Sviluppo e valutazione di un reasoner concorrente per una variante ridotta della logica $\mathcal{EL}^{++}$: ELPPReasoner

Danilo Esposito (mat. N97000376)
Raffaele Di Maso (mat. N97000411)

a.a. 2023-2024

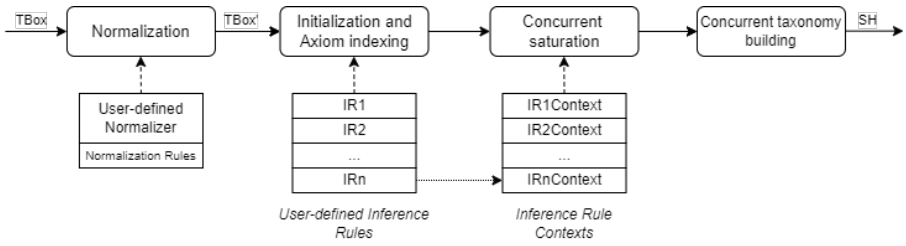
Indice

1 Introduzione

Il documento in questione propone l'implementazione di un ragionatore per una versione ridotta della logica $\mathcal{EL}^{++}$, chiamato *ELPPReasoner*. Nello specifico, si analizzerà la proprietà concorrenziale del ragionatore, impiegando una combinazione di ontologie proprie e alcune tra le più note. I risultati ottenuti verranno poi valutati e confrontati con dei ragionatori disponibili sotto licenza pubblica.

L'obiettivo preposto è la realizzazione di un ragionatore ottimizzato per il profilo OWL2-EL, la cui specificità dovrebbe portare a significativi miglioramenti delle prestazioni rispetto ai ragionatori cosiddetti *general-purpose*. L'idea implementativa, in particolare dell'aspetto concorrenziale del ragionatore, si basa sulle linee guida proposte da [1], in cui l'isolamento degli assiomi in *contesti* permette a più thread di processarli in maniera indipendente. Tale concetto è alla base dell'implementazione di ELK [2], ad opera dello stesso autore.

Un aspetto altrettanto importante è la libreria su cui si basa il ragionatore proposto. Come vedremo, il sistema in analisi offre un certo grado di personalizzazione del processo di reasoning, fornendo agli sviluppatori un ambiente controllato in cui definire e integrare le proprie regole di inferenza. Difatti, per il collaudo di tale modularità si è deciso di implementare un sottoinsieme delle *completion rules* proposte da [3], ossia le regole da `CR1` a `CR6`.



Il ragionatore[1] è un'implementazione della classe `OWLReasoner` di OWL API 5 in Java. Esso include un accesso controllato e indicizzato alla TBox in input tramite l'`OntologyAccessManager` e ai contesti definiti su tale TBox tramite il `ContextAccessManager`. Queste strutture dati vengono inizializzate sulla base delle già citate *user-defined inference rules*, che raggruppano gli assiomi della TBox secondo criteri operativi definiti. L'applicazione delle regole di normalizzazione alla TBox produce una versione normalizzata della stessa. Successivamente, il saturatore applica in modo esaustivo le regole di inferenza sugli assiomi della TBox normalizzata, sfruttando la concorrenza. Infine, a partire dalle conclusioni fornite dal saturatore, il costruttore della tassonomia genera la *gerarchia delle sussunzioni*, anch'esso sfruttando la concorrenza.

[1] Disponibile su GitHub all'indirizzo `https://github.com/ralphthegod/elpp_reasoner`.

2 Normalizzazione

La normalizzazione è una fase delicata che precede l'algoritmo che esegue la sussunzione strutturale di concetti atomici. Come si può osservare dalla figura nel capitolo di Introduzione, che mostra ad alto livello l'architettura del ragionatore proposto, un "normalizzatore", a scatola chiusa, esegue un insieme di operazioni su una TBox $\mathcal{T}$ che permettono di ottenere la stessa TBox in forma normale $\mathcal{T}'$. Avere una TBox in forma normale consente di avere un algoritmo per la sussunzione in $\mathcal{EL}^{++}$ che opera in tempo polinomiale.

2.1 Base teorica per la normalizzazione

Fino a questo punto, abbiamo più volte utilizzato il termine "TBox", con cui si intende formalmente un insieme di *general concept inclusions* (GCIs):

Definizione 2.1 (GCI) *Una general concept inclusion (GCI) è un assioma avente la seguente forma:*

$$C \sqsubseteq D$$

dove C è un concetto atomico o un nominale, e D è un concetto atomico, un nominale o un bottom ($\bot$).

Definizione 2.2 (TBox) *Una terminological box (TBox) per la logica descrittiva $\mathcal{EL}^{++}$ è un insieme di general concept inclusions (GCIs).*

Diversamente da quanto viene trattato in [3] con le TBox, infatti, il nostro lavoro si è concentrato su una versione ridotta della logica $\mathcal{EL}^{++}$ che non prevede *domini concreti* e *inclusioni di ruoli*. Per questo motivo, è corretto parlare semplicemente di TBox.

Fatta questa doverosa premessa, è ora possibile entrare nel merito della normalizzazione:

Definizione 2.3 (Forma Normale per le TBox) *Una terminological box (TBox) per la logica descrittiva $\mathcal{EL}^{++}$ è in forma normale (NF, Normal Form) se ogni GCI è formata in uno dei seguenti modi:*

$$\begin{aligned} C_1 &\sqsubseteq D \\ C_1 \sqcap C_2 &\sqsubseteq D \\ C_1 &\sqsubseteq \exists r.C_2 \\ \exists r.C_1 &\sqsubseteq D \end{aligned}$$

dove C_1 e C_2 sono concetti atomici o nominali, e D è un concetto atomico, un nominale o un bottom ($\bot$).
Una TBox in forma normale è anche detta "normalizzata".

Il prossimo paragrafo mostra un esempio di normalizzatore, `ELPPOntologyNormalizer`, sviluppato secondo i dettami teorici di [4]. In questo report viene spiegato come qualsiasi TBox $\mathcal{T}$ può essere trasformata in una TBox normalizzata $\mathcal{T}'$ - che ne rappresenta una *estensione conservativa* equivalente - introducendo nuovi concetti generici; questa trasformazione richiede tempo e spazio lineari nell'input.

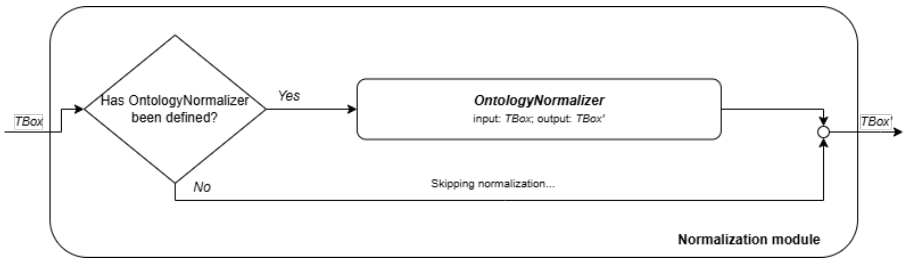
La teoria su cui si basa `ELPPOntologyNormalizer` prevede sei regole di normalizzazione (dove la GCI a sinistra della freccia viene sostituita con quanto presente a destra della freccia), a seconda della forma in cui una GCI è posta:

NF2	$C \sqcap \hat{D} \sqsubseteq E$	$\longrightarrow$	$\{\hat{D} \sqsubseteq A, C \sqcap A \sqsubseteq E\}$
NF3	$\exists r.\hat{C} \sqsubseteq D$	$\longrightarrow$	$\{\hat{C} \sqsubseteq A, \exists r.A \sqsubseteq D\}$
NF4	$\bot \sqsubseteq D$	$\longrightarrow$	$\emptyset$
NF5	$\hat{C} \sqsubseteq \hat{D}$	$\longrightarrow$	$\{\hat{C} \sqsubseteq A, A \sqsubseteq \hat{D}\}$
NF6	$B \sqsubseteq \exists r.\hat{C}$	$\longrightarrow$	$\{B \sqsubseteq \exists r.A, A \sqsubseteq \hat{C}\}$
NF7	$B \sqsubseteq C \sqcap D$	$\longrightarrow$	$\{B \sqsubseteq C, B \sqsubseteq D\}$

dove $\hat{C}$ e $\hat{D}$ non sono concetti atomici, e A è un nuovo concetto atomico generico.
Nota. **NF1** tratta le role inclusions, non considerate nel nostro lavoro, per cui verrà ignorata.

La normalizzazione è un'operazione semplice: per ogni GCI nella TBox considerata, se non è in forma normale applica la regola NF{2/.../7} che coincide con la sua forma attuale. In base alla singola GCI considerata, le GCI restituite post-normalizzazione potrebbero ancora non essere in forma normale: in questo caso è necessaria una nuova trasformazione.

2.2 Architettura del modulo `normalization`



La figura precedente riassume la filosofia del ragionatore proposto: l'utente può scegliere liberamente il normalizzatore da utilizzare, purché implementi l'interfaccia `OntologyNormalizer` (package `reasoner/normalization`).

Quest'ultima richiede la realizzazione di due metodi:

- `OWLOntology normalize(OWLOntology ontology);`
  Data una ontologia, restituire la stessa ontologia normalizzata.
- `Set<OWLAxiom> normalize(Set<OWLAxiom> axioms);`
  Data una TBox, restituire la stessa TBox normalizzata.

L'impostazione di un normalizzatore è opzionale: se non ne è stato impostato uno, la normalizzazione della TBox viene saltata, passando direttamente alla fase di indicizzazione. Saltare la normalizzazione è utile quando l'ontologia su cui effettuare reasoning è nota essere già normalizzata, consentendo di risparmiare importanti risorse di tempo e spazio.

2.3 Un esempio: `ELPPOntologyNormalizer`

Il ragionatore sviluppato in questo lavoro è dotato di un semplice normalizzatore di ontologie per la logica descrittiva $\mathcal{EL}^{++}$: `ELPPOntologyNormalizer`, che implementa l'interfaccia `OntologyNormalizer`. Nella seguente figura è posta l'intera logica del suddetto normalizzatore, il quale, data una TBox $\mathcal{T}$ da normalizzare (i.e., un insieme di GCI non in forma normale):

1. Crea una TBox vuota $\mathcal{T}'$
2. Ottiene la lista di assiomi da normalizzare da $\mathcal{T}$, `axiomsToNormalize`
3. Itera su `axiomsToNormalize`. Sia `nAxiom` l'assioma controllato in questa iterazione:
   (a) Se `nAxiom` è in forma normale, aggiungilo alla TBox $\mathcal{T}'$ e passa al prossimo assioma
   (b) Se `nAxiom` non è in forma normale:
      i. Identifica la sua **NF**
      ii. Applica la regola di normalizzazione corrispondente, ottenendo un piccolo insieme di (0 o 2) assiomi, `normalizedAxioms`
      iii. Aggiungi gli elementi di `normalizedAxioms` ad `axiomsToNormalize` per un controllo aggiuntivo [2]

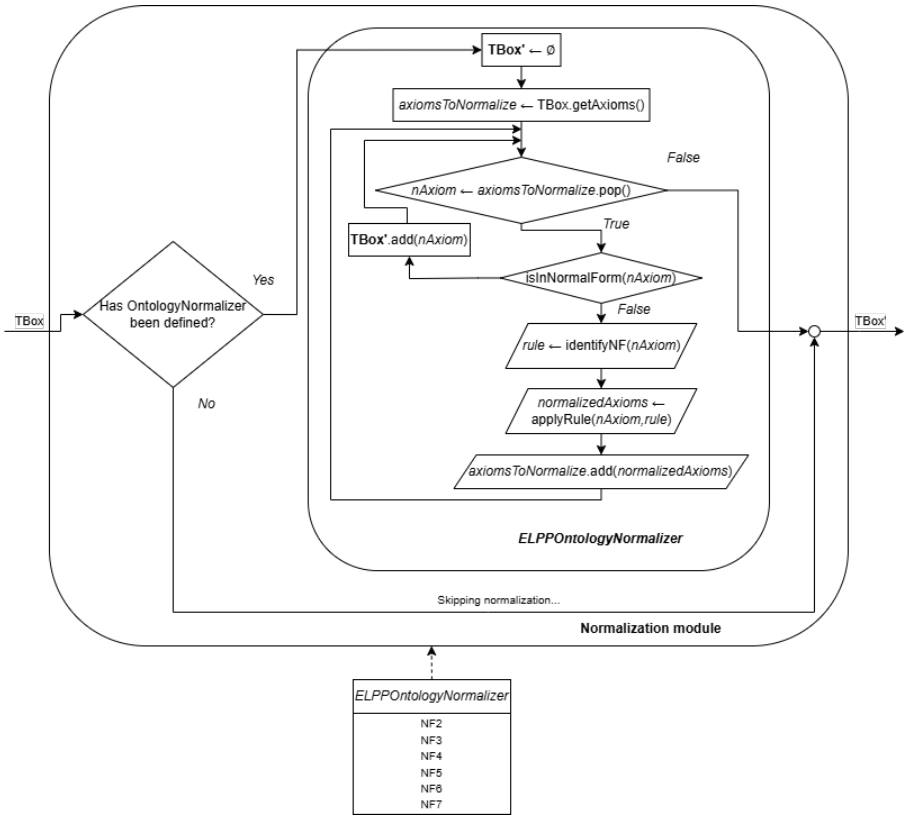
[2] Questo passaggio, seppur equivalente in termini teorici, differisce dall'implementazione: è stato inserito in questa modalità soltanto per snellire lo schema di `ELPPOntologyNormalizer`. In realtà, i `normalizedAxioms` non vengono aggiunti ad `axiomsToNormalize` per un controllo aggiuntivo, bensì viene eseguito quanto segue:

A. Itera su `normalizedAxioms`. Sia `a` l'assioma controllato in questa iterazione:

B. Se `a` è in forma normale, aggiungilo alla TBox $\mathcal{T}'$ e passa al prossimo assioma

C. Se `a` non è in forma normale, aggiungilo ad `axiomsToNormalize` per un ulteriore passo di normalizzazione

Quanto descritto è schematizzato nella seguente figura.



Oltre a `ELPPOntologyNormalizer` è anche fornita una classe di utilità, `NormalizationUtilities`, che include diversi metodi:

- `boolean isConceptName(OWLClassExpression classExpression)`
  Verifica se l'espressione fornita è una classe OWL.
- `boolean isIndividualName(OWLClassExpression classExpression))`
  Verifica se l'espressione fornita è un individuo OWL.
- `boolean isSubclassABasicConcept(OWLClassExpression subClassConcept)))`
  Verifica se l'espressione fornita (sottoclasse di GCI) è un concetto atomico.
- `boolean isSuperclassABasicConcept(OWLClassExpression subClassConcept)))`
  Verifica se l'espressione fornita (superclasse di GCI) è un concetto atomico.
- `boolean isGCIInNormalForm(OWLSubClassOfAxiom subClassOfAxiom)))`
  Verifica se la GCI fornita è in forma normale.

`NormalizationUtilities` include anche una classe interna, `NormalizationRulesManager`, che contiene metodi e oggetti per semplificare l'individuazione e l'applicazione di regole di normalizzazione:

- `enum NormalizationRule {NF2,NF3,NF4,NF5,NF6,NF7}`
  Si rimanda alla fine del Paragrafo 2.1.
- `NormalizationRule identifyNF(OWLSubClassOfAxiom axiom)`
  Identifica la regola di normalizzazione corrispondente all'assioma fornito.
- `OWLClass generateOWLClass(OWLOntology ontology, OWLClassExpression subclass, OWLClassExpression superclass)`
  Genera un nuovo concetto atomico generico, corrispondente ad A nelle regole di normalizzazione.
- `Collection<OWLSubClassOfAxiom> applyNF*X*(OWLOntology ontology, OWLSubClassOfAxiom axiom)`
  Applica la regola di normalizzazione `NF*X*`, con `X = 2,...,7`.

Nota. Il normalizzatore `ELPPOntologyNormalizer` opera su un singolo thread, ma è possibile realizzarne uno che lavora in parallelo per migliorare l'efficienza complessiva.

3 Accesso all'ontologia

Per evitare ricerche lineari nell'ottenimento delle premesse, l'indicizzazione degli assiomi è fondamentale. Il ragionatore dispone di un modulo per l'accesso controllato all'ontologia, eliminando, in fase di saturazione, la necessità di dover iterare sull'elenco degli assiomi per raccogliere quelli richiesti. Il raggruppamento avviene in automatico attraverso le *user-defined inference rules*, che mantengono delle strutture dati di tipo multi-map chiave-valore [2]. In questo modo ci si assicura che, in fase di saturazione, l'algoritmo è confinato alle responsabilità della regola di inferenza corrente, dunque accedendo rapidamente ai soli assiomi rilevanti per quel caso d'uso.

Responsabile dell'accesso controllato all'ontologia è la classe `OntologyAccessManager`, contenuta nel package `reasoner/querying`. Tale classe mantiene l'ontologia (`OWLOntology ontology`) e le regole di inferenza *user-defined* (`Map< Class<? extends InferenceRule>, InferenceRule> rules`); inoltre, il metodo `getAxiomsByRule` fornisce gli assiomi indicizzati secondo una specifica regola d'inferenza.

3.1 Regole di inferenza *user-defined*

L'inizializzazione del ragionatore richiede la definizione delle regole di inferenza. Nello specifico, l'implementazione di ogni regola di inferenza deve includere:

- Un struttura dati per l'indice degli assiomi;
- Un criterio per gli assiomi da indicizzare;
- I tipi di entità trattabili (es. classi o nominali).

Come vedremo in seguito, sarà necessario anche definire il tipo di *contesto*, che include gli aspetti relativi alla computazione delle premesse, e l'algoritmo per l'estrazione dei contesti dato un assioma. L'idea è quella di garantire l'*isolamento* ai thread che opereranno su tali contesti.

All'interno del package `reasoner`, la classe astratta `InferenceRule` mantiene l'indice nella struttura dati `axioms`. Le estensioni di tale classe devono implementare i metodi `axiomCriterion` e `extractContexts`.

3.2 Implementazione delle regole

La logica usata dal reasoner rappresenta una versione ridotta della logica polinomiale $\mathcal{EL}^{++}$, ed include: il top e bottom concept, l'intersezione, la restrizione esistenziale e i nominali. Le regole di inferenza implementate, come già anticipato, sono le prime sei *completion rules* di [3].

3.2.1 CR1: `ToldSuperclasses`

Sia $\mathcal{C}$ la base di conoscenza, la regola di inferenza CR1 è così definita:

$$\mathsf{CR1} \quad \frac{C \sqsubseteq C'}{C \sqsubseteq D} : C' \sqsubseteq D \in \mathcal{C}$$

Se C' è una superclasse di C, D è una superclasse di C' e D non è (ancora) una superclasse di C, allora D va aggiunta alle superclassi di C.
Il criterio per l'indicizzazione di tale regola è alquanto banale: si imposta come chiave ogni concetto C, associando ad esso tutte le superclassi individuate negli assiomi in input ($\in \mathcal{C}$). La struttura dati risultante è della seguente forma:

$$S_{\mathsf{CR1}}(C) = \{D \mid C \sqsubseteq D \in \mathcal{C}\}$$

dove S_{CR1} è la funzione che rappresenta l'indice mappa su CR1.

Durante l'iterazione iniziale sugli assiomi di input, per ogni assioma si verifica se questo va aggiunto agli indici delle regole di inferenza. Il responsabile di tale verifica è il metodo `axiomCriterion` implementato in ogni estensione di `InferenceRule`. Nell'implementazione `ToldSuperclassesInferenceRule`, il metodo `axiomCriterion` verifica la seguente condizione su un assioma del tipo $A \sqsubseteq B$:

```
isSubclassABasicConcept(A) && isSuperclassABasicConcept(B)
```

In tal caso, l'assioma è indicizzato in CR1.

3.2.2 CR2: `IntersectionSuperclasses`

La regola di inferenza CR2 è così definita:

$$\mathsf{CR2} \quad \frac{C \sqsubseteq C',\ C \sqsubseteq C''}{C \sqsubseteq D} : C' \sqcap C'' \sqsubseteq D \in \mathcal{C}$$

Se C' e C'' sono superclassi di C, D è superclasse dell'intersezione tra C' e C'' e D non è (ancora) una superclasse di C, allora D diventa superclasse di C.
In questo caso, l'indice multi-map si presenta nella forma:

$$S_{\mathsf{CR2}}(C) = \{D \mapsto E \mid C \sqcap D \sqsubseteq E \in \mathcal{C}\}.$$

L'indice raccoglie tutti gli C che appaiono in un assioma della forma $A \sqsubseteq C$. Se in tale indice è presente il concetto C come chiave, e $Im(S_{\mathsf{CR2}}(C)) \neq \emptyset$ (codominio di S_{CR2} su C non vuoto), allora ci si può aspettare l'esistenza di un assioma del tipo $C \sqcap D \sqsubseteq E$ in $\mathcal{C}$.

Nell'implementazione `IntersectionSuperclassesInferenceRule`, il criterio di indicizzazione dell'assioma è così definito:

```
A instanceof OWLObjectIntersectionOf && isSuperclassABasicConcept(B)
```

dove con `OWLObjectIntersectionOf` si intende il concetto `A` come un'entità del tipo $C \sqcap C'$.

3.2.3 CR3: `SubclassRoleExpansion`

La regola di inferenza CR3 è così definita:

$$\mathsf{CR3} \quad \frac{C \sqsubseteq C'}{C \sqsubseteq \exists r.E} : C' \sqsubseteq \exists r.E \in \mathcal{C}$$

Se C' è superclasse di C, C' è in relazione r con E e se C non è (ancora) in relazione r con E, allora C entra in relazione r con E.
Anche in questo caso si parla di un indice multi-map, in cui la relazione è associata al *filler*:

$$S_{\mathsf{CR3}}(C) = \{r \mapsto E \mid C \sqsubseteq \exists r.E \in \mathcal{C}\}.$$

Nell'implementazione di `SubclassRoleExpansionInferenceRule`, la condizione del metodo `axiomCriterion` è così definita:

```
isSubclassABasicConcept(A) && B instanceof OWLObjectSomeValuesFrom
```

dove `OWLObjectSomeValuesFrom` identifica un concetto della forma $\exists r.C$.

3.2.4 CR4: `SuperclassRoleExpansion`

La regola di inferenza CR4 è così definita:

$$\mathsf{CR4} \quad \frac{C \sqsubseteq \exists r.D, \quad D \sqsubseteq D'}{C \sqsubseteq E} : \exists r.D' \sqsubseteq E \in \mathcal{C}$$

Se C è in relazione r con D, D' è una superclasse di D, $\exists r.D' \sqsubseteq E$ è un assioma in $\mathcal{C}$ ed E non è (ancora) superclasse di C, allora E diventa superclasse di C per la transitività.
Nel caso di CR4, la regola di inferenza deve mantenere due indici multi-map:

$$(1) \quad S_{\mathsf{CR4}_{RF}}(r) = \{A \mapsto B \mid \exists r.A \sqsubseteq B \in \mathcal{C}\}$$

L'indice (1) torna utile nel caso in cui si processi un assioma della forma $C \sqsubseteq \exists r.D$, dove andrebbe verificato che la stessa relazione r appaia anche in un assioma della forma $\exists r.D' \sqsubseteq E$.

$$(2) \quad S_{\mathsf{CR4}_{FR}}(A) = \{r \mapsto B \mid \exists r.A \sqsubseteq B \in \mathcal{C}\}$$

Allo stesso modo, l'indice (2) copre il caso di assiomi della forma $D \sqsubseteq D'$, dove è necessario individuare gli assiomi della forma $\exists r.D' \sqsubseteq E$ dove appare D'.

L'implementazione di tale regola è la classe `SuperclassRoleExpansionInferenceRule`, che include anche un attributo aggiuntivo `fillerToRole`, ossia l'indice (2). L'indice (1) è mantenuto nella generica struttura `axioms`[3].

[3] Il tipo della struttura è specificato nel *generics* della classe `InferenceRule`. Nel caso di CR4, `SuperclassRoleExpansion` estende la classe `InferenceRule<OWLObjectPropertyExpression, Map<OWLClassExpression, Set<OWLClassExpression>>>`, che permette di mappare la relazione (`OWLObjectPropertyExpression`) con il mapping del filler alla superclasse.

Il metodo `axiomCriterion` prevede la seguente condizione:

```
A instanceof OWLObjectSomeValuesFrom && isSuperclassABasicConcept(B)
```

dove `A` è un concetto della forma $\exists r.D'$.

3.2.5 CR5: `BottomSuperclassRoleExpansion`

La regola di inferenza CR5 è così definita:

$$\mathsf{CR5} \quad \frac{C \sqsubseteq \exists r.D}{C \sqsubseteq \bot} : D \sqsubseteq (\equiv)\bot$$

Se C è in relazione r con D, D è sottoclasse di $\bot$ e $\bot$ non è (ancora) superclasse di C, allora $\bot$ diventa superclasse di C. Banalmente, non ha bisogno di un indice[4].

3.2.6 CR6: `NominalChainExpansion`

La regola di inferenza CR6 è così definita:

$$\mathsf{CR6} \quad \frac{C \sqsubseteq \{a\}, \quad D \sqsubseteq \{a\}}{\{C \sqsubseteq E \mid D \sqsubseteq E\}} : C \rightsquigarrow_R D$$

Se un nominal $\{a\}$ è superclasse di due concetti, in questo caso C e D, in relazione $\rightsquigarrow_R$[5] tra loro, allora si aggiungono tutte le superclassi di D alle superclassi di C. Anch'essa non ha indice.

L'implementazione di tale regola `NominalChainExpansionInferenceRule` è l'unica regola che tratta esclusivamente entità di tipo `OWLIndividual`. Nel sistema, questo si traduce nell'aggiunta alla struttura `entityTypes`, del solo tipo

[4] Le implementazioni di `InferenceRule` senza indice sono estese da `InferenceRule<Object,Object>`.

[5] Sia definita la relazione $C \rightsquigarrow_R D$. Allora esiste una catena $C_1, \ldots, C_k$ tale che:

- **Caso 1:** $C_1 = \{b\}$.
  Poiché (assioma) $(C_1, C_2) \in R(r_1)$, con $r_1 \in R_C$, allora segue che $C \models C_1 \sqsubseteq \exists r_1.C_2$; ma poiché C_1 è un nominale, allora tutti i modelli avranno b, e da b deve partire un arco r_1 verso un individuo di tipo C_2:
  $$b \xrightarrow{r_1} C_2$$
  Ora, poiché (assioma) $(C_1, C_2) \in R(r_2)$, con $r_1 \in R_C$, allora segue che $C \models C_2 \sqsubseteq \exists r_2.C_3$; ma poiché questo è un individuo di tipo C_2, allora deve esistere un arco r_2 verso un individuo di tipo C_3:
  $$b \xrightarrow{r_1} C_2 \xrightarrow{r_2} C_3$$
  Si procede in questo modo fino a raggiungere l'individuo $C_k = D$, e quindi D non è vuoto:
  $$b \xrightarrow{r_1} C_2 \xrightarrow{r_2} C_3 \rightarrow \ldots \rightarrow C_k = D$$
- **Caso 2:** $C_1 = C$.
  Allora ci sarà una catena come la precedente che però parte da C anziché da un individuo b:
  $$C \xrightarrow{r_1} C_2 \xrightarrow{r_2} C_3 \rightarrow \ldots \rightarrow C_k = D$$

`OWLIndividual.class` (a differenza delle altre che trattano anche entità di tipo `OWLClass.class`, ossia concept name). La struttura `entityTypes` mantiene i tipi di entità che la regola di inferenza deve considerare durante l'iterazione sulla *signature* dell'ontologia in input.

4 Saturazione

La fase di saturazione è sicuramente il processo più *time-consuming* dell'architettura, ed è per questo che diventa cruciale una sua adeguata ottimizzazione. Il processo di saturazione applica esaustivamente le regole di inferenza agli assiomi in input e a quelli derivati, finché non si arriva all'inapplicabilità di nessuna di tali regole (ossia, si è arrivati a *saturazione*).

Definizione 4.1 (Saturazione di un concetto) *Sia $BC_{\mathcal{C}}$ l'insieme dei base concept di una TBox $\mathcal{C}$ in forma normale. Sia S un mapping che associa al base concept $C \in BC_{\mathcal{C}}$ un sottoinsieme di $BC_{\mathcal{C}} \cup \{\top, \bot\}$, tale che*

$$D \in S(C) \implies C \sqsubseteq_{\mathcal{C}} D$$

Si dice che si è giunti a saturazione su un concetto C se $S(C)$ contiene esattamente tutte le superclassi di C.

Dunque, più in generale, applicando le regole su tutti i concetti, si ottiene l'insieme delle superclassi di ogni concetto.

Teorema 4.1 (Verifica della sussunzione) *Sia $\mathcal{C}$ una TBox in forma normale e siano C e D concetti in $\mathcal{C}$. Se si è giunti a saturazione su C, allora*

$$\mathcal{C} \models C \sqsubseteq D \iff \mathcal{C}' \models A_C \sqsubseteq A_D$$

dove A_C e A_D sono concetti associati rispettivamente a C e D tramite il mapping S.

Dunque, saturando $\mathcal{C}$, occorre soltanto verificare se $A_D \in S(A_C)$. Nello specifico, l'algoritmo deve provare che:

- Se $A_D \in S(A_C)$ allora $C \sqsubseteq D$ (sussunzione vera).
- Se $A_D \notin S(A_C)$ allora $C \not\sqsubseteq D$ (sussunzione falsa).

Durante il processo di saturazione vengono mantenute due strutture dati: `processedAxioms`, gli assiomi processati, e `scheduledAxioms`, gli assiomi da processare. L'algoritmo preleva gli assiomi da `scheduledAxioms`, li processa spostandoli successivamente in `processedAxioms`, e le conclusioni derivate vengono aggiunte nuovamente a `scheduledAxioms`. Tuttavia, tali strutture dati non vengono mantenute a livello *globale*: per permettere una computazione concorrente degli assiomi, esse vengono frammentate in ambienti isolati detti *contesti*.

4.1 Contesti

La soluzione lock-free al problema è la seguente: il sistema distribuisce gli assiomi iniziali in contesti, i quali vengono interpretati come premesse delle regole di inferenza e processati indipendentemente dai diversi thread[1]. Di conseguenza, ogni contesto avrà la sua coda *locale*, sia per gli assiomi processati sia per quelli da processare.

Introduciamo la nozione di *contesto attivo*.

Definizione 4.2 (Contesto attivo) *Sia* C *un contesto e sia* $S(\mathsf{C})$ *il mapping che associa a* C *il suo insieme degli assiomi da processare (scheduled), allora:*

$$\mathsf{C} \textit{ è attivo} \iff S(\mathsf{C}) \neq \emptyset.$$

L'algoritmo di saturazione mantiene una coda dei contesti attivi tale che, come conseguenza della **Definizione 4.2**, un contesto C appartiene alla coda `activeContexts` se e solo se C`.scheduledAxioms` $\neq \emptyset$.
Banalmente, un contesto diventa attivo nel momento in cui un thread aggiunge un assioma da processare alla coda `scheduledAxioms`; al contrario, quando un thread accerta che la coda `scheduledAxioms` è vuota, il contesto viene disattivato. In termini di implementazione, per permettere ciò, ai contesti è associato il flag booleano `isActiveContext`.

4.1.1 Sincronizzazione dei contesti

Secondo [1], può accadere che un thread aggiunga un assioma da processare ad un contesto poco prima la sua disattivazione da parte di un altro thread, compromettendo il suo stato di attivazione (C`.isActiveContext = false` e C`.scheduledAxioms` $\neq \emptyset$, contemporaneamente). Per risolvere questa problematica, l'algoritmo di disattivazione di un contesto deve includere un controllo aggiuntivo, come mostrato in **Algoritmo 1**.

Algorithm 1 Disattivazione di un contesto C

```
C.isActiveContext ← false
if C.scheduledAxioms ≠ ∅ then
    activeContexts.activate(C)
```

L'accesso ai contesti, come per le ontologie, è gestito in maniera controllata tramite la classe `ContextAccessManager`, inclusa nel package `reasoner/saturation`. Tale manager, oltre a fornire un accesso thread-safe ai contesti, è anche responsabile della loro inizializzazione.

4.2 Implementazione: modulo `saturation`

All'interno del modulo `saturation`, la classe centrale è `OntologySaturator`: il processo di saturazione ha inizio con la chiamata al metodo `saturate`.

4.2.1 Definizione dei contesti

Il concetto di contesto è implementato nella classe astratta `InferenceRuleContext` del package `reasoner/saturation`. Data la definizione precedente di contesto, i metodi rilevanti della classe sono i seguenti:

- `Set<OWLSubClassOfAxiom> initializeContext()`: inizializzazione del contesto;

- `boolean scheduleAxiom(OWLSubClassOfAxiom axiom)`: aggiunta di un assioma alla coda degli assiomi da processare;
- `OWLSubClassOfAxiom pollScheduledAxiom()`: ottenimento del prossimo assioma da processare;
- `boolean isActiveContext()`: flag per lo stato di attivazione del contesto;
- `Set<OWLSubClassOfAxiom> compute(OWLSubClassOfAxiom axiom)`: implementazione dell'algoritmo per la derivazione delle conclusioni per quella specifica regola di inferenza.
- Ulteriori metodi di utilità: `hasBeenInitialized`, `hasScheduledAxioms`, `getProcessedAxioms` ecc.

Inoltre, all'interno del contesto, viene anche mantenuta la regola di inferenza (`InferenceRule inferenceRule`) per permettere l'accesso agli indici durante la fase di derivazione delle conclusioni.

Le estensioni della classe astratta `InferenceRuleContext` devono implementare i metodi precedentemente esposti, adattandoli alle esigenze della regola di inferenza specifica. Tali implementazioni, presentate di seguito, sono incluse nel modulo `elppreasoner/saturation/contexts`:

- **CR1**: `ToldSuperclassesIRContext`;
- **CR2**: `IntersectionSuperclassesIRContext`;
- **CR3**: `SuperclassRoleExpansionIRContext`;
- **CR4:** `SubclassRoleExpansionIRContext`;
- **CR5**: `BottomSuperclassRoleExpansionIRContext`;
- **CR6:** `NominalChainExpansionIRContext`.

4.2.2 Assegnazione dei contesti

Inizialmente, il saturatore ordina al `ContextAccessManager` di assegnare i contesti alle entità della *signature* dell'ontologia. Per ogni *entity* (nominal o concept) vengono creati dei contesti, la cui assegnazione varia in base alla compatibilità delle regole di inferenza con quel tipo di entità, implementati estendendo la classe `InferenceRuleContext` (ad esempio, per la regola d'inferenza `ToldSuperclassesInferenceRule`, esiste il tipo di contesto `ToldSuperclassesIRContext`, esteso da `InferenceRuleContext`).
Per gestire la creazione dei contesti e la loro assegnazione alle rispettive entità, è stata realizzata la classe di utilità `ContextProvider`, che viene istanziata per ogni regola di inferenza definita. Oltre all'istanziazione dei contesti, il `ContextProvider` è anche responsabile del raggruppamento dei contesti per regola di inferenza.

Esempio 4.1. Consideriamo l'implementazione della regola di inferenza CR6, ossia `NominalChainExpansionInferenceRule`. Se l'iterazione sulla signature dell'ontologia sta correntemente considerando una entità di tipo concept[6], i `ContextProvider` delle regole da CR1 a CR5 istanzieranno i rispettivi contesti per tale entità, a differenza del *provider* per CR6 che, come ricordiamo, non considera entità di tipo concept (ma solo nominali, ossia `INDIVIDUAL`).
Invece, nel caso di una entità nominale, i `ContextProvider` istanzieranno i contesti per tutte le regole di inferenza, poiché le regole da CR1 a CR5 considerano entrambi i tipi `INDIVIDUAL` (nominale) e `CLASS` (concept), incluso CR6 che considera esclusivamente quelle di tipo `INDIVIDUAL`.

Algorithm 2 Assegnazione dei contesti

```
    Input: TBox C, inferenceRules
    activeContexts ← ∅
    discardedAxioms ← ∅
    providers ← initProviders(inferenceRules)
    for A ∈ C.signature do
        for provider ∈ providers do
7:          context = provider.createContextByEntity(entity)
            provider.addContext(A, context)
    for axiom ∈ C.axioms do
        initializeAxiom(axiom)
```

La chiamata a `provider.createContextByEntity` alla riga 7 potrebbe non ritornare un contesto, come nel caso dell'**Esempio 4.1** dove il provider per CR6 non prevede l'assegnazione di un contesto per entità di tipo `CLASS`.

4.2.3 Inizializzazione dei contesti

Una volta inizializzati i contesti per ogni entità, vuoti a questo punto, si procede con l'inizializzazione degli assiomi nei rispettivi contesti.
Per ogni contesto C, che sappiamo essere associato ad una entità A, vengono derivati due assiomi, detti `baseAxioms`:

$$\overline{A \sqsubseteq A}, \quad \overline{A \sqsubseteq \top}$$

In particolare, è il metodo `initializeContext` a ritornare tali assiomi, come vedremo nell'algoritmo che segue. L'utilità dei `ContextProvider` raggiunge l'apice in questa fase: per ogni assioma dell'iterazione, essi individuano rapidamente i contesti che devono processarlo (riga 2 e 9 dell'**Algoritmo 3**), avendo precedentemente raggruppato i contesti per regola di inferenza.

[6] Il reasoner discrimina le entità sulla base dei valori dell'enumeratore `OWLEntityType`: `INDIVIDUAL` (nominali), `CLASS` (concetto) e `ENTITY` (altre entità).

Algorithm 3 `initializeAxiom`: inizializzazione assioma nei contesti

```
1:  Input: axiom Ax
2:  contexts ← getContextsByAxiom(Ax)
3:  if contexts = ∅ then
4:      discardedAxioms.add(Ax)
5:      return
6:  for C ∈ contexts do
7:      baseAxioms ← C.initializeContext()
8:      for baseAxiom ∈ baseAxioms do
9:          baseContexts ← getContextsByAxiom(baseAxiom)
10:         for baseContext ∈ baseContexts do
11:             baseContext.scheduleAxiom(baseAxiom)
12:     C.scheduleAxiom(Ax)
13:     activeContexts.activateContext(C)
```

4.3 Algoritmo di saturazione

Vediamo nel dettaglio l'algoritmo di saturazione del `OntologySaturator`:

Algorithm 4 `saturate`: saturazione

```
1:  for ;; do
2:      C = activeContexts.poll()
3:      for ;; do
4:          Ax = C.pollScheduledAxiom()
5:          if C.hasProcessedAxiom(Ax) then
6:              continue
7:          C.addProcessedAxiom(Ax)
8:          conclusions = C.compute(Ax)
9:          processedContexts.add(C)
10:         for Ax2 ∈ conclusions do
11:             for C2 ∈ getContextsByAxiom(Ax2) do
12:                 C2.scheduleAxiom(Ax2)
13:                 activateContext(C2)
14:     deactivateContext(C).
```

L'**Algoritmo 4** può essere eseguito da thread differenti. Notiamo come le conclusioni ottenute vengono poi riassegnate ai contesti interessati (riga 10-13), finché non vi sono più contesti attivi. Di seguito descriveremo nel dettaglio le diverse implementazioni del metodo `compute` (riga 8), incluse nel package `elppreasoner/saturation/contexts`.

4.3.1 Implementazione per CR1

Il contesto della regola **CR1**, implementato in `ToldSuperclassesIRContext`, sfrutta l'indice della regola per individuare la seconda premessa negli assiomi di input.

Algorithm 5 `compute` per **CR1**

```
1: Input: assioma C ⊑ D
2: conclusions ← ∅
3: index ← rule.axioms // Gli assiomi indicizzati per tale regola (CR1)
4: if !index.containsKey(D) then
5:     return ∅
6: for D ⊑ D̄ ∈ index.get(D) do
7:     conclusions.add(C ⊑ D̄)
8: return conclusions
```

Dato un concetto C rappresentativo del contesto C, gli assiomi processati in tale contesto hanno la seguente forma:

$$\mathsf{C}_{\mathrm{CR1}}(C).\mathit{processedAxioms} = \{C \sqsubseteq D \mid C \sqsubseteq D \text{ processato}\}$$

4.3.2 Implementazione per CR2

Per la regola **CR2**: Dopo aver ottenuto la seconda premessa dall'indice dalla

Algorithm 6 `compute` per **CR2**

```
1: Input: assioma C ⊑ D
2: Scope: contesto C, regola rule
3: conclusions ← ∅
4: index ← rule.axioms
5: if !index.containsKey(D) then
6:     return ∅
7: for D ⊓ D̄ ⊑ E ∈ index.get(D) do
8:     if C.hasProcessedAxiom(C ⊑ D̄) then
9:         conclusions.add(C ⊑ E)
10: return conclusions
```

regola, l'algoritmo verifica che l'assioma $C \sqsubseteq \bar{D}$ sia già stato processato, in modo da derivarne la conclusione come previsto. La struttura degli assiomi processati ha la seguente forma:

$$\mathsf{C}_{\mathrm{CR2}}(C).\mathit{processedAxioms} = \{C \sqsubseteq D \mid C \sqsubseteq D \text{ processato}\}$$

4.3.3 Implementazione per CR3

Per la regola **CR3**:

Algorithm 7 `compute` per **CR3**

1: **Input:** assioma $C \sqsubseteq D$
2: conclusions $\leftarrow \emptyset$
3: index $\leftarrow$ rule.axioms
4: **if** !index.containsKey(D) **then return** $\emptyset$
5: **for** $D \sqsubseteq \exists r.\bar{D} \in$ index.get(D) **do** conclusions.add($C \sqsubseteq \exists r.\bar{D}$)
6: **return** conclusions

Banalmente:

$$\mathsf{C}_{\mathrm{CR3}}(C).\mathit{processedAxioms} = \{C \sqsubseteq D \mid C \sqsubseteq D \text{ processato}\}$$

4.3.4 Implementazione per CR4

La regola **CR4** necessita dell'indice multi-map ausiliario `fillerToRole` per individuare una relazione r partendo da un filler D', e ottenere assiomi della forma $\exists r.D' \sqsubseteq E$ dove appare D' (vedere sez. **3.2.4**). Tale necessità deriva dalle due casistiche che l'algoritmo deve trattare.

Algorithm 8 `compute` per **CR4**: caso $C \sqsubseteq \exists r.D$

1: **Input:** assioma $C \sqsubseteq \exists r.D$
2: **Scope:** contesto C, regola *rule*
3: conclusions $\leftarrow \emptyset$
4: index $\leftarrow$ rule.axioms
5: **if** !index.containsKey(r) **then return** $\emptyset$
6: **for** $\exists r.D' \sqsubseteq E \in$ index.get(r) **do**
7: **if** C.hasProcessedAxiom($D \sqsubseteq D'$) **then**
8: conclusions.add($C \sqsubseteq E$)
9: **return** conclusions

Algorithm 9 `compute` per **CR4**: caso $D \sqsubseteq B$

1: **Input:** assioma $D \sqsubseteq B$
2: **Scope:** contesto C, regola *rule*
3: conclusions $\leftarrow \emptyset$
4: index $\leftarrow$ rule.fillerToRole // Indice ausiliario
5: **if** !index.containsKey(B) **then return** $\emptyset$
6: **for** $\exists r.B \sqsubseteq E \in$ index.get(B) **do**
7: **if** C.subByPropertyProcessedAxioms.contains(r) **then**
8: **for** $C' \sqsubseteq \exists r.D \in$ subByPropertyProcessedAxioms.get(r) **do**
9: conclusions.add($C' \sqsubseteq E$)
10: **return** conclusions

La struttura `subByPropertyProcessedAxioms`, che compare alla riga 7 e 8

dell'**Algoritmo 9**, è un insieme ausiliario degli assiomi processati raggruppati per relazione r; in particolare tale struttura rappresenta un accesso rapido agli assiomi di forma $C \sqsubseteq \exists r.D$ del primo caso (**Algoritmo 8**). Dunque, possiamo definire per il contesto in questione le seguenti strutture:

$$\mathsf{C}_{\mathrm{CR4}}(C).\mathit{processedAxioms} = \{C \sqsubseteq D \mid C \sqsubseteq D \text{ processato}\}$$

$$\mathsf{C}_{\mathrm{CR4}}(C).\mathit{subByPropertyProcessedAxioms}(r) = \\ = \{C \sqsubseteq \exists r.E \mid C \sqsubseteq \exists r.E \text{ processato}\}$$

4.3.5 Implementazione per CR5

Anche per la regola **CR5** si presentano due casi.

Algorithm 10 `compute` per **CR5**: caso $C \sqsubseteq \exists r.D$

```
1: Input: assioma C ⊑ ∃r.D
2: Scope: contesto C
3: conclusions ← ∅
4: if C.hasProcessedAxiom(D ⊑ ⊥) then
5:     conclusions.add(C ⊑ ⊥)
6: return conclusions
```

Algorithm 11 `compute` per **CR5**: caso $D \sqsubseteq \bot$

```
1: Input: assioma D ⊑ ⊥
2: Scope: contesto C
3: conclusions ← ∅
4: for C ⊑ ∃r.D ∈ C.processedAxioms do
5:     conclusions.add(C ⊑ ⊥)
6: return conclusions
```

Banalmente:

$$\mathsf{C}_{\mathrm{CR5}}(C).\mathit{processedAxioms} = \{C \sqsubseteq \exists r.D \mid C \sqsubseteq \exists r.D \text{ processato}\}$$

4.3.6 Implementazione per CR6

L'implementazione della regola **CR6** richiede diverse soluzioni per le necessità presentate:

- Un grafo orientato che rappresenta la relazione $C \rightsquigarrow_R D$ (un arco da C a D esiste se e solo se l'assioma $C \sqsubseteq \exists r.D$ è processato);
- Una struttura che mantiene tutti i nominali processati. Sia A un nominale rappresentativo del contesto:

$$\mathsf{C}_{\mathrm{CR6}}(A).\mathit{individuals} = \{\{a\} \mid \{\{a\} \text{ processato}\}$$

- Un insieme delle sottoclassi di nominali processate. Sia A un nominale rappresentativo del contesto:

$$\mathsf{C}_{\mathrm{CR6}}(A).\mathit{subclasses} = \{C \sqsubseteq A \mid C \sqsubseteq A \text{ processato}\}$$

Algorithm 12 `compute` per **CR6**

```
1:  Input: assioma Ax (C ⊑ E oppure C ⊑ ∃r.D)
2:  Scope: contesto C, nominale rappresentativo A
3:  conclusions ← ∅
4:  if ‖C.subclasses‖ < 2 then
5:      return ∅
6:  for C ⊑ A ∈ subclasses do
7:      for D ⊑ A ∈ subclasses do
8:          if C.relationGraph.reach(C, D) then
9:              for D ⊑ E ∈ C.superclassesBySubclass.get(D) do
10:                 conclusions.add(C ⊑ E)
11: return conclusions
```

Dovendo processare assiomi di due forme, il contesto mantiene due strutture per gli assiomi processati. Sia A un nominale rappresentativo del contesto:

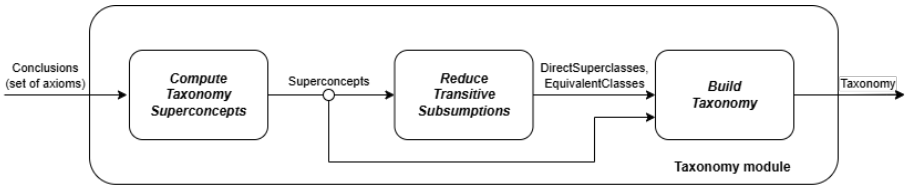
$$\mathsf{C}_{\mathrm{CR6}}(A).\mathit{processedAxioms} = \{C \sqsubseteq \exists r.E \mid C \sqsubseteq \exists r.E \text{ processato}\}$$

$$\mathsf{C}_{\mathrm{CR6}}(A).\mathit{superclassesBySubclass}(C) = \{C \sqsubseteq D \mid C \sqsubseteq D \text{ processato}\}$$

Nota. Per gestire il grafo orientato, è stata implementata la classe `RelationGraph` nel package `elppreasoner/saturation/utils`.

5 Tassonomia

La fase conclusiva del ragionatore è la costruzione di una tassonomia, ovvero la classificazione gerarchica dei concetti su cui è stato effettuato il reasoning. Infatti, come descritto in [2], nella fase di saturazione si calcolano le relazioni di sussunzione, chiuse rispetto alla transitività, tra le varie classi dell'ontologia fornita in input al ragionatore. Tuttavia, il risultato atteso della classificazione è una tassonomia che contenga soltanto sussunzioni *dirette* ed eventuali condizioni di equivalenza tra i nodi, che rappresentano i concetti della TBox $\mathcal{T}$. Ciò significa che, se la tassonomia contiene $A \sqsubseteq B$ e $B \sqsubseteq C$, allora essa non deve contenere $A \sqsubseteq C$, a meno che non siano equivalenti. Di conseguenza, le sussunzioni tra in concetti atomici devono essere *ridotte transitivamente*.



Dopo aver ottenuto l'insieme delle conclusioni (assiomi) dalla fase di saturazione, il ragionatore proposto esegue il calcolo della tassonomia in tre fasi:

1. Calcolare tutti i "superconcetti" degli assiomi scartando le sussunzioni che coinvolgono i concetti non atomici
2. Ridurre le sussunzioni transitive tra i concetti (*in parallelo* [7])
3. Costruire la tassonomia in base in base ai risultati raccolti nei due step precedenti

In sintesi, dato un insieme di assiomi, l'obiettivo è costruire una tassonomia che contenga informazioni sia sulle sussunzioni dirette tra i concetti sia sulle loro equivalenze. Al termine di questa fase, la tassonomia manterrà anche le informazioni sulle sussunzioni non-dirette tra i concetti per semplificare la fase di testing: infatti, l'interfaccia `OWLReasoner` mette a disposizione due metodi,

- `NodeSet<OWLClass> getSubClasses(OWLClassExpression ce, boolean direct)`, e
- `NodeSet<OWLClass> getSuperClasses(OWLClassExpression ce, boolean direct)`,

il cui parametro `direct` consente all'utente di richiedere sia le sottoclassi/superclassi dirette (passando `true`) sia quelle indirette (passando `false`).

[7] In [2], è soltanto accennata la possibilità di costruire la tassonomia in parallelo. `ELPPReasoner` rende concreta questa possibilità, parallelizzando l'algoritmo di riduzione transitiva per una migliore performance.

Nell'implementazione di `ELPPReasoner`, l'equivalenza, le sussunzioni dirette e non dirette, sono tutte informazioni conservate in tre hashmap: quindi, il testing del reasoner si traduce a delle semplici chiamate a queste mappe.

5.1 Calcolo dei "superconcetti"

In questa fase vengono calcolati tutti i superconcetti delle classi coinvolte negli assiomi calcolati nella fase di saturazione. Per ogni assioma del tipo $A \sqsubseteq B$ (GCI):

1. Se A è una classe e non è $\bot$, aggiungi A come superconcetto di $\bot$. Inoltre, se $A = \exists r.C$:
   (a) Se C non è $\bot$, aggiungi C come superconcetto di $\bot$
   (b) Se C non è $\top$, aggiungi $\top$ come superconcetto di C
2. Se B è una classe e non è $\bot$, aggiungi B come superconcetto di $\bot$. Inoltre, se $B = \exists r.C$:
   (a) Se C non è $\bot$, aggiungi C come superconcetto di $\bot$
   (b) Se C non è $\top$, aggiungi $\top$ come superconcetto di C
3. Se A e B sono dei letterali (classe o negato di una classe) o delle intersezioni tra concetti:
   (a) Se A non è $\bot$, aggiungi A come superconcetto di $\bot$
   (b) Se A non è $\top$, aggiungi B come superconcetto di A
   (c) Se B non è $\bot$, aggiungi B come superconcetto di $\bot$
   (d) Se B non è $\top$, aggiungi $\top$ come superconcetto di B

5.2 Riduzione delle sussunzioni transitive (concorrente)

L'algoritmo per la riduzione delle sussunzioni transitive è l'"Algorithm 3: Better Transitive Reduction" proposto da [2], posto all'inizio della seguente pagina. Esso richiede l'insieme dei superconcetti, restituito al passo precedente. Come introdotto all'inizio del capitolo, il nostro ragionatore `ELPPReasoner` rende possibile eseguire la riduzione delle sussunzioni transitive in parallelo, in questo modo:

1. Se viene passato `true` al corrispondente argomento del suo costruttore, ottieni il numero di processori disponibili `cpuCount`
2. Dividi i `superConcepts` in `cpuCount` parti uguali, assegnando ad ogni thread la parte spettante
3. Esegui l'Algorithm 3 su ogni thread in parallelo. L'algoritmo, per ogni superconcetto A, calcola due insiemi: A`.equivalentConcepts`, che contiene tutti i concetti equivalenti ad A (A stesso è ivi incluso: si suppone

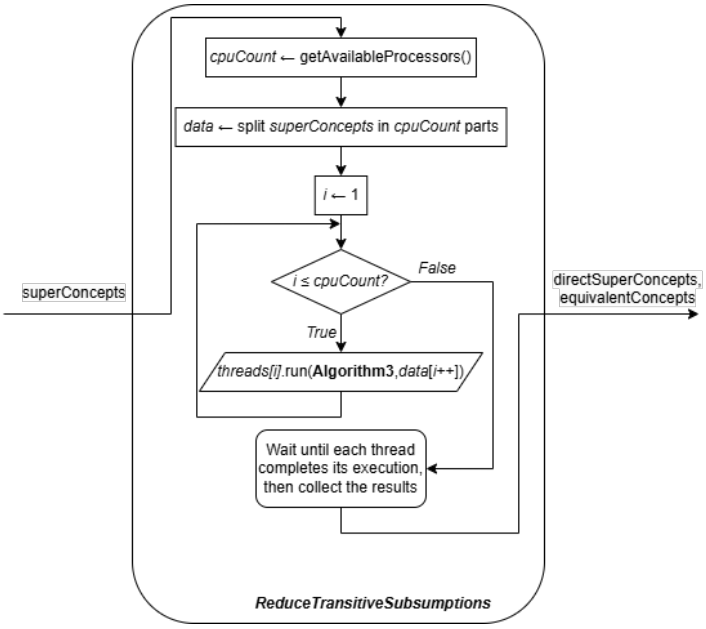
che $A \equiv A$), e A.`directSuperConcepts`, che contiene tutti i superconcetti diretti di A.

4. Raggruppa i risultati di ogni thread in un unico risultato

Algorithm 3 Better Transitive Reduction

```
for C ∈ A.superConcepts do
    if A ∈ C.superConcepts then
        A.equivalentConcepts.add(C)
    else
        isDirect ← true                    ▷ so far C is a direct superconcept of A
        for B ∈ A.directSuperConcepts do
            if C ∈ B.superConcepts then
                isDirect ← false           ▷ C is not a direct superconcept of A
                break
            if B ∈ C.superConcepts then
                A.directSuperConcepts.remove(B)        ▷ B is not a direct superconcept of A
        if isDirect then
            A.directSuperConcepts.add(C)
```

Quanto descritto è schematizzato nella seguente figura, in cui si assume di voler eseguire la costruzione della tassonomia in parallelo e che il numero di thread disponibili sia un divisore del numero dei superconcetti, per semplicità di rappresentazione. Naturalmente, nell'implementazione è opportunamente gestito il caso opposto.



5.3 Costruzione della tassonomia

Di fatto, il nucleo del modulo `taxonomy` è rappresentato dalla riduzione delle sussunzioni transitive. In questa fase, il ragionatore ha a disposizione, per ogni concetto della gerarchia, i suoi superconcetti (diretti e non) e i concetti ad esso equivalenti. Tuttavia, è necessario un ultimo semplice passaggio per adattare il risultato ottenuto con quanto richiesto dall'interfaccia `OWLReasoner`, ovvero la creazione di nodi (di tipo `OWLNode`) e relazioni tra questi ultimi. Ogni `OWLNode` rappresenta un concetto, ed una relazione (diretta) tra un nodo A e un nodo B indica che A è sussunto (direttamente) da B, ovvero $A \sqsubseteq B$, rispetto a $\mathcal{T}$.

Infatti, i seguenti metodi dell'interfaccia `OWLReasoner` implementati da `ELPPReasoner` di fatto chiamano il metodo corrispondente dell'oggetto `Taxonomy`, ottenuto proprio come output di quest'ultima fase:

- `Node<OWLClass> getTopClassNode()`
- `Node<OWLClass> getBottomClassNode()`
- `NodeSet<OWLClass> getSubClasses(OWLClassExpression ce, boolean direct)`
- `NodeSet<OWLClass> getSuperClasses(OWLClassExpression ce, boolean direct)`
- `Node<OWLClass> getEquivalentClasses(OWLClassExpression ce)`

Segue un paragrafo conclusivo, in cui sono posti i più importanti dettagli implementativi del modulo `taxonomy`.

5.4 Implementazione: modulo `taxonomy`

Questo paragrafo conclude di fatto la parte implementativa di `ELPPReasoner`. Lo stesso modulo è diviso in due parti: `reasoner/taxonomy` e `elppreasoner/taxonomy`. Il primo package contiene l'oggetto `Taxonomy` e l'interfaccia `TaxonomyBuilder`, più generali, mentre il secondo package contiene il costruttore della tassonomia e una classe di utilità che contiene tutti i metodi necessari.

5.4.1 `reasoner/taxonomy`

`TaxonomyBuilder` è un'interfaccia che contiene il seguente metodo:

```
Taxonomy build(Set<OWLSubClassOfAxiom> axioms)
```

Idealmente, l'oggetto preposto alla costruzione della tassonomia deve implementare questa classe. Un esempio è `ELPPTaxonomyBuilder`, incluso nel package `elppreasoner/taxonomy`.

`Taxonomy` è l'oggetto che rappresenta una tassonomia di classi. Contiene informazioni su:

- Equivalenza tra concetti (`classToEquivalentClasses`)
- Sussunzione tra concetti (`nodeToAllSubclasses`, `nodeToAllSuperclasses`)
- Sussunzione diretta tra concetti (`nodeToDirectSubclasses`, `nodeToDirectSuperclasses`)

Sia chiaro che questo oggetto non rappresenta una struttura a grafo e non fornisce metodi per costruire una tassonomia; bensì, fornisce degli attributi che rappresentano tutte le informazioni necessarie su una tassonomia, che dev'essere costruita con un `TaxonomyBuilder`.

Segue il workflow atteso per la costruzione della tassonomia:

1. Scrivere una classe manager che implementi `TaxonomyBuilder`
2. Nel metodo `build()` del `TaxonomyBuilder` implementato, scrivere la logica desiderata
3. Dopo la computazione al passo precedente, riempire i campi dell'oggetto `Taxonomy`

Un esempio è posto nel paragrafo seguente.

5.4.2 `elppreasoner/taxonomy`

Il ragionatore proposto implementa un modulo apposito per la costruzione della tassonomia, che segue passo dopo passo quanto viene dettagliato nei Paragrafi 5.1, 5.2 e 5.3 di questo Capitolo.

`ELPPTaxonomyBuilder` è una classe helper che facilita la costruzione di una tassonomia per le ontologie $\mathcal{EL}^{++}$. Implementa l'interfaccia `TaxonomyBuilder`: il metodo `Taxonomy build(Set<OWLSubClassOfAxiom> axioms)`, dato un insieme di conclusioni ottenute al termine della fase di saturazione, costruisce una tassonomia con informazioni su sussunzione diretta/indiretta ed equivalenza di concetti.

`TaxonomyUtilities` è la classe di utility che racchiude un metodo per ogni fase di costruzione della tassonomia:

- `Map<OWLClassExpression, Set<OWLClassExpression>> computeTaxonomySuperConcepts(Set<OWLSubClassOfAxiom> axioms)`
  Il calcolo dei superconcetti, visto nel Paragrafo 5.1.
- `TaxonomyReductionPOJO reduceTransitiveSubsumptions(Map<OWLClassExpression, Set<OWLClassExpression>> taxonomySuper-`

`Concepts, boolean concurrentMode)`
La riduzione delle sussunzioni transitive, vista nel Paragrafo 5.2.

- `Taxonomy buildTaxonomy(Map<OWLClassExpression, Set<OWLClassExpression>> taxonomySuperConcepts, Map<OWLClassExpression, Set<OWLClassExpression>> taxonomyEquivalentConcepts, Map<OWLClassExpression, Set<OWLClassExpression>> taxonomyDirectSuperConcepts)`
  La costruzione della tassonomia, vista nel Paragrafo 5.3.

6 Valutazione

Parte conclusiva dell'implementazione, nonché integrante e rilevante per questo lavoro, è la valutazione del ragionatore `ELPPReasoner`. Ogni modulo dell'applicazione è stato testato su una semplice ontologia creata ad-hoc con Protégé[8], Italian Food, e tre ontologie preesistenti: SCTO (una versione semplificata di SNOMED CT), GALEN e GENE Ontology. Infine, è stata valutata la performance del ragionatore e confrontata con due noti reasoner per il profilo OWL2-EL presenti in letteratura: ELK ([2]) e HermiT ([5]).

6.1 Ontologie coinvolte

In questo paragrafo verrà brevemente descritta ogni ontologia utilizzata per il testing, con maggiori dettagli riguardanti l'ontologia creata appositamente per valutare nello specifico il frammento della logica descrittiva $\mathcal{EL}^{++}$ considerato.

6.1.1 Italian Food

Italian Food è una semplice ontologia creata con Protégé che vuole rappresentare formalmente un sottoinsieme estremamente piccolo del dominio enogastronomico italiano. L'ontologia di base è costituita dai seguenti assiomi:

$$MargheritaTopping \equiv \exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella$$

$$PizzaMargherita \equiv Pizza \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella)$$

$$MarinaraTopping \equiv \exists hasIngredient.Tomato \sqcap \exists hasIngredient.Oregano$$

$$PizzaMarinara \equiv Pizza \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella)$$

$$Pizza \equiv PizzaBase \sqcap Topping$$

$$Food \sqsubseteq \top$$

$$Pizza \sqsubseteq Food$$

$$Bread \sqsubseteq Food$$

$$Tomato \sqsubseteq Food$$

$$Mozzarella \sqsubseteq Food$$

$$Oregano \sqsubseteq Food$$

$$MargheritaTopping \sqsubseteq Food$$

$$MarinaraTopping \sqsubseteq Food$$

$$PizzaBase \sqsubseteq Food$$

$$Bread \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella) \sqsubseteq Food$$

$$Bread \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Oregano) \sqsubseteq Food$$

$$\exists hasTopping.(\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella) \sqsubseteq MargheritaBasedFood$$

$$\exists hasTopping.(\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Oregano) \sqsubseteq MarinaraBasedFood$$

$$\bot \sqsubseteq PizzaMargherita$$

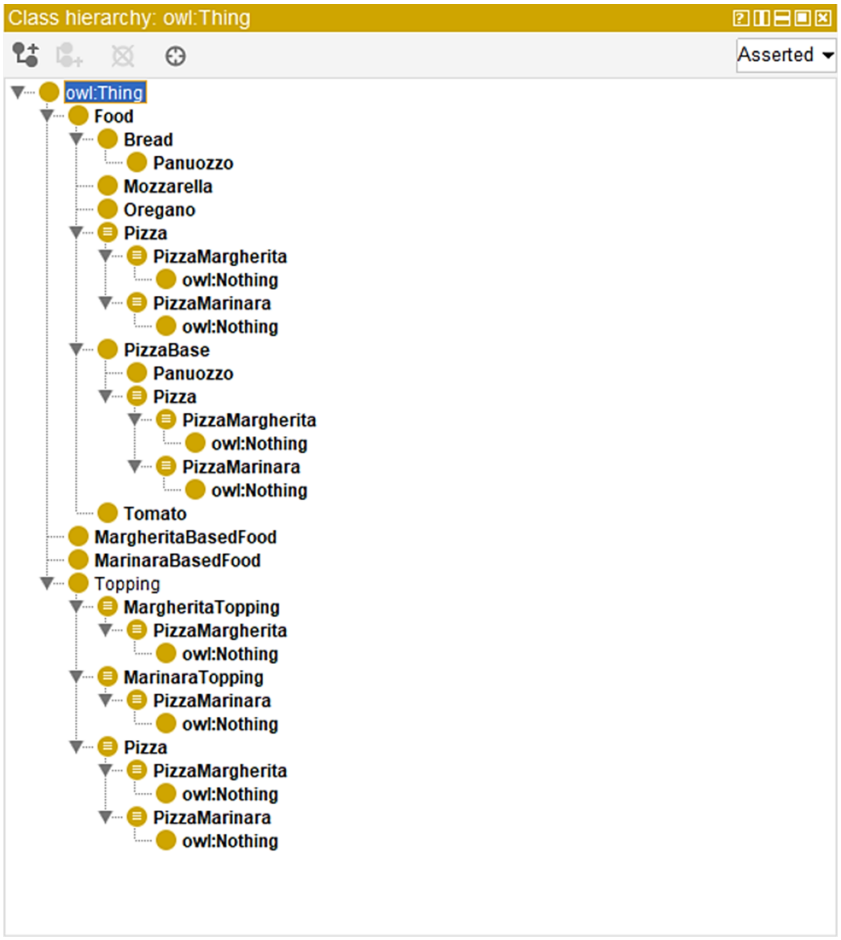
$$\bot \sqsubseteq PizzaMarinara$$

[8] *A free, open-source ontology editor and framework for building intelligent systems*: https://protege.stanford.edu/

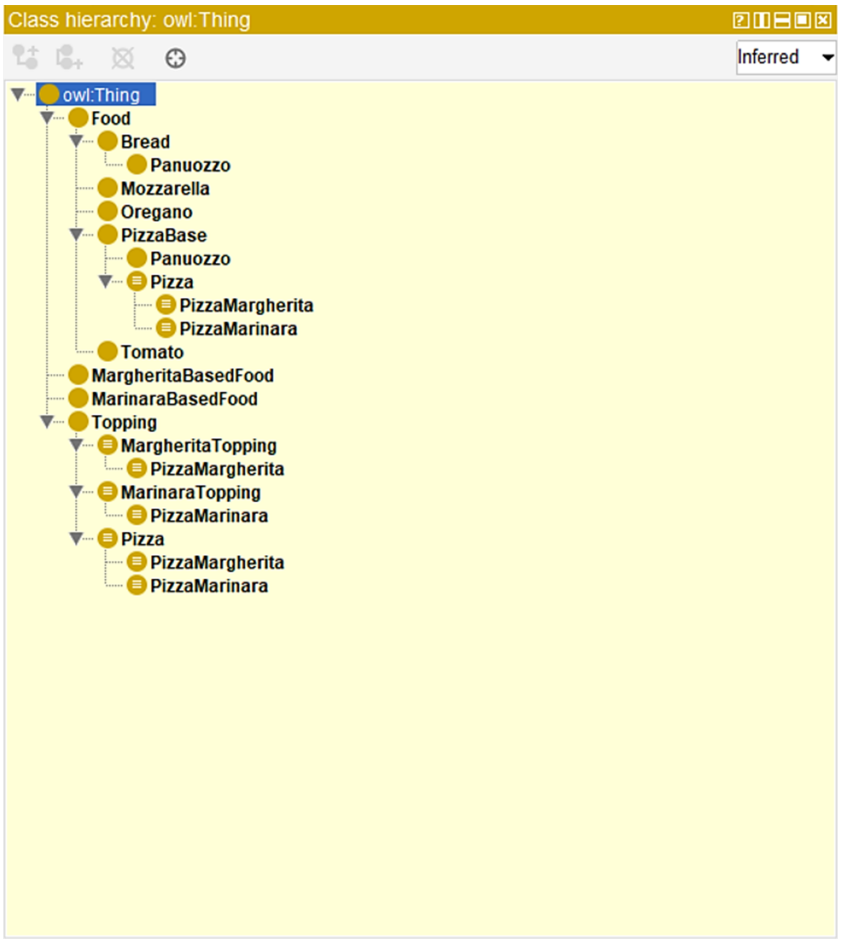
$$Pizza \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella) \sqsubseteq Pizza$$
$$Pizza \sqcap (\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Oregano) \sqsubseteq Pizza$$
$$MargheritaBasedFood \sqsubseteq \exists hasTopping.(\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Mozzarella)$$
$$MarinaraBasedFood \sqsubseteq \exists hasTopping.(\exists hasIngredient.Tomato \sqcap \exists hasIngredient.Oregano)$$
$$Panuozzo \sqsubseteq PizzaBase \sqcap Bread$$

Come si può osservare, vi sono cinque assiomi di equivalenza e venti assiomi "sottoclasse di". Gli assiomi di equivalenza vanno scomposti in assiomi di tipo "sottoclasse di": questa operazione viene eseguita in automatico da `ELPPReasoner`. Degli assiomi "sottoclasse di" elencati, gli ultimi undici devono essere normalizzati con le regole di normalizzazione opportunamente definite (si rimanda il lettore al Capitolo apposito), rispettivamente: `NF2`, `NF2`, `NF3`, `NF3`, `NF4`, `NF4`, `NF5`, `NF5`, `NF6`, `NF6`, `NF7`.

Dalla scheda *General class axioms* di Protégé è possibile inserire manualmente gli assiomi in un prompt per importarli nell'ontologia. Ecco come appare la gerarchia delle classi asserita dal software (*Asserted*) in seguito all'importazione:



Per testare il funzionamento del software, è stato eseguito il ragionatore ELK, integrato in Protégé 5.6.3 nella sua versione 0.5.0, sull'ontologia Italian Food. Terminato il reasoning, la gerarchia delle classi dedotta da ELK (*Inferred*) è la seguente:



6.1.2 SCTO

SCTO[9] è una semplificazione dell'ontologia SNOMED CT (Systematized Nomenclature of Medicine Clinical Term Ontology). L'ontologia, che ricade al 100% nella logica descrittiva $\mathcal{EL}^{++}$, implementa il modello concettuale di SNOMED CT aggiungendo assiomi ai suoi concetti top-level per prevenire ridondanze e inconsistenze dovute alla sua grande dimensione.

[9] https://bioportal.bioontology.org/ontologies/SCTO (descrizione + download)

Disclaimer. Questa ontologia è stata importata in Protégé e, in seguito alle inferenze calcolate da ELK, ne è stata esportata una versione modificata, in OWL Functional Syntax, che include soltanto assiomi di tipo `SubClassOf` e `EquivalentClasses`. Ovvero, sono stati rimossi tutti gli assiomi che varcavano lo scope del nostro ragionatore. L'ontologia modificata risultante include 314 assiomi di tipo `SubClassOf` e 2 assiomi di tipo `EquivalentClasses`.

6.1.3 GALEN

GALEN[10] è un'ontologia che traduce l'ontologia Galen completa (dal progetto OpenGALEN) nella logica descrittiva OWL. Più del 95% degli assiomi originali dell'ontologia Galen può essere espresso in $\mathcal{EL}^{++}$.

Disclaimer. Questa ontologia è stata importata in Protégé e, in seguito alle inferenze calcolate da ELK, ne è stata esportata una versione modificata, in OWL Functional Syntax, che include soltanto assiomi di tipo `SubClassOf` e `EquivalentClasses`. Ovvero, sono stati rimossi tutti gli assiomi che varcavano lo scope del nostro ragionatore. L'ontologia modificata risultante include 36.887 assiomi di tipo `SubClassOf` e 266 assiomi di tipo `EquivalentClasses`.

6.1.4 GENE Ontology

GENE Ontology [11] (GO) fornisce vocabolari strutturati per l'annotazione di prodotti genici rispetto alla loro funzione molecolare, componente cellulare e ruolo biologico. L'ontologia ricade al 100% nella logica descrittiva $\mathcal{EL}^{++}$.

Disclaimer. Questa ontologia è stata importata in Protégé e, in seguito alle inferenze calcolate da ELK, ne è stata esportata una versione modificata, in OWL Functional Syntax, che include soltanto assiomi di tipo `SubClassOf`. Ovvero, sono stati rimossi tutti gli assiomi che varcavano lo scope del nostro ragionatore. L'ontologia modificata risultante include 66.172 assiomi di tipo `SubClassOf` e nessun assioma di tipo `EquivalentClasses`, giacché l'ontologia originale non ne conteneva alcuno.

Tutti i file delle ontologie sono inclusi ed utilizzati nel testing del ragionatore e possono essere visionati nella cartella `src/test/resources/ontologies`.

[10] https://bioportal.bioontology.org/ontologies/GALEN (descrizione + download)
[11] https://bioportal.bioontology.org/ontologies/GO (descrizione)
https://geneontology.org/docs/download-ontology (download)

6.2 Testing

Il testing funzionale del ragionatore `ELPPReasoner` - package `src/test/java` - è stato suddiviso in tre test suite, una per ogni modulo:

- `normalization`: include la classe di test `ELPPOntologyNormalizer_Test` per la normalizzazione
- `saturation`: include la classe di test `OntologySaturator_Test` per la saturazione
- `taxonomy`: include la classe di test `ELPPTaxonomyBuilder_Test` per la costruzione della tassonomia

6.2.1 Testing della normalizzazione: `ELPPOntologyNormalizer_Test`

Italian Food (`ItalianFood_NormalizationTest`). Per questa ontologia sono stati realizzati due test case:

- `isGCIInNormalForm_test()`: controlla se l'ontologia Italian Food pre-normalizzazione contiene il numero atteso di assiomi normalizzati.
  TEST SUPERATO
- `ItalianFood_normalize()`: normalizza l'ontologia Italian Food e controlla se essa contiene il numero atteso di assiomi normalizzati.
  TEST SUPERATO

L'ontologia normalizzata è conservata in `italian-food-normalized.xml`. Questo file può essere importato in Protégé per una migliore visualizzazione. I 70 assiomi normalizzati formalmente definiti possono essere visualizzati nel file `Italian Food.docx` a pagina 2, non inseriti di seguito per questioni di spazio.

SCTO (`SCTO_NormalizationTest`). Per questa ontologia, già normalizzata in seguito al calcolo delle inferenze e all'esportazione tramite Protégé, è stato realizzato un test case:

- `isGCIInNormalForm_test()`: controlla se tutti gli assiomi dell'ontologia SCTO sono già normalizzati.
  TEST SUPERATO

GALEN (`GALEN_NormalizationTest`). Per questa ontologia, già normalizzata in seguito al calcolo delle inferenze e all'esportazione tramite Protégé, è stato realizzato un test case:

- `isGCIInNormalForm_test()`: controlla se tutti gli assiomi dell'ontologia GALEN sono già normalizzati.
  TEST SUPERATO

GENE Ontology (`GO_NormalizationTest`). Per questa ontologia, già normalizzata in seguito al calcolo delle inferenze e all'esportazione tramite Protégé, è stato realizzato un test case:

- `isGCIInNormalForm_test()`: controlla se tutti gli assiomi dell'ontologia GO sono già normalizzati.
  TEST SUPERATO

6.2.2 Testing della saturazione: `OntologySaturator_Test`

La classe di test contiene il seguente metodo globale che viene utilizzato da tutte le sottoclassi (una per ogni ontologia) per testare la saturazione:

```
void saturationTest(OWLOntology ontology, boolean normalized,
    boolean concurrentMode) {
    if (normalized) {
        ontology = new ELPPOntologyNormalizer().normalize(ontology);
    }

    OntologyAccessManager ontologyAccessManager = new
        OntologyAccessManager(ontology);
    ontologyAccessManager.registerRule(new
        ToldSuperclassesInferenceRule());
    ontologyAccessManager.registerRule(new
        IntersectionSuperclassesInferenceRule());
    ontologyAccessManager.registerRule(new
        SubclassRoleExpansionInferenceRule());
    ontologyAccessManager.registerRule(new
        SuperclassRoleExpansionInferenceRule());
    ontologyAccessManager.registerRule(new
        BottomSuperclassRoleExpansionInferenceRule());
    ontologyAccessManager.registerRule(new
        NominalChainExpansionInferenceRule());

    OntologySaturator saturator = new
        OntologySaturator(ontologyAccessManager, new
        ContextAccessManager(), concurrentMode);
    Set<OWLSubClassOfAxiom> conclusions = saturator.saturate();

    OWLReasonerFactory reasonerFactory = new ElkReasonerFactory();
    OWLReasoner elk = reasonerFactory.createReasoner(ontology);
    elk.precomputeInferences(InferenceType.CLASS_HIERARCHY);

    for (OWLSubClassOfAxiom axiom : conclusions) {
        assertEquals(EXPECTED_RESULT, elk.isEntailed(axiom));
    }
}
```

In sintesi, il metodo prende un'ontologia `ontology` in input, la normalizza se `normalized=true` ed esegue la saturazione sull'ontologia, in parallelo se `concurrent=true`. Il testing viene superato con successo se ogni assioma dell'insieme di conclusioni in output è *entailed* da ELK. Seguono le classi di test, ognuna delle quali contiene quattro test case:

- `[OntologyName]_saturate()`: esegue la saturazione sull'ontologia raw, senza normalizzare e senza eseguire in parallelo
- `[OntologyName]_saturate_c()`: esegue la saturazione sull'ontologia raw, senza normalizzare, eseguendo in parallelo
- `[OntologyName]_saturate_n()`: esegue la saturazione sull'ontologia normalizzata, senza eseguire in parallelo
- `[OntologyName]_saturate_nc()`: esegue la saturazione sull'ontologia normalizzata, con esecuzione in parallelo

Italian Food (`ItalianFood_SaturationTest`).

- `ItalianFood_saturate()`: TEST SUPERATO
- `ItalianFood_saturate_c()`: TEST SUPERATO
- `ItalianFood_saturate_n()`: TEST SUPERATO
- `ItalianFood_saturate_nc()`: TEST SUPERATO

SCTO (`SCTO_SaturationTest`).

- `SCTO_saturate()`: TEST SUPERATO
- `SCTO_saturate_c()`: TEST SUPERATO
- `SCTO_saturate_n()`: TEST SUPERATO
- `SCTO_saturate_nc()`: TEST SUPERATO

GALEN (`GALEN_SaturationTest`).

- `GALEN_saturate()`: TEST SUPERATO
- `GALEN_saturate_c()`: TEST SUPERATO
- `GALEN_saturate_n()`: TEST SUPERATO
- `GALEN_saturate_nc()`: TEST SUPERATO

GENE Ontology (`GO_SaturationTest`).

- `GO_saturate()`: TEST SUPERATO
- `GO_saturate_c()`: TEST SUPERATO

- `GO_saturate_n()`: TEST SUPERATO
- `GO_saturate_nc()`: TEST SUPERATO

6.2.3 Testing della tassonomia: `ELPPTaxonomyBuilder_Test`

La classe di test contiene il seguente metodo globale che viene utilizzato da tutte le sottoclassi (una per ogni ontologia) per testare la costruzione della tassonomia:

```
void taxonomyTest(OWLOntology ontology, boolean normalized, boolean
    concurrentMode) {
    if (normalized) {
        ontology = new ELPPOntologyNormalizer().normalize(ontology);
    }

    ELPPReasoner elppReasoner = new ELPPReasoner(ontology, false,
        concurrentMode);
    elppReasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY);

    OWLReasoner elk = new
        ElkReasonerFactory().createReasoner(ontology);
    elk.precomputeInferences(InferenceType.CLASS_HIERARCHY);

    ontology.classesInSignature().forEach(owlClass -> {
        assertEquals(elppReasoner.getEquivalentClasses(owlClass),
            elk.getEquivalentClasses(owlClass));
        assertEquals(elppReasoner.getSubClasses(owlClass),
            elk.getSubClasses(owlClass));
        assertEquals(elppReasoner.getSubClasses(owlClass,
            InferenceDepth.DIRECT), elk.getSubClasses(owlClass,
            InferenceDepth.DIRECT));
        assertEquals(elppReasoner.getSuperClasses(owlClass),
            elk.getSuperClasses(owlClass));
        assertEquals(elppReasoner.getSuperClasses(owlClass,
            InferenceDepth.DIRECT), elk.getSuperClasses(owlClass,
            InferenceDepth.DIRECT));
    });
}
```

In sintesi, il metodo prende un'ontologia `ontology` in input, la normalizza se `normalized=true` ed esegue il metodo `precomputeInferences(CLASS_HIERARCHY)` sia sul nostro ragionatore - in parallelo se `concurrent=true` - sia su ELK. Il testing viene superato con successo se le `equivalentClasses`, `subClasses`, `directSubClasses`, `superClasses` e `directSuperClasses` coincidono per entrambi i reasoner. Seguono le classi di test, ognuna delle quali contiene due test case:

- `[OntologyName]_buildTaxonomy_n()`: esegue il metodo `taxonomyTest()` sull'ontologia normalizzata, senza eseguire la costruzione della tassonomia in parallelo
- `[OntologyName]_buildTaxonomy_nc()`: esegue il metodo `taxonomyTest()` sull'ontologia normalizzata, con costruzione della tassonomia in parallelo

Italian Food (`ItalianFood_TaxonomyBuildingTest`).

- `ItalianFood_buildTaxonomy_n()`: TEST SUPERATO
- `ItalianFood_buildTaxonomy_nc()`: TEST SUPERATO

SCTO (`SCTO_TaxonomyBuildingTest`).

- `SCTO_buildTaxonomy_n()`: TEST SUPERATO
- `SCTO_buildTaxonomy_nc()`: TEST SUPERATO

GALEN (`GALEN_TaxonomyBuildingTest`).

- `GALEN_buildTaxonomy_n()`: TEST SUPERATO
- `GALEN_buildTaxonomy_nc()`: TEST SUPERATO

GENE Ontology (`GO_TaxonomyBuildingTest`).

- `GO_buildTaxonomy_n()`: TEST SUPERATO
- `GO_buildTaxonomy_nc()`: TEST SUPERATO

6.3 Analisi delle prestazioni

La valutazione della performance del ragionatore - package `src/test/java/-performance` - è stata realizzata mediante l'unica test suite `Performance_Test`, che include un test case per ogni ontologia. Ogni test case invoca il seguente metodo, che normalizza l'ontologia `ontology` in input e:

1. Calcola il tempo di esecuzione del metodo `precomputeInferences()` di `ELPPReasoner`, con le fasi di saturazione e tassonomia non eseguite in parallelo (media su `ITERATIONS` iterazioni).
2. Calcola il tempo di esecuzione del metodo `precomputeInferences()` di `ELPPReasoner`, con le fasi di saturazione e tassonomia eseguite in parallelo (media su `ITERATIONS` iterazioni)
3. Calcola il tempo di esecuzione del metodo `precomputeInferences()` di `ELK` (media su `ITERATIONS` iterazioni)

4. Calcola il tempo di esecuzione del metodo `precomputeInferences()` di `HermiT` (media su `ITERATIONS` iterazioni)

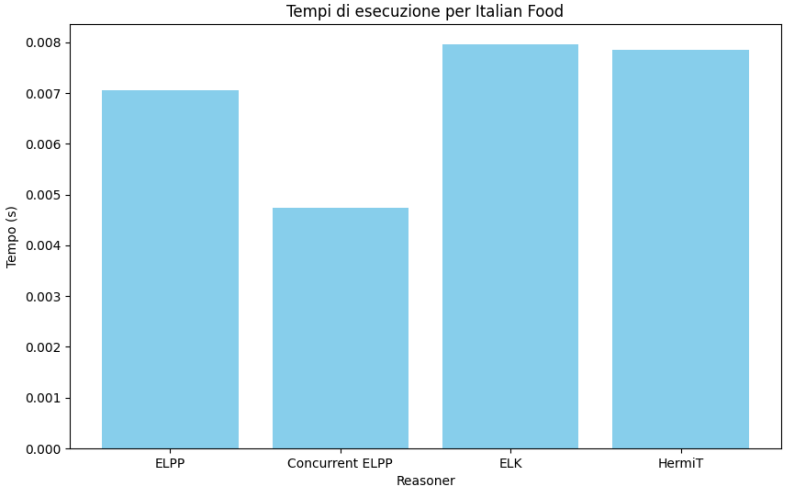
```
void performanceTest(OWLOntology ontology) throws IOException {
    ontology = new ELPPOntologyNormalizer().normalize(ontology);

    double time;
    double t0;

    // ELPPReasoner performance
    time = 0;
    for (int i = 0; i < ITERATIONS; i++) {
        t0 = System.nanoTime();
        new ELPPReasoner(ontology, false,
            false).precomputeInferences(InferenceType.CLASS_HIERARCHY);
        time += ((System.nanoTime() - t0) / 1_000_000_000);
        System.out.println("ELPPReasoner#"+i+" total elapsed time: "
            + time);
    }
    writer.append("          [ELPP] Time: " + (time / ITERATIONS) +
        "s\n");

    // Concurrent ELPPReasoner performance
    time = 0;
    for (int i = 0; i < ITERATIONS; i++) {
        t0 = System.nanoTime();
        new ELPPReasoner(ontology, true,
            true).precomputeInferences(InferenceType.CLASS_HIERARCHY);
        time += ((System.nanoTime() - t0) / 1_000_000_000);
        System.out.println("CCELPPReasoner#"+i+" total elapsed time:
            " + time);
    }
    writer.append(" [Concurrent ELPP] Time: " + (time / ITERATIONS)
        + "s\n");

    // ELK performance
    time = 0;
    for (int i = 0; i < ITERATIONS; i++) {
        t0 = System.nanoTime();
        new ElkReasonerFactory().createReasoner(ontology)
            .precomputeInferences(InferenceType.CLASS_HIERARCHY);
        time += ((System.nanoTime() - t0) / 1_000_000_000);
        System.out.println("ELK#"+i+" total elapsed time: " + time);
    }
    writer.append("           [ELK] Time: " + (time / ITERATIONS) +
        "s\n");
```

```
        // HermiT performance
        time = 0;
        for (int i = 0; i < ITERATIONS; i++) {
            t0 = System.nanoTime();
            new ReasonerFactory().createReasoner(ontology)
                .precomputeInferences(InferenceType.CLASS_HIERARCHY);
            time += ((System.nanoTime() - t0) / 1_000_000_000);
            System.out.println("HermiT#"+i+" total elapsed time: " +
                time);
        }
        writer.append("         [HermiT] Time: " + (time / ITERATIONS) +
             "s\n");
    }
```

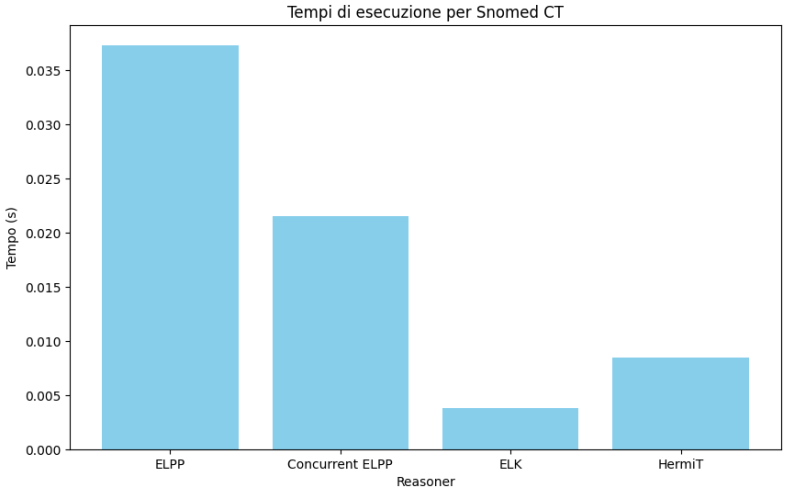
Tutti i test sono stati eseguiti su una macchina con processore Intel® Core™ i5-12500H (18M Cache, fino a 4.50 GHz) - avente 4 Performance-core e 8 Efficient-core, con cui è possibile eseguire fino a *16 thread* contemporaneamente - e 16 GB RAM. Per migliorare l'efficienza, i tempi su `ELPPReasoner` non includono la normalizzazione dell'ontologia: questo viene fatto all'inizio del metodo. I tempi sono stati valutati su un numero di iterazioni pari a `ITERATIONS = 10`, di cui è stata calcolata la media. Dopo aver ottenuto i tempi, sono stati prodotti dei grafici con uno script Python, mediante l'ausilio della libreria `matplotlib`.

Nelle seguenti immagini, con `ELPP` si intende il nostro reasoner senza parallelizzazione, mentre `Concurrent ELPP` è la versione del ragionatore che sfrutta la parallelizzazione dei moduli `saturation` e `taxonomy`.

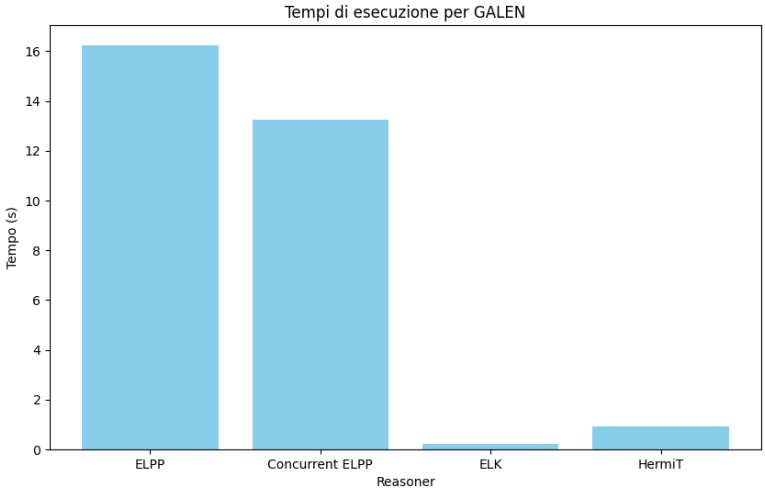
6.3.1 Prestazioni su Italian Food



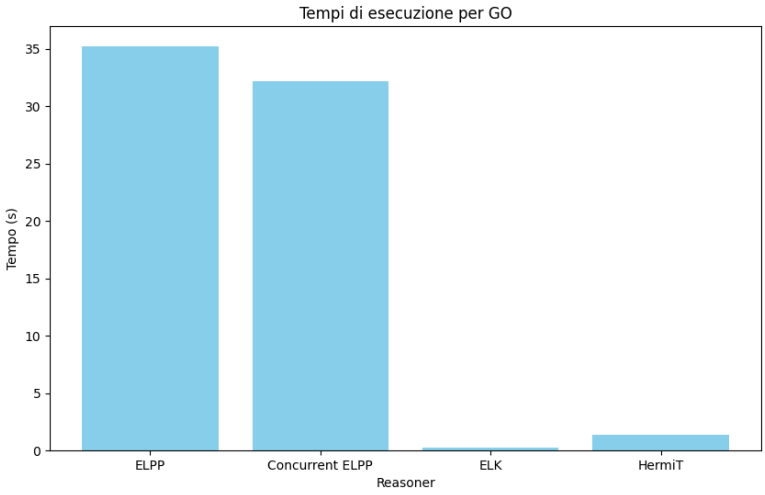
6.3.2 Prestazioni su SCTO



6.3.3 Prestazioni su GALEN



6.3.4 Prestazioni su GENE Ontology



6.4 Commenti finali

Non sorprende il performance gap tra `ELPPReasoner` e i potenti `ELK` e `HermiT` su ontologie grandi come GALEN e GENE Ontology. Sull'ontologia Italian Food, creata specificamente il nostro ragionatore, `ELPPReasoner` si impone sugli altri, probabilmente a causa dell'overhead causato dai tentativi di normalizzazione da parte degli altri reasoner. Questo non vale per SCTO che, nonostante le sue piccole dimensioni, avvalora l'utilità dei due potenti reasoner utilizzati in letteratura.

Degno di nota è il confronto tra `ELPP` e `Concurrent ELPP`:

- Su Italian Food, la versione concorrente richiede 0,0047s, mentre la versione non concorrente richiede 0,0071s. Essendo numeri molto piccoli, sarebbe poco utile calcolare il vantaggio in termini percentuali.
- Su SCTO, la versione concorrente richiede 0,0215s, mentre la versione non concorrente richiede 0,0373s. Essendo numeri molto piccoli, sarebbe poco utile calcolare il vantaggio in termini percentuali.
- Su GALEN, la versione concorrente richiede 13,234s, mentre la versione non concorrente richiede 16,237s, con un vantaggio in termini percentuali dell'18,49%.
- Su GENE Ontology, la versione concorrente richiede 32,18s, mentre la versione non concorrente richiede 35,21s, con un vantaggio in termini percentuali dell'8,605%.

Riferimenti bibliografici

[1] Y. Kazakov, M. Krötzsch, and F. Simančík, "Concurrent classification of $\mathcal{EL}$ ontologies," in *Proceedings of the Tenth International Semantic Web Conference (ISWC)*. Bonn, Germany: Springer, 2011, pp. 305–320.

[2] ——, "ELK Reasoner: Architecture and Evaluation," in *Proceedings of the International Workshop on Description Logics (DL)*, Ulm, Germany, 2012.

[3] F. Baader, S. Brandt, and C. Lutz, "Pushing the $\mathcal{EL}$ envelope," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*. Edinburgh, Scotland, UK: Morgan Kaufmann, 2005, pp. 364–369.

[4] ——, "LTCS-Report: Pushing the $\mathcal{EL}$ Envelope," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*. Edinburgh, Scotland, UK: Morgan Kaufmann, 2005, p. t.

[5] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 Reasoner," in *Journal of Automated Reasoning 53*. Springer, 2014, pp. 245–269.