

UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE

MATEMATICHE FISICHE E NATURALI

Corso di Laurea in Informatica

Tesi di Laurea

**Uno strumento per la conversione da
OWL a Prolog**

Candidato:

Danilo Bonardi

Relatore:

Ing. Federico Bergenti

Anno Accademico 2005/2006

Indice

1	Introduzione	3
1.1	Semantic Web	3
1.2	Ontologie	4
2	Description Logic e OWL	6
2.1	Description Logic	6
2.2	Description language	7
2.3	Ragionatori Description Logic	9
2.4	OWL	10
2.4.1	OWL Lite	13
2.4.2	OWL DL	19
2.4.3	OWL Full	21
2.5	OWL e i Ragionatori	21
3	Conversione da <i>OWL DLP</i> a Prolog	23
3.1	Introduzione	23
3.2	Description Logic Programs	24
3.3	Operatore di Conversione	26
3.4	Un Esempio di Conversione	29
3.5	Safe Fragment	36

4	Strumento di Conversione	39
4.1	OwlDlp2Pl	39
4.2	Class Diagram	40
4.3	Ampliamenti del Tool	40
4.4	Strumenti Utilizzati	42
5	Conclusioni	43
	Bibliografia	43

Capitolo 1

Introduzione

Il presente lavoro di tesi ha come obiettivo l'implementazione di un tool di conversione da *OWL*, un linguaggio per la scrittura di ontologie, a Prolog sfruttando le recenti tecnologie per il *Semantic Web*. La conversione si basa su due pubblicazioni, la prima tratta la conversione da una particolare logica descrittiva a Datalog [UH04], la seconda l'intersezione di OWL con la logica del primo ordine [PH05]. Il progetto si colloca nel contesto del *Semantic Web* con l'obiettivo di generare uno strumento utile alla scrittura e alla verifica di documenti ontologici sfruttando un interprete Prolog. Il progetto in questione è stato pensato per essere ampliabile ad altri linguaggi logici oltre al Prolog o ad essere convertito a ragionatore *OWL*.

1.1 Semantic Web

Il Web nasce come un'immensa rete di informazioni disponibili in diversi formati e lingue. Il passo successivo per la rete sarà rendere interrogabile, interpretabile ed elaborabile in modo automatico, ogni informazione ivi contenuta.

Questo passo porta con se non pochi problemi dovuti all'aspetto decentralizzato

del Web, che comporta, tra le molte problematiche, la duplicazione delle informazioni e difficoltà nel capire se più documenti fanno riferimento allo stesso soggetto. Il *Semantic Web* è collegato con la teoria degli *agenti* e il loro utilizzo nell'ambito della rete. Il ruolo degli agenti nel Web semantico è di fornire le più vaste capacità d'*inferenza*. Un esempio tipico di quello che si vuole ottenere è un agente in grado di capire la patologia di una persona, contattare il giusto centro medico e prendere appuntamento.

Il *Semantic Web* è legato all'*XML: L'eXtensible Markup Language* è un linguaggio atto alla definizione di entità e alla marcatura di proprietà in documenti. Al momento il *Semantic Web* è uscito dall'ambito accademico ed ha accolto l'interesse di alcune grandi aziende dell'ICT, nonostante ci siano ancora molti passi per arrivare ad un linguaggio globale di espressione dei dati e a regole universali per i processi d'*inferenza*.

1.2 Ontologie

Tom Gruber [Gru] definisce un'ontologia come la specifica di una concettualizzazione. Un'ontologia è una descrizione dei concetti e delle relazioni esistenti comprensibile ad un agente o ad una comunità di agenti.

Il termine ontologia è stato mutato da un concetto filosofico che sottende lo studio dell'essere o dell'esistere, delle fondamentali categorie e delle relazioni tra esse.

Un'ontologia può essere utilizzata per diversi scopi, tra cui il ragionamento induttivo, la classificazione e alcune tecniche di problem solving.

Legato alle basi di conoscenza ed alle ontologie è il concetto di *inferenza*. Questo termine indica il meccanismo di deduzione di nuove informazioni da una base di conoscenza.

Il World Wide Web Consortium (W3C) si è mosso da tempo in direzione del *Se-*

semantic Web, studiando standard per la definizione di ontologie. Uno dei primi linguaggi ad essere utilizzato per modellare la conoscenza è il *Resource Description Framework*. *RDF* si basa su tre principi chiave:

- Qualunque cosa può essere identificato da un *URI* (*Uniform Resource Identifier*, stringa che definisce in modo univoco una risorsa, sia questa un indirizzo Web, un documento, un'immagine, ecc...);
- Utilizzare il linguaggio meno espressivo possibile per definire qualunque cosa;
- Qualunque cosa può dire qualunque cosa su qualsiasi altra cosa.

Il modello di dati *RDF* è formato da risorse, proprietà e valori. Le proprietà sono delle relazioni che legano tra loro risorse e valori e sono anch'esse identificate da *URI*. Un valore invece può essere una risorsa o un tipo di dato primitivo (valori numerici, stringhe, ecc...).

Sulle basi di *RDF* viene costruito *OWL* (*Ontology Web Language*). Mentre *RDF* viene considerato solo come mezzo di rappresentazione, *OWL* è lo strumento giusto per rappresentare una base di conoscenza da cui inferire informazioni. Essendo costruito su *RDF*, anche *OWL* si basa sui concetti di risorsa, proprietà e valore. In *OWL* le risorse prendono il nome di classi mentre i valori vengono chiamati istanze o individui. Le caratteristiche di *OWL* verranno descritte in modo approfondito nel capitolo successivo.

Capitolo 2

Description Logic e OWL

2.1 Description Logic

Ian Horrocks [Hor06] definisce la famiglia delle logiche descrittive (*DL*) come le logiche che descrivono un dominio in termini di concetti (classi), ruoli (proprietà e relazioni) e individui. Si distinguono per il loro utilizzo della logica del primo ordine (*FOL*: First Order Logic). Le *DL* forniscono procedure decisionali utili per i processi d'*inferenza*.

Nelle *DL* si fa una distinzione tra due insiemi di sentenze: Le *TBox* (Terminological Box) e le *ABox* (Assertional Box).

La *TBox* è l'insieme degli assiomi sulle classi e delle loro relazioni (lo schema).

Un esempio di *TBox* può essere l'asserzione:

- L'insieme degli animali contiene quello dei cani

Si fa riferimento ad *ABox* per indicare l'insieme dei dati della base di conoscenza.

Per esempio:

- Fido è un Cane

Un'ontologia è formata dall'unione di una *TBox* con una *ABox*. Questa distinzione è utile perchè differenzia tra la visione del dominio in quanto a concetti astratti e la visione come insieme di entità. Inoltre nella costruzione di una procedura decisionale efficiente può essere importante conoscere la complessità della *ABox* rispetto alla *TBox* o viceversa.

Di particolare interesse per questo lavoro di tesi sono le logiche descrittive della famiglia \mathcal{SH} . I professori Horrocks, Patel-Schneider e van Harmelen [IH05] differenziano le \mathcal{SH} dalle altre DL in quanto includono i connettivi booleani di unione, intersezione e complemento, le restrizioni, la gerarchia nelle proprietà e le relazioni transitive. Membri importanti della famiglia \mathcal{SH} sono \mathcal{SHIQ} , che ha introdotto l'inverso di una proprietà e le restrizioni di cardinalità, e \mathcal{SHOQ} che aggiunge la possibilità di definire una classe come enumerazione di istanze e il supporto per i tipi di dato concreti (interi, stringhe, ...).

2.2 Description language

Le logiche descrittive vengono espresse in modo formale tramite i linguaggi descrittivi. I description language sono dei formalismi che permettono di rappresentare i descrittori elementari di un'ontologia, quali concetti e ruoli atomici, per costruirne induttivamente di più complessi.

Useremo le lettere A e R per indicare concetti e ruoli atomici, mentre C e D saranno usate per indicare dei concetti astratti generici.

I description language vengono distinti a seconda dei costruttori che forniscono. Di interesse è la famiglia dei linguaggi \mathcal{AL} (*Attributive Language*), questa fornisce i requisiti minimi per la rappresentazione dei costrutti comunemente usati nelle logiche descrittive.

Un descrittore di concetto nelle \mathcal{AL} può essere:

- A : un concetto atomico;
- \top : concetto universale o concetto top;
- \perp : concetto bottom;
- $\neg A$: negazione di un concetto atomico;
- $C \sqcap D$: intersezione di descrittori;
- $\forall R.C$: restrizione del valore di un ruolo;
- $\exists R.\top$: quantificatore esistenziale limitato.

Da notare come in \mathcal{AL} la negazione può essere applicata solo ai concetti atomici e solo al concetto universale è concesso essere oggetto del quantificatore esistenziale. Per dare una semantica formale ai concetti aspressi tramite un linguaggio \mathcal{AL} consideriamo l'interpretazione \mathcal{I} che consiste in un insieme non vuoto $\Delta^{\mathcal{I}}$ e una funzione di interpretazione che assegna:

- Ad ogni concetto atomico A un insieme $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$;
- Ad ogni ruolo atomico R una relazione binaria $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

La funzione \mathcal{I} interpreta i descrittori di concetti tramite la seguente definizione induttiva:

$$\begin{aligned}\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\ \perp^{\mathcal{I}} &= \emptyset \\ (\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \\ (C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}}\end{aligned}$$

$$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$$

$$(\exists R.\top)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\}.$$

Si dice che C e D sono equivalenti ($C \equiv D$) se $C^{\mathcal{I}} = D^{\mathcal{I}}$ per ogni interpretazione \mathcal{I} . Si scrive invece $C \sqsubseteq D$ per indicare relazioni gerarchiche tra descrittori, questo descrittore viene interpretato come $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Alcuni membri della famiglia \mathcal{AL} aggiungono ai possibili descrittori l'unione e il complemento di un descrittore generico (non solo atomico).

$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

Un ulteriore ampliamento alle \mathcal{AL} sono le restrizioni di cardinalità, rappresentate con $\geq n R$ e $\leq n R$.

$$(\geq n R)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid [b \mid (a,b) \in R^{\mathcal{I}}] \geq n\}$$

$$(\leq n R)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid [b \mid (a,b) \in R^{\mathcal{I}}] \leq n\}$$

2.3 Ragionatori Description Logic

Un ragionatore permette di derivare nuovi concetti da un'ontologia, in particolare possono essere usati per:

- Rivelare relazioni di ereditarietà tra classi;
- Ricercare individui con determinate caratteristiche;
- Determinare situazioni di inconsistenza all'interno di un'ontologia.

Per interrogare un'ontologia, un ragionatore può offrire vari mezzi, tra questi i più utilizzati sono alcuni linguaggi di query come *RDQL* (RDF Data Query Language) e *SPARQL* (Protocol and RDF Query Language) oppure un'interfaccia *DIG* [DIG] (sviluppata dal DL Implementation Group) utilizzabile mediante alcuni tool per la costruzione e la gestione di ontologie come Protégé.

2.4 OWL

L'Ontology Web Language viene introdotto dal *W3C* come la migliore alternativa per rappresentare ontologie. *OWL* è un linguaggio di markup che nasce dall'unione di tre famiglie di descrittori, *DAML+OIL* (DARPA Agent Markup Language e Ontology Inference Layer), *SHOE* (Simple HTML Ontology Extensions) e *RDF*. Ci si riferisce ad *OWL* come ad un'estensione di vocabolario di *RDF* in quanto ogni documento *OWL* è di fatto anche un documento *RDF*. *XML* e *XML-SCHEMA* danno una struttura sintattica ad *OWL* senza limitarne l'espressività. *OWL* è stato costruito appositamente per fornire il miglior compromesso tra semplicità di pubblicazione ed elaborazione di contenuti tramite un ragionatore.

OWL ha al suo interno tre sottolinguaggi: Lite, DL e Full. Tenendo questo ordine si può dire che sono uno l'estensione dell'altro.

- Ogni ontologia espressa in *OWL Lite* è valida anche per *OWL DL*;
- Ogni ontologia espressa in *OWL DL* è valida anche per *OWL Full*;
- Ogni conclusione tratta da un'ontologia *OWL Lite* è una conclusione valida anche per *OWL DL*;
- Ogni conclusione tratta da un'ontologia *OWL DL* è una conclusione valida anche per *OWL Full*.

OWL Lite e *DL* sono considerati logiche descrittive in quanto *OWL* stesso è basato sulla famiglia *SH* delle logiche descrittive.

Fatte queste precisazioni vedremo i dettagli dei tre sottolinguaggi, con i rispettivi costrutti utilizzabili, nel seguito.

RDF impone ad un documento *OWL* di iniziare con un'intestazione simile alla seguente.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
]>
```

Questo header precisa il tipo di sintassi usata (*XML 1.0*), seguito dalla dichiarazione di un documento *RDF* con i vocabolari in esso utilizzati, in questo caso *OWL* e *XSD* (XML Schema Definition, fornisce il namespace per *XML* e alcuni tipi di dato).

Oltre a queste definizioni è necessario inserire nell'ontologia l'*URI* che vorremo assegnarle. Questo è necessario perchè ogni entità che andremo a definire all'interno di *OWL* sarà inserita in un namespace identificato dall'*URI* in questione. Questa pratica rientra nella filosofia del *Semantic Web* in cui ogni risorsa si vorrebbe identificata da un *URI*.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.owl-ontologies.com/ontologia.owl#">
```

L'ultima voce fa riferimento all'*URI* dell'ontologia, le altre sono utili abbreviazioni per i namespace che utilizzeremo nel documento. Per esempio quando si vuole fare riferimento ad un elemento del vocabolario *OWL* si utilizzerà il prefisso `&owl;`.

OWL prevede la possibilità di aggiungere alcune annotazioni ad ogni entità generata e all'ontologia stessa. Le annotazioni possono essere marcate a piacere seguendo alcune regole. Alcune annotazioni sono predefinite:

- `owl:versionInfo`: Aggiunge ad un oggetto o all'ontologia una stringa di caratteri che ne specificano la versione.

```
<owl:Ontology rdf:about="">
  ...
  <owl:versionInfo> 1.0 </owl:versionInfo>
  ...
</owl:Ontology>
```

- `owl:backwardCompatibleWith`: Contiene un link ad un'ontologia, si usa per indicare una versione precedente dell'ontologia corrente con cui viene conservata la compatibilità. La compatibilità impone che tutti gli identificatori usati nella versione precedente abbiano la stessa interpretazione nell'ontologia corrente.

```
<owl:Ontology rdf:about="">
  ...
  < owl:backwardCompatibleWith >http://owl.com/old.owl
  </owl:backwardCompatibleWith >
  ...
</owl:Ontology>
```

- `rdfs:label`: Viene solitamente usata per dare un nome più leggibile ad una risorsa.

```
<rdfs:Class rdf:ID="FiglioDiUnDottore">
  <rdfs:label>Figlio di Un Dottore</rdfs:label>
  <rdfs:subClassOf rdf:resource="&rdfs;Class"/>
</rdfs:Class>
```

- `rdfs:comment`: Consente di aggiungere un commento di carattere generale ad un oggetto o all'ontologia.

```

<owl:Class rdf:ID="Autovettura">
  <rdfs:comment>Meglio usare Automobile</rdfs:comment>
  <owl:equivalentClass rdf:resource="#Automobile"/>
</owl:Class>

```

- `rdfs:seeAlso`: Solitamente usato per indicare ontologie correlate.

```

<owl:Ontology rdf:about="">
  ...
  <rdfs:seeAlso>http://owl.com/wine.owl</rdfs:seeAlso>
  ...
</owl:Ontology>

```

2.4.1 OWL Lite

OWL Lite contiene il numero minimo di istruzioni sufficienti per definire una gerarchia di Classi con alcuni vincoli. La scelta di uno scrittore di ontologie può ricadere su *OWL Lite* se non c'è bisogno di tutta l'espressività di *OWL* per esprimere un certo dominio o se si vuole sfruttare le migliori prestazioni che l'utilizzo di un linguaggio ristretto offre quando elaborato da un ragionatore.

Vediamo ora i costrutti utilizzabili in Lite, seguendo l'ordine impostato nel reference *W3C* [W3C04], con le rispettive limitazioni ed esempi.

- `Class`: Definisce una classe. In *OWL* tutte le classi ereditano da una classe `Thing`. È prevista anche una classe `Nothing` che non ha istanze ed è sottoclasse di tutte le classi definite.

```

<owl:Class rdf:ID="Persona"/>

```

- `rdfs:subClassOf`: Permette di definire una classe come sottoclasse di un'altra.

```

<owl:Class rdf:ID="Impiegato">
  <rdfs:subClassOf rdf:resource="#Persona"/>
</owl:Class>

```

- `rdf:Property`: Una proprietà è una relazione tra individui o tra individui e particolari valori (stringhe, interi, reali, ecc...). Nel primo caso la relazione viene marcata con `ObjectProperty`, altrimenti con `DatatypeProperty`.

```

<owl:ObjectProperty rdf:ID="Padre_di">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="Anno_di_nascita">
  <rdfs:domain rdf:resource="#Persona"/>
  <rdfs:range rdf:resource="#xsd:positiveInteger"/>
</owl:DatatypeProperty>

```

- `rdfs:subPropertyOf`: Analogamente a quanto visto per le classi, anche per le proprietà è possibile definire delle gerarchie.

```

<owl:ObjectProperty rdf:ID="Padre_dell_impegnato">
  <rdfs:subPropertyOf rdf:resource="#Padre_di"/>
  <rdfs:domain rdf:resource="#Persona"/>
  <rdfs:range rdf:resource="#Impiegato"/>
</owl:ObjectProperty>

```

- `rdfs:domain`: Specifica il dominio di una `Property` e serve per limitare il numero di individui a cui può essere applicata. Il dominio della proprietà `Colore_Telaio` potrebbe essere la classe `Automobile`.

```

<owl:ObjectProperty rdf:ID="Colore_Telaio">
  ...
  <rdfs:domain rdf:resource="#Automobile"/>
  ...
</owl:ObjectProperty>

```

- `rdfs:range`: In modo simile al dominio, il `range` serve per limitare i valori possibili di una `Property`. Nel precedente esempio il `range` sarà la classe `Colore`.

```

<owl:ObjectProperty rdf:ID="Colore_Telaio">
  ...

```

```

        <rdfs:range rdf:resource="#Colore"/>
        ...
    </owl:ObjectProperty>

```

- **Individual:** Un individuo è un'istanza di una classe.

```

<Persona rdf:ID="Aldo">
    <nome rdf:datatype="&xsd:string">Aldo</nome>
    <anno_di_nascita
    rdf:datatype="&xsd:positiveInteger">1983
    </anno_di_nascita>
    <padre_di rdf:resource="#Luigi"/>
</Persona>

```

- **equivalentClass:** Istruzione per definire uguali due classi.

```

<owl:Class rdf:about="#Premier">
    <equivalentClass
    rdf:resource="#Presidente_del_consiglio"/>
</owl:Class>

```

- **equivalentProperty:** Istruzione per definire identiche due relazioni.

```

<owl:ObjectProperty rdf:ID="Padre_di">
    <owl:inverseOf rdf:resource="#ha_figlio"/>
</owl:ObjectProperty>

```

- **sameAs :** Marca due individui come lo stesso individuo.

```

<Persona rdf:about="#Rodman_Edward_Serling">
    <owl:sameAs rdf:resource="#Rod_Serling"/>
</Persona>

```

- **differentFrom:** Forza la differenziazione di un individuo rispetto ad un altro, o rispetto ad un gruppo. L'uso di `differentFrom` forza il controllo da parte del ragionatore che gli individui siano effettivamente differenti.

```

<Persona rdf:about="#Aldo">
    <owl:differentFrom rdf:resource="#Luigi"/>
    <owl:differentFrom rdf:resource="#Mario"/>
</Persona>

```

- **AllDifferent**: Il funzionamento di `AllDifferent` è analogo a quello di `differentFrom`, ma permette di definire una differenziazione per un gruppo di individui. Il costrutto `owl:distinctMembers` è un'istruzione di servizio per indicare delle enumerazioni.

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Persona rdf:about="#Aldo"/>
    <Persona rdf:about="#Mario"/>
    <Persona rdf:about="#Luigi"/>
  </owl:distinctMembers>
</owl:AllDifferent>
```

- **inverseOf**: Definisce una `ObjectProperty` come l'inverso di un'altra.

```
<owl:ObjectProperty rdf:ID="Figlio_di">
  <owl:inverseOf rdf:resource="#Padre_di"/>
</owl:ObjectProperty>
```

- **TransitiveProperty**: Definisce una `ObjectProperty` come soddisfacente la proprietà transitiva.

```
<owl:TransitiveProperty rdf:ID="Sotto_regione">
  <rdfs:domain rdf:resource="#Regione"/>
  <rdfs:range rdf:resource="#Regione"/>
</owl:TransitiveProperty>
```

- **SymmetricProperty**: Definisce una `ObjectProperty` come soddisfacente la proprietà simmetrica.

```
<owl:SymmetricProperty rdf:ID="Amico_di">
  <rdfs:domain rdf:resource="#Persona"/>
  <rdfs:range rdf:resource="#Persona"/>
</owl:SymmetricProperty>
```

- **FunctionalProperty**: Definisce una `ObjectProperty` come una funzione, quindi ad ogni valore del `domain` corrisponde al più un solo valore del `range`.

```
<owl:ObjectProperty rdf:ID="Marito_di">
  <rdf:type rdf:resource="#owl:FunctionalProperty"/>
```

```

        <rdfs:domain rdf:resource="#Uomo"/>
        <rdfs:range rdf:resource="#Donna"/>
    </owl:ObjectProperty>

```

- **InverseFunctionalProperty**: Definisce una `ObjectProperty` come una relazione ovunque definita.

```

<owl:InverseFunctionalProperty rdf:ID="Madre_biologica">
    <rdfs:domain rdf:resource="#Donna"/>
    <rdfs:range rdf:resource="#Persona"/>
</owl:InverseFunctionalProperty>

```

- **allValuesFrom**: Restringe il range di una `ObjectProperty` ad una determinata classe. Questo tipo di restrizione viene usata quando si crea una sottoclasse o una sottoproprietà.

```

<owl:Class rdf:ID="Europeo">
    <rdfs:subClassOf rdf:resource="#Persona"/>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#Stato_di_nascita"/>
        <owl:allValuesFrom rdf:resource="#Europa" />
    </owl:Restriction>
</owl:Class>

```

- **someValuesFrom**: Restringe il range di una `ObjectProperty` imponendo che, in un'istanza, almeno un valore della proprietà appartenga ad una determinata classe. Nell'esempio vogliamo definire la classe delle persone di cui almeno uno dei genitori è un medico.

```

<owl:Class rdf:ID="Figlio_di_medico">
    <rdfs:subClassOf rdf:resource="#Persona"/>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#Genitore"/>
        <owl:someValuesFrom rdf:resource="#Medico" />
    </owl:Restriction>
</owl:Class>

```

- **minCardinality**: Consente di indicare il numero minimo di volte che una determinata proprietà deve essere utilizzata per ogni istanza di una classe.

Nel caso di *OWL Lite* il numero può essere 0 o 1, nel caso la proprietà debba essere utilizzata per ogni istanza della classe almeno una volta.

```
<owl:Class rdf:ID="Persona">
  ....
  <owl:Restriction>
    <owl:onProperty rdf:resource="#Genitore"/>
    <owl:minCardinality
      rdf:datatype="&xsd;nonNegativeInteger">2
    </owl:minCardinality>
  </owl:Restriction>
</owl:Class>
```

- **maxCardinality**: Precisa il numero massimo di volte che una proprietà può essere utilizzata in una classe. In *OWL Lite* i valori utilizzabili sono 0 o 1.

```
<owl:Class rdf:ID="Persona">
  ....
  <owl:Restriction>
    <owl:onProperty rdf:resource="#Genitore"/>
    <owl:maxCardinality
      rdf:datatype="&xsd;nonNegativeInteger">2
    </owl:maxCardinality>
  </owl:Restriction>
</owl:Class>
```

- **cardinality**: Stabilisce il numero preciso di volte che una determinata proprietà deve essere utilizzata all'interno di una classe. Nel caso di *OWL Lite* può assumere solo i valori 0 o 1.

```
<owl:Class rdf:ID="Persona">
  ....
  <owl:Restriction>
```

```

        <owl:onProperty rdf:resource="#Genitore"/>
        <owl:cardinality
            rdf:datatype="&xsd;nonNegativeInteger">2
        </owl:cardinality>
    </owl:Restriction>
</owl:Class>

```

- **intersectionOf**: Consente di definire una classe come intersezione di altre classi.

```

<owl:Class rdf:ID="Impiegato_uomo">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Impiegato"/>
        <owl:Class rdf:about="#uomo"/>
    </owl:intersectionOf >
</owl:Class>

```

2.4.2 OWL DL

OWL Description Logic (DL) offre la massima espressività garantendo la *completezza computazionale* (le deduzioni sull'ontologia sono tutte complete) e la *decidibilità* (tutte le deduzioni finiscono in un tempo finito) [W3C04]. In *OWL DL* possono essere usati tutti i costrutti definiti dal vocabolario *OWL*, *OWL Full* ne amplia solo le potenzialità. I Costrutti di *OWL* sono tutti quelli elencati per *OWL Lite*, tranne le restrizioni per le cardinalità, in aggiunta a quelli di seguito elencati.

- **oneOf**: Permette di descrivere una classe come insieme di un numero finito di istanze. Per esempio è possibile definire la classe **Europa** come l'insieme contenente tutti gli stati europei.

```

<owl:Class rdf:ID="Europa">
    <owl:oneOf rdf:parseType="Collection">
        <Stati rdf:about="#Italia"/>
        <Stati rdf:about="#Francia"/>
        <Stati rdf:about="#Germania"/>
    </owl:oneOf >
</owl:Class>

```

```

        . . . .
    </owl:oneOf>
</owl:Class>

```

- **hasValue:** Limita il range di una proprietà ad una sola istanza di una classe.

```

<owl:Class rdf:ID="Figlio_di_Aldo">
  <rdfs:subClassOf rdf:resource="#Persona"/>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#Figlio_di"/>
    <owl:hasValue rdf:resource="#Aldo"/>
  </owl:Restriction>
</owl:Class>

```

- **disjointWith:** Indica una classe come disgiunta dalle altre, forzando il ragionatore a verificare che un'istanza non possa appartenere ad entrambe. Per esempio si potrebbe voler dichiarare che la classe degli uomini e quella delle donne sono disgiunte.

```

<owl:Class rdf:ID="#Uomo">
  <owl:disjointWith rdf:resource="#Donna"/>
</owl:Class>

```

- **unionOf:** Definisce una classe come unione di due o più classi.

```

<owl:Class rdf:ID="Anziano_o_bambino">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Bambino"/>
    <owl:Class rdf:about="#Anziano"/>
  </owl:unionOf>
</owl:Class>

```

- **complementOf:** Definisce una classe come il complemento di un'altra. Può essere utile per stabilire che la classe `bambino` è complementare alla classe `adulto`.

```

<owl:Class rdf:ID"#Bambino">
  <owl:complementOf>

```

```
        <owl:Class rdf:about="#Adulto"/>
    </owl:complementOf>
</owl:Class>
```

2.4.3 OWL Full

OWL Full dà la possibilità di sfruttare tutta l'espressività di *RDF*, senza però offrire delle garanzie computazionali. Infatti *OWL Full* non garantisce né la *completezza computazionale* né la *decidibilità*. Al momento, non esistono ragionatori che possano sfruttare tutte le potenzialità espressive di *OWL Full* e per questo viene poco considerato dagli scrittori di ontologie.

In *OWL Full* una `owl:Class` coincide con `rdfs:Class`. Questo consente di poter utilizzare una classe come un individuo. È perfettamente legale dichiarare il nome `Fokker-100` come identificante sia la classe del tipo di aeroplano Fokker-100, che un'istanza della classe `Tipi_di_Areoplani`.

Inoltre tutti i valori numerici sono considerati parte dell'insieme degli individui. Questo implica che non ci sia alcuna distinzione tra le `DatatypeProperty` e le `ObjectProperty`.

2.5 OWL e i Ragionatori

Ad oggi sono disponibili alcuni ragionatori che supportano OWL.

- **Pellet** [Pel06]: Rationatore open source solo per ontologie *OWL DL* con interfaccia *DIG* e supporto per *RDQL* e *SPARQL*.
- **Fact++** [Fac06]: Progetto open source per ontologie *OWL DL* con solo il supporto per *DIG*.
- **Kaon2** [Kao05]: Software proprietario con interfaccia *DIG* e supporto per *SPARQL*.

- **RacerPro** [Rac06]: Rationatore proprietario molto ricco e completo che usa un linguaggio di interrogazione proprietario.

Capitolo 3

Conversione da *OWL DLP* a Prolog

3.1 Introduzione

L'idea di creare un tool per convertire un'ontologia *OWL DL* in Prolog nasce da *Dlpconvert*, una utility per *OWL DL* che permette la conversione in Datalog. *Dlpconvert* fa parte di un set di tool sviluppati tramite la tecnologia di *Kaon2*, ed è per questo un progetto a sorgente chiuso.

Dlpconvert converte solo una parte dei costrutti disponibili in *OWL DL*, questo sottoinsieme prende il nome di *DLP Fragment*. Inoltre *Dlpconvert* basa la sua conversione su una pubblicazione del 2004 [UH04] su come ridurre una logica descrittiva *SHIQ⁻* a Datalog. I primi due dei cinque passaggi adoperati per la conversione portano a tradurre la logica descrittiva in una logica del primo ordine, tramite un operatore definito per questo scopo. Parte centrale del lavoro fatto per sviluppare la traduzione da *OWL DL* a Prolog è quella di implementare l'operatore in questione adattandone il risultato al Prolog.

3.2 Description Logic Programs

I *DLP* [PH05] nascono per essere l'intersezione tra OWL e la logica del primo ordine. Ogni operatore ammesso in *DLP* può essere tradotto in una clausola di Horn ed è per questo adatto alla conversione in linguaggi come Prolog, Datalog o F-Logic. Inoltre *DLP* è traducibile in altri linguaggi ontologici differenti da *OWL*. I costrutti consentiti in *DLP* sono elencati nella tabella 3.1, ma sotto alcuni vincoli è possibile prenderne in considerazione anche altri, come ad esempio l'unione tra classi.

È doveroso aggiungere che alcune potenzialità di *DLP* dipendono non poco dall'utilizzo che se ne vuole fare. In particolare i *domini concreti* e le *restrizioni numeriche* non vengono considerate parte di *DLP* ma possono essere aggiunte, con qualche astrazione, ad una conversione. La conversione oggetto di questo lavoro di tesi ne è un esempio.

In alcuni casi l'*ugualianza tra istanze* può essere espressa tramite la teoria dell'ugualianza ovvero con la caratterizzazione di un operatore = tramite i seguenti assiomi.

$$\forall x. (x = x)$$

$$\forall x, y, z, w. ((x = y \wedge z = w) \longrightarrow (x = z \wedge y = w))$$

$$\forall x, z. ((x = z) \longleftrightarrow (f(x) = f(z)))$$

$$\forall x. (t = x \longleftrightarrow false)$$

$$f \neq g \quad \forall x, z. (f(x) = g(z) \longleftrightarrow false)$$

Con f e g simboli di funzione e t termine costante diverso da x .

I *vincoli di Integrità*, usati comunemente nella programmazione logica, sono altresì considerati non *DLP*.

Costrutti OWL DLP	Costrutti non OWL DLP
Class	InverseFunctional
Thing	AllDifferent
SubClassOf	someValuesFrom
ObjectProperty	minCardinality
DatatypeProperty	maxCardinality
subPropertyOf	Cardinality
domain	oneOf
range	disjointWith
individual	complementOf
AllValuesFrom	UnionOf
equivalentClass	
equivalentProperty	
sameAs	
hasValue	
differentFrom	
inverseOf	
TransitiveProperty	
SymmetricProperty	
FunctionalProperty	
intersectionOf	

Tabella 3.1: Tabella dei costrutti *OWL* inclusi ed esclusi dal *DLP Fragment*

3.3 Operatore di Conversione

Per meglio definire l'operatore di conversione è opportuno esprimere *OWL* nei formalismi della logica descrittiva come appare nella tabella 3.2. La sintassi utilizzata ci permetterà di definire meglio l'operatore di conversione. Per riassumere adeguatamente i costrutti *OWL* useremo una sintassi astratta.

L'operatore di conversione mostrato nella pubblicazione di Hustad, Motik e Sattler trasforma una base di conoscenza espressa tramite una logica descrittiva *SHIQ⁻* in una logica del primo ordine. L'operatore π è definito in due parti. La prima trasforma i concetti della KB, la seconda gli assiomi. L'operatore π contiene più conversioni di quante ne necessitano in quanto è parte di un algoritmo molto ampio e complesso improntato alla produzione di Datalog disgiuntivo. Di π abbiamo preso solo le parti rientranti nel *DLP Fragment*.

Indichiamo con X una meta-variabile che verrà sostituita nella seconda parte da variabili e con $R.C$ una relazione R avente come dominio la classe C . Per comodità si fa riferimento a π_x per indicare un operatore definito come π_y sostituendo gli x e gli x_i agli y e y_i rispettivamente.

$$\pi_y(\top, X) = \top$$

$$\pi_y(\perp, X) = \perp$$

$$\pi_y(A, X) = A(X)$$

$$\pi_y(\neg C, X) = \neg \pi_y(C, X) = \neg C(X)$$

$$\pi_y(C \sqcap D, X) = \pi_y(C, X) \wedge \pi_y(D, X) = C(X) \wedge D(X)$$

$$\pi_y(C \sqcup D, X) = \pi_y(C, X) \vee \pi_y(D, X) = C(X) \vee D(X)$$

$$\pi_y(\forall R.C, X) = \forall y : R(X, y) \longrightarrow \pi_x(C, y)$$

$$\pi_y(\exists R.C, X) = \exists y : R(X, y) \wedge \pi_x(C, y)$$

OWL	Description Logic Syntax
A (<i>URI</i> reference)	A
<code>owl:Thing</code>	\top
<code>owl:Nothing</code>	\perp
<code>intersectionOf(C_1, C_2, \dots)</code>	$C_1 \sqcap C_2 \sqcap \dots$
<code>unionOf(C_1, C_2, \dots)</code>	$C_1 \sqcup C_2 \sqcup \dots$
<code>complementOf(C)</code>	$\neg C$
<code>oneOf(o_1, o_2, \dots)</code>	$\{o_1, o_2, \dots\}$
<code>restriction(R someValuesFrom(C))</code>	$\exists R.C$
<code>restriction(R allValuesFrom(C))</code>	$\forall R.C$
<code>restriction(R hasValue(o))</code>	$R : o$
<code>restriction(R minCardinality(n))</code>	$\geq n R$
<code>restriction(R maxCardinality(n))</code>	$\leq n R$
<code>EquivalentClasses(C_1, C_2, \dots, C_n)</code>	$C_1 = C_2 = \dots = C_n$
<code>DisjointClasses(C_1, C_2, \dots, C_n)</code>	$C_i \sqcap C_j = \perp \forall i \neq j$
<code>EnumeratedClass(C o_1, o_2, \dots, o_n)</code>	$a = \{o_1, o_2, \dots, o_n\}$
<code>SubClassOf(C_1, C_2)</code>	$C_1 \sqsubseteq C_2$
<code>EquivalentProperty(R_1, R_2, \dots, R_n)</code>	$R_1 = R_2 = \dots = R_n$
<code>SubPropertyOf(R_1, R_2)</code>	$R_1 \sqsubseteq R_2$
<code>InverseOf(R)</code>	R^-
<code>Symmetric(R)</code>	$R = R^-$
<code>Transitive(R)</code>	$Trans(R)$
<code>Functional(R)</code>	$\top \sqsubseteq 1 R$
<code>InverseFunctional(R)</code>	$\top \sqsubseteq 1 R^-$
<code>SameIndividuals(o_1, o_2, \dots, o_n)</code>	$o_1 = o_2 = \dots = o_n$
<code>DifferentIndividuals(o_1, o_2, \dots, o_n)</code>	$o_i \neq o_j \forall i \neq j$

Tabella 3.2: Costrutti *OWL* espressi tramite la sintassi *DL*

$$\pi_y(\leq n \text{ R.C.}, X) = \forall y_1, y_2, \dots, y_{n+1} \bigwedge R(X, y_i) \wedge \bigwedge \pi_x(C, y_i) \longrightarrow \bigvee y_i = y_j$$

$$\pi_y(\geq n \text{ R.C.}, X) = \exists y_1, y_2, \dots, y_n \bigwedge R(X, y_i) \wedge \bigwedge \pi_x(C, y_i) \wedge \bigwedge y_i \neq y_j$$

Per quanto visto nella tabella 3.2, questa prima parte definisce il comportamento da tenersi nel caso si incontri un qualsiasi *URI* (sia esso classe o proprietà), l'intersezione di due classi, la loro unione e le restrizioni `someValuesFrom` e `allValuesFrom`. Non sono di interesse le ultime due voci riguardanti le restrizioni di cardinalità, mentre la terza non rientra nel *DLP Fragment* nell'ambito del complemento di una classe ma è di interesse per l'inverso di una proprietà. Illustreremo ora la seconda parte dell'operatore in merito alla traduzione degli assiomi.

$$\pi(C \sqsubseteq D) = \forall x : \pi_y(C, x) \longrightarrow \pi_y(D, x)$$

$$\pi(C \equiv D) = \forall x : \pi_y(C, x) \longleftrightarrow \pi_y(D, x)$$

$$\pi(R \sqsubseteq S) = \forall x, y : R(x, y) \longrightarrow S(x, y)$$

$$\pi(\text{Trans}(R)) = \forall x, y, z : R(x, y) \wedge R(y, z) \longrightarrow R(x, z)$$

$$\pi(R) = \forall x, y : R(x, y) \longleftrightarrow R^-(y, x)$$

La seconda parte indica come applicare l'operatore alla presenza di sottoclassi, sottoproprietá, operazioni di equivalenza e relazioni transitive. L'ultima ci aiuta nella definizione sia della proprietà simmetrica che dell'inverso di una proprietà. Come detto in precedenza, l'operatore di conversione viene applicato a logiche descrittive *SHIQ*. *OWL DLP* possiede in più rispetto alle *SHIQ* i tipi di dato, la possibilità di definire relazioni funzionali e l'uguaglianza tra individui. Per i tipi di dato abbiamo reso il passaggio dai tipi *XSD* di *OWL* il più possibile consono al Prolog, mentre per l'uguaglianza e le relazioni funzionali abbiamo preso spunto da *Dlpconvert* [Dlp06]. Considereremo questi casi con maggiore dettaglio nell'esempio di conversione.

3.4 Un Esempio di Conversione

Per dare maggior chiarezza alla conversione definita in precedenza è utile fornire un esempio di conversione da *OWL* a Prolog piuttosto che una tabella di conversione la quale risulterebbe probabilmente poco chiara.

L'ontologia che segue cerca di esprimere, in modo molto semplice ed assolutamente migliorabile, il dominio dei vini.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:j.0="http://protege.stanford.edu/plugins/owl/protege#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">

  <owl:Ontology rdf:about="" />

  <owl:Class rdf:ID="Color" />
  <owl:Class rdf:ID="Wine" />
  <owl:Class rdf:ID="Region" />
  <owl:Class rdf:ID="WineBody" />

  <owl:Class rdf:ID="WineColor">
    <rdfs:subClassOf rdf:resource="#Color" />
  </owl:Class>

  <owl:Class rdf:ID="Europe">
    <rdfs:subClassOf rdf:resource="#Region" />
  </owl:Class>

  <owl:Class rdf:ID="America">
```

```

        <rdfs:subClassOf rdf:resource="#Region" />
</owl:Class>

```

In queste prime definizioni *TBox* indichiamo alcune classi utili a rappresentare il dominio. Le classi prive di ulteriori specifiche quali `Color`, `Wine`, `Region` e `WineBody`, nella traduzione in Prolog, vengono ignorate in quanto non recano informazioni utili.

Le classi `Europe` e `America` sono sottoclassi di `Region` e vengono convertite nelle seguenti clausole.

```
region(X):- europe(X).
```

```
region(X):- america(X).
```

Consideriamo ora la classe `WhiteWine` che intendiamo essere una sottoclasse di `Wine` con una restrizione sulla proprietà `hasColor`.

```

<owl:Class rdf:ID="WhiteWine">
  <rdfs:subClassOf rdf:resource="#Wine"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor"/>
      <owl:hasValue rdf:resource="#White"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

Alla precedente classe corrisponderà:

```

whitewine(X):- hascolor(X,white).
wine(X):- whitewine(X).

```

Continuando sulle restrizioni, definiamo due classi rappresentanti i vini Europei e quelli Americani come casi particolari della classe vino localizzata nelle rispettive aree.

```

<owl:Class rdf:ID="EuropeanWine">
  <rdfs:subClassOf rdf:resource="#Wine"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#locatedIn"/>
      <owl:allValuesFrom rdf:resource="#Europe"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="AmericanWine">
  <rdfs:subClassOf rdf:resource="#Wine"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#locatedIn"/>
      <owl:allValuesFrom rdf:resource="#America"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

Il loro corrispettivo è

```

europe(Y) :- europeanwine(X), locatedin(X,Y).
wine(X) :- europeanwine(X).

```

```

america(Y) :- americanwine(X), locatedin(X,Y).
wine(X) :- americanwine(X).

```

Per continuare la rassegna di tutti i costrutti *OWL DLP* utilizzabili prendiamo in considerazione l'unione e l'intersezione definendo una classe dei vini bianchi Europei e dei vini Europei e Americani.

```

<owl:Class rdf:ID="WhiteEuropeanWine">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#EuropeanWine" />
    <owl:Class rdf:about="#WhiteWine" />
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="EuropeanAndAmericanWine">
  <owl:unionOf rdf:parseType="Collection">

```

```

                <owl:Class rdf:about="#AmericanWine" />
                <owl:Class rdf:about="#EuropeanWine" />
        </owl:unionOf>
</owl:Class>

```

In Prolog queste due classi sono

```

whiteeuropeanwine(X):- europeanwine(X),whitewine(X).
europeandamericawine(X):- americanwine(X);europeanwine(X).

```

Passiamo ora dalle classi alle proprietà

```

<owl:ObjectProperty rdf:ID="locatedIn">
    <rdfs:domain rdf:resource="#Wine"/>
    <rdfs:range rdf:resource="#Region"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasBody">
    <rdfs:domain rdf:resource="#Wine"/>
    <rdfs:range rdf:resource="#WineBody"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="AssociatedColor">
    <rdfs:domain
        rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
    <rdfs:range rdf:resource="#Color" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasColor">
    <rdfs:subPropertyOf rdf:resource="#AssociatedColor" />
    <rdfs:domain rdf:resource="#Wine"/>
    <rdfs:range rdf:resource="#WineColor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="relatedWine">
    <rdf:type rdf:resource="&owl;SymmetricProperty" />
    <rdfs:domain rdf:resource="#Wine" />
    <rdfs:range rdf:resource="#Wine" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="functionalExample">
    <rdf:type rdf:resource="&owl;FunctionalProperty" />

```

```

        <rdfs:domain rdf:resource="#Wine" />
        <rdfs:range rdf:resource="#Wine" />
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="yearValue">
    <rdfs:domain rdf:resource="#Wine" />
    <rdfs:range rdf:resource="#xsd:positiveInteger" />
</owl:DatatypeProperty>

```

Le prime tre vengono convertite nel seguente modo.

```

wine(X):- locatedin(X,_Y).
region(Y):- locatedin(_X,Y).

```

```

wine(X):- hasbody(X,_Y).
winebody(Y):- hasbody(_X,Y).

```

```

color(Y):- associatedcolor(_X,Y).

```

Da notare come in `associatedcolor` il dominio non venga tradotto in una corrispondente clausola in quanto fa riferimento alla classe generica `owl:Thing`. Le rimanenti quattro sono rispettivamente esempi di sottoproprietà, simmetria e relazione funzionale, mentre l'ultima è una `Datatype`.

```

wine(X):- hascolor(X,_Y).
winecolor(Y):- hascolor(_X,Y).
associatedcolor(X,Y):- hascolor(X,Y).

```

```

wine(X):- relatedwine(X,_Y).
wine(Y):- relatedwine(_X,Y).
relatedwine(X,Y):- relatedwine(Y,X).

```

```

wine(X):- functionalexample(X,_Y).
wine(Y):- functionalexample(_X,Y).
dlp_equal(Y,Y2):- functionalexample(X,Y),functionalexample(X,Y2).

```

```

wine(X):- yearvalue(X,_Y).
yearvalue(X,Y):- wine(X),Y>0.

```

Si noti come nella terza venga espressa l'uguaglianza tra i due termini `Y` e `Y2` tramite il predicato `dlp_equal`. Questa soluzione forza ad usare `dlp_equal` per

la verifica di una proprietà funzionale. L'idea dell'utilizzo di un predicato extra per adempiere a questo scopo viene da *Dlpconvert*.

Nella *Datatype* viene espresso un vincolo numerico. I tipi di dato consentiti sono `positiveInteger`, che viene tradotto con $Y > 0$, `nonNegativeInteger`, `unsignedLong`, `unsignedInt`, `unsignedShort`, `gMonth`, `gDay`, `gYear` e `time` che vengono considerati come vincoli $Y \geq 0$. In Prolog non è possibile verificare realmente un dominio come quello degli *short* o dei *long* in modo indipendente dall'interprete sottostante, il che motiva questa soluzione. L'utilizzo di altri tipi di dati oltre a quelli elencati causa la non considerazione del vincolo.

Concluso con le *TBox*, consideriamo infine le *ABox*.

```
<Europe rdf:ID="Italy"/>
<Europe rdf:ID="France"/>

<WineBody rdf:ID="Full" />
<WineBody rdf:ID="Medium" />
<WineBody rdf:ID="Light" />

<WineColor rdf:ID="Red" />
<WineColor rdf:ID="Rose" />
<WineColor rdf:ID="White" />

<EuropeanWine rdf:ID="Barbera">
  <hasColor rdf:resource="#Red"/>
  <locatedIn rdf:resource="#Italy"/>
  <hasBody rdf:resource="#Full"/>
  <yearValue rdf:datatype="&xsd;positiveInteger">1998
  </yearValue>
</EuropeanWine >

<EuropeanWine rdf:ID="Chianti">
  <hasColor rdf:resource="#Red"/>
  <locatedIn rdf:resource="#Italy"/>
  <hasBody rdf:resource="#Full"/>
  <yearValue rdf:datatype="&xsd;positiveInteger">1998
  </yearValue>
  <relatedWine rdf:resource="#Barbera"/>
</EuropeanWine>
```

```
<EuropeanWine rdf:ID="Barbera_DOC">
    <owl:sameAs rdf:resource="#Barbera" />
</EuropeanWine>
```

La conversione degli individui è semplice ed intuitiva, resta da notare come con Barbera_DOC venga utilizzato il predicato `dlp_equal` visto in precedenza in corrispondenza della restrizione `sameAs`.

```
europe(france).
europe(italy).
```

```
winebody(full).
winebody(medium).
winebody(light).
```

```
winecolor(rose).
winecolor(red).
winecolor(white).
```

```
europeanwine(chianti).
relatedwine(chianti,barbera).
yearvalue(chianti,1998).
hasbody(chianti,full).
locatedin(chianti,italy).
hascolor(chianti,red).
```

```
europeanwine(barbera).
yearvalue(barbera,1998).
hasbody(barbera,full).
locatedin(barbera,italy).
hascolor(barbera,red).
```

```
europeanwine(barbera_doc).
dlp_equal(barbera,barbera_doc).
```

3.5 Safe Fragment

Datalog è una restrizione di linguaggio del Prolog, studiato appositamente per le basi di dati. Le principali differenze con il Prolog sono l'assenza dei simboli di funzione e un modello di valutazione non procedurale, ovvero non tramite la risoluzione SDL del Prolog. Il metodo risolutivo del Datalog impone la ricerca di un punto fisso, questo offre la possibilità di non entrare in loop come può succedere con interrogazioni Prolog.

Per rendere più utile la conversione abbiamo cercato un insieme di costrutti *OWL* che rendesse finita ogni interrogazione Prolog su di essi. Abbiamo indicato questo insieme come il *Safe Fragment*, ed ora vedremo quali costrutti devono venirne esclusi e perchè.

- **SymmetricProperty**: Una relazione simmetrica

```
<owl:SymmetricProperty rdf:ID="Amico_di">
    <rdfs:domain rdf:resource="#Persona"/>
    <rdfs:range rdf:resource="#Persona"/>
</owl:SymmetricProperty>
```

viene tradotta in:

```
Persona(X):- Amico_di(X,_Y).
Persona(Y):- Amico_di(_X,Y).
Amico_di(X,Y):- Amico_di(Y,X).
```

L'ultima delle tre clausole causa un evidente problema di loop.

- **equivalentClass**: Se due classi sono equivalenti in Prolog è necessario esprimerlo come:

`Persona(X) :- Umano(X).`

`Umano(X) :- Persona(X).`

È evidente come anche in questo caso la richiesta di individuare se una persona è tale risulta essere un ciclo infinito.

- `equivalentProperty`: Analogamente alle classi, anche per le proprietà il rapporto di equivalenza causa qualche problema:

`Padre_di(X,Y) :- Padre(Y,X).`

`Padre(X,Y) :- Padre_di(Y,X).`

- `InverseOf`: Similmente a quanto visto per le proprietà equivalenti, anche l'inverso può essere causa di loop:

`Padre_di(X,Y) :- Figlio_di(Y,X).`

`Figlio_di(X,Y) :- Padre_di(Y,X).`

- `DatatypeProperties`: I dati numerici, possono essere di questo tipo:

```
<owl:DatatypeProperty rdf:ID="Anno_di_nascita">
  <rdfs:domain rdf:resource="#Persona"/>
  <rdfs:range rdf:resource="&xsd;positiveInteger"/>
</owl:DatatypeProperty>
```

Che viene tradotto con:

`Anno_di_nascita(X,Y) :- Persona(X), Y>0.`

Ponendo una clausola obbiettivo in cui si cerca di determinare l'anno di nascita di una persona, questo causerebbe dei problemi di unificazione in quanto l'albero SDL troverà, se c'è, l'anno di una persona ma cercherà comunque di unificare al passo successivo causando un errore di istanziazione

degli argomenti.

L'errata istanziazione di valori numerici può essere gestita tramite particolari vincoli sui tipi di dato noti. Ad esempio in SWI Prolog è possibile usare i vincoli `clp(FD)` che corrispondono agli operatori `#>`, `#<`, `#>=`, ...

Capitolo 4

Strumento di Conversione

4.1 OwlDlp2Pl

Il tool risultato di questo lavoro di tesi è stato scritto in Java, utilizzando la libreria Jena [Jen04]. Lo scopo che si prefigge è quello di convertire un'ontologia *OWL*, contenente i soli costrutti elencati nel frammento DLP, in un sorgente Prolog. OwlDlp2Pl è stato pensato come un tool da linea di comando, ma è facilmente modificabile ed inseribile in un'interfaccia utente più complessa.

L'utilizzo da riga di comando è semplice.

```
java -jar OwlDLP2Pl.jar ontologia.owl
```

Il sorgente Prolog generato avrà lo stesso nome del file *OWL* in input con l'estensione *.pl*. Nel caso in cui l'ontologia contenga errori di sintassi o di consistenza, questi verranno emessi in output e non verrà generato il file Prolog.

Per attuare una conversione corretta dell'ontologia, OwlDlp2Pl verifica che gli operatori contenuti nell'ontologia siano solo quelli del *DLP Fragment*. Per incrementare l'utilità del tool abbiamo aggiunto due opzioni.

```
User$ java -jar OwlDLP2Pl.jar
```

Usage

```
java -jar OwlDlp2Pl.jar <Ontology_File> <options>
```

```
-U Extended URI
```

```
-S Safe Fragment Only
```

Utilizzando l'opzione `-S` è possibile verificare che l'ontologia contenga solo elementi appartenenti al *Safe Fragment*, `-U` abilita il nome esteso di ogni elemento dell'ontologia. In *OWL* ogni elemento (classi, individui o proprietà) è individuato dall'*URI* dell'ontologia a cui appartiene, seguito da `#` e il suo nome. L'opzione è particolarmente utile nel caso in cui l'ontologia *OWL* che si vuole tradurre faccia riferimento ad elementi di ontologie esterne con nomi uguali a quelli utilizzati in quella corrente. In output gli *URI* vengono modificati per evitare che alcuni caratteri collidano con la sintassi Prolog.

4.2 Class Diagram

La struttura del tool è semplice. Il lavoro principale è svolto dalle classi `Converter` e `SyntaxChecker`. La prima svolge la conversione sfruttando la classe `Clause` per la rappresentazione del codice. La seconda controlla il *DLP Fragment* e il *Safe Fragment* e svolge anche i test sulla consistenza dell'ontologia. Il controllo sintattico del documento *OWL* è fatto nel `main`.

4.3 Ampliamenti del Tool

OwlDlp2Pl è predisposto per fornire in futuro anche una conversione da *OWL* a *Datalog*. Nella classe `Converter`, è possibile invocare `GetDataLogStructure`, per ottenere una lista di oggetti `Clause`. Come è possibile vedere dal class diagram ogni clausola è composta da una `Head` e da un `Body`. La testa è composta da

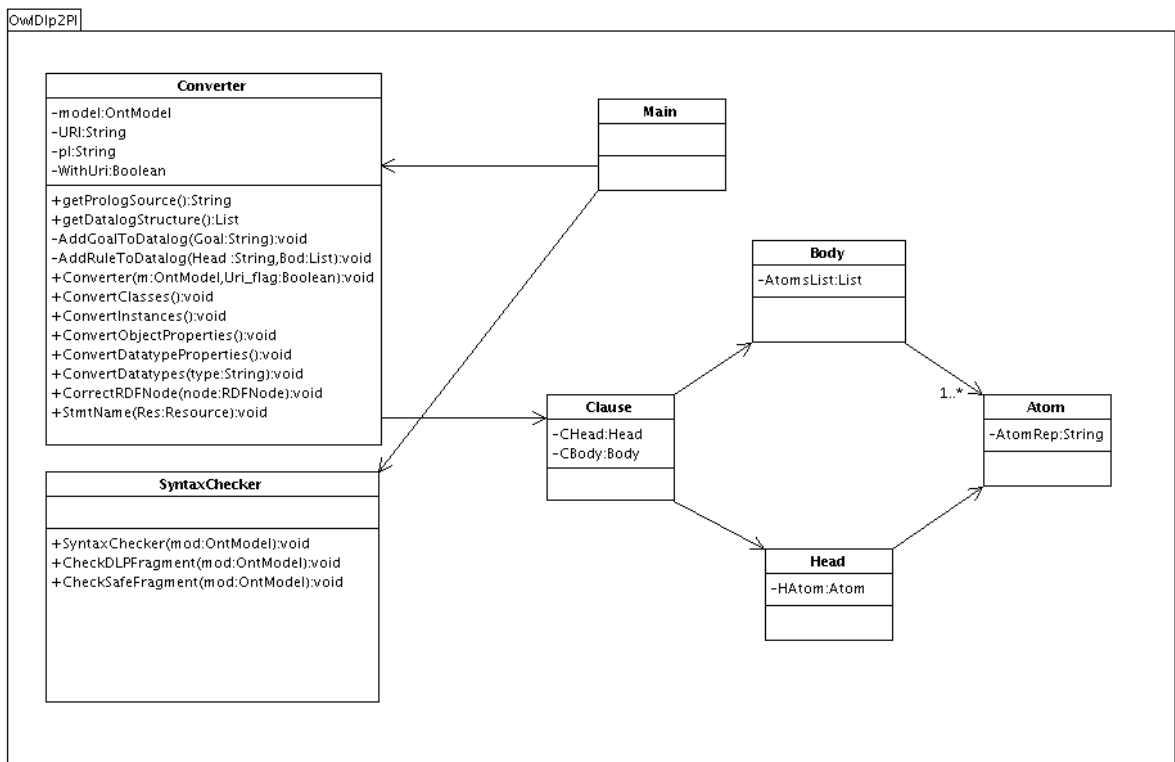


Figura 4.1: Class Diagram di OwDlp2PI

un solo atomo mentre il corpo è una lista di oggetti **Atom**. La lista potrà essere rielaborata per consentire la conversione in Datalog.

Il tool potrà essere convertito in un ragionatore *OWL* integrandogli un interprete Prolog o Datalog.

4.4 Strumenti Utilizzati

Per lo sviluppo del progetto abbiamo utilizzato la libreria Jena. Jena è un framework open source, sviluppato dal HP Semantic Lab, utile per costruire applicazioni legate al *Semantic Web*. Jena permette di gestire documenti *RDF*, *RDFS*, *OWL* e i linguaggi di query *SPARQL* ed *RDQL*. Nell'ambito del progetto abbiamo utilizzato Jena principalmente per navigare un'ontologia *OWL*, ovvero scorrerne classi, istanze e proprietà, previa verifica della sintassi.

Un ulteriore utilizzo è stato quello di verificare la consistenza del documento in modo da convertire solo le ontologie in cui nessun processo d'*inferenza* può portare ad un assurdo.

Per la scrittura del sorgente Java è stato utilizzato Netbeans 5.0 per Mac OsX 10.4.6. Per la verifica delle conversioni abbiamo usato principalmente SWI Prolog 5.6.12. È stato utilizzato Protégé [Pro] 3.1 per il design delle ontologie *OWL*.

Capitolo 5

Conclusioni

Il presente lavoro di tesi è iniziato con lo studio e l'analisi della conversione da *OWL DLP* a Prolog dal punto di vista teorico, sfruttando le pubblicazioni e gli strumenti software disponibili. Un secondo passo è stata la definizione del *Safe Fragment* (insieme dei costrutti *OWL* che rendono finita ogni interrogazione con un interprete Prolog) tramite ragionamenti teorici e prove pratiche. In ultimo si è proceduto con lo sviluppo del tool di conversione.

Lo strumento di conversione *OwlDlp2Pl* è utile sia per scrivere che per testare ontologie *OWL*, nonostante la sua utilità sia limitata dalla mancanza di alcuni costrutti. Presa visione di quali parti di *OWL* devono essere scartate e di quali hanno effetti collaterali in Prolog, il tool può essere utilizzato per il debug di ontologie.

Tra le prospettive di ampliamento che vengono reputate più interessanti c'è quella di integrare ad *OwlDlp2Pl* un interprete Prolog in modo tale da poterlo utilizzare come un ragionatore *OWL*. Un ampliamento ulteriore è l'aggiunta della conversione in Datalog per ottenere un ragionatore con maggiori possibilità espressive rispetto ad uno basato su Prolog.

Bibliografia

- [DIG] *DIG interface*, Available at <http://dig.sourceforge.net/>.
- [Dlp06] *Dlpconvert*, june 2006, Available at <http://logic.aifb.uni-karlsruhe.de/dlpconvert/>.
- [Fac06] *FaCT++ OWL DL Reasoner*, April 2006, Available at <http://owl.man.ac.uk/factplusplus/>.
- [Gru] Tom Gruber, *What is an ontology?*, Available at <http://www.ksl.stanford.edu/kst/what-is-an-ontology.html>.
- [Hor06] Ian Horrocks, *Hybrid logics and ontology languages*, August 2006, Available at <http://www.cs.man.ac.uk/~horrocks/Slides/index.html>.
- [IH05] Peter F. Patel-Schneider e Frank van Harmelen Ian Horrocks, *From SHIQ and RDF to OWL: The making of a web ontology language*, February 2005, Available at <http://www.cs.vu.nl/~frankh/postscript/JWS03.pdf>.
- [Jen04] *Jena: A Semantic Web framework for java*, May 2004, Available at <http://jena.sourceforge.net/>.
- [Kao05] *Kaon2: Ontology management for the semantic web*, December 2005, Available at <http://kaon2.semanticweb.org/>.

- [Pel06] *Pellet OWL Reasoner*, April 2006, Available at <http://www.mindswap.org/2003/pellet/>.
- [PH05] Rudi Studer e York Sure Pascal Hitzler, *Description logic programs: A practical choice for the modelling of ontologies*, 2005, Available at <http://logic.aifb.uni-karlsruhe.de/download/dlppos05.pdf>.
- [Pro] *The Protégé: Ontology Editor and Knowledge Acquisition System*, Available at <http://protege.stanford.edu/>.
- [Rac06] *RacerPro*, 2006, Available at <https://www.racer-systems.com/>.
- [UH04] Boris Motik e Ulrike Sattler Ullrich Hustadt, *Reducing SHIQ- Description Logic to Disjunctive Datalog Programs*, 2004, Available at <http://www.csc.liv.ac.uk/~ullrich/publications/HMS-KR2004.pdf>.
- [W3C04] *W3C OWL guide*, February 2004, Available at <http://www.w3.org/TR/2004/REC-owl-guide-20040210/>.