Layer0 specification

Hannu Niemistö Antti Villberg

September 1, 2011

Contents

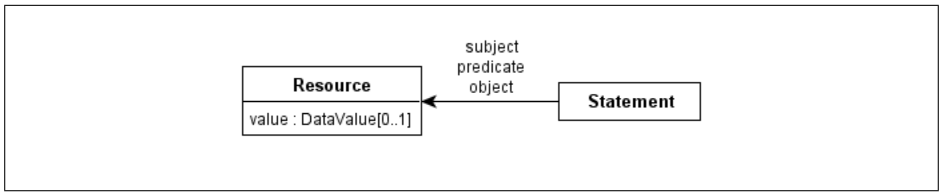


Figure 1: Metamodel of semantic graphs

1 Semantic graphs

Semantic graphs are built from *resources* and *statements*. Resources are either *literals* or references to entities such as abstract mathematical concepts or concrete objects in physical world. If a resource is a literal, it has a *value*. Statements are atomic assertations about the entities the resources refer to. Each statement is composed of three resources: *subject*, *predicate* and *object*. (Figure 1)

1.1 Graphical notation of semantic graphs

The semantic graphs are best understood graphically. Resources are drawn as nodes and statements are directed edges. Each edge starts from the subject of the statement and ends to the object of the statement. The label of the edge is the predicate of the statement.

If a resource is a literal, its value is written as a label of the node. Otherwise the label of the node is the name of the resource if the name exist.

These notations are enough to draw any semantic graph, but some additional notations makes it easier to draw big graphs. The type of the resource can be indicated by adding it to the label of the node after a colon (`:`). (Thus the colon abbreviates a statement having **InstanceOf** as a predicate.)

Another abbreviation is for the case where a resource has a statement whose object is a literal. Then the predicate and the value of the object can be added to the label of the node as

```
Predicate = Value
```

An example where all these notations are used is in Figure 2.

1.2 Textual notation of semantic graphs

Ontologies are written on Simantics platform using a textual notation describing semantic graphs. A resource can be referred in the notation in many different ways:

- `<http://www.simantics.org/Layer0/Entity>` refers to a resource with given URI.

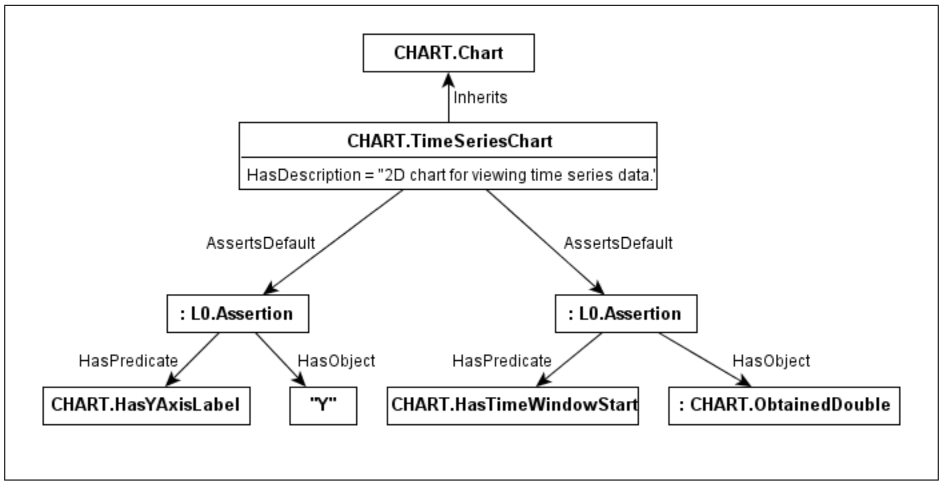


Figure 2: An example of semantic graph

- `L0` refers to a resource that has a local name `L0`. The resource does not necessarily have an URI, but it can be given an URI by writing `L0 = <http://www.simantics.org/Layer0>`.
- `L0.Entity` refers to the same resource as in the first example assuming that `L0` is given an URI as in the second example.
- `1`, `"Hello World!"`, `false` and `[1,2,3]` refer to literals with given values. The syntax of the values follows Databoard specification.

A statement is written as

```
subject predicate object
```

and each statement is written into a separate line. Example Figure 2 can be written in textual notation as

```
CHART.TimeSeriesChart L0.Inherits CHART.Chart
CHART.TimeSeriesChart L0.HasDescription "2D chart for viewing time series data."
CHART.TimeSeriesChart L0.AssertsDefault assertion1
CHART.TimeSeriesChart L0.AssertsDefault assertion2
assertion1 L0.InstanceOf L0.Assertion
assertion1 L0.HasPredicate CHART.HasYAxisLabel
assertion1 L0.HasObject "Y"
assertion2 L0.InstanceOf L0.Assertion
assertion2 L0.HasPredicate CHART.HasTimeWindowStart
assertion2 L0.HasObject timeWindowStart
timeWindowStart L0.InstanceOf CHART.ObtainedDouble
```

We can avoid repeating the subject of each statement by giving predicate–object pairs either in the same line as the subject or indented after the subject. The following definitions describe the same graph as above:

```
CHART.TimeSeriesChart L0.Inherits CHART.Chart
    L0.HasDescription "2D chart for viewing time series data."
    L0.AssertsDefault assertion1
    L0.AssertsDefault assertion2
assertion1 L0.InstanceOf L0.Assertion
    L0.HasPredicate CHARY.HasYAxisLabel
    L0.HasObject "Y"
assertion2 L0.InstanceOf L0.Assertion
    L0.HasPredicate CHART.HasTimeWindowStart
    L0.HasObject timeWindowStart
timeWindowStart L0.InstanceOf CHART.ObtainedDouble
```

In the similar way, repeating a predicate can be avoided by giving multiple objects indented after the predicate. The definitions can be chained by giving statements about an object indented after the object:

```
CHART.TimeSeriesChart L0.Inherits CHART.Chart
    L0.HasDescription "2D chart for viewing time series data."
    L0.AssertsDefault
        assertion1 L0.InstanceOf L0.Assertion
            L0.HasPredicate CHARY.HasYAxisLabel
            L0.HasObject "Y"
        assertion2 L0.InstanceOf L0.Assertion
            L0.HasPredicate CHART.HasTimeWindowStart
            L0.HasObject timeWindowStart L0.InstanceOf CHART.ObtainedDouble
```

If a resource does not have an URI and it is referred only in one place, it can be referred as _ instead of a local name. Finally, there are short hand notation for some relations: `L0.InstanceOf` is written as `:`, `L0.Inherits` is written as `<T` and `L0.SubrelationOf` is written as `<R`.

```
CHART.TimeSeriesChart <T CHART.Chart
    L0.HasDescription "2D chart for viewing time series data."
    L0.AssertsDefault
        _ : L0.Assertion
            L0.HasPredicate CHARY.HasYAxisLabel
            L0.HasObject "Y"
        _ : L0.Assertion
            L0.HasPredicate CHART.HasTimeWindowStart
            L0.HasObject _ : CHART.ObtainedDouble
```

The example can be shortened further by using templates as described in subsection 2.3:

```
CHART.TimeSeriesChart <T CHART.Chart
    L0.HasDescription "2D chart for viewing time series data."
    @L0.assertDefault CHARY.HasYAxisLabel "Y"
    @L0.assertDefault CHART.HasTimeWindowStart
        _ : CHART.ObtainedDouble
```

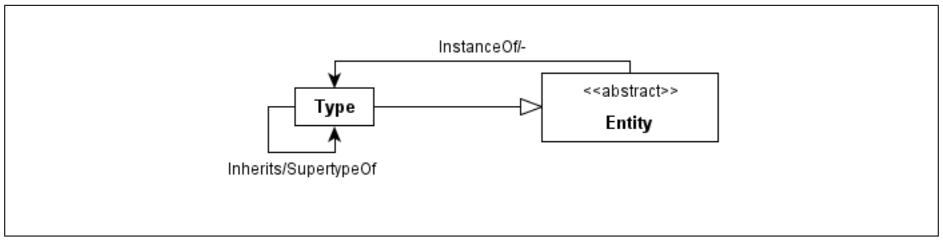


Figure 3: Types

2 Concepts

All names in this and the following sections written in boldface font are concepts defined in the namespace `http://www.simantics.org/Layer0-1.0`.

2.1 Types

Types categorize resources and give them their semantics. Relation **InstanceOf** gives a resource its type. Relation **Inherits** between types tells that all instances of the first type are also instances of the second. **Entity** is the supertype of all other types. Every type is an instance of **Type**. (Figure 3)

All types but **Entity** must inherit another type and the inheritance relation must be acyclic. This implies that every type inherits **Entity** directly or indirectly. Every resource must have at least one type. Thus every resource is an instance of **Entity**.

There are three rules that determine when a resource a is an instance of a type T.

- if there exists a statement $(a, \textbf{InstanceOf}, T)$
- if a is an instance of T' and there exists a statement $(T', \textbf{Inherits}, T)$
- if a' is an instance of T and there exists a statement $(a, \textbf{Inherits}, a')$ or $(a, \textbf{SubrelationOf}, a')$.

2.2 Relations

The predicate of each statement is a *relation* and we say that the statement belongs to that relation. We can also think that the interpretation of each relation resource is a binary relation that is formed by taking all subject–object pairs from the statements belonging to the relation. All relations have type **Relation**.

A relation may be a *subrelation* of other relation. This is indicated by relation **SubrelationOf**. All statements belonging to a relation also belong to its *superrelations*.

A relation may also have an *inverse relation*. Relation **InverseOf** tells that two relations are inverses of each other. If R has an inverse relation I, then for each statement (a, R, b) there must also exist a statement (b, I, a).

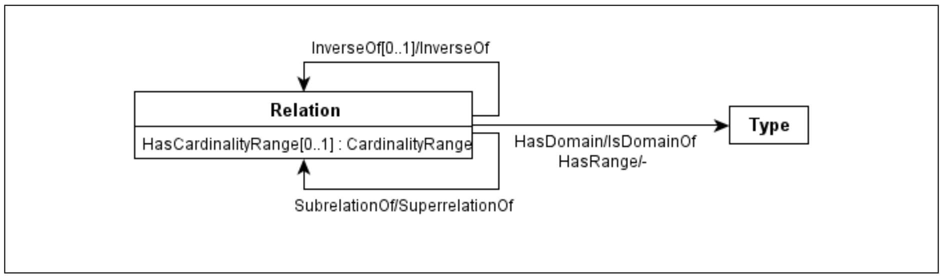

InverseOf[0..1]/InverseOf
Relation
HasCardinalityRange[0..1] : CardinalityRange
Type
HasDomain/IsDomainOf
HasRange/-
SubrelationOf/SuperrelationOf

Figure 4: Relations

Each relation has a *domain* (*range*). It is the set of resources that are valid subjects (objects) of the statements belonging to the relation. The domain (range) is the set of all instances of all types given by **HasDomain** (**HasRange**) in the relation. If no types are given, the domain is the intersection of domains (ranges) of the immediate superrelations of the relation. It is invalid to give a relation domain or range that is larger than the domain or range of its superrelation.

Finally, a relation has also a *cardinality range*. It is the range of integers given by **HasCardinalityRange**. If it is not specified in the relation itself, it is the intersection of the cardinality ranges of the superrelations. It is invalid to specify a cardinality range that is larger than the cardinality of any superrelation of a relation. The cardinality range gives lower and upper bounds for the cardinality of the relation on each resource in the domain of the relation. The cardinality of a relation R on resource a is the number of statements which belong to R and have a as a subject. (Figure 4)

There are two subtypes of **Relation** with fixed cardinality ranges: **FunctionalRelation** is the type of relations with cardinality zero or one and **TotalFunction** is the type of relations with cardinality one.

2.3 Assertions

Assertions are a mechanism to describe facts about resources in their types. An assertion is added to a type with relations **Asserts** and **AssertsDefault**. The first relation asserts an irrevocable fact that is always true for the instances of the type (such as *birds have wings*) and the second relation asserts facts that are usually true for their instances but can be revoked in their instances (such as *birds fly*).

The assertion itself is an instance of **Assertion** and consists of a predicate (**HasPredicate**) that must be a relation and an object (**HasObject**). It adds one statement with those predicate and object to every instance of the type. (Figure 5)

Assertions can be specified in graph -files using the templates **assert** and **assertDefault**:

```
Animals.Bird <T Animals.Animal
```

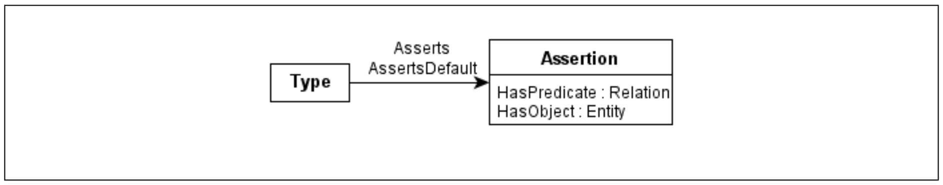


Figure 5: Assertions

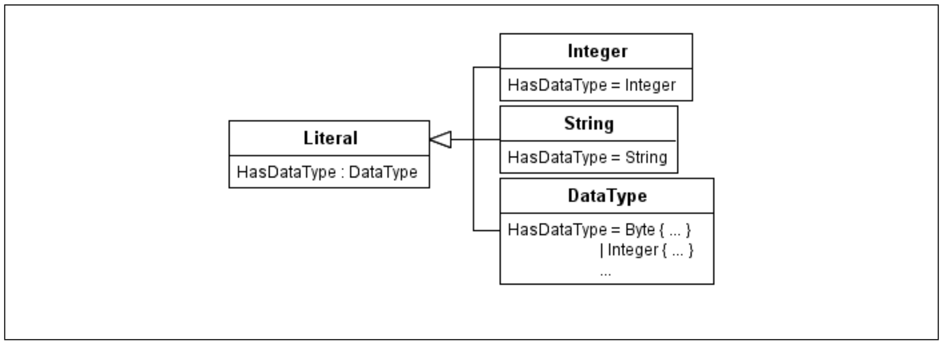


Figure 6: Literals

```
    @L0.assert Animals.HasBodyPart Animals.Wings
    @L0.assertDefault Animals.CanFly true
```

2.4 Literals

Literals represent numerical and other values in semantic graphs. They are resources that are instances of type **Literal**. Each literal must specify the type of the literal using **HasDataType** and must contain a value of that type. (Figure 6)

Layer0 ontology defines many Literal types that assert their data types, for example:

```
L0.String <T L0.Literal
    @L0.assert L0.HasDataType $String
```

The graph compiler knows the literal types for primitive types and their arrays and infers the type of the literal automatically if not given. For other literal types, the type must be given in the normal way

```
"http://en.wikipedia.org/wiki/Literal" : L0.URL
```

The following is the list of all literal types and the corresponding data types defined on Layer0:

literal type	data type
Boolean	Boolean
Byte	Byte
Integer	Integer
Long	Integer
Float	Float
Double	Double
String	String
BooleanArray	Boolean[]
ByteArray	Byte[]
IntegerArray	Integer[]
LongArray	Integer[]
FloatArray	Float[]
DoubleArray	Double[]
StringArray	String[]
Variant	Variant
DataType	Byte { ... } | Integer { ... } | ...
CardinalityRange	{ min : Optional(Integer), max : Optional(Integer) }
URI	String
Graph	{ resourceCount : Integer, identities : { resource : Integer, definition : | Root { name : String, type : String } | External { parent : Integer, name : String } | Optional { parent : Integer, name : String } | Internal { parent : Integer, name : String } }[], statements : Integer[], values : { resource : Integer, value : Byte[] }[] }

There are also some literal constants defined in Layer0:

resource	type	value
True	**Boolean**	true
False	**Boolean**	false
Cardinality0	**CardinalityRange**	{ max=0 }
Cardinality1	**CardinalityRange**	{ min=1, max=1 }
Cardinality2	**CardinalityRange**	{ min=2, max=2 }
CardinalityAtLeast1	**CardinalityRange**	{ max=1 }
CardinalityAtMost1	**CardinalityRange**	{ min=1 }

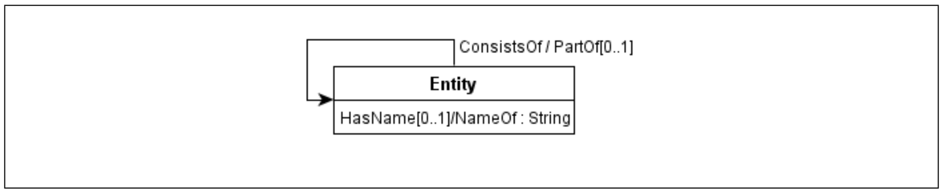


Figure 7: Unique resource identifiers

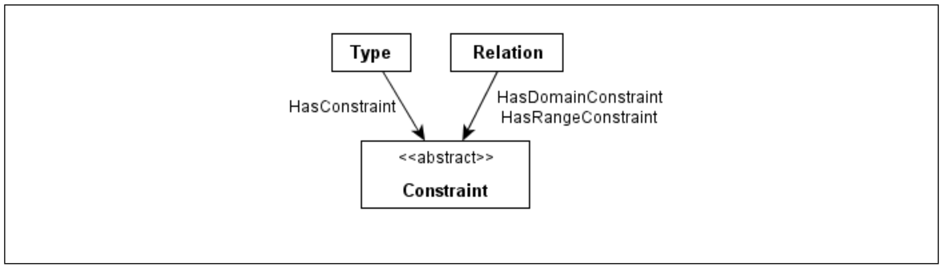


Figure 8: Constraints

2.5 Unique resource identifiers

Unique resource identifiers (URIs) are global references to resources. They are formed using relations **ConsistsOf** and **HasName** and a special resource **Root** that is the root of the semantic database. (Figure 7)

The URI of **Root** is `http:/`. For all resources p and c such that there is a statement $(p, \mathbf{ConsistsOf}, c)$, if p has URI $\mathrm{URI}(p)$, then the URI of c is $\mathrm{URI}(c) = \mathrm{URI}(p) + "/" + \mathrm{escape}(\mathrm{name}(c))$, where the function name gives the name (**HasName**) of a resource and the function escape escapes it as described in [1].

2.6 Constraints

Domain, range and cardinality range restrictions of the relations allow ontology developers to specify simple commonly occuring validity constraints in the ontologies. They have however very weak expressive power incapable of describing any complex modelling rules.

Constraints may describe arbitrarily complex rules about resources. A constraint can be attached to a type with **HasConstraint** when all instances of the type must satisfy the constraint. It can also be added to relations with **HasDomainConstraint** or **HasRangeConstraint** when the subjects or objects of the statements beloning to the relation must satisfy the constraint. (Figure 8)

RelationConstraint is one of the constraint types. It restricts how the relation specified with **ConcernsRelation** can be used in the context of the constraint. It can restrict the range and cardinality range of the relation and add domain and range constraints. (Figure 9)

A typical situation where **RelationConstraint** is needed is when a supertype of a type has a relation but in the subtype the relation must be used in

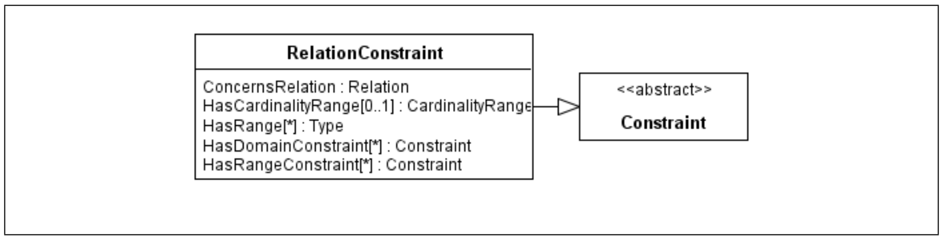

RelationConstraint
ConcernsRelation : Relation
HasCardinalityRange[0..1] : CardinalityRange
HasRange[*] : Type
HasDomainConstraint[*] : Constraint
HasRangeConstraint[*] : Constraint
<<abstract>>
Constraint

Figure 9: Relation constraints

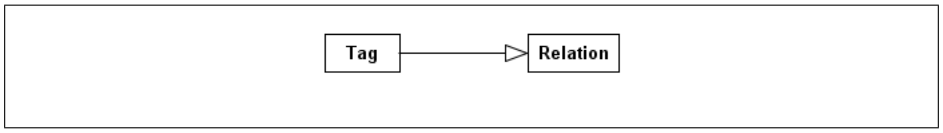


Figure 10: Tags

a certain way. For example, **HasName** has cardinality range 0..1 in **Entity**. Using **RelationConstraint** the property can be made mandatory:

```
L0.Library <T L0.Entity
    L0.HasConstraint _ : L0.RelationConstraint
        L0.ConcernsRelation L0.HasName
        L0.HasCardinalityRange L0.Cardinality1
```

In many cases, the same modeling restriction can be modelled either using domain or range restrictions or using constraints. Domain and range restrictions should always be preferred because they are easier to analyze for example for code generation, and because the complexity of their validation is low, they can be enforced in write transactions.

2.7 Tags

Tags are relations that are used to encode sets of resources. All statements using them as a predicates, must have the same subject and object. (Figure 10)

Layer0 ontology defines the following tags:

tag	domain	description
Abstract	**Type**, **Relation**	Prevents the use of type or relation directly in the graph. The concept is meant to be inherited.
Final	**Type**, **Relation**	Prevents the inheritance of type or relation.
Enumeration	**Type**	Marks an enumeration type. All instances of the type must be children (**ConsistsOf**) of the type.
Deprecated	**Entity**	Marks a concept that is deprecated and should not be used anymore.
Immutable	**Entity**	Prevents any modification to the resource after the write transaction that added **Immutable** tag.
SharedRange	**Relation**	Indicates a relation whose objects can be shared by multiple contexts [needs to be clarified].

2.8 Relation hierarchy

Layer0 defines a hierarchy of base relations. They are abstract, i.e. they are not meant to be used directly but inherited. The first two relations do not have inverses, the inverses of the last two are functional relations. Each relation is a subrelation of the previous relation in the table.

relation	inverse	description
IsWeaklyRelatedTo	-	Base relation of all relations.
IsRelatedTo	-	A relation that is used in garbage collection and exporting subgraphs.
DependsOn	**IsDependencyOf**	A relation that is used to propagate change events. [is this needed anymore?]
IsComposedOf	**IsOwnedBy**	If a resource is deleted, all resources linked to it with **IsComposedOf** are deleted.
HasProperty	**PropertyOf**	Links a literal with a resource. The range or the relation is **Literal** [or should it be **Property**?].

2.9 Annotations

Annotations add information to resources in human understandable form. The following annotation properties are defined in Layer0. They are all string-valued:

annotation	description
HasLabel	Gives a descriptive label of the resource. If given it should be used instead of name in user interfaces.
HasDescription	Describes the resource. Long descriptions are formatted using Mediawiki-markup.
HasComment	Adds a comment about the resource. A comment may be for example about a change made to resource or a found problem in its defintion.

2.10 Organization of data

All *referrable resources* (those having a URI) form a tree with **Root** as a root and **ConsistsOf** relation linking parents to children (subsection 2.5). Any **Entity** may have children. Because sometimes we just want to group resources into namespaces that by themselves do not have any special meaning, we have a type **Library** for this purpose.

An important goal of the data organization is to be able to move parts of the data between different semantic graphs. When moving the data, it is necessary to identify, which parts of the graph belong together so that data that is moved is consistent and useful in the target graph.

We associate movable parts of the graph with referrable resources with type **Context**. Let C be a context (resource). Resources that *belong to context* C are determined by the the following rules:

1. C belongs to C.
2. If r' belongs to context C, r' is referrable and there is a statement $(r', \textbf{ConsistsOf}, r)$, then r belongs to context C.
3. If r' belongs to context C, there is a statement $(r', \textbf{IsRelatedTo}, r)$ and r is not referrable, then r belongs to context C.

There an another way to define “belonging to the context”: A *strong path* from a to b is a sequence of referrable resources $p_0, \ldots, p_n$ such that $p_0 = a$, $p_n = b$ and for all $0 \leq i < n$, there is a statement $(p_i, \textbf{ConsistsOf}, p_{i+1})$. A *weak path* from a to b is a sequence of resources $p_0, \ldots, p_n$ such that $p_0 = a$, $p_n = b$ and for all $0 \leq i < n$, there is a statement $(p_i, \textbf{IsRelatedTo}, p_{i+1})$ and p_{i+1} is *not* referrable. The first resource of a weak path may be referrable. Now a resource r belongs to C if there is a strong path from C to some resource r' and a weak path from r' to r.

In order to make contexts exportable and importable separately, we need to restrict the organization of the data in the following way:

> If a non-referrable resource r belongs to context C and there is a weak path from a referrable resource r' to r, then r' must belong to context C.

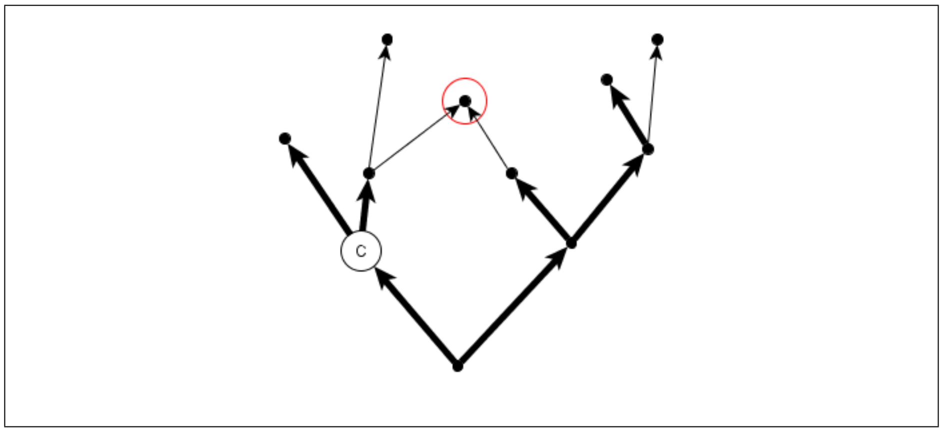


Figure 11: An example of an invalid context. Thick edges are **ConsistsOf** statements and thin lines other **IsRelatedTo** statements. A context is marked with C and a resource violating the restriction is marked with a red circle.

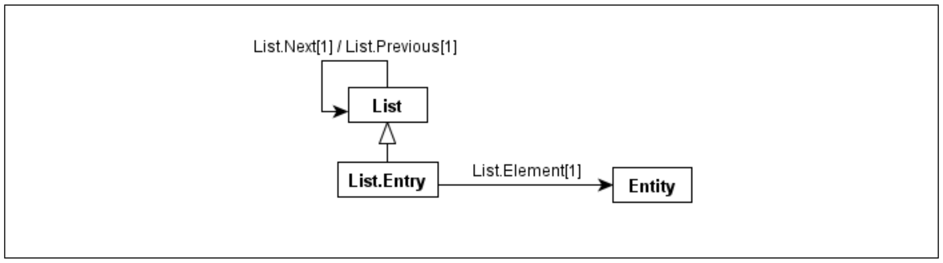


Figure 12: Lists

A situation this restriction forbids is visualized in Figure 11. This restriction implies that if a resource belongs to two contexts then one of the contexts must belong to the other.

Ontology is a special kind of context containing shared concepts that are meant to be used in many different models and other ontologies.

Sometimes a certain resource is needed as a child of an other resource, but **ConsistsOf** cannot be used, because the resource already has a parent and **PartOf** is functional. In these cases, the resource can be linked using **IsLinkedTo** whose inverse is not functional.

2.11 Lists

List is a structure that can be used to encode ordered data. The list consists **List.Entry** resources linked into a cycle with **List.Next**. **List** itself is one element in the cycle. Elements of the list are attached to entries with **List.Element**.

A list can be written in graph notation as

```
@L0.list
```

```
    element1
    ...
    elementN
```

It generates statements:

```
l : L0.List
    L0.ListEntry.Next entry1
entry1 : L0.ListEntry
    L0.ListEntry.Element element1
    L0.ListEntry.Next entry2
...
entryN : L0.ListEntry
    L0.ListEntry.Element elementN
    L0.ListEntry.Next l
```

An empty list is thus a list that refers to itself with **ListEntry.Next**:

```
emptyList : L0.List
    L0.ListEntry.Next emptyList
```

2.12 Templates

Templates are parametrized semantic graphs. A template has type **Template** and it has two properties. **HasTemplateParameters** is a string array that gives the parameter names of the template. **HasTemplate** points to a **Graph**.

The following templates are defined on Layer0:

template	parameters	description
assert	Relation,Entity	
assertDefault	Relation,Entity	
tag	Tag	Tags a resource.
defTag		Defines a tag.
symmetric		States that a relation is its own inverse.
property		deprecated
singleProperty		deprecated
singlePropertyDefault		deprecated
optionalProperty		deprecated
new		Marks a resource and all its children as a new resource that is written to a transferrable graph as an internal resource.
list	Entity*	Creates a **List** -structure of given elements.
loadBytes	String	Creates a **ByteArray** with data loaded from a file given as a parameter.
loadString	String	Creates a **String** with data loaded from a file given as a parameter.
loadDataValue	String	Creates a **Variant** with data loaded from a file given as a parameter

2.13 Miscellaneous concepts

Property is a type of slightly larger set of resource than **Literal**. Also enumerations are properties althought they are not literals. [Is this type necessary?]

Value is a type of SCL values. [Need to be fully specified.]

3 Conventions

When designing a new ontology from scratch the following naming rules should be followed:

- The names of the types and relations are capitalized and multiword concepts are written in CamelCase.
- The names of templates and SCL values are written in lower case.

When transforming an existing data to ontology, it may be better to follow the conventions of the original data.

All shared concepts should be defined in ontologies. For small ontologies, all types can be defined directly in the namespace of the ontology itself. Larger ontologies are often generated from existing data and the original organization of the concepts can be mimiced in the ontology using libraries.

If a relation has a single domain type, the relation should be defined in the namespace of that type:

```
VP.BrowseContext <T L0.Entity
VP.BrowseContext.Includes <R L0.IsRelatedTo
    L0.HasDomain VP.BrowseContext
    L0.HasRange VP.BrowseContext
    L0.InverseOf VP.BrowseContext.IsIncludedIn <R L0.IsRelatedTo
```

In this way, the same relation name can be used with different types and the temptation to use the same relation for different purposes in different types is reduced.

An obvious exception of the rule is the situation where a relation is defined in a different ontology from its domain type. It is very bad practice to define a relation with minimum cardinality greater than zero in a different ontology than its domain type.

If a relation has an inverse, but it is too unimportant to be specially named, it should be defined in the namespace of the relation and named as **Inverse**. Graph compiler does this automatically when the relation must have an inverse, but it is not explicitly specified.

Almost all relations should have both domain and range specified. Only exceptions are relations that really are meaningful in all resources (such as **HasDescription**).

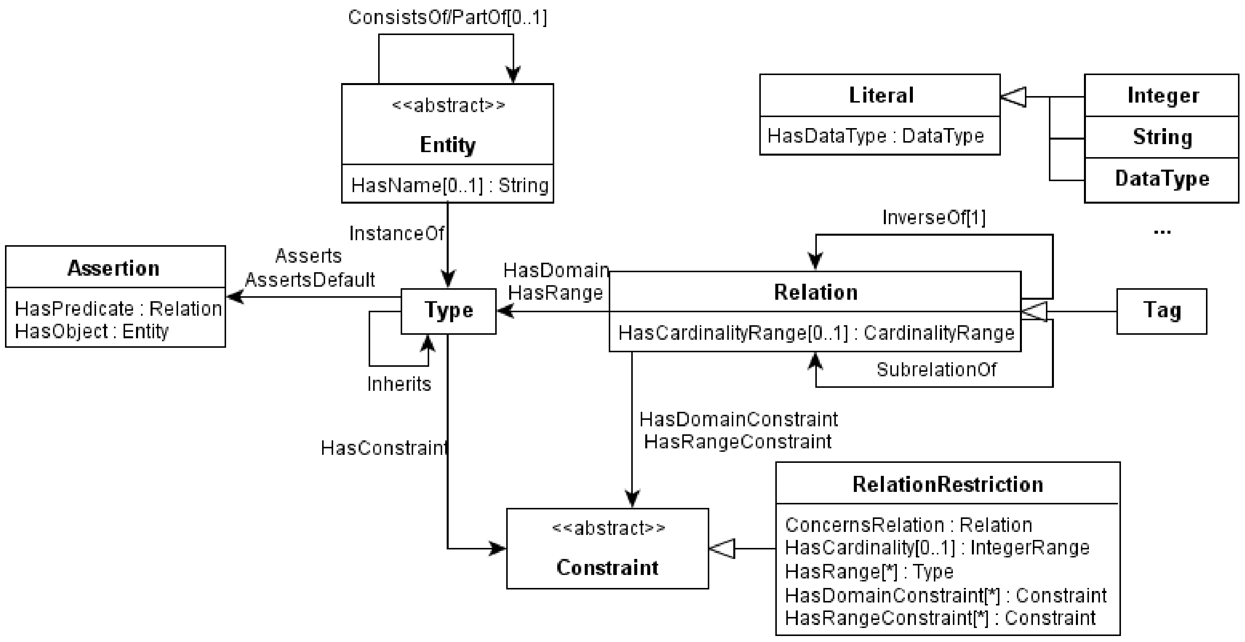
Domain and range restrictions should be preferred to the constraints. A constraint is however needed in the case, where a subtype of a type wants to restrict the cardinality or range of a relation that has the supertype in its domain.

A relation whose minimum cardinality is greater than zero should not have domain constraints, but the constraints should be specified in the domain types of the relation.

References

[1] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.

A Summary of concepts



B Summary of validity rules

This section lists the vality rules for semantic graphs. All other rules of the Layer0 ontology are described in the ontology itself as cardinality, domain and range restrictions and specified in the diagrams of this document (for example the rule *"Each resource has at most one URI"* is enforced by restricting the cardinality of **PartOf** and therefore the rule is not mentioned here). Most of the rules could be written to the ontology as constraints, but then the invalidity of the graph could prevent validator from finding them.

B.1 Basic rules

Rel1 *Predicates are relations.* If (a, b, c) is a statement, b must either has **SubrelationOf** -statement or be equal to **IsWeaklyRelatedTo**.

Type1 *Every resource has a type.* If a is a resource, it must have a statement belonging to one of the relation **InstanceOf**, **Inherits** or **SuperrelationOf**.

Lit1 *Literals have values.* If a is an instance of **Literal**, a must have a value.

Lit2 *Only literals have values.* If a has a value, a must be an instance of **Literal**.

Lit3 *Values must respect data types.* If a has a value, the value must be a valid instance of the data type given by **HasDataType** in a.

Str1 *URIs must be unique.* If a and b have the same parent (**PartOf**), and equal names (**HasName**), then they must be the same resource ($a = b$).

Str2 *ConsistsOf is acyclic.* There must be no cycles in the relation **ConsistsOf**.

Str3 *Contexts must not intersect.* See subsection 2.10.

B.2 Inheritance hierarchy

Hier1 *Inheritance is acyclic.* There must be no cycles in the relation **Inherits**.

Hier2 *SubrelationOf is acyclic.* There must be no cycles in the relation **SubrelationOf**.

Hier3 *Entity is the supertype of all types.* If resource is an instance of **Type**, it must either inherit some other type or be **Entity**.

Hier4 *IsWeaklyRelatedTo is the superrelation of all relations.* If resource is an instance of **Relation**, it must either be a subrelation of some other relation or be **IsWeaklyRelatedTo**.

Hier5 *The domain of a relation must be contained in the domains of its superrelations.* If there is a statement $(R_0, \mathbf{HasDomain}, T)$, then for each sequence of relations $R_0, \ldots, R_n$ such that for each $0 \leq i < n$ we have $(R_i, \mathbf{SubrelationOf}, R_{i+1})$ and for each $0 < i < n$ we don't have any statement of the form $(R_i, \mathbf{HasDomain}, T')$ and there are statements of the form $(R_n, \mathbf{HasDomain}, T')$, then there must be T' such that T inherits or is equal to T' and $(R_n, \mathbf{HasDomain}, T')$.

Hier6 *The range of a relation must be contained in the ranges of its superrelations.* As above.

Hier7 *The cardinality range of a relation must be contained in the cardinality ranges of its superrelations.* If there is a statement $(R_0, \mathbf{HasCardinality}, h)$, then for each sequence of relations $R_0, \ldots, R_n$ such that for each $0 \leq i < n$ we have $(R_i, \mathbf{SubrelationOf}, R_{i+1})$ and for each $0 < i < n$ we don't have any statement of the form $(R_i, \mathbf{HasCardinality}, h')$ and there is h' such that $(R_n, \mathbf{HasDomain}, h')$, then h must be included in the range h'.

B.3 Inverses

Inv1 *Inverse relations must have inverse statements.* If there are statements $(R, \textbf{InverseOf}, I)$ and (a, R, b), there must also be a statement (b, I, a).

Inv2 *Superrelations of inverse relations must be compatible.* If there are statements $(R, \textbf{InverseOf}, I)$, $(R, \textbf{SubrelationOf}, R')$ and $(R', \textbf{InverseOf}, I')$, then there must be also a statement $(I, \textbf{SubrelationOf}, I')$.

Inv3 *Domains and ranges of inverse relations must be compatible.* If there are statements $(R, \textbf{InverseOf}, I)$ and $(R, \textbf{HasDomain}, T)$, there must also be a statement $(I, \textbf{HasRange}, T)$.

Inv4 *Domains and ranges of inverse relations must be compatible.* If there are statements $(R, \textbf{InverseOf}, I)$ and $(R, \textbf{HasRange}, T)$, there must also be a statement $(I, \textbf{HasDomain}, T)$.

B.4 Modelled restrictions

Res1 *Subject must belong to the domain of the predicate.* If (a, b, c) is a statement, then a must be in the domain of the relation b. If b has **HasDomain** statements, then the domain is the union of all instances of the type given by the statements. Otherwise the domain is the intersection of the domains of the superrelations of b.

Res2 *Object must belong to the range of the predicate.* As above.

Res3 *Cardinality must be in the cardinality range.* If a is a resource and b is a relation such that a is in its domain, then the number of objects c such that (a, b, c) is a statement, must be in the cardinality range of b. If b has property **HasCardinalityRange**, the cardinality range is given by the property. Otherwise the cardinality range is the intersection of the cardinality ranges of the superrelations of b.

Res4 *An instances of a type must satisfy the constraints of the type.*

Res5 *A subject of a statement must satisfy the domain constraints of the predicate of the statement.*

Res6 *An object of a statement must satisfy the range constraints of the predicate of the statement.*

B.5 Tags

Tag1 *Tags encode unary relations.* If R is an instance of **Tag** and (a, R, b) is a statement, then $a = b$.

Abs1 *Abstract types must not be instantiated directly.* If a type T has tag **Abstract**, then there must not be resources a such that $(a, \textbf{InstanceOf}, T)$.

Abs2 *Abstract relations must not be used directly.* If a relation R has tag **Abstract**, then there must not be resources a and b such that (a, R, b).

Final1 *Final types must not be inherited.* If a type T has tag **Final**, then there must not be resources T' such that $(T', \textbf{Inherits}, T)$.

Final2 *Final relations must not be used as superrelations.* If a relation R has tag **Final**, then there must not be resources R' such that $(R', \textbf{SubrelationOf}, R)$.

C Query semantics

Assume that the raw statements of the semantic database are given as a relation **Stats**. The following rules define when one type inherits other type ($\leq_T$), when one relation is subrelation of other relation ($\leq_R$), when a resource is an instance of a type (:) and when there is a statement $\langle a, R, b \rangle$.

$$\frac{}{A \le_T A} \qquad \frac{(A, \textbf{Inherits}, B) \in \textbf{Stats}}{A \le_T B} \qquad \frac{A \le_T B \qquad B \le_T C}{A \le_T C}$$

$$\frac{}{A \le_R A} \qquad \frac{(A, \textbf{SubrelationOf}, B) \in \textbf{Stats}}{A \le_R B} \qquad \frac{A \le_R B \qquad B \le_R C}{A \le_R C}$$

$$\frac{(a, \textbf{InstanceOf}, T) \in \textbf{Stats}}{a \ : \ T} \qquad \frac{a \ : \ T' \qquad T' \le_T T}{a \ : \ T} \qquad \frac{a \le_T a' \qquad a' \ : \ T}{a \ : \ T}$$

$$\frac{a \le_R a' \qquad a' \ : \ T}{a \ : \ T}$$

$$\frac{(a, R, b) \in \textbf{Stats}}{\langle a, R, b \rangle} \qquad \frac{\langle a, R', b \rangle \qquad R' \le_R R}{\langle a, R, b \rangle} \qquad \frac{\textbf{asserts}(T, R, b) \qquad a \ : \ T}{\langle a, R, b \rangle}$$

$$\frac{\textbf{assertsDefault}(T, R, b) \qquad a \ : \ T \qquad \neg\textbf{covers}(a, R, T)}{\langle a, R, b \rangle}$$

$$\frac{\langle T, \textbf{Asserts}, A \rangle \qquad \langle A, \textbf{HasPredicate}, R \rangle \qquad \langle A, \textbf{HasObject}, b \rangle}{\textbf{asserts}(T, R, b)}$$

$$\frac{\langle T, \textbf{AssertsDefault}, A \rangle \qquad \langle A, \textbf{HasPredicate}, R \rangle \qquad \langle A, \textbf{HasObject}, b \rangle}{\textbf{assertsDefault}(T, R, b)}$$

$$\frac{(a, R', b) \in \textbf{Stats} \qquad R' \le_R R}{\textbf{covers}(a, R, T)}$$

$$\frac{a \ : \ T' \qquad T' \le_T T \qquad T' \ne T \qquad \textbf{asserts}(T, R', b) \qquad R' \le_R R}{\textbf{covers}(a, R, T)}$$

$$\frac{a \ : \ T' \qquad T' \le_T T \qquad T' \ne T \qquad \textbf{assertsDefault}(T, R', b) \qquad R' \le_R R}{\textbf{covers}(a, R, T)}$$

D Changes to Layer0 in Simantics 1.5

- **AssertsDefault** is a new concept. It replaces the previous implicit assertion with functional relations.
- **Constraint** is a new concept.
- **List** is a new concept.

- **PropertyDefinition**, **HasPropertyDefinition** and **IsPropertyDefinitionOf** are removed. They are replaced by **RelationConstraint**.
- **OrderedSet**, **HasNext**, **HasPrevious** and **HasElement** are removed. They are replaced by **List**.
- **HasResourceClass** is removed and replaced by a separate graph independent mechanism in graph compiler.
- Domain, range and cardinality ranges are specified preferrably in relations instead of current practice to specify them in types.
- **FunctionalRelation** and **TotalFunction** don't have any special semantics. They assert **HasCardinalityRange**.

E Open issues

E.1 Inverses

One technical choice made in Simantics semantic graph implementation is to index statements only by subjects and predicates, not by objects. This means, that it is possible to find efficiently the statements only with given subject or with given subject and predicate. For finding statements with a given object, relations have inverse relations: If R is a relation, (a, R, b) is a statement and I is the inverse relation of R, then (b, I, a) is the inverse statement of (a, R, b).

It is still open question, when exactly a statement has an inverse statement. The following candidate solutions are proposed:

1. Every relation has an inverse relation that has all inverse statements.
2. Some relations don't have inverse relations. This is possible only when the superrelations of the relation don't have inverse. If a relation has an inverse relation, every statement of the relation has an inverse statement. (Current solution)
   (Optionally) Resources maintain a reference count of the missing inverse statements.
3. Every relation has an inverse relation, but not all statements have inverse statements. For example statements from ontologies to models can be left out.

The first solution is conceptually the simplest one: each relation must also have an inverse relation and the graph API enforces that all statements have inverse statements. The problem with that solution is that the degree of some inverse relations is enormous, in particular for relations that are typically from models to ontologies (such as InstanceOf). It is hard to implement statement addition and removal efficiently in this case (the current implementation is based on the assumption that is always acceptable to load all statements of a single resource to memory). It can also be argued that these high degree inverse relations are never really needed and even that they might be sometimes used accidentally with catastrofical consequences.

The idea of the second solution is to define inverses only if they are really useful and there is no risk of high degree. The most obvious candidates for inverseless relations are InstanceOf, and relations having an enumeration as a range. On the other hand, inverse relation is needed for propagating modification events from leaves of the models upwards: therefore DependsOn and its subrelation HasProperty must have inverses.

This solution has some drawbacks:

- In some cases it is useful to know if a resource has inverse statements even if they are not browsed. It can be used for example to check if a resource can be safely removed (it is not referred anywhere) or migrated to a new version. A particular inverse relation, SuperrelationOf, can be used to check if the relation has subrelations. This information can be used to optimize queries. This drawback could be removed by maintaining reference counts for statements without inverses, but that complicates the implementation and API of semantic graph.

- HasProperty (or any other subrelation of DependsOn) must not be used to refer resources in ontologies.

- Data modelling is more complicated: It is hard to foresee if the inverse is needed or not when writing the ontology.

In the third solution, relations always have inverses. However, some inverse statements may be left out. If it were up to implementations to choose whether to add inverse or not, inverse relations would be totally useless, because there would never be any guarantee that inverses statements correspond to statements. A systematical way to leave inverse statements out is to specify in the resources that certain relations are not written to them. This solution doesn't have any of the previous drawbacks:

- It is possible to say in many cases that resource does not have certain inverse statements. If a certain inverse relation is disabled for the resource, it is very probable that there are inverse statements.

- HasProperty can be used. DependsOn is disabled for enumeration items and literals in ontologies.

- All relations have inverses, so ontology writer does not have to make this decision.

Also this solution has drawbacks:

- It might be hard to decide whether to disable certain relation on certain resource in the ontology.

- It might be impossible to list all relations a certain resource might be referred with. However, this should be based on using suitable superrelations.

E.2 Subrelations of ConsistsOf

In the currect specification, using **ConsistsOf** relation is the only way to create URIs for resources. There is often a need for resources that contain other resources. It is also desirable that the URIs reflect this containment. There are different solutions for this pattern:

1. The relation **ConsistsOf** is used to indicate the containment. In this way URIs are automatically generated as desired. (Current solution)
   The drawback of this solution is that because **ConsistsOf** is used to indicate containment, it cannot be used to just add "unrelated" resources into the namespace of the resource. Current solution is to separate unrelated resources by type analysis.

2. Create a containment relation that is subrelation of **ConsistsOf**. In this way containment is indicated by a dedicated relation but URIs are still automatically correct. Unrelated resources can be added into the namespace of the container by using **ConsistsOf**.

3. Use both **ConsistsOf** and a dedicated containment relation that is not a subrelation of **ConsistsOf**. The drawback of this solution is that all developers writing the models of the ontology must remember to add both statements.

4. Change URI -semantics so that also other relations may generate URIs.

E.3 Default assertions

The default assertions are the most complicated mechanism in Layer0 ontology. Its completely correct implementation is very hard to implement efficiently. It is also not needed in its full power. In particular, some way to avoid checking assertations of subrelations in covering relations should be found.

E.4 Organization of Layer0 concepts

In the section 3, we stated a convention of putting relations into the namespace of their types. This convention is not followed in Layer0 ontology but all concepts are directly under the ontology. Should this convention be followed also in Layer0 ontology (which means quite heavy refactoring).

E.5 Immutable -tag

In order to define the exact semantics of **Immutable** -tag, we need to define what we mean by a modification to a resource.

E.6 Property

Is **Property** needed? How it is related to **Enumeration**? What is the exact semantics of **HasProperty**?

E.7 Constraint types defined in Layer0

Currently there is only one **Constraint** type in Layer0. There should be one generic constraint rule based on SCL. Additionally commonly occuring constraints should have their own constraint types:

- A certain relation must not have a cycle (could this be generalized to more general cases when the transitive closure of multiple relations must have no cycles)
- Types are disjoint (no resource is allowed to instantiate more than one of the disjoint types).
- Restrict the type of the elements in a list.
- Restrict values of a literal. (see also `RequiresDataType`)

Index