ConML Technical Specification

version 1.5.2 · 1 June 2020

This document and its contents were created by Cesar Gonzalez-Perez.

Partial funding was provided by Incipit · CSIC and projects MIRFOL (grant number 09SEC002606PR, INCITE Programme, Xunta de Galicia, Spain), ARIADNE (grant number 313193, FP7-INFRASTRUCTURES-2012-1) and MARIOL (grant number HAR2013-41653-R, Retos de la Sociedad, Plan Estatal 2013-2016, Spain).

1 June 2020 19:17 revision 1168

Table of Contents

1 Introduction

This document contains the technical specification of ConML version 1.5.1. The abstract syntax of ConML is specified as a UML [2, 3] class metamodel, and the ConML graphical notation is described through natural language and diagrams.

For additional information on ConML, please visit www.conml.org. Thank you.

2 Motivation and Requirements

Conceptual modelling is often described as a complex discipline that only engineers with proper training can understand and practice. Whereas this is true to some extent, it is also true that people in a wide range of disciplines often construct models that represent their domain of discourse, be it molecular biology, computer programming or cultural heritage. Over the years, we have observed that people with very little exposure to information technologies are perfectly capable of constructing valid and useful conceptual models if they are given a simple and affordable language in which they can express their ideas. This allows them not only to understand and study relatively complex models, but also to create and maintain them from the very beginning, perhaps assisted by information modelling experts, thus becoming owners of their conceptualizations. Therefore, it is crucial that the modelling language that they use be easily usable by non-experts in information technologies. This, in turn, imposes certain implications regarding simplicity and independence of implementation details. Existing modelling languages, such as UML [2], are too complex, computer-oriented, and reliant on specific implementations as to be useful in this scenario.

Especially for the disciplines in the humanities and social sciences, aspects such as vagueness, subjectivity and temporality, which are rarely considered by conventional modelling languages, must be incorporated in an integrated fashion in the models that are developed. This means that the modelling language being used must be capable of expressing information related to these aspects without putting a big burden on the users.

Finally, the modelling language must be as understandable and familiar to external parties as possible, aiding to the transfer of knowledge and expertise between domains. This makes the object-oriented paradigm a very good choice.

In summary, the requirements of such a modelling language are as follows:

- It must be capable of representing *structural models*, using the *object-oriented* paradigm.
- It must be oriented towards *conceptual modelling* rather than software implementations.
- It must extend the conventional object-oriented paradigm to accommodate often-neglected “soft” issues such as *vagueness*, *temporality* and *subjectivity*.
- It must be easily *affordable to non-experts* in information technologies. This means that it must exhibit high syntactic, semantic and notational *simplicity*.
- It must be *incrementally understandable and applicable*, so that a basic subset of it can be comprehended and used before more advanced areas are tackled.
- It must be precise and complete enough so that it can *work as a formal basis for information systems development*.

- The associated notation must be easily *used by hand* (on paper or whiteboard, for instance), as well as on a computer (on screen or hard copy).
- As long as it is viable, and without jeopardising the above mentioned requirements, it must keep syntactic, semantic and notational *compatibility* with UML [2].

A comprehensive description of the ConML design goals can be found in [1].

These requirements have become the characteristics of the ConML conceptual modelling language, which is described in depth in the remaining sections of this document.

3 Architecture

ConML is composed of the following packages:

- **Types**. This package contains classes such as TypeModel, Class, Attribute and Association, which allow for the creation of *type models*, which represent the world in terms of categories of things.
- **Instances**. This package contains classes such as InstanceModel, Object and Link, which allow for the creation of *instance models*, which represent the world in terms of actual entities.

Figure 1 shows a graphical overview of the different model element types in ConML.

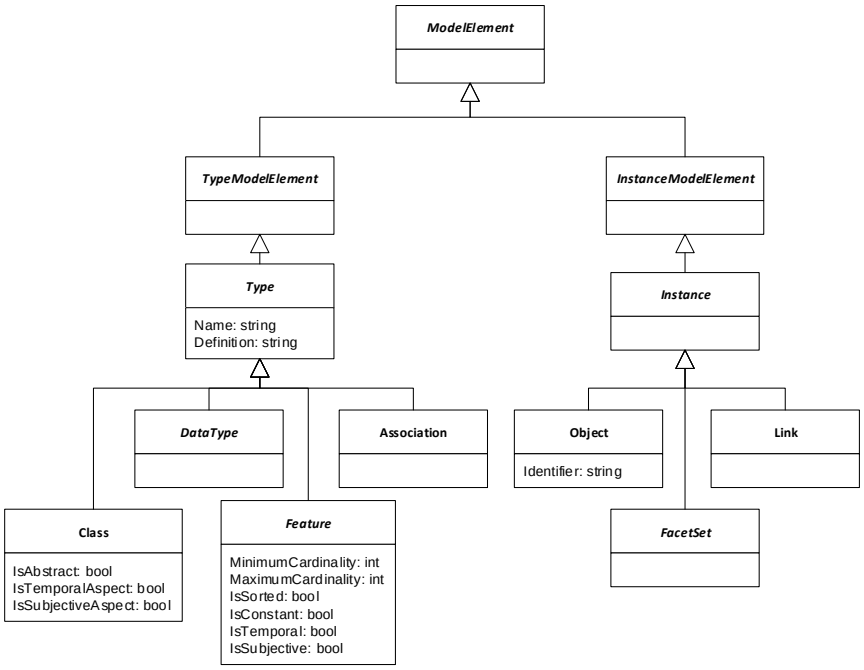

Figure 1. Overview of the major model element types in ConML.

4 Metamodel

This section contains a detailed specification of each class, attribute and association in the ConML metamodel.

4.1 General Classes

4.1.1 Model

A model is *a collection of elements that represents a portion of the world*.

This is an abstract class, which is specialized into TypeModel and InstanceModel.

Figure 2 shows the Model class and its subclasses.

4.1.1.1 Attributes

Name	Type	Description
Name	multilingual string	The name of the model. For example, “CHARM09”.
Version	object	The version of the model. This can be displayed as a string (e.g. “1.0.15.206”) and has comparable semantics that take into account each numeric element in the version string.
Description	multilingual string	The description of the model, in natural language.

4.1.1.2 Associations

Name/Role	Opposite class	Description
HasTags	Tag	A model may have a number of tags.
HasLanguages	Language	A model has a number of languages.
DefaultLanguage	Language	A model has a default language.
OwnsElements	ModelElement	A model may own a number of model elements.

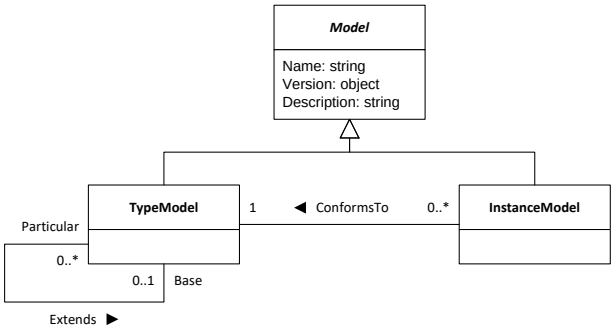

Figure 2. Model-related classes in the ConML metamodel.

4.1.2 ModelPart

A model part is *an entity that belongs to a model*.

This is an abstract class, which is specialized into Language, Tag and ModelElement.

Figure 3 shows the ModelPart class in context.

4.1.2.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.1.2.2 Associations

Name/Role	Opposite class	Description
(this class has no associations)		

4.1.3 Language

A language is *a model part corresponding to the specification of a human language that may be employed by model elements*.

This class specializes from ModelPart.

Languages can be useful to describe model elements, both at the type and instance levels, in multiple languages. See *Multilingualism*, p. 40, for more information.

Figure 3 shows the Language class in context.

4.1.3.1 Attributes

Name	Type	Description
Name	string	The unique name of the language, following the IETF language tag recommendation (https://en.wikipedia.org/wiki/IETF_language_tag), but using underscores instead of hyphens, such as "en_GB".
Description	multilingual string	The description, or display name, of the language, such as "English".
IsDefault	bool	Indicates whether the language is the default one in the model.

4.1.3.2 Associations

Name/Role	Opposite class	Description
BelongsTo	Model	A language always belongs to a given model.
IsTranslation-QualifierOf	FacetSet	A language may be the translation qualifier of a number of facet sets (see *Multilingualism*, p. 40).

4.1.4 Tag

A tag is *a model part corresponding to a label that may be applied to model elements*.

This class specializes from ModelPart.

Tags can be useful to assign custom text strings to model elements.

Figure 3 shows the Tag class in context.

4.1.4.1 Attributes

Name	Type	Description
Name	multilingual string	The name of the tag.

4.1.4.2 Associations

Name/Role	Opposite class	Description
BelongsTo	Model	A tag always belongs to a given model.
AppliesTo	ModelElement	A tag may apply to a number of model elements.

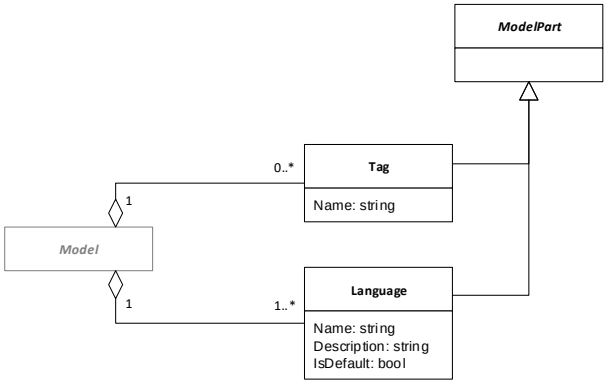

Figure 3. Tags and languages in the ConML metamodel.

4.1.5 ModelElement

A model element is *a model part that represents an entity in the world that is relevant to the model*.

This is an abstract class, which specializes from ModelPart and is further specialized into TypeModelElement and InstanceModelElement.

Figure 1 shows an overview of the main model element types in ConML.

4.1.5.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.1.5.2 Associations

Name/Role	Opposite class	Description
BelongsTo/ OwnerModel	Model	A model element always belongs to a given owner model.
IsTaggedWith	Tag	A model element may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	A model element may be documented by a number of metainformation objects (see *Metainformation*, p. 43).

4.2 Types Package

The following sections describe the classes and associated elements in this package.

4.2.1 TypeModel

A type model is *a model that contains types*.

For example, a type model could contain types that describe the realm of interest for architects as composed of cities, buildings, streets and people, plus the relationships between these.

This class specializes from Model.

Figure 2 shows the TypeModel class in context.

4.2.1.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Model] The name of the type model.
Version	object	[Inherited from Model] The version of the type model. This can be displayed as a string (e.g. “1.0.15.206”) and has comparable semantics that take into account each numeric element in the version string.
Description	multilingual string	[Inherited from Model] The description of the type model, in natural language.

4.2.1.2 Associations

Name/Role	Opposite class	Description
HasTags	Tag	[Inherited from Model] A type model may have a number of tags.
HasLanguages	Language	[Inherited from Model] A type model has a number of languages.
HasDefaultLanguage	Language	[Inherited from Model] A type model has a default language.
OwnsElements	TypeModelElement	[Redefined from Model] A type model may own a number of type model elements.
HasElements	TypeModelElement	A type model may have a number of type model elements.

Name/Role	Opposite class	Description
n/a	InstanceModel	There may be a number of instance models that conform to a type model.
/TemporalAspect	Class	A type model may have a temporal aspect class (see *Temporality*, p. 37).
/SubjectiveAspect	Class	A type model may have a subjective aspect class (see *Subjectivity*, p. 38).
Extends/Base	TypeModel	A type model may extend a base type model.
/Particular	TypeModel	A type model may have a number of particular type models.

4.2.2 TypeModelElement

A type model element is *an element in a type model*.

This is an abstract class, which specializes from ModelElement and is further specialized into Type, Generalization, EnumeratedItem and Package.

Figure 1 shows an overview of the main model element types in ConML.

4.2.2.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.2.2.2 Associations

Name/Role	Opposite class	Description
BelongsTo/ OwnerModel	TypeModel	[Redefined from ModelElement] A type model element always belongs to an owner type model.
IsTaggedWith	Tag	[Inherited from ModelElement] A type model element may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A type model element may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
IsAssignedTo	TypeModel	A type model element is assigned to a number of type models.

4.2.3 Type

A type is *an element in a type model that can be instantiated*.

For example, a type can describe a category of things such as *Person* or *House* (i.e. classes), a characteristic of a category of things such as *Age* (i.e. a feature), a relationship between categories of things such as *Owns* (i.e. an association), or a data type for which values may occur such as *Number* (i.e. a data type).

This is an abstract class, which specializes from TypeModelElement and is further specialized into Class, Feature, DataType and Association.

Figure 1 shows an overview of ConML including the Type class in context.

4.2.3.1 Attributes

Name	Type	Description
Name	multilingual string	The name of the type. For example, “Person”.
Definition	multilingual string	The definition of the type, in natural language.

4.2.3.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A type may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A type may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A type always belongs to an owner type model.

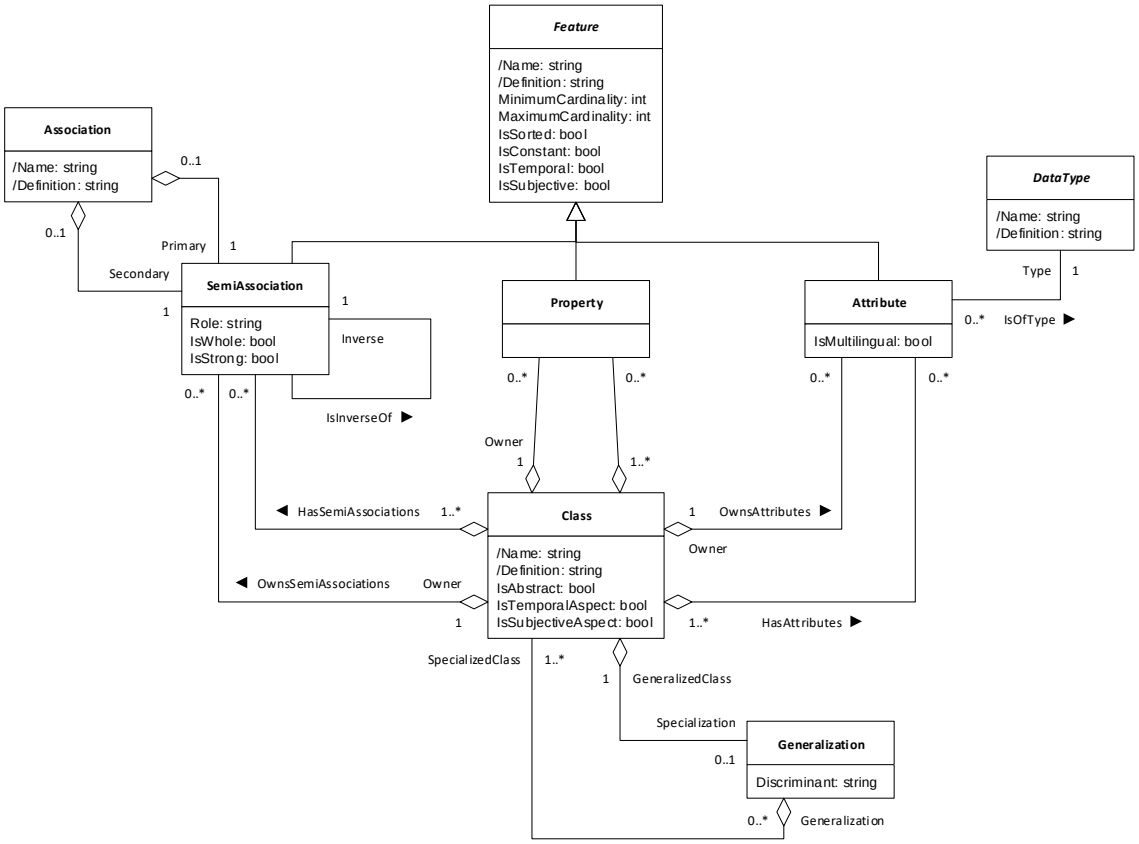

Figure 4. Overview of the Types package. Please see further figures for additional information.

4.2.4 Class

A class is *the formalization of a category that is relevant to the model*.

For example, a model about archaeology might include classes such as *Site*, *Structure* or *Excavation*.

This class specializes from Type.

Figure 4 and Figure 5 show portions of the ConML metamodel including the Class class.

4.2.4.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the class. For example, “Site”.
Definition	multilingual string	[Inherited from Type] The definition of the class, in natural language.
IsAbstract	bool	Indicates whether the class is abstract, i.e. whether it cannot possess direct instances.

Name	Type	Description
IsTemporalAspect	bool	Indicates whether the class constitutes the temporal aspect in the model (see *Temporality*, p. 37).
IsSubjectiveAspect	bool	Indicates whether the class constitutes the subjective aspect in the model (see *Subjectivity*, p. 38).

4.2.4.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A class may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A class may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A class always belongs to an owner type model.
/Specialization	Generalization	A class may be specialized; i.e. it may participate in a generalization that works as its specialization (see *Multiple Generalization of Classes*, p. 34).
/Generalization	Generalization	A class may be generalized; i.e. it may participate in a number of generalizations in which it is the specialized class (see *Multiple Generalization of Classes*, p. 34).
/Dominant-Generalization	Generalization	If a class is generalized, then one of the generalizations that it has is the dominant generalization (see *Multiple Generalization of Classes*, p. 34).
OwnsProperties	Property	A class may own multiple properties.
HasProperties	Property	A class may have multiple properties. These include the owned plus the inherited properties (see *Feature Inheritance*, p. 34).
OwnsAttributes	Attribute	A class may own multiple attributes.
HasAttributes	Attribute	A class may have multiple attributes. These include the owned plus the inherited attributes (see *Feature Inheritance*, p. 34).
OwnsSemi-Associations	SemiAssociation	A class may own multiple semi-associations.
HasSemiAssociations	SemiAssociation	A class may have multiple semi-associations. These include the owned plus the inherited semi-associations (see *Feature Inheritance*, p. 34).
n/a	SemiAssociation	A class may be the opposite class of a number of semi-associations.
IsTemporalAspectOf	TypeModel	A class may be the temporal aspect of a number of type models (see *Temporality*, p. 37).
IsSubjectiveAspectOf	TypeModel	A class may be the subjective aspect of a number of type models (see *Subjectivity*, p. 38).
/Instance	Object	A class may have a number of instance objects.
n/a	Package	A class may belong to a package.

Figure 5. Fragment of the ConML metamodel showing the Class and Generalization classes.

4.2.5 Generalization

A generalization is *the formalization of a subsumption relationship between two categories that is relevant to the model*. The "generalization" notion corresponds to the viewpoint of the subsumed category; the opposite concept is that of "specialization", which corresponds to the viewpoint of the subsuming category.

ConML implements multiple generalization, but not multiple specialization. This means that a class may be generalized through zero, one or multiple generalizations; but a class may be only specialized through zero or one (never multiple) generalizations. See *Multiple Generalization of Classes*, p. 34 for details. Generalization relationships determine what features are inherited by each class; see *Feature Inheritance*, p. 34.

For example, the *Structure* and *Artefact* classes in a model about archaeology could be related to the *MaterialElement* class via a generalization in which *MaterialElement* is the generalized class and both *Structure* and *Artefact* are specialized classes.

This class specializes from TypeModelElement.

Figure 5 shows a portion of the ConML metamodel including the Generalization class.

4.2.5.1 Attributes

Name	Type	Description
Discriminant	multilingual string	The characteristic of the generalized class that acts as differentiating factor among the specialized classes.

4.2.5.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A generalization may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A generalization may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A generalization always belongs to an owner type model.
/GeneralizedClass	Class	A generalization is always rooted on a given generalized class, of which it is the specialization.
/SpecializedClass	Class	A generalization always involves one or more specialized classes, of which it is a generalization.
Dominates-InheritanceOn	Class	A generalization always dominates the inheritance on one or more classes (see *Multiple Generalization of Classes*, p. 34).

4.2.6 Feature

A feature is *the formalization of a characteristic of a category that is relevant to the model*.

This is an abstract class, which specializes from Type and is further specialized into Property, Attribute and SemiAssociation.

Figure 4 shows a portion of the ConML metamodel including the Feature class.

4.2.6.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the feature. For example, “Age”.
Definition	multilingual string	[Inherited from Type] The definition of the feature, in natural language.
MinimumCardinality	int	The minimum number of discrete entities that may be related to the associated characteristic.
MaximumCardinality	int	The maximum number of discrete entities that may be related to the associated characteristic.
IsSorted	bool	Whether multiple instances of the feature are sorted in a specific order.
IsConstant	bool	Whether the feature has constant semantics, i.e. no changes may be made to instances of the feature. See *Temporality*, p. 37.
IsTemporal	bool	Whether the feature has temporal semantics, i.e. changes to the instances of the feature may entail different phase parts of the associated object. See *Temporality*, p. 37.
IsSubjective	bool	Whether the feature has subjective semantics, i.e. changes to the instances of the feature may entail different perspective parts of the associated object. See *Subjectivity*, p. 38.

4.2.6.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A feature may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A feature may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A feature always belongs to an owner type model.
Redefines/ RedefinedOriginal	Feature	A feature may redefine a given original feature.
IsRedefinedBy/ Redefinition	Feature	A feature may be redefined by a number of redefinition features.

4.2.7 Property

A property is *the abstract formalization of a characteristic of a category that is relevant to the model*. A property is an abstraction of the underlying characteristic and does not express whether it will be implemented as an attribute or a semi-association.

For example, the *Person* class in a model could initially contain an *Address* property; this captures the need to take into account the fact that people have addresses, but does not specify whether the property is to be implemented as an attribute or a semi-association of the *Person* class to some other class. Further versions of the model may refine the *Address* property into an attribute or a semi-association.

A class that contains one or more properties cannot be instantiated (i.e. there cannot be objects of that class), since no implementation is provided for the property.

This class specializes from Feature.

Figure 4 shows a portion of the ConML metamodel including the Property class.

4.2.7.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the property. For example, “Address”.
Definition	multilingual string	[Inherited from Type] The definition of the property, in natural language.
MinimumCardinality	int	[Inherited from Feature] The minimum number of pieces of data or objects that may be related to the associated characteristic.
MaximumCardinality	int	[Inherited from Feature] The maximum number of pieces of data or objects that may be related to the associated characteristic.
IsSorted	bool	[Inherited from Feature] Whether multiple instances of the implemented property will be sorted in a specific order.
IsConstant	bool	[Inherited from Feature] Whether the property has constant semantics, i.e. no changes may be made to instances of the implemented property. See *Temporality*, p. 37.
IsTemporal	bool	[Inherited from Feature] Whether the property has temporal semantics, i.e. changes to the instances of the implemented property may entail different phase parts of the associated object. See *Temporality*, p. 37.
IsSubjective	bool	[Inherited from Feature] Whether the property has subjective semantics, i.e. changes to the instances of the implemented property may entail different perspective parts of the associated object. See *Subjectivity*, p. 38.

4.2.7.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A property may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A property may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A property always belongs to an owner type model.
Redefines/ RedefinedOriginal	Property	[Redefined from Feature] A property may redefine a given original property. See *Feature Redefinition*, p. 35.
IsRedefinedBy/ Redefinition	Property	[Redefined from Feature] A property may be the original that is redefined by a number of redefinition properties. See *Feature Redefinition*, p. 35.
/Owner	Class	A property is always owned by a given owner class.
n/a	Class	A property is always assigned to one or more classes.

4.2.8 Attribute

An attribute is *the formalization of an atomic characteristic of a category that is relevant to the model*. An attribute always corresponds to an atomic characteristic, i.e. it cannot be decomposed into simpler parts as far as the model is concerned.

For example, the *Site* class in a model about archaeology may contain attributes such as *Name* or *Dimensions*.

This class specializes from Feature.

Figure 4 and Figure 6 show portions of the ConML metamodel including the Attribute class.

4.2.8.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the attribute. For example, “Dimensions”.
Definition	multilingual string	[Inherited from Type] The definition of the attribute, in natural language.
MinimumCardinality	int	[Inherited from Feature] The minimum number of values that may be associated to this attribute.
MaximumCardinality	int	[Inherited from Feature] The maximum number of values that may be associated to this attribute.
IsSorted	bool	[Inherited from Feature] Whether multiple instances of the attribute (i.e. values) are sorted in a specific order.
IsConstant	bool	[Inherited from Feature] Whether the attribute has constant semantics, i.e. no changes may be made to the values of this attribute. See *Temporality*, p. 37.
IsTemporal	bool	[Inherited from Feature] Whether the attribute has temporal semantics, i.e. changes to the values of this attribute may entail different phase parts of the associated object. See *Temporality*, p. 37.
IsSubjective	bool	[Inherited from Feature] Whether the attribute has subjective semantics, i.e. changes to the values of this attribute may entail different perspective parts of the associated object. See *Subjectivity*, p. 38.
IsMultilingual	bool	Whether the attribute has multilingual semantics, i.e. changes to the values of this attribute may entail different translation parts of the associated object. See *Multilingualism*, p. 40.

4.2.8.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An attribute may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An attribute may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] An attribute always belongs to an owner type model.
Redefines/ RedefinedOriginal	Property	[Redefined from Feature] An attribute may redefine a given original attribute. See *Feature Redefinition*, p. 35.
IsRedefinedBy/ Redefinition	Property	[Redefined from Feature] An attribute may be the original that is redefined by a number of redefinition attributes. See *Feature Redefinition*, p. 35.
/Owner	Class	An attribute is always owned by a given owner class.
n/a	Class	An attribute is always assigned to one or more classes.
IsOfType/Type	DataType	An attribute is always of a given data type.
/Instance	ValueSet	An attribute may have a number of instance value sets.

4.2.9 DataType

A data type is *a specification of what kind of data may be used to represent atomic values*. Atomic values are those that cannot be decomposed in simpler parts as far as the model is concerned.

This is an abstract class, which specializes from Type and is further specialized into SimpleDataType and EnumeratedType.

Figure 6 shows a portion of the ConML metamodel including the DataType class.

4.2.9.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the data type. For example, “Number”.
Definition	multilingual string	[Inherited from Type] The definition of the data type, in natural language.

4.2.9.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A data type may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A data type may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A data type always belongs to an owner type model.
n/a	Attribute	A data type may be the type of a number of attributes.

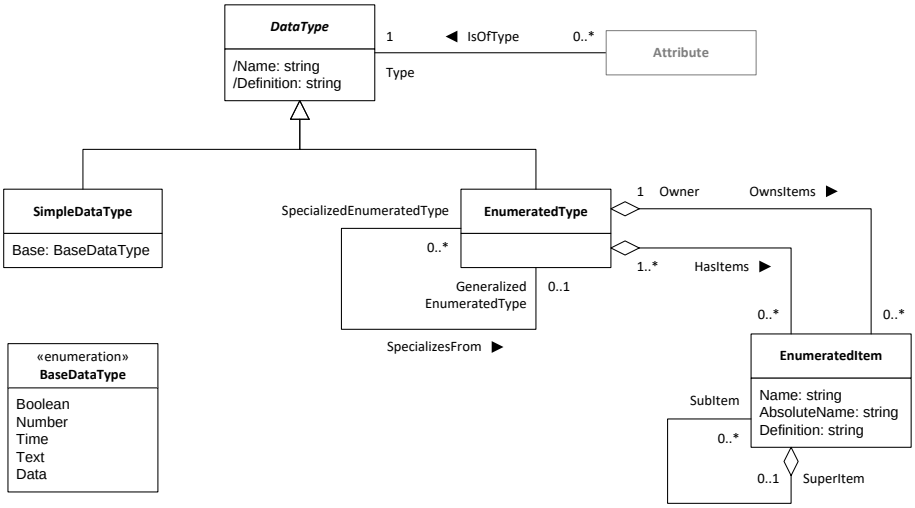

Figure 6. Fragment of the ConML metamodel showing the DataType class, its subclasses and other related metamodel elements.

4.2.10 SimpleDataType

A simple data type is *a data type that represents values of one of the pre-defined base data types*.

All simple data types, such as Number or Text, are defined by the ConML specification. No additional simple data types may be added.

This class specializes from DataType.

Figure 6 shows a portion of the ConML metamodel including the SimpleDataType class.

4.2.10.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the simple data type; this always corresponds to the text value of the DataType attribute. For example, “Number”.
Definition	multilingual string	[Inherited from Type] The definition of the simple data type, in natural language.
Base	BaseDataType	The base data type this simple data type refers to.

4.2.10.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A simple data type may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A simple data type may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A simple data type always belongs to an owner type model.
n/a	Attribute	[Inherited from DataType] A simple data type may be the type of a number of attributes.

4.2.11 BaseDataType

This enumeration corresponds to the pre-defined base data types that exist in any model expressed in ConML.

Figure 6 shows a portion of the ConML metamodel including the BaseDataType enumeration.

4.2.11.1 Elements

Name	Description
Boolean	Values may only be *true* or *false*.
Number	Values are real numbers; integer or not; positive, zero or negative.
Time	Values are time points of variable precision, and not limited to the usual scheme of days, months, years, hours, minutes and seconds.
Text	Values are character strings of arbitrary length, including zero length (i.e. empty strings). They can be multilingual.
Data	Values are raw, uninterpreted data (i.e. byte lists) of arbitrary length, including zero length.

A complete description of the semantics of the pre-defined base data types is given in *Semantics of Data Types*, p. 32.

4.2.12 EnumeratedType

An enumerated type is *a data type that defines a list of named items that can be associated to a value of this type*. Enumerated types, in opposition to simple data types, are defined by the ConML user as part of the model, rather than being pre-defined by ConML itself. A complete description of the semantics of enumerated types is given in *Semantics of Data Types*, p. 32.

For example, a model that includes an *Artefact* class with a *Material* attribute might also include an *ArtefactMaterial* enumerated type that lists possible values for that attribute, such as *Clay*, *Metal*, *Wood*, etc.

This class specializes from DataType.

Figure 6 shows a portion of the ConML metamodel including the EnumeratedType class.

4.2.12.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the enumerated type. For example, “ArtefactMaterial”.
Definition	multilingual string	[Inherited from Type] The definition of the enumerated type, in natural language.

4.2.12.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An enumerated type may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An enumerated type may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] An enumerated type always belongs to an owner type model.
n/a	Attribute	[Inherited from DataType] An enumerated type may be the type a number of attributes.
OwnsItems	EnumeratedItem	An enumerated type may own multiple enumerated items.
HasItems	EnumeratedItem	An enumerated type may have multiple enumerated items. These include the owned plus the inherited enumerated items (see *Generalization of Enumerated Types*, p. 33).
SpecializesFrom /Generalized-EnumeratedType	EnumeratedType	An enumerated type may specialize from a generalized enumerated type.
/Specialized-EnumeratedType	EnumeratedType	An enumerated type may be generalized from a number of specialized enumerated types.
n/a	Package	An enumerated type may belong to a package.

4.2.13 EnumeratedItem

An enumerated item is *a unique name within a given enumerated type*. Enumerated items can be hierarchically arranged within an enumerated type, so that subsumption or whole/part relationships are conveyed (see *Enumerated Item Hierarchies*, p. 33).

For example, the *ArtefactMaterial* enumerated type in the previous example may include enumerated items *Clay*, *Metal*, *Metal/Iron*, *Metal/Brass*, *Wood*, *Wood/Birch*, *Wood/Birch/SilverBirch*, *Wood/Birch/WhiteBirch*, *Wood/Oak*, etc. The *Wood/Birch* and *Wood/Oak* enumerated items are sub-items of the *Wood* item; similarly, *Wood/Birch/SilverBirch* and *Wood/Birch/WhiteBirch* are sub-items of *Wood/Birch*.

This class specializes from TypeModelElement.

Figure 6 shows a portion of the ConML metamodel including the EnumeratedItem class.

4.2.13.1 Attributes

Name	Type	Description
Name	multilingual string	The name of the enumerated item. This is a local name and does not take into account the position of the item within the hierarchy. For example, “Birch”.
AbsoluteName	multilingual string	The absolute name of the enumerated item, taking into account the position that it occupies in the hierarchy. For example, “Wood/Birch”.

Name	Type	Description
Definition	multilingual string	The definition of the enumerated item, in natural language.

4.2.13.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An enumerated item may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An enumerated item may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] An enumerated item always belongs to an owner type model.
/Owner	EnumeratedType	An enumerated item always has a given owner enumerated type.
n/a	EnumeratedType	An enumerated item is always assigned to one or more enumerated types.
/SuperItem	EnumeratedItem	An enumerated item may have a super-item, of which it is a sub-item.
/SubItem	EnumeratedItem	An enumerated item may have multiple sub-items, of which it is the super-item.

4.2.14 SemiAssociation

A semi-association is *the description of an association from the viewpoint of one of the classes that participate in it*. A semi-association always has an inverse semi-association, which describes the same association seen from the opposite end, i.e. from the viewpoint of the class at the other end. In the context of any given semi-association, the class that gives it its viewpoint, i.e. the class that owns the semi-association, is called the *participant class*. The class the semi-association refers to, i.e. the class at the opposite end (which is usually the participant class of the inverse semi-association) is called the *opposite class*. A particular case occurs for symmetric self-associations, for which the participant and opposite classes are the same, and a semi-association's inverse is itself (see *Symmetric Self-Associations*, p. 37).

For example, the *Site* class in a model about archaeology might be associated to the *Structure* class via the *Contains* semi-association (a site *contains* structures). *Site* would be the participant class of *Contains*, and *Structure* would be the opposite class. Looking at this from the opposite viewpoint, a different semi-association would exist, inverse to the former, and probably named *IsLocatedOn* (a structure *is located on* a site). *Contains* and *IsLocatedOn* are inverse semi-associations that define the same association.

This class specializes from Feature.

Figure 4 and Figure 7 show portions of the ConML metamodel including the SemiAssociation class.

4.2.14.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the semi-association. For example, "Contains".
Definition	multilingual string	[Inherited from Type] The definition of the s, in natural language.
MinimumCardinality	int	[Inherited from Feature] The minimum number of objects of the opposite class that may be linked to an object of the participant class through this semi-association.

Name	Type	Description
MaximumCardinality	int	[Inherited from Feature] The maximum number of objects of the opposite class that may be linked to an object of the participant class through this semi-association.
IsSorted	bool	[Inherited from Feature] Whether objects reachable through multiple links of this semi-association are sorted in a specific order.
IsConstant	bool	[Inherited from Feature] Whether the semi-association has constant semantics, i.e. no changes may be made to the references of this semi-association. See *Temporality*, p. 37.
IsTemporal	bool	[Inherited from Feature] Whether the semi-association has temporal semantics, i.e. changes to the references of this semi-association may entail different phase parts of the associated object. See *Temporality*, p. 37.
IsSubjective	bool	[Inherited from Feature] Whether the semi-association has subjective semantics, i.e. changes to the references of this semi-association may entail different perspective parts of the associated object. See *Subjectivity*, p. 38.
Role	multilingual string	The role that the opposite class plays in the association from this viewpoint, expressed in singular regardless of the associated cardinality. For example, “Owner”.
IsWhole	bool	Indicates whether the participant class possesses “whole” semantics in a “whole/part” relationship. This means that the participant class is defined as an aggregate of the opposite class, which has “part” semantics.
IsStrong	bool	Indicates whether the participant class is strongly connected to the opposite class through this semi-association. This means that the semi-association plays a significant role in the definition of the participant class.

4.2.14.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A semi-association may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A semi-association may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A semi-association always belongs to an owner type model.
Redefines/ RedefinedOriginal	Property	[Redefined from Feature] A semi-association may redefine a given original semi-association. See *Feature Redefinition*, p. 35.
IsRedefinedBy/ Redefinition	Property	[Redefined from Feature] A semi-association may be the original that is redefined by a number of redefinition semi-associations. See *Feature Redefinition*, p. 35.
IsInverseOf/Inverse	SemiAssociation	A semi-association always has an inverse semi-association.

Name/Role	Opposite class	Description
/Owner	Class	A semi-association is always owned by a given owner class.
n/a	Class	A semi-association is always assigned to one or more classes.
RefersTo /OppositeClass	Class	A semi-association always refers to a given opposite class.
/Instance	ReferenceSet	A semi-association may have a number of instance reference sets.
IsPrimaryIn	Association	A semi-association may be primary in a given association.
IsSecondaryIn	Association	A semi-association may be secondary in a given association.

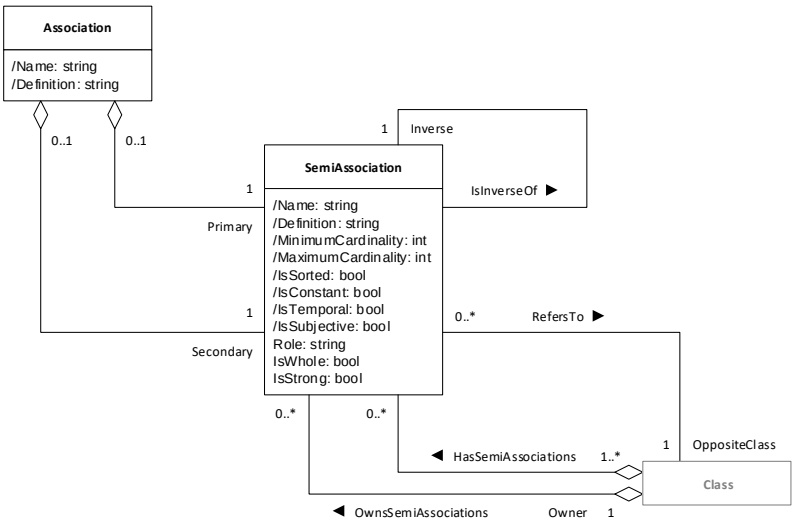

Figure 7. Fragment of the ConML metamodel showing the Association and SemiAssociation classes.

4.2.15 Association

An association is *the formalization of a structural connection relationship between two categories that is relevant to the model*. An association is always defined through semi-associations that are inverse of each other, i.e. describe the association as seen from opposite and complementary viewpoints. One semi-association is called *primary* and is used to name and describe the association as a whole, whereas the inverse semi-association is called *secondary*. Usually, two different semi-associations are involved; a particular case occurs in the case of symmetric self-associations, for which the primary and secondary semi-associations are the same (see *Symmetric Self-Associations*, p. 37).

For example, the *Site* class in a model about archaeology might be associated to the *Structure* class via the *Contains* semi-association (a site *contains* structures). From the opposite viewpoint, a different semi-association would exist, inverse to the former, and probably named *IsLocatedIn* (a structure *is located in* a site). *Contains* is the primary semi-association and *IsLocatedIn* is the secondary one; both are inverse semi-associations that define the same association.

This class specializes from Type.

Figure 7 shows a portion of the ConML metamodel including the Association class.

4.2.15.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Type] The name of the association, which coincides with that of its primary semi-association. For example, “Contains”.
Definition	multilingual string	[Inherited from Type] The definition of the association, in natural language.

4.2.15.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An association may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An association may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] An association always belongs to an owner type model.
HasPrimary/Primary	SemiAssociation	An association always has a primary semi-association.
HasSecondary /Secondary	SemiAssociation	An association always has a secondary semi-association.
/Instance	Link	An association may have a number of instance links.

4.2.16 Package

A package is *a group of related classes, enumerated types and possibly sub-packages*.

This class specializes from TypeModelElement.

Figure 8 shows the Package class in the context of the Types package.

4.2.16.1 Attributes

Name	Type	Description
Name	multilingual string	The name of the package. For example, “Organization”.
Description	multilingual string	The description of the package, in natural language.

4.2.16.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A package may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A package may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	TypeModel	[Inherited from TypeModelElement] A package always belongs to an owner type model.
n/a	Class	A package may contain a number of classes.
n/a	EnumeratedType	A package may contain a number of enumerated types.
/Owner	Package	A package may be contained in an owner package.
/SubPackage	Package	A package may be the owner of a number of sub-packages.

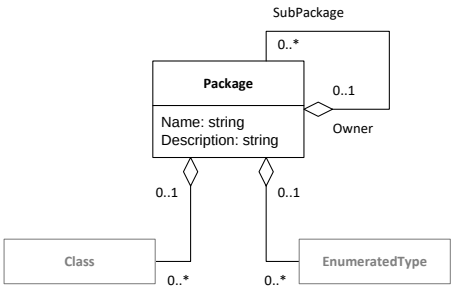

Figure 8. Fragment of the ConML metamodel showing the Package class in the Types package.

4.3 Instances Package

The following sections describe the classes and associated elements in this package.

4.3.1 InstanceModel

An instance model is *a model that contains instances*.

For example, an instance model could contain instances describing that person *p1*, who is 38 years old, owns house *h1* located in Moscow, in which person *p2* also lives.

This class specializes from Model.

Figure 2 shows InstanceModel class in context.

4.3.1.1 Attributes

Name	Type	Description
Name	multilingual string	[Inherited from Model] The name of the instance model.
Version	object	[Inherited from Model] The version of the instance model. This can be displayed as a string (e.g. “1.0.15.206”) and has comparable semantics that take into account each numeric element in the version string.
Description	multilingual string	[Inherited from Model] The description of the instance model, in natural language.

4.3.1.2 Associations

Name/Role	Opposite class	Description
HasTags	Tag	[Inherited from Model] An instance model may have a number of tags.
HasLanguages	Language	[Inherited from Model] An instance model has a number of languages.
HasDefaultLanguage	Language	[Inherited from Model] An instance model has a default language.
OwnsElements	InstanceModel-Element	[Redefined from Model] An instance model may own a number of instance model elements.
ConformsTo	TypeModel	An instance model always conforms to at least one type model.

4.3.2 InstanceModelElement

An instance model element is *an element in an instance model*.

This is an abstract class, which specializes from ModelElement and is further specialized into Instance and Facet.

Figure 1 shows an overview of the main model element types in ConML.

4.3.2.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.2.2 Associations

Name/Role	Opposite class	Description
BelongsTo/ OwnerModel	InstanceModel	[Redefined from ModelElement] An instance model element always belongs to an owner instance model.
IsTaggedWith	Tag	[Inherited from ModelElement] An instance model element may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An instance model element may be documented by a number of metainformation objects (see *Metainformation*, p. 43).

4.3.3 Instance

An instance is *an element in an instance model that is a particular occurrence of a type in the type model to which the instance model conforms*.

For example, an instance can represent a thing such as person *p1* or house *h2* (i.e. objects), a piece of data about a thing such as *Age = 38* (i.e. a value), or a connection between two things such as *p1 Owns h2* (i.e. a link).

This is an abstract class, which specializes from InstanceModelElement and is further specialized into Object, FacetSet and Link.

Figure 1 shows an overview of ConML including the Instance class in context.

4.3.3.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.3.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An instance may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An instance may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] An instance always belongs to an owner instance model.

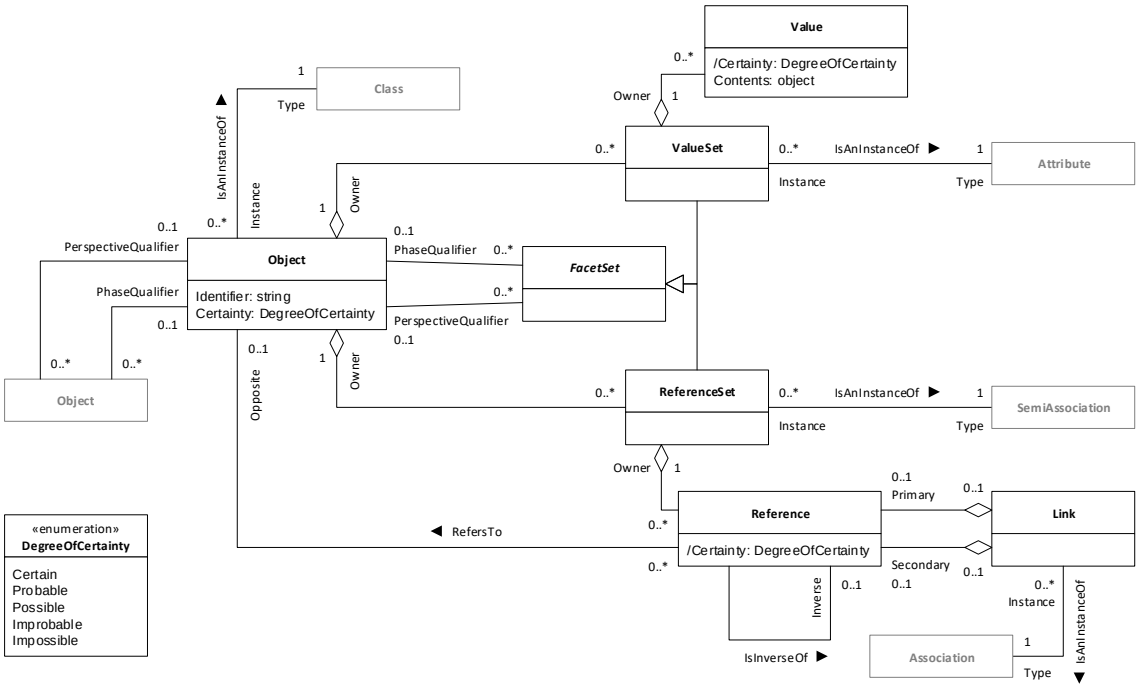

Figure 9. Overview of the Instances package.

4.3.4 Object

An object is *the formalization of an entity that is relevant to the model*. The entity represented by an object may be a real one or it may correspond to a fictitious entity that is used, for example, for illustration or simulation purposes. Any given object is an instance of a given class.

For example, a model about archaeology may show objects that represent specific sites such as *Stonehenge* or *Mount Athos*.

This class specializes from Instance.

Figure 9 shows a portion of the ConML metamodel including the Object class.

4.3.4.1 Attributes

Name	Type	Description
Identifier	string	The identifier or unique name of the object. For example, “site173”. Object identifiers cannot coincide with a ConML keyword (see *Keywords*, p. 57).
Certainty	DegreeOfCertainty	The degree of certainty for the existence conveyed by the object (see *Vagueness*, p. 39).

4.3.4.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] An object may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] An object may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] An object always belongs to an owner instance model.
IsAnInstanceOf/ Type	Class	An object is always an instance of a given type class.
OwnsValueSets	ValueSet	An object possesses multiple value sets, one per attribute assigned to the corresponding class.

Name/Role	Opposite class	Description
OwnsReferenceSets	ReferenceSet	An object possesses multiple reference sets, one per semi-association assigned to the corresponding class.
IsOppositeIn	Reference	An object may be opposite in a number of references.
IsPhaseQualifierOf	FacetSet	An object may be the phase qualifier of a number of facet sets (see *Temporality*, p. 37).
IsPerspective-QualifierOf	FacetSet	An object may be the perspective qualifier of a number of facet sets (see *Subjectivity*, p. 38).
/TemporalExistential-Qualifier	Object	An object may have another object as temporal existential qualifier (see *Temporality*, p. 37).
/Subjective-ExistentialQualifier	Object	An object may have another object as subjective existential qualifier (see *Subjectivity*, p. 38).
IsTemporalExistence-QualifierOf	Object	An object may be the temporal existence qualifier of a number of other objects (see *Temporality*, p. 37).
IsSubjective-ExistenceQualifierOf	Object	An object may be the subjective existence qualifier of a number of other objects (see *Subjectivity*, p. 38).
Documents	ModelElement	An object may document a number of model elements (see *Metainformation*, p. 43).

4.3.5 DegreeOfCertainty

This enumeration corresponds to the different degrees of certainty that an instance model may express about a fact.

Figure 9 shows a portion of the ConML metamodel including the DegreeOfCertainty enumeration.

4.3.5.1 Elements

Name	Description
Certain	The expressed fact is known to be true.
Probable	The expressed fact is probably true.
Possible	The expressed fact is possibly true.
Improbable	The expressed fact is probably not true.
Impossible	The expressed fact is known to be not true.

A complete description of the semantics of the different degrees of certainty is given in *Vagueness*, p. 39.

4.3.6 FacetSet

A facet set is *the formalization of a quality of an entity that is relevant to the model according to a feature in the associated class*. Any given facet set belongs to a particular object and is an instance of a particular feature of the associated class.

This is an abstract class, which specializes from Instance and is further specialized into ValueSet and ReferenceSet.

Figure 9 shows a portion of the ConML metamodel including the FacetSet class.

4.3.6.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.6.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A facet set may be tagged with a number of tags.

Name/Role	Opposite class	Description
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A facet set may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A facet set always belongs to an owner instance model.
/PhaseQualifier	Object	A facet set may have a particular object as phase qualifier (see *Temporality*, p. 37).
/PerspectiveQualifier	Object	A facet set may have a particular object as perspective qualifier (see *Subjectivity*, p. 38).
IsComposedOf	Facet	A facet set is composed of a number of facets.

4.3.7 ValueSet

A value set is *a facet set that corresponds to an instance of an attribute*. Any given value set is a collection of values.

For example, the *Stonehenge* object in a model about archaeology may contain a value set instantiated from the *Chronology* attribute containing the value *Chronology = "2500 BC"*.

This class specializes from FacetSet.

Figure 9 shows a portion of the ConML metamodel including the ValueSet class.

4.3.7.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.7.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A value set may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A value set may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A value set always belongs to an owner instance model.
IsComposedOf	Value	[Redefined from FacetSet] A value set is composed of a number of values.
/PhaseQualifier	Object	[Inherited from FacetSet] A value set may have a particular object as phase qualifier (see *Temporality*, p. 37).
/PerspectiveQualifier	Object	[Inherited from FacetSet] A value set may have a particular object as perspective qualifier (see *Subjectivity*, p. 38).
/TranslationQualifier	Language	A value set may have a particular language as translation qualifier (see *Multilingualism*, p. 40).
IsAnInstanceOf/ Type	Attribute	A value set is always an instance of a given type attribute.
/Owner	Object	A value set is always owned by a given owner object.

4.3.8 ReferenceSet

A reference set is *a facet set that corresponds to an instance of a semi-association*. Any given reference set is a collection of references.

For example, the *Stonehenge* object of class *Site* in a model about archaeology may contain a reference set instantiated from the *IsLocatedIn* semi-association containing a reference pointing to object *Wiltshire, UK* of class *Place*.

This class specializes from FacetSet.

Figure 9 shows a portion of the ConML metamodel including the ReferenceSet class.

4.3.8.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.8.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A reference set may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A reference set may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A reference set always belongs to an owner instance model.
IsComposedOf	Reference	[Redefined from FacetSet] A reference set is composed of a number of references.
/PhaseQualifier	Object	[Inherited from FacetSet] A reference set may have a particular object as phase qualifier (see *Temporality*, p. 37).
/PerspectiveQualifier	Object	[Inherited from FacetSet] A reference set may have a particular object as perspective qualifier (see *Subjectivity*, p. 38).
IsAnInstanceOf/ Type	SemiAssociation	A reference set is always an instance of a given type semi-association.
/Owner	Object	A reference set is always owned by a given owner object.

4.3.9 Facet

A facet is *the formalization of an atomic quality of an entity that is relevant to the model*. Any given facet is part of a particular facet set, which is in turn an instance of a particular feature.

This is an abstract class, which specializes from InstanceModelElement and is further specialized into Value and Reference.

4.3.9.1 Attributes

Name	Type	Description
Certainty	DegreeOfCertainty	The degree of certainty for the predication conveyed by the facet (see *Vagueness*, p. 39).

4.3.9.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A facet may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A facet may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A facet always belongs to an owner instance model.
/Set	FacetSet	A facet is always owned by a given facet set.

4.3.10 Value

A value is *a facet corresponding to a piece of data*. Any given value is part of a given value set, which is in turn an instance of a particular attribute.

For example, the *Stonehenge* object in a model about archaeology may contain a value set instantiated from the *Chronology* attribute containing the value *Chronology = "2500 BC"*.

This class specializes from Facet.

Figure 9 shows a portion of the ConML metamodel including the Value class.

4.3.10.1 Attributes

Name	Type	Description
Certainty	DegreeOfCertainty	[Inherited from Facet] The degree of certainty for the predication conveyed by the value (see *Vagueness*, p. 39).
Contents	object	The actual data contained in the value. For example, "2500 BC". Empty (i.e. null) contents in a value indicate unknown semantics (see *Null and Unknown Semantics*, p. 36).

4.3.10.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A value may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A value may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A value always belongs to an owner instance model.
/Set	ValueSet	[Redefined from Facet] A value is always owned by a given value set.

4.3.11 Reference

A reference is *a facet corresponding to a pointer to an object, which describes a link from the viewpoint of one of the objects that participate in it*. Any given reference is part of a given reference set, which is in turn an instance of a particular semi-association. A reference usually has an inverse reference, which describes the same link seen from the opposite end, i.e. from the viewpoint of the object at the other end. References that link an object to an unknown object lack an inverse. In the context of any given reference, the object that gives it its viewpoint, i.e. the object that owns the reference, is called the *participant object*. The object the reference refers to, i.e. the object at the opposite end (which is always the participant object of the inverse reference) is called the *opposite object*.

For example, the *Stonehenge* object of class *Site* in a model about archaeology may contain a reference set instantiated from the *IsLocatedIn* semi-association containing a reference pointing to object *Wiltshire, UK* of class *Place*. Looking at this from the opposite viewpoint, a different reference would exist, inverse to the former, and possibly named *Contains*, owned by participant object *Wiltshire, UK* and referencing object *Stonehenge*.

This class specializes from Facet.

Figure 9 shows a portion of the ConML metamodel including the Reference class.

4.3.11.1 Attributes

Name	Type	Description
Certainty	DegreeOfCertainty	[Inherited from Facet] The degree of certainty for the predication conveyed by the reference (see *Vagueness*, p. 39).

4.3.11.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A reference may be tagged with a number of tags.
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A reference may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A reference always belongs to an owner instance model.
/Set	ReferenceSet	[Redefined from Facet] A reference is always owned by a given reference set.
IsInverseOf/Inverse	Reference	A reference usually has an inverse reference. Only references that point to an unknown object lack an inverse reference (see *Null and Unknown Semantics*, p. 36).
RefersTo/Opposite	Object	A reference always refers to an opposite object, unless it points to an unknown object (see *Null and Unknown Semantics*, p. 36).
IsPrimaryIn	Link	A reference may be primary in a given link.
IsSecondaryIn	Link	A reference may be secondary in a given link.

4.3.12 Link

A link is *the formalization of a connection between two entities that is relevant to the model*. Any given link is an instance of a given association and links a primary and a secondary object as dictated by the primary and secondary semi-associations of the corresponding association. A link is usually defined through two references that are inverse of each other, i.e. describe the link as seen from opposite and complementary viewpoints. One reference is called *primary* and is used to name and describe the link as a whole, whereas the inverse reference is called *secondary*. An exception occurs in the case of objects linked to an unknown object; in these cases, only one reference (either the primary or the secondary) is associated to the link.

For example, the *Stonehenge* object (a site) in the model about archaeology may be linked to the *Wiltshire, UK* object (a place) in the form of a link; this link would be an instance of the association defined by the semi-association *IsLocatedOn* plus its inverse.

This class specializes from Instance.

Figure 9 shows a portion of the ConML metamodel including the Link class.

4.3.12.1 Attributes

Name	Type	Description
(this class has no attributes)		

4.3.12.2 Associations

Name/Role	Opposite class	Description
IsTaggedWith	Tag	[Inherited from ModelElement] A link may be tagged with a number of tags.

Name/Role	Opposite class	Description
IsDocumentedBy/ Metainformation	Object	[Inherited from ModelElement] A link may be documented by a number of metainformation objects (see *Metainformation*, p. 43).
BelongsTo/ OwnerModel	InstanceModel	[Inherited from InstanceModelElement] A link always belongs to an owner instance model.
IsAnInstanceOf/ Type	Association	A link is always an instance of a given type association.
HasPrimary/ Primary	Reference	A link usually has a primary reference. Only links from an unknown object lack a primary reference (see *Null and Unknown Semantics*, p. 36).
HasSecondary /Secondary	Reference	A link usually has a secondary reference. Only links to an unknown object lack a secondary reference (see *Null and Unknown Semantics*, p. 36).

4.4 Namespaces

Certain realms in a model define independent spaces where names given to model parts must be unique. These independent spaces are called *namespaces*. Namespaces roughly correspond to the groups of classes in the metamodel given by whole/part relationships. Namespaces are language-local; i.e. there is a separate namespace for every language in a multilingual model (see *Multilingualism*, p. 40).

The following sections explain the details of the different namespaces in ConML models.

4.4.1 Type Models

4.4.1.1 Model

Each type model is a namespace with regard to:

- The classes and data types assigned to the model and not contained in any package. Each class or data type assigned to a type model that are not contained in a package must have a unique name, and the name of a class or data type not contained in any package cannot coincide with the name of another one assigned to the same type model.
- The packages in the model. Each package in a type model that is not contained in any package must have a unique name.

4.4.1.2 Package

Each package is a namespace with regard to:

- The classes and enumerated types in the package. Each class or enumerated type in a package must have a unique name, and the name of a class or enumerated type in a package cannot coincide with the name of another one in the same package.
- The sub-packages in the package. Each sub-package in a given package must have a unique name.

4.4.1.3 Class

Each class is a namespace with regard to:

- The total collection of properties, attributes and semi-associations assigned to (i.e. owned or inherited; see *Feature Inheritance*, p. 34) the class. Each property, attribute and semi-association of a class, including those inherited, must have a unique name, and the name of a property, attribute or semi-association cannot coincide with the

name of another. Similarly, the role of a semi-association cannot coincide with the name of a property, attribute or semi-association.

4.4.1.4 EnumeratedType

Each enumerated type is a namespace with regard to:

- The enumerated items assigned to (i.e. owned or inherited; see *Generalization of Enumerated Types*, p. 33) the enumerated type. Each enumerated item assigned to an enumerated type must have a unique absolute name.

4.4.1.5 EnumeratedItem

Each enumerated item is a namespace with regard to:

- The sub-items owned by the enumerated item. Each sub-item of an enumerated item must have a unique name.

4.4.2 Instance Models

4.4.2.1 Model

Each instance model is a namespace with regard to:

- The objects in the model. Each object in an instance model must have a unique identifier.

4.4.3 Model Part Names

Model part names cannot coincide with a ConML keyword.

In some contexts, and especially for model elements, the name of a model part is not enough to identify it within a model. This often happens when the element exists inside a namespace and other elements with the same name exist in different namespaces. For example, a *Person* class may have a *Name* attribute. Another class, *City*, may also have a *Name* attribute. Any reference to "the Name attribute" is thus ambiguous, since it might point to either.

In order to resolve ambiguities like this, full names and fully qualified names are used.

4.4.3.1 Names of Languages and Tags

Languages and tags are not model elements, but still they exist within models. For this reason, the full name of a language or tag coincides with its simple name.

4.4.3.2 Full and Relative Names of Packages

The full name of a package is composed from the full name of its owner package, if any, plus its own name, and separated by a full stop (“.”). For example: “Organization.People.Staff”.

The relative name of a package *P* in relation to a containing package *C* is the subset of *P*’s full name that stems from *C*. In the previous example, “People.Staff” is the relative name of the *Staff* package in relation to *Organization*; “Staff” is the relative name of the *Staff* package in relation to *Organization.People*.

Full and relative names of packages are meaningful within the context of the containing type model.

4.4.3.3 Full Names of Other Elements

The full names of model elements other than packages are obtained as follows:

- If the element exists in the model directly, rather than within any other namespace in the model, then its full name coincides with its simple name. This is the case of classes, simple data types and enumerated types (whose Name acts as simple name), plus objects (whose Identifier attribute acts as simple name).

- If the element exists within a sub-namespace inside the model, then its full name is formed by its own simple name prefixed by the full name of its enclosing element plus a full stop (“.”). This is the case of features (whose Name acts as simple name), generalizations (whose Discriminant acts as simple name) and enumerated items (whose AbsoluteName acts as simple name). For example, the full name of attribute *Name* in class *Person* would be “Person.Name”. The full name of the enumerated item *Europe/Spain/Galicia* of enumerated type *WorldLocations* would be “World-Locations.Europe/Spain/Galicia”.

4.4.3.4 Fully Qualified Names

The fully qualified name of a model part is formed as follows:

- If the model part is a language or a tag, then its fully qualified name coincides with its full name.
- Otherwise, the model part is a model element:
    - If the element is a package, then its fully qualified name coincides with its full name.
    - Otherwise:
        - If the model part belongs to an instance model or exists in a type model outside of any package, then its fully qualified name coincides with its full name.
        - If the element is contained in a package in a type model, then its fully-qualified name is formed by joining together the full name of the containing package, plus a full stop (“.”), plus the element’s full name.

4.5 Further Semantics

This section describes the semantics of some complex areas that are not covered by the descriptions of individual metamodel elements.

4.5.1 Semantics of Data Types

There are two kinds of data types in ConML: simple data types and enumerated types. Simple data types are pre-defined and cannot be changed. Also, it is assumed that an instance of class SimpleDataType for each element of BaseDateType is automatically included in every ConML type model. Enumerated types, on the contrary, are defined by the user as part of a type model.

There are some compatibility rules between data types that dictate what conversions are possible and which are not. These rules are based on whether automatic type coercion is possible from a source into a destination data type. These rules are important, for example, when changing the type of an attribute through feature redefinition (see *Feature Redefinition*, p. 35). The following table specifies the compatibility rules between data types.

Can a value of this type… \ *…be coerced into this type?*	Boolean	Number	Time	Text	Data	Enumerated
Boolean	yes	no	no	yes	yes	no
Number	no	yes	no	yes	yes	no
Time	no	no	yes	yes	yes	no
Text	no	no	no	yes	yes	no
Data	no	no	no	no	yes	no

	…be coerced into this type?					
	Boolean	Number	Time	Text	Data	Enumerated
Enumerated	no	no	no	yes	yes	(see text)

A value of an enumerated type can be coerced into another enumerated type only in the following cases:

- The source and target enumerated types are the same; this is the trivial case.
- The source enumerated type is a descendant of the target enumerated type (see *Generalization of Enumerated Types*, p. 33).

4.5.2 Enumerated Item Hierarchies

Enumerated items define named values within a well-known semantic domain. Enumerated item can be organized hierarchically in order to capture subsumption, whole/part or any other similar relationship that there might exist in said domain. For example:

- Items representing materials can be arranged to depict subsumption relationships: e.g. *Metal*, *Metal/Iron*, *Metal/Brass*, *Wood*, etc. The hierarchy in this case captures the fact that Iron and Brass are sub-categories of *Metal*.
- Items representing geographical areas can be arranged to depict whole/part relationships: e.g. *Europe*, *Europe/Spain*, *Europe/Germany*, *Africa*, etc. The hierarchy in this case captures the fact that Spain and Germany are parts of Europe.

Complex hierarchies can be assembled by combining criteria depending on modelling needs.

In any case, enumerated item hierarchies take the form of multi-rooted trees: each enumerated type can have multiple root items (i.e. items with no super-items), and each item may have at most one super-item. No mechanism equivalent to multiple generalization is allowed for enumerated items as it is for classes.

Thus, any enumerated item in a given enumerated type is always in one of the following two situations:

- It has no super-item; in this case, it is a root item.
- It has a super-item; in this case, it is a non-root item. The super-item must be an enumerated item owned by the same enumerated type or, alternatively, assigned to the generalized enumerated type, if there is one.

The absolute name of an enumerated item is equal to its name if it does not have a super-item. If it does, then it is composed as the absolute name of its super-item plus its own name and separated by a slash (“/”) character.

4.5.3 Generalization of Enumerated Types

An enumerated type can be defined to be specialized from another enumerated type. This reflects the fact that the former is a semantic refinement of the latter. An enumerated type that specializes from another is expected to add details to the latter while preserving its semantics.

With regard to generalization/specialization hierarchies of enumerated types, the following concepts are important:

- An enumerated type is an *ancestor* of another enumerated type if and only if the former is a generalized enumerated type of the latter, or an ancestor of a generalized enumerated type of the latter.
- An enumerated type is a *descendant* of another enumerated type if and only if the former is a specialized enumerated type of the latter, or a descendant of a specialized enumerated type of the latter.

Enumerated types that are defined to specialize from another enumerated type are subject to the following rules:

- They inherit all the enumerated items of the generalized enumerated type.
- They cannot declare (i.e. own) root enumerated items, but they can own enumerated items that are sub-items of those inherited from their generalized enumerated type.

For example, an enumerated type *Colours* may define items *Red* and *Blue*. Then, a new enumerated type *ColoursPlus* may be defined as specializing from *Colours* and adding items *Red/LightRed* and *Red/DarkRed*.

4.5.4 Multiple Generalization of Classes

ConML implements multiple generalization for classes, i.e. a class can have multiple generalized classes, each one through a different generalization. From the metamodel perspective, this means that for each Class instance in a model, multiple Generalization instances may exist with the Generalization role, each one, in turn, having exactly one associated Class instance with the GeneralizedClass role.

For classes having multiple generalized classes, it must be explicitly specified which of the corresponding generalizations dominates for inheritance purposes. Despite the fact that the class specializes from multiple classes, it is expected that the dominant generalization provides especially strong semantics to the class. For classes having a single generalized class, on the contrary, the only generalization involved is implicitly the dominant one.

ConML does not implement multiple specialization, i.e. a class can be specialized into multiple specialized classes, but all of them must obey to the same discriminant. In other words, only one way to partition the associated category is allowed per class. From the metamodel perspective, this means that for each Class instance in a model, only one Generalization instance may exist at most with the Specialization role; this Generalization instance, in turn, would group multiple Class instances with the SpecializedClass role under the same Discriminant value.

With regard to generalization/specialization hierarchies of classes, the following concepts are important:

- A class is an *ancestor* of another class if and only if the former is a generalized class of the latter, or an ancestor of a generalized class of the latter.
- A class is a *descendant* of another class if and only if the former is a specialized class of the latter, or a descendant of a specialized class of the latter.

A class with no generalized classes, or whose generalized classes are all abstract, may be abstract or concrete. A class with at least one concrete generalized class must be concrete.

Whether a class is involved in generalization relationships or not, how many, and what generalization is defined as dominant, determine what features it inherits and how. See *Feature Inheritance*, p. 34 for more information.

4.5.5 Feature Inheritance

Inheritance is the mechanism by which a class is automatically assigned features owned by its generalized classes. All kinds of features (properties, attributes and semi-associations) are accessible from a class through two different associations: one beginning with “Owns” and another one beginning with “Has”. For example, class Class has an OwnsAttributes as well as a HasAttributes association. The “Owns” association represents the features that are *owned* by the class, i.e. introduced and defined by the class itself (either by declaration or redefinition; see *Feature Redefinition*, p. 35), rather than inherited. The “Has” association, on the contrary, represents the collection of all the features *assigned* to the class, including both those owned by the class plus those that are inherited from generalized classes.

In ConML, any class in a model inherits the features of its generalized classes according to the following rules:

1. If the class has no generalized classes, its assigned features equal its owned features, i.e. no inheritance occurs.
2. If the class has one or more generalized classes, its assigned features equal the owned features of the class plus the assigned features of each of the generalized classes, except for those that are redefined by the class.
3. If two or more generalized classes of a class have a common assigned feature, because they have inherited it from a common ancestor class, this feature is inherited by the class only through the generalization marked as dominant (see *Multiple Generalization of Classes*, p. 34).
4. If a feature inherited by a class, and regardless of whether it is redefined by the class (see *Feature Redefinition*, p. 35), results in a name clash with a feature owned by the class, the class is deemed illegal.
5. If a feature inherited by a class from a given ancestor class results in a name clash with a different feature inherited by the class from a second ancestor class, and regardless of whether these features are redefined by the class (see *Feature Redefinition*, p. 35), the class is deemed illegal.

4.5.6 Strong Semi-Associations

A semi-association is considered to be strong if it is definitional, i.e. if it captures information that is directly relevant to the definition of the participant class. For example, Figure 11 shows class *Structure* as strongly associated to class *Place*, meaning that the definition of the former relies on that of the latter. Marking which semi-associations are strong on a model helps understand how classes work together and serves as a base for the use of the model.

The two semi-associations in a binary association can be strong; this would signal a pair of classes that are strongly coupled.

Strong semantics in the context of self-associations indicate that the involved class is recursively defined in terms of itself, either with a different role (for asymmetric self-associations) or even with the same role (for symmetric self-associations; see *Symmetric Self-Associations*, p. 37).

4.5.7 Feature Redefinition

A feature (property, attribute or semi-association) assigned to a class (see *Feature Inheritance*, p. 34) is always in one of the following situations:

- It is *declared* by the class. This means that the class declares the feature of its own. The feature is owned by the class.
- It is *inherited* by the class. This means that the class has inherited the feature from an ancestor class. The feature is not owned by the class.
- It is *redefined* by the class. This means that the class has inherited the feature from an ancestor class and redefined it to change some of its characteristics. The redefinition feature is owned by the class.

Feature redefinition requires inheritance; only features that are inherited may be redefined. The changes that a class can produce on a feature through redefinition are of the following kinds:

- The feature can be *renamed*. This may affect the outcome of the name clash detection rules described in *Namespaces*, p. 30 and *Feature Inheritance*, p. 34.
- The *definition* of the feature can be changed.

- The *cardinality* can be changed if the redefined cardinality is more restrictive than (i.e. is a subset of) the original one. For example, if the cardinality of the original feature is 1..*, the redefinition may specify 2..* or 1, but not 0..*.
- *Sorted* semantics can be changed from non-sorted to sorted, i.e. a non-sorted original feature can be redefined as sorted. However, the opposite change is not possible.
- *Constant* semantics can be changed from non-constant (i.e. variable) to constant, i.e. a non-constant original feature can be redefined as constant. However, the opposite change is not possible.
- In the case of attributes:
    - The *data type* can be changed if the data type of the redefinition can be coerced into the original one, according to the compatibility table in *Semantics of Data Types*, p. 32. For example, if the data type of the original attribute is Text, the redefinition attribute may specify Number, but not the other way around.
    - *Multilingual* semantics can be changed from multilingual to non-multilingual and vice versa.
- In the case of semi-associations:
    - The *role* can be changed. This may affect the outcome of the name clash detection rules described in *Namespaces*, p. 30.
    - *Strong* semantics can be changed from non-strong to strong, i.e. a non-strong original semi-association can be redefined as strong. However, the opposite change is not possible.
    - The *opposite class* can be changed if the opposite class of the redefinition is a descendant of the opposite class of the original semi-association.

Although changing the name and definition of features, and the role of semi-associations, has no effect on the model static semantics, it is expected that the overall sense of the redefinition does not deviate significantly from that of the original feature.

In the case of semi-associations, and from the metamodel perspective, a redefinition is an instance of the SemiAssociation class that is not directly connected to an instance of Association, as would an instance of SemiAssociation that does not represent a redefinition. Instead, a semi-association redefinition would obtain its linked association, inverse semi-association and other related data through its original feature.

Finally, please note that temporal and subjective semantics of features, as well as whole/part semantics of semi-associations, cannot be altered by redefinition.

4.5.8 Null and Unknown Semantics

In ConML, “null” means absence of fact, whereas “unknown” means presence of fact, but absence of knowledge or doubt about it. From the metamodel perspective, this means that a null-valued attribute has a matching value set with zero values; however, an unknown-valued attribute has a matching value set with at least one value having unspecified (i.e. null) contents.

There can also be the case that an object has multiple values for a given attribute (cardinality permitting), some of which are known and some unknown. In cases like this, each value instance would have non-null or null contents, respectively.

The following table summarizes the semantics of “null” and “unknown” as opposed to that of regular, non-null, non-unknown expressions of information.

Expression	Statement about fact	Statement about knowledge about fact
null	It does not exist.	n/a
unknown	It does exist.	It is not known.

Expression	Statement about fact	Statement about knowledge about fact
(other)	It does exist.	It is known.

4.5.9 Symmetric Self-Associations

Most associations are binary, meaning that they involve two different classes. It is also common to find unary associations, also called self-associations, which relate one class to itself. Of these, most are asymmetric, meaning that the single class involved plays two different roles in the association through the two corresponding semi-associations. It is the case, for example, of *Person.IsParentOf* with inverse *Person.IsChildOf*; if a given person A is parent of another person B, then B cannot be parent of A, but child of A.

However, a small number of self-associations are symmetric, meaning that the single class involved plays only one role in the association through a single semi-association. It is the case, for example, of *Person.IsMarriedTo*; if a given person A is married to another person B, then B is married to A as well.

Symmetric self-associations are well supported by the ConML metamodel. The only peculiarity that must be taken into account is that the primary and secondary semi-associations are the same one; in other words, there is a single semi-association in the association, the inverse of which is itself.

4.6 Soft Issues

Issues such as the passage of time or the different views about the same thing that different people may have are often left out of models. ConML incorporates mechanisms specifically designed to capture such “soft” issues in conceptual models. Some of these issues are dealt with in ConML as *aspects*, i.e. cross-cutting concerns that are specified separately from core modelling themes, and then “woven” into them.

4.6.1 Temporality

In order to capture temporality, a *temporal aspect* must be introduced in a type model. A temporal aspect is one class in the model, which is designated as such (see *TypeModel*, p. 7). Instances of this class will be able to work as *moments* that situate temporal versions of objects along the time axis. Moments represent time and may be as brief or as long as necessary.

Once a temporal aspect has been introduced in a model, it can be used to qualify existence and predication.

4.6.1.1 Temporal qualification of existence

Objects in an instance model may be qualified with an instance of the temporal aspect class in the corresponding type model, to indicate that the entity represented by the object only exists at that moment. Any object may be qualified in this way, no matter what its type class is.

In conventional object-oriented models, it is assumed that objects exist eternally, or uncertainly, as no indication is provided about their period of existence. In ConML, existentially unqualified objects also behave this way, but qualified objects have a well-defined lifetime.

4.6.1.2 Temporal qualification of predication

Facet sets (value sets or reference sets) in an instance model may be qualified with an instance of the temporal aspect class in the corresponding type model, to indicate that the characteristic represented by the facet set only exists at that moment. The collection of facet sets qualified for temporality by any given object is called a *phase*.

For a facet set to be predication-qualified regarding temporality, its type feature must be marked as temporal through its IsTemporal attribute (see *Feature*, p. 11).

In conventional object-oriented models, it is assumed that changing the value of an attribute of an object, or modifying the links of that object to other objects, overwrite the previous state with the new one, and that this new one becomes valid. In ConML this is the semantics of regular, non-temporal features. Temporal features, on the other hand, support the creation of a new object phase upon instance alteration. Creating a new phase means that the previous and new states are both kept separately but linked together as part of the same object. This way, a diachronic view of objects can be maintained in instance models.

Alternatively, a facet set is considered constant if its type feature is marked as such through its IsConstant attribute (see *Feature*, p. 11). Constant semantics imply that no changes may occur to the facet set once it has been created. Constant and temporal semantics are mutually exclusive.

In summary, and in relation to temporal qualification of predication, features may be:

- **Constant**, meaning that their instances cannot change. This is achieved by marking the feature as constant.
- **Variable**, meaning that their instances may change by overwriting previous states. This is the default mode and is achieved by not marking the feature as either constant or temporal.
- **Temporal**, meaning that their instances may change by optionally creating new phases. This is achieved by marking the feature as temporal.

4.6.1.3 Relationships between existential and predication temporal qualification

There are ontological connections between existential and predication qualification of temporality. Specifically, temporal existential qualification must encompass every phase. That is, an object may not have a phase whose qualifier "falls outside" the object's temporal existential qualifier. Here, the phrase "falls outside" has complex semantics, and it is often difficult to enforce the corresponding rules in an automatic way, as the containment or coverage semantics between instances of the designated temporal aspect class are not always immediate in the model. For example, the temporal qualifier *1937* (interpreted as a year) "falls inside" *20th century*, but this may not be derivable from the information in the model.

4.6.2 Subjectivity

In order to capture subjectivity, a *subjective aspect* must be introduced in a type model. A subjective aspect is one class in the model, which is designated as such SubjectiveAspect (see *TypeModel*, p. 7). Instances of this class will be able to work as *subjects* that situate subjective versions of objects over multiple voices. Subjects represent the different opinions and interpretations that different people or communities may have about the same things, and may correspond to an individual, a group or a whole community. Note that subjectivity, as defined in ConML, is not related to reported speech (such as in "Alice: Bob thinks the wall is white") but to subjectively-constructed knowledge (such as in "Alice: the wall is beautiful"); in this regard, subjectivity is related to the fact/value difference described by [4].

Once a subjective aspect has been introduced in a model, it can be used to qualify existence and predication.

4.6.2.1 Subjective qualification of existence

Objects in an instance model may be qualified with an instance of the subjective aspect class in the corresponding type model, to indicate that the entity represented by the object only exists according to that subject. Any object may be qualified in this way, no matter what its type class is.

In conventional object-oriented models, it is assumed that objects exist objectively for everyone, or uncertainly, as no indication is provided about their subjective status. In ConML,

existentially unqualified objects also behave this way, but qualified objects have a well-defined subjective status.

4.6.2.2 Subjective qualification of predication

Facet sets (value sets or reference sets) in an instance model can be qualified with an instance of the subjective aspect class in the corresponding type model, to indicate that the characteristic represented by the facet set only exists according to that subject. The collection of facet sets qualified for subjectivity by any given object is called a *perspective*.

For a facet set to be predication-qualified regarding subjectivity, its type feature must be marked as subjective through its IsSubjective attribute (see *Feature*, p. 11).

In conventional object-oriented models, it is assumed that changing the value of an attribute of an object, or modifying the links of that object to other objects, overwrite the previous state with the new one, and that this new one becomes valid. In ConML this is the semantics of regular, non-subjective features. Subjective features, on the other hand, support the creation of a new object perspective upon instance alteration. Creating a new perspective means that the previous and new states are both kept separately but linked together as part of the same object. This way, a multivocal view of objects can be maintained in instance models.

In summary, and in relation to subjective qualification of predication, features may be:

- **Objective**, meaning that a single perspective is allowed for their instances. This is the default mode and is achieved by not marking the feature as subjective. This usually corresponds to fact propositions in [4].
- **Subjective**, meaning that their instances may change depending on the agent by optionally creating new perspectives. This is achieved by marking the feature as subjective. This usually corresponds to value propositions in [4].

4.6.2.3 Relationships between existential and predication subjective qualification

Like in the case of temporality, there are ontological connections between existential and predication qualification of subjectivity. Specifically, subjective existential qualification must cover every perspective. That is, an object cannot have a perspective whose qualifier is not "included" in the object's subjective existential qualifiers. Here, the term "included" has complex semantics, and it is often difficult to enforce the corresponding rules in an automatic way, as the containment or coverage semantics between instances of the designated subjective aspect class are not always immediate in the model. For example, the subjective qualifier *Alice* may be "included" in the qualifier *Acme Ltd.* if Alice is part of Acme Ltd., but this may not be derivable from the information in the model.

4.6.3 Vagueness

Vagueness comes in two kinds: ontic vagueness or *imprecision*, which refers to the nature of things not being clear-cut (such as the boundaries of a city); and epistemic vagueness or *uncertainty*, which refers to our knowledge of things not being accurate (such as the rough estimation of a radiocarbon date).

4.6.3.1 Imprecision

Imprecision is supported in ConML through the Time data type, which may express values of variable resolution (see *BaseDataType*, p. 16), as well as the use of abstract enumerated items (see *Enumerated Item Hierarchies*, p. 33).

4.6.3.2 Uncertainty

ConML adopts an "open world" assumption by which explicit statements about not knowing something can be made. In particular, stating that a value is unknown implies that the value is

known to exist, but its contents are not known. This is different to stating that a value is null, which implies that the value is known not to exist (see *Null and Unknown Semantics*, p. 36).

In addition, certainty markers may be used to qualify existence and predication, according to the DegreeOfCertainty enumeration. This allows an instance model to express specific beliefs about the existence of objects (see *Object*, p. 24) as well as individual predicative facts conveyed by facets (see *Facet*, p. 27).

4.6.4 Multilingualism

Models can be expressed in multiple human languages. In order to make a model multilingual, a number of languages may be defined within it. Specific languages will be able to index translated versions of objects. A model always has at least one language but may have many. One of them is the default language, which is assumed when no explicit language reference is made.

Language names follow the IETF language tag recommendation (https://en.wikipedia.org/wiki/IETF_language_tag), but employ underscores instead of hyphens so that legal model part names are obtained.

On creation, a model contains a single language, which by default is en-GB. New languages may be added to a model at any time, and any language may be made the default one in a model at any time. Languages may be deleted from a model at any time, except for the default language.

4.6.4.1 Type models

In a type model, most text-typed properties of model elements may be specified in multiple languages. In particular, the following model texts are multilingual:

- Model.Name and Model.Description
- Language.Description
- Tag.Name
- Package.Name and Package.Description
- Type.Name and Type.Definition
- EnumeratedItem.Name and EnumeratedItem.Definition
- Generalization.Discriminant
- SemiAssociation.Role

4.6.4.2 Instance models

In a type model, attributes of type Text can be marked as multilingual through the Is-Multilingual attribute (see *Attribute*, p. 13). An attribute marked as such is one that supports the specification of contents in multiple languages in conforming instance models.

In an instance model, translated versions of objects can be created; they are called *translations*. A translation, therefore, is a version of an object as expressed in a given language and, as such, it must refer to a particular language in the instance model. This is achieved by linking each translation to a language. From the metamodel perspective, this means that the object's value sets that represent the object state as expressed in said language will have a non-null language qualifier.

4.7 Type Model Extension

A type model can be defined as either a standalone model or as an extension from a base type model. A model that is constructed through extension is called a *particular model*. Particular models are guaranteed to be Liskov-compatible with their base. This means that any instance model conforming to a particular model is guaranteed to conform also to its base model. Figure 2 shows the TypeModel.Extends association that enables this.

When a particular model is created, it contains every element in its base model. From that point on, changes can be made to the particular model to add, modify or delete model elements. Model elements in the particular model that are taken from the base model are called *reused elements*; model elements that have been added during extension, and which do not exist in the base model, are called *extended elements*.

During extension, Liskov compatibility is not maintained by restricting what changes may be carried out on a base model; in fact, almost any change is possible during extension. Rather, compatibility is maintained through the application of various *reinterpretation rules* that can recast an instance model conforming to the particular model as to conform to the base model. These rules apply to specific extension mechanisms, which are outlined below. In the next sections, B refers to the base model, P to the particular model, and K to an instance model of P.

4.7.1 Adding Enumerated Types and Items

Enumerated types or items may be added to a model during extension. If an enumerated type is added that specializes from another in the base model, the new enumerated type may not declare root enumerated items (as described in *Enumerated Item Hierarchies*, p. 33).

If a non-root enumerated item is added to an enumerated type in the base model, the following reinterpretation rule applies:

RR.1

A value in K pointing at an extended non-root enumerated item in P is reinterpreted as pointing to the most immediate ancestor of said enumerated item that exists in B.

If a root enumerated item is added to an enumerated type in the base model, the following reinterpretation rule applies:

RR.2

A value in K pointing at an extended root enumerated item in P is reinterpreted as unknown.

4.7.2 Adding Classes

Classes may be added to a model during extension. If a non-root class (i.e. a class that specializes from other classes that pre-exist in the base model) is added, the following reinterpretation rule applies:

RR.3

An object in K having a non-root extended class in P as type is reinterpreted to have the most immediate ancestor of said class that exists in B as type.

If a root class (i.e. a class that does not specialize from any other class) is added, the following reinterpretation rule applies:

RR.4

An object in K having a root extended class in P as type is reinterpreted as non-existing.

Note that adding root classes during model extension is uncommon, as it tends to alter the base model's scope and purpose.

4.7.3 Adding Features

Properties, attributes and associations may be added to a model during extension, involving either reused or extended classes. If they involve extended classes, no reinterpretation rule applies, as the corresponding objects will be ignored or abstracted during reinterpretation, as per RR.3 and RR.4. However, if the added features pertain to reused classes, then the following reinterpretation rules apply:

RR.5

A value in K having an extended attribute in P as type that belongs to a reused class is reinterpreted as non-existing.

RR.6

A link in K having an extended association in P as type that connects reused classes is reinterpreted as non-existing.

4.7.4 Modifying Packages, Enumerated Types, Enumerated Items, and Classes

Packages, enumerated types, enumerated items and classes may be reamed during extension, so that the model elements better fit the extended model's terminological preferences. When doing this, the following reinterpretation rule applies:

RR.7

An object in K referring to a renamed model element in P as type is reinterpreted as referring to the original model element in B.

4.7.5 Modifying Features

Features may be renamed during extension in the same manner as other kinds of model elements, as described in the previous section. The same reinterpretation rule applies.

In addition, some details of properties, attributes and semi-associations maybe changed during extension by using the redefinition mechanism described in *Feature Redefinition*, p. 35. Feature redefinition is simple and safe, and the recommended way to modify features in a particular model. No specific reinterpretation rules apply.

4.7.6 Hiding Attributes

Attributes may be hidden during extension if there is the guarantee that every one of their instances will always have the same value. To do this, the attribute to be hidden is annotated as such, and a default value to use is specified. When doing this, the following reinterpretation rule applies:

RR.8

An object in K having a class in P as type for which an attribute has been hidden is reinterpreted as having the specified default values in P for said attribute.

4.7.7 Deleting Enumerated Types or Items

Enumerated types may be deleted during extension if every class (see below) that contains attributes of that type is also deleted. Also, enumerated items may be deleted during extension without any limitations. No reinterpretation rules apply.

4.7.8 Deleting Classes

Classes may be deleted during extension. A reused class may be deleted if:

1. Every one of its descendant classes is also deleted.
2. No classes are kept in the model having semi-associations pointing to the deleted class with a minimum cardinality greater than zero.
3. The class is not an aspect class (subjective or temporal) or, if it is, there are no features in the model marked with the corresponding aspect.

No reinterpretation rules apply.

4.7.9 Deleting Features

Features may be deleted during model extension only if they have a minimum cardinality of zero. In the case of associations, a complete binary association may be deleted if both semi-associations have minimum cardinalities of zero.

When doing this for attributes, the following reinterpretation rule applies:

RR.9

An object in K having as type a reused class in P from which an attribute has been deleted is reinterpreted as having a value with null contents for that attribute in B.

In the case of associations, no reinterpretation rules apply.

4.8 Metainformation

In ConML, metainformation refers to instance model elements that document other model elements. Metainformation may be applied to any kind of model elements, either type or instance. Since metainformation is composed by objects, there may be metainformation that documents other metainformation.

The objects that comprise metainformation usually reside in a different model to that being documented. However, they may also be part of the same model. For example, it is common that an object in an instance model documents a class in a type model, but it may also document another object in the same instance model. In this regard, metainformation constitutes regular information, made of regular objects, which happen to document model elements (in the same or a different model). Consequently, the term "metainformation object" must be understood as "an object that works as metainformation in relation to some well-known model elements". Also, there is no such a thing as a metainformation model; objects that work as metainformation are contained in regular instance models.

An object may document zero to many model elements, and a model element may be documented by zero to many metainformation objects, as expressed by the Object.Documents association (see *Object*, p. 24).

4.9 References between Models

Models may refer to other models in a variety of situations:

- **Conformance**. An instance model conforms to a type model.
- **Extension**. A particular type model extends a base type model.
- **Metainformation**. A metainformation instance model describes elements in a model.

In any of these situations, the model making the reference is *dependent* on the model being referred to, which is its *dependee*. Optionally, the dependee model may be sealed before the dependency is established. A *sealed* model cannot be changed (unless the version number is incremented) and is thus safe to redistribute as a reliable source for reference. New versions of a sealed model can be generated and altered, though. A model that is not sealed is called unsealed or *open*.

4.9.1 Reference Updating

In general, and regardless of whether a dependee model is sealed or not, it may be *updated* on demand to a different dependee model, so that the dependent model's reference is recalculated. The updated dependee model is often a newer version of the previous one, although it may also be an older version, or even a different model altogether if it is a direct or indirect extension of any version of the currently referenced model. More specifically:

- **Conformance**. An instance model may update its type model to a different version of it, or to a type model that is directly or indirectly extended from any version of it. On updating, the instance model is reinterpreted to conform to the updated type model.
- **Extension**. A particular type model may update its base model to a different version of it, or to a type model that is directly or indirectly extended from any version of it. On updating, the particular model is reinterpreted as an extension of the updated model so that it stays Liskov-compatible with its previous form.
- **Metainformation**. No updating may take place regarding metainformation relationships, as metainformation is tightly dependent on the identity and semantics of the model elements it documents.

For example, the type model of an instance model may be updated so that a newer version is obtained, replaced for the current one, and the contents of the instance model reinterpreted according to this newer version.

4.9.2 Reuse of Referenced Model Elements

Model elements in the dependee model are available to, and may be reused by, the dependent model according to the following rules.

4.9.2.1 Aspect classes

The aspect classes of a base model are visible and usable by a particular model.

A particular model may change an aspect class only to a subclass of the one specified by its base model.

4.9.2.2 Languages

Languages in a base model are visible and usable by a particular model.

A particular model may add new languages. It may also delete reused languages.

Languages in a type model are invisible and unavailable to an instance model. The languages defined and used by an instance model are completely independent of those in its type model.

4.9.2.3 Tags

Tags in a base model are visible and usable by a particular model.

A particular model may add new tags and apply them to reused or extended elements. It may also apply or un-apply reused tags to reused or extended elements, and even delete reused tags.

Tags in a type model are invisible and unavailable to an instance model. The tags defined and used by an instance model are completely independent of those in its type model.

4.9.2.4 Packages

Packages in a base model are visible and usable by a particular model, as they organize the contents of potentially most of its elements.

A particular model may repackage reused or extended elements in any manner, by adding them to packages (either reused or extended) or by unpackaging them. A particular model may also delete reused packages.

5 Notation

This section describes how to compose diagrams and represent ConML models on paper, screen or other graphical medium.

5.1 General Notation

Some notational artefacts may be used in multiple diagram or table types; they are described in the following sections.

5.1.1 Comments

A comment is a text with no conceptual value to the model, but which is included in a diagram or table as an annotation or for assistance to the human reader. To show a comment, the comment text must be written in a compact block and prefixed by two consecutive slash characters (“/”). Figure 11 shows an example next to the *Structure* class.

5.1.2 Markers

Markers are short abbreviations that are attached to model element names or other texts in order to add extra information (e.g. “A” to indicate that a class is abstract).

If multiple markers of the same kind are to be applied together, they must be shown as a list separated by commas (“,”).

5.1.2.1 Abstract marker

The abstract marker indicates that a class is abstract. It is shown by a capital “**A**”.

5.1.2.2 Aspect markers

Aspect markers are short abbreviations that are attached to model element names or other expressions in order to show that they possess specific semantics in relation to an aspect in the model (e.g. “T” to indicate that a class constitutes the temporal aspect of a model).

The following aspect markers exist:

- Constant semantics, indicated by a capital “**K**”.
- Temporal semantics, indicated by a capital “**T**”.
- Subjective semantics, indicated by a capital “**S**”.
- Multilingual semantics, indicated by a capital “**L**”.

5.1.2.3 Certainty markers

Certainty markers are simple symbols that are attached to objects or facets in order to show their degree of certainty, according to the DegreeOfCertainty enumeration.

The following certainty markers exist:

- Certain, indicated by an asterisk “*****”.
- Probable, indicated by a plus sign “**+**”.
- Possible, indicated by a swung dash “**~**”.
- Improbable, indicated by a minus sign “**-**”.
- Impossible, indicated by an exclamation mark “**!**”.

5.1.3 Selector Dots

A selector dot is a small black dot placed next to an element in a diagram in order to “select” it amongst multiple alternatives. They are used in a number of situations, such as to designate a dominant generalization.

5.1.4 Cardinalities

The cardinality of a feature may be shown through a text that is composed by using the following rules:

- If the minimum and maximum cardinalities coincide, show a single number that expresses them. An example is shown for the *Location* role of class *Place* in Figure 11.

- If the minimum and maximum cardinalities are different, show the minimum cardinality followed by two consecutive full stop characters (“.”) plus the maximum cardinality. An example is shown for attribute *Place.Region* in Figure 11.
- If any cardinality is very high or indeterminate, use an asterisk character (“*”) to depict it. Various examples can be seen in Figure 11.

Furthermore, an upwards angle character (“^”) must be appended to the cardinality text if the feature is sorted. The *Area.Contains* semi-association in Figure 11 is depicted as sorted.

5.1.5 References to Data Types

Whenever a reference to a data type is needed, the following rules apply:

- If the referenced data type is a simple data type, then state the base data type directly (*Text* or *Number*, for example).
- If the reference data type is an enumerated type, then use the keyword “**enum**” followed by the name of the enumerated type in question (see example for *StructureType* in Figure 11).

5.1.6 Languages

The language selection operator is the pipe character (“|”).

There are three manners to reference a multilingual text-typed value set in ConML:

- **Multilingual**, such as in *a.Name*|. This resolved to a collection of value sets indexed by language.
- **Explicit**, such as in *a.Name*|*en*. This resolves to a single value set corresponding to the specified language, or to that that most closely matches the provided language name.
- **Implicit**, such as in *a.Name*. This resolves to a single value set corresponding to the default language.

The keyword “**language**” may be used to establish the default language for a diagram. A diagram should be interpreted in relation to language as follows:

- If a language declaration of the form "language <name>" is present at the top, then the specific language should be assumed as default.
- Implicit references should be assumed to employ said default language.
- Explicit references should be honoured.
- No multilingual references may be used in diagrams.

5.1.7 Applies Relationships

An “applies” relationship depicts the fact that something in a diagram applies to a particular model element. Applies relationships are shown by a solid line from the source element towards the target element and having a hollow circular end on the latter side. An example may be seen in Figure 11 for metainformation object *cd1* documenting the *Point* class.

The circular end of the applies relationship line must be drawn so that it intersects or overlaps the target element. For example, if the target element is a class, then the circular end of the applies relationship line must intersect the class rectangle border.

Applies relationships are used to depict metainformation objects documenting model elements in diagrams of any kind.

5.1.8 Multi-Node Boxes

Large or complex diagrams usually involve attaching connectors (such as generalization/association or link lines) to many diagram nodes (such as class or object boxes).

In these cases, the diagram can become cumbersome and difficult to understand. To simplify situations like this, a multi-node box notation can be used, as shown in Figure 10.

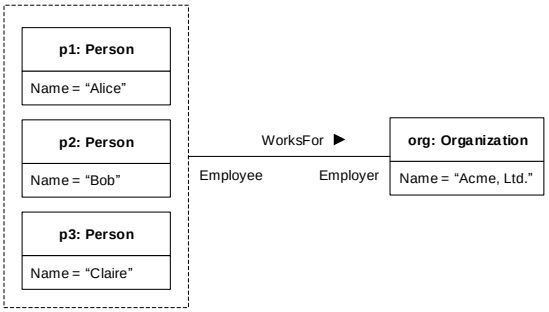

Figure 10. Sample object diagram showing a link line attached to a multi-node box containing three objects.

In the example, a single *WorksFor* link is depicted, having one end attached to a multi-node box. This is equivalent to drawing three individual links, each attached to an object box inside the multi-node box.

5.2 Class Diagrams

A class diagram depicts a collection of classes and other elements of a given type model in a condensed and visual fashion. Class diagrams are not meant to be complete, and they do not show all the information that exists in the underlying model. On the contrary, their objective is to give the reader the right amount of information to gain a good understanding of the model, leaving out the details. These details can be expressed separately in the form of tables; see *Specification Tables*, p. 56.

ConML class diagrams are relatively similar to those defined by UML [2], and it is expected that a reader who is familiar with UML will understand a ConML class diagram with ease.

5.2.1 Classes, Properties and Attributes

As Figure 11 shows, each class is depicted as a rectangle divided in two sections.

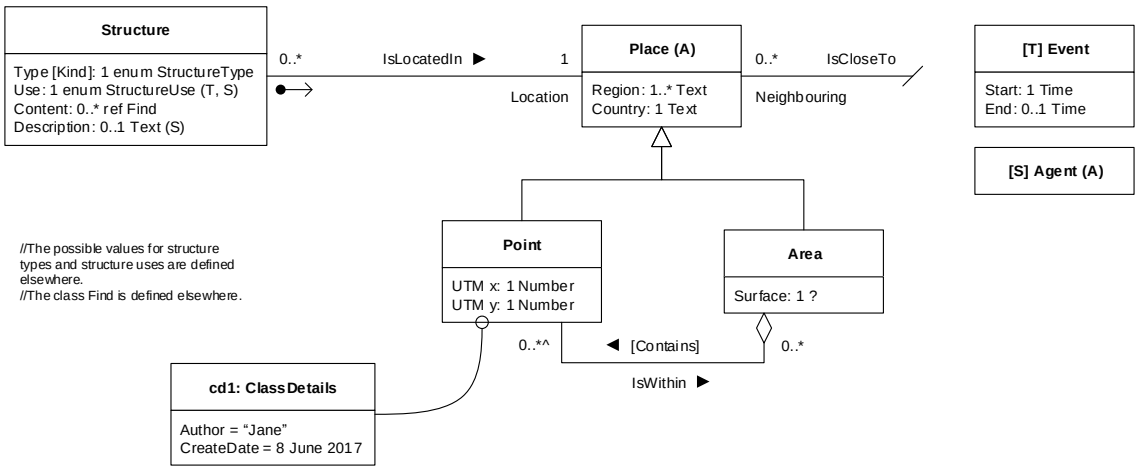

Figure 11. Sample class diagram showing classes *Structure*, *Place*, *Point* and *Area*, as well as their properties and attributes, plus some associations. Aspect classes *Event* and *Agent* are also shown, as well as a metainformation object *cd1*.

The upper section of the rectangle contains the class name, centred, and followed by an abstract marker (see *Abstract marker*, p. 45) enclosed in parentheses (“(“, “)”) if the class is abstract.

If the class constitutes one or more aspects (see *Soft Issues*, p. 37), its name must be preceded by the necessary aspect markers (see *Aspect markers*, p. 45) enclosed in brackets ("[", "]"). Figure 11 shows class *Event* marked as constituting the temporal aspect of the model.

The lower section of the rectangle contains a list of the properties and attributes that are owned by the class, one per line. For each property or attribute, its name is shown followed by a colon character (":"), an optional cardinality expression and then the attribute type or a question mark ("?") for properties. If no cardinality expression is shown, a cardinality of exactly 1 is assumed by default.

Properties and attributes that redefine others are shown by adding the name of the original feature in brackets ("[", "]") after the feature name. Figure 11 shows an example with attribute *Type* of class *Structure*, which redefines attribute *Kind* of an ancestor class. If the feature name does not change in the redefinition, repeating its name is not necessary.

Properties or attributes may be marked as constant, temporal, subjective, or a combination by appending the necessary aspect markers (see *Aspect markers*, p. 45) enclosed in parentheses ("(", ")") to the corresponding line. Bear in mind that constant and temporal markers are incompatible (see *Temporal qualification of predication*, p. 37). Figure 11 shows attribute *Use* of class *Structure*, which is marked as both temporal and subjective.

Properties and attributes of a class may be hidden in order to save space and present a more compact view of the model; to accomplish this, the lower section of the class rectangle must be completely omitted, as shown for *Agent* in the sample diagram.

5.2.2 Generalization Relationships

Each generalization relationship is shown as an arrow with multiple bases, starting at each of the specialized classes and pointing at the generalized class. Line style is continuous, and the arrowhead is a hollow triangle. Figure 11 shows an example with classes *Point* and *Area*, which are specialized from class *Place*.

The discriminant of the generalization relationship may be optionally shown next to the corresponding arrowhead.

In the case of a class with multiple generalization relationships, a selector dot must be placed next to the point where the line corresponding to the dominant generalization meets said class.

Indirect generalization relationships (i.e. a relationship between a class and an ancestor that is not its immediate generalized class) can also be depicted in diagrams by using a dashed line.

5.2.3 Associations and Semi-Associations

There are two manners in which associations and semi-associations can be depicted: expanded and compact.

5.2.3.1 Expanded style

The expanded style is the most expressive, since it can depict all the information pertaining to an association and its package semi-associations. The notation varies depending on whether the association is asymmetric (most common case) or a symmetric self-association (see *Symmetric Self-Associations*, p. 37).

5.2.3.1.1 Asymmetric associations

Using the expanded style, each asymmetric association is shown as a continuous line linking the two participant classes, or the single participant class to itself in the case of self-associations. Details of the two related semi-associations are shown next to this line, as described below.

The name of each semi-association may be shown centred on the line, midway between the participant class rectangles. A solid triangular arrowhead with no line must be placed next to this text in order to indicate in which way the name of the semi-association must be read. For example, in Figure 11 it can be read that "each structure *is located in* a place" (rather than the incorrect "each place is located in multiple structures"). In most cases, showing the name of the primary semi-association is enough. If the names for both semi-associations are displayed, they must be shown on opposite sides of the association line, as shown in Figure 11 for the association between *Point* and *Area*. Semi-associations that redefine others must show the name of the original semi-association in brackets ("[", "]") after the semi-association name; note that this notation applies only to the semi-association being explicitly mentioned, and not to its inverse. If the semi-association name does not change in the redefinition, repeating its name is not necessary; Figure 11 shows an example with semi-association *Contains* of class *Area*, which keeps the name unchanged. Similarly, the roles of semi-associations that redefine others must show the original role in brackets ("[", "]") after the semi-association role; if the role does not change in the redefinition, repeating it is not necessary. Semi-associations may be marked as temporal, subjective, or both by appending the necessary aspect markers (see *Aspect markers*, p. 45) enclosed in parentheses ("(", ")") to the corresponding line.

The role for each semi-association, if any, must be shown next to the end of the association line that touches the opposite class. For example, Figure 11 shows that the place where a structure is located is called its *location*. Similarly, the cardinalities of each semi-association must be shown next to the end of the association line that touches the opposite class.

If a semi-association has "whole" semantics, a hollow diamond must be added to the end of the association line that touches the participant class rectangle. This diamond is read as "whole", "composite" or "aggregate". An example can be seen for the class *Area* in Figure 11.

Finally, if a semi-association has strong semantics, a short arrow with a solid circle at its base and an open arrow head pointing in the direction of the opposite class must be shown next to the end of the association line that touches the participant class. An example can be seen for the class *Structure* in Figure 11.

5.2.3.1.2 Symmetric self-associations

Using the expanded style, each symmetric self-association is shown as a continuous line protruding from the participant class and finished with a short diagonal stroke. Figure 11 shows an example regarding class *Place*. Details of the related semi-association are shown next to this line, as described below.

The name of the semi-association may be shown on the line. An arrowhead, identical to those used for asymmetrical associations, can be used, but is not necessary since directionality of reading is meaningless. Semi-associations that redefine others must show the name of the original semi-association in brackets ("[", "]") after the semi-association name; note that this notation applies only to the semi-association being explicitly mentioned, and not to its inverse. If the semi-association name does not change in the redefinition, repeating its name is not necessary. Similarly, the roles of semi-associations that redefine others must show the original role in brackets ("[", "]") after the semi-association role; if the role does not change in the redefinition, repeating it is not necessary. Semi-associations may be marked as temporal, subjective, or both by appending the necessary aspect markers (see *Aspect markers*, p. 45) enclosed in parentheses ("(", ")") to the corresponding line.

The role for the semi-association, if any, must be shown next to the end of the association line that touches the participant class. For example, Figure 11 shows that the place that is close to another place is called its *neighbouring* place. Similarly, the cardinality of the semi-association must be shown next to the end of the association line that touches the participant class. Strong semantics can be shown in the same way as for asymmetric associations.

5.2.3.2 Compact style

The compact style is less expressive than the expanded style, since it can only depict a subset of the information pertaining to an association and its package semi-associations. However, it takes less room in the diagram and may be more convenient in some situations.

Symmetric self-associations (see *Symmetric Self-Associations*, p. 37) cannot be depicted using the compact style, since they cannot involve whole/part semantics. In addition, the compact style can only depict asymmetric associations that match the following patterns:

- "Contents" associations. These correspond to whole/part semi-associations with a defined role for the opposite class and a 0..1 cardinality for the participant class.
- "Shared" associations. These correspond to whole/part semi-associations with a defined role for the opposite class and a 0..* cardinality for the participant class.
- "Reference" associations. These correspond to non-whole/part semi-associations with a defined role for the opposite class and a 0..* cardinality for the participant class.

In either case, each semi-association is shown as a line of text in the lower section of the class rectangle, together with properties and attributes. For each semi-association, the role of the opposite class is shown followed by a colon character (":"), an optional cardinality expression, the keyword "**con**" (for "contents"), "**sha**" (for "shared"), or "**ref**" (for "reference"), depending on the semi-association type, and then the name of the opposite class. If no cardinality expression is shown, a cardinality of exactly 1 is assumed by default.

Semi-associations that redefine others must show the role of the original semi-association in brackets ("[", "]") after the semi-association role; note that this notation applies only to the semi-association being explicitly mentioned, and not to its inverse. If the semi-association role does not change in the redefinition, repeating its role is not necessary. Semi-associations may be marked as temporal, subjective, or both by appending the necessary aspect markers (see *Aspect markers*, p. 45) enclosed in parentheses ("(", ")") to the corresponding line.

Strong semantics cannot be shown using the compact style.

An example of a reference semi-association depicted using the compact style can be seen for the class *Structure* in Figure 11.

5.2.4 Packages

The grouping of elements into packages can be shown in class diagrams by using two different mechanisms. Figure 12 shows an example.

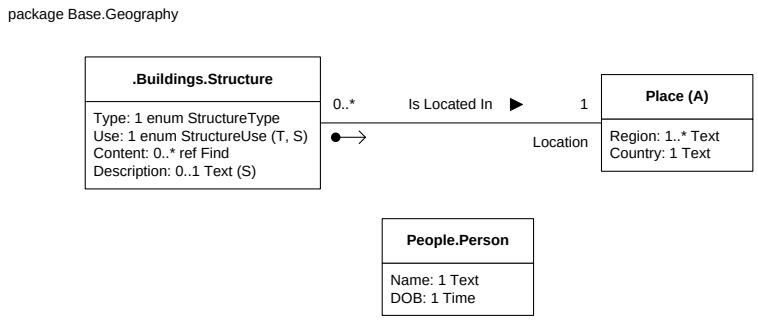

Figure 12. Sample class diagram showing an overall package declaration at the top left, plus explicit package declarations for class *Structure* and *Person*.

An overall package declaration can be displayed anywhere in a class diagram by using the keyword "**package**" followed by the full name of the package (*Full and Relative Names of Packages*, p. 31); Figure 12 shows an example for the *Base.Geography* package. This means that any class or enumerated type that is introduced in the diagram must be considered to be a member of the package unless explicitly indicated otherwise. In the Figure 12 example, the *Place* class is an implicit member of the *Base.Geography* package.

In addition, an explicit package declaration may be made for individual model elements by preceding their name by a package name plus a full stop (“.”). This is interpreted as a full package name unless an extra full stop (“.”) is used as a prefix, in which case it is interpreted to be a package name relative to the overall diagram-level package. Figure 12 shows the example of the *Structure* class, which is declared as a member of the relative-named *Buildings* package within the overall *Base.Geography* package declared at the diagram level. The same figure also shows the example of the *Person* class, which is declared as a member of the fully named *People* package.

5.2.5 Languages

The language of a class diagram may be assumed to be the default or, alternatively, explicitly established by using a language declaration. In Figure 13, the same class is shown in two diagrams, using English and Spanish respectively.

Figure 13. Sample class diagrams showing an overall language declaration at the top left of each one.

An overall language declaration can be displayed anywhere in a class diagram by using the keyword “**language**” followed by a language name (see *Multilingualism*, p. 40).

Text-valued attributes may be marked as multilingual by appending the corresponding aspect marker (see *Aspect markers*, p. 45) enclosed in parentheses (“(“, “)”) to the corresponding line. Figure 13 shows attribute *Name* of class *Building*, which is marked as multilingual.

5.3 Object Diagrams

An object diagram depicts a collection of objects plus other elements in a given instance model in a visual fashion that is easy to apprehend. Since objects, values and links are instances of classes, attributes and associations, an object diagram may also include elements from the type model to which the instance model conforms. Like in the case of class diagrams, ConML object diagrams are relatively similar to those defined by UML [2].

5.3.1 Objects and Values

As Figure 14 shows, each object is depicted as a rectangle divided in two sections.

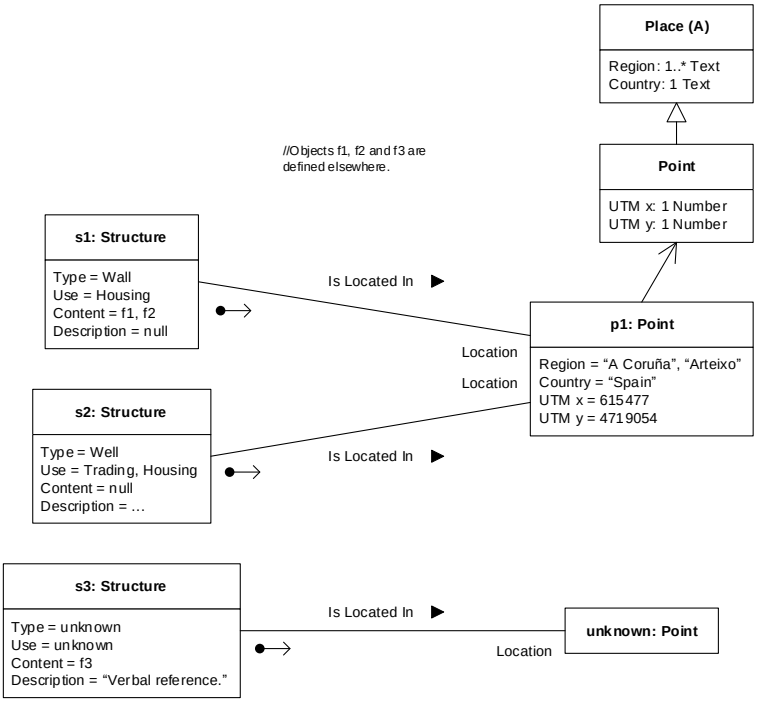

Figure 14. Sample object diagram showing objects *s1*, *s2* and *s3* (of class *Structure*) and object *p1* (of class *Point*), together with their values and links. Classes *Point* and *Place* are shown as well for the sake of clarity.

The upper section of the rectangle shows the identifier of the object followed by a colon character (":") plus the name of its type class, all of it centred. For example, objects *s1* and *s2* in Figure 14 are instances of class *Structure*.

The lower section of the rectangle contains a list of the value sets of the object, one per line. For each value set, the associated attribute name is shown followed by an equals sign ("=") and the contents of the values in the set. If no values exist in the set, the keyword "**null**" must be used, as shown for the value of *Description* of object *s1* in Figure 14. Otherwise, contents must be depicted following some rules that depend on the data type of the corresponding attribute:

- **Boolean**. Show the keyword "**true**" or "**false**".
- **Number**. Show the number in question, using a minus sign character ("-") for negative numbers and a decimal dot (".") to separate decimal fractions. Avoid thousands separators and other accessory characters. See, for example, the values for *UTM x* and *UTM y* of object *p1* in Figure 14.
- **Time**. Show the time in question.
- **Text**. Show the text in question between double quotation marks (""). See, for example, the value for *Country* of object *p1* in Figure 14.
- **Data**. Raw data contents can rarely be displayed on a diagram, so an ellipsis sign ("…") is often used. See, for example, the value for *Description* of object *s2* in Figure 14. A comment can be added to indicate any relevant information regarding the contents of a value of this type.
- **Enumerated types**. Show the absolute name, full name or fully qualified name of the enumerated item in question, depending on the situation.

If there is a single value in the value set, show it as explained. If there are multiple values in the set, the contents of each value must be shown forming a list using the comma character (",") as a separator. The values of attribute *Region* of object *p1* in Figure 14 depict an example.

If the content for a value is known to exist, but it is not known or there are doubts about it, the keyword "**unknown**" must be used (see *Vagueness*, p. 39), as shown for the value of *Type* or

Use of object *s3* in Figure 14. Alternatively, if the content of a value is not relevant, or cannot be depicted satisfactorily, an ellipsis character ("…") may be used instead of the content to signal its purposeful omission. The value of *Description* of object *s2* in Figure 14 shows an example.

Object values may be hidden altogether in order to save space and present a summarized view of the model; to accomplish this, the lower section of the object rectangle must be completely omitted, as shown for *unknown: Point* in the sample diagram.

5.3.2 Links

There are two manners in which links can be depicted in object diagrams: expanded and compact. These correspond to the two styles in which semi-associations can be shown in class diagrams (see *Associations and Semi-Associations*, p. 48).

It is recommended that the style chosen to represent a link matches the style chosen to represent the associated association.

5.3.2.1 Expanded style

Using the expanded style, each link is shown as a continuous line linking the two participant objects. The names of the semi-associations that define the type association of the link may be shown centred on the line, midway between the participant object rectangles. Like in the case of semi-associations in class diagrams, a solid triangular arrowhead with no line must be placed next to this text in order to indicate in which way the name of the semi-association must be read. The role of each semi-association, if any, may be shown next to the end of the link line that touches the object playing the role. For whole-part relationships, a hollow diamond must be shown next to the "whole" object rectangle, similarly to that for whole-part relationships in class diagrams. For links involving strong semantics, a short arrow identical to those used to mark strong semi-associations in class diagrams must be shown next to the corresponding object rectangle. Examples of links can be seen in Figure 14 between objects *s1* and *p1*, and between *s2* and *p1*.

If one of the participant objects in the link is unknown, it must be shown by using the keyword "**unknown**" instead of an object identifier, the appropriate class name, and an empty value list. An example can be seen in Figure 14 for the link connected to object *s3*. If one of the participant objects in the link is not relevant, or cannot be depicted satisfactorily, an ellipsis character ("…") may be used instead of an object identifier to signal its purposeful omission.

5.3.2.2 Compact style

Using the compact style, links are shown by displaying references in the lower section of the object rectangle together with values, grouped by reference set and showing one per line. For each reference set, the role of the opposite class is shown followed by an equals sign ("=") and the identifiers of the referenced objects. If no references exist in the reference set, the keyword "**null**" must be used. If there are multiple references in a set, the identifier of each referenced object must be shown forming a list using the comma character (",") as a separator. If the referenced object is unknown, use the keyword "**unknown**" instead of an object identifier. If one of the participant objects in the link is not relevant, or cannot be depicted satisfactorily, an ellipsis character ("…") may be used instead of an object identifier to signal its purposeful omission. Strong semantics cannot be shown when using the compact style.

Figure 14 contain various examples of compact-style links through the semi-association with role *Content*.

5.3.3 Elements of the Type Model

Since object diagrams focus on depicting information elements from an underlying instance model, which must conform to a particular type model, it is sometimes convenient to include

some elements from said type model in an object diagram for the sake of reference clarity and additional information.

Instantiation relationships between classes and objects can be depicted in object diagrams; each instantiation relationship is shown as a continuous arrow with open head, based on the instance object rectangle and pointing at the type class rectangle.

For example, although Figure 14 is an object diagram, it includes classes *Point* and *Place*, because the objects in it (*p1*, majorly) are best understood in the presence of these classes.

Indirect instantiation relationships (i.e. a relationship between an object and an ancestor of its type class) can also be depicted in diagrams by using a dashed line. For example, a dashed line should be used to connect *p1* to *Place* in Figure 14.

Any element from the type model that is shown in an object diagram must follow the notational rules and guidelines described for class diagrams.

5.3.4 Qualification

Different kinds of qualifiers can be included in object diagrams both for existential and predication qualification.

5.3.4.1 Existential qualification

Existential qualification can be depicted in object diagrams by adding a *qualifier expression* under the object identifier for either temporality or subjectivity. A temporal qualifier expression is composed of the "at" symbol ("@") followed by an expression that resolves to the corresponding moment object. Figure 15 shows an example for object *s7*, which displays a perspective existence qualifier corresponding to *John's Team*.

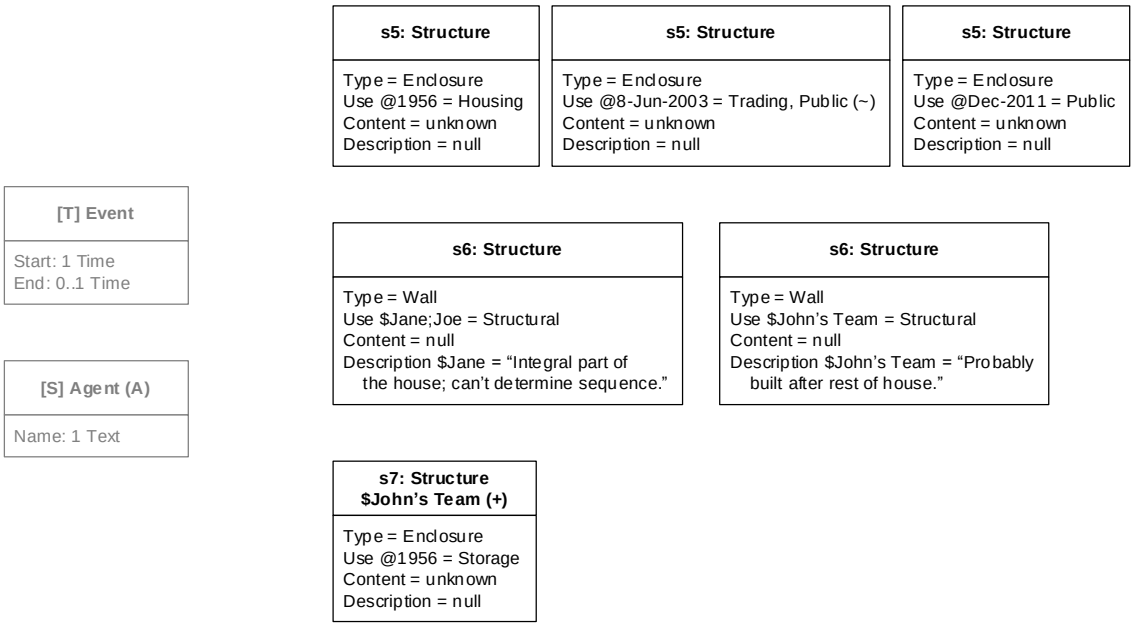

Figure 15. Sample object diagram showing some phases for object *s5* and some perspectives for object *s6*, as well as an existence-qualified object, *s7*, and a predication-qualified value for *s5.Use*. The expressions starting with "@" are temporal qualifiers, and those starting with "$" are subjectivity qualifiers. Aspect classes are shown as well for reference purposes.

In addition, certainty of existence may be indicated by adding a certainty marker in parenthesis under the object identifier. Figure 15 shows an example for object *s7*.

5.3.4.2 Predication qualification

In addition to existential qualification, predication qualification can also be shown in object diagrams. In Figure 15, three phases for object *s5* are shown, each with different values for the *Use* attribute, which is marked as temporal in the class diagram (see Figure 11). This represents the fact that the use of the structure represented by *s5* has changed over time. Each phase is described by a phase qualifier attached to *Use* that contains enough information as to resolve to an instance of the temporal aspect class, *Event* in this example.

Similarly, perspectives can be depicted by adding a perspective (i.e. subjective) qualifier expression under the object identifier. This is composed of the "according to" symbol ("$") followed by an expression that resolves to the corresponding subject objects. In Figure 15, three perspectives for object *s6* are shown as an example, each with different values for the *Description* attribute, which is marked as subjective in the class diagram (see Figure 11). This represents the fact that the description of this structure is different to different people and groups. Each perspective is described by a perspective qualifier attached to *Description* that contains enough information as to resolve to an instance of the subjective aspect class, *Agent* in this example. Note that two perspectives of *s6* in Figure 15 are shown within a single object box (the one on the left-hand side) with a compound perspective qualifier attached to *Description*, meaning that both referenced subjects share the same point of view in this case.

Predication qualifiers can also be displayed on links; Figure 16 shows an example.

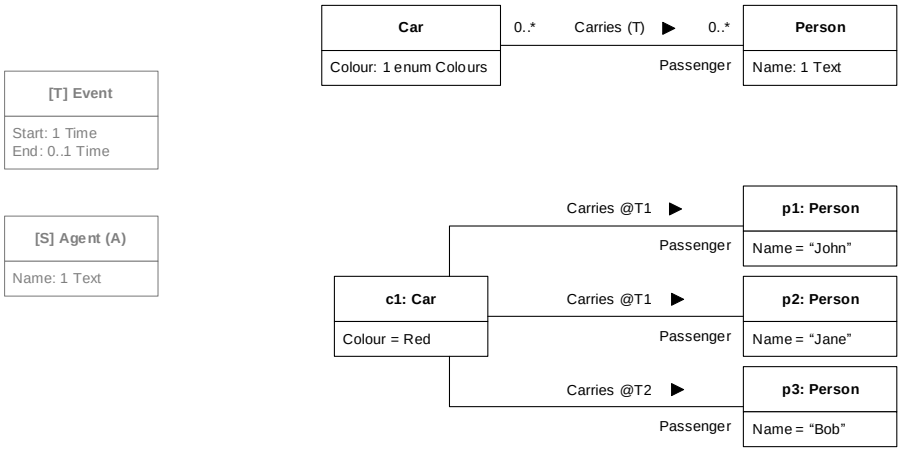

Figure 16. Sample class and object diagram showing two phases for object *c1*. Phase qualifiers are shown on links, indicating that *c1* is associated to *p1*, *p2* and *p3* at different times. Aspect classes are shown as well for reference purposes.

In Figure 16, the whole *c1* object, rather than specific phases, is shown as a box. However, two phases, corresponding to moments T1 and T2 can be inferred from the phase qualifiers displayed on links.

In addition, certainty of predication may be indicated by adding a certainty marker in parenthesis after a value or reference text. Figure 15 shows an example for object *s5*.

Finally, a complete object can be depicted as a single box even if it has qualifiers by using an ellipsis character ("…") for those values or linked objects that vary along the corresponding aspect.

5.3.5 Languages

The language of an object diagram may be assumed to be the default or, alternatively, explicitly established by using a language declaration. Figure 17 shows an example.

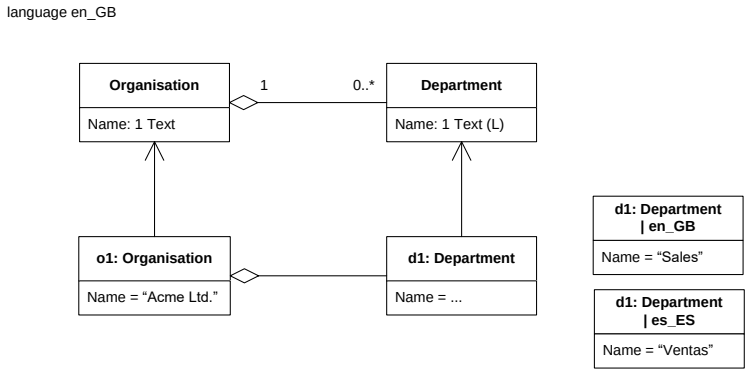

Figure 17. Sample object diagram showing two translations for object *d1*. The lines starting with “|” are translation qualifiers.

Translations are object versions that pertain to a specific language (see *Multilingualism*, p. 40). Explicit notation may be used to provide translations for an object. This is achieved by using the object notation described in *Objects and Values*, p. 51, but adding a *translation qualifier* under the object identifier, as in *d1* in the figure. The translation qualifier is composed of the pipe symbol (“|”) followed by a language name.

5.4 Specification Tables

A specification table shows the full details of an element in a model (a class, an attribute, an association, an enumerated type, etc.), including the details of strongly related elements. For example, the specification table of a given class shows the name of the class, whether it is abstract or not, the names of its generalized class (if any), the specialization discriminant, if any, the names of its specialized classes (if any), its definition, a full specification of each of its owned properties, a full specification of each of its owned attributes, and a full specification of each of its owned semi-associations. In contrast to class and object diagrams, specification tables aim to display all the available information in the model about a given element, making readability a second priority. In this way, specification tables and diagrams are complementary forms of expression of the underlying model.

In specification tables, no field must be left blank; those fields for which a value is not defined (e.g. “Discriminant” for a class with no specialized classes) must be filled in with the expression “n/a”. Lists must be separated by a comma character (“,”).

Specification tables may adopt any appearance and layout as long as they fulfil the above requirements.

5.5 Informal Variations

Given the requirement for simplicity and affordability to non-experts in information technologies, some informal variations of ConML are allowed in certain scenarios. Informal variations involve the following:

- Using localized versions of keywords and other model elements.
- Using masculine and feminine forms of words interchangeably in model element names and other identifiers.

Typical scenarios for the usage of informal variants include informal sketching on a whiteboard or paper, composing teaching or educational materials, or exemplifying language usage.

In any case, localized variants of ConML are completely informal. This means that they cannot be claimed to comply with the technical specification described in this document, and support from tools and other formalisms cannot be guaranteed for them.

The following sections describe the details of informal variations.

5.5.1 Localization

Localized variants of the ConML notation can be informally used to suit non-English settings. For example, a project whose usual language is French may want to use the French variant of the ConML notation.

A localized variant of the ConML notation includes the following:

- Translated equivalents of **enumeration items** in the metamodel.
- Translated equivalents of **keywords** in the notation.

Localized variants of the ConML notation should not be mixed: a model should use one and only one localized variant. For example, avoid mixing keywords in one language and enumerated items or keywords in a different language.

Localization should not be confused with multilingualism (see *Multilingualism*, p. 40). Multilingualism provides formal and fully supported features to work with multiple languages per model, whereas localization allows for informal variants of the ConML notation. When working with a model in a particular language through multilingualism, it is common to employ the corresponding localized variant of the ConML notation, although this is not compulsory.

The following sections describe some localized variants of the notation. Additional localized variants may be added by extending these tables.

5.5.1.1 Enumeration Items

	Localized Variants		
English	**French**	**Galician**	**Spanish**
BaseDataType			
Boolean	Booléen	Booleano	Booleano
Number	Nombre	Número	Número
Time	Temps	Tempo	Tiempo
Text	Texte	Texto	Texto
Data	Données	Datos	Datos

5.5.1.2 Keywords

	Localized Variants		
English	**French**	**Galician**	**Spanish**
con	con	con	con
enum	énum	enum	enum
false	non faux fausse	non falso falsa	no falso falsa
language	lange	linguaxe	lenguaje
null	nul	nulo	nulo
package	paquet	paquete	paquete
ref	ref	ref	ref
sha	par	com	com
true	oui vrai vraie	si verdadeiro verdadeira	sí verdadero verdadera
unknown	inconnu inconnue	descoñecido descoñecida	desconocido desconocida

5.5.2 Gender Alternation

When using a localized variant in a language that has grammatical gender, masculine or feminine forms of words may be used in model element names and other identifiers as necessary, interchangeably depending on the context. For example, consider a *Person* class having an *IsMarried* attribute, as well as an *Informant* subclass of *Person*. When this model is expressed in Spanish through multilingualism, *Person* becomes *Persona*, *Informant* becomes *Informante*, and *IsMarried* becomes *EstáCasada*. Since *Persona* in Spanish is feminine, *EstáCasada* adopts a feminine form as well. However, *Informante* is masculine, so when the attribute is mentioned in the context of this subclass, it becomes *EstáCasado*. This gender alternation does not mean that the attribute is redefined or changed in any manner; rather, it constitutes an informal adjustment to make model expressions easier to read. Formal versions of the model would need to adopt either the masculine or feminine form of the attribute name and use it throughout.

Acknowledgments

Very special thanks to Charlotte Hug and Patricia Martín-Rodilla for their contributions to ConML.

Special thanks to Alba Llavina, Brian Henderson-Sellers, Chris Partridge, Lucía Prieto, Martín Pereira-Fariña, Patricia Martín-Rodilla, Sergio España and Yaiza Mouttet for their valuable suggestions. Thanks also to the MIRFOL team.

References

[1] Gonzalez-Perez, C., 2012. A Conceptual Modelling Language for the Humanities and Social Sciences, in *Sixth International Conference on Research Challenges in Information Science (RCIS), 2012*, C. Rolland, J. Castro, and O. Pastor (eds.). IEEE Computer Society. 396-401.

[2] ISO/IEC, 2012. *Information technology -- Object Management Group Unified Modeling Language (OMG UML)* Part 1: Infrastructure. ISO/IEC 19505-1:2012.

[3] ISO/IEC, 2012. *Information technology -- Object Management Group Unified Modeling Language (OMG UML)* Part 2: Superstructure. ISO/IEC 19505-2:2012.

[4] Wagemans, J., 2019. Four Basic Argument Forms. *Research in Language*. **17**(1): 57-69.