type *object* or *feature space* and at least one relationAttribute OR at least three individuals of type *object* or *feature space*.
- A *relation class* involves only individuals of type *object* or *feature space*.

Usage

Instances of *relation class* are merely auxiliary constructs to represent n-ary relations. Consequently, there is no need to give meaningful names to the instances of a *relation class* (cf. Noy & Rector, 2006). A possible naming convention is to use the identifier of the *relation class* and append an underscore ("_"), followed by a unique number, i.e., <identifier of relation class>_<unique number>.

Unique origin n-ary relation

Description

A *unique origin n-ary relation* is a relation among three or more individuals or attribute values. Exactly one of the individuals involved in the *unique origin n-ary relation* is distinguished from the others in that it is the origin of the relation.

Relations

- *Unique origin n-ary relation* is a subclass of *directed n-ary relation*.
- A *unique origin n-ary relation* has exactly one relation origin of type *object*.

Value partition

Description

A *value partition* is a *feature space* that represents its possible values as disjoint subclasses. These subclasses exhaustively partition the *feature space* and can in turn be further subpartitioned. It is possible to define alternative partitions of the same feature space. Further details about this particular type of feature space can be found elsewhere (Rector, 2005: “Pattern 2: Values as subclasses partitioning a feature”).

Usage

For practical use, the subclasses of the *value partition* must be instantiated. Two variants can be distinguished:

- Variant 1: Each time an *object* is assigned a value, a new individual will be created individually for the *object*. A naming convention is required for such individuals, for instance: name of the instantiated subclass, followed by “_of_”, followed by the name of the assigned *object*, i.e., <subclass name>_of_<object name> (e.g., RedColor_of_myCar).
- Variant 2: Each subclass of the partition is instantiated exactly once. Consequently, a value can be linked to several *objects*. The individuals representing the values are usually named after their respective subclasses.

Value set

Description

A *value set* is a *feature space* that represents its possible values through individuals. The individuals, which are typically declared to be all different from each other, are instances of the *value set*. The *value set* is sufficiently defined by an exhaustive enumeration of its instances.

Relations

- object-featureRelation and its inverse feature-objectRelation subsume all relations between *objects* and instances of *feature space;*

Relations

hasOrigin

Description

The relation identifies the *object* that is the origin of a *directed n-ary relation*.

Characteristics

- Specialization of involvesObject
- Domain: *directed n-ary relation*
- Range: *object*
- Inverse: isOriginOf

hasTaget

Description

The relation hasTarget identifies the *objects* or feature values (i.e., instances of *feature space*) that are the targets of a *directed n-ary relation*.

Characteristics

- Specialization of involves
- Domain: *directed n-ary relation*
- Range: *object* or *feature space*

hasTagetObject

Description

The relation hasTargetObject identifies the *objects* that are the targets of a *directed n-ary relation*.

Characteristics

- Specialization of hasTarget
- Specialization of involvesObject
- Domain: *directed n-ary relation*
- Range: *object* or *feature space*
- Inverse: isTargetOf

inter-objectRelation

Description

The relation inter-objectRelation subsumes all types of binary relations between *objects*.

Characteristics

- Domain: *object*
- Range: *object*

involves

Description

The relation identifies the *objects* and feature values (i.e., instances of *feature space*) that are involved in an n-ary relation.

Characteristics

- Domain: *relation class*
- Range: *object* or *feature space*

involvesObject

Description

The relation identifies the *objects* involved in an n-ary relation.

Characteristics

- Specialization of involves
- Domain: *relation class*
- Range: *object*

isInvolvedInN-aryRelation

Description

The relation isInvolvedInN-aryRelation denotes the relation between an *object* and a *relation class*.

Characteristics

- Domain: *object*
- Range: *relation class*
- Inverse: involvesObject

isOfType

Description

The relation isOfType assigns *value types* to *objects*. Based on these characteristics, a reasoner can infer if an *object* belongs to a predefined ontology view.

isOriginOf

Description

The relation points from an *object* that is the origin of an n-ary relation to a *directed n-ary relation*.

Characteristics

- Specialization of isInvolvedInN-aryRelation
- Domain: *object*
- Range: *directed n-ary relation*
- Inverse: hasOrigin

isTargetOf

Description

The relation points from an *object* that is the target of an n-ary relation to a *directed n-ary relation*.

Characteristics

- Specialization of isInvolvedInN-aryRelation
- Domain: *object*
- Range: *directed n-ary relation*
- Inverse: hasTargetObject

object-featureRelation

Description

The relation object-featureRelation denotes the relation between an *object* and its feature values (i.e., instances of *feature space*).

Characteristics

- Domain: *object*
- Range: *feature space*

Attributes

relationAttribute

Description

The attribute relationAttribute identifies an attribute value that is an attribute of a *relation class*.

Characteristics

- Domain: *relation class*
- Datatype: any (built-in XML Schema Datatype)

3. Polyhierarchies and Ontology Views

Practical applications require different *views* on the ontology. Comparable to a view on a relational database[10], an *ontology view* is a set of concepts (classes or instance data) that is retrieved from the ontology as the result of a pre-defined query class.

To realize ontology views, Rector (2003) proposes to establish alternative axes of classification in the ontology, where each axis assembles concepts for a particular use. Generally, such axes can be implemented by means of multiple classification, as presented in Fig. 10 for the classification of *objects*: First, different *object types* (here: *Type_1*, *Type_2*, and *Type_3*) are introduced; then the actual *objects* (here: *A*, *B*, and *C*) are explicitly assigned to one or more of these *object types* through subclassing.

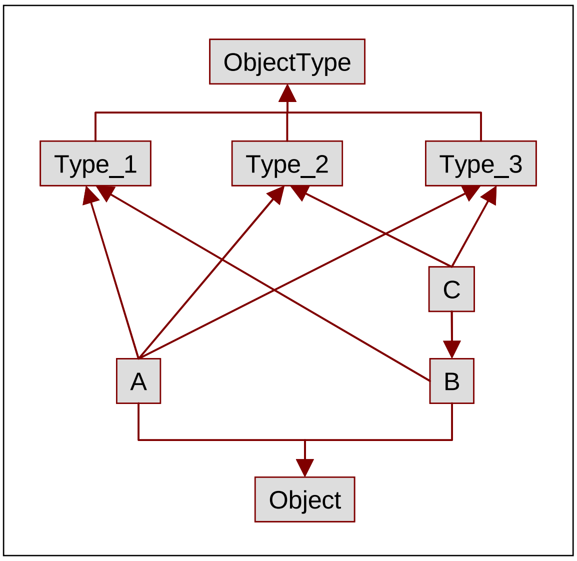

Fig. 10: Realization of ontology views by multiple classification

The problem with this approach is that complex *polyhierarchies* will evolve, which are hard to grasp for human users and thus difficult to manage and to maintain. Therefore, another approach is recommended here: Adopting the mechanism for *ontology normalization* suggested by Rector (2003), *objects* are explicitly classified along a single axis only. Specialization along this classification axis should preferably be based on the same (or progressively narrower) criteria, throughout. The classes introduced on this axis may be either primitive (i.e., characterized through necessary conditions only) or defined (i.e., characterized through necessary and sufficient conditions).

All further axes must be defined implicitly by (1) assigning *value types* to the *objects*, and (2) defining the *object types* as classes that are of a particular *value type* (cf. Fig. 11). That is, having an isOfType relation to a particular *value type* is a necessary and sufficient condition for an object being subsumed by a particular *object type*. Following this mechanism, a pattern evolves which is comparable to one shown in Fig. 11. From this pattern, a polyhierarchy like the one presented in Fig. 10 can be automatically inferred by a reasoner.

[10] A view of a relational database is a virtual or logical table that is composed of the result set of a pre-compiled query.

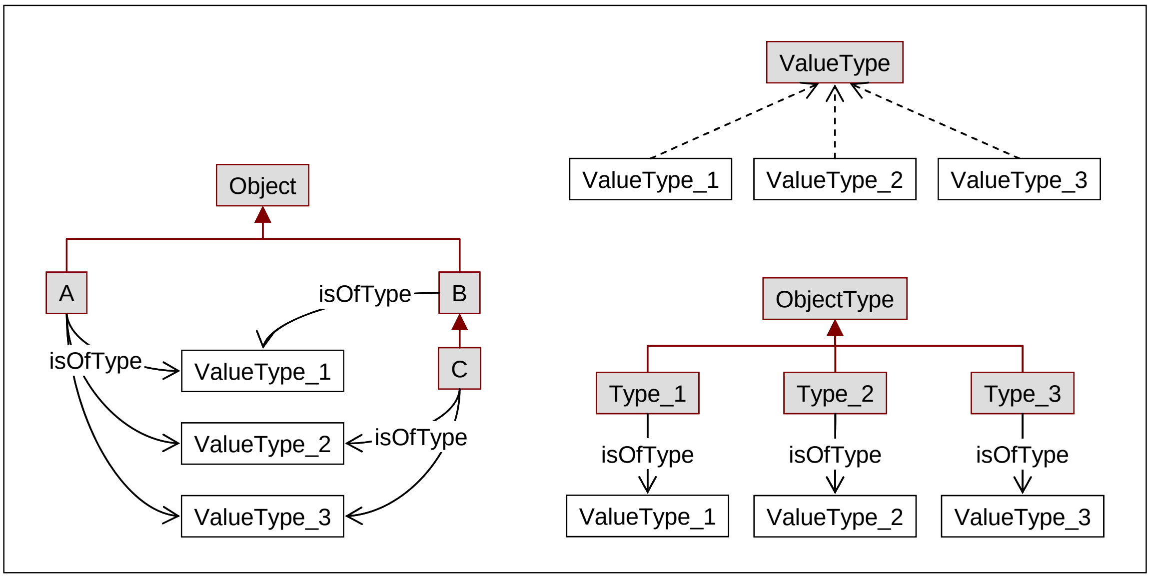

Fig. 11: Implicit classification through value types

Value types can either be subclasses or, as exemplarily shown in Fig. 11, instances of a *value type* class. Thus, *value types* can either be subclasses of *value partition* or (*non-exhaustive*) *value set*. They can again be organized in hierarchies.

Usage

The above classes (*object type*, *value type*, etc.) were introduced for explication only. They do not form part of the OWL implementation of the Meta Model, but should be introduced in the target ontology. Only the relation isOfType is implemented in the Meta Model. To simplify matters, it belongs to the ontology module *fundamental_concepts* (as it does not make sense to establish a new module for a single relation).

The following notation convention is recommended for target ontologies:

- Classes that represent value types should be labeled the suffix '*VT*'.
- Classes that implement ontology views may be labeled by the suffix '*Type*'.

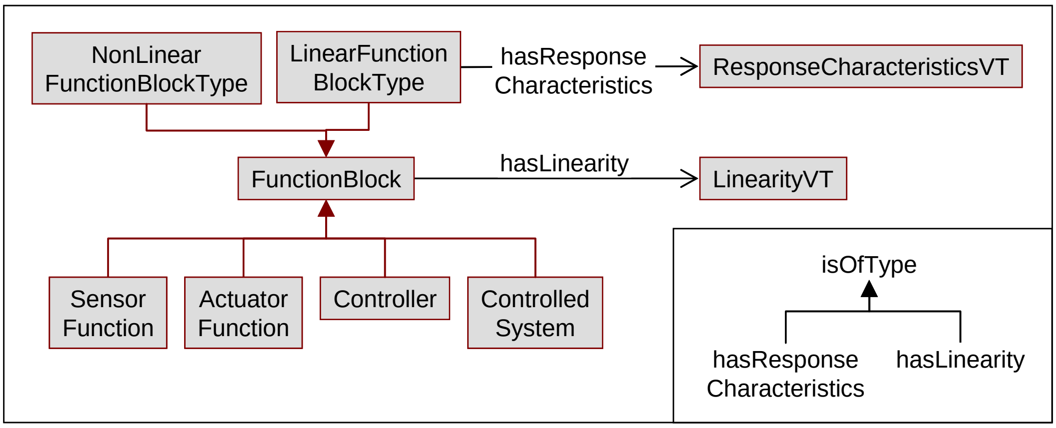

Fig. 12: Classification of function blocks

An application example from the area of control theory is presented in Fig. 12 and Fig. 13: A control loop is composed of different types of *function blocks*, which can be classified as *sensor function*, *actuator function*, *controller*, and *controlled system*. Two further features are of interest about a *function block*: its linearity and its response characteristics. These features are modeled through the value types *linearity VT* and *response characteristics VT*.

- *Linearity VT* is a *value set* made up of the individuals linear and nonlinear.
- *Response characteristics VT* is a *non-exhaustive value set*, which comprises the individuals P-Element, I-Element, D-Element, PID-Element, PT1-Element, and others.

Instances of these two value types are linked to a *function block* via the relations hasLinearity and hasResponseCharacteristics, which are specializations of the relation isOfType.

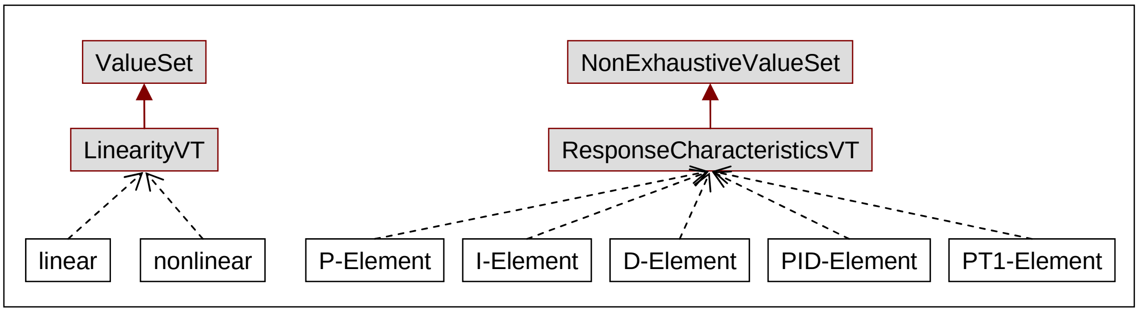

Fig. 13: The value types *linearity VT* and *response characteristics VT*

Based on these concepts, different ontology views on *function blocks* can be established. For example, all linear *function blocks* can be retrieved by introducing a class *function block linear type*, which is defined as follows:

function block linear type ≡ *function block* AND hasLinearity linear.

Similarly, all PID *controllers* could be retrieved via the class

controller PID type ≡ *controller* AND hasResponseCharacterisitcsVT PID-Element.

Further ontology views can be realized in an analogous manner.

Concept Definitions

Relations

isOfType

Description

The relation isOfType assigns *value types* to *objects*. Based on these characteristics, a reasoner can infer if an *object* belongs to a predefined ontology view.

Characteristics

- Specialization of object-featureRelation
- Domain: *Object*
- Range: *Feature space*
- Functional

4. Mereology

Mereology is the theory of parthood relations (a.k.a. part-whole relations), i.e., the relations that exist between a part and the whole. There are numerous publications on this subject, e.g. by Simons (1987) or by Casati and Varzi (1999); Varzi (2006) gives an excellent introduction to the field in the Standford Encyclopedia of Philosophy. Different axiomatic systems of mereology exist, which have dissimilar properties. However, the following three axioms form the basis of any mereological theory and can thus be considered as the core principles of mereology. The axioms state the parthood relation to be

- *transitive*: an object that is a part of a part of a whole is itself a part of the whole – if object A is part of object B, and if B is part of object C, then A is part of C;
- *reflexive*: an object is part of itself – A is part of A;
- *antisymmetric*: two distinct objects cannot be part of each other – if A is part of B and A ≠ B, then B cannot be part of A.

Unlike other modeling languages such as UML (e.g., Fowler, 1997), OWL does not provide any built-in primitives for part-whole relations. There are various possibilities to model such parthood relations, and the respective approaches have different effects on the usability, expressiveness, and reusability of the ontology as well as on the performance of a reasoner for classifying the ontology. Thus, a design pattern needs to be established that defines a standard way of modeling mereological relations.

The mereology design pattern suggested below follows the best-practice guidelines set out by Rector and Welty (2005) for representing part-whole relations in OWL. In addition, it takes up an idea from the UML to distinguish two types of the part-whole relationship: *aggregation* and *composition*:

- Aggregation is the binary relation that exists between an *aggregate* (or whole) and one of its *parts*. A *part* may be part of more than one *aggregate*, i.e., it may be shared by several *aggregates*. A *part* can exist independently from the *aggregate*.
- Composition is a special type of an aggregation relation that exists between a *composite object* and its parts (hereafter: *part of composite object*). *Parts of composite objects* are non-shareable, i.e., they cannot be part of more than one *composite object*. If the *composite object* ceases to exist, its parts cease to exist, as well.

Mereology makes no assumptions on the nature of *aggregates* or *parts*: “They can be material bodies, events, geometric entities, or geographical regions, […] as well as numbers, sets, types, or properties” (Varzi, 2006). Thus, both *aggregates* and *parts* are defined as specializations of the generic *object* class, without imposing any further constraints on them. The two classes are not declared to be disjoint, as an *aggregate* could be at the same time a *part* of another *aggregate*. The relation between a *part* and an *aggregate* is modeled via the relation isPartOf and its inverse hasPart; it is usually depicted through a line with a white diamond-shaped arrowhead pointing towards the *aggregate* class (cf. Fig. 14).

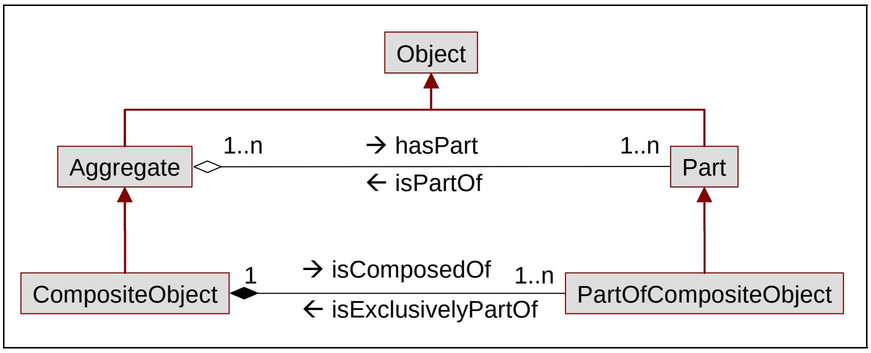

Fig. 14: Aggregation and composition

At present, OWL does not provide any language constructs for representing the aforementioned axiom of antisymmetry; neither can the reflexive properties of the parthood relation be properly modeled with the current version of OWL (cf. Rector and Welty, 2005). The required extensions to the modeling language have been announced to be incorporated in the next release of OWL (Patel-Scheider and Horrocks, 2006). Transitivity, on the other hand, can already be modeled in current OWL by declaring the relations isPartOf and hasPart to be transitive (Fig. 15). This enables an OWL-compatible reasoner to infer that, if *object* A is a part of *object* B and B is in turn a part of *object* C, then A must be a part of C, as well.

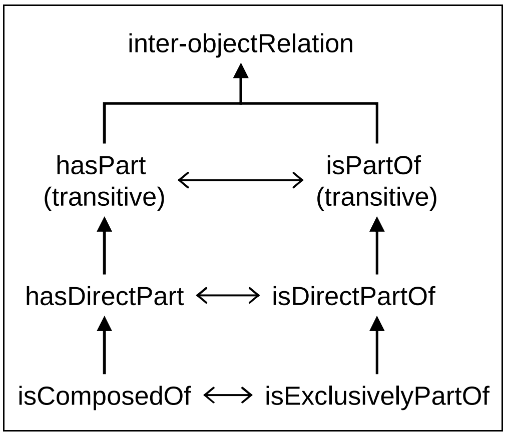

Fig. 15: Mereologic relations

Many applications require not a list of all parts but rather a list of the next level breakdown of parts, the so-called *direct parts* of a given entity (Rector and Welty, 2005). To this end, the relation hasDirectPart is introduced as a specialization of hasPart; similarly, its inverse isDirectPartOf is declared to be a specialization of the isPartOf relation. These relations are non-transitive and link each subpart just to the next level. Declaring hasDirectPart (and isDirectPartOf) to be a specialization of a hasPart (isPartOf) has the following advantage: If *objects* are repeatedly linked via hasDirectPart (or isDirectPartOf) relation, a reasoner can still infer that a hasPart (isPartOf) relation exists between the *aggregate* and the *part* of a *part*. For example, if A is a direct part of B, and B is a direct part of C, it can be inferred that A is a part of C. That way, an *aggregate* can be repeatedly decomposed into *parts* and sub-*parts* until the desired decomposition level is achieved.

While the declaration of direct parts seems intuitive at first sight, a possible problem pointed out by Rector and Welty (2005) is that, “the mere idea of a direct part is subjective, one may invent intermediate direct parts depending on numerous factors, or eliminate them. For example, we may choose not to represent engine as a part of cars, but rather represent all the components of engines as direct car parts. Grouping subparts into larger parts may also be subjective, a common example is a flywheel in a car, which can be viewed as an engine part or a transmission part in an ontology that includes those classes”. Thus, care must be taken when applying these relations.

For some applications (cf. Sec. 5), it is advantageous to know to which decomposition level a certain *part* belongs. This requires the definition of ‘real’ *parts*, i.e., *parts* that have no parts of their own; alternatively, ‘real’ *aggregates* may be introduced. These concepts are located on the top and bottom level, respectively, of the decomposition hierarchy. An exemplary decomposition across four levels is depicted in Fig. 16: The class *aggregate only* is defined as an *aggregate* that is not a *part* of any *object*. *First level part* is defined as an *object* that is linked to an *aggregate only* by an isDirectPartOf relation. Similarly, *second level part* is a direct part of a *first level part*, and arbitrary higher-level parts can be defined in an analogous manner. Eventually, the class *part only* is defined as a *part* that does not have any *parts* of its own.

Due to the open world assumption customarily made by DL reasoners, membership to the classes *part only* and *aggregate only* cannot be inferred, but must be declared explicitly[11]. Once the top (or bottom) of a decomposition hierarchy has been defined that way, the membership to the intermediate decomposition levels can be inferred automatically. Utilizing these class definitions, a reasoner should be able to assign an *object* to one of these decomposition levels[12].

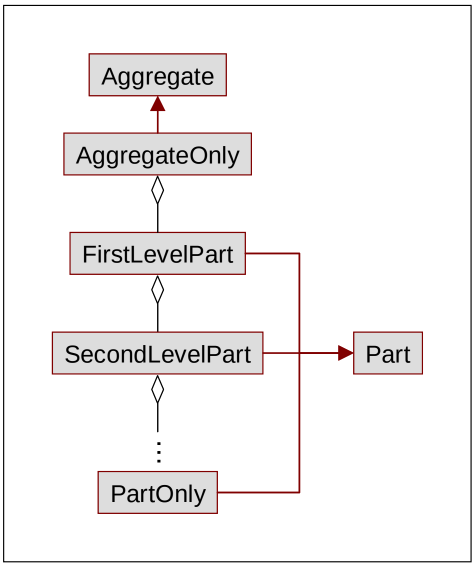

Fig. 16: Decomposition structure

To represent composition, the classes *composite object* and *part of composite object* are introduced as specializations of *aggregate* and *part*, respectively (Fig. 14). Moreover, the relations isComposedOf and its inverse isExclusivelyPartOf are introduced as specializations of the relations hasDirectPart and isDirectPartOf, respectively[13] (cf. Fig. 15; in figures, these relations are often depicted through a line with a black diamond-shaped arrowhead pointing towards the *composite object*). A cardinality restriction is imposed on the isExclusivelyPartOf relation to ensure that a *part of composite object* is part of exactly one *composite object*.

Unfortunately, the current OWL reasoners scale very badly when processing large collections of individuals connected via transitive, inverse relations. Hence, part-whole hierarchies that are connected by both hasPart and its inverse isPartOf can cause performance problems. Consequently, Rector and Welty (2005) advice to use either hasPart or isPartOf but not both in large-scale applications. Which one to choose depends very much on the occasion: isPartOf is frequently needed for query formulation, as the most common queries ask for the parts of an object (e.g., the equipment list for a particular plant). On the other hand, many class definitions require a hasPart relation – in OntoCAPE, for instance, the class *plant* is defined as the sum of its parts. Thus, as no relation can be ruled out in advance, both relations are provisionally defined in the Meta Model. Yet for large-scale applications using a reasoner, it might be necessary to abandon one of these.

[11] Consider the example of an individual P, which is an instance of *part* and does not have any parts of its own. While this is a perfect example of a *part only*, membership to this class cannot be inferred by the reasoner since the reasoner assumes that P may potentially be assigned some parts in the future.

[13] If isExclusivelyPartOf was a specialization of isPartOf, it would be impossible to state that a *part of composite object* is part of exactly one *composite object*, as there might be additional *composite objects* on higher aggregation levels.

Usage

The parthood relation is broadly applicable. According to Varzi (2006), it can be used to indicate any portion of a given entity, whatever the nature of the entity, and regardless of whether the portion is material or immaterial, whether it is connected to the remainder or disconnected, whether the part-whole relation has a spatial or a temporal character, and so on. Odell (1994) and Varzi (2006) discuss different kinds of relationships that can be considered as special types of part-whole relations, among which are the following:

- *Material constitution* describes the relation between an object and the material it is made of. This type of relation denotes the constituents of a mixture (Gin is part of Martini) as well as the construction material of a technical artifact (a car is partly of steel). Material constitution is a special form of aggregation, as a part (i.e., the material) can exist independently of the whole. Hence, hasPart should be used to model this type of relation.
- *Membership* is also a special form of aggregation, as a member can be part of different groups and exists independently of these groups. The isPartOf relation is used to indicate a membership to a group.
- A *portion* is a part that is of the same type as the whole; for example, a slice of bread is a portion of a loaf of bread. A portion relationship is a special type of composition, as the part cannot exist on its own if the whole ceases to exist. Thus, a portion can be linked to the whole via the isExclusivelyPartOf relation.

Rector and Welty (2005) list some potential applications of a mereological ontology; among those are

- a parts inventory for a technical artifact, which requires the "explosion" of parts;
- a fault detection system for a technical device in which one progressively narrows down the functional region of the fault; or.
- a document retrieval system, in which documents are divided into subunits, such as chapters, sections, paragraphs, etc.

Typically, the functionality of such applications can be summarized by the following competency questions:

- Query for the *parts* of an *object*.
- Query for the direct *parts* of an *object*.
- Query for the *first* (*second* …) *level parts* of an *object*.
- Query for the bottom-level *parts* (*part only*) of an *object*.
- Query for all *aggregates* an *object* is part of.
- Query for the *aggregates* an *object* is directly part of.
- Query for the top-level *aggregates* (*aggregates only*) an *object* is directly part of.
- Query if a particular *object* is a *part only*.
- Query if a particular *object* is a *first* (*second* …) *level part*.
- Query if a particular *object* is an *aggregate only*.
- Query if a particular *object* is a part of an *object*.
- Query if a particular *object* is a direct part of an *object*.
- Query if an *object* has a particular *object* as a part.
- Query if an *object* has a particular *object* as a direct part.
- Tests have been performed to ensure that the *mereology* ontology module complies with these competency questions: To this end, a test data set was generated, against which the reasoner RacerPro (Racer Systems, 2006) evaluated the above queries. The queries were formulated

partly as class definitions in OWL and partly as query expression in the nRQL (new Racer Query Language) (Haarslev et al. 2004).

Concept Descriptions

Individual concepts of the module *mereology* are defined below.

Classes

Aggregate

Description

An *object* that has one or more distinct parts.

Definition

An *object* that has some parts of type *object*.

Relations

- *Aggregate* is a specialization of *object*.
- An *aggregate* has at least one part of type *object*.

Aggregate only

Description

An *object* that has one or more distinct *parts* and is not part of any *object* itself.

Definition

An *aggregate* that is not a *part*.

Relations

- *Aggregate only* is a specialization of *aggregate*.
- *Aggregate only* is disjoint with *part*.

Composite object

Description

An *object* that is composed of one or more *objects*. The parts of the *composite object* are non-shareable, i.e. an *object* that is part of a *composite object* cannot be part of another *composite object*.

Definition

An *object* that is composed of some *objects*.

Relations

- *Composite object* is a specialization of *aggregate*.
- A *composite object* is composed of at least one *object*.

First level part

Description

A *part* at the first level of decomposition.

Definition

A *part* that is a direct part of *aggregate only*.

Relations

- *First level part* is a specialization of *part*.
- A *first level part* is a direct part of *aggregate only*.
- *First level part* is disjoint with *second level part*.

Part

Description

An *object* that is part of another *object*. A *part* can be part of more than one *object*.

Definition

An *object* that is part of an *object*.

Relations

- *Part* is a specialization of *object*.
- A *part* is a part of at least one *object*.

Part of composite object

Description

An *object* that is part of a *composite object*. The parts of the *composite object* are non-shareable, i.e. an *object* that is part of a *composite object* cannot be part of another *composite object*.

Definition

An *object* that is exclusively part of an *object*.

Relations

- *Part of composite object* is a specialization of *part*.
- A *part of composite object* is exclusively part of exactly one *object*.

Part only

Description

An *object* that is part of another *object* and has no *parts* of its own.

Definition

A *part* that is not an *aggregate*.

Relations

- *Part only* is a specialization of *part*.
- *Part only* is disjoint with *aggregate*.

Second level part

Description

A *part* at the second level of decomposition.

Definition

A *part* that is a direct part of a *first level part*.

Relations

- *Second level part* is a specialization of *part*.

- A *second level part* is a direct part of a *first level part*.
- *Second level part* is disjoint with *first level part*.

Relations

hasDirectPart

Description

Parthood relation that indicates the direct *parts* of an *object*, i.e., the *parts* on the next level breakdown.

Characteristics

- Specialization of hasPart
- Domain: *Aggregate*
- Range: *Part*
- Inverse: isDirectPartOf

hasPart

Description

Parthood relation that refers from an *aggregate* to its *parts*.

Characteristics

- Specialization of inter-objectRelation
- Domain: *Aggregate*
- Range: *Part*
- Inverse: isPartOf
- Transitive
-

isComposedOf

Description

Parthood relation that indicates the direct parts of a *composite object*. The parts of the *composite object* are non-shareable, i.e. a part cannot be part of more than one *composite object*. If the *composite object* is destroyed, all its parts are destroyed, as well.

Characteristics

- Specialization of hasDirectPart
- Domain: *Aggregate*
- Range: *Part*
- Inverse: isExclusivelyPartOf

isDirectPartOf

Description

Parthood relation that links a *part* to the *object* on the next aggregation level.

Characteristics

- Specialization of isPartOf
- Domain: *Part*
- Range: *Aggregate*
- Inverse: hasDirectPart

isExclusivelyPartOf

Description

Parthood relation that links a part to a *composite object* on the next aggregation level. The parts of the *composite object* are non-shareable, i.e. a part cannot be part of more than one *composite object*. If the *composite object* is destroyed, all its parts are destroyed, as well.

Characteristics

- Specialization of isDirectPartOf
- Domain: *Part*
- Range: *Aggregate*
- Inverse: isComposedOf

isPartOf

Description

Parthood relation that refers from a *part* to the *aggregate*.

Characteristics

- Specialization of inter-objectRelation
- Domain: *Part*
- Range: *Aggregate*
- Inverse: hasPart
- Transitive

5. Topology

The ontology module *topology* defines a theory of connectedness. It provides concepts for describing topological relations between distributed entities where there exists the possibility of emergent[14] or supervenient[15] relations between items of interest (Little and Rogova, 2005). Examples of topological relations are the connections between geographical and/or physical entities in 2D and 3D space. Moreover, the concepts of the topology module can be used to describe the connectedness of abstract entities, such as the unit operations in a process flowsheet or the activities in a business process model.

According to Borst (1997), there are two different approaches to create a topological ontology. One is to extend an existing theory of mereology with topological relations. The other is to integrate mereological and topological concepts and relations into one mereo-topological theory. The approach employed in the Meta Model responds to the former and will be introduced subsequently.

The most fundamental concept of the ontology module *topology* is the relation isConnectedTo, which denotes the connectivity between *objects*. A first requirement for such a basic topological relation is *symmetry*: if *object* A is connected to *object* B, then B is connected to A, as well. A second requirements is *transitivity* – that is, if A is connected to B, and B is in turn connected to C, then A is also (indirectly) connected to C. As an example, consider a vessel (A) that is connected to a pipe (B), which is again connected to storage tank (C) – in this case, storage tank and vessel are (indirectly) connected via the pipe.

Frequently, only the *direct* connections between *objects* are of interest – in the above example, these would be the relations between A and B, and between B and C, respectively. The relation isDirectlyConnectedTo is introduced to represent direct connectivity. Similar to the definition of the hasDirectPart relation in the *mereology* module, isDirectlyConnectedTo is declared to be a non-transitive specialization of isConnectedTo. This way of defining direct connectivity enables a reasoner to infer the existence of (indirect) connections from explicitly stated direct connections: For example, if A is directly connected to B, and B is directly connected to C, it can be inferred that A is (indirectly) connected to C (cf. Fig. 18).

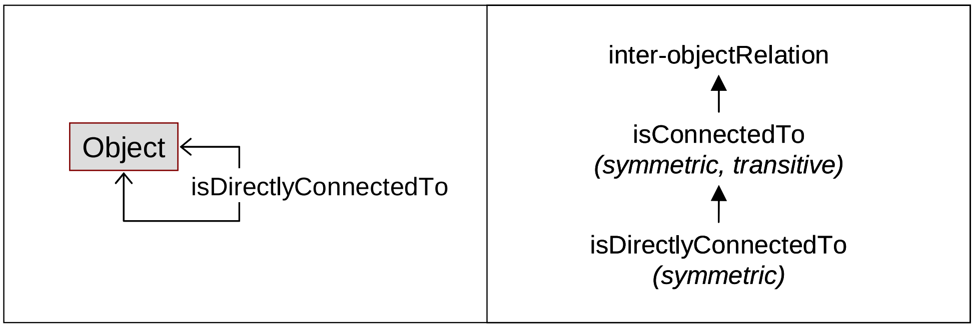

Fig. 17: Basic concepts of module *topology*

[14] Emergence is the process of complex pattern formation from more basic constituent parts or behaviors, and manifests itself as an emergent property of the relationships between those elements (Wikipedia, 2006).

[15] A set of properties A supervenes upon another set B just in case no two things can differ with respect to A-properties without also differing with respect to their B-properties (McLaughlin & Bennett, 2005).

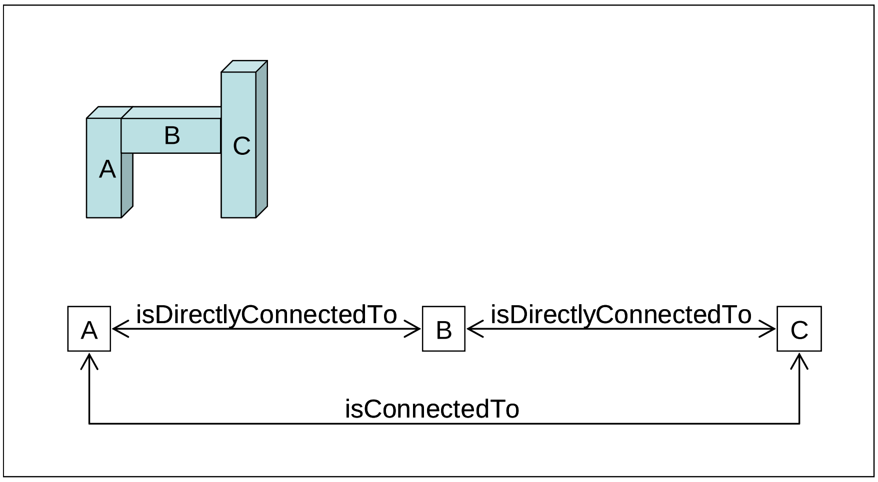

Fig. 18: Application example of module *topology*

Mereotopology

A significant aspect of this approach is that mereological and topological relations exclude each other, which prohibits topological relations (connections) between parts and wholes. Hence, a *part* that is linked to an *aggregate* via an isPartOf relation cannot refer to this *aggregate* by any topological relation. An example is given in Fig. 19. It shows an *aggregate* Y which has the distinct *parts* a, b, and c. A *part* cannot be directly connected to an *aggregate* (Fig. 19 a); however, the *parts* may be directly connected to each other (Fig. 19 b).

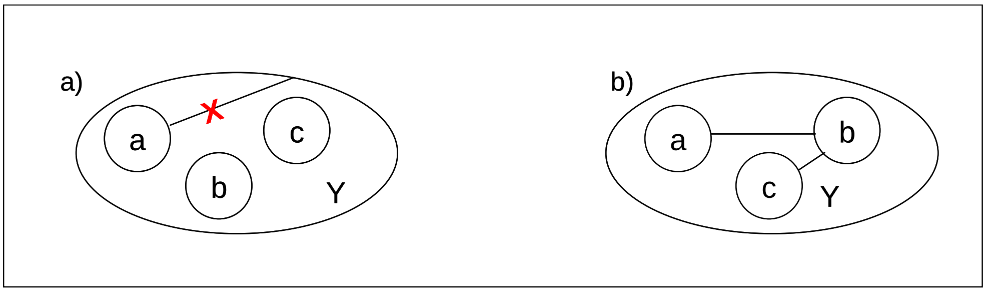

Fig. 19: Interdependency between mereological and topological relations

For a more concrete example, consider a cartwheel (a) that isPartOf a car (Y). Hence, an isConnectedTo relation between the cartwheel (a) and the car (Y) is prohibited (Fig. 19 a). Yet a cartwheel isDirectlyConnectedTo an axis (b), an axis isDirectlyConnectedTo the car body (c), and for the sake of completeness the cartwheel (a) isConnectedTo the car body (c) (Fig. 19 b)

To prevent that a direct connection between an *aggregate* and one of its *parts* is established, the following range restrictions are imposed on the isDirectlyConnectedTo relation:

- A *first* (*second* …) *level part* can only be directly connected to (connected to) a *first* (*second* …) *level part*.
- A *part only* can only be directly connected to (connected to) a *part only*.
- An *aggregate only* can only be directly connected to (connected to) an *aggregate only*.

A violation of these restrictions will be considered as an error. Hence, *objects* can only be topologically connected if they are situated at the same level of decomposition. That way, mereological and topological relations are strictly kept apart, only the former or the latter relation can be applied between individuals.

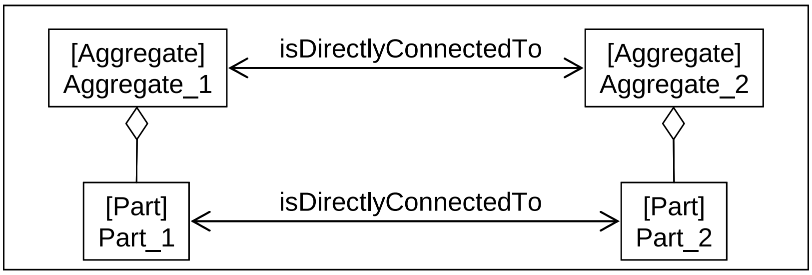

Fig. 20: A connection between *parts* implies a connection between *aggregates*

Another important point to make is that a connection between two *parts* of distinct *aggregates* implies a connection between these *aggregates* (cf. Fig. 20). This can be formulated as a rule: If the *parts* of distinct *aggregates* are directly connected, then these *aggregates* must be directly connected, as well. In contrary, if distinct aggregates are directly connected, the reasoning of connectivity of parts is by no means valid. Unfortunately, there is no means to implement such a rule in OWL; thus, the rule must currently be enforced by the user.

Connectors

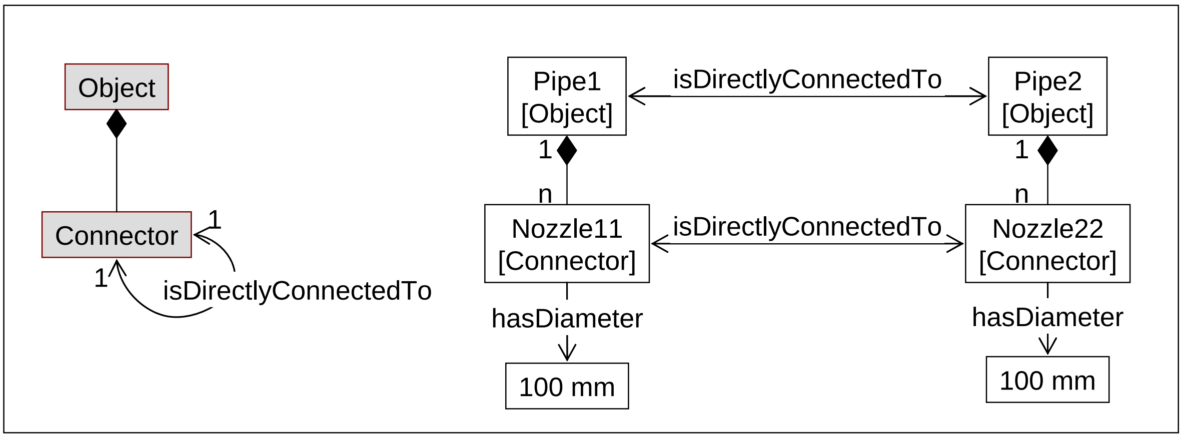

Fig. 21: Connecting *objects* via *connectors*

The type and number of connections that an *object* may have can be constrained by means of *connectors*. A *connector* represents the interface through which an *object* can be connected to another. A *connector* is a *part* that is linked to an *object* via the isExclusivelyPartOf relation, and it can be connected to exactly one other *connector* via the isDirectlyConnectedTo relation (cf. left-hand side of Fig. 21). Optionally, the possible connections of a *connector* can be further restrained, for instance by postulating that certain properties of two linked *connectors* need to match for a feasible connection. Take the example of two pipes that are to be connected: A connection between two pipes is feasible if the diameters of their nozzles are the same. This situation can be modeled by representing the pipes as instances of *object*, the nozzles as *connectors*, and their diameters as attributes of the respective nozzles (right-hand side of Fig. 21). An additional constraint permits only connections between nozzles that have the same diameter.

Representation of graphs

An extended topology which allows for the representation of graphs is represented in Fig. 22; the major concepts of this approach are *nodes* and *arcs*. Basically, an *arc* cannot connect to more than two *nodes*, which excludes *arcs* that fork. A *node*, on the other hand, can be connected to several *arcs*.

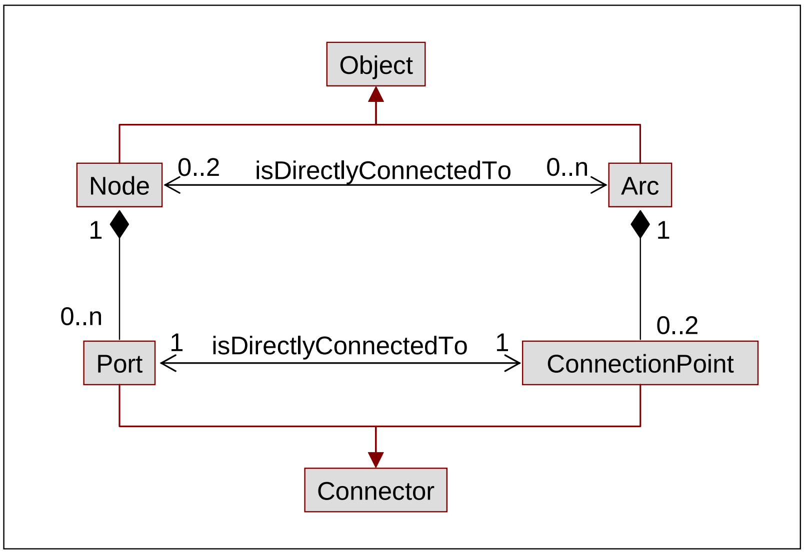

Fig. 22: *Nodes* and *arcs*

Optionally, a *node* may have a list of *ports*, and an *arc* may have up to two *connection points. Ports* and *connection points* are specializations of the *connector* class; they are linked to the corresponding *node* or *arc* via the isExclusivelyPartOf relation and can be connected to each other via the isDirectlyConnectedTo relation. *Ports* and *connection points* act as interfaces to *nodes* and *arcs*, respectively: like *connectors*, they carry specific characteristics that have to match if a *port* is to be connected to a *connection point*. That way, they restrict and further specify the type and number of connections that a *node* or an *arc* can have.

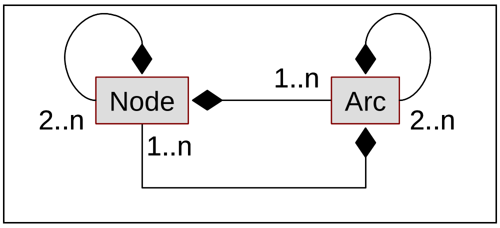

Fig. 23: Decomposition of *nodes* and *arcs*

Another important issue is that both *nodes* and *arcs* can be decomposed into a number of sub-*nodes* and sub-*arcs*, respectively (Fig. 23). When a *node* is decomposed into a number of sub-*nodes*, it is necessary for these sub-*nodes* to be connected by internal *arcs*. Similarly, when *arcs* are decomposed into sub-*arcs*, there must be internal *nodes* between the sub-*arcs*. Thus, a *node* has to be decomposed into two *nodes* and one connecting *arc*, at least; likewise, an *arc* cannot be decomposed in less than two *arcs* and one *node*[16]. The sub-*nodes* and sub-*arcs* are connected via isDirectlyConnectedTo relations (Fig. 24).

[16] Unfortunately, it is presently not possible to model this decomposition axiom in OWL, as the current version of OWL does not support qualified cardinality restrictions (QCR). However, QCRs will be incorporated in the upcoming OWL 1.1 (Patel-Scheider and Horrocks, 2006).

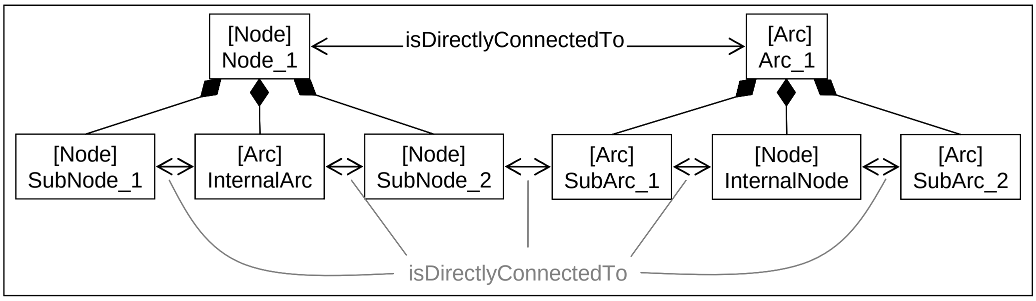

Fig. 24: Connections between sub-*nodes* and sub-*arcs*

A special situation arises if a *node* is decomposed while the connected *arc* is not[17]. Such a pattern occurs, for example, when a process flowsheet is hierarchically refined, as it is exemplarily shown in Fig. 25: Here, the *node* representing the overall process is decomposed into a reaction section and a separation section, whereas the *arcs* representing the feed and product streams are not decomposed at all. Now the question arises, which sub-*node* is connected to which *arc* (in the example, the feed stream enters the reaction section, whereas the product stream leaves the separation section).

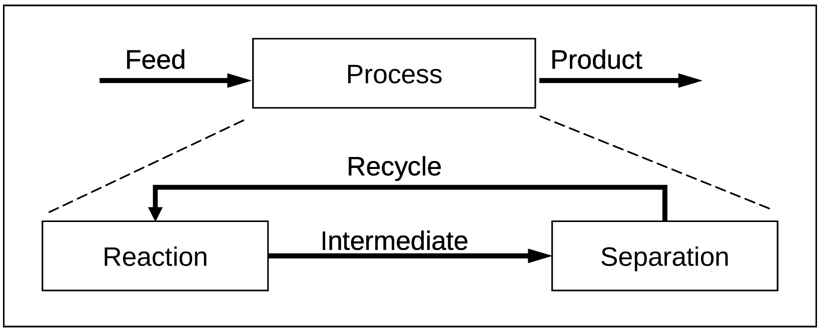

Fig. 25: Hierarchical refinement of a process flowsheet

A straight-forward solution is to connect the *arcs* representing the feed and product streams directly to the *nodes* representing the reaction and separation sections, as indicated in Fig. 26. Remember however, that topological connections are only permitted between *nodes* and *arcs* that are situated on the same level of decomposition. Therefore, this solution is only applicable if the mereological levels of the feed and product *arcs* are not fixed, that is, if the feed and product *arcs* can be assigned the same level of decomposition as the sub-*nodes* for reaction and separation. If this is not possible, an alternative solution must be chosen. Note that for the sake of clarity, the internal *arc* representing the recycle stream is neglected in the following.

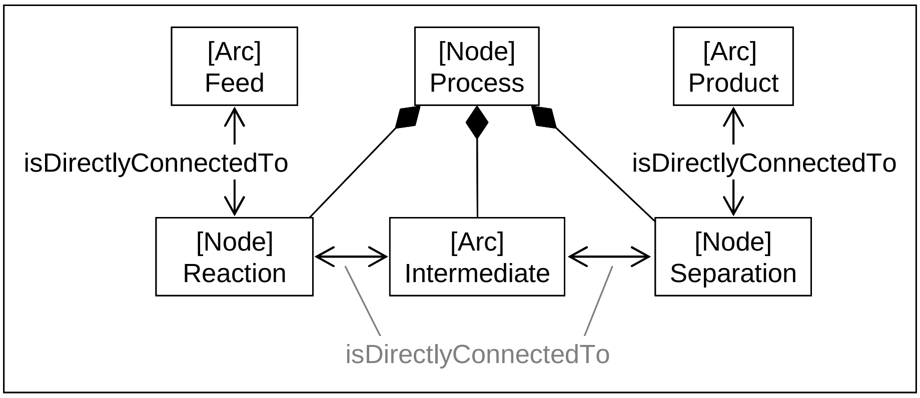

Fig. 26: Solution alternative 1 – the *arcs* are directly connected to the refined sub-*nodes*

[17] Of course, the same considerations apply to the opposite case, when the *arc* is decomposed while the *node* is not. To simplify matters, only the first case is discussed here.

If a direct connection between an *arc* and a sub-*node* is not feasible, the two may still be indirectly linked via their respective *ports* and *connection points*. Fig. 27 presents the corresponding pattern: *Port* and *connection point* are to be linked via an isDirectlyConnectedTo relation. While the *port* may be a direct part of the sub-*node*, the *connection point* must only be an indirect part of the *arc*. The reason for this is, again, the required compatibility of the decomposition levels: If the *connection point* was a direct part of the *arc*, then *port* and *connection point* would be situated on different levels and thus could not be connected. Note that this alternative is not feasible for the refinement of the process flowsheet and thus represented in a generic way.

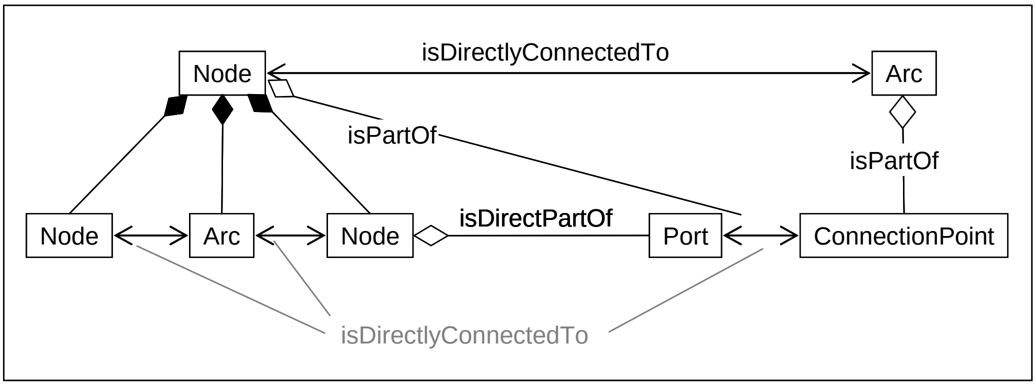

Fig. 27: Alternative 2 – an *arc* is indirectly linked to a sub-*node* via a *port* and a *connection point*

In case that the *node* and *arc* do not have designated *ports* and *connection points*, the above solution is not applicable. As an alternative, the *arc* may simply be duplicated; that is, a placeholder *arc* is to be introduced. The correspondence between the placeholder *arc* and the original *arc* is established via the relation sameAs. A generic example is shown in Fig. 28.

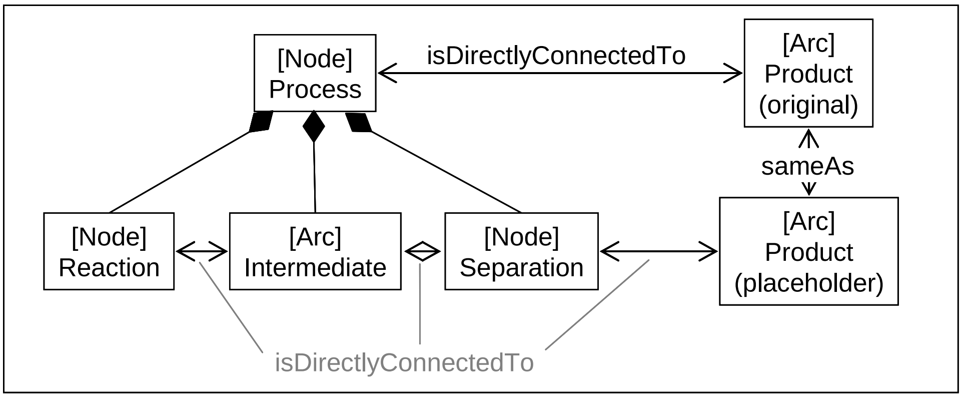

Fig. 28: Alternative 3 – duplication of the *arc*; correspondence is established via the sameAs relation

Directed Graphs

A further extension of the *topology* module allows for the representation of directed graphs: To this end, the class *directed arc* is introduced, which can be employed to indicate a directed connection between *nodes* such that one *node* is the predecessor or the successor of the other. As shown in Fig. 29 a *directed arc* is linked to a *node* via the relations enters and leaves, respectively.

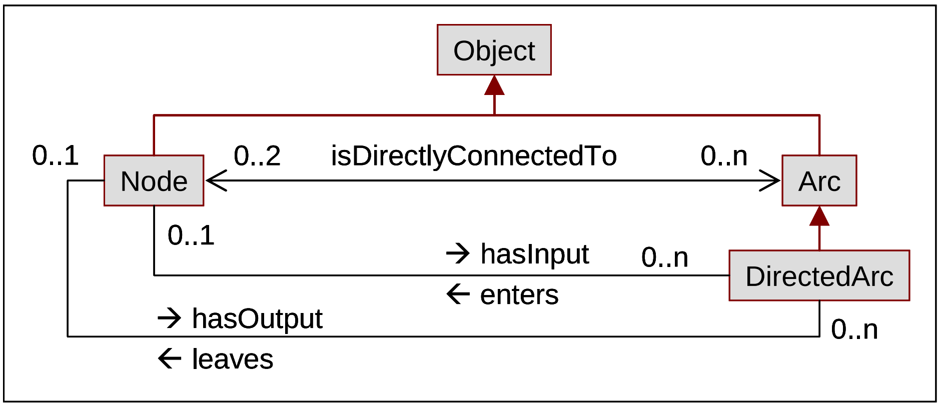

Fig. 29: Extended topology including directed arcs

The relation taxonomy presented in Fig. 17 is extended, as shown in Fig. 30.

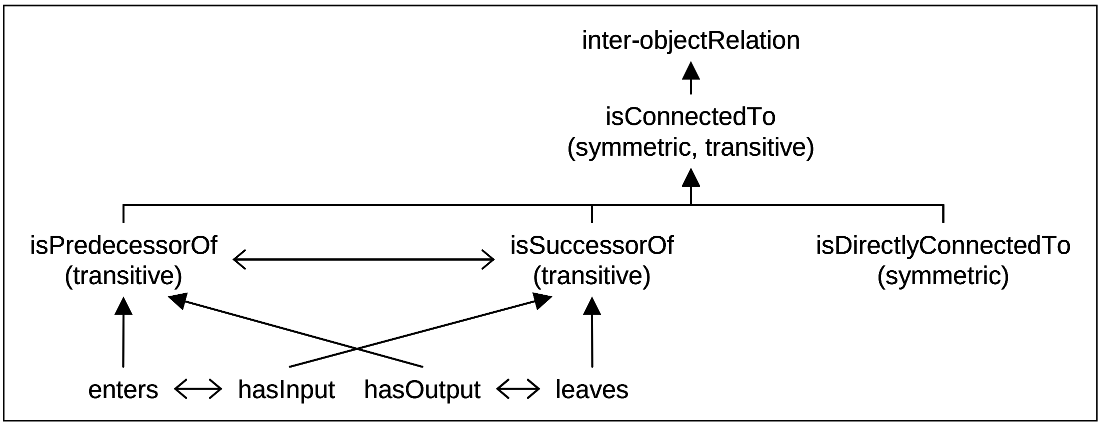

Fig. 30: Extended relation taxonomy

The relation enters and its inverse hasOutput are specializations of the transitive relation isPredecessorOf, which is a specialization of isConnectedTo. Similarly, the relations leaves and its inverse hasInput are specializations of isSuccessorOf, which is the inverse of isPredecessorOf. The relations hasInput and hasOutput are to be used to identify the *directed arcs* that are directly attached to a *node*. The main purpose of the supplementary relations isPredecessorOf and isSuccessorOf is to identify the *nodes* (or *directed arcs*) that precede or succeed a specific *node* (or *directed arc*) in a directed graph[18].

As mentioned in the specification of the *mereology* module, the current OWL reasoners scale badly when processing large collections of individuals connected via transitive, inverse relations. Thus, for large-scale applications, it might be necessary to abandon either the isPredecessorOf relation or the isSuccessorOf relation.

Usage

To illustrate the functionality of the *topology* module, several competency questions are introduced; afterwards two examples are given to demonstrate that the *topology* module complies with the competency questions.

A primary distinction is made between directed and non-directed connections. For the non-directed connections, the following competency questions are defined (the classes in parenthesis are optional):

- Query for all *objects* (*nodes*, *arcs*) that are directly connected to a specific *object*
- Query for all *objects* (*nodes*, *arcs*) that are connected to a specific *object*

[18] In graph theory, a node B is considered to be the successor of node A, if a path leads from A to B.

- Query for all *arcs* that are connected to a *node* via a particular *port*.
- Query if two *objects* (*nodes*, *arcs*) are connected directly
- Query if two *objects* (*nodes*, *arcs*) are connected (either directly or indirectly)
- Check if topological relations are wrongly defined across different levels of aggregation.

Competency questions for the directed connections may comprise the former as well as the subsequent ones:

- Query for all *directed arcs* that enter a specified *node*
- Query for all *directed arcs* that leave a specified *node*
- Query for all *objects* (*nodes*, *arcs*) that are predecessors of a specified *object* (*node*, *arc*)
- Query for all *objects* (*nodes*, *arcs*) that are successors of a specified *object* (*node*, *arc*)

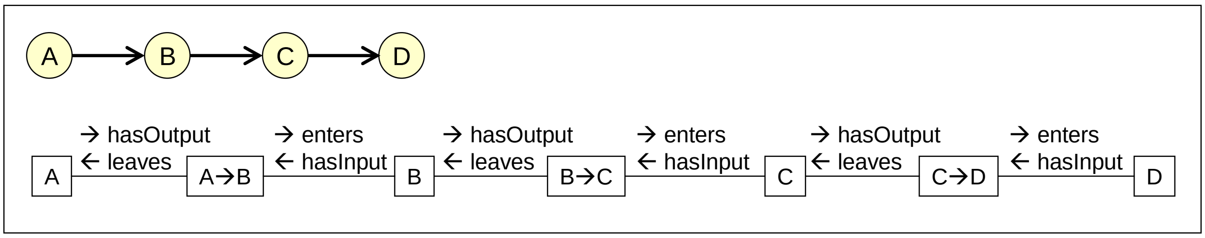

Fig. 31: Application example 1: acyclic directed graph

The first example presented in Fig. 31 shows a directed graph consisting of four different *nodes* (A, B, C, D), which are connected by three *directed arcs* (A→B, B→C, C→D). The lower part of Fig. 31 shows how such a graph is represented through an instantiation of the above concepts. If the example is used as a test case for the *topology* module, a reasoner will give the following results (relating to individual C) for directed relations. Please note that for the sake of clarity, the respective class names in brackets are omitted hereafter :

- *Objects* connected to C: {A, A→B, B, B→C, C, C→D, D} (as C is directly connected to B→C, and B→C is in turn connected to C, the reasoner infers that C is indirectly connected to itself)
- *Nodes* connected to C: {A, B, C, D}
- *Arcs* connected to C : {A→B, B→C, C→D}
- *Objects* preceding C: {A, A→B, B, B→C}
- *Nodes* preceding C: {A, B}
- *Arcs* preceding C: {A→B, B→C}
- *Objects* succeeding C: {C→D, D}
- *Nodes* succeeding C: {D}
- *Arcs* succeeding C: {C→D }
- *Arcs* entering C: {B→C}
- *Arcs* leaving C: {C→D}

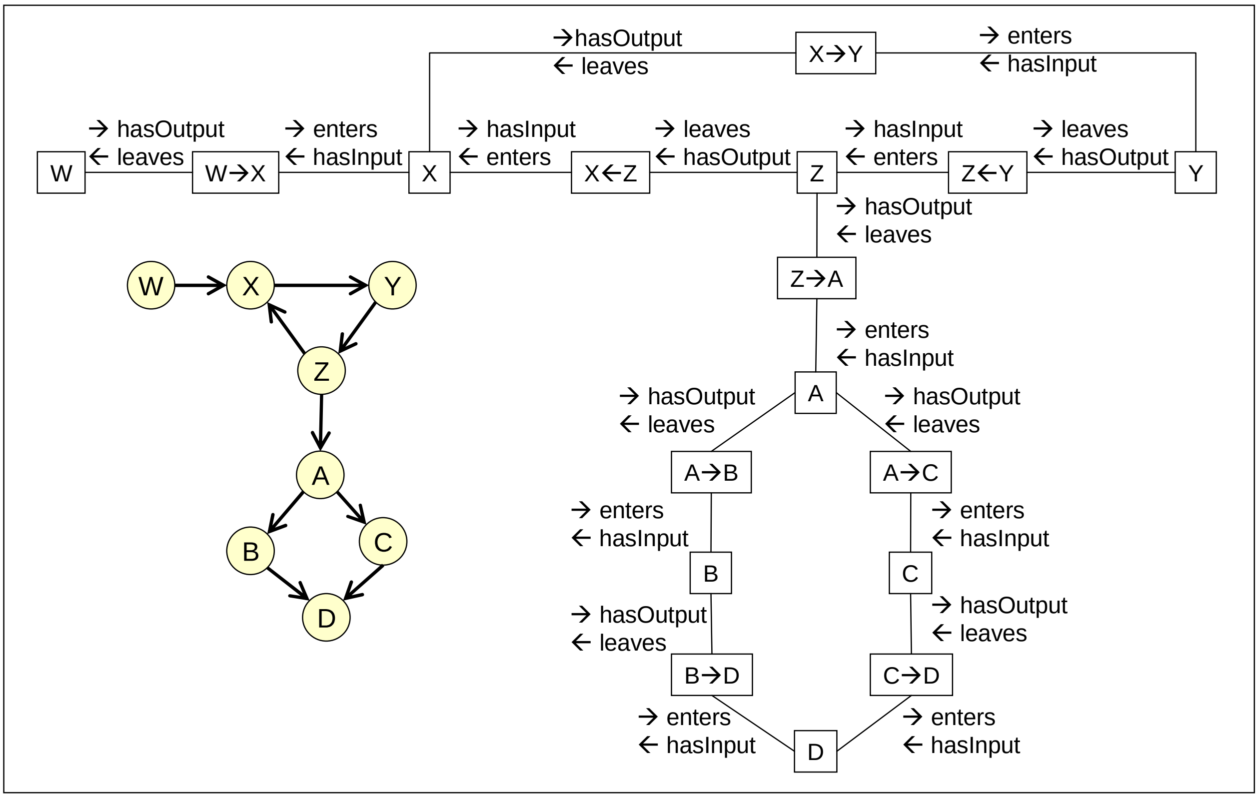

Fig. 32: Application example 2: directed graph with a cycle and a bifurcation

The second example presented in Fig. 32 shows a directed graph consisting of 8 different *nodes* (A, B, C, D, W, X, Y, Z) connected by *directed arcs*. The graph includes a cycle (X, Y, Z) as well as a bifurcation (A, B, C). Fig. 32 illustrates how this example is represented by an instantiation of the topological concepts. Taking this example as a test case, the following interesting results are obtained:

- *Nodes* preceding D: {A, B, C, W, X, Y, Z}
- *Nodes* succeeding W: {A, B, C, D, X, Y, Z}
- *Nodes* succeeding Y: {A, B, C, D, X, Y, Z} (i.e., the reasoner infers that all cycle *nodes*, including Y, are successors of Y)

Concept Descriptions

Individual concepts of the module *topology* are defined below.

Classes

Arc

Description

Arc is a specialization of *object* and represents the connecting element between *nodes*.

Relations

- *Arc* is a subclass of *object*.
- An *arc* can only be directly connected to a *node*.
- An *arc* cannot be directly connected to more than two *nodes*.
- An *arc* can only be a direct part of a *node* or *arc*.
- An *arc* can only have *arcs* or *nodes* or *connection points* as a direct parts.

- An *arc* can only have *arcs* or *nodes* or *connection points* or *ports* as parts.

Usage

An arc can be decomposed into a required number of sub-arcs. However, these sub-arcs have to be connected by so called internal nodes again. This in turn means that an arc must be decomposed into three sub-elements, at least: two sub-arcs and one internal node19.

Connection point

Description

Connection point represents the interface through which an *arc* can be connected to the *port* of a *node*. *Connection points* may have certain attributes that further specify the type of connection. *Connection points* are *parts* of the corresponding *arc* or *directed arc*.

Relations

- *Connection point* is a subclass of *connector*.
- A *connection point* can only be directly connected to a *port*.
- A *connection point* cannot be directly connected to more than one *port*.
- A *connection point* is part of at least one *arc*.
- A *connection point* can only be a direct part of an *arc*.

Usage

Connection points constrain the connections that an *arc* can have to a *node*. A connection between an *arc* and a *node* is feasible only if the attributes of the *connection point* and the corresponding *port* match. Special care must be taken that the *connection point* is situated at the same decomposition level as the connected *port* (cf. Fig. 27)[20].

Connector

Description

A *connector* represents the interface through which an *object* can be connected to another *object*. Typically, the possible connections of the *connector* are further constrained, for instance by postulating that certain properties of the connected *connectors* need to match for a feasible connection.

Relations

- *Connector* is a subclass of *part*.
- A *connection point* can only be directly connected to a *connector*.
- A *connection point* cannot be directly connected to more than one *connector*.

---

[19] Implementation advice: Unfortunately, this decomposition axiom cannot be properly represented in OWL, as OWL does currently not support qualified cardinality restrictions.

Once qualified cardinality constraints become available, the following restrictions should be implemented:

- An arc is composed of either zero or at least two arcs and one interconnecting node.
- An arc cannot have more than two connection points as direct parts.

[20] Implementation advice: The following restriction has not been implemented as it caused severe performance problems:

- A connection point can only be a part of an arc or of an node that is a direct part of an arc (i.e., an internal node).

Directed arc

Description

Directed arc is a specialization of *arc* and represents likewise the connecting element between *nodes*. However, the usage of *directed arc* implies the indication of a directed connection.

Relations

- A *directed arc* is a subclass of *arc*.
- A *directed arc* enters either zero or one *node*.
- A *directed arc* leaves either zero or one *node*.
- The isDirectlyConnectedTo relation is not applicable to a *directed arc*.

Node

Description

Node is a specialization of *object* and is used to model the crucial elements (joints) which are connected by *arcs*.

Relations

- *Node* is a subclass of *object*.
- A *node* may have only *directed arcs* as inputs.
- A *node* may have only *directed arcs* as outputs.
- A *node* can only be directly connected to *arcs*.
- A *node* can only be a direct part of a *node* or *arc*.
- A *node* can only have *nodes* or *arcs* or *ports* as direct parts.
- A *node* can only have *nodes* or *arcs* or *ports* or *connection points* as parts.

Usage

If a *node* is decomposed into sub-*nodes* (connected by internal *arcs*), these sub-*nodes* can only be connected to external *arcs* that are on the same level of decomposition. This in turn means that there must be at least three sub-elements – two sub-*nodes* and one internal *arc*. Unfortunately, this decomposition axiom cannot properly represented in OWL, as OWL does currently not support qualified cardinality restrictions[21].

Port

Description

Ports represents the interfaces through which *nodes* are connected to *arcs*. A *port* may have certain attributes that characterize the type of connection.

Relations

- *Port* is a subclass of *connector*.
- A *port* can only be directly connected to a *connection point*.
- A *port* cannot be directly connected to more than one *connection point*.
- A *port* is part of at least one *node*.

---

[21] Implementation advice: Once qualified cardinality constraints become available, the following restriction should be implemented:

- A node is composed of either zero or at least two nodes and one interconnecting arc.

- A *port* can only be a direct part of a *node*.

Usage

Ports constrain the number and type of connections that a *node* can have: A *node* can only be connected to as many *arcs* as it has designated *ports*. Moreover, a connection between a *node* and an *arc* is feasible only if the attributes of the *port* and the corresponding *connection point* match.

Ports may have an isDirectPartOf relation instead of the isPartOf relation to sub-*nodes* on the same level of decomposition. However, care must be taken that the *port* is situated at the same decomposition level as the connected *connection point* (cf. Fig. 27)[22].

Relations

enters

Description

The relation enters connects an incoming *directed arc* to its target *node*.

Characteristics

- Specialization of isPredecessorOf
- Domain: *directed arc*
- Range: *node*
- Inverse: hasInput

hasInput

Description

The relation hasInput connects a *node* to an incoming *directed arc*.

Characteristics

- Specialization of isSuccessorOf.
- Domain: *node*
- Range: *directed arc*
- Inverse: enters

hasOutput

Description

The relation hasOutput connects a *node* to an outgoing *directed arc*.

Characteristics

- Specialization of isPredecessorOf
- Domain: *node*
- Range: *directed arc*
- Inverse: leaves

---

[22] Implementation advice: The following restriction has not been implemented as it caused severe performance problems:

- A port can only be a part of a node or an arc that is a direct part of a node (i.e., an internal arc).

isConnectedTo

Description

The relation isConnectedTo represents topological connectivity between *objects.*

Characteristics

- Specialization of inter-objectRelation
- Domain: *object*.
- Range: *object*.
- Symmetric
- Transitive

isDirectlyConnectedTo

Description

The relation isDirectlyConnectedTo denotes the direct topological connectedness of two *objects.*

Characteristics

- Specialization of isConnectedTo
- Domain: *object*
- Range: *object*
- Symmetric

isSuccessorOf

Description

The relation isSuccessorOf identifies all *nodes* and *directed arcs* that are successors of the considered one*.*

Characteristics

- Specialization of isDirectlyConnectedTo
- Domain: *node* or *directed arc*
- Range: *node* or *directed arc*
- Inverse: isPredecessorOf
- Transitive

isPredecessorOf

Description

The relation isPredecessorOf identifies all *nodes* and *directed arcs* that are predecessors of the considered one.

Characteristics

- Specialization of isDirectlyConnectedTo
- Domain: *node* or *directed arc*
- Range: *node* or *directed arc*
- Inverse: isSuccessorOf
- Transitive

leaves

Description

The relation leaves connects an outgoing *directed arc* to its source *node*.

Characteristics

- Specialization of isSuccessorOf
- Domain: *directed arc*
- Range: *node*
- Inverse: hasOutput

sameAs

Description

The relation denotes a correspondence between an *arc* and its placeholder in a decomposition hierarchy.

Characteristics

- Specialization of inter-objectRelation
- Domain: *arc*
- Range: an *arc* that is directly connected to a *node*
- Symmetric

6. Data Structures

The partial model **data_structures** provides a design pattern for the representation of customary data structures. The below pattern provide some data structures which occur repeatedly with special relevance to the modeling of OntoCAPE. However, these data structures are fully consistent with those typically defined and applied in computer science (e.g., Black 2004). It comprises the ontology modules *binary_tree*, *array*, *linked_list*, *multiset*, and *loop*, which are presented in the following. Note that the design patterns incorporate transitive, inverse relations, which may cause performance problems (cf. Sec. 4). Thus, for large-scale applications, it might prove necessary to abstain from implementing the inverse relations.

6.1. Binary Tree

A *binary tree* is a tree-like data structure that is formed by a set of linked *nodes*. A node can have zero, one, or two *child nodes*. Each child node is identified as either the *left* child or the *right* child. Fig. 33 shows an exemplary binary tree. The topmost[23] element of the tree is called the *root node* (node A in Fig. 33). A node that has a child is called the child's *parent node*. Except for the root node, each node has one parent node.

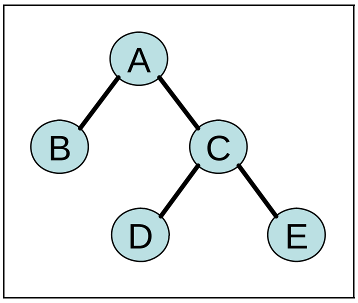

Fig. 33: Example of a binary tree

Nodes that lie below a certain node (i.e., its children, grandchildren, etc.) are called the *descendents* of this node. Similarly, a node’s *ancestors* are the nodes that are traversed when moving up the tree (i.e., the node’s parent, grandparent, etc.). In Fig. 33, for example, the nodes B, C, D, and E are descendents of node A; node E has the ancestors A and C.

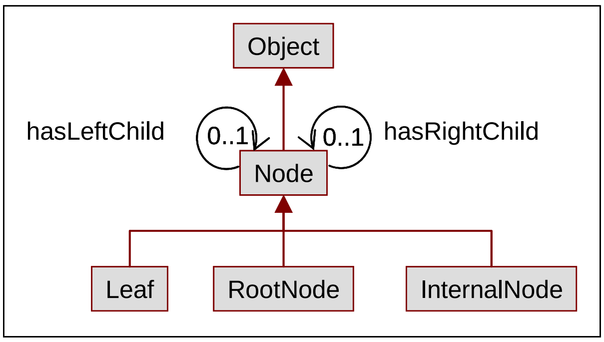

Fig. 34: Design pattern for the representation of binary trees

Fig. 2 and Fig. 35 illustrate how a binary tree is represented in the Meta Model. *Node* is the basic element for the construction of a tree. Three specializations of *node* are introduced:

- the *root node* is a node without a parent;
- an *internal node* is a node that has both a parent node and a child node;

[23] By convention, binary trees are depicted top-down.

- a *leaf* is a node without children.

A *node* is linked to its child *nodes* via the relations hasLeftChild and hasRightChild. The relation hasChild subsumes these two relations, as shown in Fig. 35. The relation hasParent is defined as the inverse of hasChild. It has two specializations: isLeftChildOf, which is the inverse of hasLeftChild, and isRightChildOf, which is the inverse of hasRightChild. Finally, the relation hasAncestor and its inverse hasDescendent are introduced to denote the ancestors and descendents of a particular *node*.

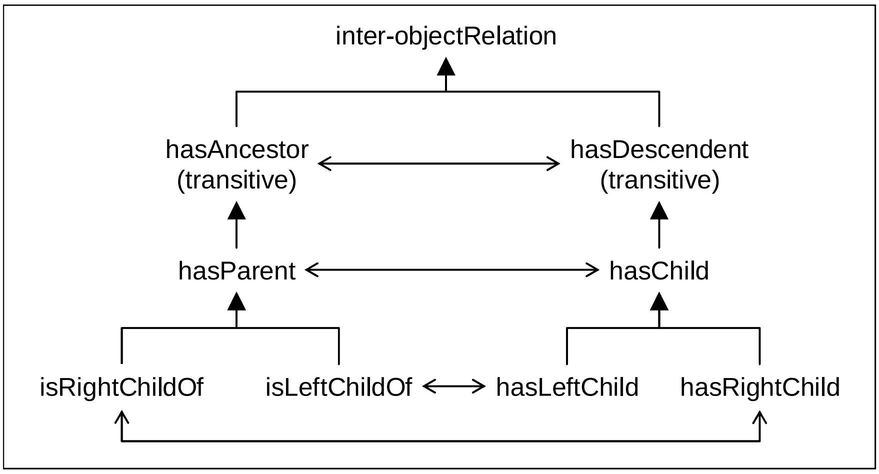

Fig. 35: Relations for the representation of binary trees

An application example is shown in Fig. 36, which uses the above concepts to represent the tree depicted in Fig. 33. Note that only the relations hasLeftChild and hasRightChild need to be explicitly defined between the *nodes*. All the other relations (i.e., the parent, ancestors, and descendents of a particular *node*) can be automatically inferred by a reasoner.

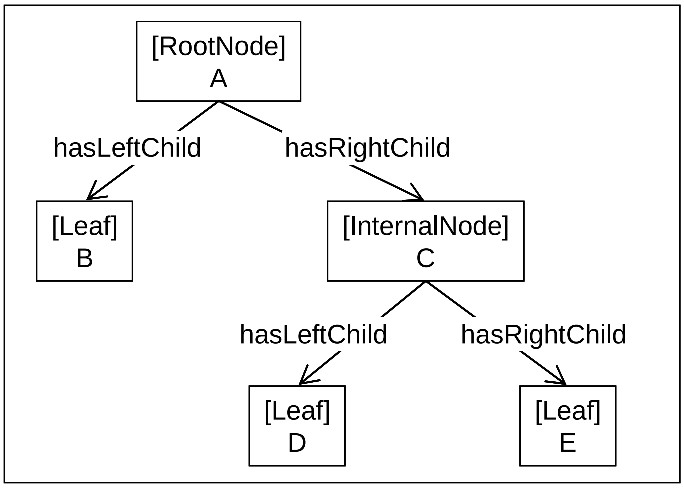

Fig. 36: Application example

Note that a *node* may have more than one parent *node*, if that particular *node* forms part of more than one binary tree.

Usage

The design pattern *binary_tree* complies with the following competency questions:

- Query for all *nodes* of a particular tree (which is identified through its *root node*).
- Query for the *leaves* of a particular tree (which is identified through its *root node*).
- Query for the direct children of a particular *node*.

- Query for the left/right child of a particular *node*.
- Query for the descendents of a particular *node*.
- Query for the direct parent of a particular *node*.
- Query for the ancestors of a particular *node*.
- Query for the *root node* of a particular tree (which is identified through one of its *nodes*).

A possible application of the *binary_tree* pattern is the representation of mathematical expressions. The *leaves* of such an expression tree denote the operands in the expression, and the *internal nodes* denote the operators[24].

Concept Descriptions

Individual concepts of the module *binary_tree* are defined below:

Classes

Internal node

Description

An *internal node* is a *node* that has one parent and at least one child.

Definiton

A *node* that has both a parent *node* and a child *node*.

Relations

- An *internal node* is a specialization of a *node*.
- An *internal node* has at least one parent *node*.
- An *internal node* has one or two child *nodes*.

Leaf

Description

A *leaf* is a *node* without any children.

Definiton

A *node* that has no child nodes.

Relations

- A *leaf* is a specialization of a *node*.
- A *leaf* has at least one parent *node*.
- A *leaf* has zero child *nodes*.

---

[24] Implementation advice: In principle, it is not necessary to explicitly declare a node to be a root node or leaf, as this can be inferred by a reasoner. However, some reasoners cannot evaluate this kind of statement (i.e., that a node has zero ancestors/descendents) – in this case, root node and leaves must be explicitly identified if required for an application. Internal nodes are automatically found by a reasoner.

Node

Description

A *node* is the basic element of a binary tree. It can be linked to up to two child *nodes*.

Definition

A *node* is either a *leaf* or a *root node* or an *internal node*.

Relations

- A *node* is a specialization of *object*.
- A *node* has zero or one left child *node*.
- A *node* has zero or one right child *node*.
- A node may have some parent *node*.

Root node

Description

A *root node* is the root element of a binary tree. All other *nodes* are descendents of the *root node*.

Definiton

A *node* without any parent.

Relations

- A *root node* is a specialization of a *node*.
- A *root node* has at least one child *node*.
- A *root node* has no parent *node*.

Relations

hasAncestor

Description

The ancestors of a *node* are the *nodes* that precede the current *node* in the tree (i.e., the *node*'s parent, grandparent, etc.).

Characteristics

- Specialization of inter-objectRelation
- Domain: *node*
- Range: *node*
- Inverse: hasDescendent
- Transitive

hasChild

Description

The relation hasChild points to the children of a *node*; it subsumes the relations hasLeftChild and hasRightChild.

<u>Characteristics</u>

- Specialization of hasDescendent
- Domain: *node*
- Range: *node*
- Inverse: hasParent

hasDescendent

<u>Description</u>

The descendents of a *node* are the *nodes* that succeed the current *node* in the tree (i.e., the *node*'s children, grandchildren, etc.).

<u>Characteristics</u>

- Specialization of inter-objectRelation
- Domain: *node*
- Range: *node*
- Inverse: hasAncestor
- Transitive

hasLeftChild

<u>Description</u>

The relation hasLeftChild links a parent *node* to its left child *node*.

<u>Characteristics</u>

- Specialization of hasChild
- Domain: *node*
- Range: *node*

hasParent

<u>Description</u>

The relation hasParent denotes the parent of a *node*.

<u>Characteristics</u>

- Specialization of hasAncestor
- Domain: *node*
- Range: *node*
- Inverse: hasChild

hasRightChild

<u>Description</u>

The relation hasRightChild links a parent *node* to its right child *node*.

<u>Characteristics</u>

- Specialization of hasChild

- Domain: *node*
- Range: *node*

isLeftChildOf

Description

The relation isLeftChildOf points from the left child *node* to its parent *node*.

Characteristics

- Specialization of hasParent
- Domain: *node*
- Range: *node*

isRightChildOf

Description

The relation isRightChildOf points from the right child *node* to its parent *node*.

Characteristics

- Specialization of hasParent
- Domain: *node*
- Range: *node*

6.2. Multiset

A *multiset* differs from an ordinary set in that there may be multiple appearances of the same element. For example, the multiset {a, a, b, b, b, c} has two appearances of element a and three appearances of element b.

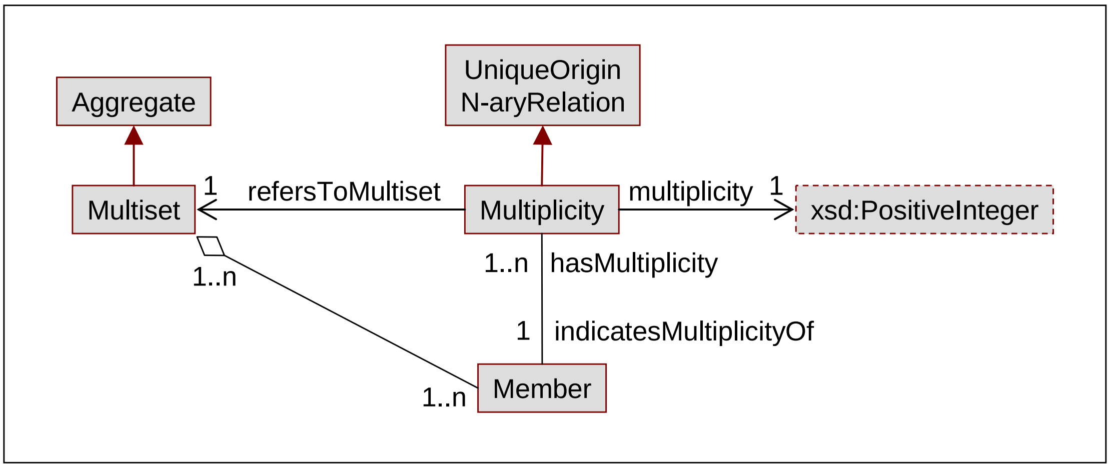

Fig. 37: Design pattern for the representation of multisets

In the Meta Model, a *multiset* is modeled as a special type of *aggregate* (cf. Fig. 37). Its elements, which are direct parts of the *multiset*, are called *members*. Each *member* has a *multiplicity* that indicates the number of its appearances within the multiset. A *member* may be a member of more than one *multiset*; in this case, the *member* must have one *multiplicity* for each of these memberships. For this reason, the *multiplicity* is modeled as a *unique origin n-ary relation* that relates the various *multiplicities* of a *member* to the respective *multisets*.

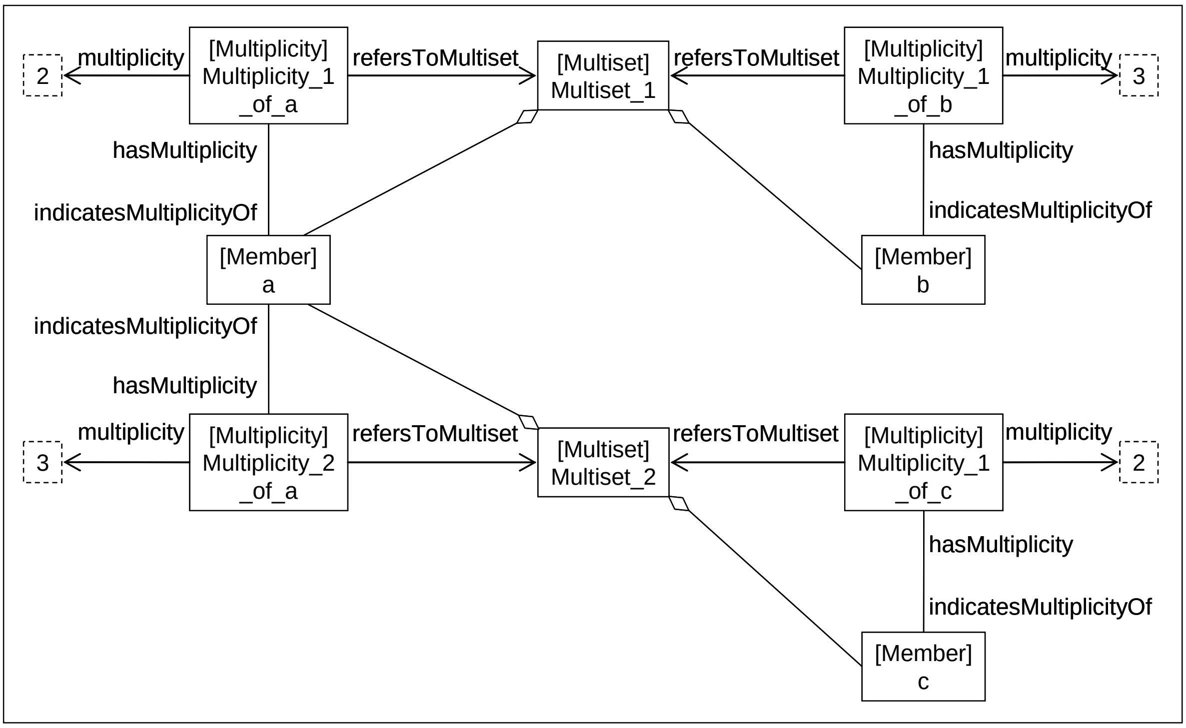

Fig. 38: Application example: element a is a member of both multisets

An application example is given in Fig. 38. It shows the ontological representation of two *multisets*:

- Multiset 1 = {a, a, b, b, b}, and
- Multiset 2 = {a, a, a, c, c}.

Obviously, individual a is a *member* of both *multisets*. a has a *multiplicity* of two in Multiset 1, and a *multiplicity* of three in Multiset 2. The relation refersToMultiset indicates which *multiplicity* is related to which *multiset*.

Usage

The design pattern *multiset* complies with the following competency questions:

- Query for the *members* of a particular *multiset*.
- Query for the *multiplicity* that a *member* has in a particular *multiset*.
- Query for the *multiset* of which a member is a part.

A *multiset* may be used as a shorthand to specify an *object* that is composed of alike parts (in this context, 'alike' means that the parts share certain characteristic features). Instead of specifying all these parts individually, it is sufficient to describe one representative part (*member*) and indicate its *multiplicity*. For example, a distillation column can be thermodynamically characterized by (1) describing the VLE on one tray and (2) indicating the total number of trays.

Concept Descriptions

Individual concepts of the module *multiset* are defined below.

Class Descriptions

Multiplicity

Description

The *multiplicity* of a *member* indicates the number of its appearances in the associated *multiset*.

Definition

A *multiplicity* indicates the multiplicity of some *member*.

Relations

- *Multiplicity* is a specialization of *unique origin n-ary Relation*.
- A *multiplicity* indicates the multiplicity of exactly one *member*.
- A *multiplicity* refers to exactly one *multiset*.
- A *multiplicity* has exactly one positive integer value that indicates the numerical value of the multiplicity.

Usage

Instances of *multiplicity* should be named according to the following convention:

Multiplicity_<unique ID>_of_<name of member>.

Example: Multiplicity_01_of_a.

Multiset

Description

A *multiset* differs from an ordinary *aggregate* in that each of its parts (*members*) has an associated *multiplicity*, which indicates the number of its appearances in the *multiset*.

Definition

A *multiset* is an *aggregate* that has at least one *member*.

Relations

- *Multiset* is a specialization of *aggregate*.
- A *multiset* has at least one *member*.
- A *multiset* has only *members* as direct parts.

Member

Description

A *member* is a direct part of a *multiset*; it has a *multiplicity* that indicates the number of its appearances in the *multiset*.

Definition

A *member* is a *part* that has a *multiplicity*.

Relations

- *Member* is a specialization of *part*.
- A *member* is a direct part of at least one *multiset*.
- A *member* is a direct part of only a *multiset*.
- A *member* has at least one *multiplicity*.

Relations

hasMultiplicity

Description

The relation hasMultiplicity points from a *member* to a *multiplicity* that indicates the number of its appearances in a particular *multiset*.

Characteristics

- Specialization of isOriginOf
- Domain: *member*
- Range: *multiplicity*
- Inverse: indicatesMultiplicityOf
- Inverse functional

indicatesMultiplicityOf

Description

The relation indicatesMultiplicityOf links a *multiplicity* to the corresponding *member*.

Characteristics

- Specialization of hasOrigin
- Domain: *multiplicity*
- Range: *member*
- Inverse: hasMultiplicity
- Functional

refersToMultiset

Description

The relation refersToMultiset assigns a *multiplicity* to the corresponding *multiset*.

Characteristics

- Specialization of hasTargetObject
- Domain: *multiplicity*
- Range: *multiset*

Usage

An *object* can be a *member* of several multisets. In this case, the object has several *multiplicities*, and the relation refersToMultiset is used to indicate which of these *multiplicity* refers to which *multiset*.

Attributes

multiplicity

Description

The attribute multiplicity indicates the numerical value of a *multiplicity*.

Characteristics

- Specialization of relationAttribute
- Domain: *multiplicity*
- Datatype: positiveInteger (built-in XML Schema Datatype)
- Functional

6.3. Array

An *array* holds an ordered collection of objects, which are called the *elements* of the array. Similar to a *multiset*, an element can have multiple appearances in the array. The elements are ordered by an *index*, which specifies the position of an *element* within the array through a consecutive sequence of integer values. Individual elements can be accessed via their respective index values.

Fig. 39 shows the design pattern that defines an array in the Meta Model. An *array* is a specialization of a *composite object* which is composed of two or more *elements*. The position of an *element* within the *array* is specified by the *index*. An *index* is a *coequal n-ray relation* between an *array*, one of its *elements*, and the integer attribute value that denotes the position of the *element* in the *array*.

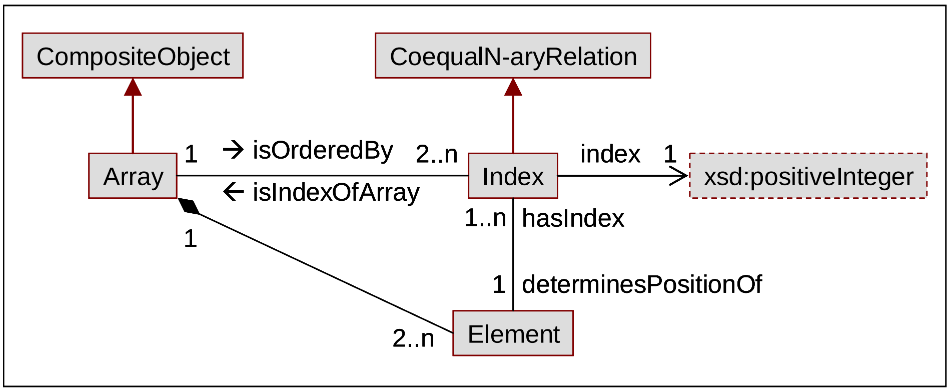

Fig. 39: Design pattern for the representation of arrays

An application example of the *array* design pattern is given in Fig. 40. The *array* A[i] has the *elements* x and y. The *index* of x (Index_of_x) has an index value of 1, whereas the *index* of y (Index_of_y) has an index value of 2. Thus, x is the first *element* of A[i], and y is the second one.

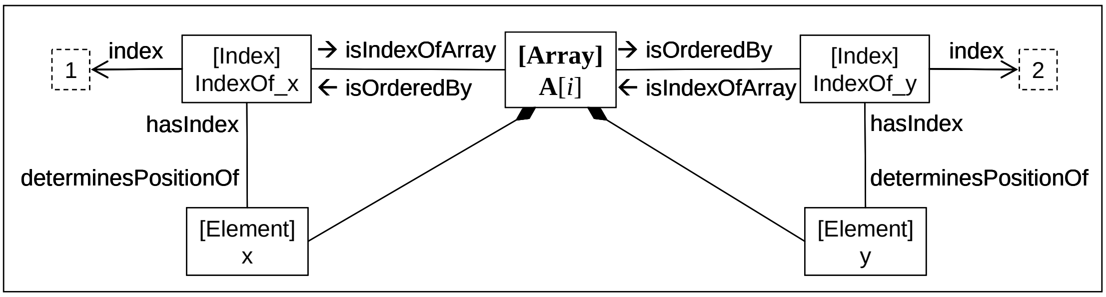

Fig. 40: Application example: representation of an array **A**[*i*] with elements **A**[1] = *x* and **A**[2] = *y*

Usage

The design pattern *array* complies with the following competency questions:

- Query for an *element* with a particular index value.
- Query for the *index* of a particular *element*.
- Query for all *elements* of an *array*.
- Query for the *array* to which a particular *elements* belongs.
- Query for the *array* to which a particular *index* belongs.

In the ontology OntoCAPE, the design pattern *array* is applied to model tensor quantities, such as vectors and matrices.

Concept Descriptions

Individual concepts of the module *array* are defined below.

Class Descriptions

Array

Description

An *array* is an ordered list that is composed of two or more *elements*. The position of an *element* within the *array* is specified by the *index*.

Relations

- *Array* is a *composite object*
- An *array* is composed of two or more *elements*
- An *array* is ordered by two or more instances of the *index* class.

Element

Description

An *element* is part of an *array*. Its position within the *array* is determined by an *index*.

Relations

- *Element* is a *part of a composite object*
- An *element* is part of exactly one *array*
- An *element* is ordered by at least one *index*

Index

Description

An *index* represents the *coequal n-ary relation* between an *array*, one of its *elements*, and the integer attribute value that denotes the position of the *element* in the *array*.

Definition

An *index* determines the position of some *element*.

Relations

- *Index* is a specialization of *coequal n-ary relation*.
- An *index* determines the position of exactly one *element*.
- An *index* is index of exactly one *array*.
- The numerical value of the *index* is specified by the index attribute, which takes exactly one value of type integer.

Usage

Instances of *index* should be named according to the following convention:

<name of index class>_of_<name of element>.

Example: Index_of_ElementX.

Relations

determinesPositionOf

<u>Description</u>

The one-to-one relation between an *index* and the corresponding *element*.

<u>Characteristics</u>

- Specialization of involvesObject
- Domain: *Index*
- Range: *Element*
- Inverse: hasIndex
- Functional

hasIndex

<u>Description</u>

The one-to-one relation between an *element* and its *index*.

<u>Characteristics</u>

- Specialization of isInvolvedInN-aryRelation
- Domain: *Element*
- Range: *Index*
- Inverse: determinesPositionOf
- Inverse functional

isIndexOfArray

<u>Description</u>

The relation isIndexOfArray points form an *index* to the associated *array*

<u>Characteristics</u>

- Specialization of involvesObjec
- Domain: *index*
- Range: *array*
- Inverse: isOrderedBy
- Functional

isOrderedBy

<u>Description</u>

The relation isOrderedBy points from an *array* and to the sorting *index*.

<u>Characteristics</u>

- Specialization of isInvolvedInN-aryRelation
- Domain: *array*
- Range: *index*
- Inverse: isIndexOfArray

- Inverse functional

Attributes

index

Description

The attribute index indicates the numerical value of an *index*.

Characteristics

- Specialization of relationAttribute
- Domain: *index*
- Datatype: integer[25] (built-in XML Schema Datatype)
- Functional

6.4. Linked List

Similar to an array, a *linked list* is an ordered collection of objects. It is formed by a sequence of *list elements*, each pointing to the next (and possibly the previous) element in the list. List elements can be inserted and removed at any point in the list. Unlike an array, a linked list does not allow random access[26].

In the Meta Model, a *linked list* is modeled as a specialization of a *composite object* that is composed of two or more *list elements*. A *list element* points to the next as well as to the previous *list element* through the relations nextElement and previousElement, respectively. Three disjoint subclasses of *list element* are introduced:

- the *first element* of the list, which is a *list element* that does not point to a previous *list element*; it must have one next *list element*;
- the *last element* of the list, which is a *list element* that does not point to a next *list element*; it must have one previous *list element*; and
- *internal element*, which is defined as a *list element* that points to both a previous and a next *list element*.

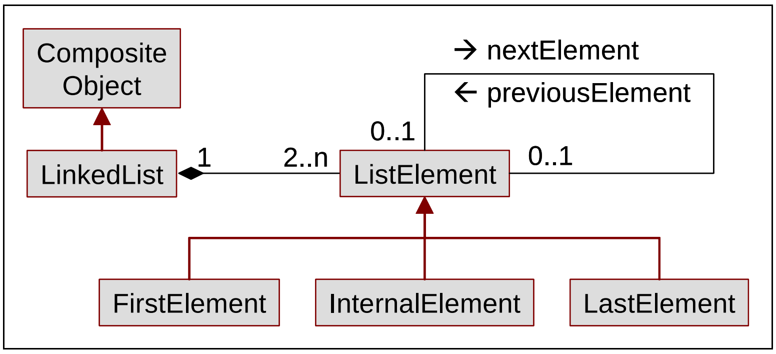

Fig. 41: Design pattern for the representation of a linked list

---

[25] For the current version of OntoCAPE we have assumed to apply only inter values for indexing. However, without loss of generality a distinction may be made between orderable indices and un-orderable indices.

[26] Random access is the ability to access any particular element in the list in constant time

Usage

An application example is given in Fig. 42. It demonstrates how to represent a *linked list* with the *list elements* x, y ,and z; x is the *first element*, y is an *internal element*, and z is the *last element* of the *linked list*.

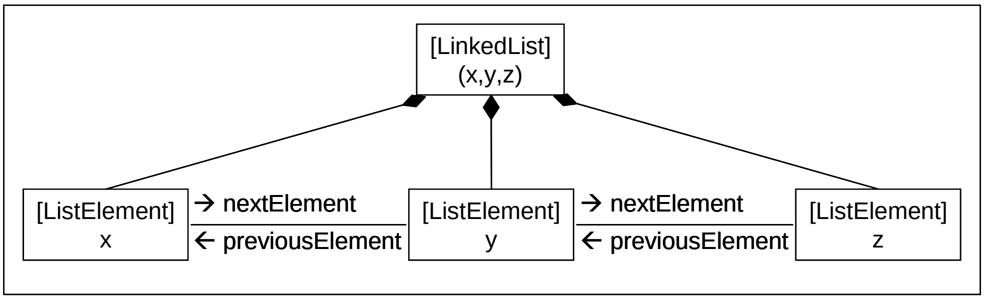

Fig. 42: Application example – a linked list with elements x, y, and z

The design pattern *linked list* complies with the following competency questions:

- Query for the *first element* of a particular *linked list*.
- Query for the second (third, fourth, …) *list element* of a particular *linked list*.
- Query for the *last element* of a particular *linked list*.
- Query for all *list elements*
- Query for the *list element* succeeding a particular *list element*.
- Query for the *list element* preceding a particular *list element*.

Note that a *list element* cannot have multiple appearances in a *linked list*[27].

Concept Descriptions

Individual concepts of the module *linked list* are defined below.

Class Descriptions

First element

Description

[27] Implementation advice: In principle, it is not necessary to declare the first element and last element of a linked list explicitly, as they can be automatically found by a reasoner. This facilitates to add and remove list elements at arbitrary positions. However, some reasoners cannot evaluate the definition – in this case, the first element and last element must be explicitly defined.

The following ontological assertions cause problems with the reasoner RacerPro and have therefore been omitted in the current version of the Meta Model:

- A list element is either a first element or an internal element or a last element.
- A linked list is composed of some first element (for some reason does the corresponding statement "A linked list is composed of some last element not cause any trouble).
- nextElement is only of type internal element or last element (requires too much computation time).
- previousElement is only of type internal element or first element (as above).
- first element is a list element that does not have a previous list element.
- last element is a list element that does not have a next list element.

The first *list element* of a *linked list*.

<u>Relations</u>

- *First element* is a subclass of *list element*.
- A *first element* points to one next *list element*.
- A *first element* does not point to a previous *list element*.

Internal element

<u>Description</u>

A *list element* that is neither the first nor the last element of a *linked list*.

<u>Definition</u>

Internal element is a *list element* that points to both a next and a pervious *list element*.

<u>Relations</u>

- *Internal element* is a subclass of *list element*.
- An *internal element* points to one next *list element*.
- An *internal element* points to one previous *list element*.

Last element

<u>Description</u>

The last *list element* of a *linked list*.

<u>Relations</u>

- *Last element* is a subclass of *list element*.
- A *last element* does not point to a next *list element*.
- A *last element* points to one previous *list element*.

Linked list

<u>Description</u>

A *linked list* is formed by a sequence of *list elements*, each pointing to the next as well as to the previous *list element*.

<u>Relations</u>

- *Linked List* is a subclass of *composite object*.
- A *linked list* is composed of at least two *list elements*.
- A *linked list* can only be composed of *list elements*.

List element

<u>Description</u>

A *list element* is an element of a *linked list*; it may point to a next as well as to a previous *list element*.

<u>Relations</u>

- *List element* is a subclass of *part of composite object*.
- A *list element* is exclusively part of a *linked list*.
- A *list element* points to zero or one next *list element*.
- A *list element* points to zero or one previous *list element*.

Relations

nextElement

Description

The relation nextElement points from a *list element* to the next *list element*.

Characteristics

- Specialization of inter-objectRelation
- Domain: *list element*
- Range: *list element*
- Inverse: previousElement

previousElement

Description

The relation previousElement points from a *list element* to the previous *list element*.

Characteristics

- Specialization of inter-objectRelation
- Domain: *list element*
- Range: *list element*

6.5. Loop

The design pattern *loop*[28] introduces a shorthand for representing structures that consist of repetitive, interlinked *objects*. This is best explained by means of an example: Consider the structure displayed in Fig. 43: the individuals O1 to O5 are sequentially connected via the relation 'o'. …Furthermore, O1 is linked to A1, O2 is linked to A2, and so forth, via the relation 'a'. Also, each of the O_i is linked to individual X via the relation 'x'; thus, X represents a common feature of the O_i. Individual R is linked to O1, and O5 is linked to S; thus, R and S represent the endpoint conditions of the structure.

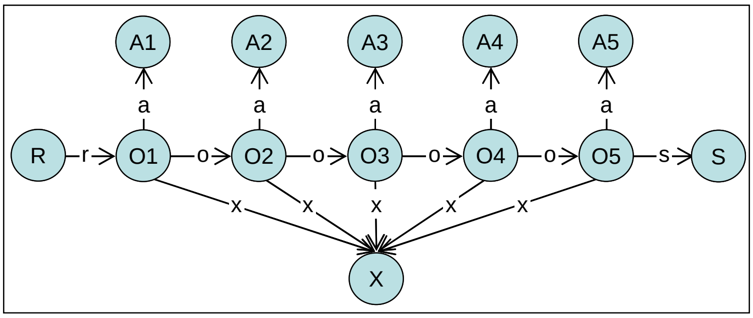

Fig. 43: Repetitive, interlinked structure

Instead of defining this structure explicitly, it can be represented as indicated in Fig. 44: First, a Loop is introduced, which has 5 numbersOfIteration. Several individuals are linked to the Loop. The relation 'statementFor_i' identifies those individuals that depend on the iterations *i*; in this example, these are the individuals O_i and A_i. This means that, for each iteration *i*, one O_i and one A_i exists.

[28] The name 'forloop' is chosen because the syntax used to represent the *loop* pattern is similar to that of a 'for loop' in a programming language.

O_i and A_i are connected via the relation ‘a’; consequently, one ‘a’ is established between each occurrence of O_i and A_i.

The relation ‘x’ points from O_i to individual X, which is not linked to the Loop; thus, each occurrence of O_i points to the same X.

For *i* = 1 and *i* = 5, the O_i have connections to R and S, respectively. To represent these additional connections, the individuals O_1 and O_N are introduced. O_1 is linked to R, and O_N is linked to S. To indicate that O_1 and O_N represent the first and last occurrence of O_i in the Loop, they are (1) linked to O_i via the relation sameObject, and (2) connected to the Loop by means of the relations initialStatement and finalStatement.

Finally the relations between the O_i must be specified. To this end, the individual O_i+1 is introduced, which represents the occurrence of O_i in the iteration (*i* + 1). The semantics of O_i+1 is explicitly defined by (1) establishing a sameObject relationship between O_i and O_i+1, and (2) linking O_i+1 to the Loop via the relation statementFor_iPlus1. Lastly, the relation ‘o’ is declared between O_i and O_i+1, thus indicating that ‘o’ exists in between the different O_i.

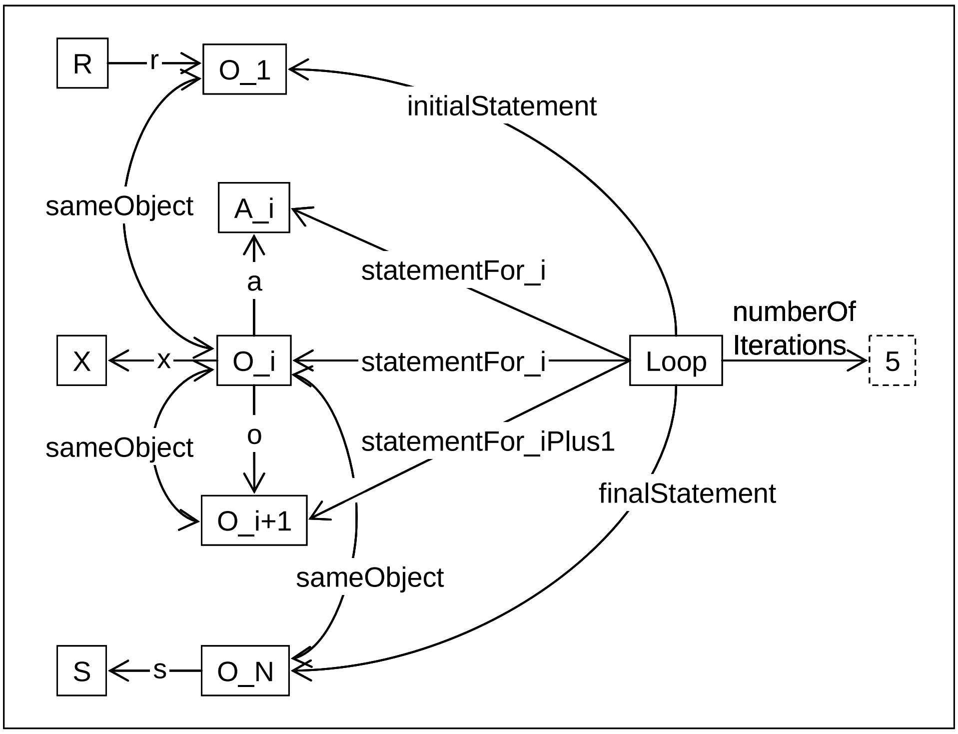

Fig. 44: Representation of the structure shown in Fig. 43 by using the *loop* design pattern

Fig. 45 defines the classes and relations that are required to establish a *loop* pattern as the one exemplarily shown above. A *for loop* must have at least one statementFor_i. Additionally, a *for loop* may have an initialStatement, a finalStatement, and a statementFor_iPlus1. Any *object* that is linked to a *for loop* via one of these latter relations must have a sameObject relation to an *object* that is linked to a *for loop* via a statementFor_i relation. This is guaranteed by logical constraints imposed on the *for loop* class.

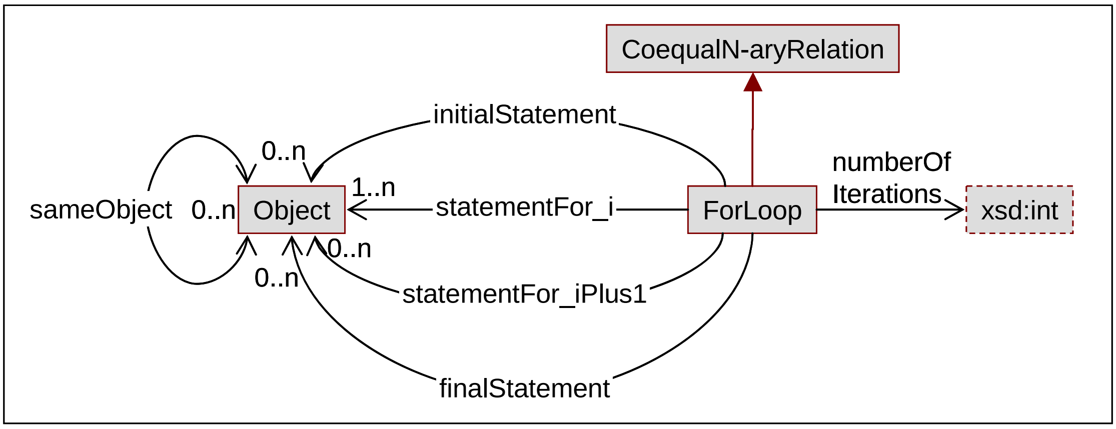

Fig. 45: Class diagram of the *loop* design pattern

The relations of the *loop* design pattern are given in Fig. 46.

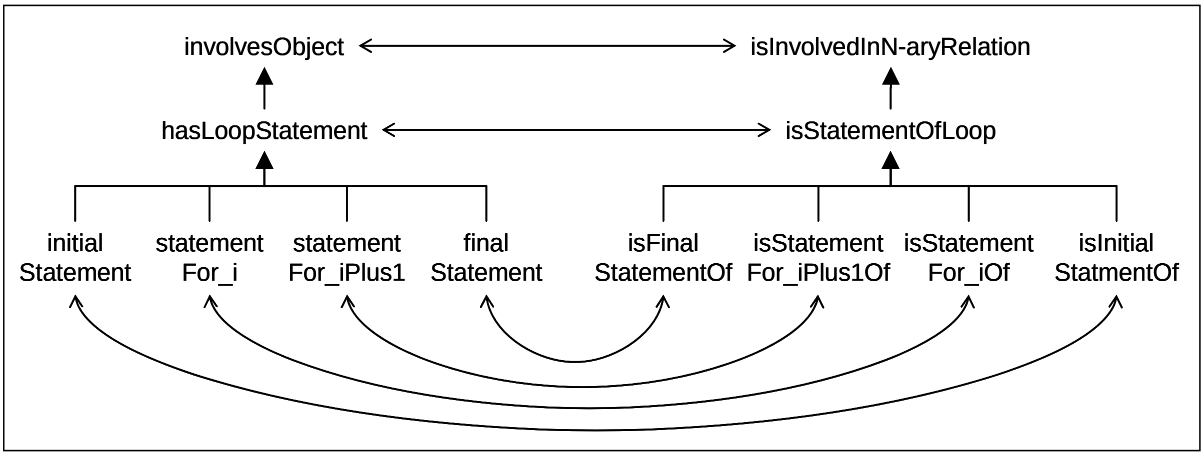

Fig. 46: Hierarchy of relations introduced in the *loop* ontology module

Usage

The *loop* pattern is used to represent structures that consist of repetitive, interlinked *objects*.

If the structure represented by the *for loop* is part of an *aggregate*, then all *objects* that are linked to the *for loop* should be declared as parts of the *aggregate*.

Properties (i.e. relations or attributes) that are common to all *objects* of the structure must only be declared once, namely as properties of the individual that linked to the *for loop* via a statementFor_i relation. Individuals that are linked to the *for loop* via a ininitalStatement, finalStatement, or statementFor_iPlus1 relation should carry only those properties that are specific for the respective iteration.

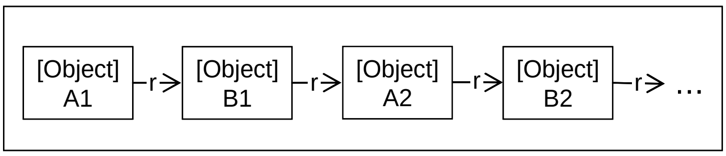

Fig. 47: Structure consisting of alternating As and Bs

Note that the *loop* pattern also allows for the representation of structures that consist of alternating elements. An example of such a structure composed of alternating As and Bs is given in Fig. 47. The equivalent *loop* pattern is shown in Fig. 48.

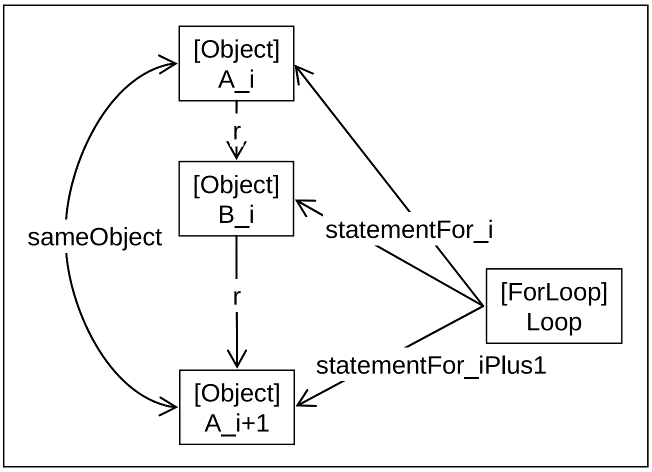

Fig. 48: Loop pattern representing the structure displayed in Fig. 47

Concept Definitions

Classes

For loop

<u>Description</u>

A *for loop* is used to represent structures that consist of repetitive, interlinked *objects*.

<u>Relations</u>

- *For loop* is a subclass of *coequal n-ary relation*.
- A *for loop* has at least one statementFor_i.
- A *for loop* may have some statementFor_i+1, which must be connected to statementFor_i via a sameObject relation.
- A *for loop* may have some initialStatement, which must be connected to statementFor_i via a sameObject relation.
- A *for loop* may have a finalStatement, which must be connected to statementFor_i via a sameObject relation.
- A *for loop* has exactly one numberOfIterations.

Relations

finalStatment

<u>Description</u>

Denotes the final statement in a *for loop*.

<u>Characteristics</u>

- Specialization of hasLoopStatement
- Domain: *for loop*
- Range: *object*
- Inverse: isFinalStatementOf

hasLoopStatement

Description

Subsumes the different statements of a *for loop*.

Characteristics

- Specialization of involvesObject
- Domain: *for loop*
- Range: *object*
- Inverse: isStatementOfLoop

ininitalStatement

Description

Denotes the initial statement in a *for loop*.

Characteristics

- Specialization of hasLoopStatement
- Domain: *for loop*
- Range: *object*
- Inverse: isInitialStatementOf

isFinalStatmentOf

Description

Denotes the final statement in a *for loop*.

Characteristics

- Specialization of isStatementOfLoop
- Domain: *object*
- Range: *for loop*
- Inverse: finalStatement

isIninitalStatementOf

Description

Denotes the initial statement in a *for loop*.

Characteristics

- Specialization of isStatementOfLoop
- Domain: *object*
- Range: *for loop*
- Inverse: initialStatement

isStatementFor_iOf

Description

Denotes the *objects* that appear in each iteration of a *for loop*.

Characteristics

- Specialization of isStatementOfLoop

- Domain: *object*
- Range: *for loop*
- Inverse: statementFor_i

isStatementFor_iPlus1Of

Description

Denotes the *objects* that in the next iteration of a *for loop*.

Characteristics

- Specialization of isStatementOfLoop
- Domain: *object*
- Range: *for loop*
- Inverse: statementFor_iPlus1

isStatementOfLoop

Description

Subsumes all the individuals that represent statements in a *for loop*.

Characteristics

- Specialization of isInvolvedInN-aryrelation
- Domain: *object*
- Range: *for loop*
- Inverse: hasLoopStatement

sameObject

Description

Identity relation between an *object* involved in a statementFor_i and an *object* that appears in an initialStatement, a finalStatement, or a statementFor_iPlus1.

Characteristics

- Specialization of inter-objectRelation
- Domain: *object*
- Range: *object*
- Symmetric

statementFor_i

Description

Denotes the *objects* that appear in each iteration of a *for loop*.

Characteristics

- Specialization of hasLoopStatement
- Domain: *for loop*
- Range: *object*
- Inverse: isStatementFor_iOf

statementFor_iPlus1

Description

Denotes the *objects* that in the next iteration of a *for loop*.

Characteristics

- Specialization of hasLoopStatement
- Domain: *for loop*
- Range: *object*
- Inverse: isStatementFor_iPlus1Of

Attributes

numberOfIterations

Description

Indicates the number of iterations of a particular *for loop*.

Characteristics

- Specialization of relationAttribute
- Domain: *for loop*
- Datatype: positiveInteger (built-in XML Schema Datatype)

References

Atkinson C, Kühne T (2002) The role of metamodeling in MDA. In: *Proceedings of the Workshop in Software Model Engineering (in conjunction with UML'02, Dresden, Germany).* Online available at http://www.meta model.com/wisme-2002/papers/atkinson.pdf. Accessed January 2008.

Biron PV, Permanente K, Malhotra A (2004) *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation. Online available at http://www. w3.org/TR/xmlschema-2/. Accessed January 2007.

Black PE, ed. (2004) Data structure. In: *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. Online available at http://www.itl.nist.gov/div897/sqg/dads/. Accessed January 2009.

Blitz D (1992) Emergent Evolution, Qualitative Novelty and the Kinds of Reality. Springer.

Borst WN (1997) *Construction of Engineering Ontologies for Knowledge Shar-ing and Reuse*. PhD Thesis, Centre for Telematics and Information Tech-nology, University of Twente.

Casati R, Varzi A (1999) Parts and Places: The Structures of Spatial Representation. MIT Press.

Clark P, Thompson J, Porter B (2000) Knowledge patterns. In: Cohn A, Giunchiglia F, Selman B (eds.): *KR-2000: Proceedings of the Conference on Knowledge Representation and Reasoning*. Morgan Kaufmann:591-600.

Eggersmann M, Hai R, Kausch B, Luczak H, Marquardt W, Schlick C, Schneider N, Schneider R, Theißen M (2008) Work process models. In: Nagl M, Marquardt W (eds.): *Collaborative and Distributed Chemical Engineering Design Processes: From Understanding to Substantial Support*. Springer, Berlin:126–152.

Ferstl OK, Sinz EJ (2001) *Grundlagen der Wirtschaftsinformatik*, Bd. 1. Oldenbourg, München.

Fowler M (1997) UML Distilled- Applying the Standard Object Modeling Language. Addison-Wesley.

Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Grüninger M, Fox MS (1995) Methodology for the design and evaluation of ontologies. In: Skuce D (ed.): *Proceedings of the IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing.*

Haarslev V, Möller R, van der Straeten R, Wessel M (2004) Extended query facilities for Racer and an application to software-engineering problems. In: *Proceedings of the 2004 International Workshop on Description Logics (DL-2004*):148–157.

Little EG, Rogova GL (2005) *Ontology Meta-Model for building a situational picture of catastrophic events*. In: 8th International Conference on Information Fusion 2005, DOI: 10.1109/ICIF. 2005.1591935.

McLaughlin B, Bennett K (2005) *Supervenience*. Online available at http://pla-to.stanford.edu/entries/supervenience/. Accessed December 2006.

Morbach J, Yang A, Marquardt W (2007) OntoCAPE – a large-scale ontology for chemical process engineering. *Eng. Appl. Artif. Intell.* **20** (2):147–161.

Morbach J, Hai R, Bayer B, Marquardt W (2008) Document models. In: Nagl M, Marquardt W (eds.): *Collaborative and Distributed Chemical Engineering*. Springer, Berlin:111–125.

Noy N, Rector A, eds. (2006) *Defining N-ary Relations on the Semantic Web*. W3C Working Group Note, 12 April 2006. Online available at http:// www.w3.org/TR/swbp-n-aryRelations/. Accessed November 2006.

Odell JJ (1994) Six different kinds of composition. *Journal of Object-Oriented Programming* **5**(8). Online available at http://www.conradbock.org/ compkind.html. Accessed December 2006.

OWL (2002) OWL representation ontology. Online available at http://www.w3. org/2002/07/owl. Accessed October 2007.

Patel-Schneider, Horrocks I (2006) *OWL 1.1 Web Ontology Language Overview*. W3C Member Submission, 19 December 2006. Online available at http://www.w3.org/Submission/owl11-overview/. Accessed October 2007.

Racer Systems (2006) *What is Racer Pro?* Webpage, http://www.racer-systems. com/products/racerpro/index.phtml. Accessed December 2006.

Rector A (2003) Modularisation of domain ontologies implemented in description logics and related formalisms including OWL. In: Genari J (ed.): *Knowledge Capture 2003*. ACM Press:121-128.

Rector A, ed. (2005) *Representing Specified Values in OWL: "value partitions" and "value sets"*. W3C Working Group Note, 17 May 2005. Online available at http://www.w3.org/TR/swbp-specified-values. Accessed November 2007.

Rector A, Welty C, eds. (2005) *Simple part-whole relations in OWL Ontologies*. W3C Editor's Draft, 11 August 2005. Online available at http:// www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/. Accessed November 2006.

Simons P (1987) Parts: A Study in Ontology. Oxford University Press.

Smith MK, Welty C, McGuinness DL, eds. (2004) OWL Web Ontology Language Guide. W3C Recommendation, 10 February 2004. Online available at http://www.w3.org/TR/owl-guide/. Accessed October 2007.

Stanford Center for Biomedical Informatics Research (2008) The Protégé Ontol-ogy Editor and Knowledge Acquisition System. Online available at http://protege.stanford.edu/. Accessed January 2008.

Theißen M, Marquardt W (2008) Decision models. In: Nagl M, Marquardt W (eds.): *Collaborative and Distributed Chemical Engineering*. Springer, Berlin:153–168.

Varzi A (2006) Mereology. In: Zalta EN (ed.): *The Stanford Encyclopedia of Philosophy (Winter 2006 Edition)*. Online available at http://plato.stanford.edu/archives/win2006/entries/mereology/. Accessed January 2007.

Appendix

Appendix A Documentation Format

Classes

Classes are characterized by the following categories:

Description: A lexical description of the class, for example “A chemical reactor is an apparatus for holding substances that are undergoing a chemical reaction.” The description explains the meaning of the class to the user.

Definition: Unlike a description, a definition can be transcribed into a formal ontology language, where it establishes the set of necessary and sufficient conditions from which the membership of an ontological concept (class or individual) to the class can be inferred. Classes for which such a definition can not be indicated are called primitive classes.

Relations: The following characteristics are indicated, if existent:

- *Specialization*. A list of parent classes from which the current class is derived via specialization.
- *Disjointness*. A list of classes which are disjoint with the present class. Disjointness between classes means that an instance of the first class cannot simultaneously be an instance of the second class.
- *Restrictions*. Restrictions of binary relations (or attributes) specify the existence of a relation (or attribute) as well as its cardinality and value range with respect to the current class.

Usage: Some recommendations for the use of the class may be given if such advice is required.

Relations

Binary relations are characterized by the following categories:

Description: Similar to that of classes mentioned above.

Characteristics: The following characteristics are listed, if existent:

- *Specialization*. A listing of the relations from which the relation is derived via specialization.
- *Domain*. The domain of the relation.
- *Range*. The value range of the relation.
- *Inverse*. The inverse of a relation.
- Further characteristics, such as if the relation is *transitive*, *symmetric*, or (*inverse*) *functional*.

Usage: As above.

Attributes

Attributes are characterized by the following categories:

Description: As above.

Characteristics: The following characteristics are listed, if existent:

- *Specialization*. A listing of the attributes from which the attribute is derived via specialization.
- *Domain*. The domain of the attribute.

- *Range* or *datatype*. The value range of the attribute, which is usually indicated by referring to a built-in XML Schema Datatype (Biron et al., 2004).
- Further characteristics, such as if the attribute is *functional*.

Usage: As above.

Individuals

Predefined individuals are characterized by the following categories:

Description: As above.

Characteristics: The following characteristics are indicated, if existent:

- *Instance of*. The classes from which the individual is instantiated.
- *Different from*. A list of individual which are explicitly declared to be different from the present individual.
- *Relations*. Instances of binary relations the individual is involved in.
- *Attributes*. Attribute values of the individual.

Usage: As above.

Index of Concepts