OWL 2

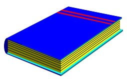
Oscar Corcho, María del Carmen Suárez de Figueroa Baonza, Oscar Muñoz García
{ocorcho,mcsuarez,omunoz}@fi.upm.es
http://www.oeg-upm.net/

Ontological Engineering Group
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*

Capítulo 4: Ontology languages



Baader F, McGuinness D, Nardi D, Patel-Schneider P (2003)
The Description Logic Handbook: Theory, implementation and applications.
Cambridge University Press, Cambridge, United Kingdom

W3C OWL Working Group (2009) OWL2 Web Ontology Language Document Overview.
http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

Dean M, Schreiber G (2004) *OWL Web Ontology Language Reference.* W3C Recommendation.
http://www.w3.org/TR/owl-ref/

Jena web site:	*http://jena.sourceforge.net/*
Jena API:	*http://jena.sourceforge.net/tutorial/RDF_API/*
Jena tutorials:	*http://www.ibm.com/developerworks/xml/library/j-jena/index.html*
	http://www.xml.com/pub/a/2001/05/23/jena.html

Pellet:	http://clarkparsia.com/pellet
RACER:	http://www.racer-systems.com/
FaCT++:	http://owl.man.ac.uk/factplusplus/
HermIT:	http://hermit-reasoner.com/

Table of Contents

What doesn't RDFS give us?

- **RDFS is too weak to describe resources in sufficient detail**
  - No localised range and domain constraints
    - Can't say that the range of hasEducationalMaterial is Slides when applied to TheoreticalSession and Code when applied to HandsonSession
      - TheoreticalSession hasEducationalMaterial Slides
      - HandsonSession hasEducationalMaterial Code
  - No existence/cardinality constraints
    - Can't say:
      - Sessions must have some EducationalMaterial
      - Sessions have at least one Presenter
  - No boolean operators
    - Can't say:
      - Or / not
  - No transitive, inverse or symmetrical properties
    - Can't say that presents is the inverse property of isPresentedBy

Description Logics

- **A family of logic based Knowledge Representation formalisms**
  - Descendants of semantic networks and KL-ONE
  - Describe domain in terms of concepts (classes), roles (relationships) and individuals
    - Specific languages characterised by the constructors and axioms used to assert knowledge about classes, roles and individuals.
    - Example: ALC (the least expressive language in DL that is propositionally closed)
      - Constructors: boolean (and, or, not)
      - Role restrictions

- **Distinguished by:**
  - Model theoretic semantics
    - Decidable fragments of FOL
    - Closely related to Propositional Modal & Dynamic Logics
  - Provision of inference services
    - Sound and complete decision procedures for key problems
    - Implemented systems (highly optimised)

Structure of DL Ontologies

- **A DL ontology can be divided into two parts:**
  - **Tbox** (Terminological KB): a set of axioms that describe the structure of a domain :
    - Doctor $\subseteq$ Person
    - Person $\subseteq$ Man $\cup$ Woman
    - HappyFather $\subseteq$ Man $\cap$ $\forall$hasDescendant.(Doctor $\cup$ $\forall$hasDescendant.Doctor)
  - **Abox** (Assertional KB): a set of axioms that describe a specific situation :
    - John $\in$ HappyFather
    - hasDescendant (John, Mary)

Most common constructors in class definitions

- **Intersection: $C_1 \cap \ldots \cap C_n$** — **Human $\cap$ Male**
- **Union: $C_1 \cup \ldots \cup C_n$** — **Doctor $\cup$ Lawyer**
- **Negation: $\neg C$** — **$\neg$Male**
- **Nominals: $\{x_1\} \cup \ldots \cup \{x_n\}$** — **$\{john\} \cup \ldots \cup \{mary\}$**
- **Universal restriction: $\forall P.C$** — **$\forall$hasChild.Doctor**
- **Existential restriction: $\exists P.C$** — **$\exists$hasChild.Lawyer**
- **Maximum cardinality: $\leq nP.C$** — **$\leq$3hasChild.Doctor**
- **Minimum cardinality: $\geq nP.C$** — **$\geq$1hasChild.Male**
- **Specific Value: $\exists P.\{x\}$** — **$\exists$hasColleague.{Matthew}**

- **Nesting of constructors can be arbitrarily complex**
  - Person $\cap$ $\forall$hasChild.(Doctor $\cup$ $\exists$hasChild.Doctor)
- **Lots of redundancy**
  - $A \cup B$ is equivalent to $\neg(\neg A \cap \neg B)$
  - $\exists P.C$ is equivalent to $\neg\forall P. \neg C$

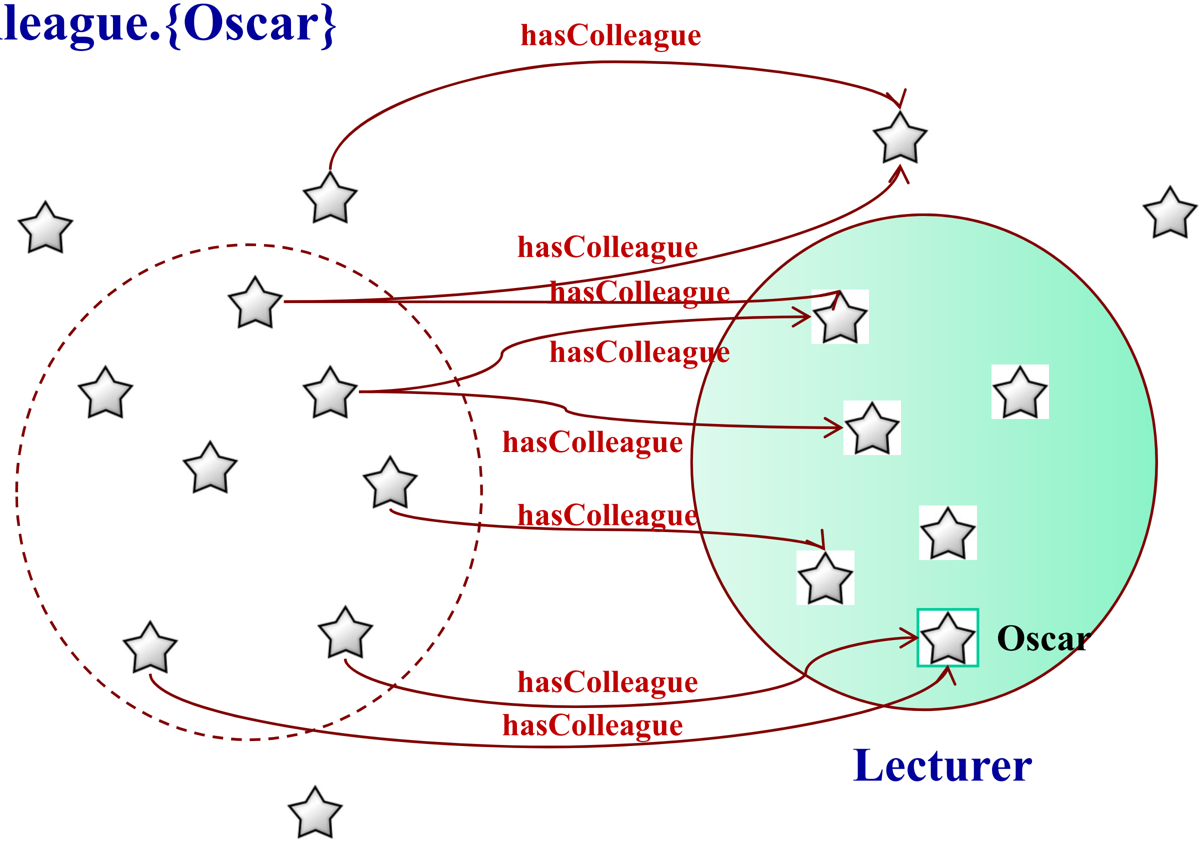
Description Logics

Understand the meaning of universal and existential restrictions

- Decide which is the set that we are defining with different expressions, taking into account Open and Close World Assumptions

Do we understand these constructors?

- $\exists$hasColleague.Lecturer
- $\forall$hasColleague.Lecturer
- $\exists$hasColleague.{Oscar}



Most common axioms in definitions

- **Classes**

Subclass	$C1 \subseteq C2$	$Human \subseteq Animal \cap Biped$
Equivalence	$C1 \equiv C2$	$Man \equiv Human \cap Male$
Disjointness	$C1 \cap C2 \subseteq \bot$	$Male \cap Female \subseteq \bot$

- **Properties/roles**

Subproperty	$P1 \subseteq P2$	$hasDaughter \subseteq hasChild$
Equivalence	$P1 \equiv P2$	$cost \equiv price$
Inverse	$P1 \equiv P2^{-}$	$hasChild \equiv hasParent^{-}$
Transitive	$P+ \subseteq P$	$ancestor+ \subseteq ancestor$
Functional	$T \subseteq \leq 1P$	$T \subseteq \leq 1hasMother$
InverseFunctional	$T \subseteq \leq 1P^{-}$	$T \subseteq \leq 1hasPassportID^{-}$

  – And also… Reflexive, Irreflexive, Symmetric, Asymmetric

- **Individuals**

Equivalence	$\{x1\} \equiv \{x2\}$	{oeg:OscarCorcho} $\equiv$ {img:Oscar}
Different	$\{x1\} \equiv \neg\{x2\}$	{john} $\equiv \neg${peter}

  – Negative object property and datatype property assertions

	$\neg\{P\ a1\ a2\}$	$\neg$ {hasChild john peter}

- **Most axioms are reducible to inclusion ($\cup$)**
  – $C \equiv D$ iff both $C \subseteq D$ and $D \subseteq C$; C disjoint D iff $C \subseteq \neg D$

DL constructors and DL languages

Construct	Syntax	Language			
Concept	A				
Role name	R				
Intersection	$C \cap D$	FL_0			
Value restriction	$\forall R.C$		FL^-		
Limited existential quantification	$\exists R$			AL	
Top or Universal	$\top$				S[14]
Bottom	$\bot$				
Atomic negation	$\neg A$				
Negation[15]	$\neg C$			C	
Union	$C \cup D$			U	
Existential restriction	$\exists R.C$			E	
Number restrictions	$(\geq n\ R)\ (\leq n\ R)$			N	
Nominals	$\{a_1 \ldots a_n\}$			O	
Role hierarchy	$R \subseteq S$			H	
Inverse role	R^-			I	
Qualified number restriction	$(\geq n\ R.C)\ (\leq n\ R.C)$			Q	

OWL1 is SHOIN(D+)
OWL2 is SROIQ(D+)

→ {Colombia, Argentina, México, ...} → MercoSur countries

→ ≤2 hasChild.Female, ≥1 hasParent.Male

[12] Names previously used for Description Logics were: terminological knowledge representation languages, concept languages, term subsumption languages, and KL-ONE-based knowledge representation languages.
[13] In this table, we use A to refer to atomic concepts (concepts that are the basis for building other concepts), C and D to any concept definition, R to atomic roles and S to role definitions. FL is used for structural DL languages and AL for attributive languages (Baader et al., 2003).
[14] S is the name used for the language ALC_{R+}, which is composed of ALC plus transitive roles.
[15] ALC and ALCUE are equivalent languages, since union (U) and existential restriction (E) can be represented using negation (C).

Other:

Concrete datatypes: hasAge.(<21)

Transitive roles: hasChild* (descendant)

Role composition: hasParent o hasBrother (uncle)

Some basic DL modelling guidelines

• **X must be Y, X is an Y that...**	**→ $X \subseteq Y$**
• **X is exactly Y, X is the Y that...**	**→ $X \equiv Y$**
• **X is not Y (*not the same as X is whatever it is not Y*)**	**→ $X \subseteq \neg Y$**
• **X and Y are disjoint**	**→ $X \cap Y \subseteq \bot$**
• **X is Y or Z**	**→ $X \subseteq Y \cup Z$**
• **X is Y for which property P has only instances of Z as values**	**→ $X \subseteq Y \cap (\forall P.Z)$**
• **X is Y for which property P has at least an instance of Z as a value**	**→ $X \subseteq Y \cap (\exists P.Z)$**
• **X is Y for which property P has at most 2 values**	**→ $X \subseteq Y \cap (\leq 2.P)$**
• **Individual X is a Y**	**→ $X \in Y$**

Description Logics Formalisation

Develop a sample ontology in the domain of people, pets, vehicles, and newspapers

- Understand how to formalise knowledge in description logics

Chunk 1. Formalize in DL

1. **Concept definitions**:

   Grass and trees must be plants. Leaves are parts of a tree but there are other parts of a tree that are not leaves. A dog must eat bones, at least. A sheep is an animal that must only eat grass. A giraffe is an animal that must only eat leaves. A mad cow is a cow that eats brains that can be part of a sheep.

2. **Restrictions**:

   Animals or part of animals are disjoint with plants or parts of plants.

3. **Properties**:

   Eats is applied to animals. Its inverse is eaten_by.

4. **Individuals**:

   Tom.
   Flossie is a cow.
   Rex is a dog and is a pet of Mick.
   Fido is a dog.
   Tibbs is a cat.

Chunk 2. Formalize in DL

1. **Concept definitions**:
   Bicycles, buses, cars, lorries, trucks and vans are vehicles. There are several types of companies: bus companies and haulage companies.
   An elderly person must be adult. A kid is (exactly) a person who is young. A man is a person who is male and is adult. A woman is a person who is female and is adult. A grown up is a person who is an adult. And old lady is a person who is elderly and female. Old ladies must have some animal as pets and all their pets are cats.

2. **Restrictions**:
   Youngs are not adults, and adults are not youngs.

3. **Properties**:
   Has mother and has father are subproperties of has parent.

4. **Individuals**:
   Kevin is a person.
   Fred is a person who has a pet called Tibbs.
   Joe is a person who has at most one pet. He has a pet called Fido.
   Minnie is a female, elderly, who has a pet called Tom.

Chunk 3. Formalize in DL

1. Concept definitions:

   A magazine is a publication. Broadsheets and tabloids are newspapers. A quality broadsheet is a type of broadsheet. A red top is a type of tabloid. A newspaper is a publication that must be either a broadsheet or a tabloid.

   White van mans must read only tabloids.

2. Restrictions:

   Tabloids are not broadsheets, and broadsheets are not tabloids.

3. Properties:

   The only things that can be read are publications.

4. Individuals:

   Daily Mirror

   The Guardian and The Times are broadsheets

   The Sun is a tabloid

Chunk 4. Formalize in DL

1. **Concept definitions**:

   A pet is a pet of something. An animal must eat something. A vegetarian is an animal that does not eat animals nor parts of animals. Ducks, cats and tigers are animals. An animal lover is a person who has at least three pets. A pet owner is a person who has animal pets. A cat liker is a person who likes cats. A cat owner is a person who has cat pets. A dog liker is a person who likes dogs. A dog owner is a person who has dog pets.

2. **Restrictions**:

   Dogs are not cats, and cats are not dogs.

3. **Properties**:

   Has pet is defined between persons and animals. Its inverse is is_pet_of.

4. **Individuals**:

   Dewey, Huey, and Louie are ducks.
   Fluffy is a tiger.
   Walt is a person who has pets called Huey, Louie and Dewey.

Chunk 5. Formalize in DL

1. **Concept definitions**

   A driver must be adult. A driver is a person who drives vehicles. A lorry driver is a person who drives lorries. A haulage worker is who works for a haulage company or for part of a haulage company. A haulage truck driver is a person who drives trucks ans works for part of a haulage company. A van driver is a person who drives vans. A bus driver is a person who drives buses. A white van man is a man who drives white things and vans.

2. **Restrictions:**

   --

3. **Properties:**

   The service number is an integer property with no restricted domain

4. **Individuals:**

   Q123ABC is a van and a white thing.
   The42 is a bus whose service number is 42.
   Mick is a male who read Daily Mirror and drives Q123ABC.

Chunk 1. Formalisation in DL

$$grass \subseteq plant$$

$$tree \subseteq plant$$

$$leaf \subseteq \exists partOf.tree$$

$$dog \subseteq \exists eats.bone$$

$$sheep \subseteq animal \cap \forall eats.grass$$

$$giraffe \subseteq animal \cap \forall eats.leaf$$

$$madCow \equiv cow \cap \exists eats.(brain \cap \exists partOf.sheep)$$

$$(animal \cup \exists partOf.animal) \cap (plant \cup \exists partOf.plant) \subseteq \bot$$

Chunk 2. Formalisation in DL

$$bicycle \subseteq vehicle; bus \subseteq vehicle; car \subseteq vehicle; lorry \subseteq vehicle; truck \subseteq vehicle$$

$$busCompany \subseteq company; haulageCompany \subseteq company$$

$$elderly \subseteq person \cap adult$$

$$kid \equiv person \cap young$$

$$man \equiv person \cap male \cap adult$$

$$woman \equiv person \cap female \cap adult$$

$$grownUp \equiv person \cap adult$$

$$oldLady \equiv person \cap female \cap elderly$$

$$oldLady \subseteq \exists hasPet.animal \cap \forall hasPet.cat$$

$$young \cap adult \subseteq \perp$$

$$hasMother \subseteq hasParent$$

$$hasFather \subseteq hasParent$$

Chunk 3. Formalisation in DL

$$magazine \subseteq publication$$

$$broadsheet \subseteq newspaper$$

$$tabloid \subseteq newspaper$$

$$qualityBroadsheet \subseteq broadsheet$$

$$redTop \subseteq tabloid$$

$$newspaper \subseteq publication \cap (broadsheet \cup tabloid)$$

$$whiteVanMan \subseteq \forall reads.tabloid$$

$$tabloid \cap broadsheet \subseteq \perp$$

Chunk 4. Formalisation in DL

$$pet \equiv \exists isPetOf.\mathrm{T}$$

$$animal \subseteq \exists eats.\mathrm{T}$$

$$vegetarian \equiv animal \cap \forall eats.\neg animal \cap \forall eats.\neg(\exists partOf.animal)$$

$$duck \subseteq animal; cat \subseteq animal; tiger \subseteq animal$$

$$animalLover \equiv person \cap (\geq 3 hasPet)$$

$$petOwner \equiv person \cap \exists hasPet.animal$$

$$catLike \equiv person \cap \exists likes.cat; catOwner \equiv person \cap \exists hasPet.cat$$

$$dogLike \equiv person \cap \exists likes.dog; dogOwner \equiv person \cap \exists hasPet.dog$$

$$dog \cap cat \subseteq \perp$$

Chunk 5. Formalisation in DL

$$driver \subseteq adult$$

$$driver \equiv person \cap \exists drives.vehicle$$

$$lorryDriver \equiv person \cap \exists drives.lorry$$

$$haulageWorke \equiv \exists worksFor.(haulageCompany \cup \exists partOf.haulageCompany)$$

$$haulageTruckDriver \equiv person \cap \exists drives.truck \cap \exists worksFor.(\exists partOf.haulageCompany)$$

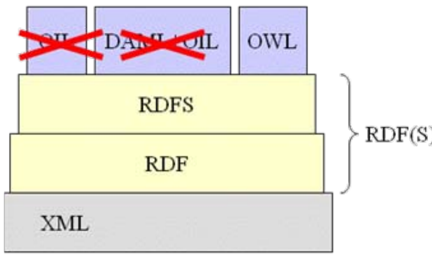
$$vanDriver \equiv person \cap \exists drives.van$$

$$busDriver \equiv person \cap \exists drives.bus$$

$$whiteVanMan \equiv man \cap \exists drives.(whiteThing \cap van)$$

Table of Contents

OWL (1.0 and 1.1)



February 2004

Web Ontology Language

Built on top of RDF(S) and renaming DAML+OIL primitives

Three layers:
- OWL Lite
  - A small subset of primitives
  - Easier for frame-based tools to transition to
- OWL DL
  - Description logic
  - Decidable reasoning
- OWL Full
  - RDF extension, allows metaclasses

Several syntaxes:
- Abstract syntax
- Manchester syntax
- RDF/XML

OWL 2 (I). New features

- **October 2009**

- **New features**
  - Syntactic sugar
    - Disjoint union of classes
  - New expressivity
    - Keys
    - Property chains
    - Richer datatypes, data ranges
    - Qualified cardinality restrictions
    - Asymmetric, reflexive, and disjoint properties
    - Enhanced annotation capabilities

- **New syntax**
  - OWL2 Manchester syntax

OWL 2 (II). Three new profiles

- **OWL2 EL**
  - Ontologies that define very large numbers of classes and/or properties,
  - Ontology consistency, class expression subsumption, and instance checking can be decided in polynomial time.
- **OWL2 QL**
  - Sound and complete query answering is in LOGSPACE (more precisely, in AC^0) with respect to the size of the data (assertions),
  - Provides many of the main features necessary to express conceptual models (UML class diagrams and ER diagrams).
  - It contains the intersection of RDFS and OWL 2 DL.
- **OWL2 RL**
  - Inspired by Description Logic Programs and pD*.
  - Syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies, and presenting a partial axiomatization of the OWL 2 RDF-Based Semantics in the form of first-order implications that can be used as the basis for such an implementation.
  - Scalable reasoning without sacrificing too much expressive power.
  - Designed for
    - OWL applications trading the full expressivity of the language for efficiency,
    - RDF(S) applications that need some added expressivity from OWL 2.

OWL: Most common constructors

Intersection:	$C_1 \cap ... \cap C_n$	intersectionOf	Human ∩ Male
Union:	$C_1 \cup ... \cup C_n$	unionOf	Doctor ∪ Lawyer
Negation:	$\neg C$	complementOf	¬Male
Nominals:	$\{x_1\} \cup ... \cup \{x_n\}$	oneOf	{john} ∪ ... ∪ {mary}
Universal restriction:	$\forall P.C$	allValuesFrom	∀hasChild.Doctor
Existential restriction:	$\exists P.C$	someValuesFrom	∃hasChild.Lawyer
Maximum cardinality:	$\leq nP[.C]$	maxCardinality (qualified or not)	≤3hasChild[.Doctor]
Minimum cardinality:	$\geq nP[.C]$	minCardinality (qualified or not)	≥1hasChild[.Male]
Exact cardinality:	$=nP[.C]$	exactCardinality (qualified or not)	=1hasMother[.Female]
Specific Value:	$\exists P.\{x\}$	hasValue	∃hasColleague.{Matthew}
Local reflexivity:	--	hasSelf	Narcisist ≡ Person ∩ hasSelf(loves)
Keys	--	hasKey	hasKey(Person, passportNumber, country)
Subclass	$C1 \subseteq C2$	subClassOf	Human ⊆ Animal ∩ Biped
Equivalence	$C1 \equiv C2$	equivalentClass	Man ≡ Human ∩ Male
Disjointness	$C1 \cap C2 \subseteq \perp$	disjointWith, AllDisjointClasses	Male ∩ Female ⊆ ⊥
DisjointUnion	$C \equiv C_1 \cup ... \cup C_n$ and $C_i \cap C_j \subseteq \perp$ forall $i \neq j$	disjointUnionOf	Person DisjointUnionOf (Man, Woman)

Metaclasses and annotations on axioms are also valid in OWL2, and declarations of classes have to provided.

Full list available in reference specs and in the Quick Reference Guide: http://www.w3.org/2007/OWL/refcard

OWL: Most common constructors

Subproperty	$P1 \subseteq P2$	subPropertyOf	$hasDaughter \subseteq hasChild$
Equivalence	$P1 \equiv P2$	equivalentProperty	$cost \equiv price$
DisjointProperties	$P1 \cap ... \cap Pn \subseteq \perp$	disjointObjectProperties	$hasDaughter \cap hasSon \subseteq \perp$
Inverse	$P1 \equiv P2-$	inverseOf	$hasChild \equiv hasParent-$
Transitive	$P+ \subseteq P$	TransitiveProperty	$ancestor+ \subseteq ancestor$
Functional	$T \subseteq \leq 1P$	FunctionalProperty	$T \subseteq \leq 1hasMother$
InverseFunctional	$T \subseteq \leq 1P-$	InverseFunctionalProperty	$T \subseteq \leq 1hasPassportID-$
Reflexive		ReflexiveProperty	
Irreflexive		IrreflexiveProperty	
Asymmetric		AsymmetricProperty	
Property chains	$P \equiv P1 \text{ o } ... \text{ o } Pn$	propertyChainAxiom	$hasUncle \subseteq hasFather \text{ o } hasBrothe$
Equivalence	$\{x1\} \equiv \{x2\}$	sameIndividualAs	{oeg:OscarCorcho}≡{img:Oscar}
Different	$\{x1\} \equiv \neg\{x2\}$	differentFrom, AllDifferent	$\{john\} \equiv \neg\{peter\}$
NegativePropertyAssertion		NegativeDataPropertyAssertion	¬{hasAge john 35}
		NegativeObjectPropertyAssertion	¬{hasChild john peter}

Besides, top and bottom object and datatype properties exist

Table of Contents

Basic Inference Tasks

- **Subsumption – check knowledge is correct (captures intuitions)**
  - Does C subsume D w.r.t. ontology O? (in *every* model I of O, $C^I \subseteq D^I$)
- **Equivalence – check knowledge is minimally redundant (no unintended synonyms)**
  - Is C equivalent to D w.r.t. O? (in *every* model I of O, $C^I = D^I$)
- **Consistency – check knowledge is meaningful (classes can have instances)**
  - Is C satisfiable w.r.t. O? (there exists *some* model I of O s.t. $C^I \neq \emptyset$)
- **Instantiation and querying**
  - Is x an instance of C w.r.t. O? (in *every* model I of O, $x^I \in C^I$)
  - Is (x,y) an instance of R w.r.t. O? (in *every* model I of O, $(x^I,y^I) \in R^I$)
- **All reducible to KB satisfiability or concept satisfiability w.r.t. a KB**
- **Can be decided using highly optimised tableaux reasoners**

Tableaux Algorithms

- **Try to prove satisfiability of a knowledge base**
- **How do they work**
  - They try to build a model of input concept C
    - Tree model property
      - If there is a model, then there is a tree shaped model
    - If no tree model can be found, then input concept unsatisfiable
  - Decompose C syntactically
    - Work on concepts in negation normal form (De Morgan's laws)
    - Use of tableaux expansion rules
    - If non-deterministic rules are applied, then there is search
  - Stop (and backtrack) if clash
    - E.g. A(x), ¬A(x)
  - Blocking (cycle check) ensures termination for more expressive logics

- **The algorithm finishes when no more rules can be applied or a conflict is detected**

Tableaux rules for ALC and for transitive roles

$x \bullet \{C_1 \sqcap C_2, \ldots\}$	$\rightarrow_{\sqcap}$	$x \bullet \{C_1 \sqcap C_2, C_1, C_2, \ldots\}$
$x \bullet \{C_1 \sqcup C_2, \ldots\}$	$\rightarrow_{\sqcup}$	$x \bullet \{C_1 \sqcup C_2, C, \ldots\}$ for $C \in \{C_1, C_2\}$
$x \bullet \{\exists R.C, \ldots\}$	$\rightarrow_{\exists}$	$x \bullet \{\exists R.C, \ldots\}$ $R \downarrow$ $y \bullet \{C\}$
$x \bullet \{\forall R.C, \ldots\}$ $R \downarrow$ $y \bullet \{\ldots\}$	$\rightarrow_{\forall}$	$x \bullet \{\forall R.C, \ldots\}$ $R \downarrow$ $y \bullet \{C, \ldots\}$
$x \bullet \{\forall R.C, \ldots\}$ $R \downarrow$ $y \bullet \{\ldots\}$	$\rightarrow_{\forall_+}$	$x \bullet \{\forall R.C, \ldots\}$ $R \downarrow$ $y \bullet \{\forall R.C, \ldots\}$

Tableaux example

- Example
  - $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)$

Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role

Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role

$$\mathcal{L}(w) = \{\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$$

w

Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role

$$\mathcal{L}(w) = \{\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$$

w

Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role

$$\mathcal{L}(w) = \{\exists S.C, \forall S.(\neg C \sqcup \neg D), \exists R.C, \forall R.(\exists R.C)\}$$

$$w$$

Tableaux Algorithm — Example

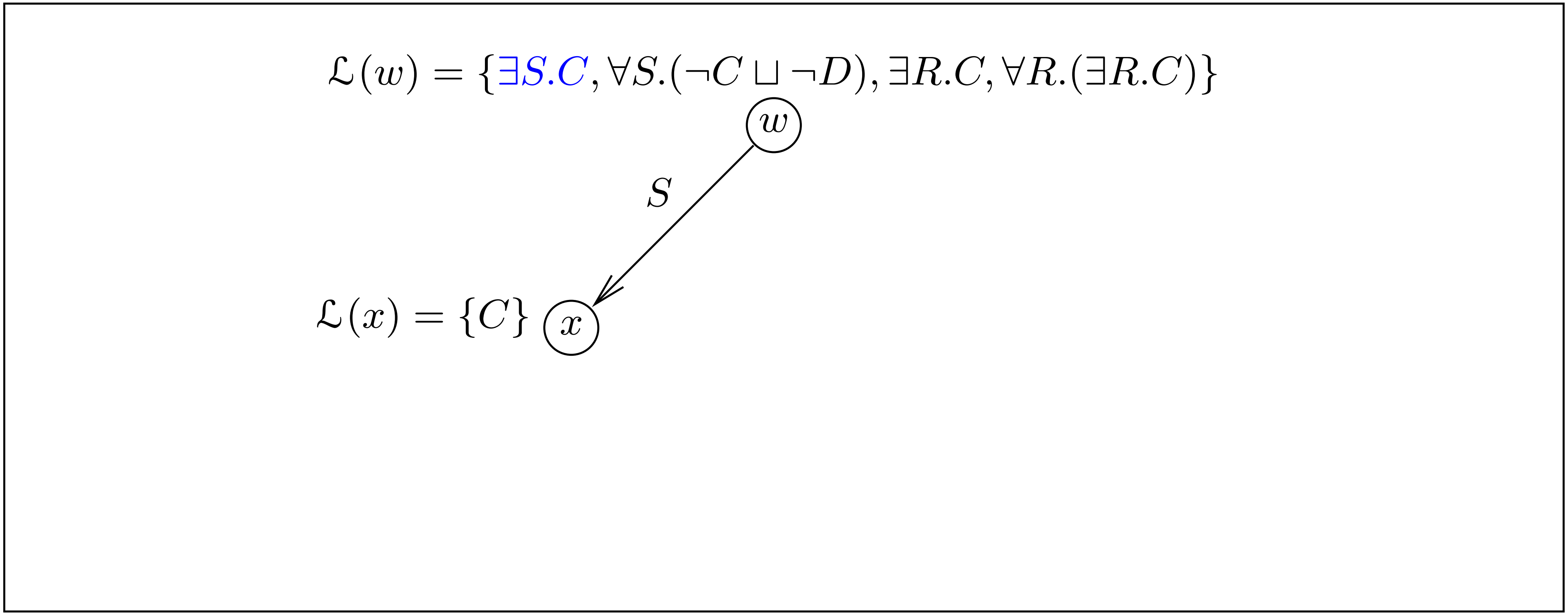
Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role

$$\mathcal{L}(w) = \{\exists S.C, \forall S.(\neg C \sqcup \neg D), \exists R.C, \forall R.(\exists R.C)\}$$

$$w$$

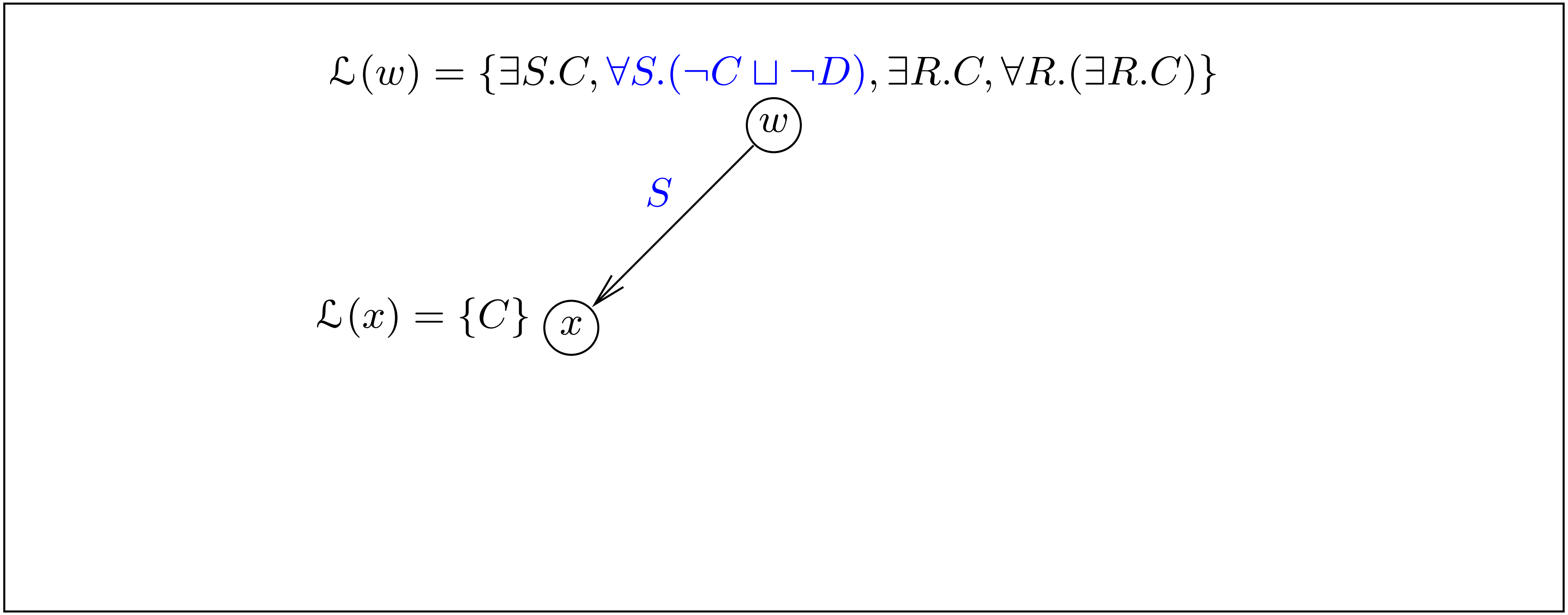
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



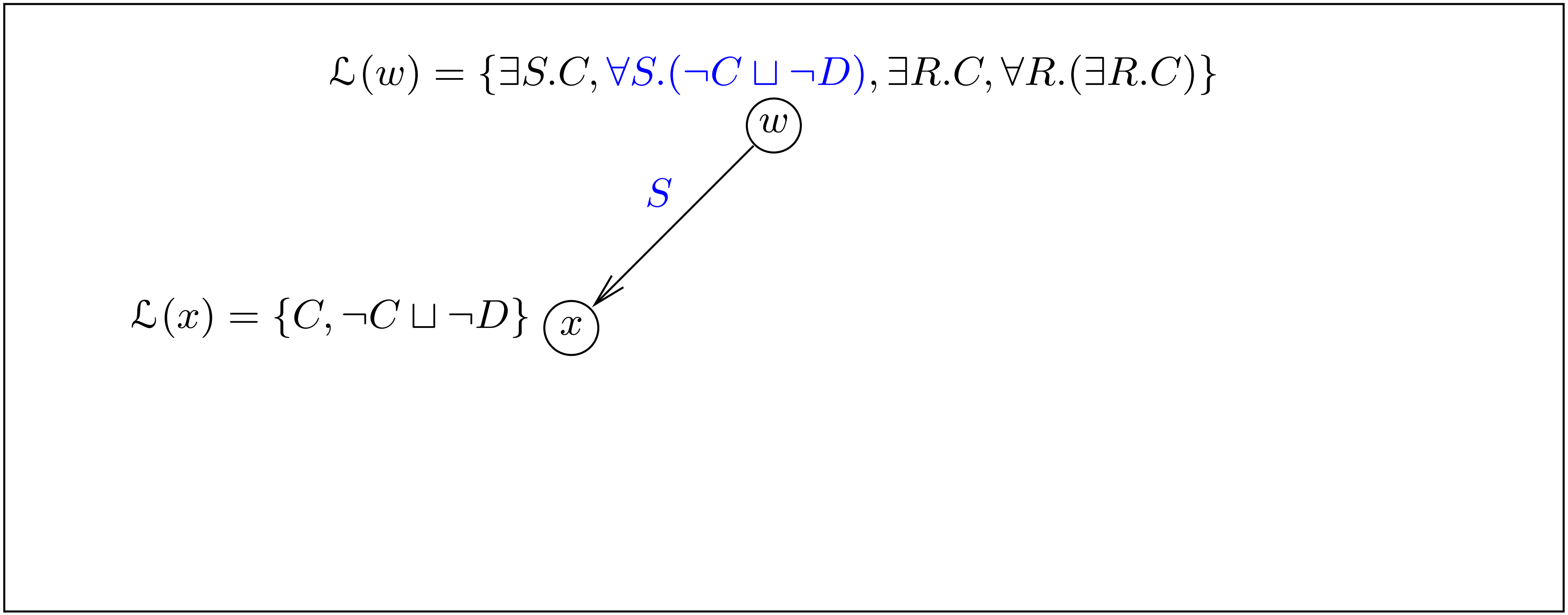
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



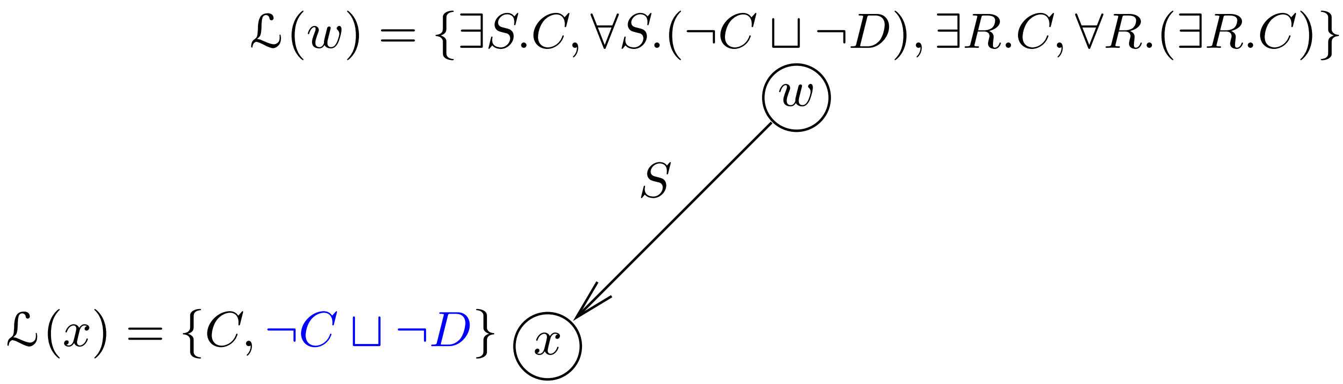
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



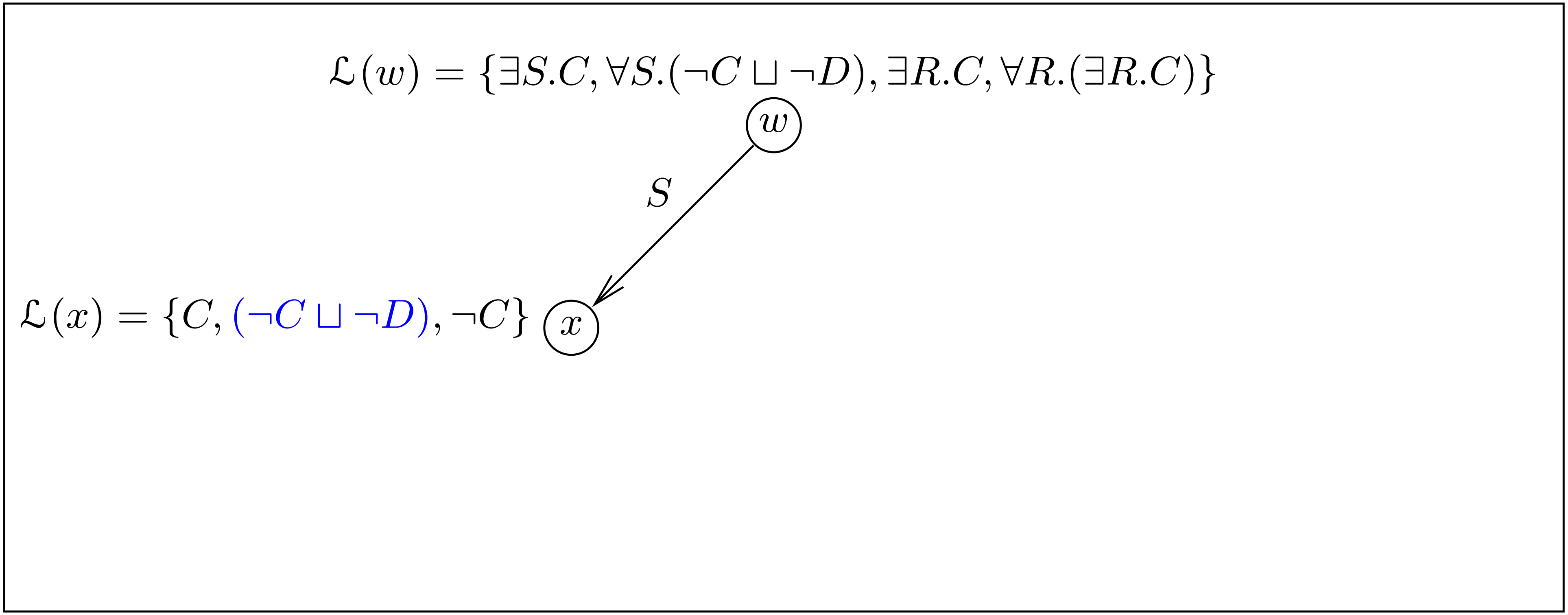
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



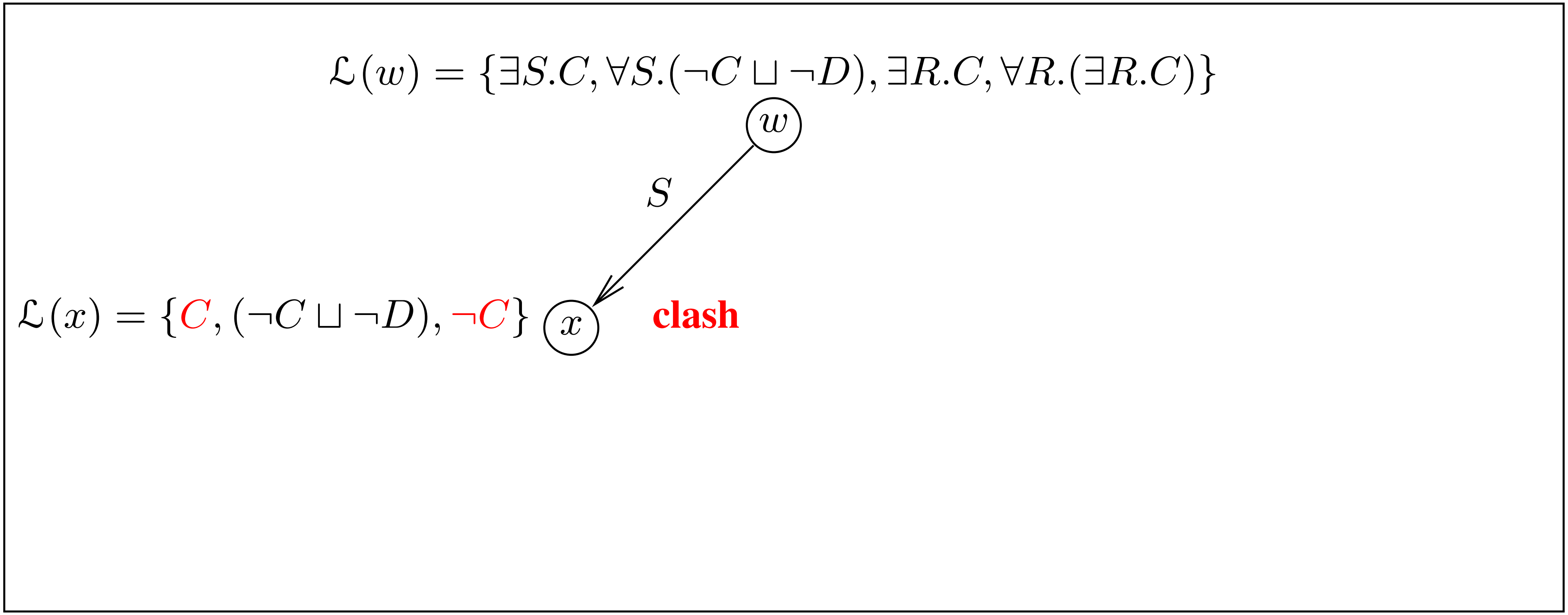
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



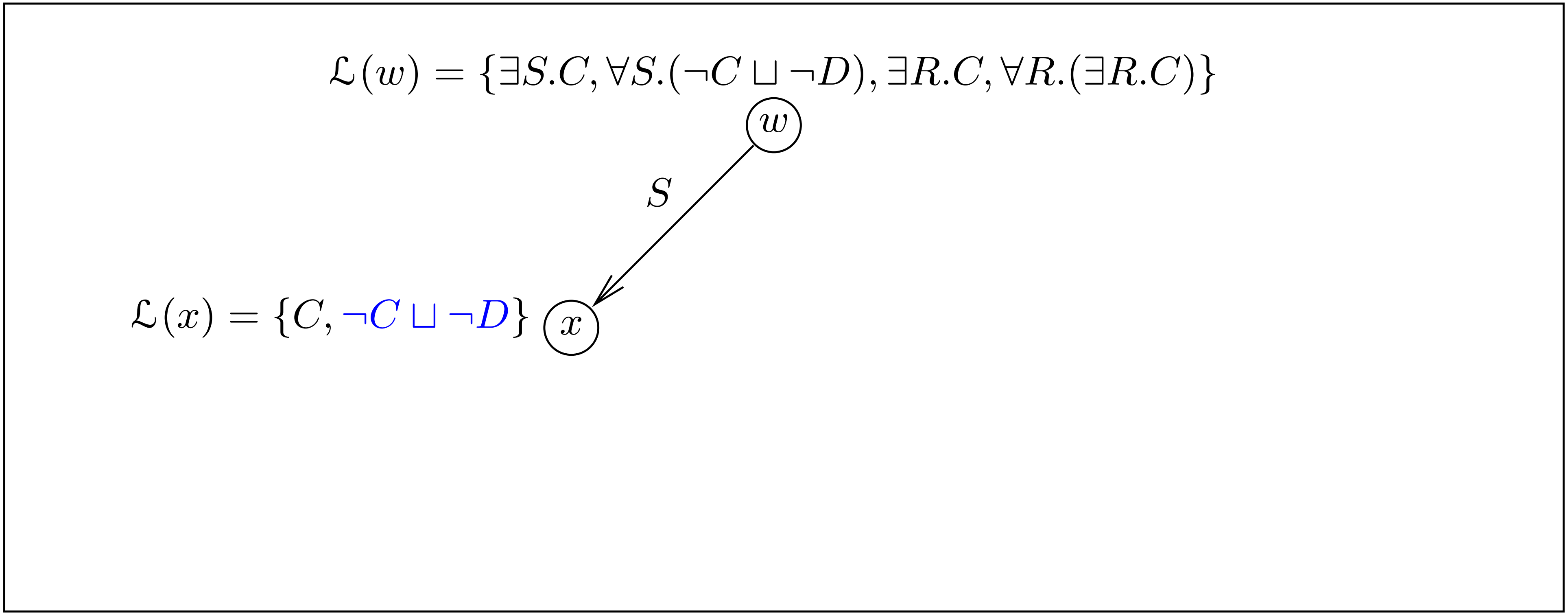
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



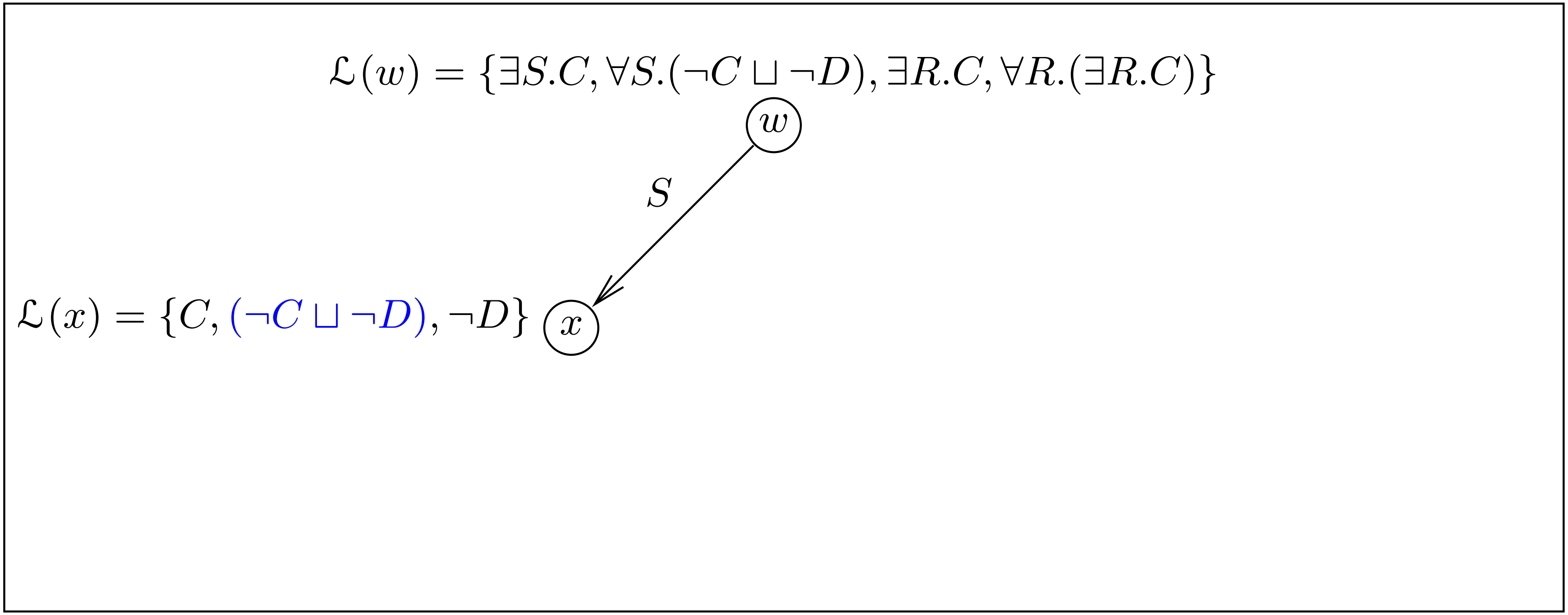
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



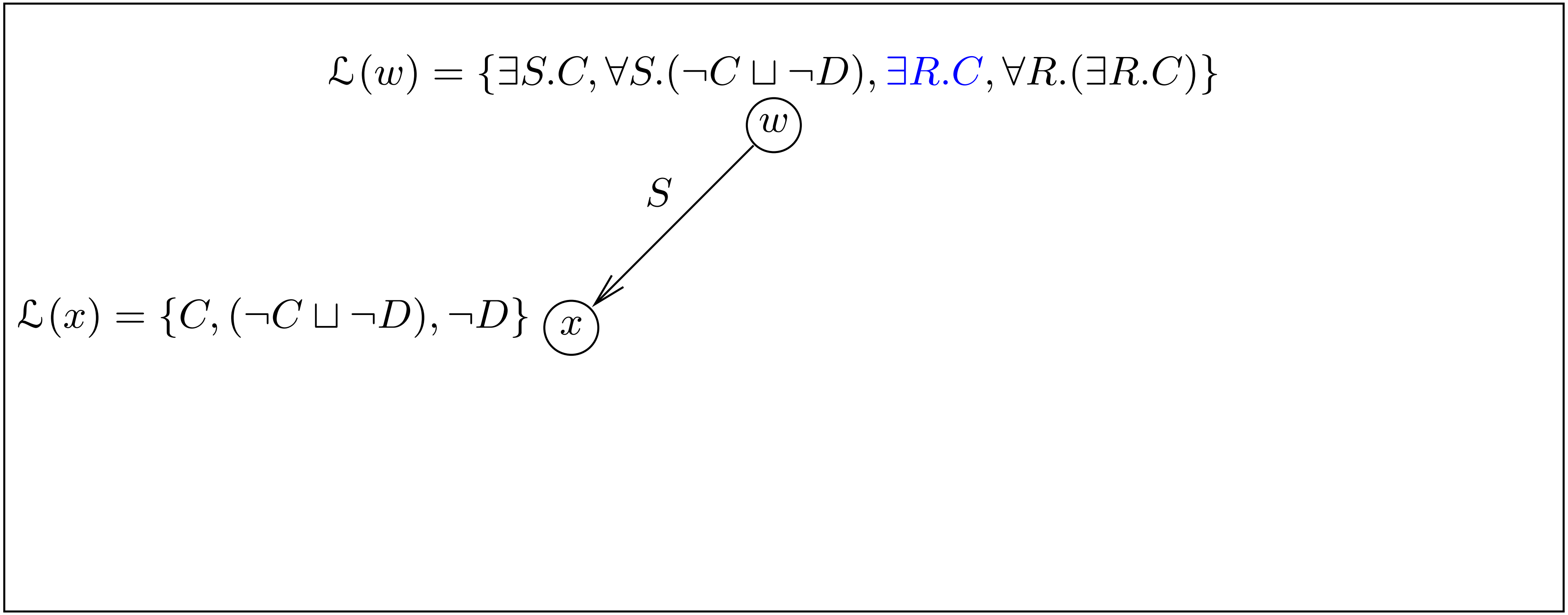
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



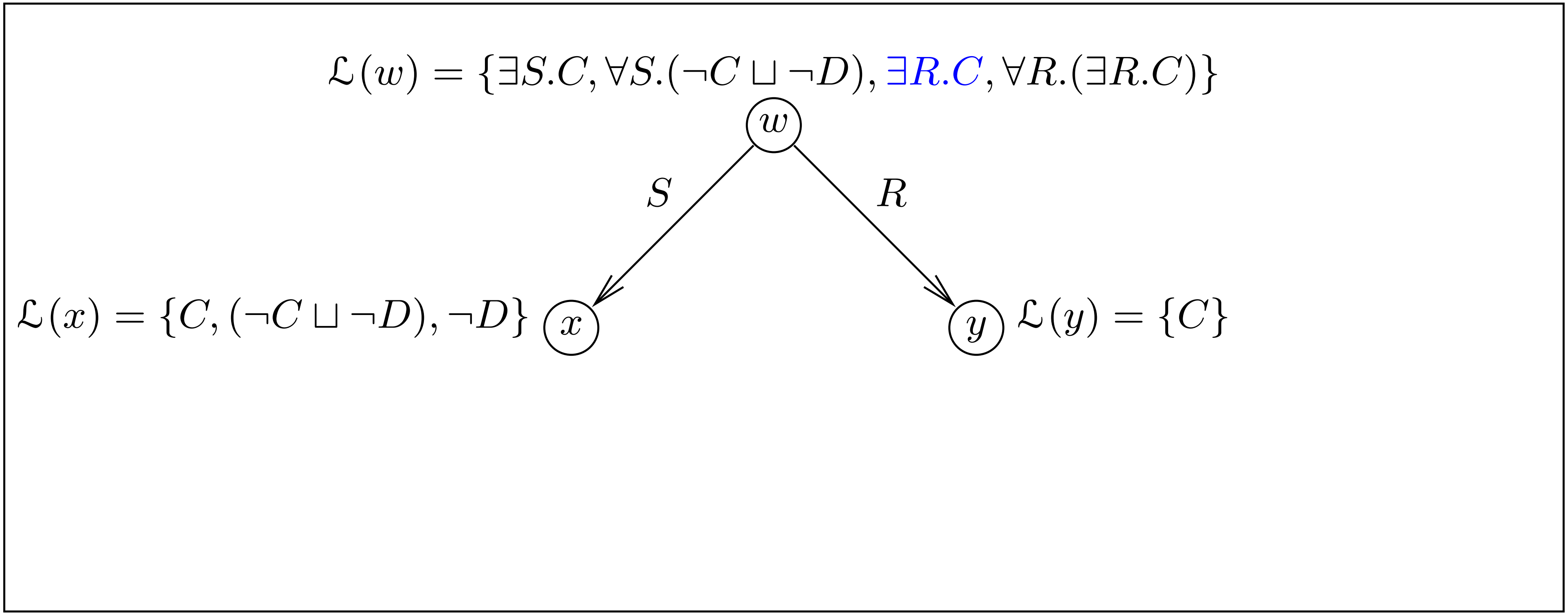
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



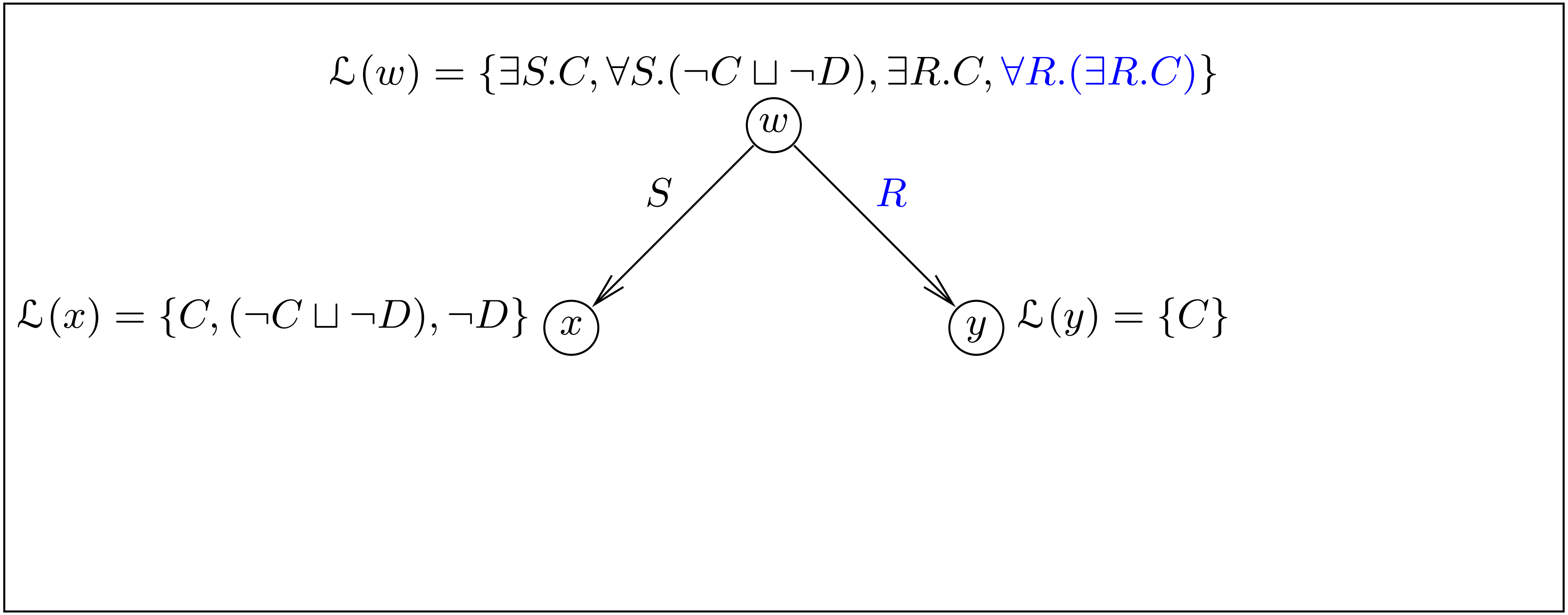
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



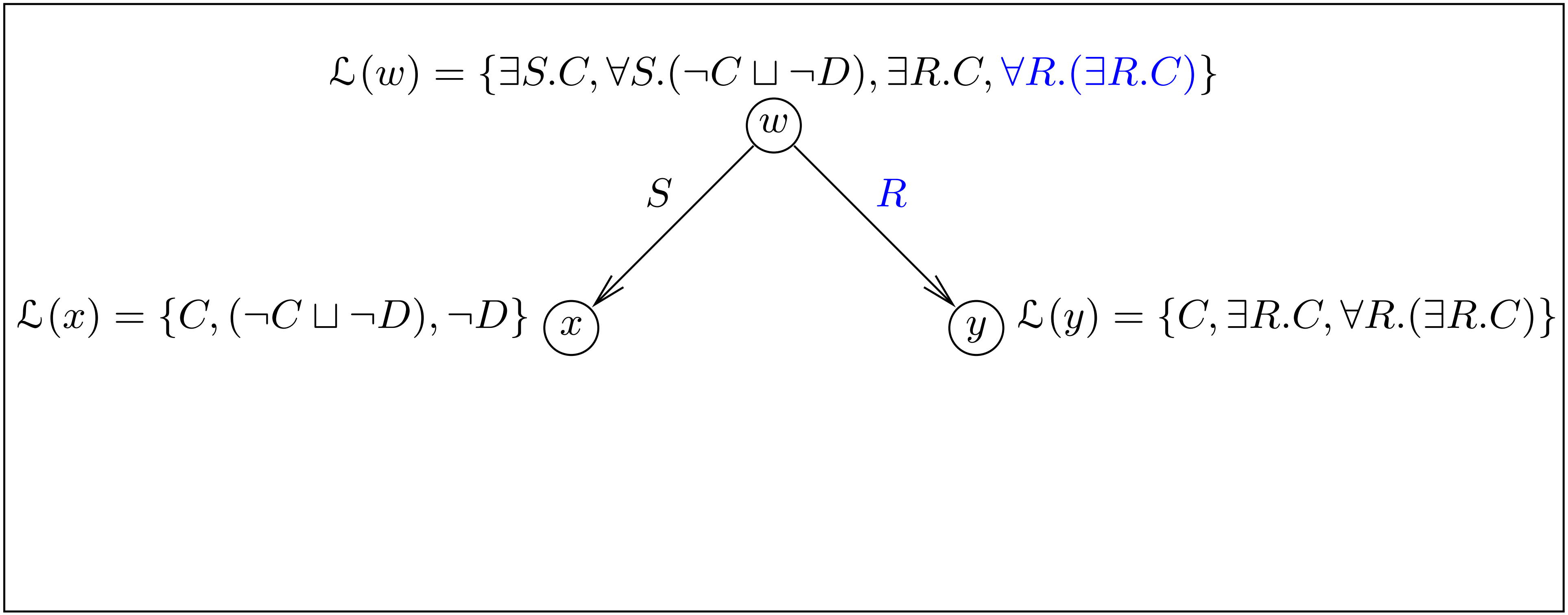
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



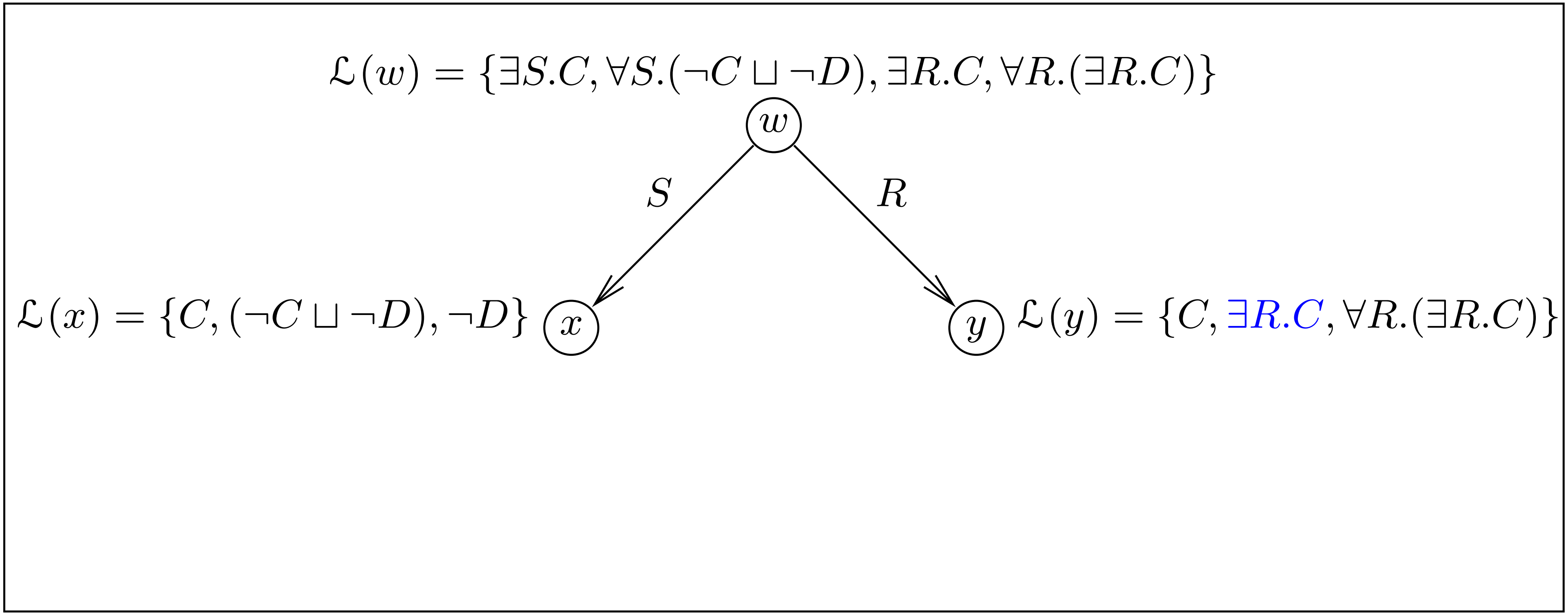
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



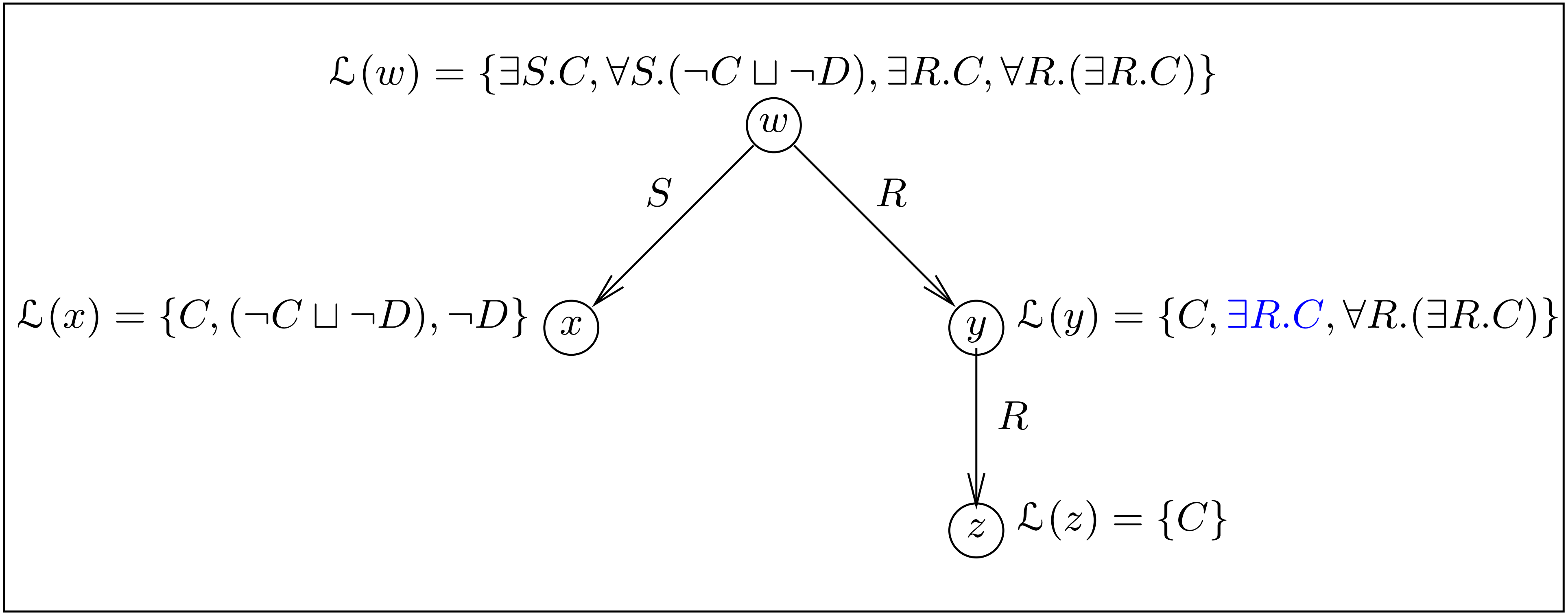
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



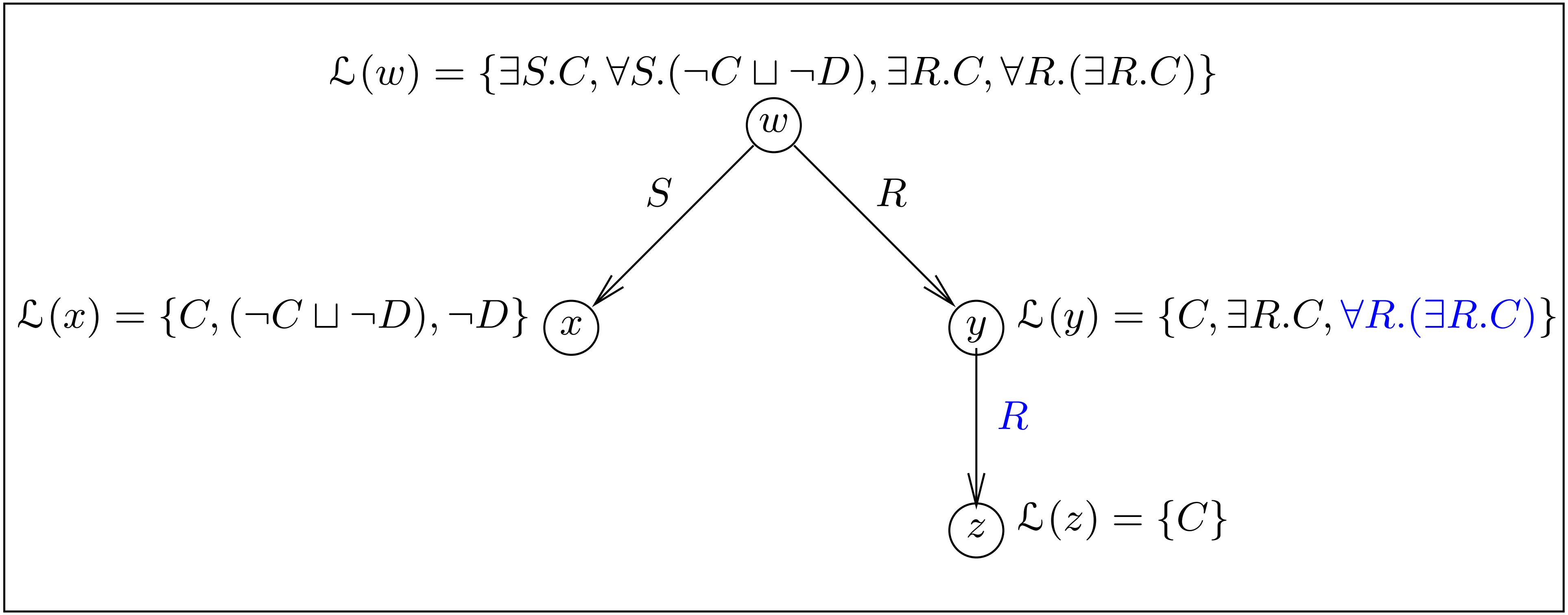
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



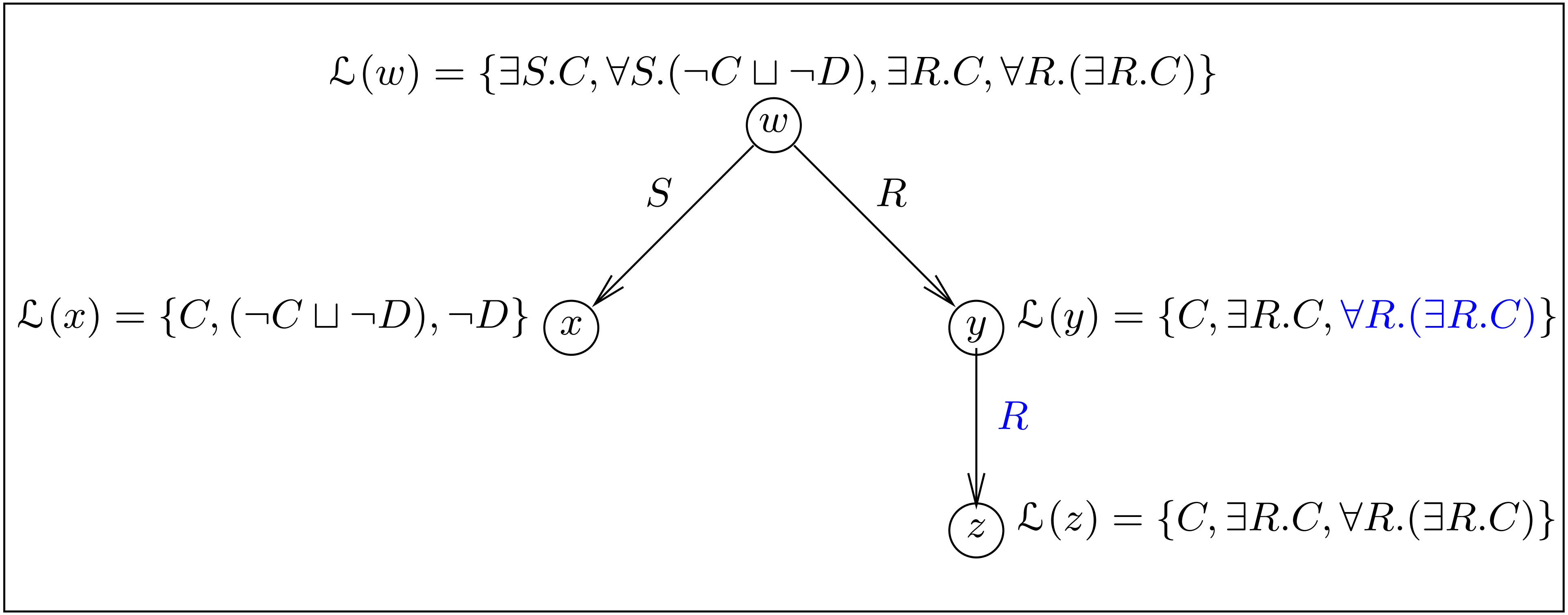
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



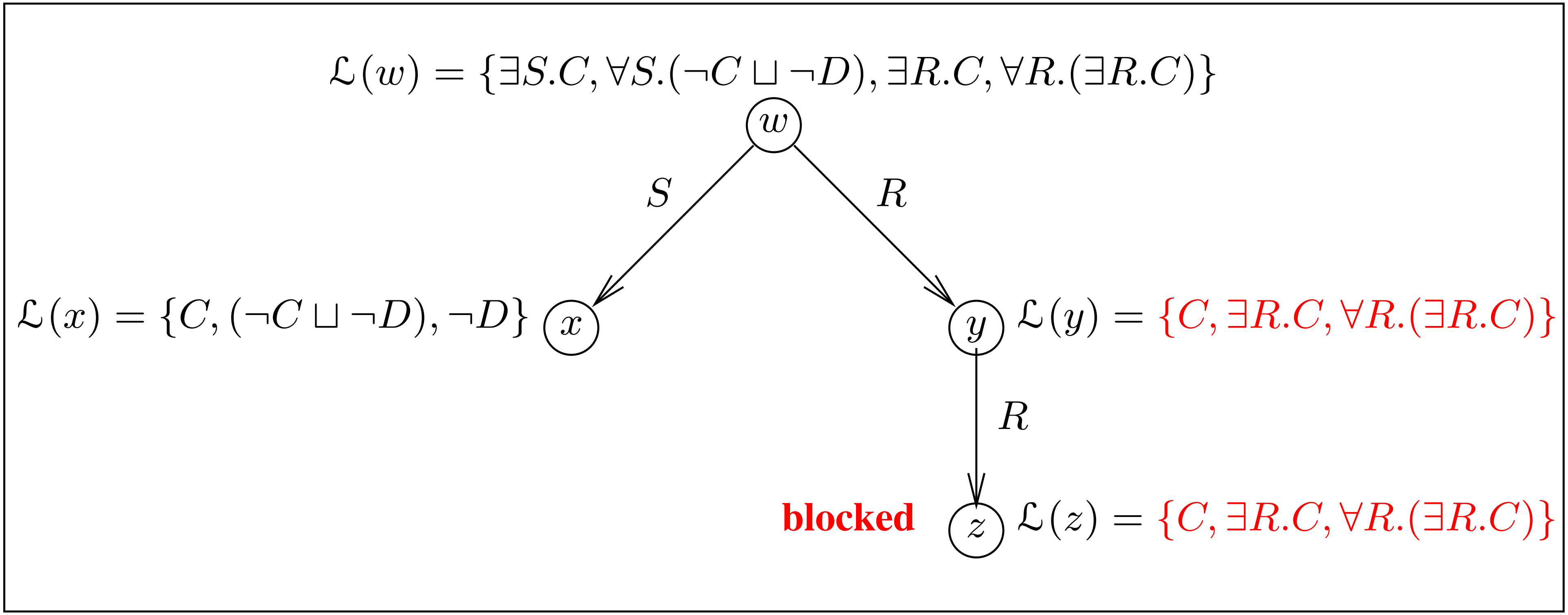
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



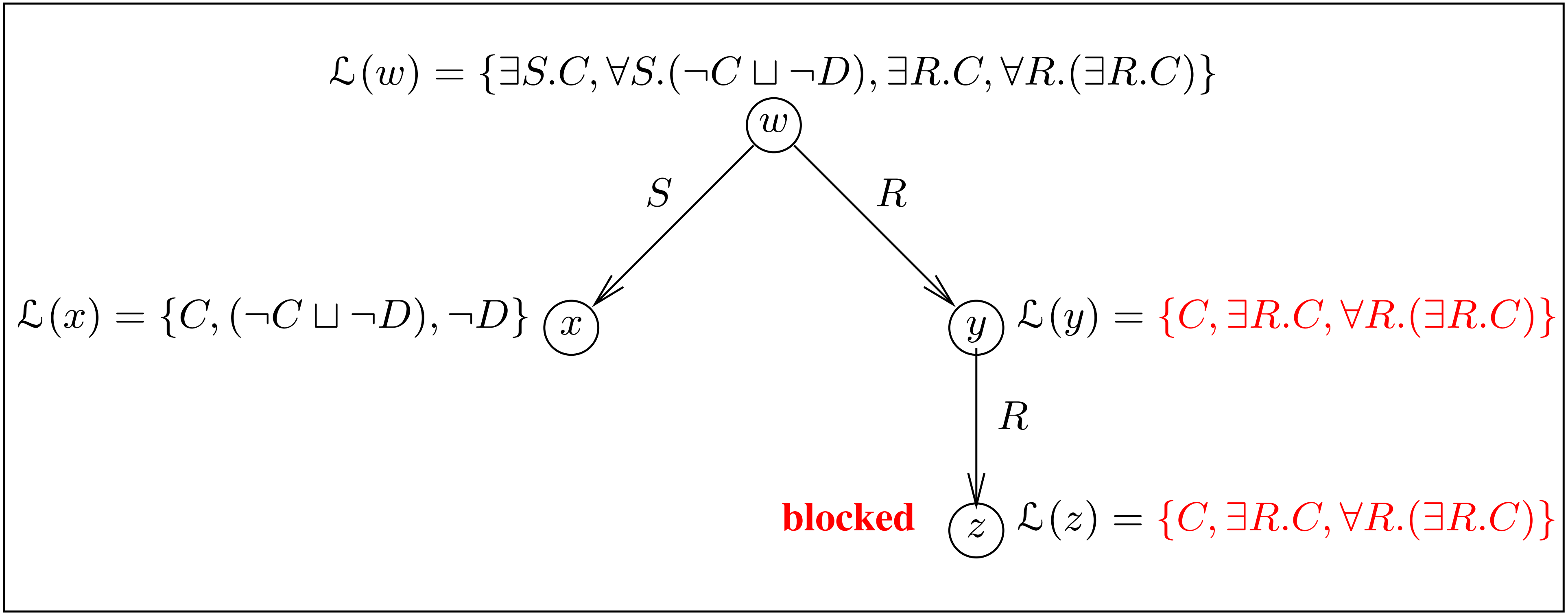
Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



Tableaux Algorithm — Example

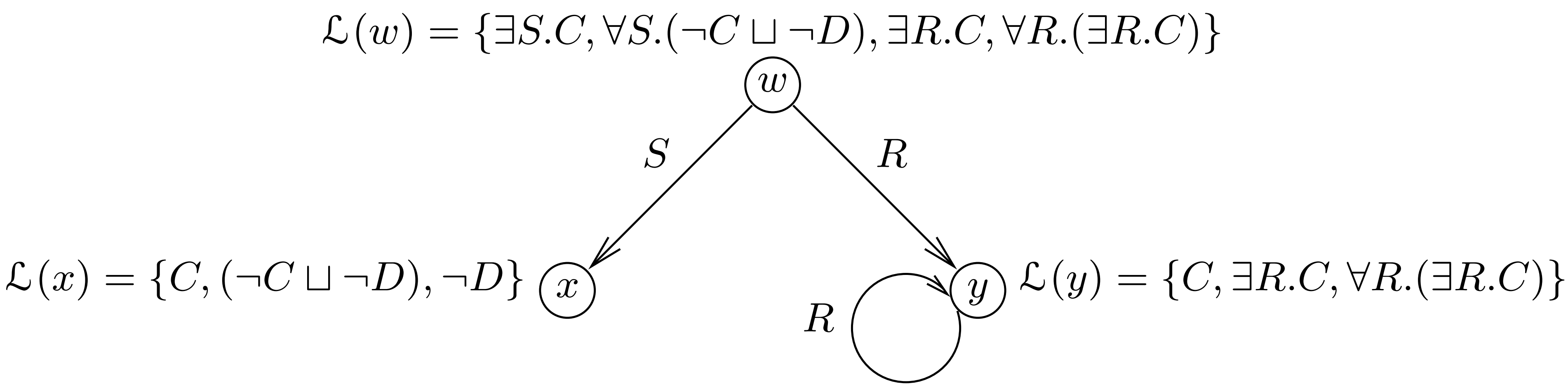
Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



Concept is **satisfiable**: $\mathbf{T}$ corresponds to **model**

Tableaux Algorithm — Example

Test satisfiability of $\exists S.C \sqcap \forall S.(\neg C \sqcup \neg D) \sqcap \exists R.C \sqcap \forall R.(\exists R.C)\}$ where R is a **transitive** role



Concept is **satisfiable**: $\mathbf{T}$ corresponds to **model**

Description Logics Reasoning with Tableaux

Use tableaux algorithms to determine whether the following formulae are satisfiable or not

Exercise 1: $\exists R.(\exists R.D) \wedge \exists S.\neg D \wedge \forall S.(\exists R.D)$

Exercise 2: $\exists R.(C \vee D) \wedge \forall R.\neg C \wedge \neg \exists R.D$

Description Logics Reasoning

Develop a sample ontology in the domain of people, pets, vehicles, and newspapers

- **Understand the basic reasoning mechanisms of description logics**
  - **Subsumption**
    - **Automatic classification: an ontology built collaboratively**
    - **Instance classification**
    - **Detecting redundancy**
  - **Consistency checking: unsatisfiable restrictions in a Tbox (are the classes coherent?)**

Interesting results (I). Automatic classification

And old lady is a person who is elderly and female.
Old ladies must have some animal as pets and all their pets are cats.

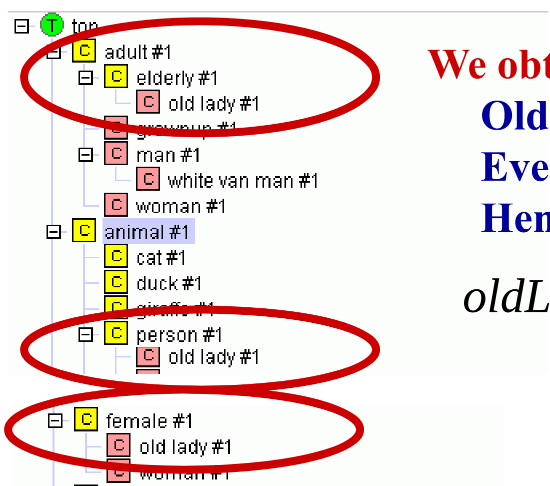
$$elderly \subseteq person \cap adult$$

$$woman \equiv person \cap female \cap adult$$

$$catOwner \equiv person \cap \exists hasPet.cat$$

$$oldLady \equiv person \cap female \cap elderly$$

$$oldLady \subseteq \exists hasPet.animal \cap \forall hasPet.cat$$



We obtain:

Old ladies must be women.
Every old lady must have a pet cat
Hence, every old lady must be a cat owner

$$oldLady \subseteq woman \cap elderly \cap catOwner$$

Interesting results (II). Instance classification

A pet owner is a person who has animal pets
Old ladies must have some animal as pets and all their pets are cats.
Has pet has domain person and range animal
Minnie is a female, elderly, who has a pet called Tom.

$$petOwner \equiv person \cap \exists hasPet.animal$$

$$oldLady \subseteq \exists hasPet.animal \cap \forall hasPet.cat$$

$$hasPet \subseteq (person, animal)$$

$$Minnie \in female \cap elderly$$

$$hasPet(Minnie, Tom)$$

We obtain:

- **Minnie is a person**
- **Hence, Minnie is an old lady**
- **Hence, Tom is a cat**

$$Minnie \in person; Tom \in animal$$

$$Minnie \in petOwner$$

$$Minnie \in oldLady$$

$$Tom \in cat$$

Interesting results (III). Instance classification and redundancy detection

An animal lover is a person who has at least three pets
Walt is a person who has pets called Huey, Louie and Dewey.

$$animalLover \equiv person \cap (\geq 3 hasPet)$$

$$Walt \in person$$

$$hasPet(Walt, Huey)$$

$$hasPet(Walt, Louie)$$

$$hasPet(Walt, Dewey)$$

We obtain:

Walt is an animal lover

Walt is a person is redundant

$$Walt \in animalLover$$

Interesting results (IV). Instance classification

A van is a type of vehicle
A driver must be adult
A driver is a person who drives vehicles
A white van man is a man who drives vans and white things
White van mans must read only tabloids
Q123ABC is a white thing and a van
Mick is a male who reads Daily Mirror and drives Q123ABC

$$van \subseteq vehicle$$

$$driver \subseteq adult$$

$$driver \equiv person \cap \exists drives.vehicle$$

$$whiteVanMan \equiv man \cap \exists drives.(van \cap whiteThing)$$

$$whiteVanMan \subseteq \forall reads.tabloid$$

$$Q123ABC \in whiteThing \cap van$$

$$Mick \in male$$

$$reads(Mick, DailyMirror)$$

$$drives(Mick, Q123ABC)$$

We obtain:

Mick is an adult
Mick is a white van man
Daily Mirror is a tabloid

$$Mick \in adult$$

$$Mick \in whiteVanMan$$

$$DailyMirror \in tabloid$$

Interesting results (V). Consistency checking

Cows are vegetarian.
A vegetarian is an animal that does not eat animals nor parts of animals.
A mad cow is a cow that eats brains that can be part of a sheep

$$cow \subseteq vegetarian$$

$$vegetarian \equiv animal \cap \forall eats.\neg animal \cap \forall eats.\neg(\exists partOf.animal))$$

$$madCow \equiv cow \cap \exists eats.(brain \cup \exists partOf.sheep)$$

$$(animal \cup \exists partOf.animal) \cap (plant \cup \exists partOf.plant) \subseteq \perp$$

We obtain:

Mad cow is unsatisfiable

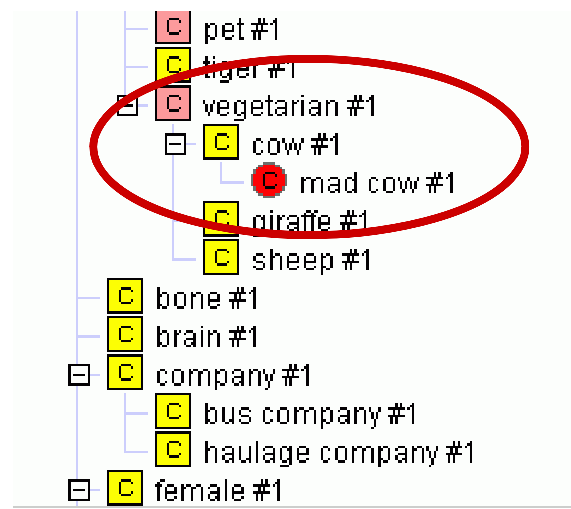


Table of Contents

Named Classes (I)

An ontology contains classes – indeed, the main building blocks of an OWL ontology are classes.

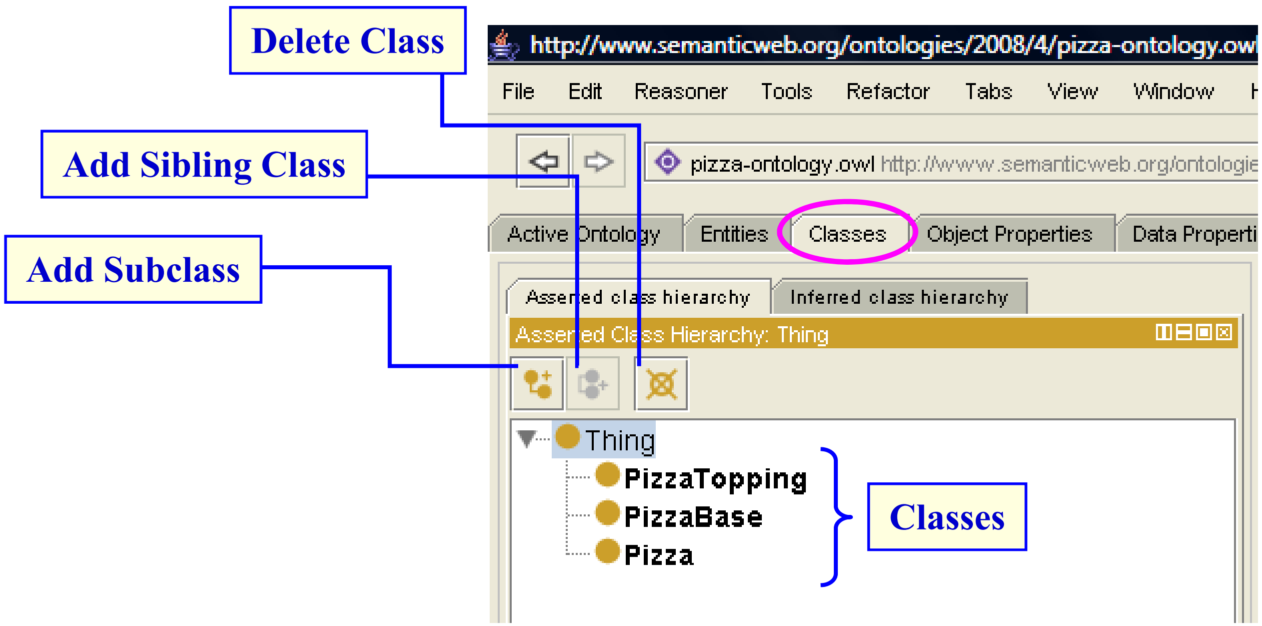
An empty ontology contains one class called *Thing*.

OWL classes are interpreted as sets of individuals or sets of objects. The class Thing is the class that represents the set containing all individuals.

Because of this all classes are subclasses of Thing.

Named Classes (II)

Creating classes in the pizza example: Pizza, PizzaBase, and PizzaTopping.

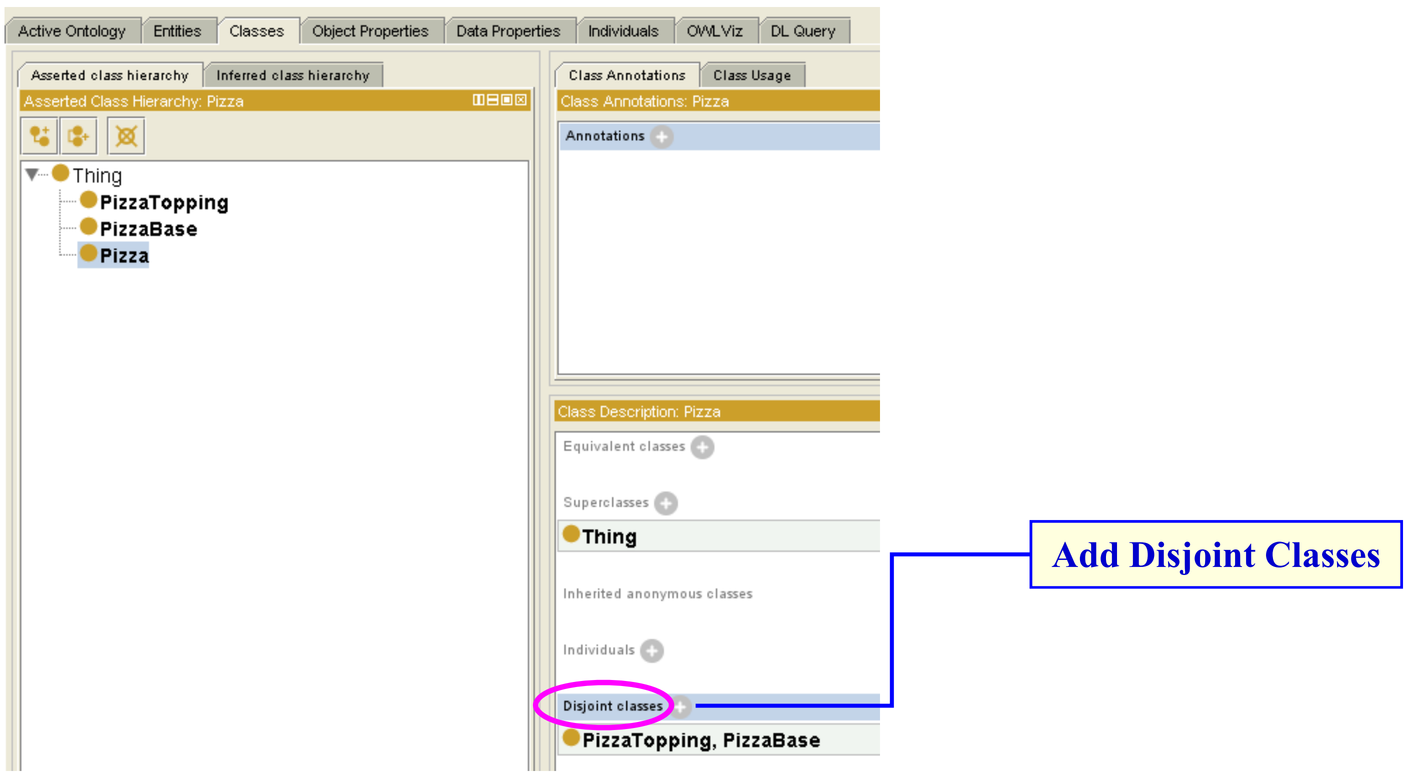


Disjoint Classes (I)

- OWL Classes are assumed to '**overlap**'. We therefore cannot assume that an individual is not a member of a particular class simply because it has not been asserted to be a member of that class.
- In order to 'separate' a group of classes we must make them **disjoint** from one another. This ensures that an individual which has been asserted to be a member of one of the classes in the group cannot be a member of any other classes in that group.

Disjoint Classes (II)

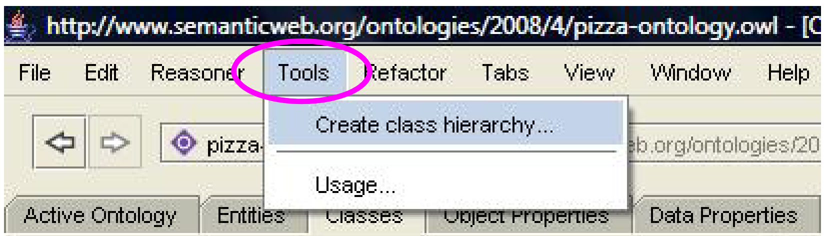
- Making the classes Pizza, PizzaTopping and PizzaBase disjoint from one another.
- This means that it is not possible for an individual to be a member of a combination of these classes – it would not make sense for an individual to be a Pizza and a PizzaBase.



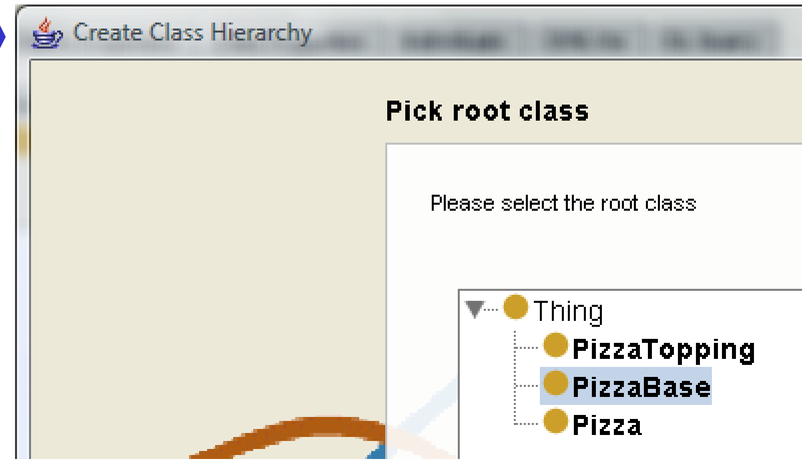
Named Classes (III)

Creating subclasses in the pizza example: ThinAndCrispyBase and DeepPanBase.

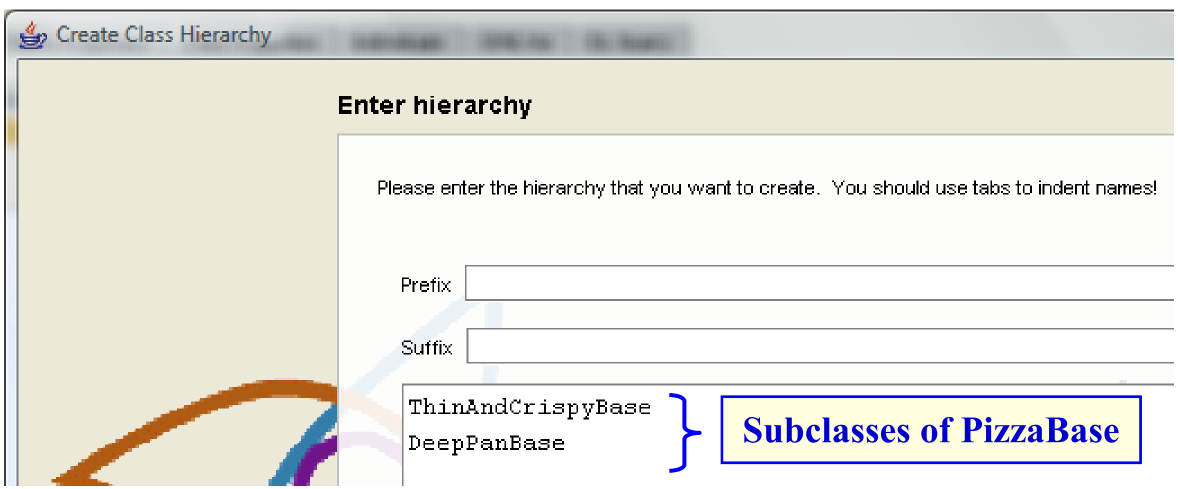
1



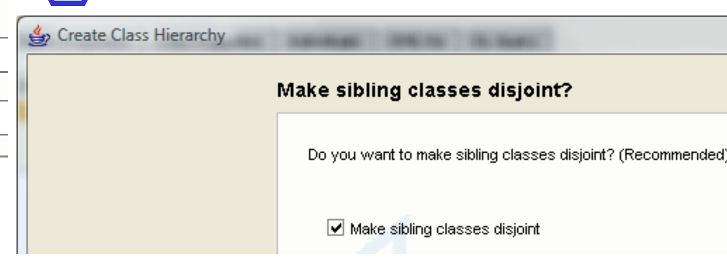
2



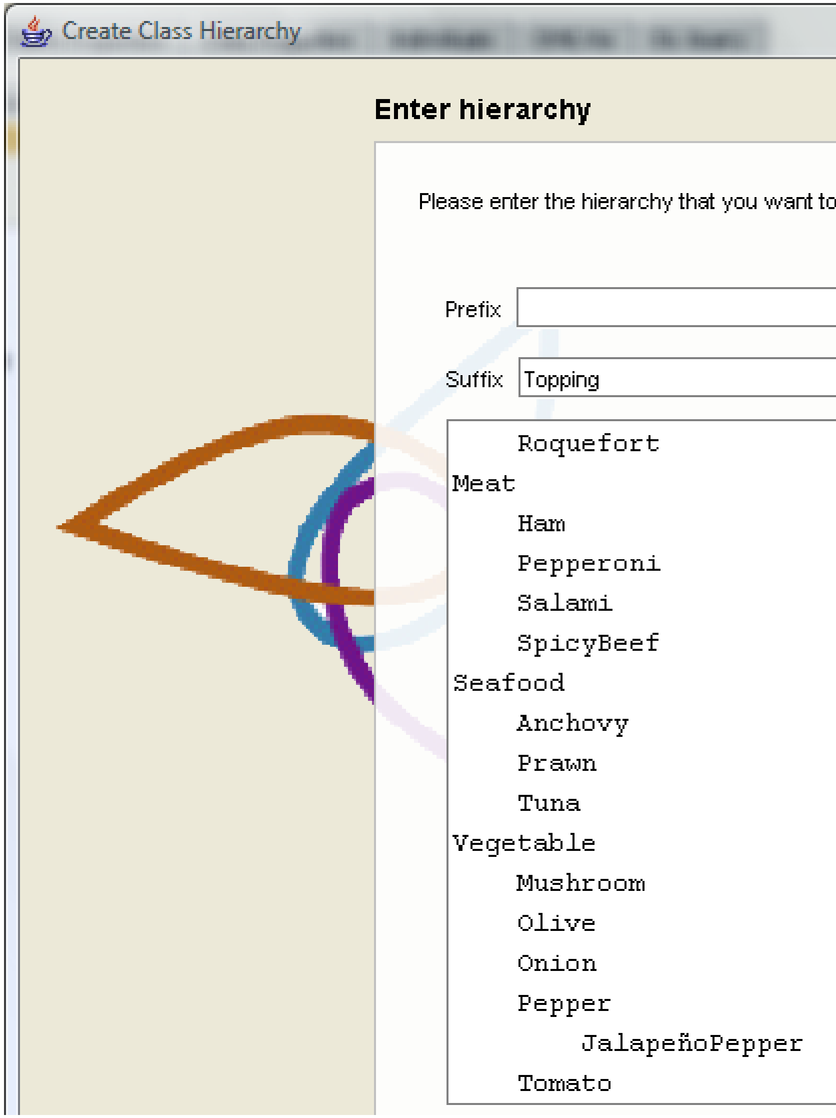
3

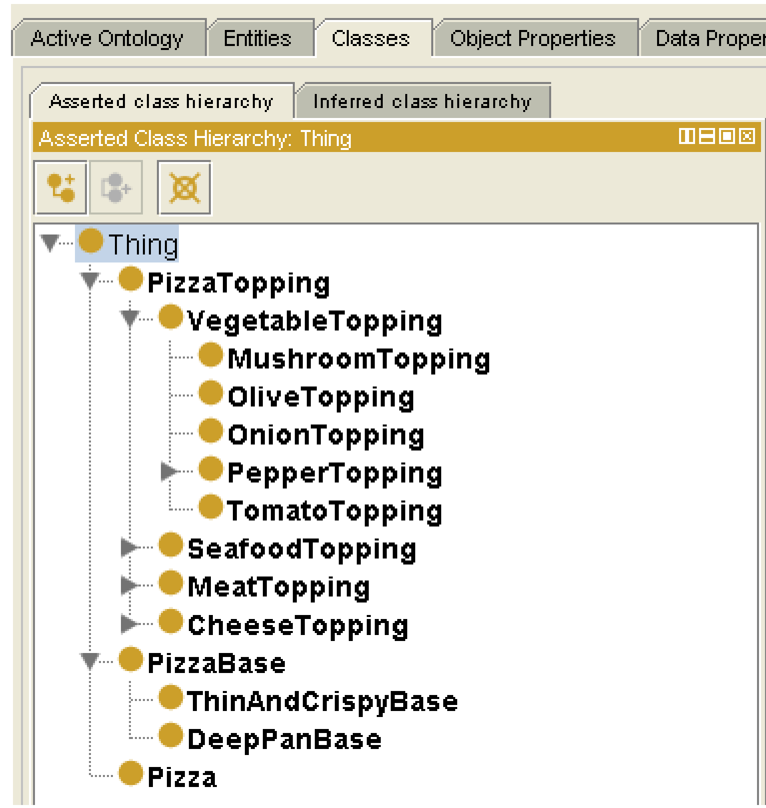


4



Class Hierarchy (I)

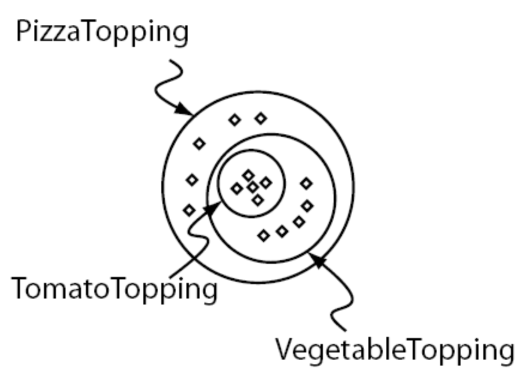




Class Hierarchy (II)

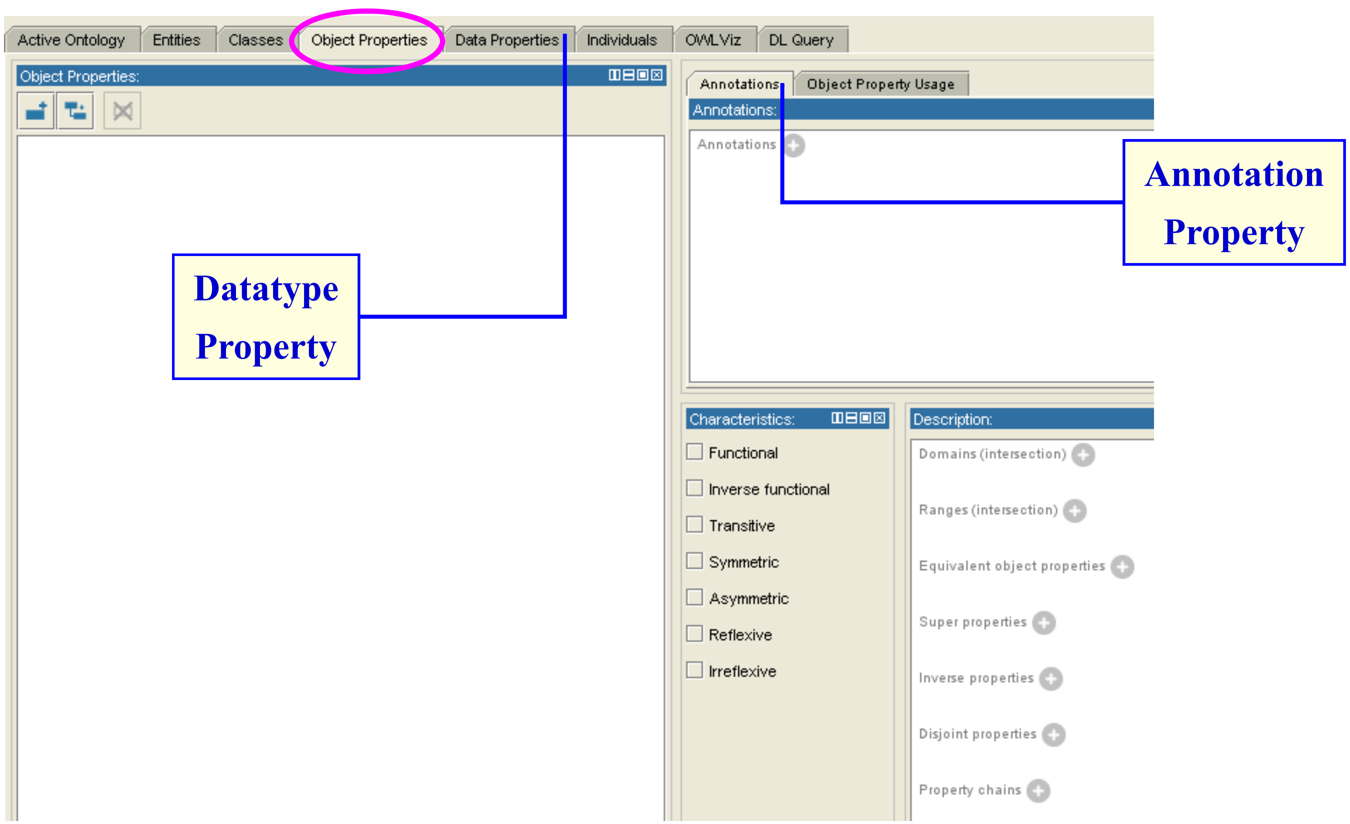
The Meaning of Subclass: all individuals that are members of the class TomatoTopping are members of the class VegetableTopping and members of the class PizzaTopping as we have stated that TomatoTopping is a subclass of VegetableTopping which is a subclass of PizzaTopping.

In OWL subclass means necessary implication. In other words, if VegetableTopping is a subclass of PizzaTopping then ALL instances of VegetableTopping are instances of PizzaTopping, without exception — if something is a VegetableTopping then this implies that it is also a PizzaTopping.



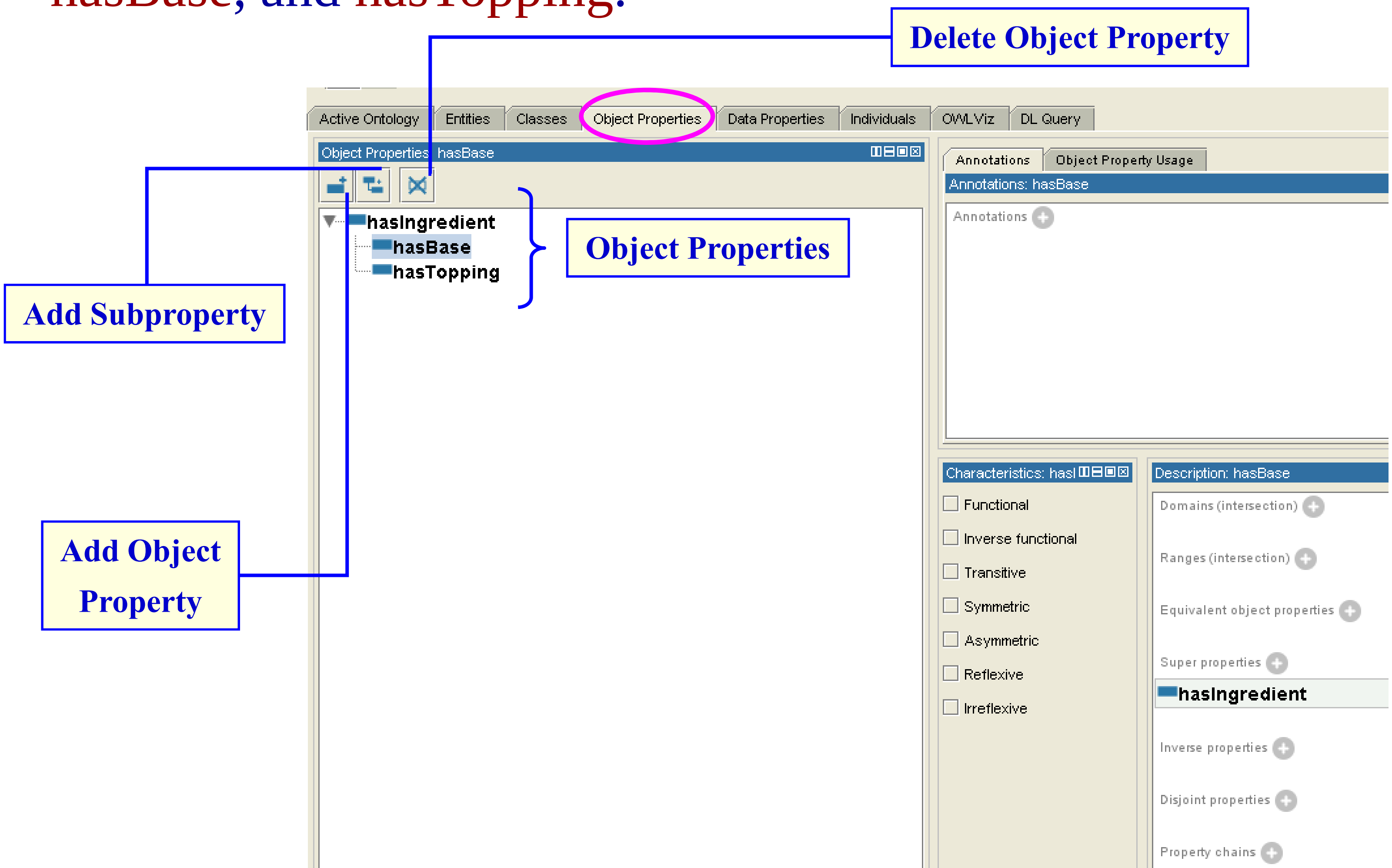
OWL Properties

OWL Properties represent relationships. There are two main types: Object properties and Datatype properties. OWL also has a third type of property: Annotation properties.



Object Properties

Creating object properties in the pizza example: hasIngredient, hasBase, and hasTopping.

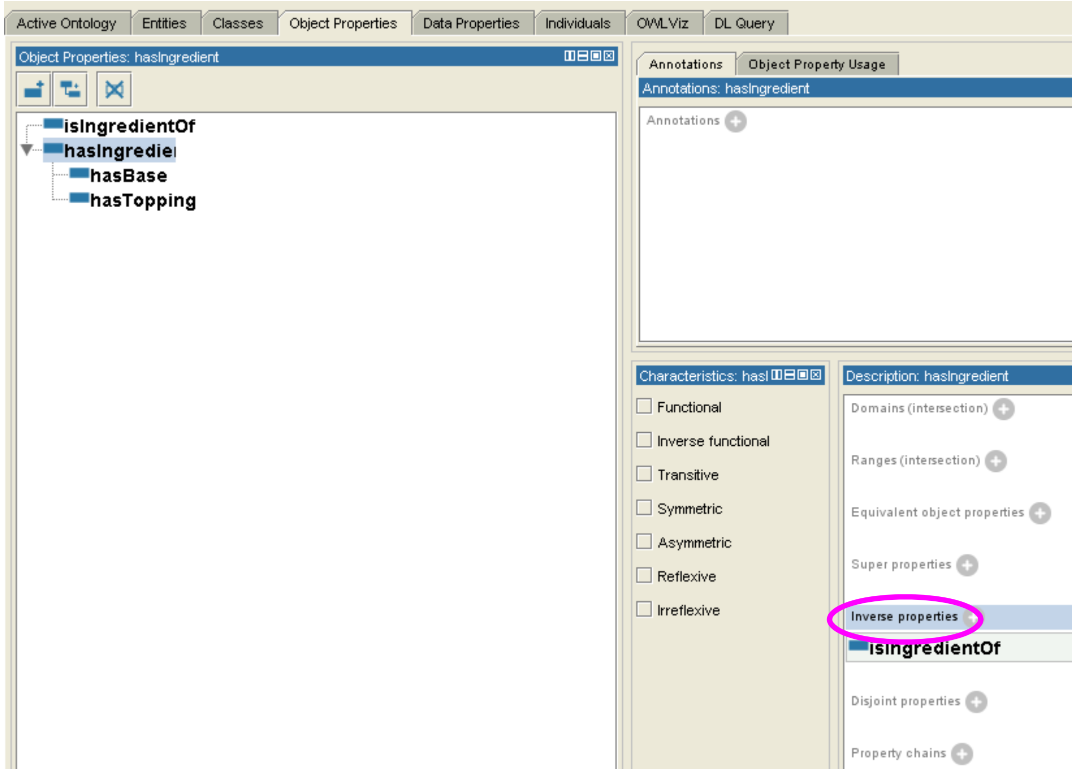


Inverse Properties

Each object property may have a corresponding inverse property.

If some property links individual a to individual b, then its invers property ling individual b to individual a.

Creating inverse properties in the pizza example: isIngredientOf.



OWL Object Properties Characteristics (I)

OWL allows the meaning of properties to be enriched through the use of property characteristics.

Functional Properties

Inverse Functional Properties

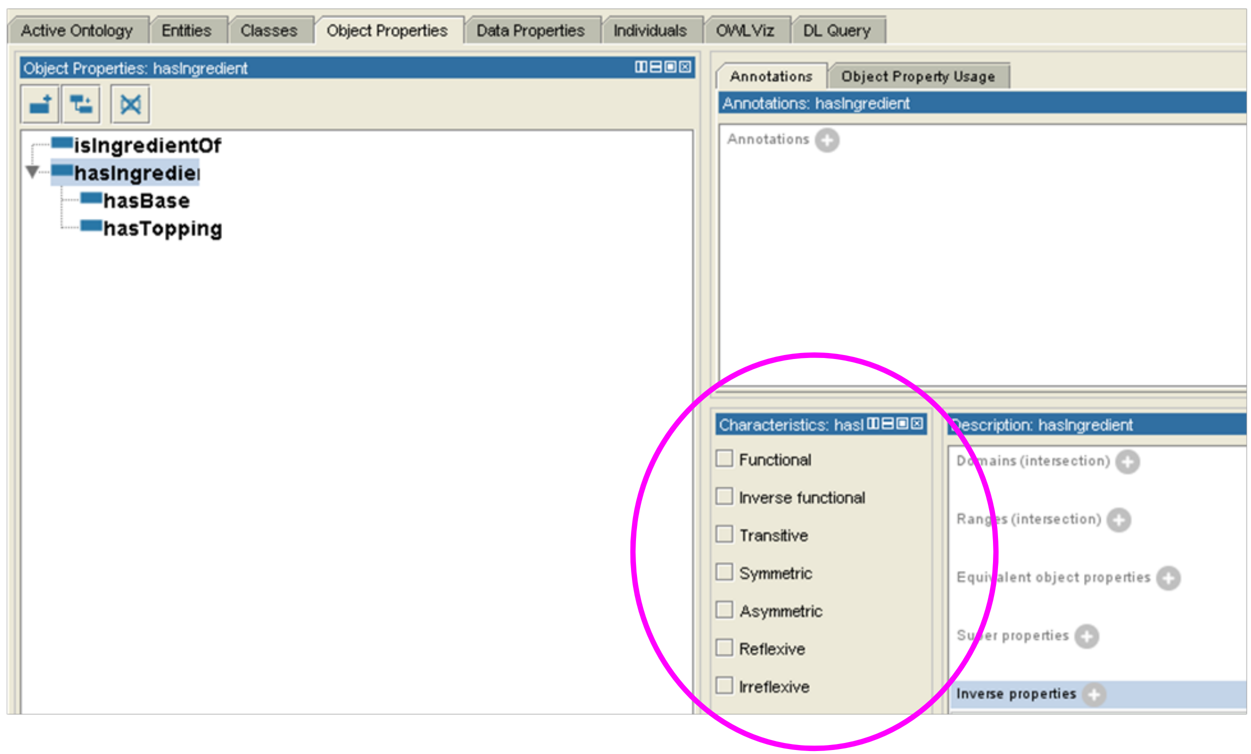
Transitive Properties

Symmetric Properties

Antisymetric Properties

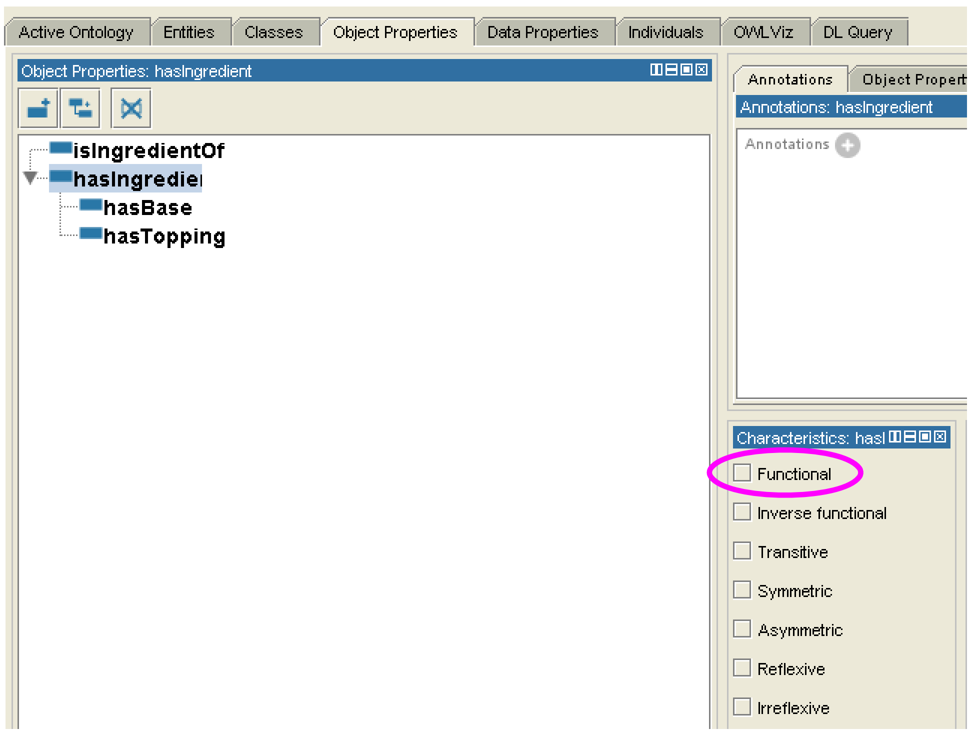
Reflexive Properties

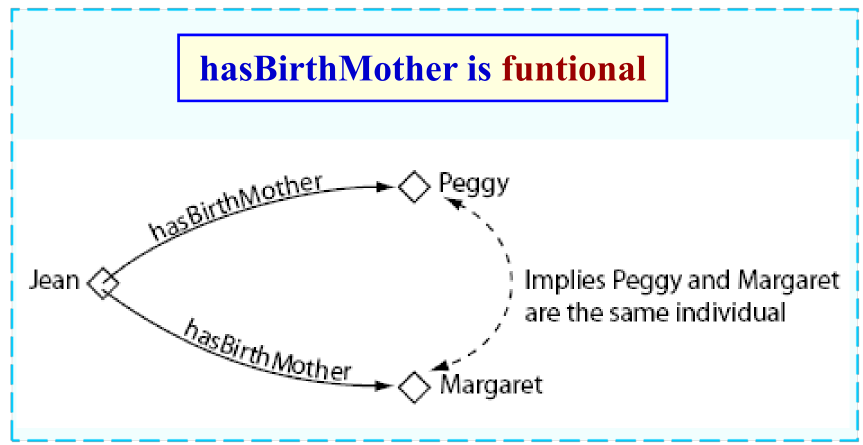
Irreflexive Properties



Functional Properties

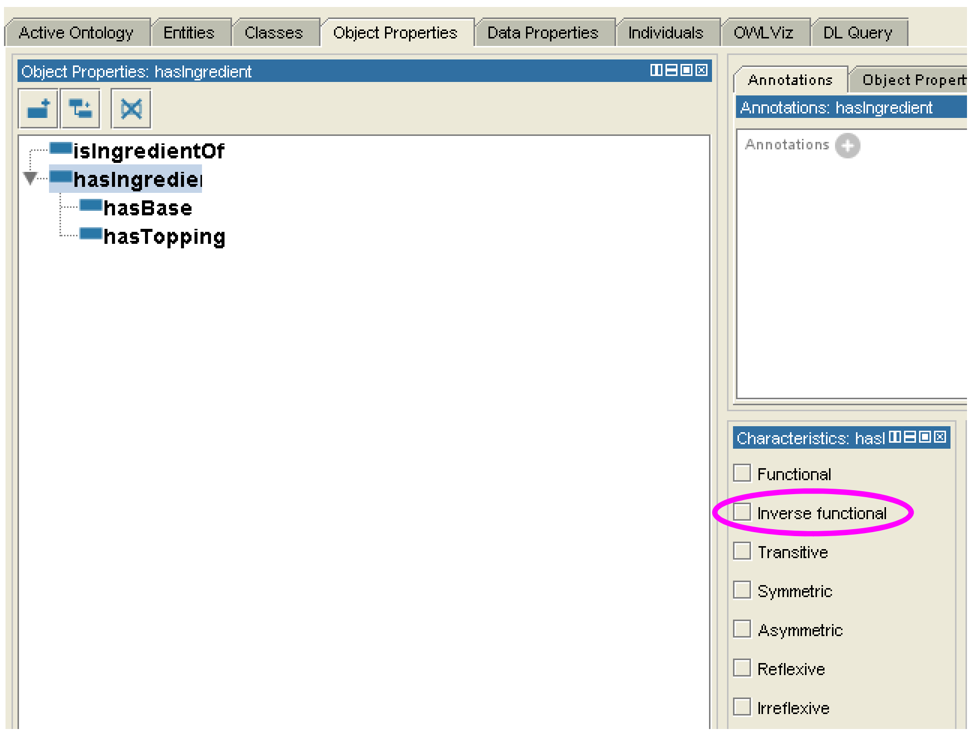
If a property is **functional**, for a given individual, there can be at most one individual that is related to the individual via the property.





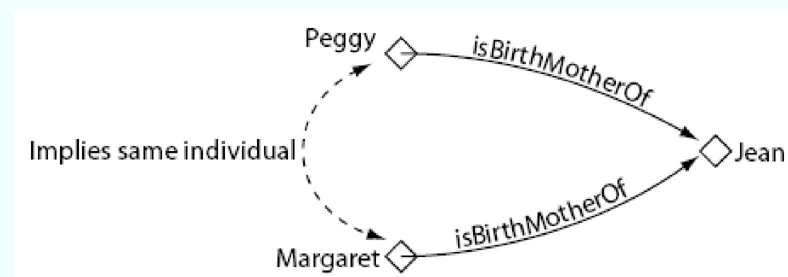
Inverse Functional Properties

If a property is **inverse functional** then it means that the inverse property is functional. For a given individual, there can be at most one individual related to that individual via the property.



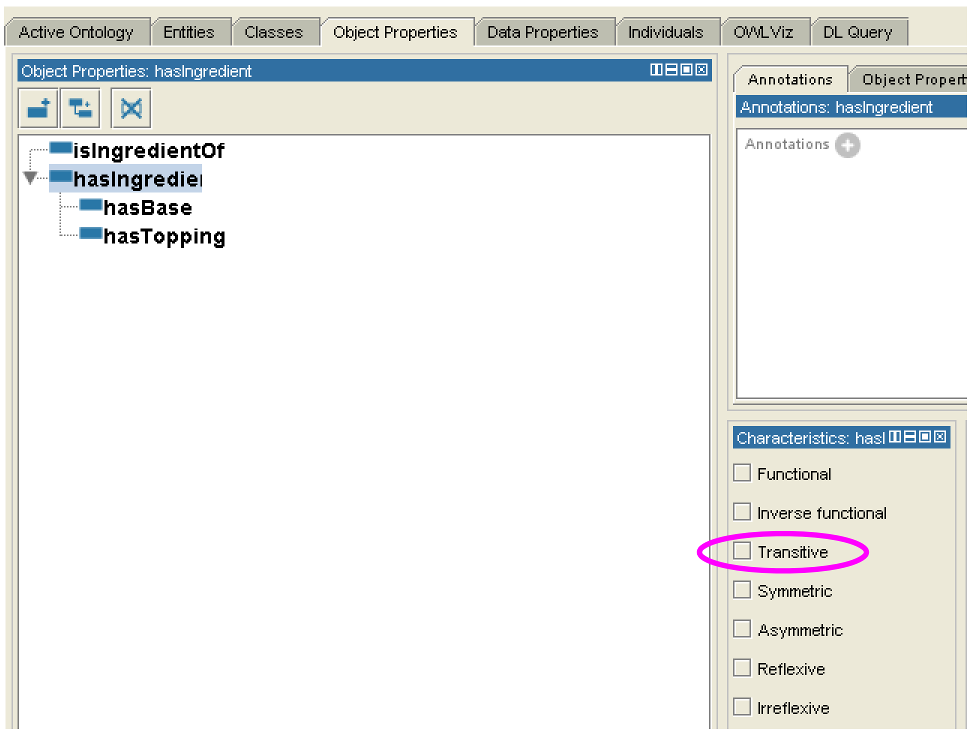
isBirthMotherOf is inverse functional

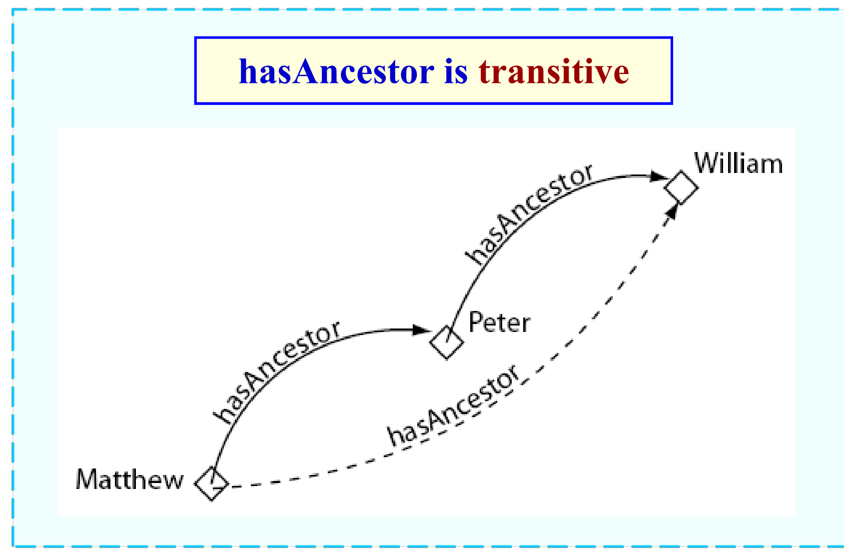
This is the inverse property of hasBirthMother — since hasBirthMother is functional, isBirthMotherOf is inverse functional



Transitive Properties

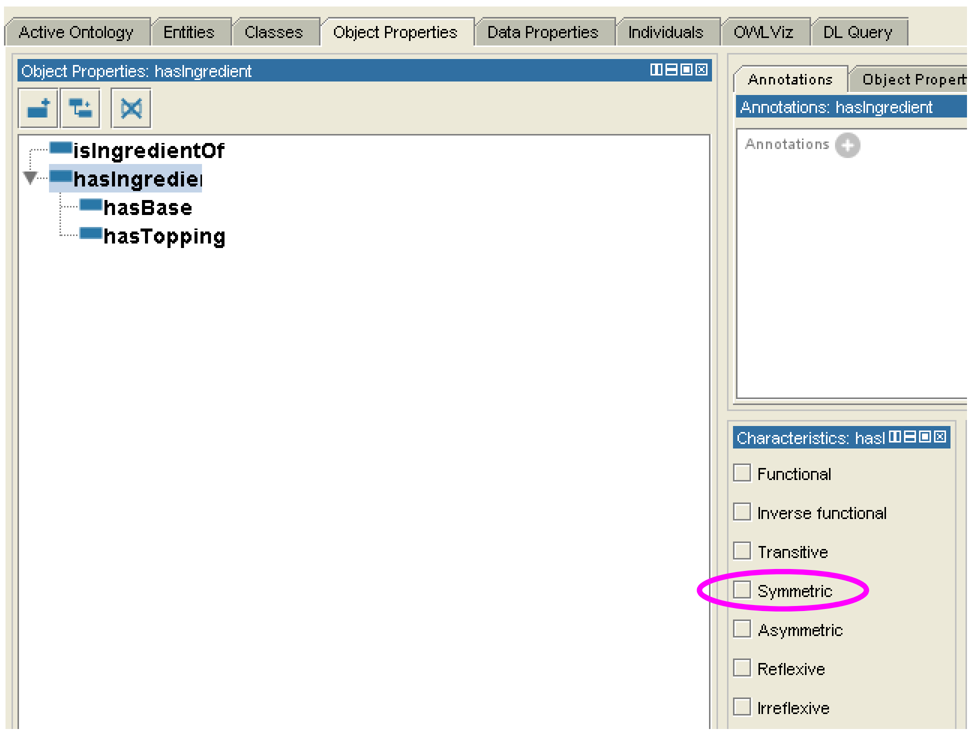
If a property is **transitive**, and the property relates individual a to individual b, and also individual b to individual c, then we can infer that individual a is related to individual c via the property.

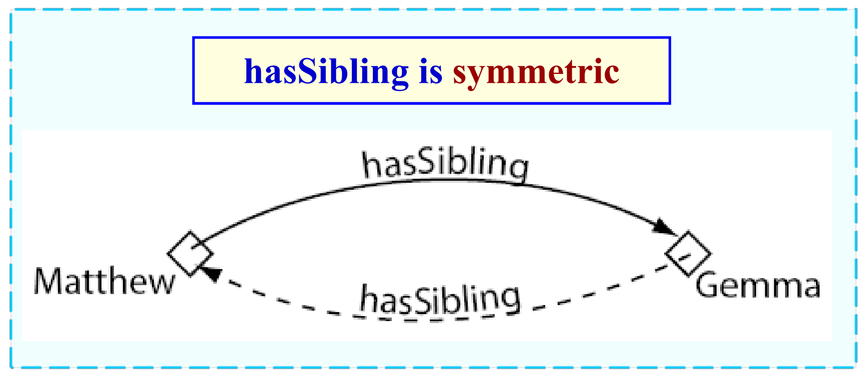




Symmetric Properties

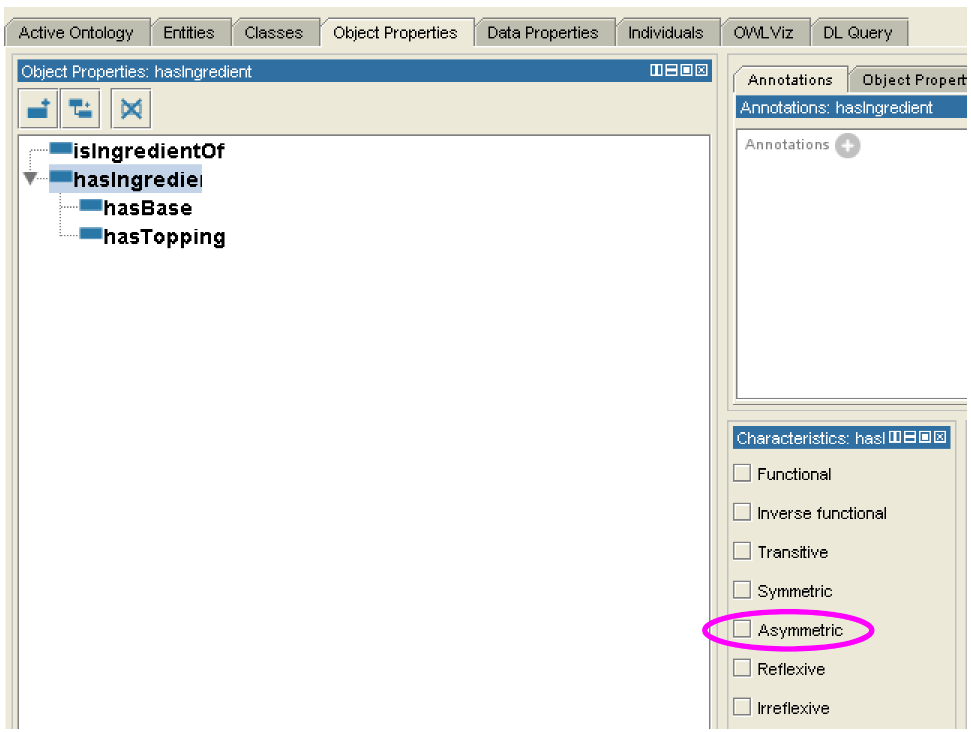
If a property is **symmetric**, and the property relates individual a to individual b then individual b is also related to individual a via the property.

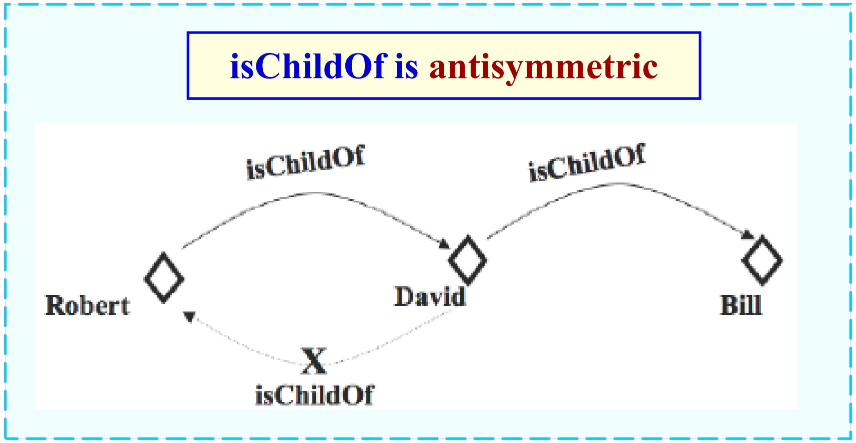




Antisymmetric Properties

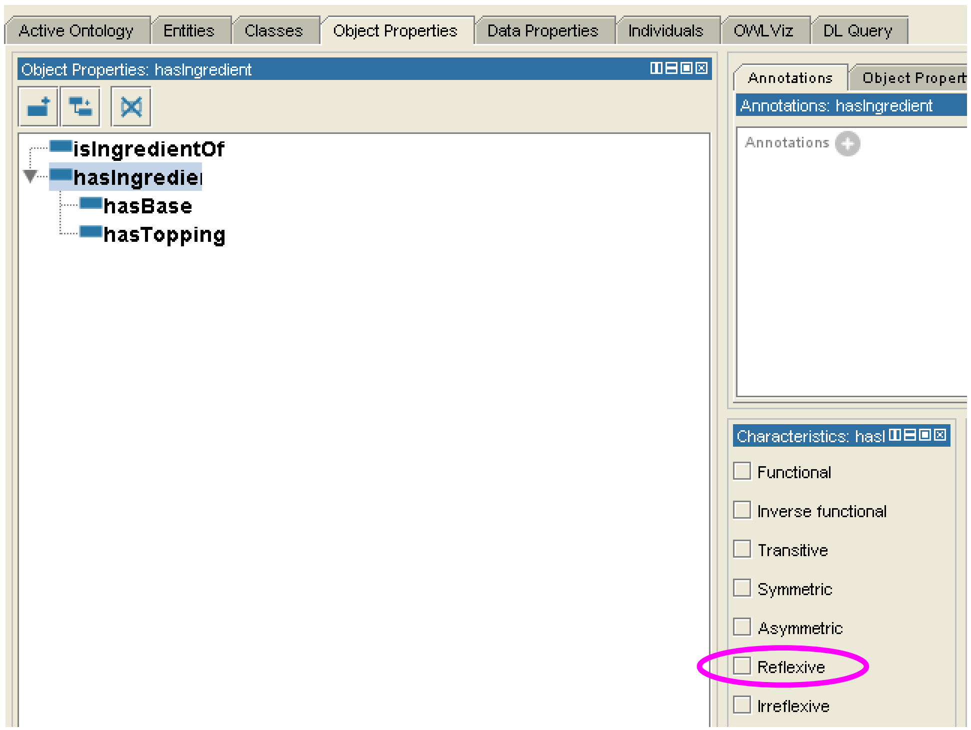
If a property is **antisymmetric**, and the property relates individual a to individual b then individual b cannot be related to individual a via the property.



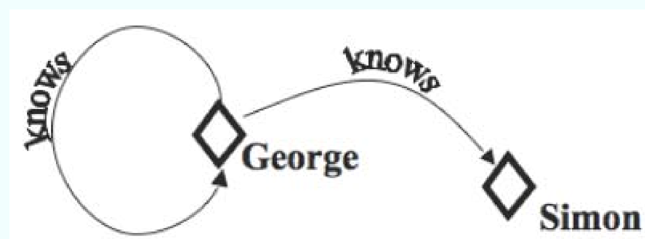


Reflexive Properties

A property is said to be **reflexive** when the property must relate individual a to itself.

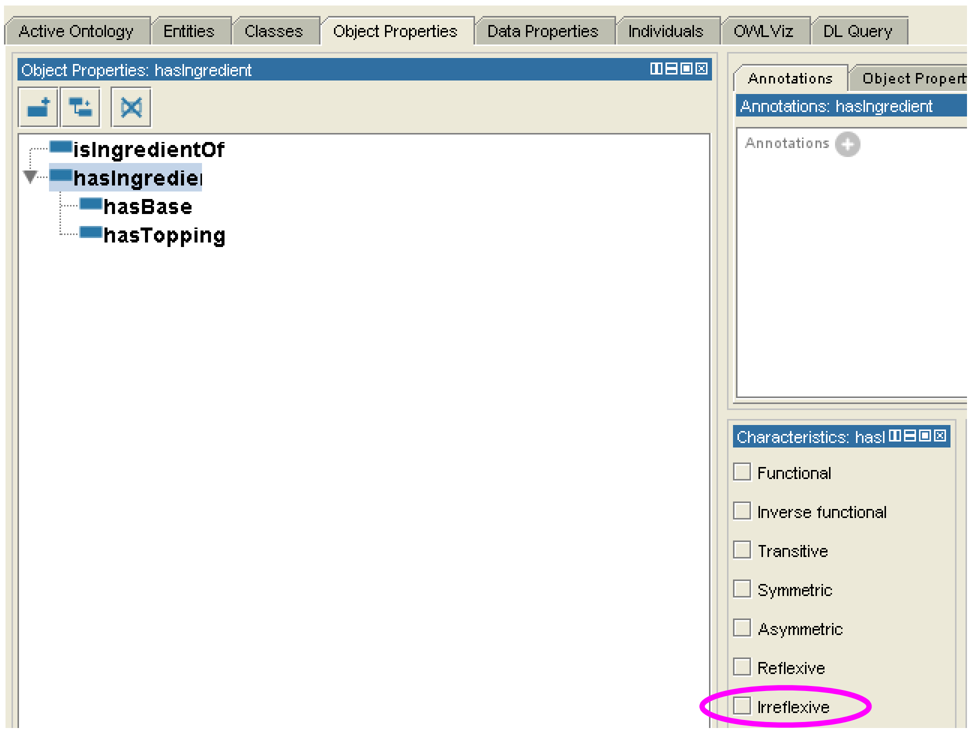


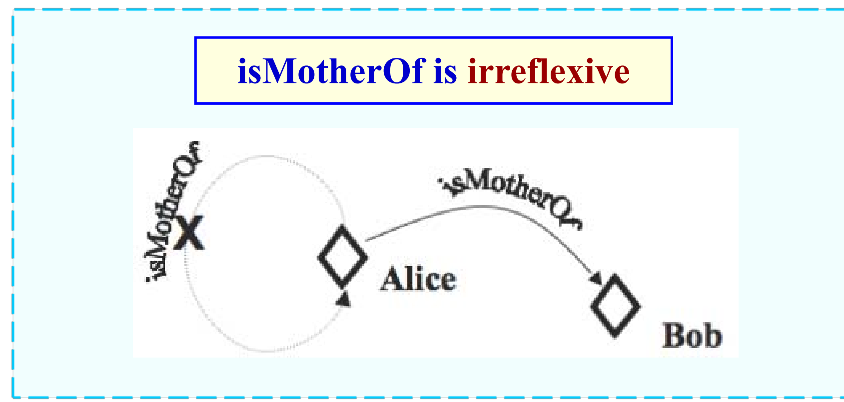
knows is reflexive



Irreflexive Properties

If a property is **irreflexive**, it can be described as a property that relates an individual a to individual b, where individual a and individual b are not the same.





Exercise

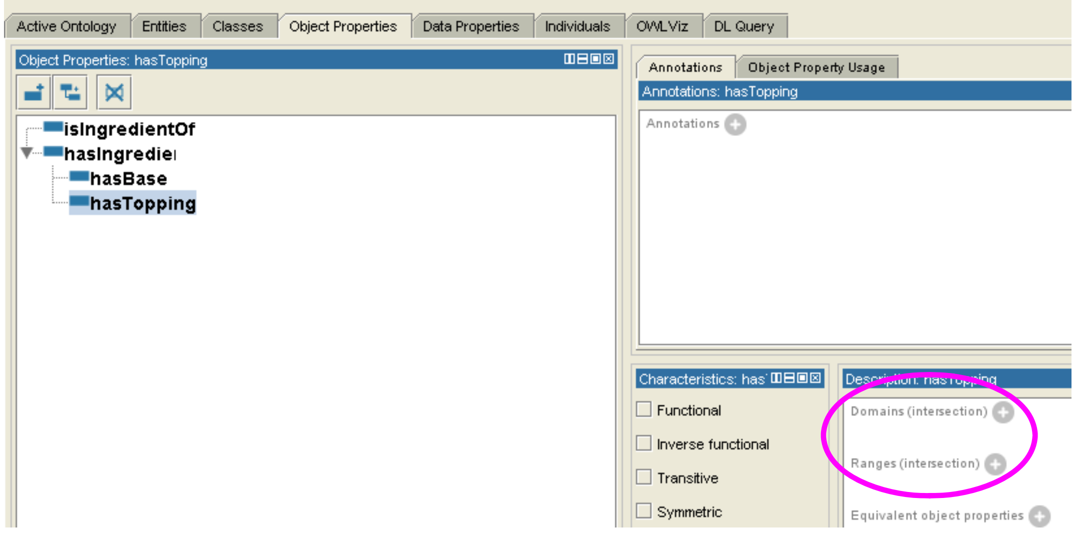
Modelling that a pizza has only one pizza base; and that if a pizza topping has ingredients, then the pizza itself contains also such ingredients.

OWL Properties: Domain and Range (I)

Properties may have a domain and a range specified.

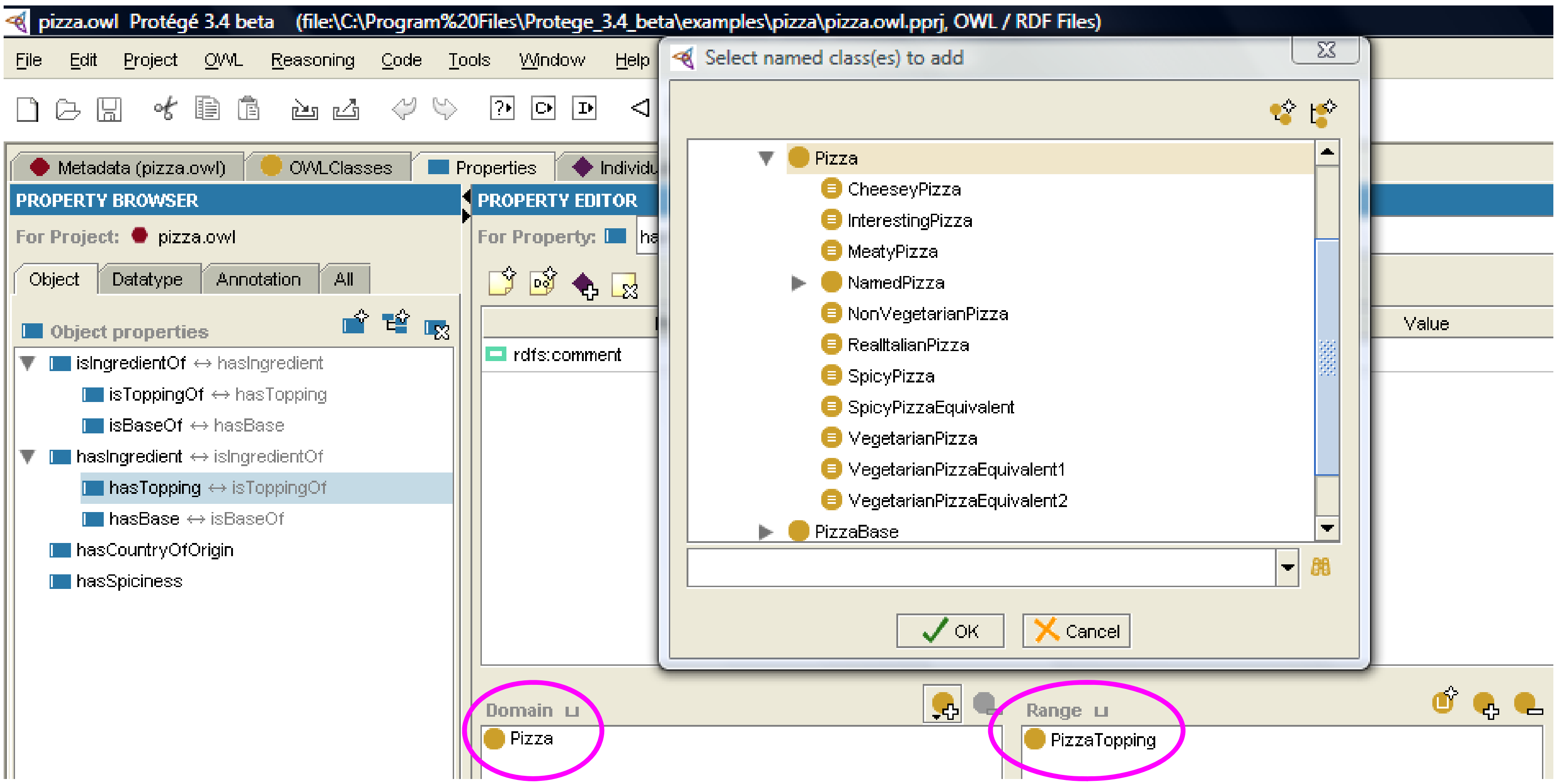
Properties link individuals from the domain to individuals from the range.

Specifiying the **domain (Pizza)** and **range (PizzaTopping)** of hasTopping property.



OWL Properties: Domain and Range (II)

Protege 3.4



Property Restrictions

A **restriction** describes an anonymous class (an unnamed class). The anonymous class contains all of the individuals that satisfy the restriction (i.e. all of the individuals that have the relationships required to be a member of the class).

Restrictions are used in OWL class descriptions to specify anonymous superclasses of the class being described.

Existential restrictions describe classes of individuals that participate in at least one relationship along a specified property to individuals that are members of a specified class.

For example, “the class of individuals that have at least one (some) hasTopping relationship to members of MozzarellaTopping”.

Universal restrictions describe classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class.

For example, “the class of individuals that only have hasTopping relationships to members of VegetableTopping”.

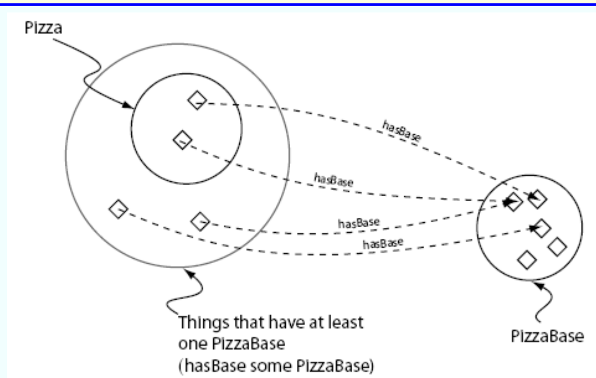
Existential Restrictions (I)

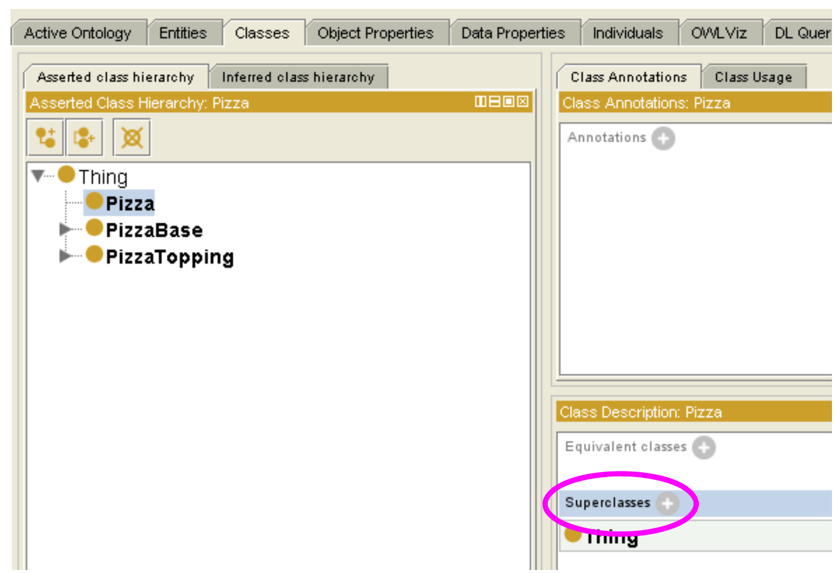
An existential restriction describes a class of individuals that have at least one (some) relationship along a specified property to an individual that is a member of a specified class.

Existential restrictions are also known as Some Restrictions, or as some values from restrictions.

Adding a restriction to Pizza that specifies a Pizza must have a PizzaBase (**hasBase some PizzaBase**). You are creating a *necessary condition*.

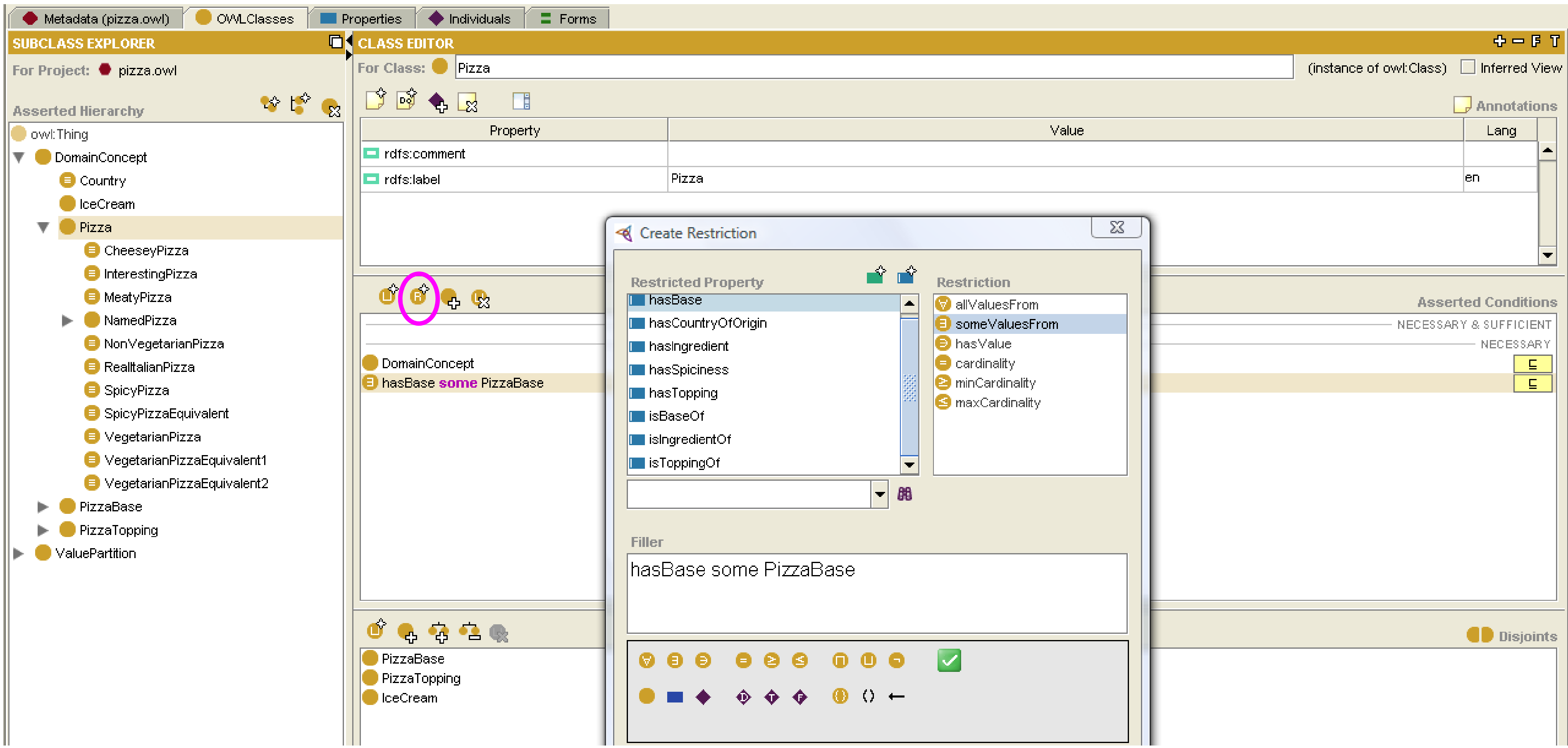
A Pizza is a subclass of the things that have at least one PizzaBase





Existential Restrictions (II)

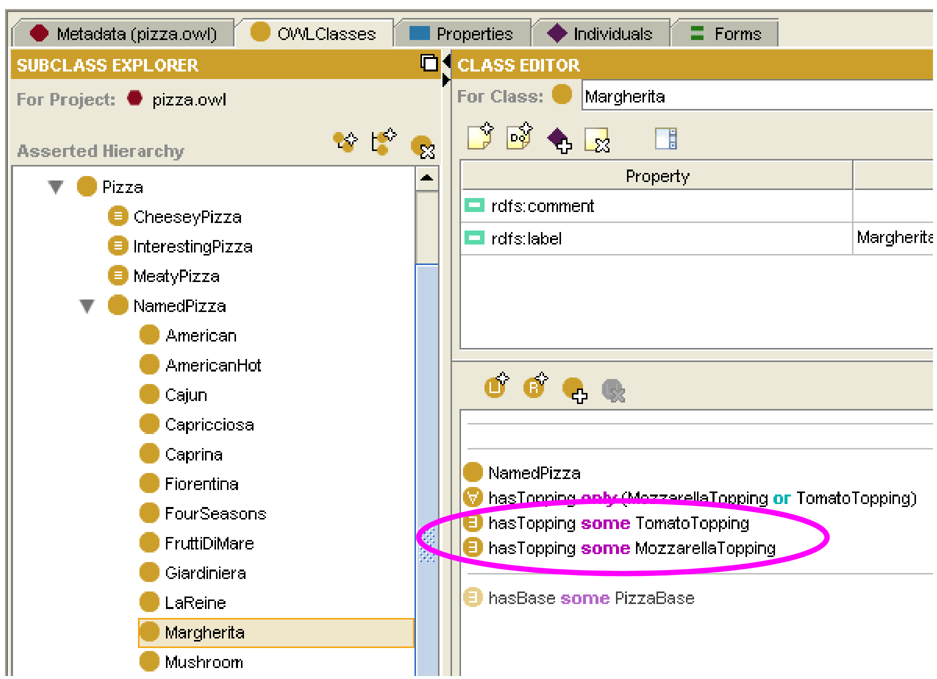
Protege 3.4



Existential Restrictions (III)

Adding two restrictions to say that a MargheritaPizza has the toppings MozzarellaTopping and TomatoTopping.

If something is a member of the class MargheritaPizza it is necessary for it to be a member of: the class NamedPizza, the anonymous class of things that are linked to at least one member of the class MozzarellaTopping via the property hasTopping, and the anonymous class of things that are linked to at least one member of the class TomatoTopping via the property hasTopping.



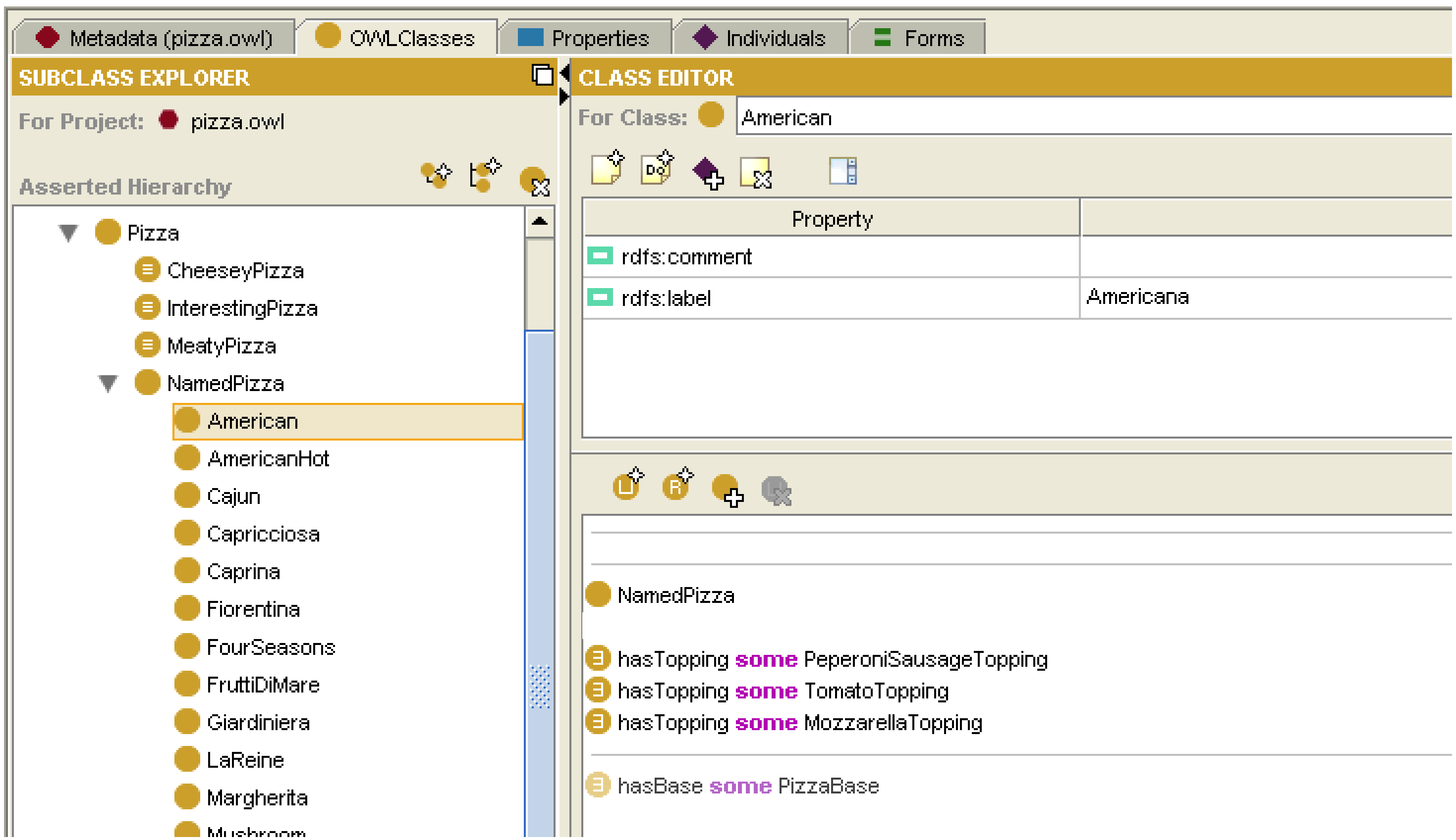
Protege 3.4

Exercise

Create an **American Pizza** that is almost the same as a Margherita Pizza but with an extra topping of pepperoni.

Exercise: Solution

Create an **American Pizza** that is almost the same as a Margherita Pizza but with an extra topping of pepperoni.



Using a Reasoner

One of the key features of ontologies that are described using OWL-DL is that they can be processed by a **reasoner**.

One of the main services offered by a reasoner is to test whether or not one class is a subclass of another class. By performing such tests on the classes in an ontology it is possible for a reasoner to compute the inferred ontology class hierarchy.

Another standard service that is offered by reasoners is consistency checking. Based on the description (conditions) of a class the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.

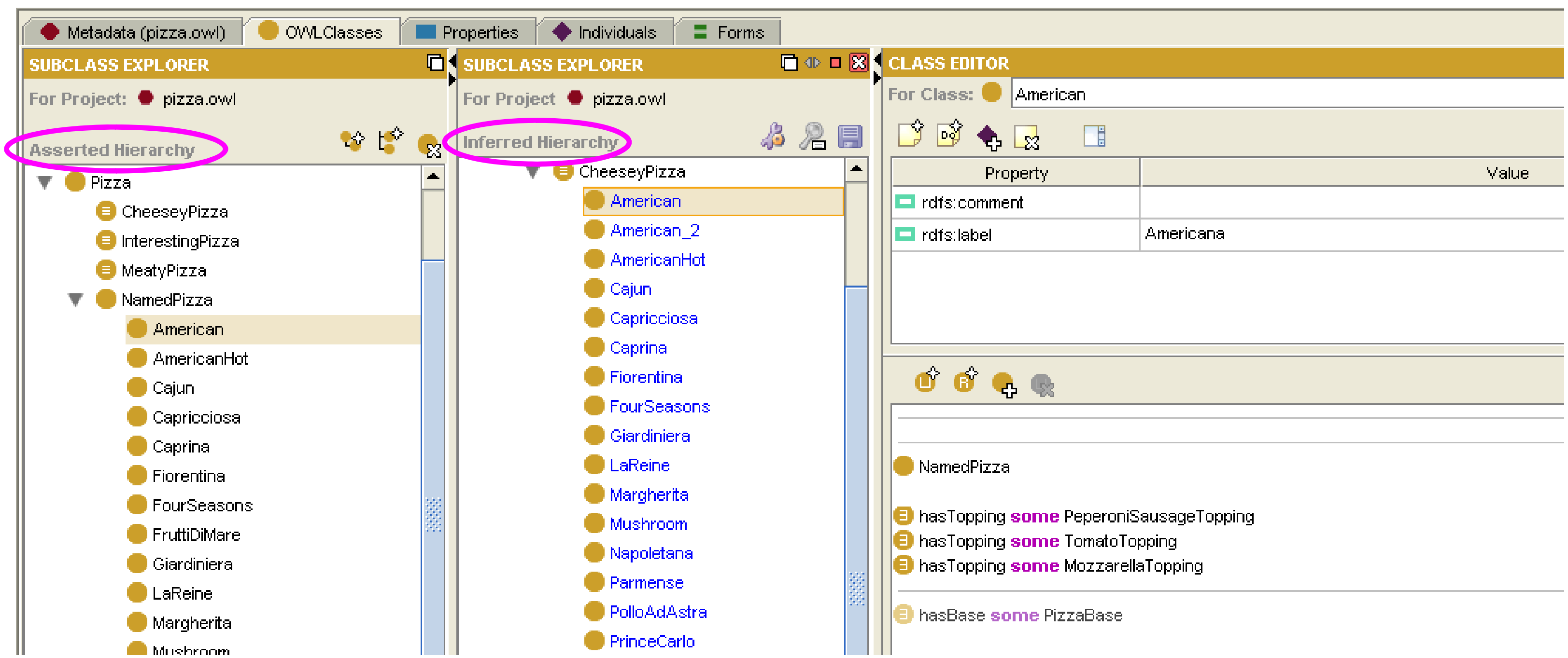
Inferring Ontology Class Hierarchy

The ontology can be ‘sent to the reasoner’ to automatically compute the classification hierarchy.

The ‘manually constructed’ class hierarchy is called the **asserted hierarchy**.

The class hierarchy that is automatically computed by the reasoner is called the **inferred hierarchy**.

Protege 3.4. Pellet 1.5.1

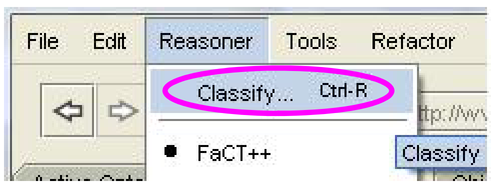


Checking Ontology Consistency

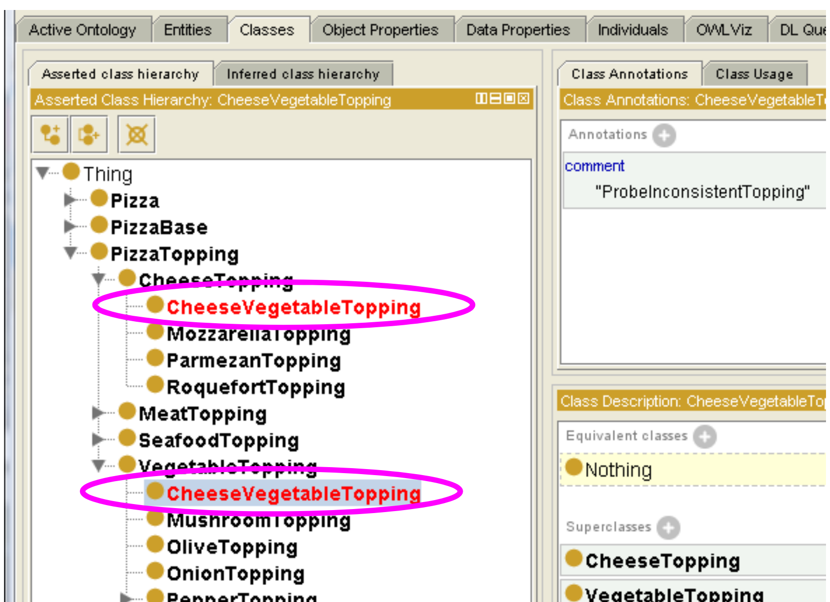
This strategy is often used as a check so that we can see that we have built our ontology correctly.

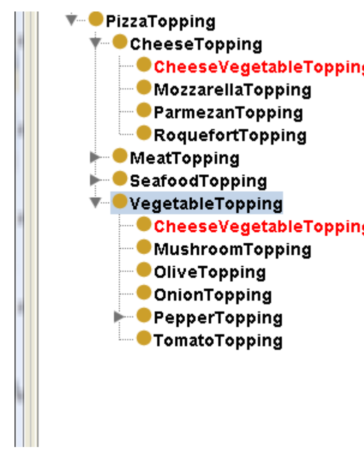
Creating a **CheesyVegetableTopping** as subclass of CheesyTopping and VegetableTopping.

1

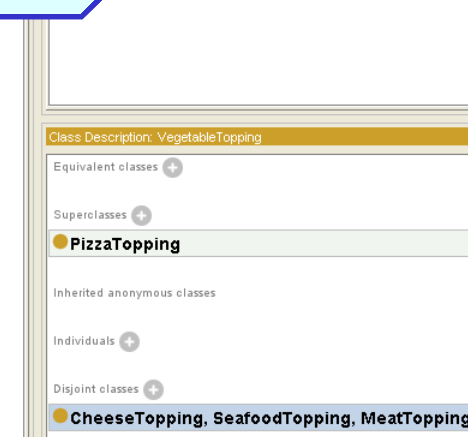


2





Why?



Necessary and Sufficient Conditions (I)

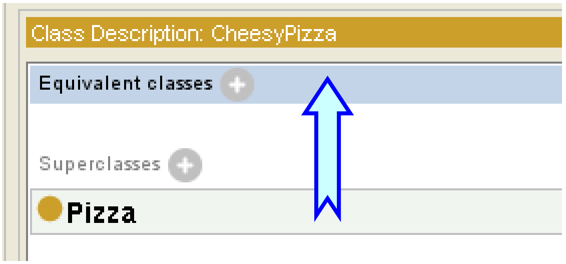
All of the classes that we have created so far have only used necessary conditions to describe them.

Necessary conditions can be read as: "If something is a member of this class then it is necessary to fulfil these conditions".

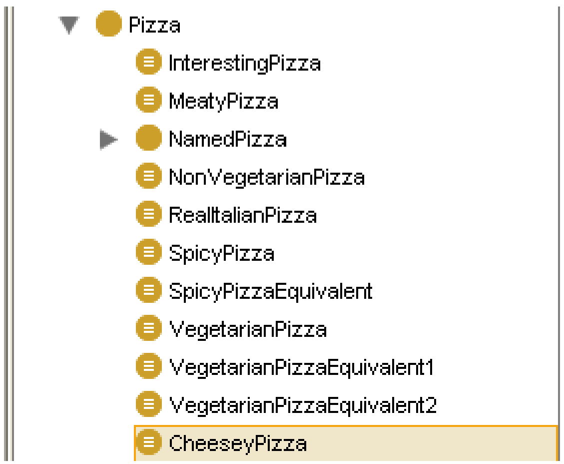
A class that only has necessary conditions is known as a Primitive Class or Partial Class.

With necessary conditions alone, we cannot say that, "*If something fulfils these conditions then it must be a member of this class*". To make this possible we need to change the conditions from necessary conditions to **necessary AND sufficient conditions**.

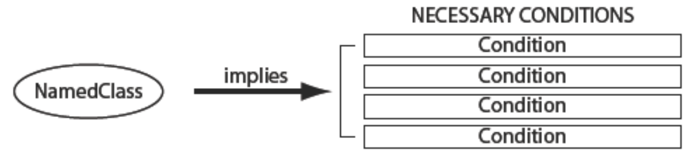
A class that has at least one set of necessary and sufficient conditions is known as a Defined Class or Complete Class.



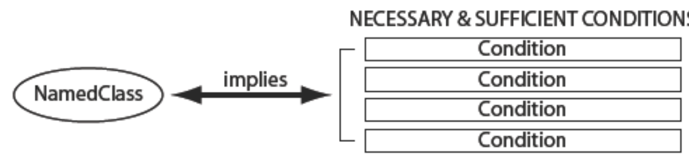
Necessary and Sufficient Conditions (II)



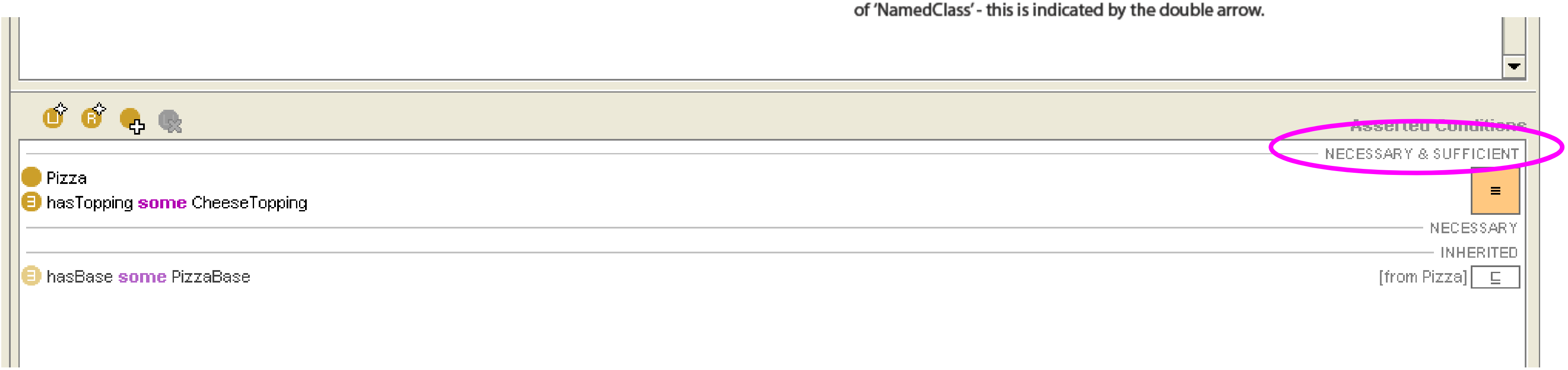
Protege 3.4



If an individual is a member of 'NamedClass' then it must satisfy the conditions. However if some individual satisfies these necessary conditions, we cannot say that it is a member of 'Named Class' (the conditions are not 'sufficient' to be able to say this) - this is indicated by the direction of the arrow.



If an individual is a memeber of 'NamedClass' then it must satisfy the conditions. If some individual satisfies the condtions then the individual must be a member of 'NamedClass' - this is indicated by the double arrow.



Universal Restrictions (I)

All of the restrictions that we have created so far have been existenctial ones (some).

However, existential restrictions do not mandate that the only relationships for the given property that can exist must be to individuals that are members of the specified filler class. To restrict the relationships for a given property to individuals that are members of a specific class we must use a **universal restriction**.

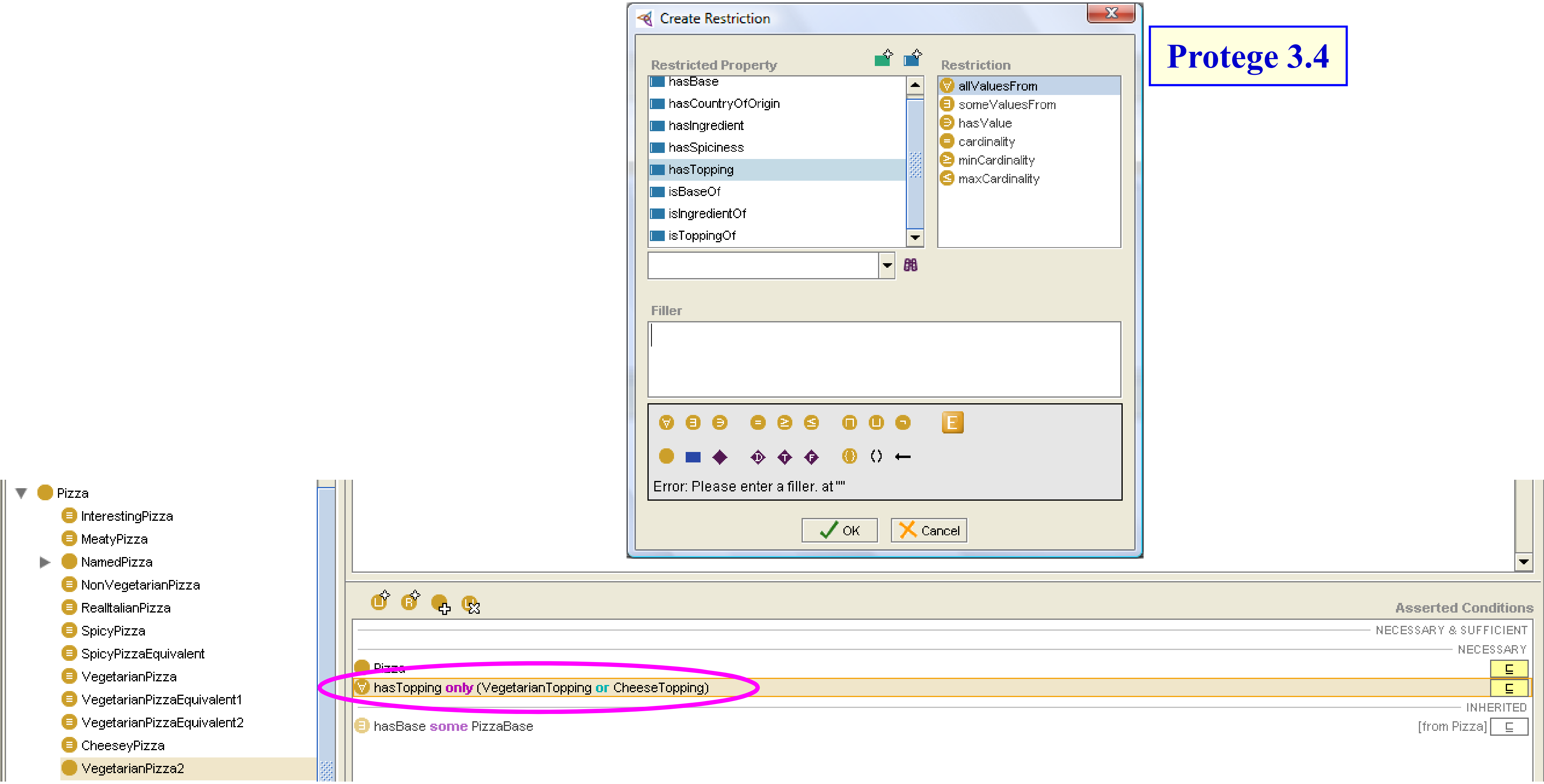
Universal restrictions constrain the relationships along a given property to individuals that are members of a specific class.

For example the universal restriction ∀ hasTopping MozzarellaTopping describes the individuals all of whose hasTopping relationships are to members of the class MozzarellaTopping.

Universal Restrictions (II)

Creating a Vegetarian Pizza that only have toppings that are CheeseTopping or VegetableTopping.

Protege 3.4



Automatic Classification and Open World Assumption (I)

We want to use the reasoner to automatically compute the superclass-subclass relationship (subsumption relationship) between MargheritaPizza and VegetarianPizza.

We believe that MargheritaPizza should be vegetarian pizza (they should be subclasses of VegetarianPizza). This is because they have toppings that are essentially vegetarian toppings — by our definition, vegetarian toppings are members of the classes CheeseTopping or VegetableTopping and their subclasses.

Having previously created a definition for VegetarianPizza (using a set of necessary and sufficient conditions) we can use the reasoner to perform automatic classification and determine the vegetarian pizzas in our ontology.

Automatic Classification and Open World Assumption (II)

MargheritaPizza has not been classified as subclass of VegetarianPizza.

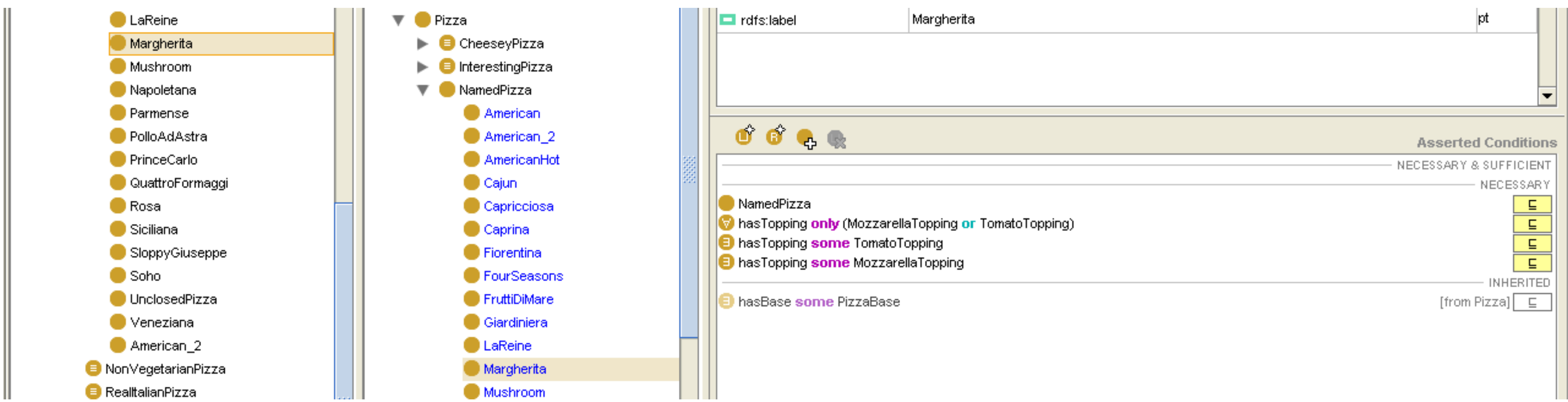
Reasoning in OWL (Description Logics) is based on what is known as the open world assumption (OWA). The open world assumption means that we cannot assume something does not exist until it is explicitly stated that it does not exist.

In the case of our pizza ontology, we have stated that MargheritaPizza has toppings that are kinds of MozzarellaTopping and also kinds of TomatoTopping. Because of the open world assumption, until we explicitly say that a MargheritaPizza **only** has these kinds of toppings, it is assumed (by the reasoner) that a MargheritaPizza could have other toppings.

Automatic Classification and Open World Assumption (III)

To specify explicitly that a MargheritaPizza has toppings that are kinds of MozzarellaTopping or kinds of MargheritaTopping and only kinds of MozzarellaTopping or MargheritaTopping, we must add what is known as a **closure axiom or restriction** on the hasTopping property.

Protege 3.4



Cardinality Restrictions (I)

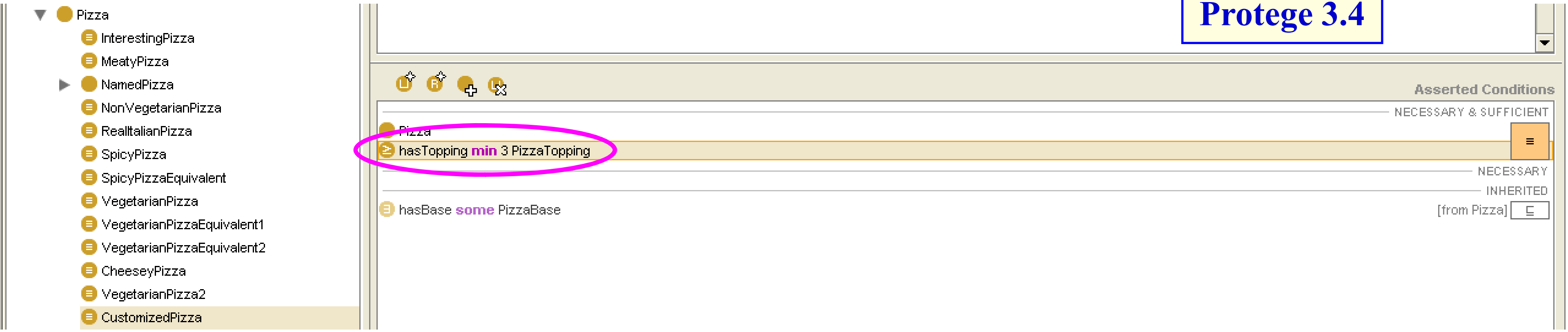
In OWL we can describe the class of individuals that have at least, at most or exactly a specified number of relationships with other individuals or datatype values. The restrictions that describe these classes are known as **Cardinality Restrictions**.

- A Minimum Cardinality Restriction specifies the minimum number of P relationships that an individual must participate in.
- A Maximum Cardinality Restriction specifies the maximum number of P relationships that an individual can participate in.
- A Cardinality Restriction specifies the exact number of P relationships that an individual must participate in.

Cardinality Restrictions (II)

Creating a Customized Pizza that has at least three toppings.

Protege 3.4

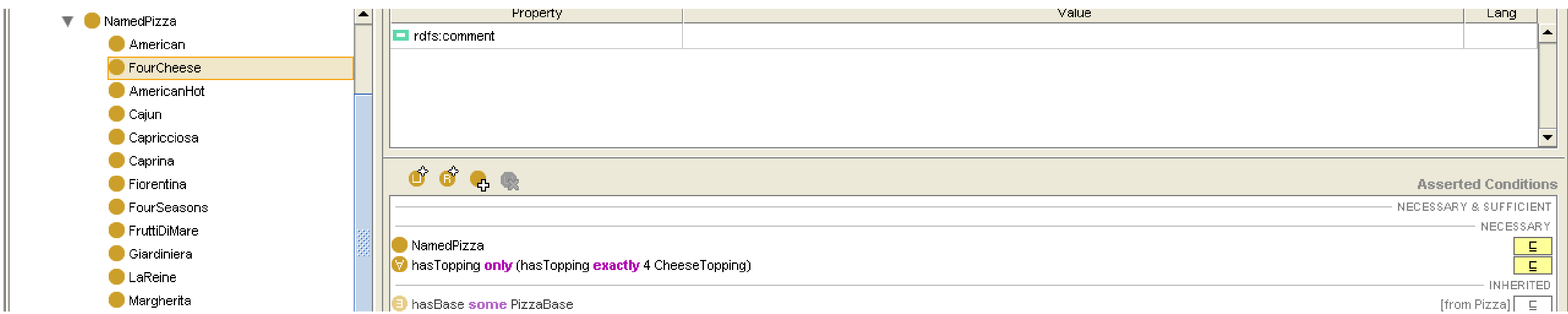


Qualified Cardinality Restrictions (I)

Qualified Cardinality Restrictions (QCR), which are more specific than cardinality restrictions in that they state the class of objects within the restriction.

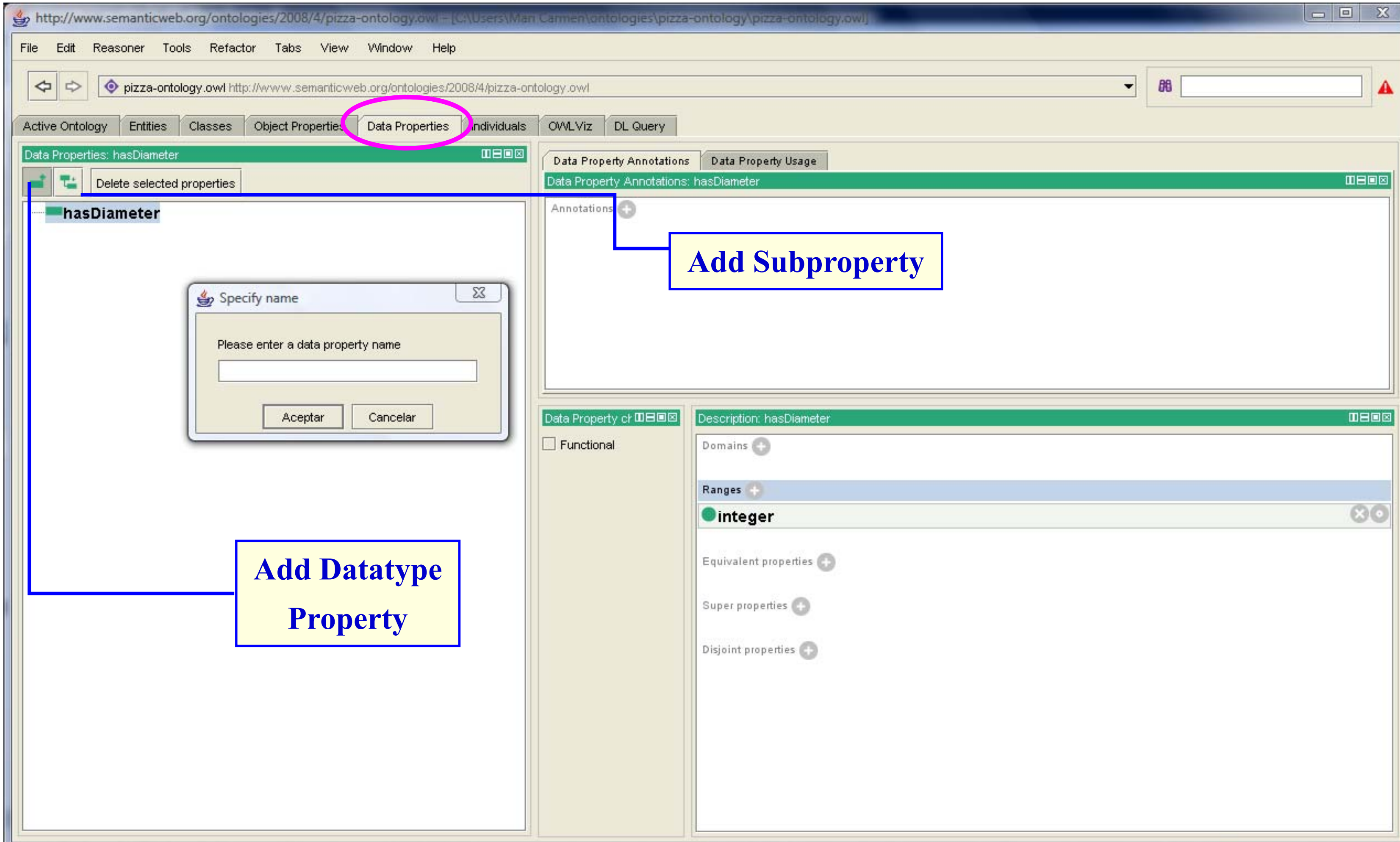
Creating a **Four Cheese Pizza**, as subclass of NamedPizza, which has exactly four cheese toppings.

Protege 3.4



Datatype Properties

Creating a datatype property in the pizza example: hasDiameter.



Restrictions and Boolean Class Constructors

OWL	DL Symbol	Manchester OWL Syntax Keyword	Example
someValuesFrom	$\exists$	**some**	`hasChild some Man`
allValuesFrom	$\forall$	**only**	`hasSibling only Woman`
hasValue	$\ni$	**value**	`hasCountryOfOrigin value England`
minCardinality	$\geq$	**min**	`hasChild min 3`
cardinality	$=$	**exactly**	`hasChild exactly 3`
maxCardinality	$\leq$	**max**	`hasChild max 3`

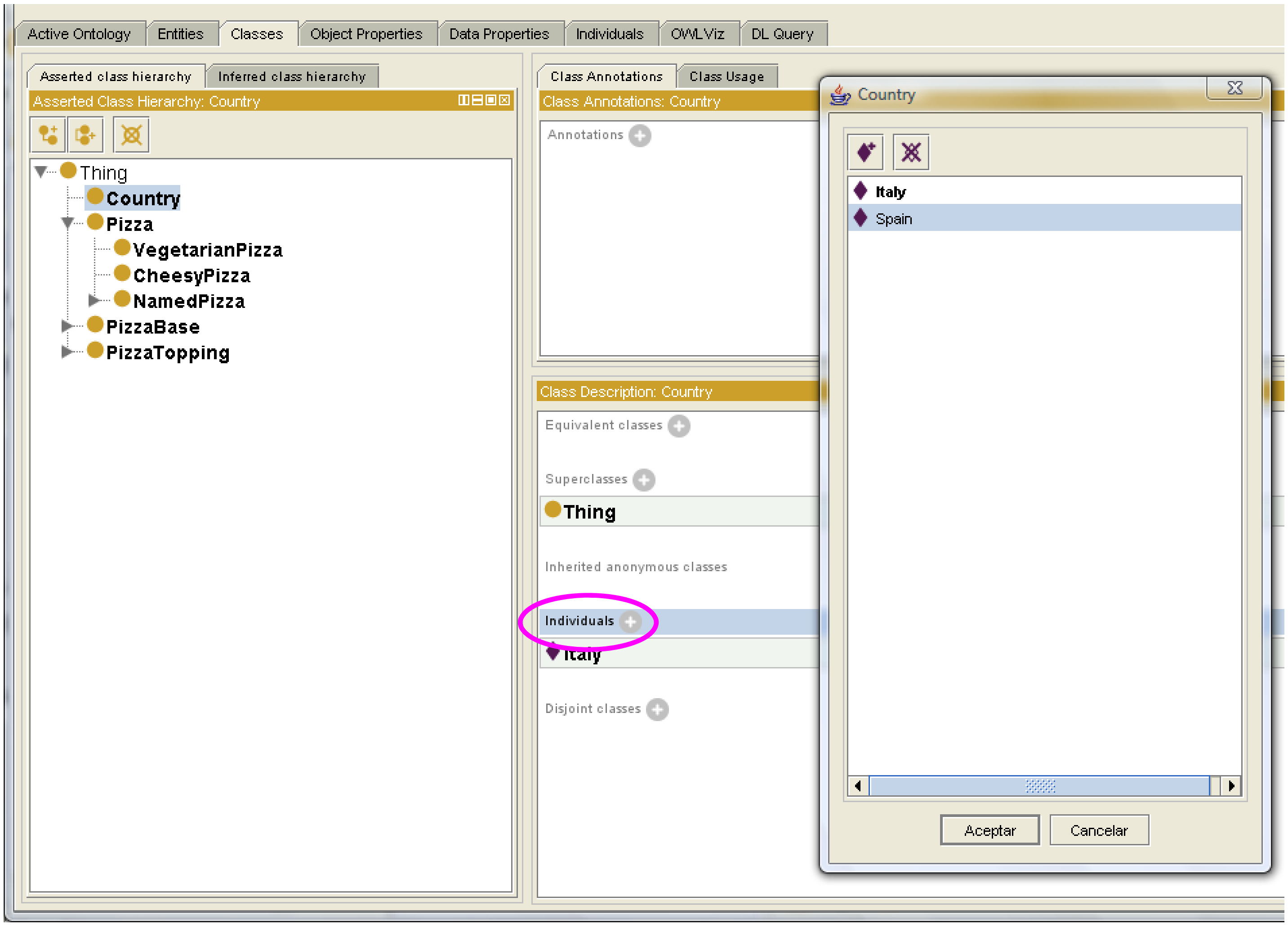
OWL	DL Symbol	Manchester OWL Syntax Keyword	Example
intersectionOf	$\sqcap$	**and**	`Doctor and Female`
unionOf	$\sqcup$	**or**	`Man or Woman`
complementOf	$\neg$	**not**	`not Child`

Exercise

Create a Meaty Pizza.
Create a Vegeterian Pizza, which have no meat and no fish toppings.
Create a Real Italian Pizza, which only have bases that are ThinandCrispy.
Create a subclass of Named Pizza with a topping of Mozzarela.

Individuals

Creating individuals of class Country.



hasValue Restriction

Specifying Italy as country of origin for Mozzarella.

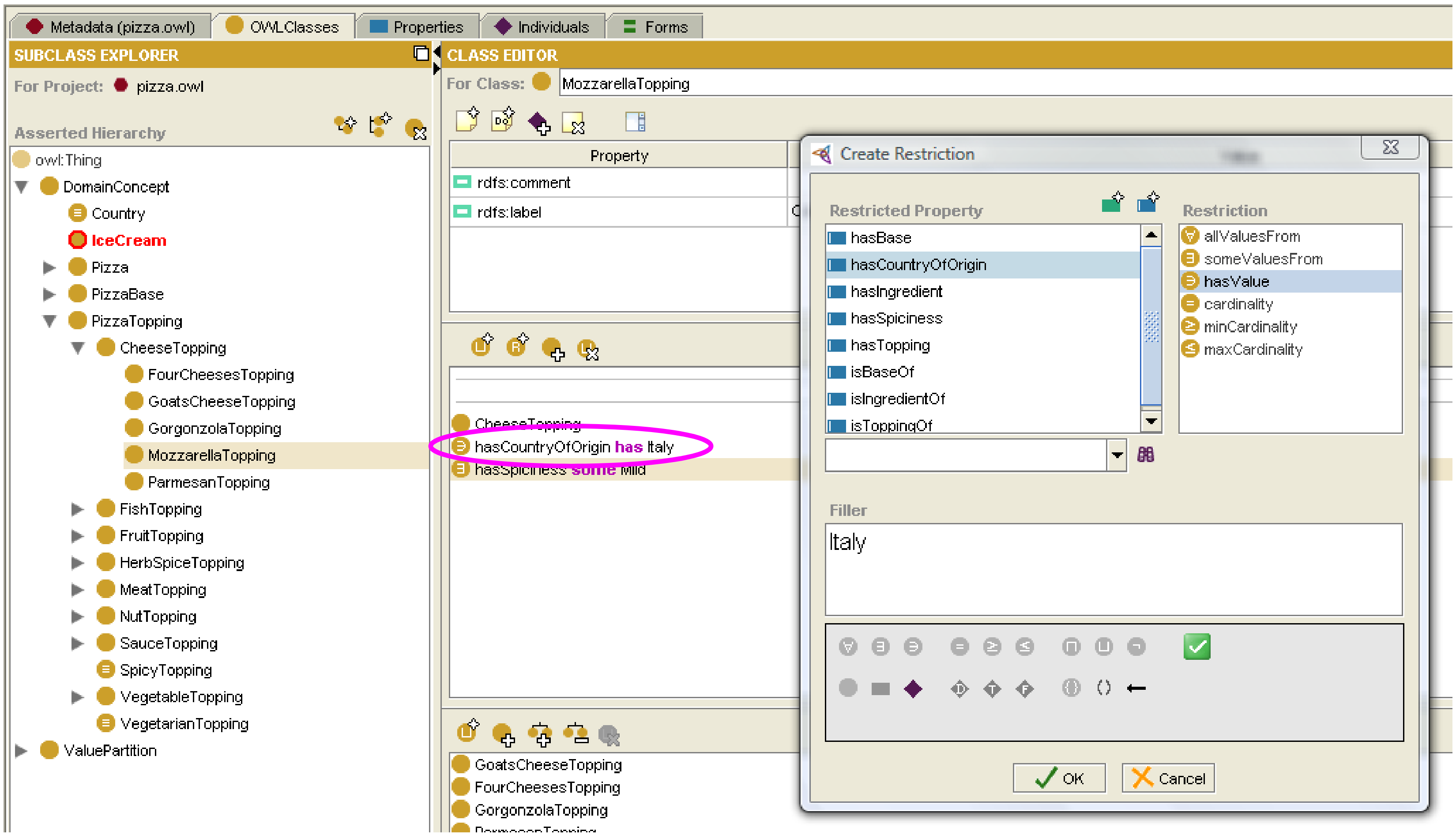


Table of Contents

Loading and Saving an Ontology

```
public class Example1 {

    public static void main(String[] args) {
        try {
            // A simple example of how to load and save an ontology
            // We first need to obtain a copy of an OWLOntologyManager, which, as the
            // name suggests, manages a set of ontologies.  An ontology is unique within
            // an ontology manager.  To load multiple copies of an ontology, multiple managers
            // would have to be used.
            OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
            // We load an ontology from a physical URI - in this case we'll load the pizza
            // ontology.
            URI physicalURI = URI.create("http://www.co-ode.org/ontologies/pizza/2007/02/12/pizza.owl");
            // Now ask the manager to load the ontology
            OWLOntology ontology = manager.loadOntologyFromPhysicalURI(physicalURI);
            // Print out all of the classes which are referenced in the ontology
            for(OWLClass cls : ontology.getReferencedClasses()) {
                System.out.println(cls);
            }
            // Now save a copy to another location in OWL/XML format (i.e. disregard the
            // format that the ontology was loaded in).
            // (To save the file on windows use a URL such as  "file:/C:\\windows\\temp\\MyOnt.owl")
            URI physicalURI2 = URI.create("file:/tmp/MyOnt2.owl");
            manager.saveOntology(ontology, new OWLXMLOntologyFormat(), physicalURI2);
            // Remove the ontology from the manager
            manager.removeOntology(ontology.getURI());
        }
        catch (OWLOntologyCreationException e) {
            System.out.println("The ontology could not be created: " + e.getMessage());
        }
        catch (OWLOntologyStorageException e) {
            System.out.println("The ontology could not be saved: " + e.getMessage());
        }
    }
}
```

Creating an Empty Ontology, Adding Axioms, and Saving (I)

```
public class Example2 {

    public static void main(String[] args) {
        try {
            // We first need to obtain a copy of an OWLOntologyManager, which, as the
            // name suggests, manages a set of ontologies.  An ontology is unique within
            // an ontology manager.  To load multiple copies of an ontology, multiple managers
            // would have to be used.
            OWLOntologyManager manager = OWLManager.createOWLOntologyManager();

            // All ontologies have a URI, which is used to identify the ontology.  You should
            // think of the ontology URI as the "name" of the ontology.  This URI frequently
            // resembles a Web address (i.e. http://...), but it is important to realise that
            // the ontology URI might not necessarily be resolvable.  In other words, we
            // can't necessarily get a document from the URI corresponding to the ontology
            // URI, which represents the ontology.
            // In order to have a concrete representation of an ontology (e.g. an RDF/XML
            // file), we MAP the ontology URI to a PHYSICAL URI.  We do this using a URIMapper

            // Let's create an ontology and name it "http://www.co-ode.org/ontologies/testont.owl"
            // We need to set up a mapping which points to a concrete file where the ontology will
            // be stored. (It's good practice to do this even if we don't intend to save the ontology).
            URI ontologyURI = URI.create("http://www.co-ode.org/ontologies/testont.owl");
            // Create a physical URI which can be resolved to point to where our ontology will be saved.
            URI physicalURI = URI.create("file:/tmp/MyOnt.owl");
            // Set up a mapping, which maps the ontology URI to the physical URI
            SimpleURIMapper mapper = new SimpleURIMapper(ontologyURI, physicalURI);
            manager.addURIMapper(mapper);

            // Now create the ontology - we use the ontology URI (not the physical URI)
            OWLOntology ontology = manager.createOntology(ontologyURI);
            // Now we want to specify that A is a subclass of B.  To do this, we add a subclass
            // axiom.  A subclass axiom is simply an object that specifies that one class is a
            // subclass of another class.
```

Creating an Empty Ontology, Adding Axioms, and Saving (II)

```
        // We need a data factory to create various object from.  Each ontology has a reference
        // to a data factory that we can use.
        OWLDataFactory factory = manager.getOWLDataFactory();
        // Get hold of references to class A and class B.  Note that the ontology does not
        // contain class A or classB, we simply get references to objects from a data factory that represent
        // class A and class B
        OWLClass clsA = factory.getOWLClass(URI.create(ontologyURI + "#A"));
        OWLClass clsB = factory.getOWLClass(URI.create(ontologyURI + "#B"));
        // Now create the axiom
        OWLAxiom axiom = factory.getOWLSubClassAxiom(clsA, clsB);
        // We now add the axiom to the ontology, so that the ontology states that
        // A is a subclass of B.  To do this we create an AddAxiom change object.
        AddAxiom addAxiom = new AddAxiom(ontology, axiom);
        // We now use the manager to apply the change
        manager.applyChange(addAxiom);

        // The ontology will now contain references to class A and class B - let's
        // print them out
        for(OWLClass cls : ontology.getReferencedClasses()) {
            System.out.println("Referenced class: " + cls);
        }
        // We should also find that B is a superclass of A
        Set<OWLDescription> superClasses = clsA.getSuperClasses(ontology);
        System.out.println("Superclasses of " + clsA + ":");
        for(OWLDescription desc : superClasses) {
            System.out.println(desc);
        }

        // Now save the ontology.  The ontology will be saved to the location where
        // we loaded it from, in the default ontology format
        manager.saveOntology(ontology);

    }
    catch (OWLException e) {
        e.printStackTrace();
    }
```

Adding an Object Property

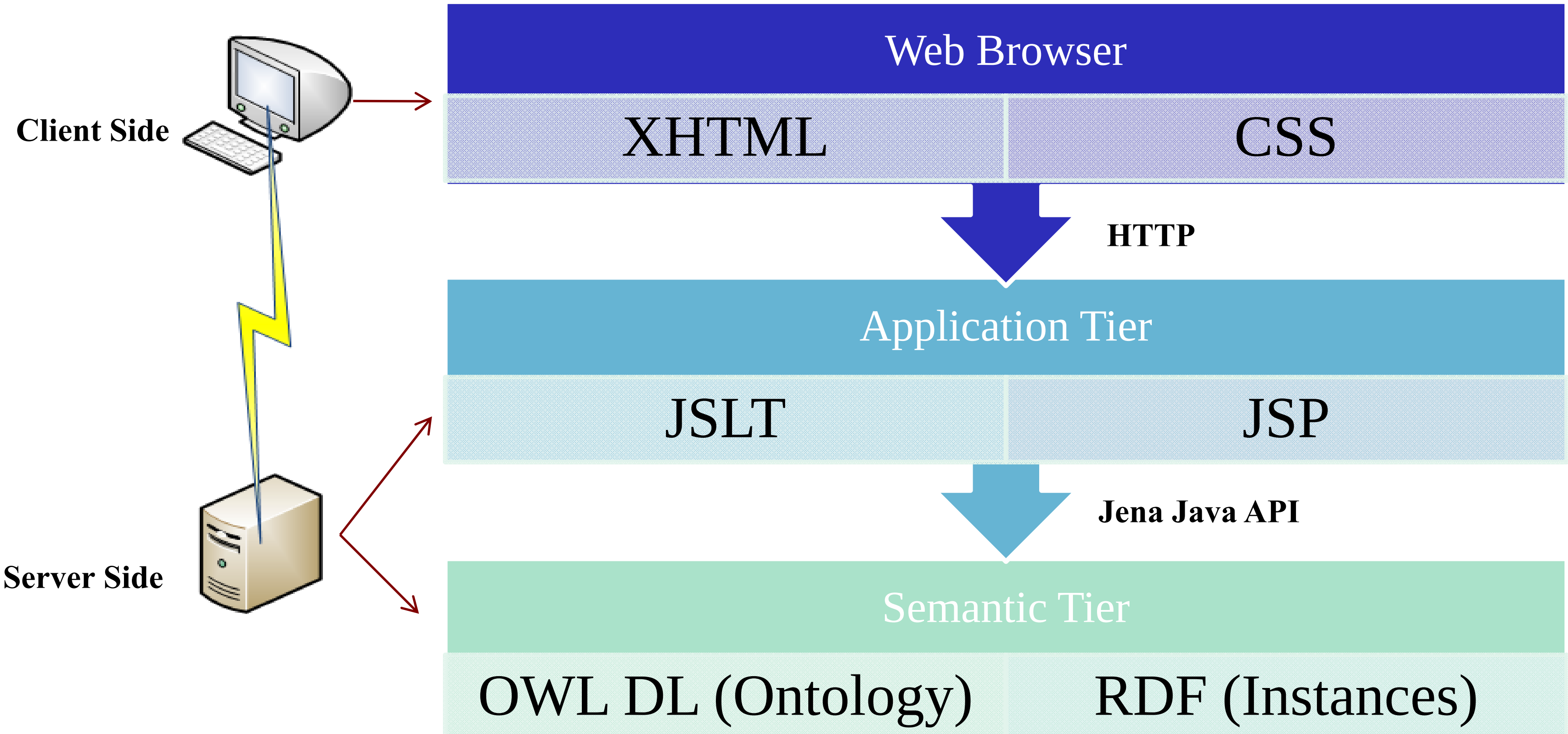
```java
public class Example4 {

    public static void main(String[] args) {
        try {
            OWLOntologyManager man = OWLManager.createOWLOntologyManager();

            String base = "http://www.semanticweb.org/ontologies/individualsexample";

            OWLOntology ont = man.createOntology(URI.create(base));

            OWLDataFactory dataFactory = man.getOWLDataFactory();

            // In this case, we would like to state that matthew has a father
            // who is peter.
            // We need a subject and object - matthew is the subject and peter is the
            // object.  We use the data factory to obtain references to these individuals
            OWLIndividual matthew = dataFactory.getOWLIndividual(URI.create(base + "#matthew"));
            OWLIndividual peter = dataFactory.getOWLIndividual(URI.create(base + "#peter"));
            // We want to link the subject and object with the hasFather property, so use the data factory
            // to obtain a reference to this object property.
            OWLObjectProperty hasFather = dataFactory.getOWLObjectProperty(URI.create(base + "#hasFather"));
            // Now create the actual assertion (triple), as an object property assertion axiom
            // matthew --> hasFather --> peter
            OWLObjectPropertyAssertionAxiom assertion = dataFactory.getOWLObjectPropertyAssertionAxiom(matthew, hasFather, peter);
            // Finally, add the axiom to our ontology and save
            AddAxiom addAxiomChange = new AddAxiom(ont, assertion);
            man.applyChange(addAxiomChange);

            man.saveOntology(ont, URI.create("file:/tmp/example.owl"));
        }
        catch (OWLOntologyCreationException e) {
            System.out.println("Could not create ontology: " + e.getMessage());
        }
        catch(OWLOntologyChangeException e) {
            System.out.println("Problem editing ontology: " + e.getMessage());
        }
```

Deleting Entities

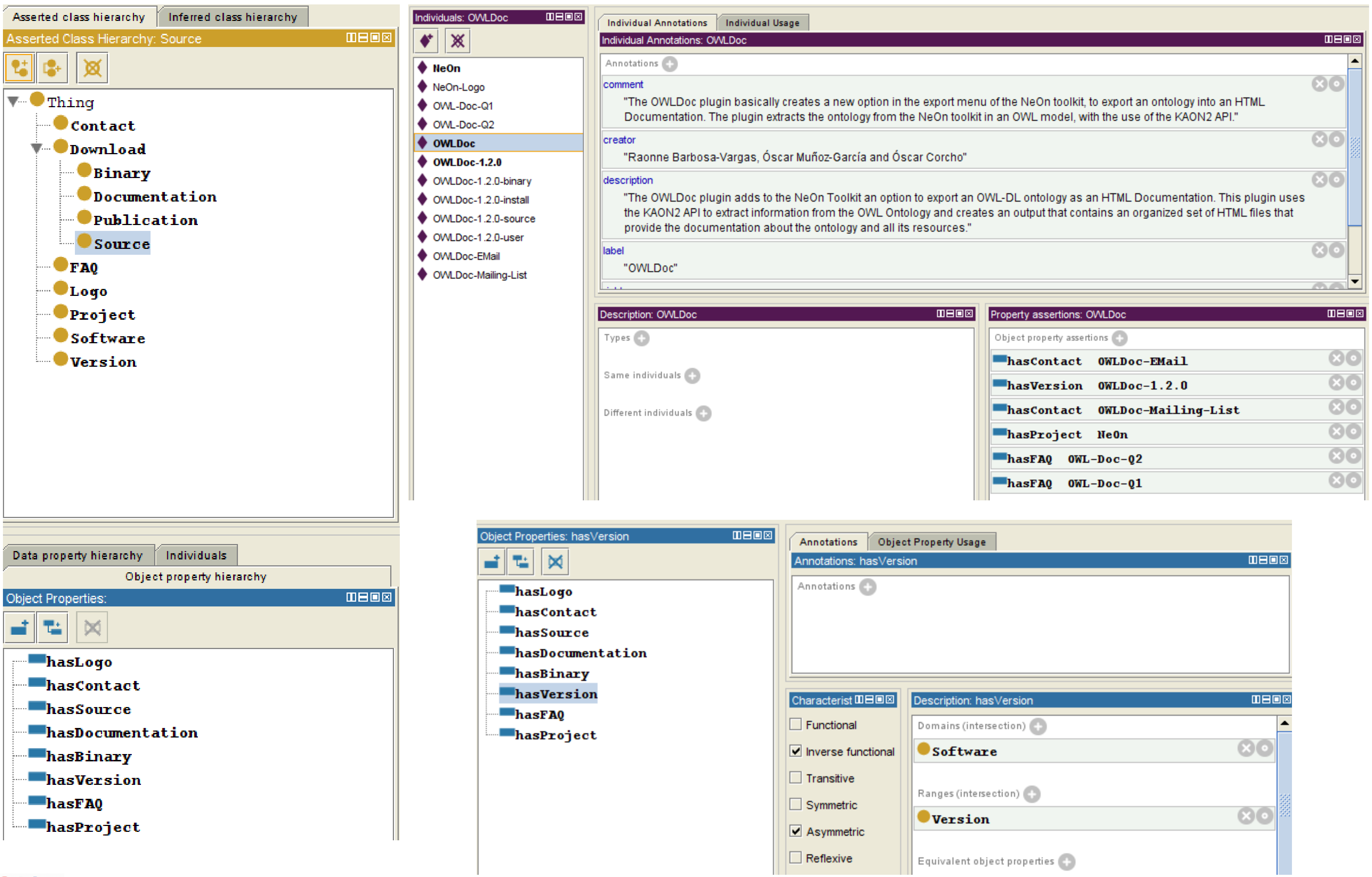
```
try {
    // The pizza ontology contains several individuals that represent
    // countries, which describe the country of origin of various pizzas
    // and ingredients.  In this example we will delete them all.
    // First off, we start by loading the pizza ontology.
    OWLOntologyManager man = OWLManager.createOWLOntologyManager();
    OWLOntology ont = man.loadOntologyFromPhysicalURI(URI.create("http://www.co-ode.org/ontologies/pizza/2007/02/12/pizza.owl"));

    // We can't directly delete individuals, properties or classes from an ontology because
    // ontologies don't directly contain entities -- they are merely referenced by the
    // axioms that the ontology contains.  For example, if an ontology contained a subclass axiom
    // SubClassOf(A, B) which stated A was a subclass of B, then that ontology would contain references
    // to classes A and B.  If we essentially want to "delete" classes A and B from this ontology we
    // have to remove all axioms that REFERENCE class A and class B (in this case just one axiom
    // SubClassOf(A, B)).  To do this, we can use the OWLEntityRemove utility class, which will remove
    // an entity (class, property or individual) from a set of ontologies.

    // Create the entity remover - in this case we just want to remove the individuals from
    // the pizza ontology, so pass our reference to the pizza ontology in as a singleton set.
    OWLEntityRemover remover = new OWLEntityRemover(man, Collections.singleton(ont));
    System.out.println("Number of individuals: " + ont.getReferencedIndividuals().size());
    // Loop through each individual that is referenced in the pizza ontology, and ask it
    // to accept a visit from the entity remover.  The remover will automatically accumulate
    // the changes which are necessary to remove the individual from the ontologies (the pizza
    // ontology) which it knows about
    for(OWLIndividual ind : ont.getReferencedIndividuals()) {
        ind.accept(remover);
    }
    // Now we get all of the changes from the entity remover, which should be applied to
    // remove all of the individuals that we have visited from the pizza ontology.  Notice that
    // "batch" deletes can essentially be performed - we simply visit all of the classes, properties
    // and individuals that we want to remove and then apply ALL of the changes afer using the
    // entity remover to collect them
    man.applyChanges(remover.getChanges());
    System.out.println("Number of individuals: " + ont.getReferencedIndividuals().size());
    // At this point, if we wanted to reuse the entity remover, we would have to reset it
```


Table of Contents

Application Architecture



Software Ontology



Example Code (logo.jsp)

```
<%@ page contentType="text/html; charset=utf-8" language="java" import="com.hp.hpl.jena.ontology.*, com.hp.hpl.jena.rdf.model.*"
 errorPage="" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page isELIgnored="false" %>

<%
    Individual project = (Individual) session.getAttribute("project");
    pageContext.setAttribute("title", project.getPropertyValue((AnnotationProperty) application.getAttribute("title")));
    ObjectProperty hasLogo = (ObjectProperty) application.getAttribute("hasLogo");
    AnnotationProperty identifier = (AnnotationProperty) application.getAttribute("identifier");
    OntResource logo = (OntResource) project.getPropertyValue(hasLogo);
    Literal logoIdentifier = (Literal) logo.getPropertyValue(identifier).as(Literal.class);
```

Jena Code

```
    pageContext.setAttribute("logoIdentifier", logoIdentifier);
    session.setAttribute("label", project.getLabel(""));
%>
```

```
<a href="<c:url value="${projectIdentifier.string}"/>">
    <img src="<c:url value="${logoIdentifier.string}"/>"
        alt="<c:out value="${label}"/>"
        longdesc="<c:out value="${title.string}"/>"
        style="border-style: none"/>
</a>
```

JSLT Code

Screenshot


OEG Software
Edición Visualización Historial Favoritos Ventana Ayuda
http://localhost:8080/oegsw/index.jsp;jsessionid=10196974EF6D3C20FB411C3EAEEC9F58?sw=http%3a%2f%2fwww.oeg-upm.net%2fsoftw
Google
Apple Yahoo! Google Maps YouTube Wikipedia News (310) Popular
Ontology Engineer ingGroup
OEG SOFTWARE
OWLDoc
NeOn
NeOn OWL-Doc Plugin
The OWLDoc plugin adds to the NeOn Toolkit an option to export an OWL-DL ontology as an HTML Documentation. This plugin uses the KAON2 API to extract information from the OWL Ontology and creates an output that contains an organized set of HTML files that provide the documentation about the ontology and all its resources.
NeOn OWL-Doc is open source and published under the XXXXX license.
More Information
Authors: Raonne Barbosa-Vargas, Óscar Muñoz-García and Óscar Corcho
Mailing List
ocorcho@fi.upm.es
Download
Version | Date | Binaries | Source | Documentation
NeOn OWL-DOC 1.2.0 | 7/4/2008 | Usage Manual How to install
FAQ
Is it possible to generate the documentation for an ontology in F-Logic?
No, it isn't. Although it is possible to create a F-Logic ontology using the NeOn Toolkit, OWL-Doc only generates documentation for ontologies expressed in OWL.
Can I use NeOn OWL-Doc as a stand-alone application?
No, you can't. NeOn OWL-Doc is a plugin that can only be deployed inside the NeOn Toolkit.
© NeOn: Lifecycle Support for Networked Ontologies (all rights reserved). This product development has been supported by NeOn, an Integrated Project funded by the European Commission. Further information available from http://www.neon-project.org. The information in this website is subject to a disclaimer notice.
W3C XHTML 1.1
W3C CSS