

Table of Contents

- Unit 1 {
 - 1. What is an ontology?
 - 1.1 The Role of Ontologies in the Semantic Web
 - 1.2 Theoretical Foundations of Ontologies
- Unit 2 {
 - 2. How can we build ontologies? Methods, techniques and methodologies
- Unit 3 {
 - 3. How can we implement ontologies? Ontology languages
- Unit 4 {
 - 4. How can we use ontologies? Reasoners and ontology APIs
- Unit 5 {
 - 5. How can we build Semantic Web applications?



How can we implement ontologies? Reasoners and Ontology APIs

Asunción Gómez-Pérez
Mariano Fernández-López
Oscar Corcho

asun@fi.upm.es, mfernandez.eps@ceu.es, ocorcho@cs.man.ac.uk

Grupo de Ontologías
Laboratorio de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo sn,
28660 Boadilla del Monte, Madrid, Spain



Main References



Gómez-Pérez, A.; Fernández-López, M.; Corcho, O. **Ontological Engineering**. Springer Verlag. 2003



<http://jena.sourceforge.net/>



<http://www.dl.kr.org/>

Acknowledgements

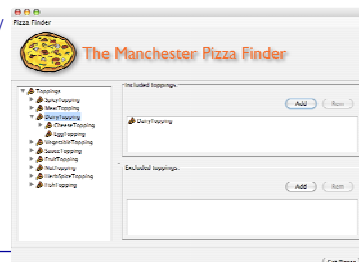
- **Asunción Gómez-Pérez and Mariano Fernández-López**
 - Most of the slides have been done jointly with them
- **Nick Drummond and Matthew Horridge (University of Manchester)**
 - Reasoning with OWL ontologies
- **Kim, Hyun-joo (ISLAB, Hanyang University), Jing deng (University of Colorado), Philip McCarthy (IBM DeveloperWorks)**
 - Jena 2

Table of contents

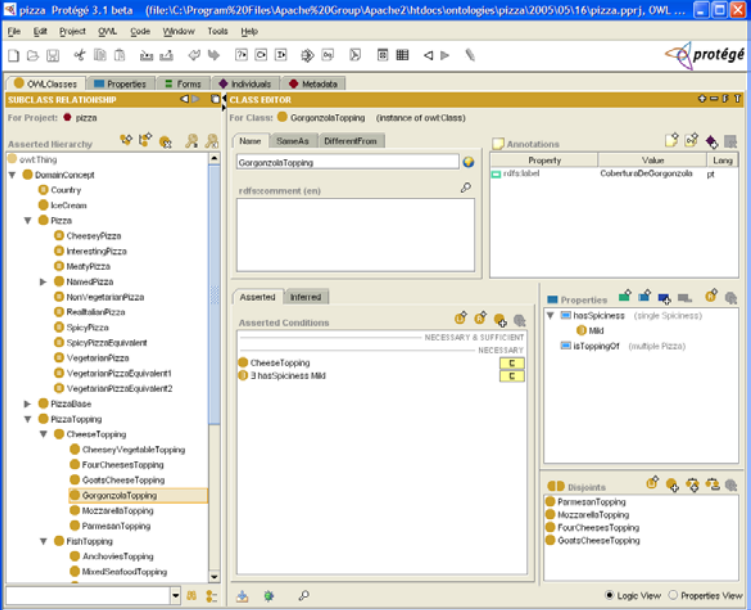
- Reasoning with OWL ontologies
 - Consistency checking
 - Disjointness
 - Restrictions
 - Primitive and Defined classes
 - Polyhierarchies (multiple classifications)
 - Untangling
 - Alternative definitions for a class (Vegetarian Pizzas: only vegetarian toppings, no meat or fish toppings or not a MeatyPizza?)
 - Union classes and covering axioms
 - The Open World Assumption (closure)
 - Negation in OWL
 - Elephant Traps – Common modelling errors
 - Functional properties
 - Intersection classes
 - Universal restrictions
- Using an ontology API to deal with OWL ontologies

Our Domain and Our Application

- Pizzas selected as a domain for several reasons:
 - They are fun and fairly neutral
 - They are internationally known
 - They are highly compositional
 - They have a natural limit to their scope
- Application
 - The PizzaFinder
 - www.co-ode.org/downloads/pizzafinder/



Starting with a Pizza Ontology...

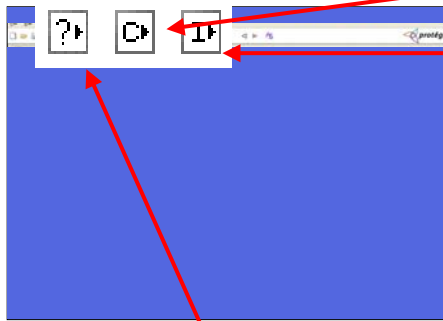


Ontological Engineering 7 ©Asunción Gómez-Pérez, M. Fernández-López, O. Corcho

Consistency Checking

- We've just created a class that doesn't really make sense
 - What is a *MeatyVegetableTopping*? *What is a MadCow*?
- We'd like to be able to check the logical consistency of our model
 - This is one of the tasks that can be done by a Reasoner/Classifier
- Protégé-OWL supports the use of reasoners implementing the DIG interface
 - The reasoner is independent of the ontology editor
 - We can choose an implementation depending on our needs (eg some may be more optimised for speed/memory, others may have more features)
- These reasoners typically set up a service running locally or on a remote server
 - Protégé-OWL can only connect to reasoners over an `http://` connection

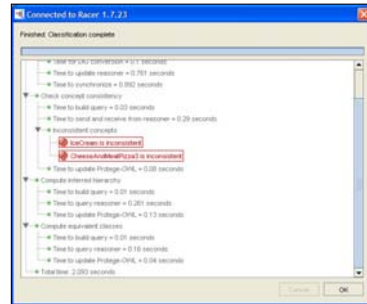
Accessing the Reasoner



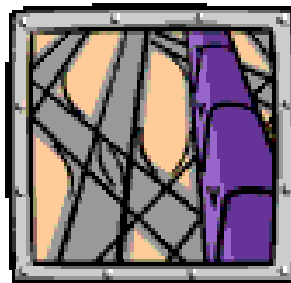
Classify taxonomy
(and check consistency)

Compute inferred types
(for individuals)

Just check consistency
(for efficiency)

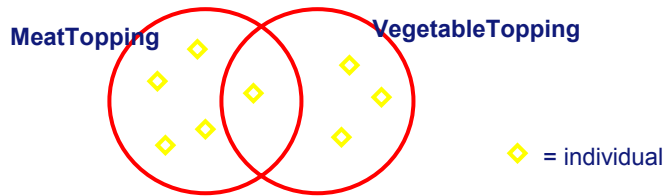


Check consistency



Disjointness

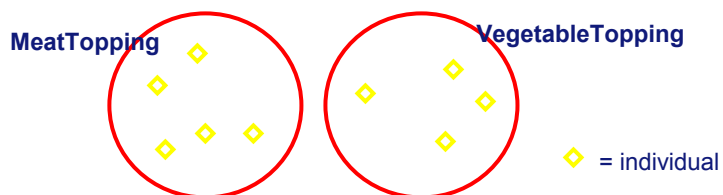
- OWL assumes that classes overlap



- ▶ This means an individual could be both a **MeatTopping** and a **VegetableTopping** at the same time
- ▶ We want to state this is not the case

Disjointness

- If we state that classes are disjoint

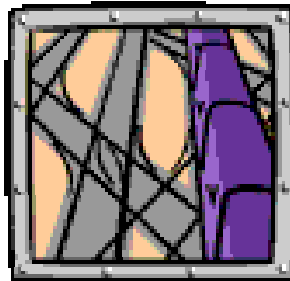


- ▶ This means an individual cannot be both a **MeatTopping** and a **VegetableTopping** at the same time
- ▶ We must do this explicitly in the interface

ClassesTab: Disjoints Widget



Check consistency



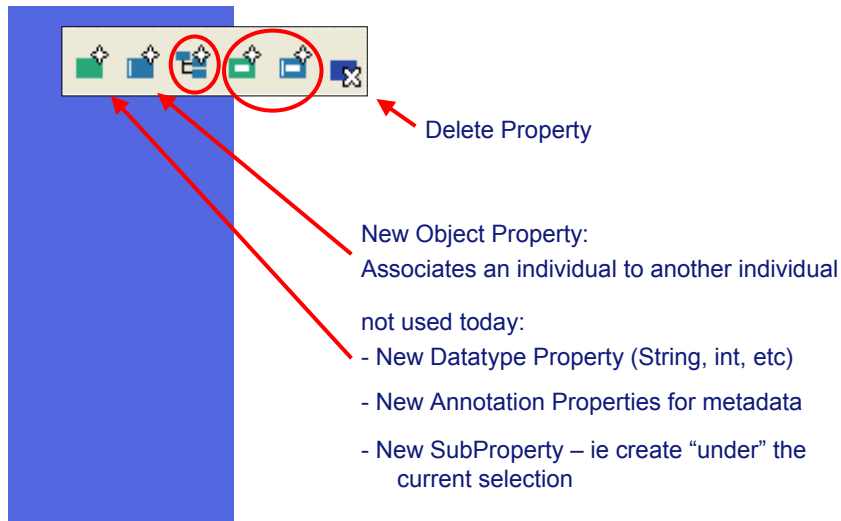
Why is MeatyVegetableTopping Inconsistent?

- We have asserted that a MeatyVegetableTopping is a subclass of two classes we have stated are disjoint
- The disjoint means nothing can be a MeatTopping and a VegetableTopping at the same time
- This means that MeatyVegetableTopping can never contain any individuals
 - The class is therefore **inconsistent**
 - This is what we expect!
- It can be useful to create classes we expect to be inconsistent to “test” your model – often we refer to these classes as “probes”
 - generally it is a good idea to document them as such to avoid later confusion

Table of contents

- Reasoning with OWL ontologies
 - Consistency checking
 - Disjointness
 - Restrictions
 - **Primitive and Defined classes**
 - Polyhierarchies (multiple classifications)
 - Untangling
 - Alternative definitions for a class (Vegetarian Pizzas: only vegetarian toppings, no meat or fish toppings or not a MeatyPizza?)
 - Union classes and covering axioms
 - The Open World Assumption (closure)
 - Negation in OWL
 - Elephant Traps – Common modelling errors
 - Functional properties
 - Intersection classes
 - Universal restrictions
- Using an ontology API to deal with OWL ontologies

Creating Properties



Delete Property

New Object Property:
Associates an individual to another individual
not used today:

- New Datatype Property (String, int, etc)
- New Annotation Properties for metadata
- New SubProperty – ie create “under” the current selection

Creating Properties. Naming conventions

- **Use camelNotation**
 - Lowercase letter to begin
- **Create properties using 2 standard naming patterns:**
 - has... (eg hasColour)
 - is...Of (eg isTeacherOf) or other suffixes (eg ...In ...To)
- **Advantages:**
 - It is easier to find properties
 - It is easier for tools to generate a more readable form (see tooltips on the classes in the hierarchy later)
 - Inverses properties typically follow this pattern eg hasPart, isPartOf

Class Restrictions: Associating Properties with Classes

- **Property that we want to use to describe Pizza individuals**
 - **hasTopping**
- **Steps**
 - **Go back to the Pizza class and add some further information**
 - **Use the Conditions widget**
 - **Conditions can be any kind of Class**
 - Named superclasses (already added)
 - Class restrictions of type “Anonymous Class”

Conditions Widget

Conditions asserted by the ontology engineer

Add different types of condition

Definition of the class (later)

Description of the class

The screenshot shows the 'Conditions Widget' interface. It has two tabs: 'Asserted' (selected) and 'Inferred'. Below the tabs is a list of 'Asserted Conditions' for the 'Pizza' class. The conditions are:

- hasTopping CheeseTopping**: NECESSARY & SUFFICIENT
- ∃ hasGreasiness VeryGreasy**: NECESSARY
- ∃ hasBase PizzaBase**: INHERITED [from Pizza]

Red arrows point from the text labels to the corresponding parts of the widget: 'Definition of the class (later)' points to the 'Pizza' class name; 'Description of the class' points to the 'hasTopping CheeseTopping' condition; 'Conditions asserted by the ontology engineer' points to the top toolbar; 'Add different types of condition' points to the '+' icon in the toolbar; and 'Conditions inherited from superclasses' points to the '∃ hasBase PizzaBase' condition.

Conditions inherited from superclasses

Conditions Types

Logical (Anonymous) Classes

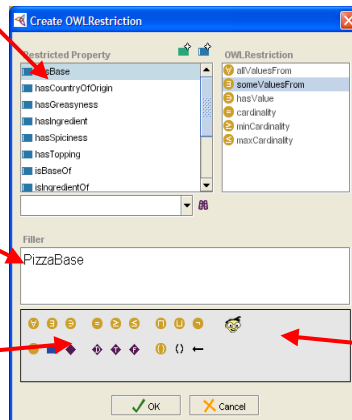


Creating Restrictions

Restricted Property

Filler Expression

Expression Construct Palette

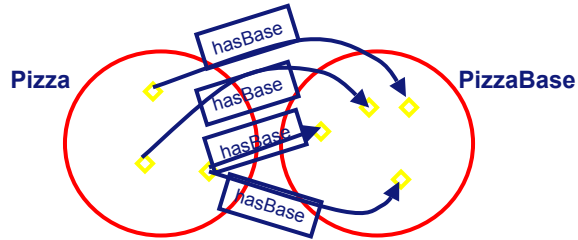


Restriction Type

Syntax check

What does this mean?

- We have created a restriction: \exists hasBase PizzaBase on Class Pizza as a necessary condition

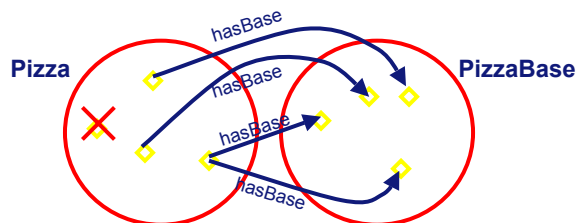


“If an individual is a member of this class, it is necessary that it has at least one hasBase relationship with an individual from the class **PizzaBase**”

“Every individual of the **Pizza** class must have at least one base from the class **PizzaBase**”

What does this mean?

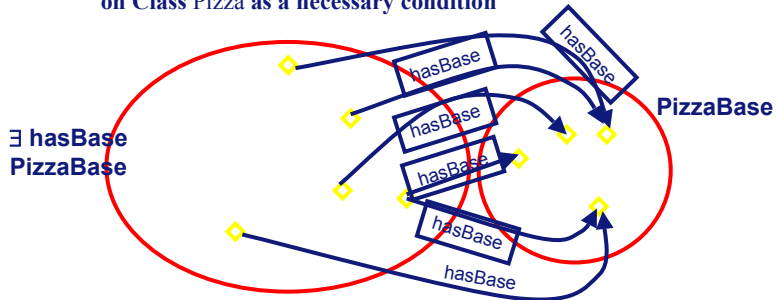
- We have created a restriction: \exists hasBase PizzaBase on Class Pizza as a necessary condition



- ▶ “There can be no individual, that is a member of this class, that does not have at least one hasBase relationship with an individual from the class **PizzaBase**”

Why?

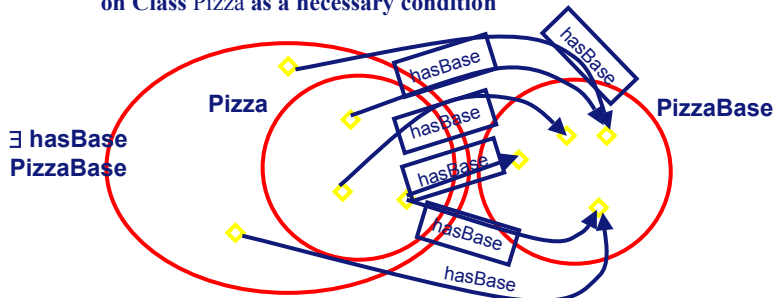
- We have created a restriction: \exists hasBase PizzaBase on Class Pizza as a necessary condition



Each Restriction or Class Expression describes the set of all individuals that satisfy the condition

Why? Necessary conditions

- We have created a restriction: \exists hasBase PizzaBase on Class Pizza as a necessary condition



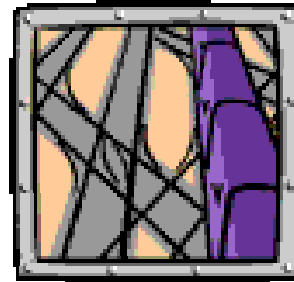
- ▶ Each necessary condition on a class is a superclass of that class
- ▶ ie The restriction \exists hasBase **PizzaBase** is a superclass of **Pizza**
- ▶ As **Pizza** is a subclass of the restriction, all **Pizzas** must satisfy the restriction that they have at least one base from **PizzaBase**

Define Cheesy Pizza and Classify



Define a Cheesy Pizza, as a Pizza that has some cheese on it

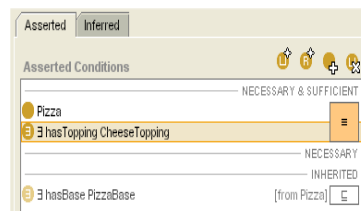
- Usual steps
 - Create primitive classes and then migrate them to defined classes
 - All the defined pizzas will be direct subclasses of Pizza
 - So, we create a CheesyPizza Class (do not make it disjoint) and add a restriction: “Every CheesyPizza must have at least one CheeseTopping”



Use the reasoner to help us produce a polyhierarchy without having to assert multiple parents

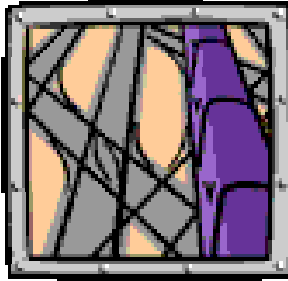
Creating a CheesyPizza

- Classifying shows that we currently don't have enough information to do any classification
- We then move the conditions from the Necessary block to the Necessary & Sufficient block which changes the meaning



- And classify again...

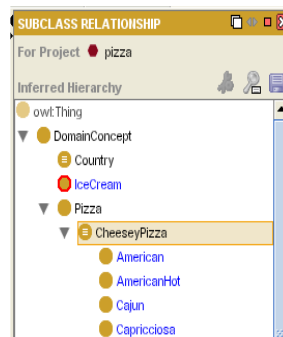
Classify



Reasoner Classification

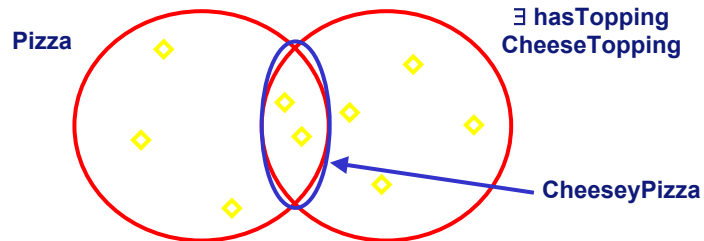
- **The reasoner has been able to infer that anything that is a Pizza that has at least one topping from CheeseTopping is a CheesyPizza**

The inferred hierarchy is updated to reflect this and moved classes are highlighted in blue



Why? Necessary & Sufficient Conditions

- ▶ Each set of necessary & sufficient conditions is an Equivalent Class



CheesyPizza is equivalent to the intersection of **Pizza** and **∃ hasTopping CheeseTopping**

Classes, all of whose individuals fit this definition are found to be subclasses of **CheesyPizza**, or are subsumed by **CheesyPizza**

Primitive Classes

- All classes in our ontology so far are Primitive
- We describe primitive pizzas
- Primitive Class = only Necessary Conditions
- They are marked as plain orange circles in the class hierarchy

We condone building a disjoint tree of primitive classes

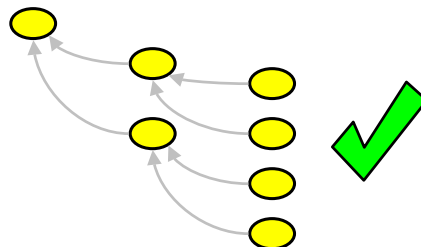


Table of contents

- Reasoning with OWL ontologies
 - Consistency checking
 - Disjointness
 - Restrictions
 - Primitive and Defined classes
 - Polyhierarchies (multiple classifications)
 - Untangling
 - Alternative definitions for a class (Vegetarian Pizzas: only vegetarian toppings, no meat or fish toppings or not a MeatyPizza?)
 - Union classes and covering axioms
 - The Open World Assumption (closure)
 - Negation in OWL
 - Elephant Traps – Common modelling errors
 - Functional properties
 - Intersection classes
 - Universal restrictions
- Using an ontology API to deal with OWL ontologies

Polyhierarchies

- By the end of this tutorial we intent to create a VegetarianPizza
- Some of our existing Pizzas should be types of VegetarianPizza
- However, they could also be types of SpicyPizza or CheeseyPizza

- We need to be able to give them multiple parents in a principled way
- We could just assert multiple parents like we did with MeatyVegetableTopping (without disjoints)

BUT...

Asserted Polyhierarchies

We believe asserting polyhierarchies is bad

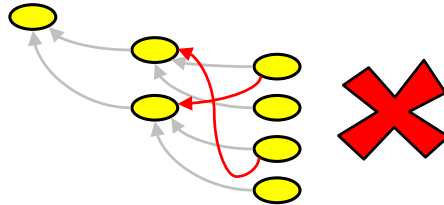
We lose some encapsulation of knowledge

Why is this class a subclass of that one?

Difficult to maintain

Adding new classes becomes difficult because all subclasses may need to be updated

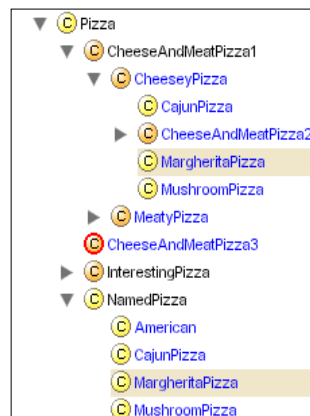
Extracting from a graph is harder than from a tree



let the reasoner do it!

Untangling

- We can see that certain Pizzas are now classified under multiple parents
- MargheritaPizza can be found under both NamedPizza and CheeseyPizza in the inferred hierarchy



Mission Successful!

Untangling

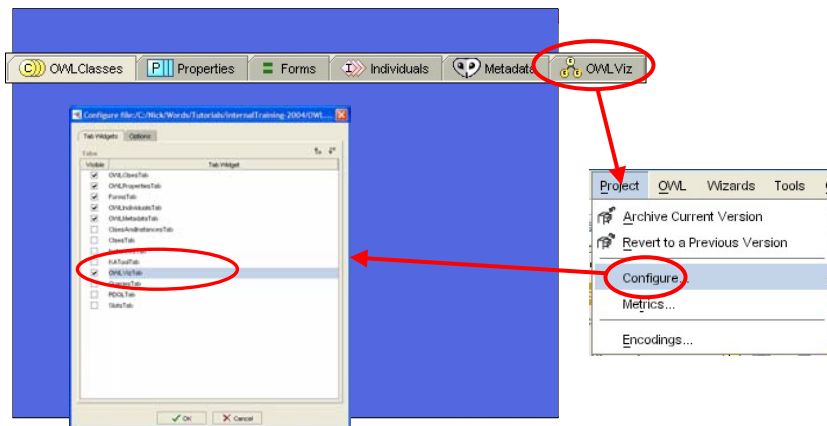
- However, our unclassified version of the ontology is a simple tree, which is much easier to maintain
- We've now got a polyhierarchy without asserting multiple superclass relationships
- Plus, we also know why certain pizzas have been classified as CheeseyPizzas
- We don't currently have many kinds of primitive pizza but its easy to see that if we had, it would have been a substantial task to assert CheeseyPizza as a parent of lots, if not all, of them
- And then do it all over again for other defined classes like MeatyPizza or whatever

Viewing polyhierarchies

- As we now have multiple inheritance, the tree view is less than helpful in viewing our “hierarchy”

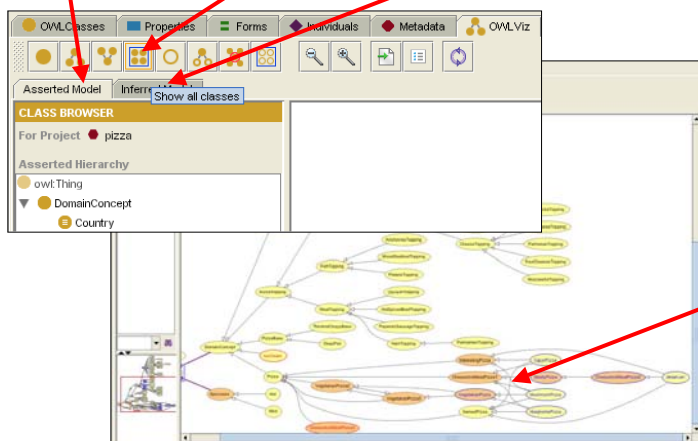


Viewing our Hierarchy Graphically



OWLviz Tab

View Asserted Model Show All Classes View Inferred Model



Polyhierarchy tangle

Using OWLViz to untangle

- **Asserted hierarchy**
 - It should be a tidy tree of disjoint primitives
- **Inferred hierarchy**
 - Tangled
- **By switching from the asserted to the inferred hierarchy, it is easy to see the changes made by the reasoner**
- **OWLViz can be used to spot**
 - Tangles in the primitive tree
 - Disjoints (including inherited ones) are marked (with a \neg)

Defined Classes

- **We've created a Defined Class, CheesyPizza**
 - It has a definition. That is *at least one* Necessary and Sufficient condition
 - Classes, all of whose individuals satisfy this definition, can be inferred to be subclasses
 - Therefore, we can use it like a query to “collect” subclasses that satisfy its conditions
 - Reasoners can be used to organise the complexity of our hierarchy
- **It's marked with an equivalence symbol in the interface**
- **Defined classes are rarely disjoint**

Table of contents

- Reasoning with OWL ontologies
 - Consistency checking
 - Disjointness
 - Restrictions
 - Primitive and Defined classes
 - Polyhierarchies (multiple classifications)
 - Untangling
 - Alternative definitions for a class (Vegetarian Pizzas: only vegetarian toppings, no meat or fish toppings or not a MeatyPizza?)
 - Union classes and covering axioms
 - The Open World Assumption (closure)
 - Negation in OWL
 - Elephant Traps – Common modelling errors
 - Functional properties
 - Intersection classes
 - Universal restrictions
- Using an ontology API to deal with OWL ontologies

Define a Vegetarian Pizza

- Not as easy as it looks...
- Define in words?
 - “a pizza with only vegetarian toppings”?
 - “a pizza with no meat (or fish) toppings”?
 - “a pizza that is not a MeatyPizza”?
- More than one way to model this

We'll start with the first example

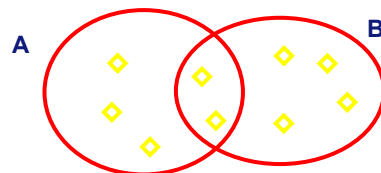
Vegetarian Pizza = Pizza with only vegetarian toppings

- **Requirements**

- Create a vegetarian topping → Union Class (aka disjunction)
- “Only” → Universal Restriction

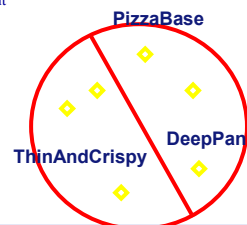
Vegetarian Topping: Union Classes and Covering Axioms

- $A \cup B$ includes
all individuals of class A and
all individuals from class B and
all individuals in the overlap
(if A and B are not disjoint)

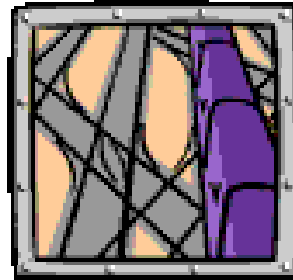


- **Covering axiom**
 - Union expression containing several covering classes
 - A covering axiom in the Necessary & Sufficient Conditions of a class means:
the class cannot contain any instances other than those from the covering classes
 - Note: If the covering classes are subclasses of the covered class, the covering axiom only needs to be a Necessary condition
 - It doesn't harm to make it Necessary & Sufficient though – its just redundant

- **Example: $\text{PizzaBase} \equiv \text{ThinAndCrispy} \cup \text{DeepPan}$**
 - The class **PizzaBase** is covered by **ThinAndCrispy** or **DeepPan**
 - All **PizzaBases** must be **ThinAndCrispy** or **DeepPan**
 - “There are no other types of **PizzaBase**”



Define Vegetarian Pizza and Classify



VegetarianPizza Classification

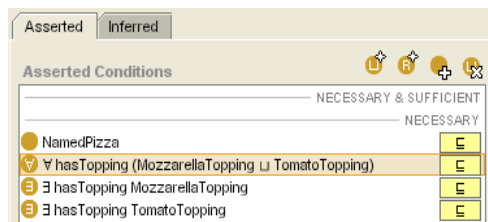
- **Nothing classifies under VegetarianPizza**
 - Actually, there is nothing wrong with our definition of VegetarianPizza
 - It is actually the descriptions of our Pizzas that are incomplete
- **The reasoner has not got enough information to infer that any Pizza is subsumed by VegetarianPizza**
- **This is because OWL makes the Open World Assumption**
 - **In a closed world (like DBs), the information we have is everything**
 - A database, for example, returns a negative if it cannot find some data.
 - **In an open world, we assume there is always more information than is stated**
 - The reasoner makes no assumption about the completeness of the information it is given
 - The reasoner cannot determine something does not hold unless it is explicitly stated in the model

Open World Assumption

- **Typical pattern**
 - Several existential restrictions on a single property with different fillers
 - Example: primitive pizzas on hasTopping
- **Must state whether a description is complete or not**
 - **Incomplete:**
 - Existential restrictions should be paraphrased by “amongst other things...”
 - **Complete:**
 - Existential restrictions should be paraphrased by “and no other XXX”
- **In our example:**
 - **We need closure for the property hasToppings**
 - In the form of a Universal Restriction with a filler that is the Union of the other fillers for that property
 - Closure works along a single property

Closure example: MargheritaPizza

- **All MargheritaPizzas must have:**
 - at least 1 topping from MozzarellaTopping and**
 - at least 1 topping from TomatoTopping and**
 - only toppings from MozzarellaTopping or TomatoTopping**



NECESSARY & SUFFICIENT	NECESSARY
NamedPizza	⊆
\forall hasTopping (MozzarellaTopping \sqcup TomatoTopping)	⊆
\exists hasTopping MozzarellaTopping	⊆
\exists hasTopping TomatoTopping	⊆

- **The last part is paraphrased into “no other toppings”**
- **The union closes the hasTopping property on MargheritaPizza**

Define Margherita Pizza and Classify

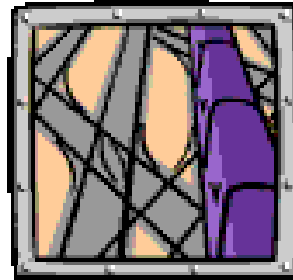
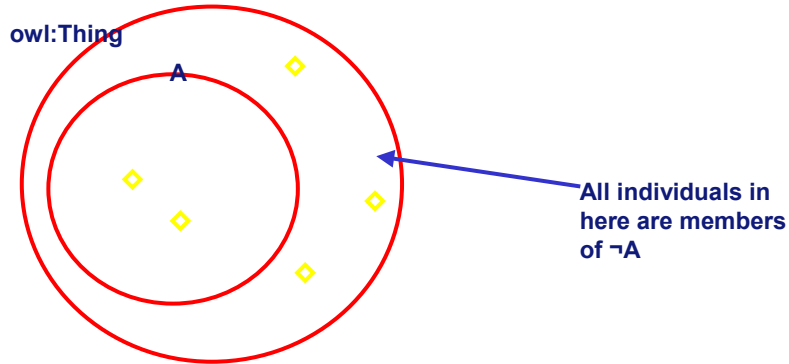


Table of contents

- Reasoning with OWL ontologies
 - Consistency checking
 - Disjointness
 - Restrictions
 - Primitive and Defined classes
 - Polyhierarchies (multiple classifications)
 - Untangling
 - Alternative definitions for a class (Vegetarian Pizzas: only vegetarian toppings, no meat or fish toppings or not a MeatyPizza?)
 - Union classes and covering axioms
 - The Open World Assumption (closure)
 - Negation in OWL
 - **Elephant Traps** – Common modelling errors
 - Functional properties
 - Intersection classes
 - Universal restrictions
- Using an ontology API to deal with OWL ontologies

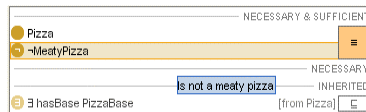
ComplementOf Classes

- ▶ aka "Negation" "Not"
- ▶ Not Something
- ▶ \neg Something

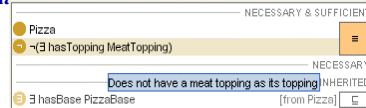


ComplementOf Classes

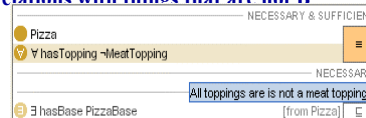
- Commonly used to model 3 things:
 - A is any C that is not B



- A does not have some relation with B



- A only has relations with things that are not B



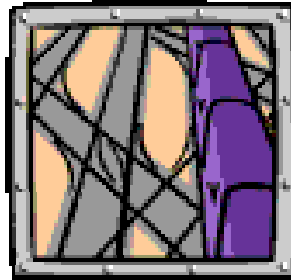
Define Vegetarian Pizza and Classify



**Define a Vegetarian topping
using negation**

“a pizza with no meat (or fish) toppings”?

“a pizza that is not a MeatyPizza”?



Elephant Traps

- **Common Errors in OWL generally include:**
 - **Disjoint misuse**
 - Often used on defined classes by mistake
 - **Confusing AllValuesFrom and SomeValuesFrom**
 - Some doesn't imply only, and only doesn't imply some
 - **Forgetting to close class descriptions**
 - **Incorrect expectations of Domain and Range defined for properties**
 - **Incorrect use of Functional Properties**
 - **Using intersection (AND) instead of union (OR), where the members of the intersection are disjoint**
 - **Negation and Open World Assumption**

Property Characteristics

- **Inverses**
 - If property p has inverse property q , and p relates A-B, then it can be inferred q relates B-A
- **Functional**
 - If property p relates A-B then all relations along p relate A-B (m..1 relation)
(B could also be a datatype value)
- **Inverse Functional**
 - The inverse of the property is functional
- **Symmetric**
 - If a property relates A-B then it also relates B-A
- **Transitive**
 - If a property relates A-B and B-C then it relates A-C

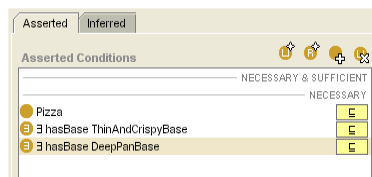
Functional Properties

- An individual can only have relationships with at most one other individual along a functional property

eg Setting hasBase to Functional means:

“Every Pizza can have at most one PizzaBase”

Description of DoubleBasePizza:



Asserted Conditions	
NECESSARY & SUFFICIENT	
NECESSARY	
Pizza	
∃ hasBase ThinAndCrispyBase	E
∃ hasBase DeepPanBase	E

- The reasoner finds this inconsistent
- It looks like the interface is warning us that we can't use the property more than once, but actually...

Trap: Functional Property Misuse

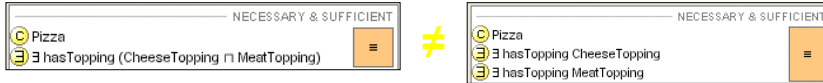
- If a property is functional and is used in several Existential restrictions on a class, the reasoner will infer that the filler classes must overlap
- If any of the fillers are disjoint from each other then this cannot be the case and therefore causes an inconsistency
- If they are not, no inconsistency is found

Intersection



- People often ask what the difference is between
 - 2 existential restrictions (which are, by default, in an intersection in the interface)
 - Using a single restriction with a filler containing both classes

Trap: Intersection

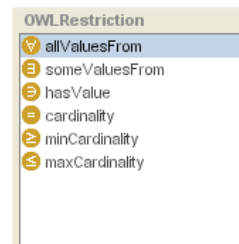


There are 2 problems:

1. Often we paraphrase “AND” when we logically mean “OR”
The filler “CheeseTopping AND MeatTopping” cannot contain any individuals as they are disjoint, and is therefore inconsistent
2. If we correct this to OR, it is still wrong as we’ve got a class description that can be fulfilled by a Pizza with a single topping – either Cheese or Meat. If we had 2 existential restrictions, there would have to be at least 2 (disjoint) toppings

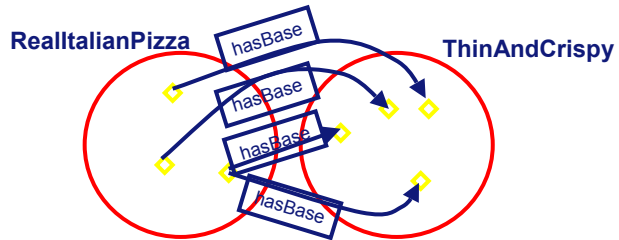
Universal Restrictions

- Example
 - “RealItalianPizzas only have bases that are ThinAndCrispy”
 - A Universal Restriction is added just like an Existential one, but the restriction type is different
 - For now, this can be primitive – you can make it defined if you like



What does this mean?

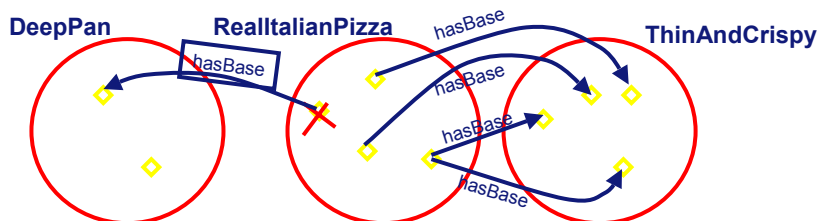
- ▶ We have created a restriction: \forall hasBase **ThinAndCrispy** on Class **RealltalianPizza** as a necessary condition



- ▶ “If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy**”

What does this mean?

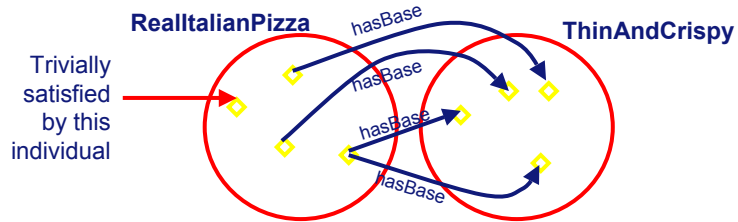
- ▶ We have created a restriction: \forall hasBase **ThinAndCrispy** on Class **RealltalianPizza** as a necessary condition



- ▶ “No individual of the **RealltalianPizza** class can have a base from a class other than **ThinAndCrispy**”
- ▶ NB. DeepPan and ThinAndCrispy are disjoint

Trap: Universal Restrictions

► If we had not already inherited: \exists hasBase **PizzaBase** from Class **Pizza** the following could hold



► “If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy**, or no hasBase relationship at all”

► Universal Restrictions by themselves do not state “at least one”

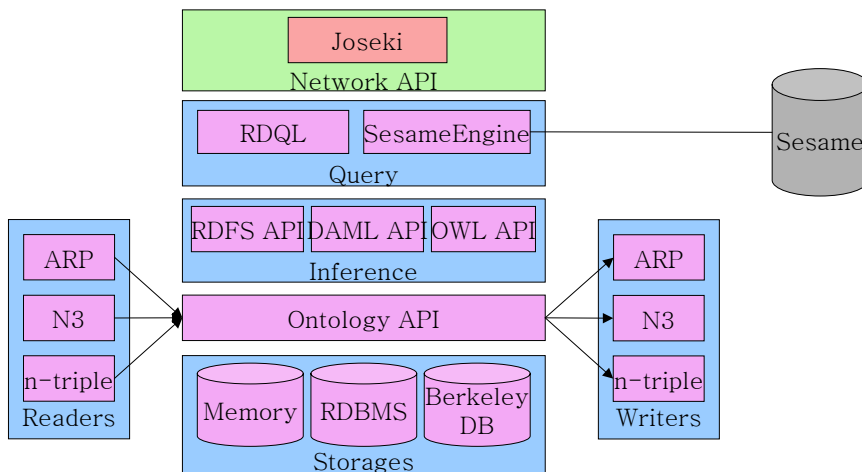
Table of contents

- Reasoning with OWL ontologies
 - ...
- Using an ontology API to deal with OWL ontologies
 - General introduction and architecture
 - Access to classes and properties
 - Access to class hierarchies
 - Access to instances
 - Programmatic use of reasoners

Introduction

- A Java Framework for RDF, DAML and OWL
- Developed by Brian McBride of HP Labs
- Derived from [SiRPAC](#)
- Can create, parse, navigate and search RDF, DAML and OWL model
- Current Jena release: 2.3
- Available at <http://jena.sourceforge.net>

Jena Architecture



Jena API Structure

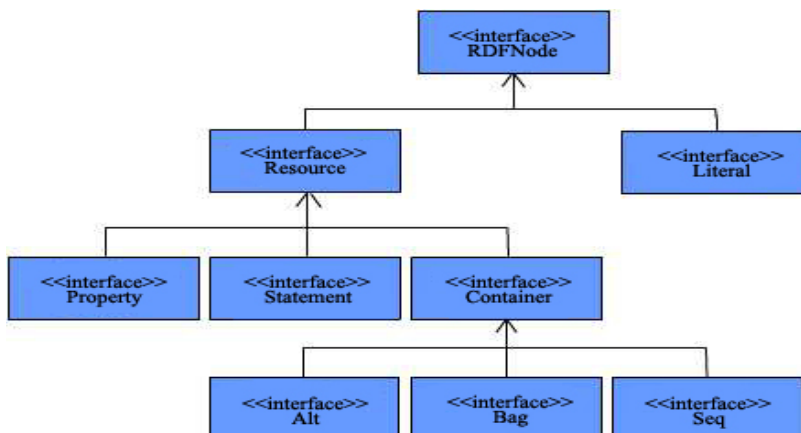
- **Statement-Centric View**
 - Convenient for manipulating graphs as a whole

```
graph.add(graph1)
      .add(graph2)
      .add(graph3)
      .write(outputStream);
```

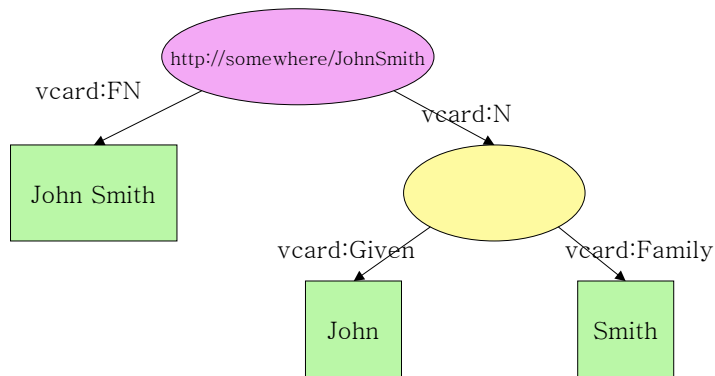
- **Frame-Centric View**
 - Analogous to the OOP paradigm
 - Convenient for navigating a graph or manipulating individual resources

```
article.getProperty(publishedIn)
      .getObject()
      .getProperty(DC.title)
      .getString();
```

Jena API Structure



Ontology Instance Example



RDF Ontology Instance

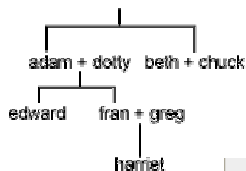
```

Model model = ModelFactory.createDefaultModel();
Resource jsmith =
model.createResource("http://somewhere/johnsmith")
.addProperty(VCARD.FN, "John Smith")
.addProperty(VCARD.N, model.createResource()
.addProperty(VCARD.Given, "John")
.addProperty(VCARD.Family, "Smith"));
model.write(new PrintWriter(System.out));
    
```

```

<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
  >
  <rdf:Description rdf:nodeID='A0'>
    <vcard:Given>John</vcard:Given>
    <vcard:Family>Smith</vcard:Family>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/johnsmith'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID='A0' />
  </rdf:Description>
</rdf:RDF>
    
```

Create resources



```
// URI declarations
String familyUri = "http://family/";
String relationshipUri = "http://purl.org/vocab/relationship/";

// Create an empty Model
Model model = ModelFactory.createDefaultModel();

// Create a Resource for each family member, identified by their URI
Resource adam = model.createResource(familyUri+"adam");
Resource beith = model.createResource(familyUri+"beith");
Resource chuck = model.createResource(familyUri+"chuck");
Resource dotty = model.createResource(familyUri+"dotty");
// and so on for other family members

// Create properties for the different types of relationship to represent
Property childOf = model.createProperty(relationshipUri,"childOf");
Property parentOf = model.createProperty(relationshipUri,"parentOf");
Property siblingOf = model.createProperty(relationshipUri,"siblingOf");
Property spouseOf = model.createProperty(relationshipUri,"spouseOf");

// Add properties to adam describing relationships to other family members
adam.addProperty(siblingOf,beith);
adam.addProperty(spouseOf,dotty);
adam.addProperty(parentOf,edward);

// Can also create statements directly . . .
Statement statement = model.createStatement(adam,parentOf,fran);

// but remember to add the created statement to the model
model.add(statement);
```

Querying a model

```
// List everyone in the model who has a child:
ResIterator parents = model.listSubjectsWithProperty(parentOf);

// Because subjects of statements are Resources, the method returned a ResIterator
while (parents.hasNext()) {

    // ResIterator has a typed nextResource() method
    Resource person = parents.nextResource();

    // Print the URI of the resource
    System.out.println(person.getURI());
}

// Can also find all the parents by getting the objects of all "childOf" statements
// Objects of statements could be Resources or literals, so the Iterator returned
// contains RDFNodes
NodeIterator moreParents = model.listObjectsOfProperty(childOf);

// To find all the siblings of a specific person, the model itself can be queried
NodeIterator siblings = model.listObjectsOfProperty(edward, siblingOf);

// But it's more elegant to ask the Resource directly
// This method yields an iterator over Statements
StmtIterator moreSiblings = edward.listProperties(siblingOf);
```

Using selectors to query a model

```
// Find the exact statement "adam is a spouse of dotty"
model.listStatements(adam,spouseOf,dotty);

// Find all statements with adam as the subject and dotty as the object
model.listStatements(adam,null,dotty);

// Find any statements made about adam
model.listStatements(adam,null,null);

// Find any statement with the siblingOf property
model.listStatements(null,siblingOf,null);
```

Importing and persisting models

- **Not all applications will start with an empty model.**
 - Commonly, a model will be populated from existing data at startup.
- **Solutions**
 - 1. Use `Model.write()` to serialise the model to the filesystem, and `Model.read()` to deserialise it on startup.
 - 2. Use persistent models, which are continually and transparently persisted to a backing store. The database engines currently supported are PostgreSQL, Oracle, and MySQL.
- **Steps (solution 2)**
 - **Instantiate the MySQL driver class**
 - **Create a `DBConnection` instance, with parameters:**
 - ID and password of the user to log in to the database as.
 - Database URL, which contains the name of the MySQL database for Jena to use, in the form "jdbc:mysql://localhost/dbname".
 - Database type, which for MySQL is "MySQL".
 - **Use the `DBConnection` instance with Jena's `ModelFactory` to create the database-backed model.**

Importing and persisting models

```
// Instantiate the MySQL driver
Class.forName("com.mysql.jdbc.Driver");

// Create a database connection object
DBConnection connection = new DBConnection(DB_URL, DB_USER, DB_PASSWORD, DB_TYPE);

// Get a ModelMaker for database-backed models
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);

// Create a new model named "wordnet." Setting the second parameter to "true" causes an
// AlreadyExistsException to be thrown if the db already has a model with this name
Model wordnetModel = maker.createModel("wordnet", true);

// Start a database transaction. Without one, each statement will be auto-committed
// as it is added, which slows down the model import significantly.
model.begin();

// For each wordnet model: . . .
InputStream in = this.getClass().getClassLoader().getResourceAsStream(filename);
model.read(in, null);

// Commit the database transaction
model.commit();
```

Now that the wordnet model is populated, you can access it later with a call to `ModelMaker.openModel("wordnet", true)`;

OWL Models

- **OWL ontologies are treated as a special type of RDF models, `OntModel`.**
- **It's possible to add ontological statements to an existing data model, or merge an ontology model with a data model using `Model.union()`.**

```
// Make a new model to act as an OWL ontology for WordNet
OntModel wnOntology = ModelFactory.createOntologyModel();

// Use OntModel's convenience method to describe
// WordNet's hyponymOf property as transitive
wnOntology.createTransitiveProperty(WordnetVocab.hyponymOf.getURI());

// Alternatively, just add a statement to the underlying model to express that
// hyponymOf is of type TransitiveProperty
wnOntology.add(WordnetVocab.hyponymOf, RDF.type, OWL.TransitiveProperty);
```


Using a reasoner with Jena

- Given an ontology and a model, Jena's inference engine (OWLReasoner) can derive additional statements that the model doesn't express explicitly.
- Steps
 - Get an OWLReasoner from the ReasonerRegistry.
ReasonerRegistry.getOWLReasoner() returns an OWL reasoner in its standard configuration, which is fine for a simple case.
 - Bind the reasoner to the ontology. This operation returns a reasoner ready to apply the ontology's rules.
 - Use the bound reasoner to create an InfModel from the WordNet model.
- The inference model can be treated just like any other Model instance.

```
// Get a reference to the WordNet plants model
ModelMaker maker = ModelFactory.createModelIRDBMaker(connection);
Model model = maker.openModel("wordnet-plants", true);

// Create an OWL reasoner
Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();

// Bind the reasoner to the WordNet ontology model
Reasoner wrReasoner = owlReasoner.bindSchema(wrOntology);

// Use the reasoner to create an inference model
InfModel infModel = ModelFactory.createInfModel(wrReasoner, model);

// Set the inference model as the source of the query
query.setSource(infModel);

// Execute the query as normal
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```



RDQL Syntax

- **SELECT**
 - Specify the variable to be returned
 - SELECT ?x, ?y
- **FROM**
 - Indicates the RDF source to be queried
 - FROM <doc.rdf>, <http://example.com/sample.rdf>
- **WHERE**
 - The most important part of the RDQL expression
 - Indicate constraints that RDF triples (subject, predicate, object)
 - WHERE (?x,<foo:has>,<?y>), (?y,<foo:color>,<?z>)
- **AND**
 - Specifies the Boolean expressions
 - AND ?z=="blue"
- **USING**
 - declares all the namespaces
 - USING foo for <http://foo.org/properties#>, col for <http://props.com/catalog#>

```
SELECT vars
FROM documents
WHERE Expressions
AND Filters
USING Namespace declarations
```



RDQL

- Jena's `com.hp.hpl.jena.rdql` package contains all of the classes and interfaces needed to use RDQL.
- Steps
 - Create an RDQL query as a `String`, and pass it to the constructor of `Query`.
 - It's usual to explicitly set the model to use as the source for the query, unless otherwise specified with a `FROM` clause in the RDQL itself.
 - Once a `Query` is prepared, a `QueryEngine` can be created from it, and the query executed.

```
// Create a new query passing a string containing the RDQL to execute
Query query = new Query(queryString);

// Set the model to run the query against
query.setSource(model);

// Use the query to create a query engine
QueryEngine qe = new QueryEngine(query);

// Use the query engine to execute the query
QueryResults results = qe.exec();
```

RDQL

- `QueryEngine.exec()` returns an object that implements `java.util.Iterator`. Its `next()` method returns `ResultBinding` objects.
- All of the variables used in the query can be obtained from the `ResultBinding` by name, regardless of whether they were part of the `SELECT` clause.

```
SELECT
  ?definition
WHERE
  (?concept, <wn:wordForm, "domestic dog"),
  (?concept, <wn:glossaryEntry, ?definition)
USING
  wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>;
```

```
// Execute a query
QueryResults results = qe.exec();

// Loop over the results
while (results.hasNext()) {
  ResultBinding binding = (ResultBinding)results.next();

  // Print the literal value of the "definition" variable
  RDFNode definition = (RDFNode) binding.get("definition");
  System.out.println(definition.toString());

  // Get the RDF resource used in the query
  Resource concept = (Resource)binding.get("concept");

  // Query the concept directly to find other wordforms it has
  List wordForms = concept.listObjectsOfProperty(wordForm);
}
```



How can we implement ontologies? Reasoners and Ontology APIs

Asunción Gómez-Pérez
Mariano Fernández-López
Oscar Corcho

asun@fi.upm.es, mfernandez.eps@ceu.es, ocorcho@cs.man.ac.uk

Grupo de Ontologías
Laboratorio de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n.
28660 Boadilla del Monte, Madrid, Spain