

Ontology Views

1. Introduction

The ontology view mechanism allows a Description Logic (DL) ontology to be accessed and manipulated, via a simplified “view”. The view consists of a set of relatively simple “view entities” that map to more complex constructs in the underlying DL ontology. As the entities are manipulated in the view, corresponding modifications will be produced in the ontology.

The view mechanism also allows the ontology to be used in conjunction with an instance store. A set of suitable view entities is provided for maintaining instances in the instance store, and forming and executing queries over those instances.

The manner in which the entities in a particular ontology view, map to the constructs in the underlying DL ontology, is determined by a “view configuration”, specifically created for that ontology, and stored in XML format.

When an ontology is loaded into the view mechanism, and whilst it is being manipulated in the view, all of the information required to specify the latest state of the view comes from a combination of the view configuration and the DL ontology, and when instances are involved, the instance store. Hence no additional information need be maintained by the view mechanism.

The view mechanism includes a Java API that allows the client code to perform the following types of manipulation:

- Creation and maintenance of view configurations.
- Creation and maintenance of views, using specific view configurations.
- Query formation and execution.

The following GUIs are built on top of the Java API, and are designed for performing each of the above types of operation:

- **View configuration GUI:** A knowledge engineers tool that generally mirrors the view configuration part of the API.
- **Ontology editing GUI:** A tool for knowledge engineers or domain experts who are involved in ontology building, that generally mirrors the view manipulation part of the API.
- **Query GUI:** Designed to be incorporated into ontology-based applications, and therefore aimed at providing as simple an interface as possible for the end-user.

This document gives a general overview of the view mechanisms, describing the creation and manipulation of views and view configurations in general terms, without going into detail about the specifics of the API and the GUIs.

2. Basics

2.1 Overview

Node/Link Structure

A view is a node/link structure, with a set of named nodes being connected by a set of directed links. Each node represents either an ontological concept, or an instance of an ontological concept, and each link represents a specific type of relationship between two concepts. The nodes can be divided into two main categories:

- **Abstract-nodes:** Representing ontological concepts.
- **Concrete-nodes:** Representing stored instances, or queries to retrieve sets of stored instances.

A link has a source node, which can be any node in the view, and a target node, which must be a node representing an ontological concept. Links fall into two basic categories:

- **Is-a-links:** Representing hierarchical relationships.
- **Simple-links:** Representing non-hierarchical relationships.

A link of either category represents an instance of a named relationship, with the name being provided by an associated “link-type” (see below).

Node-Type/Link-Type Structure

Each node has an associated “node-type”, and each link an associated “link-type”. In both cases, types are drawn from pre-defined sets that have been defined as part of the view configuration.

Amongst other things, these type definitions specify the way in which the nodes and links will map to the underlying DL constructs. They also define a basic schema, specifying:

- The permissible types of target-node for each type of link.
- The permissible types of link for which a node may be a source-node.

The nodes of a specific type constitute a single contiguous section of the ontology view, with all nodes of the type being descendents of a single root-node for the type, and with each node, other than the root, having at least one parent of the relevant type. For each node-type within a view, there must be at least one abstract-node (i.e. the root-node), possibly others, and possibly a set of instance-nodes.

Each node-type has a name, but within the view, the name is used only indirectly. The name will be the name for the root-node of the type, and will also be used in constructing the name for the is-a-link type for which it is the target-node type. The names of the is-a-link-types will be of the format: “*is-a- \langle target-node-type-name \rangle ”.*

There are two categories of link-type reflecting the two basic link-categories. These are:

- **Is-a-link-types:** There is a one-to-one correspondence between node-types and is-a-link-types. For every node-type that is defined, an is-a-link-type will be defined automatically with that node-type as its target-node-type.
- **Simple-link-types:** Simple-link-types are defined explicitly, as part of the view configuration.

By default, the is-a-link-type for a specific node-type will only apply to the nodes of that type (other than the root). If it is required that it apply to other nodes of other types, then this must be explicitly specified as part of the view configuration (see below for more details).

Each link-type has a name, by which it is identified within the view. As described above, the name for a particular is-a-link-type is directly derived from the name of the corresponding target-node type. For simple-links the name is part of the link-type definition.

Concrete-Nodes and Data-Fields

The main differences between concrete-nodes and abstract-nodes, from the point of view of the view-user, are:

- Concrete-nodes can have attached concrete data-type values.
- Concrete-nodes cannot be used as link target-nodes, and hence cannot have any sub-nodes.

As part of the view configuration, a set of named “data-fields” can be specified, each of which can be attached to one or more node-types. Each data-field when attached to a node-type specifies a concrete value, or an expression involving concrete values, that can be attached to concrete-nodes of that type.

There are two categories of concrete-node:

- **Instance-nodes:** Representing stored instances.
- **Query-nodes:** Representing queries to retrieve sets of stored instances.

The difference between these two, from the point of view of the view-user, is that for instance-nodes the expressions associated with data-fields must be single values, whereas for query nodes they can also be disjunctions and numerical limits.

2.2 View Entities

The ontology view consists of the current state of a set of view entities of the following types:

Node:

- **Represents in Ontology View.** Either an ontological concept, or an instance of an ontological concept.
- **Sub-Entities:** abstract-node, concrete-node.

Abstract-Node:

- **Represents in Ontology View.** Ontological concept.
- **Mapping to underlying DL:** Maps directly to a DL class.

Concrete-Node:

- **Represents in Ontology View.** Ontological concept with concrete data-values attached.
- **Sub-Entities:** instance-node, query-node.

Instance-Node:

- **Represents in Ontology View:** Instance of ontology concept.
- **Mapping to underlying DL:** Maps to an instance in the instance store, whose description consists of property restrictions and hierarchical relationships to ontological concepts, plus concrete data values.

Query-Node:

- **Represents in Ontology View:** Query for retrieving instance-nodes.
- **Mapping to underlying DL:** Maps to a general DL expression for querying over the instance store, consisting of property restrictions and hierarchical relationships referencing ontological concepts, plus expressions involving concrete data-values.

Link:

- **Represents in Ontology View:** Relationship between concepts.
- **Sub-Entities:** is-a-link, simple-link.

Is-A-Link:

- **Represents in Ontology View:** Hierarchical relationship between concepts.
- **Mapping to underlying DL:** For abstract-nodes and instance-nodes there is a one-to-one mapping from is-a-links to hierarchical relationship between DL classes. For query-nodes, it is possible for a set of is-a-links for a particular node, to map to a single super-class relationship, whose filler is a disjunction. Adding and removing is-a-links may also involve the manipulations of sets of disjoint axioms or covering constructs.

Simple-Link:

- **Represents in Ontology View:** Non-hierarchical relationship between concepts.

- **Mapping to underlying DL:** Sets of simple-links will map to sets of restrictions for a particular property. In general there is not a one-to-one relationship between links and restrictions. The restrictions can be of various types (including cardinality restrictions), and fillers can consist of various types of expression. See below for a further discussing of simple-link semantics.

Data-Field-Value:

- **Represents in Ontology View:** Concrete data-value.
- **Mapping to underlying DL:** Maps to a concrete value for a data-type property, which forms part of the description of some instance in the instance store.

Data-Field-Filler:

- **Represents in Ontology View:** Expression involving concrete data-values. This can be either a single-value, a disjunction of values, or a numerical limit or range.
- **Mapping to underlying DL:** Maps to an expression involving concrete data-values that can be associated with a data-type property, which forms part of some instance-store query expression.

2.3 Configuration Entities

A view configuration consists of the definition of a set of the configuration entities, of the following types:

- **Node-Type:** Represents a type of node.
- **Link-Type:** Represents a type of link. *Sub-Entities: is-a-link-type, simple-link-type.*
- **Is-A-Link-Type:** Represents a type of hierarchical link, with specific target node-type.
- **Simple-Link-Type:** Represents a type of non-hierarchical link, associated with specific property, and with specific target node-type.
- **Data-Field:** Represents a type of field associated with a concrete-node, to which data-field-values and data-field-fillers can be attached.
- **Node-Map:** Represents one of a pre-defined set of mapping-types between nodes and DL classes.
- **Link-Map:** Represents one of a pre-defined set of mapping-types between sets of links and sets of DL property restrictions.
- **Namespace:** Represents namespaces to which certain DL constructs that map to view entities may be assigned.

Although instances of most of these configuration entities are part of the structure of the view, play a role in presenting the view to the user, and help to determine what types of manipulations can, and cannot, be performed, they are not things that can be manipulated within the view.

2.4 Additional Terminology

In addition to the view entities and the configuration entities, the description of the view mechanisms also requires the following additional terminology:

- **Type-Root-Node:** The single node from which all nodes of a particular type are descended. This node will have the same name as the node-type.
- **Type-Root-Class:** The DL class that maps to the type-root-node, and hence will also have the same name as the node-type.
- **Anchor Is-A-Link:** An is-a-link that links a type-root-node to a node of the parent node-type (see below for a description of hierarchical relationships between node-types).
- **Homogenous Is-A-Link:** An is-a-link for which the source node-type and the target node-type are identical.
- **Primary Is-A-Link:** Either an anchor is-a-link or a homogeneous is-a-link.
- **Secondary Is-A-Link:** Any is-a-link that is not a primary is-a-link (see below for a discussion of primary and secondary is-a-links).
- **Link-Group:** A set of links of a specific type originating from a particular node.

Note: Each of these terms is relevant to the description of the view mechanism, but none of them are directly relevant to the user of the view.

3. Semantic Mappings and Cardinality

3.1 Overview

The ways in which specific nodes and links will map to DL constructs is specified via a set of pre-defined node-maps and link-maps. A node-map specifies how an individual node will map to an individual DL class, whereas a link-map specifies how a link-group (consisting of a set of links of a specific type originating from a particular node) will map to a set of DL property restrictions.

As part of the view configuration, a node-map is specified for each node-type and a link-map for each link-type. Both node-map and link-map will be specified as being either “fixed” or “default”. If a node-map is defined as “default” then it can be overridden within the view for individual nodes. Similarly a default link-map can be overridden for individual link-groups.

Note: The “fixed” or “default” status of the semantic mappings for node-types and link-types is always applicable as far as abstract-nodes are involved, but when concrete-nodes are involved there are the following complications:

- The node-map is always fixed to “description” for instance-nodes, and “definition” for query-nodes, no matter what node-map and node-map status is defined for the node-type.

- No matter what the status of the link-map for the link-type, for links attached to query-nodes, the link-map is always fixed to “any” when the link-type is defined as unique, and is always editable otherwise.

For simple-links, it is also possible for the view-user to specify cardinality values to be associated with the links. Such values will map to additional property restrictions in the underlying ontology.

3.2 Node-Maps

There are just two types of node-map that can be attached to nodes, both of which map to the underlying DL in an obvious way. The options are:

“Description”:

- **Meaning:** The node describes a concept, but does not provide a full definition.
- **Mapping to underlying DL:** Node maps to a “primitive” class.

“Definition”:

- **Meaning:** The node provides a full definition of a concept.
- **Mapping to underlying DL:** Node maps to a “definition” class.

3.3 Link-Maps (Simple-Links)

The set of simple-links of a specific type, attached to a particular node, will map to a set of property restrictions attached to the corresponding DL class. These restrictions will all be on the particular property that corresponds to that simple-link-type. The semantics of a link-map, when applied to such a group of links, specify what types of values that property may or may not have for that concept. The available link-maps are as follows:

“Each”:

- **Meaning:** The set of link target nodes represents a set of values, each of which the property must have.
- **Mapping to underlying DL:** Maps to a set of “some” restrictions, whose fillers are the classes that map to the target nodes.

“Any”:

- **Meaning:** The set of link target-nodes represent a set of values, at least one of which, the property must have.
- **Mapping to underlying DL:** Maps to a single “some” restrictions, whose filler is a disjunction of the classes that map to the target nodes.

“Only”:

- **Meaning:** The set of link target-nodes represent a set of values, which are the only possible values for the property.
- **Mapping to underlying DL:** Maps to a single “all” restriction, whose filler is a disjunction of the classes that map to the target nodes

Ontology View	Description Logic	Query GUI		
<i>hasPart</i> : each : roof wall door	some (<i>hasPart</i> , <i>roof</i>) some (<i>hasPart</i> , <i>wall</i>) some (<i>hasPart</i> , <i>door</i>)	<i>hasPart</i>		
			<i>roof</i>	
		AND	<i>wall</i>	
		AND	<i>door</i>	
		AND POSSIBLY OTHERS		
<i>hasPart</i> : any : roof wall door	some (<i>hasPart</i> , or (<i>roof</i> , <i>wall</i> , <i>door</i>))	<i>hasPart</i>		
			<i>roof</i>	
		OR	<i>wall</i>	
		OR	<i>door</i>	
		AND POSSIBLY OTHERS		
<i>hasPart</i> : only : roof wall door	all (<i>hasPart</i> , or (<i>roof</i> , <i>wall</i> , <i>door</i>))	NONE		
<i>hasPart</i> : each+only : roof wall door	some (<i>hasPart</i> , <i>roof</i>) some (<i>hasPart</i> , <i>wall</i>) some (<i>hasPart</i> , <i>door</i>) all (<i>hasPart</i> , or (<i>roof</i> , <i>wall</i> , <i>door</i>))	<i>hasPart</i>		
			<i>roof</i>	
		AND	<i>wall</i>	
		AND	<i>door</i>	
		AND NO OTHERS		
<i>hasPart</i> : any+only : roof wall door	some (<i>hasPart</i> , or (<i>roof</i> , <i>wall</i> , <i>door</i>)) all (<i>hasPart</i> , or (<i>roof</i> , <i>wall</i> , <i>door</i>))	<i>hasPart</i>		
			<i>roof</i>	
		OR	<i>wall</i>	
		OR	<i>door</i>	
		AND NO OTHERS		
<i>hasPart</i> : none : roof wall door	all (<i>hasPart</i> , not (or (<i>roof</i> , <i>wall</i> , <i>door</i>)))	<i>hasPart</i>		
		NOT	<i>roof</i>	
		OR	<i>wall</i>	
		OR <i>door</i>		

Figure 2: Example Link-Maps

“None”:

- **Meaning:** The set of link target-nodes represent a set of values none of which the property may have.
- **Mapping to underlying DL:** Maps to a single “all” restriction, whose filler is a negated disjunction of the classes that map to the target nodes.

“Each+Only”:

- **Meaning:** The set of link target nodes represents a set of values, each of which the property must have, and which are the only possible values for the property.
- **Mapping to underlying DL:** Combination of the mappings for “each” and “only”.

“Any+Only”:

- **Meaning:** The set of link target nodes represents a set of values, at least one of which the property must have, and which are the only possible values for the property.
- **Mapping to underlying DL:** Combination of the mappings for “any” and “only”.

3.3 Link-Maps (Is-A-Links)

Only two of the above link-maps can be applied to groups of is-a-links. The semantics of these link-maps when applied to is-a-links are compatible with their semantics when applied to simple-links. The relevant link-maps are as follows:

“Each”:

- **Meaning:** The set of link target nodes represents a set of values, each of which is a parent of the node.
- **Mapping to underlying DL:** Maps to a set of sub-class/super-class relationships between the class that maps to the source node, and the classes that map to the target nodes.

“Any”:

- **Meaning:** The set of link target nodes represents a set of values, at least one of which is a parent of the node.
- **Mapping to underlying DL:** Maps to a single of sub-class/super-class relationships between the class that maps to the source node, and a disjunction of the classes that map to the target nodes.

Note: The “any” link-map is only applicable to query-nodes. The semantics of abstract nodes and instance-nodes is always specified by the “each” link-map.

Figure 2 shows a set of examples of how the different types of link-map are responsible for map a set of links to a set of DL constructs, and a rough depiction of what each expression will look like in the query GUI.

3.4 Simple-Link Cardinality

Within the view, it is also possible to associate cardinalities with individual simple-links. For each simple-link it is possible to specify an exact cardinality, a minimum cardinality, a maximum cardinality, or a range. This will cause appropriate cardinality restrictions to be created on the DL class associated with the relevant source-node, thus adding to the set of property restrictions involved in the mapping of the link-group to the underlying DL.

Figure 3 shows an example of how cardinality values act in combination with the link-maps in the ontology view to specify a set of DL constructs.

Ontology View	Description Logic	Query GUI		
<i>hasPart</i> : each : <i>roof</i> (1) <i>wall</i> (>=4) <i>door</i> (1-3)	some (<i>hasPart</i> , <i>roof</i>) some (<i>hasPart</i> , <i>wall</i>) some (<i>hasPart</i> , <i>door</i>) exactCardinality (<i>hasPart</i> , <i>roof</i> ,1) minCardinality (<i>hasPart</i> , <i>wall</i> ,4) minCardinality (<i>hasPart</i> , <i>door</i> ,1) maxCardinality (<i>hasPart</i> , <i>door</i> ,3)	<i>hasPart</i>		
			<i>roof</i>	1
		AND	<i>wall</i>	min 4
		AND	<i>door</i>	1 - 3
		AND	POSSIBLY OTHERS	

Figure 3: Example Link-Map with Cardinalities

4. Additional Complications

4.1 Node Fields

Each node can be viewed as having a set of fields requiring fillers. These can be derived from link-type definitions that provide fields whose fillers must be sets of nodes, or data-fields definitions that provide fields whose fillers must be data-type values, or when attached to query-nodes, expressions involving data-type values.

For each node, a set of applicable fields can be derived directly from the node-type definition. Every link-type and data-field specified for the node-type provides a field for the node. In addition however, there will also be fields corresponding to extra is-a-link-types that will automatically apply to different sub-sets of these nodes.

If T is the node-type, P is the parent node-type (if there is one) and R is the root-node for T, then:

- For all nodes other than R, the is-a-link-type with target node-type T will apply.

- For R (if P exists) the is-a-link-type with target node-type P will apply.

Since the is-a-link-type with target node-type T will always apply to nodes other than R, and since it will never make sense for it to apply to R (since R is the root-node for T, and it therefore will not have any is-a-links to other nodes of type T), this link-type cannot be one of those specified as part of the node-type definition.

On the other hand, if P exists, the is-a-link-type with target node-type P, will always apply to R, but it may also make sense for it to apply to the other nodes of type T (the example in section 4 provides an illustration of this). Therefore it is possible to specify this link-type as part of the node-type definition to ensure that it will also apply to nodes other than R.

4.2 Primary and Secondary Node Inheritance Structures

The ontology view mechanism makes the distinction between “primary” and “secondary” node inheritance structures. The first is the main inheritance structure of the ontology, whereas the second provides extra is-a-links between node-types, and hence provides additional axes of classification. These two structures are constructed from the following categories of is-a-link:

- **Primary is-a-links:** All links of the following types:
 - Anchor is-a-links.
 - Homogenous is-a-links.
- **Secondary is-a-links:** All other is-a-links in the ontology view.

The concepts of primary and secondary inheritance structures form a natural division within the ontology, which can be useful in helping to disentangle multiple-inheritance.

The primary and secondary is-a-links are distinguished in the following ways:

- The concepts of “sub-node” and “super-node” within the API, are defined only in terms of primary is-a-links.
- Only primary is-a-links are represented as hierarchical links in the tree-view within the GUI.
- Primary and secondary is-a-links can be treated differently when defining other aspects of the view configuration, such as uniqueness of is-a-links (see below).

Example

Figure 1 depicts an example ontology showing the distinction between primary and secondary node structures. The primary is-a-links connect nodes within the “Animal” section, and nodes within the “Domestic-Animal” section, as well as anchoring the “Domestic-Animal” root-node, to the “Animal” node. Secondary is-a-links can then be inserted, specifying that certain types of animal also belong to one of the

“Domestic-Animal” categories (as shown by the secondary is-a-link, linking “Goldfish” to “Pet”).

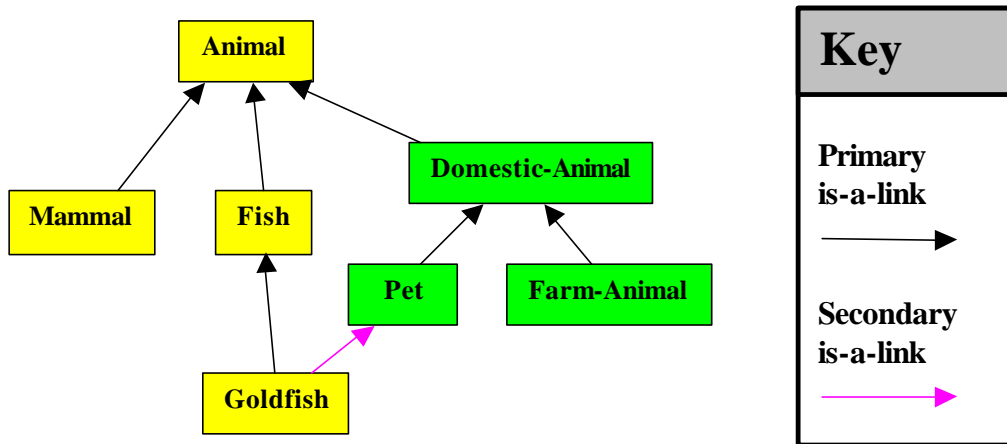


Figure 1: Primary and Secondary Node Inheritance Structures

4.3 Disjoint and Covering Strategies

It is possible to specify as part of the node-type definition, that as the ontology view is edited, sets of constructs should be maintained in the underlying DL ontology, specifying that sets of sibling-nodes of the relevant node-type be disjoint, or form a covering, or both. In both cases it is possible to specify three different strategies:

- **“Maintain”**: Maintain relevant constructs, adding/modifying the relevant constructs as siblings are created, and modifying/removing them as siblings are removed.
- **“Remove”**: Ensure that any relevant constructs that exist in the DL ontology, either as the result of editing with a different view configuration, or editing outside of the view mechanism, are removed.
- **“Ignore”**: Make no modifications involving the relevant constructs.

Note: There are no view entities concerned with disjoints and coverings, and all manipulation of the DL ontology resulting from the specification of the above strategies are entirely invisible to the user of the ontology view.

4.4 Node-Type Hierarchy

The node-types can form a simple hierarchy, with every type having either one or zero parents. The position of a node-type in the hierarchy can have various effects. If node-type C has parent type P, then these effects will be as follows:

- **Node-Section Hierarchy:** The hierarchical relationship between node-types must be reflected in the relationship between the sections of the ontology

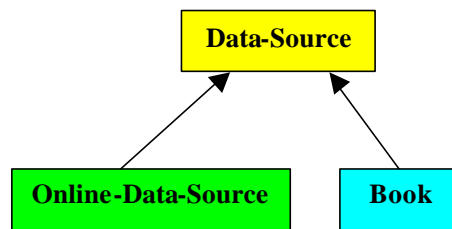
containing the respective sets of nodes, hence the type-root-node for C will be the sub-node of some node of type P

- **Inheritance of Fields:** All fields applicable to P will also be applicable to C (with the exception of the is-a-link-type that has P as the target node-type, which must be explicitly specified if required for C). Field inheritance is recursive. Hence, any fields inherited by type P will also be inherited by type C.
- **Inheritance of Link-Target Status:** Whenever nodes of type P are valid as targets for some link-type, nodes of type C will also be valid.

The node-section hierarchy must always hold, whereas both the inheritance of fields, and the inheritance of link-target status are optional, and can be turned on or off for each node-type, as part of the view configuration.

Example

Figure 4 shows the skeleton ontology that will be produced when a new ontology is created using a view configuration containing three node-types “Data-Source”, “Online-Data-Source” and “Book”, with the latter two types being sub-types of “Data-Source”. The skeleton node structure directly reflects this node-type hierarchy.

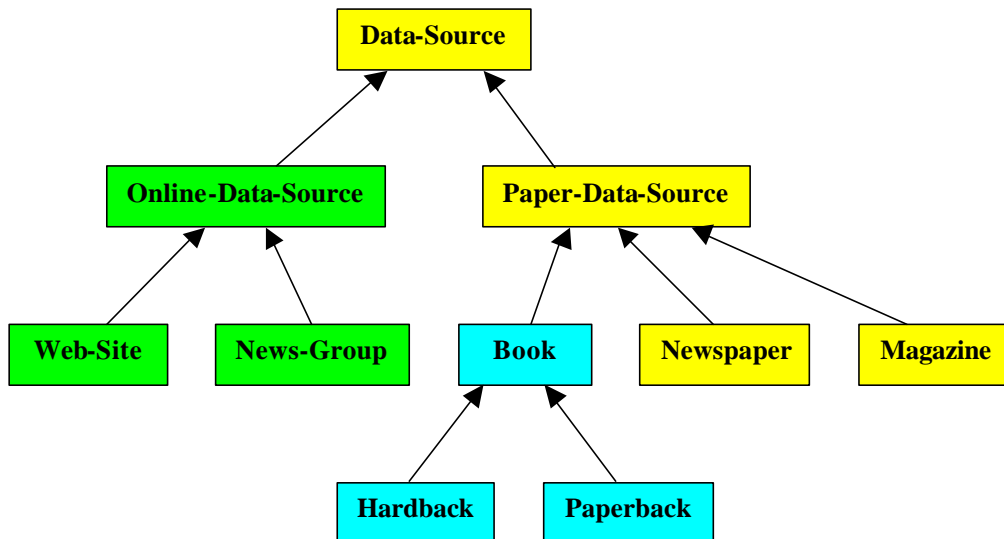


**Figure 4: Skeleton Ontology View
(Reflecting Node-Type Hierarchy)**

Figure 5 shows a more populated version of this same ontology. Note that, despite the fact that the node structure has been expanded, and the root-node for the “Book” section has been given a new parent node, the original hierarchical relationships still hold between the three sections of the ontology, as defined by the three different types of node.

4.5 Simple-Link-Type Hierarchy

The simple-link-types can form a simple hierarchy, with every type having either one or zero parents. If a simple-link-type C has a parent link-type P, then whenever the view is started up, the mechanism will ensure that the appropriate hierarchical relationships are defined between the corresponding properties in the DL ontology.



**Figure 5: Populated Ontology View
(Compatible with Node-Type Hierarchy)**

4.6 Namespaces

All entities in the ontology view are identified by a simple leaf-name, with no namespace component being visible within the view (either to the user of the GUI or to the user of the API). By default all entities created in the DL ontology will be created in the default namespace for the ontology, but that namespace will be hidden from the view.

However, within the view configuration it is possible to define other, non-default, namespaces and to allocate node-types, link-types, and data-fields to these namespaces, causing the associated DL entities to be created in, or in the case of existing entities, moved to, the relevant namespaces. These non-default namespaces will still be hidden from the ontology view, and hence the entities in the view will still be identified by simple leaf names. This means that even when multiple namespaces are being used, names of entities in the view cannot be duplicated.

5. Complete Entity Definitions

5.1 View Entities

The definitions of the view entities described above involves the specification of the following types of information:

Node:

- **Name:** Name of node (and also name of DL class).
- **Node-Type:** Type of node.
- **Outward-Links:** Set of links originating from node.

Instance-Node:

- **Name:** Name of node (and also name of DL class).
- **Node-Type:** Type of node.
- **Outward-Links:** Set of links originating from node.
- **Data-Field-Values:** Set of data-field-values attached to node.

Query-Node:

- **Node-Type:** Type of node.
- **Outward-Links:** Set of links originating from node.
- **Data-Field-Fillers:** Set of data-field-fillers attached to node.

Is-A-Link:

- **Is-A-Link-Type:** Type of link.
- **Source-Node:** Node at which link originates.
- **Target-Node:** Node at which link terminates.

Simple-Link:

- **Simple-Link-Type:** Type of link.
- **Source-Node:** Node at which link originates.
- **Target-Node:** Node at which link terminates.

Data-Field-Value:

- **Data-Field:** The field to which the value applies.
- **Value:** The value for the field.

Data-Field-Filler:

- **Data-Field:** The field to which the filler applies.
- **Filler-Expression:** The single-value, disjunction of values, or numerical restrictions, for the field.

4.6 Configuration Entities

The definitions of the configuration entities described above involves the specification of the following types of information:

Node-Type:

- **Name:** Name of node-type.
- **Namespace:** Namespace to which classes will be assigned.
- **Node-Map:** The type of node-map that specifies the node semantics.
- **Semantics-Status:** Either “fixed” or “default”.
- **Parent-Node-Type:** The parent node-type (possibly none).
- **Inherit-Fields:** Boolean specifying whether node-type will inherit fields from any parent node-type.
- **Inherit-Link-Target-Status:** Boolean specifying whether node-type will inherit its link-target-status from any parent node-type.
- **Disjoint-Strategy:** Specifies whether disjoints between sets of siblings (with relevant node-type) should be “maintained”, “removed” or “ignored”.
- **Covering-Strategy:** Specifies whether constructs representing the covering of parent classes by offspring (with the relevant node-type) should be “maintained”, “removed” or “ignored”.
- **Unique-Primary-Is-A-Links:** Boolean specifying whether there should be only a single primary is-a-link for nodes of relevant type (or, in the case of query-nodes, possibly a disjunction).
- **Unique-Secondary-Is-A-Links:** Similar to the above.
- **Link-Types:** Set of link-types that can originate from nodes of relevant type.
- **Data-Fields:** Set of applicable data-fields.
- **Node-Edit-Status (GUI Only):** Specifies whether nodes of relevant type are “editable” or “non-editable” within the GUI.
- **Node-Colour (GUI Only):** Specifies what colour nodes will appear within the GUI.

Is-A-Link-Type:

- **Target-Node-Type:** Node-type of target nodes for links of relevant type.

Simple-Link-Type:

- **Name:** Name of link-type. This is also the name of the property in the underlying DL to whose property restrictions the links will map.
- **Namespace:** Namespace to which DL property will be assigned.
- **Parent-Link-Type:** Parent simple-link-type (possibly none).
- **Target-Node-Type:** Node-type of target nodes for links of relevant type.
- **Link-Map:** The type of link-map that specifies the link semantics.
- **Link-Map-Status:** Either “fixed” or “default”.
- **Unique:** Boolean specifying whether the DL property will be unique. Also specifies whether the link-map will be restricted to be of a type compatible with a unique value (i.e. not “each” or “each+only”).
- **Symmetric:** Boolean specifying whether the DL property will be symmetric.
- **Transitive:** Boolean specifying whether the DL property will be transitive.

Data-Field:

- **Name:** Name of data-field.
- **Namespace:** Namespace to which DL property will be assigned.
- **Data-Type:** Either String, boolean, integer, real, URI or date.

Namespace:

- **URI:** URI of namespace.

5. Loading and Editing Views

5.1 Loading a View

A view is loaded by loading a DL ontology into the view mechanism, with a particular view configuration. There are three possible scenarios:

- **New Ontology:** The DL ontology does not yet exist, in which case a skeleton node-structure will be automatically created based on the view configuration.
- **Existing Ontology View:** The DL ontology exists, and was created using the same view configuration, in which case the view will be reconstituted exactly as it was.
- **Imported Ontology:** The DL ontology exists, but was created by some other means, and is being imported into the view mechanism, in which case a view will be created that provides as good a match as possible match between the ontology and the view configuration.

In each of these cases the view is created from the combination of the DL ontology and the view configuration, via a single view-loading procedure that consists of the following sequence of operations (each of which will be described in more detail below):

- Checking/creation of type-root-classes.
- Checking/creation of simple-link-type properties.
- Prioritising of node-types.
- Creation of nodes (for each node-type, in priority order).
- Creation of is-a-links (with appropriate types).
- Creation of missing anchor is-a-links.
- Creation of simple-links (with appropriate types, semantics, and cardinalities).
- Checking/modification of DL class semantics.
- Checking/creation of hierarchical relationships between DL properties.
Checking/modification of DL property attributes.
- Checking/creation/removal of disjoint axioms and covering constructs.

Note: If the ontology did not previously exist, most of these steps to not apply (only the first, second and sixth steps are required).

In each of the scenarios described above, the net result of this process will be a loaded ontology whose structure is compatible with the view configuration. If an existing

ontology has been newly imported into the view, the loading process may have involved some modifications to the ontology itself.

Checking/Creation of Type-Root-Classes

For each node-type, a check is made to ensure that the type-root-class exists in the DL ontology. If it does not currently exist (as will be the case for a new ontology, and possibly for an imported ontology) it will be created.

Checking/Creation of Simple-Link-Type Properties

For each simple-link-type, a check is made to ensure that the associated DL property exists. If it does not currently exist (as will be the case for a new ontology, and possibly for an imported ontology) it will be created.

Prioritising of Node-Types

As the ontology is reloaded, when a node is created for a class that is the descendent of the root-classes of more than one node-type, there must be some means of deciding to which type the node will be allocated. This is achieved by fixing a priority ordering of node-types, with the highest priority node-type having first grab at descendent classes.

For example, when the view for the ontology shown in figure 5 is being reconstituted, the “Animal” node-type should have the highest-priority, allowing it to grab “Goldfish” before “Domestic-Animal” gets there. The appropriate priority can be determined from the fact that a secondary is-a-link-type has been defined as going from “Animal” to “Domestic-Animal”, whereas no such secondary is-a-link-type exists going in the opposite direction.

In cases where node-type priority is relevant, there will always be a definite ordering of this type (since cyclic chains of secondary is-a-link-types are not permitted). Hence, we can arrive at the following general rule for deciding priority:

“If a node-type A has an associated secondary is-a-link-type with target node-type B, then, type A will have priority over type B”

Creation of Nodes

The nodes are recreated for each node-type in turn, with the types being taken priority order. For each node-type a downward traversal of the inheritance graph for the DL ontology is performed, beginning with the type-root-class. A node of the appropriate type will be created for any class encountered. The traversal will stop short of any classes that are root-classes for other node-types, or for which nodes have already been created.

This process creates a node of the appropriate type for every type-root-class, and every class that is a descendent of a type-root-class. If the ontology has been created using the view, this should cover every class in the ontology, whereas if the ontology

is being imported, there may be some classes that do not show up as nodes in the view.

Creation of Is-A-Links

Once all the nodes have been created and allocated appropriate types, the recreation of the is-a-links, and the correct allocation of is-a-link-types is a reasonably straightforward process.

Creation of Missing Anchor Is-A-Links

A check is made to ensure that the hierarchical relationship between node-types is reflected in the ontology view. For, any node-type that has a parent type defined, but does not have an anchor is-a-link (as will be the case for a new ontology, and possibly for an imported ontology), an anchor is-a-link will be created, linking the type-root-node to the parent root-node.

Recreation of Simple-Links

All of the simple-links in the view are then reconstructed, with the semantics for each group of links, as well as any link-cardinalities, being reconstructed from an analysis of the relevant sets of property restrictions.

If for any class, the existing set of property restrictions of a particular type conflicts with the semantics of the link-type (as may happen if the ontology is being imported), and if the link-map-status for that link-type is defined as “fixed”, the relevant set of restrictions will be removed and replaced with a new set conforming to the required link-type semantics. The fillers for the new set of restrictions are obtained from the fillers for the existing restrictions.

Checking/Modification of DL Class Semantics

If the semantics of any DL class (primitive/definition) conflicts with the semantics of the relevant node-type, they will be altered accordingly.

Checking/Creation of Hierarchical Relationships between DL Properties

If the required hierarchical relationships between DL properties, as specified by the hierarchical structuring of the simple-link-type definitions, do not exist, they will be created.

Checking/Modification of DL Property Attributes

If any DL properties do not have the required uniqueness, symmetry and transitivity statuses, as specified by the simple-link-type definitions, these values will be set accordingly.

Checking/Creation/Removal of Disjoint Axioms and Covering Constructs

If the settings of the “Disjoints-Strategy” and “Coverings-Strategy” attributes that have been defined for the node-types, do not match with the current state of the DL ontology, then the two will be brought into line by the addition and removal of appropriate constructs in the ontology.

5.2 Importing an Ontology

As mentioned above, it is possible to take an existing DL ontology, created by other some means such as OilEd or conversion from RDF, and import it into the ontology view mechanism, using an appropriate view configuration. This not only allows the ontology to be maintained via the view mechanism, but is also a means of imposing a semantic structure that did not exist in the original ontology.

For instance, a DL ontology may have been converted from RDF format, by a process where each RDF triple is converted into a “some” type property restriction. By default, sets of such restrictions would map to sets of links with each such set having a link-map of “each”. However, it would be possible to create a view configuration in which other link-maps (e.g. “each+only”, “any”) are defined as having “fixed” status for various link-types. Then when the existing ontology is loaded, the original sets of “some” restrictions will be replaced by other sets of restrictions that conform to the semantics of the fixed link-maps.

5.3 Editing a View

Once an ontology view exists, either an initial skeleton, or one that has been further populated, it can be edited either via the API, or via the ontology editing GUI in the following ways:

- Addition of new node.
- Removal of existing node.
- Renaming of existing node.
- Creation of new link.
- Removal of existing link.
- Setting of cardinality for existing simple-link (exact, minimum, maximum or range).
- Re-setting of link-map specifying the semantics of a set of simple-links of a particular type.

The exact workings of the API and GUI will not be described here, but the following is a set of general remarks describing the way in which certain aspects of the editing process are handled by the view mechanisms.

- Any new node to be created must be specified as being a sub-node of an existing node. Any node thus created will assume the type of its parent node

(hence, there will always be an appropriate is-a-link-type available for creating the required link to the parent node).

- It is not possible to remove the last primary is-a-link for a node. Hence, the primary inheritance structure will remain intact.
- The API (and hence the GUI) will not allow cyclic chains of is-a-links to be created. If an ontology being imported into a view does contain cyclic inheritance, then as it is loaded, it will be doctored in an essentially random way to remove such cycles, with appropriate warnings being given.
- The API will not allow redundant is-a-links to exist in an ontology. If an ontology being imported into a view does contain redundant is-a-links, they will be removed, and an appropriate warning will be given.
- After the ontology has been loaded, the API operates a default strategy that prevents the addition of redundant is-a-links, and causes newly redundant is-a-links to be removed.
- The GUI overrides the default is-a-link strategy, with one that involves prompting the user whenever the addition of a new is-a-link will result in the redundancy of any links (either the new one, or existing ones).

6. Pizza Example

This section works through a simple example to show how an ontology can be created and maintained using the ontology view mechanism. This example is based on an adapted version of the pizza ontology used in the OilEd training course. Figure 6 shows the completed ontology displayed in the ontology editing GUI.

It is assumed that the following people will be involved in creating the ontology:

- **Knowledge Engineer:** Performs initial domain analysis and creates and manages the various views.
- **Domain Expert(s):** Provide expert knowledge of the pizza domain.

The process of ontology creation will involve the following steps:

- Knowledge engineer creates “Master View” (via view configuration GUI).
- Knowledge engineer creates skeleton ontology (via ontology editing GUI, configured with Master View).
- Knowledge engineer creates domain experts views (via view configuration GUI).
- Domain experts populate relevant sections of the ontology (via ontology editing GUI, configured with the appropriate views).
- Knowledge engineer runs classifier and displays the results (via ontology editing GUI, configured with Master View).

Each of these steps is described in turn below.

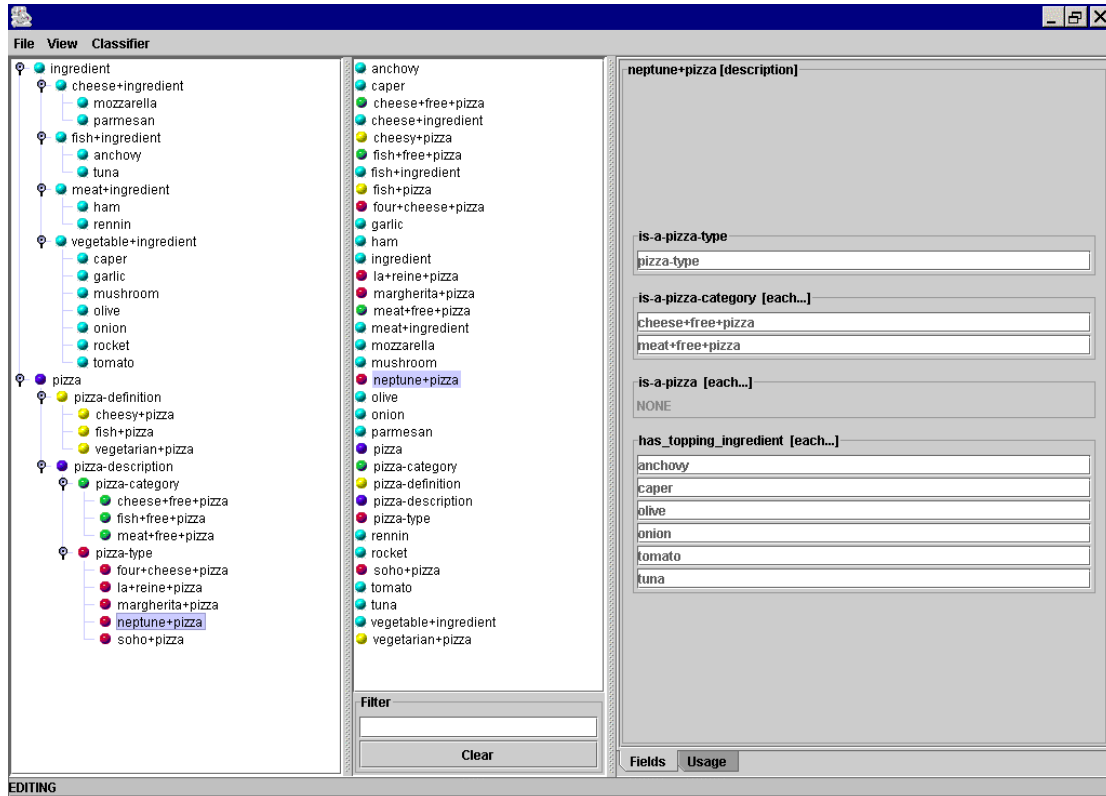


Figure 6:
View Configuration GUI: Node-Types (Master View)

6.1 Configuration of “Master View”

The “Master View” is the view created by the knowledge engineer, and is used to display and manage the entire ontology. The other views will consist of edited subsets of the Master View.

The Master View consists a set of node-type and link-type definitions. There are no data-fields defined, and no namespaces. Hence, each entity in the DL ontology will be created in the default namespace. The node-type and link-type definitions are described below, and their representation in the view configuration GUI is shown in figures 7 and 8.

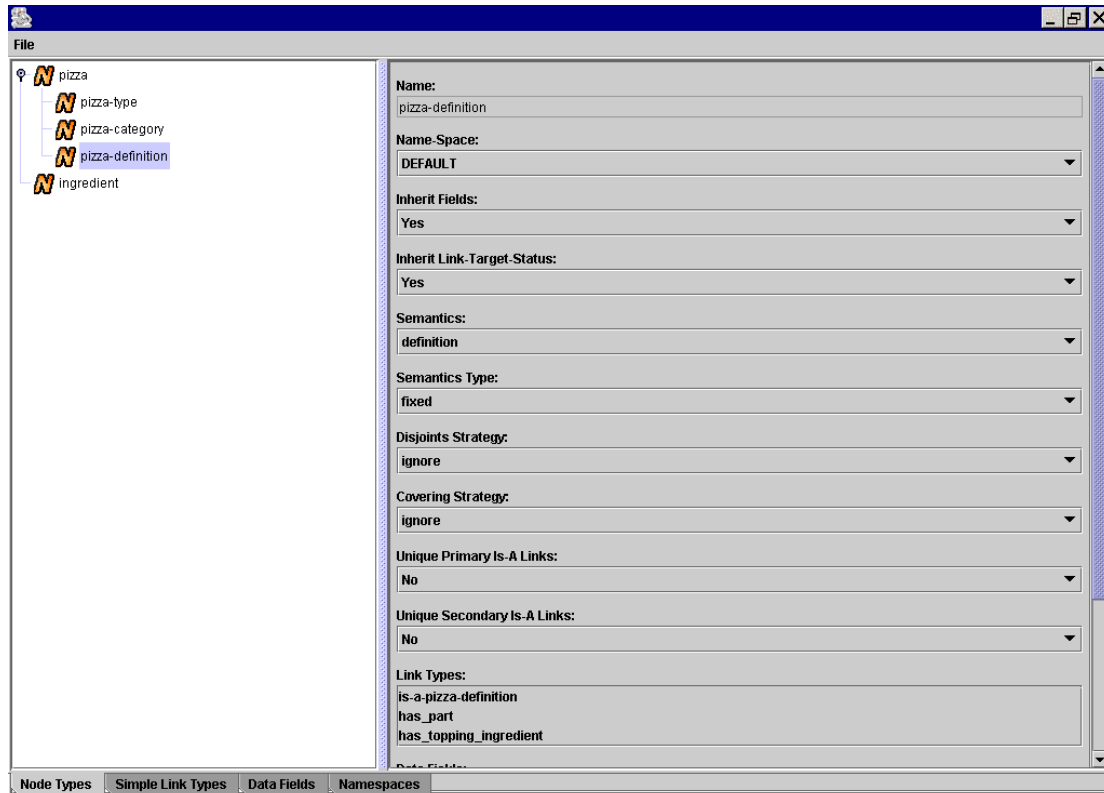


Figure 7:
View Configuration GUI: Node-Types (Master View)

Definition of “ingredient” Node-Type

This node-type covers the section of the ontology where the pizza ingredients will be specified. This will be a simple hierarchy with no simple links between the nodes, hence there will be no link-types specified (other than the “is-a-ingredient” link-type which is defined automatically for the “ingredient” node-type). The definition of the node-type includes the following attribute settings:

- **Node-Map:** Set to “description”, specifying that the ingredient nodes represent partial ingredient descriptions, rather than full definitions.
- **Node-Map-Status:** Set to “fixed”, specifying that the semantics cannot be edited.
- **Unique Primary Is-A Links:** Set to “yes”, specifying that the ingredient nodes will form a strict hierarchy.

Note: For all of the node-types in this view, “Node Edit-Status” is set to “editable”, and “Node-Colour” is given some value unique to that node-type.

Definition of “pizza-type” Node-Type

This node-type covers the section of the ontology where specific pizza types are described (e.g. “margherita”). As with the ingredients section, this will be a simple hierarchy where each node has fixed node-map of “description”. Hence, with the exception of “Node-Colour” the attribute settings are identical to those for the “ingredient” node-type.

The definition for this node-type also includes the specification of the following link-types:

- **has-topping-ingredient:** Allows the specification of a set of ingredients for each pizza-type (see below for further description of this link-type).
- **is-a-pizza-category:** Allows each pizza-type to be assigned to one or more broad categories of pizza (see below for a description of the “pizza-category” node-type).
- **is-a-pizza-definition:** Allows each pizza-type to be automatically classified as conforming to one or more pizza definitions (see below for a description of the “pizza-definition” node-type).

The body of the description of each pizza-type consists of a set of links of the first two of these link-types. The first is used for specifying a set of ingredients that the pizza must have, and the second for specifying a set of broad categories into which the pizza-type fits (e.g. “meat-free-pizza”).

Definition of “pizza-category” Node-Type

This node-type covers the section of the ontology where broad categories of pizza can be described. These category descriptions can then be used (via inheritance) in the descriptions of “pizza-type” nodes. Since it may be useful to describe compound categories that inherit their descriptions from multiple more basic categories, multiple-inheritance within the node-type is permitted. Also, since inheritance by “pizza-type” nodes is permitted, secondary is-a links become relevant. Therefore the following attribute settings are required:

- **Unique Primary Is-A Links:** Set to “no”, specifying that multiple parents within the node-type, should be permitted.
- **Unique Secondary Is-A Links:** Set to “no”, specifying that other types of nodes, for which this type is a valid parent, can have multiple parents of this type (since a “pizza-type” may be defined as belonging to more than one “pizza-category”).

There is a single link-type specified for this node-type:

- **has-part:** Allows the specification of the parts that must be present or absent for pizzas of each category (see below for further description of this link-type).

Definition of “pizza-definition” Node-Type

This node-type is similar to the “pizza-category” type, except that, rather than providing chunks of description to be used in building up the pizza type descriptions, it will provide definitions that will be used to automatically classify the pizza types. Hence, the attribute settings for this node-type are similar to those for “pizza-category”, except that the node-map is set to “definition” (the link-map-status is still set to fixed, since all nodes of this type will be definitions rather than descriptions).

For this node-type, both “has-part” and “has-topping-ingredient” link-types are specified. This is because they are both relevant to the “pizza-type” descriptions, the first directly, the second indirectly, via inheritance from “pizza-category”.

Definition of “pizza” Node-Type

This node-type is used to provide some top-level structuring of the ontology. It covers the entire pizza section of the ontology, and is the parent node-type for “pizza-type”, “pizza-category”, and “pizza-definition”.

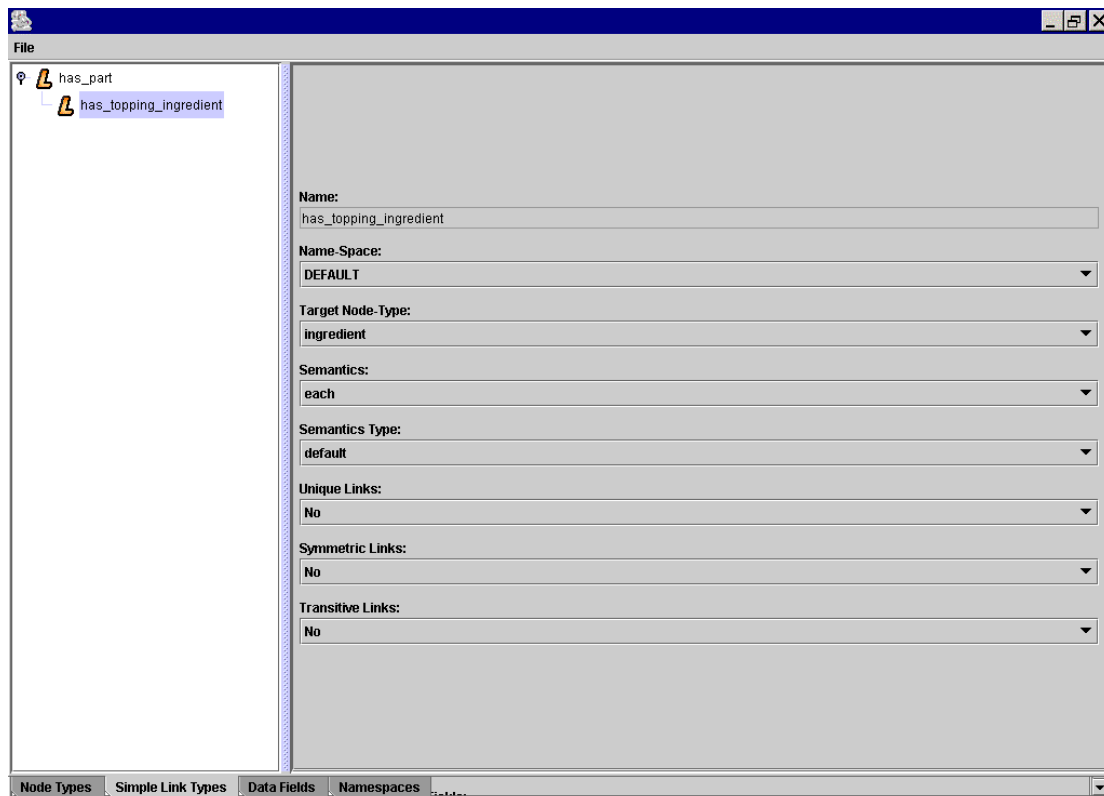


Figure 8:
View Configuration GUI: Link-Types (Master View)

Definition of “has-part” Link-Type

This link-type specifies anything that can be part of a pizza. The definition includes the following attribute settings:

- **Target Node-Type:** Set to “ingredient”.
- **Link-Map:** Set to “each”, since it will most commonly be used to specify a list of ingredients, each of which the pizza must have.
- **Link-Map-Status:** Set to “default”, since in some instances, different types of semantics will be required (e.g. “none” is used in the definition of “vegetarian-pizza”).

Definition of “has-topping-ingredient” Link-Type

This link-type specifies anything that can be a topping-ingredient for a pizza, and is a sub-type of “has-part”. The definition includes the following attribute settings:

- **Target Node-Type:** Set to “ingredient”.
- **Link-Map:** Set to “each”, since it will be used to specify a list of ingredients, each of which the pizza must have.
- **Link-Map-Status:** Set to “fixed” since the assumption is made that it will always be used in the above fashion.

6.2 Creation of Skeleton Ontology

Once the Master View configuration has been defined, the initial skeletal pizza ontology can be created. This is done by starting up the Ontology Editing GUI (configured by the new view), and opening up a new ontology file. A skeleton pizza ontology will be created automatically. A root-node will be created for each node-type, and the hierarchical relationships between node-types will be mirrored by the creation of appropriate is-a-links between these nodes. The ontology will now appear as in figure 9.

The knowledge engineer will now use this view to perform a couple of initial ontology edits, before handing over the ontology to the various domain experts to edit it via their respective views. These edits are as follows:

Creation of “pizza-description” node: This node is a direct sub-node of pizza, and will act as a parent node for both the “pizza-category” and “pizza-type” nodes (i.e. the root-nodes for the section of the ontology responsible for providing the pizza description).

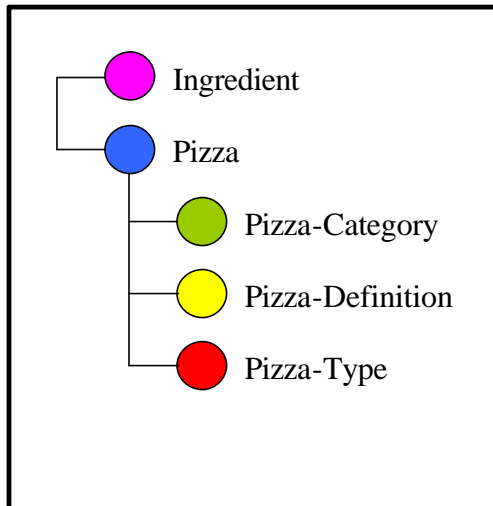


Figure 9:
Initial Skeleton Ontology

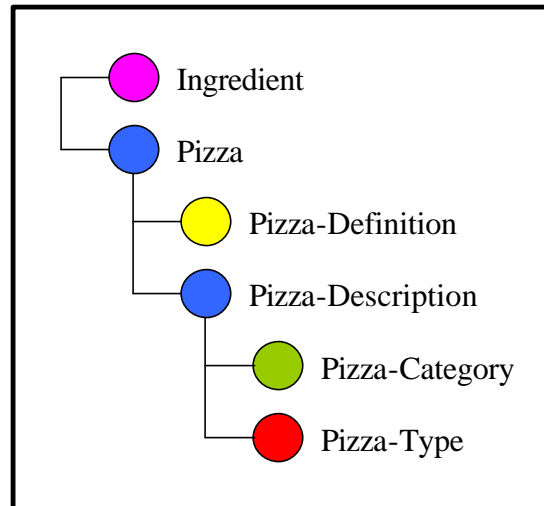


Figure 10:
Modified Skeleton Ontology

Re-positioning of “pizza-category” and “pizza-type” nodes: The is-a-links from these nodes to the “pizza” node must be replaced with is-a-links to the new “pizza-description” node. The ontology editing GUI is used to add the new links, and will, after checking with the user, automatically remove the old (now redundant) is-a-links.

Addition of is-a-link from “pizza-type” node to “pizza-definition” node: This is necessary to allow the classifier to classify the pizza types in the required fashion.

The ontology will now appear as in figure 10. The is-a-link from “pizza-type” to “pizza-definition” will not show up in the hierarchical view, since it is a secondary, rather than a primary, is-a-link.

The insertion of the “pizza-description” node, and the re-positioning of the “pizza-type” and “pizza-category” nodes, demonstrates how the skeletal structure of the ontology can be manipulated, whilst maintaining the hierarchical relationship between node-types.

6.3 Configuration and Use of Domain Experts Views

A set of views is created to allow the domain experts to create the main body of the ontology. Each of these views consists of a sub-set of the node-types and link-types defined for the Master View, with a few additional tweaks. The skeleton node structure for each view is shown in figure 11, and the individual views are described below.

Ingredients View

This view is used to describe the pizza ingredients. It consists only of the “ingredient” node-type, and no link-types, and allows a domain expert to create a simple hierarchy of ingredients.







Ingredients	Pizza-Description	Pizza-Definition
 Ingredient	 Ingredient  Pizza-Category  Pizza-Type	 Ingredient  Pizza-Definition

Figure 11:
Domain Experts Views: Skeleton Ontologies

Pizza Description View

This view is used to provide the descriptions of the basic pizza types. It consists of the “ingredient”, “pizza-category” and “pizza-type” node-types, with the following modifications:

- The “Node Edit-Status” attribute for “ingredient” is set to “not-editable”, so that the ingredients section of the ontology can be referenced by the pizza descriptions, but cannot be edited.
- “pizza-type” does not have the “is-a-pizza-definition” link-type attached, since this link-type is not required for creating the pizza descriptions, but only for displaying the results of classification (also, the “pizza-definition” section of the ontology is not visible in this view).

The domain expert will produce the “pizza-type” descriptions by specifying sets of ingredients in the “has-topping-ingredient” fields, and sets of general categories in the “is-a-pizza-category” fields. The expert will also define these categories in the “pizza-category” section of the ontology.

Whereas, the definition of the “pizza-type” nodes will only involve creating lists of ingredients and categories, and in some cases specifying cardinalities, the definition of “pizza-category” nodes may also involve editing the link semantics. For instance, in descriptions such as “meat-free-pizza”, the link-map for “has-part” must be set to “none”.

Figures 12 and 13 are rough depictions of what some of the “pizza-type” and “pizza-category” descriptions look like in the ontology editing GUI.

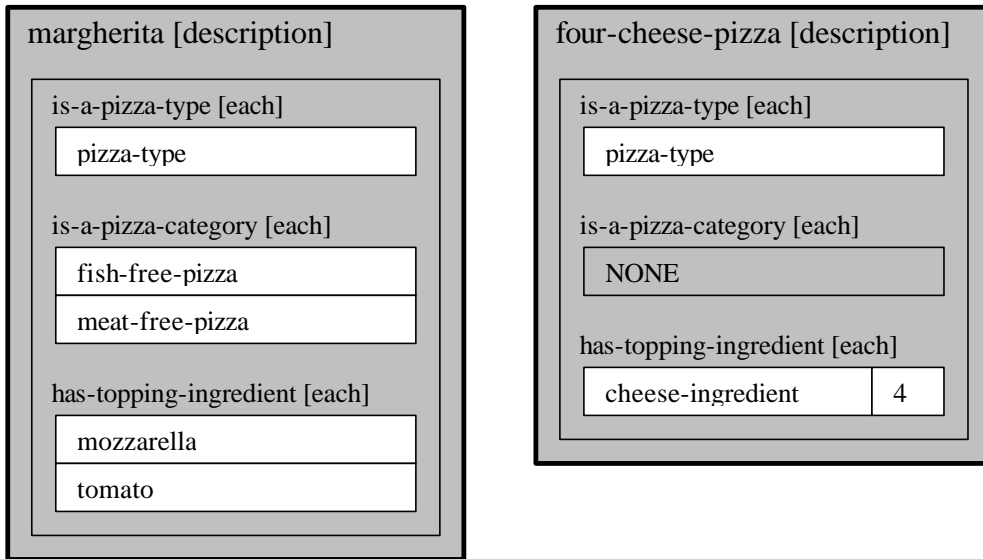


Figure 12:
“pizza-type” Descriptions

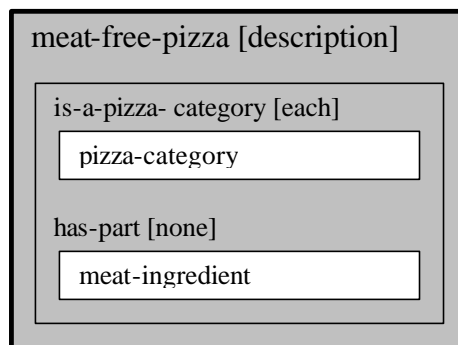


Figure 13:
“pizza-category” Description

Pizza Definition View

This view is used to define the broad categories under which the pizzas can be classified. It consists of the “ingredient” and “pizza-definition” node-types. As with the “Pizza Description View” the “Node Edit-Status” attribute for “ingredient” is set to “not-editable”.

The “pizza-definition” nodes will be similar to the “pizza-category” nodes, only they have both “has-part” and “has-topping-ingredient” fields.

Figure 14 is a rough depiction of what a couple of the “pizza-definition” definitions look like in the ontology editing GUI.

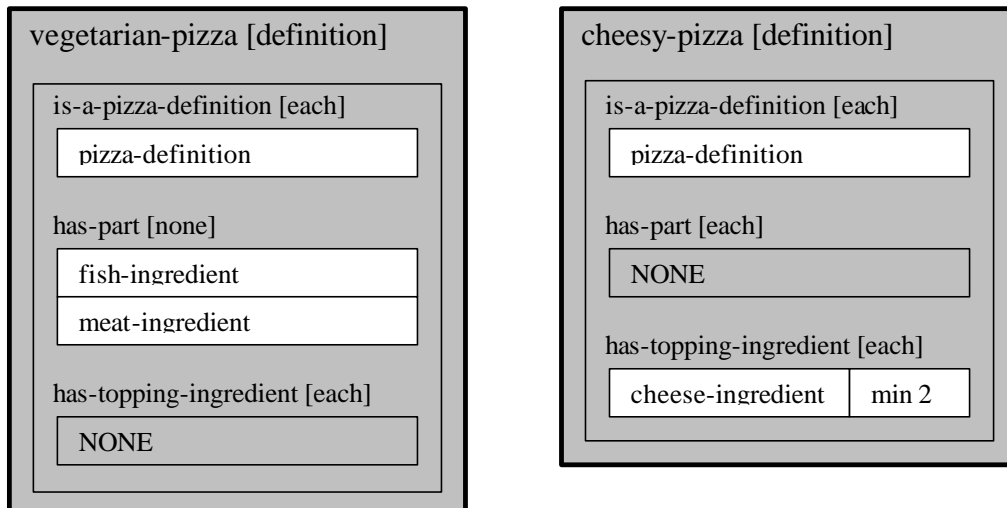


Figure 14:
“pizza-definition” Definitions

6.4 Classification

Once the body of the ontology has been created by the domain experts, or possibly at intervals during the creation process, the knowledge engineer will run the classifier (which can be done either from the ontology editing GUI, or from OilEd), and once the ontology has been classified, the ontology editing GUI, configured with the master view, can be used to display the results.

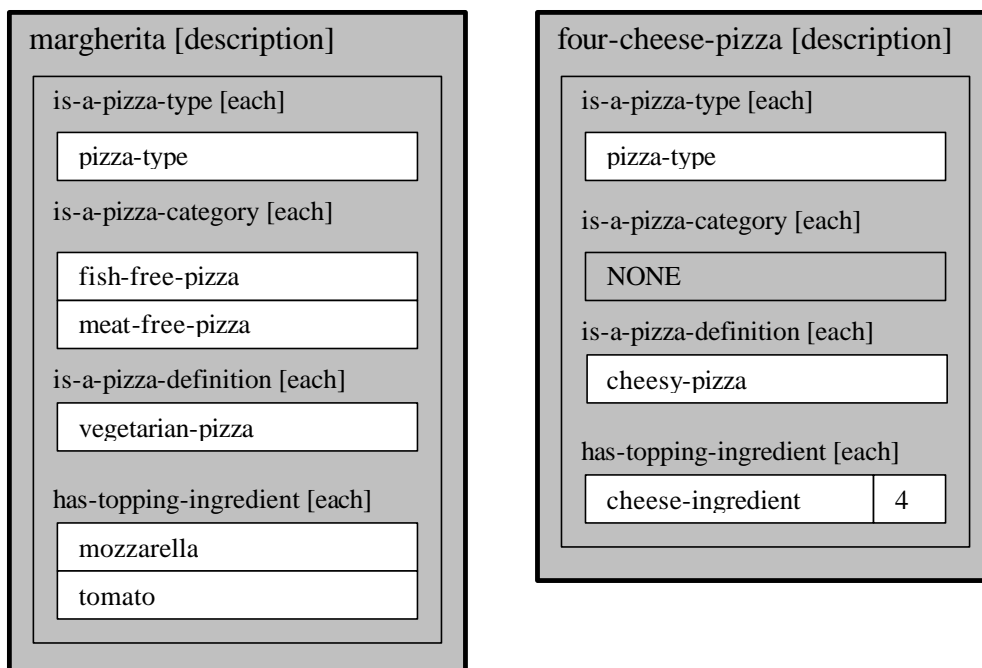


Figure 15:
Post-Classification “pizza-type” Descriptions

Running the classifier will cause various “pizza-type” classes in the DL ontology to be categorised under specific “pizza-definition” classes. When the classified ontology is displayed in the ontology editing GUI, these new sub-class relationships will appear as “is-a-pizza-definition” links on the “pizza-type” nodes (since these links are secondary, rather than primary is-a-links, they will not show up in the hierarchical view window). Figure 15 shows what the “pizza-type” descriptions in figure 12 look like after they have been classified.

6.5 Alternative Version of View Configuration

With the view configurations described in the preceding parts of this section, in order for the classification to produce the desired results, some of the “pizza-type” descriptions must include explicit information specifying that the pizza does not contain certain ingredients. This information can not be deduced automatically.

For instance for the “margherita” to be classified as a “vegetarian-pizza”, it must be explicitly stated that it does not contain either meat or fish (this is done via inheritance from the “meat-free-pizza” and “fish-free-pizza” nodes from the “pizza-category” section of the ontology).

This section describes an alternative view configuration where such information does not need to be stated explicitly by the ontology developers. To achieve this, the view utilises both more complex link semantics, and the specification of a node-type disjoints strategy.

The new view configuration requires the following modifications to the master view configuration, as described above:

- The “has-part” and “has-topping-ingredient” link-types are merged into one, with the knock-on effects on the node-type definitions.
- The link-map for “has-part” is set to “each+only”. This means that when the domain expert provides the list of ingredients for each pizza type, the DL constructs that are created will specify, not only that these ingredients must be present, but also that no other ingredients are present (the link-map-status retains its setting of “default”, since variable semantics will still be required for the “pizza-definition” section).
- The “Disjoints-Strategy” attribute is set to “maintain” for the “ingredient” node-type. This means that as ingredient nodes are added and removed, a set of disjoint axioms will be automatically maintained in the DL ontology, specifying that every member of each set of siblings, is disjoint from every other member.

With these modifications to the view configuration, the ontology creation process is simplified. It is no longer necessary to explicitly specify those ingredients that are not present in a particular pizza type. This information can now be derived from the “each+only” link-map of the “has-part” links that are created, together with the set of

disjoint axioms being automatically maintained for the ingredients. Therefore, the classifier can automatically deduce that, for instance, a “margherita” is a “vegetarian-pizza”.

This makes the task of describing pizza-types more straightforward. In fact, with this alternative configuration, it would be possible to remove the “pizza-category” node-type since, within the view as it stands, it is used only for specifying the types of ingredients that are not present (although it could foreseeably be used for defining other kinds of category that would be useful).