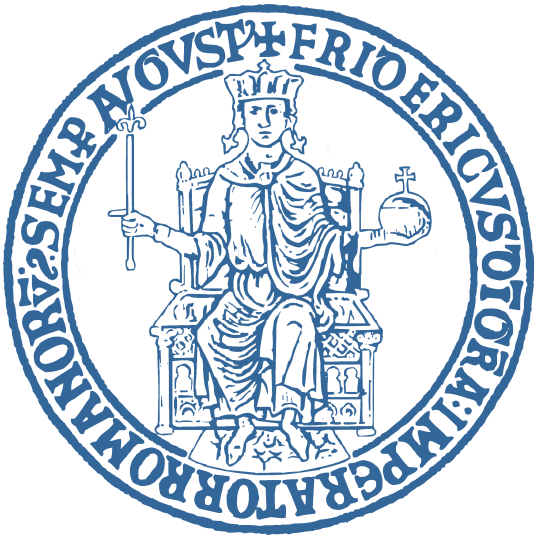


Università degli Studi di Napoli Federico II

Classificazione concorrente di ontologie $\mathcal{EL}$++

2023/2024

Professore	*Studente*
Luigi Sauro	Vincenzo Tramo

Indice

Introduzione

La seguente relazione ha lo scopo di presentare l'architettura di un reasoner che implementa una procedura di classificazione di una TBox. In particolare, data una TBox $\mathcal{T}$, il reasoner computa la gerarchia di sussunzione (*subsumption hierarchy*) di tutti concept names che occorrono in $\mathcal{T}$ rispetto a $\mathcal{T}$. Per ogni coppia A e B di concept names che occorrono in $\mathcal{T}$, il reasoner controlla se A è sussunto da B rispetto a $\mathcal{T}$ e se B è sussunto da A rispetto a $\mathcal{T}$ utilizzando un approccio basato sulle conseguenze (*consequence-based reasoning*).

Le ontologie che il reasoner prende in input sono espresse in una versione ridotta della logica polinomiale $\mathcal{EL}$++ (profilo OWL 2 EL), così come descritta in [1]. Nello specifico, la logica usata dal reasoner include solo il top concept, il bottom concept, la restrizione esistenziale, l'intersezione e i nominals. Le regole di inferenza sono quelle presentate in [1], in particolare il reasoner implementa solo le regole `CR1`, `CR2`, `CR3`, `CR4`, `CR5` e `CR6`.

Il reasoner è implementato in Java e usa le OWL API versione 5 per implementare le interfacce `OWLReasonerFactory` e `OWLReasoner`. Nello specifico, i metodi più rilevanti implementati sono i seguenti:

- `public boolean isEntailed(OWLAxiom axiom);`
- `public NodeSet<OWLClass> getSubClasses(OWLClassExpression owlClass, boolean direct);`
- `public NodeSet<OWLClass> getSuperClasses(OWLClassExpression owlClass, boolean direct);`
- `public Node<OWLClass> getEquivalentClasses(OWLClassExpression owlClassExpression);`

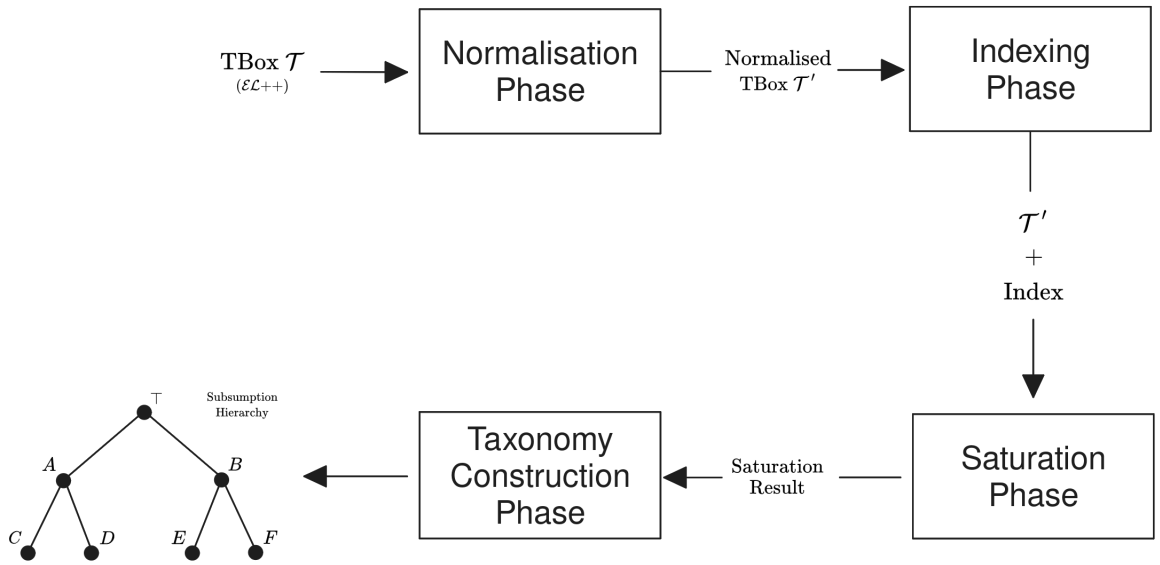
I metodi appena elencati sono possibili grazie al metodo:

- `public void precomputeInferences(InferenceType... inferenceTypes);`

dove `inferenceTypes` può essere soltanto `CLASS_HIERARCHY`. Tale metodo si occupa di computare la gerarchia di sussunzione dei concept names.

Una caratteristica fondamentale del Reasoner è la sua capacità di sfruttare più processori/cores per eseguire il processo di saturazione di una TBox $\mathcal{T}$. L'idea principale che rende tutto questo possibile è l'assegnamento degli assiomi in *contesti* che possono essere processati in maniera indipendente da più thread. L'implementazione del reasoner tiene basso l'overhead introdotto dalla gestione delle computazioni concorrenti utilizzando strutture dati lock-free e operazioni 'compare-and-swap'. L'idea alla base del reasoner concorrente è descritta in [2] ed è anche l'idea fondamentale alla base del reasoner ELK [3].

Per computare la gerarchia di sussunzione delle classi, il reasoner prima di tutto normalizza la TBox $\mathcal{T}$ in input. Chiamiamo questa fase *normalisation phase*.
Successivamente, il reasoner costruisce degli indici sugli assiomi di input per applicare più velocemente le regole di inferenza aumentando notevolmente le prestazioni del reasoner. Chiamiamo questa fase *indexing phase*.
Una volta costruito l'indice, il reasoner applica esaustivamente tutte le regole di inferenza (`CR1-CR6`) sugli assiomi di input e sugli assiomi derivati. Chiamiamo questa fase *saturation phase*.
Infine, usando il risultato della saturazione, il reasoner costruisce la subsumption hierarchy dei concept names. Chiamiamo questa ultima fase *taxonomy construction phase*.



https://github.com/vtramo/concurrent-el-reasoner

1 Normalisation Phase

L'algoritmo di sussunzione implementato dal reasoner funziona sulle TBox espresse in una versione ridotta della logica $\mathcal{EL}^{++}$. Inoltre, le TBox devono essere normalizzate. In questo capitolo verrà data una breve definizione di che cosa si intende per TBox normalizzata e verrà presentata la procedura di normalizzazione e i vari componenti coinvolti in questo processo.

1.1 Definizione di TBox in forma normale (o normalizzata)

L'implementazione dell'algoritmo di sussunzione può essere semplificata se si restringe l'attenzione soltanto su assiomi rappresentati in forma normale.
Una $\mathcal{EL}^{++}$ TBox $\mathcal{T}$ è in *forma normale* (o *normalizzata*) se contiene soltanto GCIs (General Concept Inclusions) dalla seguente forma:

$$A_1 \sqsubseteq B, \quad A_1 \sqcap A_2 \sqsubseteq B, \quad A_1 \sqsubseteq \exists r.A_2, \quad \text{oppure} \quad \exists r.A_1 \sqsubseteq B,$$

dove A_1, A_2 sono:

- concept names, oppure
- il top concept $\top$, oppure
- un nominal $\{a\}$

che appaiono in $\mathcal{T}$, mentre D può anche essere il bottom concept $\bot$.
Una qualsiasi TBox $\mathcal{T}$ in $\mathcal{EL}^{++}$ può essere convertita in una TBox $\mathcal{T}'$ in forma normale.

1.2 Regole di normalizzazione

Le regole di normalizzazione utilizzate dal reasoner solo le seguenti:

$$\begin{array}{lrcllrcl}
\texttt{NF0} & \hat{D} & \sqsubseteq & \hat{E} & \longrightarrow \hat{D} \sqsubseteq A, & A & \sqsubseteq & \hat{E} \\
\texttt{NF1}_r & C \sqcap \hat{D} & \sqsubseteq & B & \longrightarrow \hat{D} \sqsubseteq A, & C \sqcap A & \sqsubseteq & \hat{E} \\
\texttt{NF1}_l & \hat{D} \sqcap C & \sqsubseteq & B & \longrightarrow \hat{D} \sqsubseteq A, & A \sqcap C & \sqsubseteq & B \\
\texttt{NF2} & \exists r.\hat{D} & \sqsubseteq & B & \longrightarrow \hat{D} \sqsubseteq A, & \exists r.A & \sqsubseteq & B \\
\texttt{NF3} & B & \sqsubseteq & \exists\, r.\hat{D} & \longrightarrow A \sqsubseteq \hat{D}, & B & \sqsubseteq & \exists r.A \\
\texttt{NF4} & B & \sqsubseteq & D \sqcap E & \longrightarrow B \sqsubseteq D, & B & \sqsubseteq & E
\end{array}$$

dove:

- C, D, E denotano concetti $\mathcal{EL}^{++}$ arbitrari,
- $\hat{D}, \hat{E}$ denotano concetti $\mathcal{EL}^{++}$ che non sono né concept names né $\top$,
- B è un concept name,
- A è un *nuovo* concept name

1.3 Algoritmo di normalizzazione di una $\mathcal{EL}^{++}$ TBox

L'Algoritmo 1 riceve in input una TBox $\mathcal{T}$, applica esaustivamente le regole di normalizzazione presentate in 1.2 e restituisce una TBox $\mathcal{T}'$ in forma normale.

Algorithm 1: normalise(input): normalizzazione TBox $\mathcal{T}$ di input

```
   Data: input: TBox T
   Output: TBox normalizzata T'
 1 T' ← ∅
 2 abnormalAxioms ← T
 3 while abnormalAxioms ≠ ∅ do
 4 |   axiom ← abnormalAxioms.removeFirst()
 5 |   if axiom is not abnormal then
 6 |   |   T'.add(axiom)
 7 |   else
 8 |   |   normalisationRule ← determineNormalisationRule(axiom)
 9 |   |   axiomsNormalisationResult ← normalisationRule.apply(axiom)
10 |   |   abnormalAxioms.addAll(axiomsNormalisationResult)
11 return T'
```

L'Algoritmo 1 inizializza una coda di assiomi anormali (non in forma normale) inserendo in essa tutti gli assiomi della TBox $\mathcal{T}$ in input. Nel ciclo while, ad ogni iterazione, l'algoritmo rimuove dalla coda un assioma. Se l'assioma risulta essere in forma standard, allora questo viene aggiunto nella TBox normalizzata $\mathcal{T}'$. Altrimenti, viene determinata la regola di normalizzazione da applicare sull'assioma. Il risultato dell'applicazione di una regola di normalizzazione è un insieme di assiomi. Ogni applicazione della regola produce al massimo due nuovi assiomi. Il risultato della normalizzazione viene successivamente aggiunto nella coda di assiomi anormali. L'algoritmo si arresta quando non ci sono più assiomi anormali da processare.

Determinare la regola di normalizzazione da applicare sull'assioma anormale e applicare effettivamente la regola di normalizzazione sono due operazioni che possono essere fatte in tempo costante.

È facile dimostrare che una TBox $\mathcal{T}$ generica può essere trasformata in una TBox $\mathcal{T}'$ normalizzata eseguendo un numero lineare di applicazioni delle regole di normalizzazione. Inoltre, può anche essere dimostrato che la dimensione della TBox $\mathcal{T}'$ risultante è lineare nella dimensione di $\mathcal{T}$.

L'Algoritmo 1 può essere eseguito in parallelo.

1.4 Il package `normalisation`

Il package `normalisation` contiene le classi che:

- implementano le regole di normalizzazione presentate in 1.2,
- implementano l'Algoritmo 1

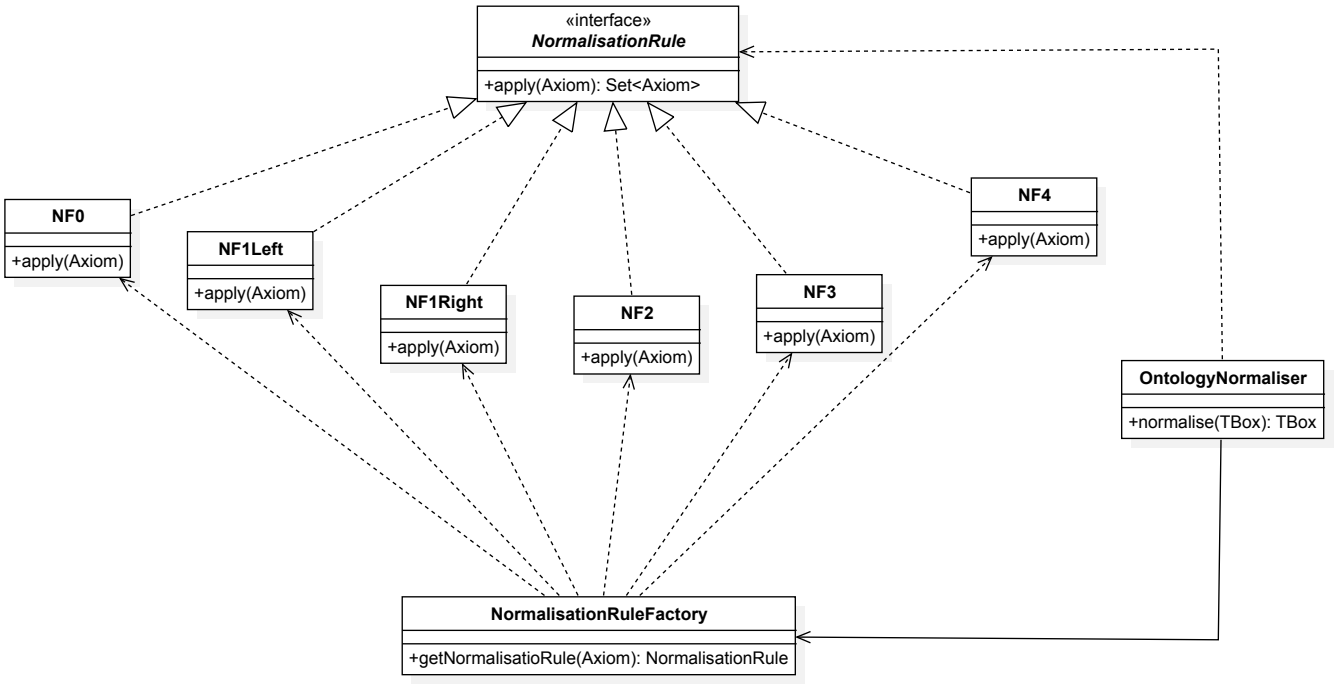


Figura 1: Classi del package `normalisation`

Il package `normalisation` contiene le classi mostrate in Figura 1.

- `NormalisationRule`: è una interfaccia che espone un metodo `apply(Axiom): Set<Axiom>`. Dato un assioma anormale, applica la regola e restituisce un insieme di assiomi. L'interfaccia è contenuta nel sottopackage `rules`;
- Package `rules`: contiene le implementazioni delle regole di normalizzazione (implementazioni interfaccia `NormalisationRule`);
- `OntologyNormaliser`: è la principale classe che, insieme alle altre classi del package `normalisation`, implementa l'Algoritmo 1. Data una TBox $\mathcal{T}$ computa e restituisce la corrispondente TBox $\mathcal{T}'$ normalizzata tramite il metodo `normalise(TBox): TBox`;
- `NormalisationRuleFactory`: è una classe che separa la logica di determinazione della regola di normalizzazione da utilizzare dalla logica di applicazione della regola di normalizzazione di un assioma. In questo modo, come mostrato in Figura 1, la classe `OntologyNormaliser` dipenderà solamente da questa classe e dall'interfaccia `NormalisationRule`.

2 Indexing Phase

La fase di indicizzazione degli assiomi di input è fondalmentale per evitare una ricerca lineare quando si cercano le premesse per applicare le regole di inferenza mostrate in [1]. La fase di indicizzazione itera sugli assiomi di input (normalizzati) per costruire degli indici. In questo capitolo, verranno esaminate le regole di inferenza `CR1-CR6` per individuare quali indici costruire per velocizzare la ricerca delle premesse e assumeremo che la TBox di input a questa fase sia già normalizzata. Infine, verrà mostrato il package `indexing` che implementa queste idee.

2.1 Indice per la regola `CR1`

La regola `CR1` è la seguente:

$$\frac{C \sqsubseteq C'}{C \sqsubseteq D} : C' \sqsubseteq D \in \mathcal{T}$$

Con $C' \sqsubseteq D \in \mathcal{T}$ intendiamo dire che l'assioma è di input (è un assioma della TBox di input). La premessa $C \sqsubseteq C'$ può essere un assioma derivato o un assioma di input. Quando bisogna processare un assioma dalla forma $C \sqsubseteq C'$, per verificare se la regola `CR1` può essere applicata è necessario individuare almeno una superclasse D di C' tale che $C' \sqsubseteq D \in \mathcal{T}$. Per velocizzare questa ricerca, costruiamo il seguente indice che chiamiamo `toldSups`:

$$A.\texttt{toldSups} = \{B \mid A \sqsubseteq B \in \mathcal{T}\}$$

Attraverso questo indice, risalire alle superclassi di C' tali che $C' \sqsubseteq D \in \mathcal{T}$ è banale: basta usare come chiave proprio C' per ottenere l'insieme delle sue superclassi e costruire gli assiomi derivati per ciascuna superclasse individuata.

2.2 Indice per la regola `CR2`

La regola `CR2` è la seguente:

$$\frac{C \sqsubseteq C_1 \quad C \sqsubseteq C_2}{C \sqsubseteq D} : C_1 \sqcap C_2 \sqsubseteq D \in \mathcal{T}$$

Per verificare se la regola di completamento `CR2` può essere applicata quando si incontra un assioma dalla forma $C \sqsubseteq C_1$ bisogna innanzitutto verificare se C_1 appare in un assioma di input dalla forma $C_1 \sqcap C_2 \sqsubseteq D \in \mathcal{T}$. Per eseguire questa ricerca in tempo costante è sufficiente costruire il seguente indice che chiamiamo `leftIntersections`:

$$A.\texttt{leftIntersections} = \{\langle B, C\rangle \mid A \sqcap B \sqsubseteq C \in \mathcal{T}\}$$

Si noti che questo indice può essere implementato come un key-value multimap da B a C. Nel processare $C \sqsubseteq C_1$, se $C_1.\texttt{leftIntersections} = \emptyset$ sicuramente la regola `CR2` non può essere applicata. Se invece risulta che $C_1.\texttt{leftIntersections} \neq \emptyset$, allora bisogna proseguire la ricerca per trovare la seconda premessa $C \sqsubseteq C_2$. Tale premessa può far parte degli assiomi di input o degli assiomi derivati. In questa fase di indicizzazione, lo scopo è quello di realizzare indici esclusivamente sugli assiomi di input. Nel Capitolo 3, quando verrà introdotto il concetto di *contesto*, vedremo come ulteriori indici sugli assiomi (derivati) vengono costruiti e utilizzati per velocizzare la ricerca.

2.3 Indice per la regola `CR3`

La regola `CR3` è la seguente:

$$\frac{C \sqsubseteq C'}{C \sqsubseteq \exists r.D} : C' \sqsubseteq \exists r.D \in \mathcal{T}$$

Dato l'assioma $C \sqsubseteq C'$ da processare, per velocizzare la ricerca degli assiomi dalla forma $C' \sqsubseteq \exists r.D \in \mathcal{T}$ costruiamo il seguente indice che chiamiamo `existentialsBySubclass`:

$$A.\texttt{existentialsBySubclass} = \{\langle r, B\rangle \mid A \sqsubseteq \exists r.B \in \mathcal{T}\}$$

Se $C'.\texttt{existentialsBySubclass} = \emptyset$, sicuramente non è possibile applicare la regola `CR3`. Altrimenti, è possibile farlo e basta utilizzare l'indice per costruira la conclusione (o le conclusioni).

2.4 Indice per la regola `CR4`

La regola `CR4` è la seguente:

$$\frac{C \sqsubseteq \exists r.D \quad D \sqsubseteq D'}{C \sqsubseteq E} : \exists r.D' \sqsubseteq E \in \mathcal{T}$$

In questo caso, possono presentarsi due casi:

1. Si sta processando un assioma dalla forma $C \sqsubseteq \exists r.D$
2. Si sta processando un assioma dalla forma $D \sqsubseteq D'$

Nel primo caso, vale la pena immediatamente verificare se il ruolo r appaia in un assioma di input dalla forma $\exists r.D' \sqsubseteq E \in \mathcal{T}$. A tal proposito, costruiamo un indice dal nome `superclassesByRoleFiller`:

$$c.\texttt{superclassesByRoleFiller} = \{\langle B, A\rangle \mid \exists c.B \sqsubseteq A \in \mathcal{T}\}$$

Se $r.\texttt{superclassesByRoleFiller} \neq \emptyset$, allora bisogna proseguire la ricerca consultando anche gli assiomi derivati.

Nel secondo caso, bisogna verificare che la superclasse D' appaia in un assioma di input dalla forma $\exists r.D' \sqsubseteq E \in \mathcal{T}$. Introduciamo, quindi, il seguente indice che chiamiamo `superclassesByFillerRole`:

$$B.\texttt{superclassesByFillerRole} = \{\langle r, A\rangle \mid \exists r.B \sqsubseteq A \in \mathcal{T}\}$$

Se $D'.\texttt{superclassesByFillerRole} \neq \emptyset$, allora bisogna proseguire la ricerca non solo tra gli assiomi di input ma anche tra quelli derivati.

2.5 Indici per le regole `CR5` e `CR6`

Le regole `CR5` e `CR6` sono rispettivamente:

$$\frac{C \sqsubseteq \exists r.D \quad D \sqsubseteq \bot}{C \sqsubseteq \bot}$$

e:

$$\frac{C \sqsubseteq \{a\} \quad D \sqsubseteq \{a\} \quad C \rightsquigarrow_R D}{\{C \sqsubseteq E \mid D \sqsubseteq E \text{ è un assioma già processato }\}}$$

dove $\rightsquigarrow_R$ è una relazione binaria che viene spiegata in [1].

Le regole `CR5` e `CR6` non hanno bisogno di premesse appartenenti agli assiomi di input. Dunque, non è necessario creare alcun indice sugli assiomi di input per queste regole di completamento, dato che, come già accennato, la fase di indicizzazione ha lo scopo di indicizzare esclusivamente gli assiomi di input.

2.6 Algoritmo di indicizzazione degli assiomi di input normalizzati

L'Algoritmo 2 riceve in input una TBox $\mathcal{T}$ normalizzata e restituisce in output una struttura dati contenente tutti gli indici presentati nelle sezioni precedenti.

Algorithm 2: buildIndex($\mathcal{T}$): creazione indice per la TBox $\mathcal{T}$ normalizzata

```
   Data: input: TBox T in forma normale
   Output: index
 1 toldSups ← ∅
 2 leftIntersections ← ∅
 3 existentialBySubclass ← ∅
 4 superclassesByRoleFiller ← ∅
 5 superclassesByFillerRole ← ∅
 6 for axiom ∈ T do
 7     if axiom is of type C ⊑ C' then
           // Aggiungi in toldSups
 8         continue
 9     if axiom is of type C1 ⊓ C2 ⊑ D then
           // Aggiungi in leftIntersections
10         continue
11     if axiom is of type C' ⊑ ∃r.D then
           // Aggiungi in existentialBySubclass
12         continue
13     if axiom is of type ∃r.D' ⊑ E then
           // Aggiungi in superclassesByRoleFiller
           // Aggiungi in superclassesByFillerRole
14         continue
15 return (toldSups,
16   leftIntersections,
17   existentialBySubclass,
18   superclassesByRoleFiller,
19   superclassesByFillerRole)
```

2.7 Il package `indexing`

Il package `indexing` contiene le classi che:

- rappresentano i vari indici presentati nelle sezioni precedenti
- implementano l'Algoritmo 2

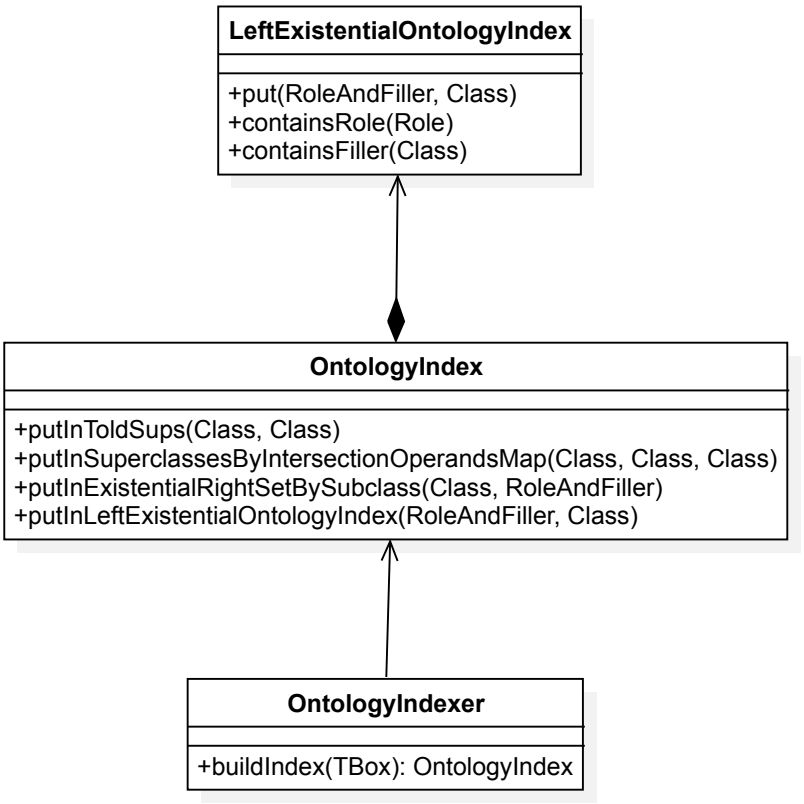


Figura 2: Classi del package `indexing`

Le principali classi del package `indexing` sono rappresentate in Figura 2:

- `LeftExistentialOntologyIndex`: è la classe che rappresenta l'indice per la regola `CR4`;
- `OntologyIndex`: è la classe che rappresenta tutti gli indici presentati nelle sezioni precedenti. Contiene anche l'indice `LeftExistentialOntologyIndex`;
- `OntologyIndex`: è la classe che implementa l'Algoritmo 2.

3 Saturation Phase

Il processo di saturazione è quel processo che applica esaustivamente le regole `CR1`, `CR2`, `CR3`, `CR4`, `CR5` e `CR6` sugli assiomi della TBox $\mathcal{T}$ di input. La TBox $\mathcal{T}^*$ ottenuta da questo processo viene chiamata TBox *saturata* e vale la seguente relazione:

- Per ogni concept name A e B (dove A e B sono concetti che appaiono in $\mathcal{T}$) si ha che:

$$\mathcal{T} \models A \sqsubseteq B \iff A \sqsubseteq B \in \mathcal{T}^*$$

Il processo di saturazione avviene in maniera concorrente. Gli assiomi vengono assegnati in una coda di assiomi da processare e i workers continuamente estraggono dalla coda un assioma da processare. Per ridurre al minimo l'overhead delle computazioni concorrenti, gli assiomi vengono assegnati in *contesti* che possono essere processati in maniera indipendente dai workers.

In questo capitolo, verrà spiegata la strategia di saturazione concorrente e le strutture dati utilizzate.

3.1 Distribuzione degli assiomi in contesti

L'idea di base è quella di suddividere gli assiomi in due collections:

1. Un insieme di assiomi processati
2. Una coda di assiomi da processare

Le due collections sono condivise tra più thread. Ogni volta che un assioma viene derivato, questo viene inserito nell'insieme degli assiomi processati. Tuttavia, affinché questa idea possa funzionare correttamente, le collections condivise devono essere thread-safe. Più nello specifico, quando un worker sta applicando le regole di inferenza su un nuovo assioma, l'insieme attuale degli assiomi processati non dovrebbe essere modificabile, altrimenti potrebbe accadere che delle conclusioni non vengano mai dedotte. Il modo più semplice per risolvere questo problema è quello di monitorare l'accesso alle collections condivise usando un lock. Tuttavia, questo non avrebbe senso perché distruggerebbe il vantaggio della concorrenza.

Esiste una *soluzione lock-free* al problema. L'idea è distribuire gli assiomi in *contesti* che possono essere processati in maniera indipendente dai workers.

La proprietà fondamentale da rispettare è la seguente:

- Ogniqualvolta che una inferenza tra differenti assiomi è possibile, gli assiomi devono avere *almeno* un contesto in comune.

Il rispetto della proprietà appena descritta implica che se su un insieme di assiomi è possibile applicare una regola di inferenza, allora questi assiomi saranno contenuti tutti insieme in almeno un contesto. Questa proprietà è fondamentale perché assicura che nessuna inferenza tra gli assiomi sia persa dato che le inferenze vengono applicate soltanto localmente all'interno dei contesti.

3.1.1 Suddivisione degli assiomi all'interno di un contesto

Ogni contesto ha una propria coda di assiomi da processare (che chiamiamo `todoAxioms`) e un insieme di assiomi processati (che chiamiamo, in generale, `processedAxioms`). Per velocizzare la ricerca tra gli assiomi processati all'interno di un contesto, l'insieme degli assiomi processati, in alcuni casi, viene partizionato ulteriormente in due o più strutture dati, in base al tipo di contesto.

3.1.2 Contesti attivi

L'algoritmo lock-free è basato sulla nozione di *contesti attivi*. Un contesto è *attivo* se la coda degli assiomi da processare `todoAxioms` non è vuota. Viene quindi mantenuta una coda globale di contesti attivi che chiamiamo `activeContexts`.

L'attivazione e la disattivazione di un contesto deve essere thread-safe e deve rispecchiare realmente lo stato della coda nel senso che:

- Un contesto C appartiene ad `activeContexts` se e soltanto se C.`todoAxioms` $\neq \emptyset$

L'attivazione di un contesto avviene quando il contesto ha la coda `todoAxioms` vuota e un worker aggiunge un assioma da processare nella coda. L'attivazione di un contesto provoca l'inserimento del contesto all'interno della coda `activeContexts`.

La disattivazione di un contesto avviene quando un worker ha appena finito di processare tutti gli assiomi

all'interno della coda `todoAxioms`. La disattivazione di un contesto deve avvenire soltanto quando la coda `todoAxioms` di quel contesto è vuota.

L'attivazione o disattivazione di un contesto può essere rappresentata con un flag booleano `isActive`.

Affinché un contesto C possa essere processato indipendentemente da un worker senza lockare l'insieme di assiomi processati C.`processedAxioms`, bisogna assicurare che il contesto C non venga inserito più di una volta nella coda `activeContexts` oppure venga inserito nella coda di contesti attivi quando un altro worker lo sta già processando.

Siccome le operazioni di attivazione e disattivazione di un contesto possono essere eseguite contemporaneamente da più workers su uno stesso contesto, sono necessari dei meccanismi di sincronizzazione.

3.1.3 Algoritmi lock-free di attivazione e disattivazione di un contesto

L'Algoritmo 3 riceve in input un contesto C, cambia il flag C.isActive da `false` a `true` e, se ha successo, aggiunge C in `activeContexts`.

Algorithm 3: `activeContexts`.activate(C)

```
   Data: input: Contesto C
1  if C.isActive.compareAndSwap(false, true) then
2      activeContexts.put(C)
```

L'Algoritmo 3 assicura che quando due workers provano a cambiare il valore di C`.isActive` da `false` a `true`, soltanto uno di loro avrà successo. Questo viene garantito utilizzando una operazione lock-free atomica `compare-and-swap` che testa e aggiorna il valore di un flag in una sola istruzione.

L'Algoritmo 4 riceve in input un contesto C e pone il flag C.isActive uguale a `false`.

Algorithm 4: `activeContexts`.deactivate(C)

```
   Data: input: Contesto C
1  C.isActive ← false
2  if C.todoAxioms ≠ ∅ then
3      activeContexts.activate(C)
4
```

Prima che un worker W_1 disattivi un contesto C con `activeContexts.deactivate(`C`)` dopo averlo processato, potrebbe accadere che, un attimo prima, un altro worker W_2 aggiunga un assioma nella coda C`.todoAxioms`. Successivamente, il worker W_1 entra in `activeContexts.deactivate(`C`)` e disattiva il contesto settando C`.isActive` a `false`. Dunque, si finisce con uno stato indesiderato: il contesto C non sarà inserito in `activeContexts` e quindi si finisce con un contesto che è attivo nel senso di avere C.`todoAxioms` $\neq \emptyset$, ma con C.`isActive` == `false`. Per risolvere questo problema, basta aggiungere un test addizionale su C.`todoAxioms` come mostrato nell'Algoritmo 4 (riga 2).

3.1.4 La funzione `getContexts(axiom)`: assegnamento dei contesti

La funzione `getContexts(axiom)` assegna ad ogni assioma un insieme di contesti tale che, ogni volta che una inferenza tra differenti assiomi è possibile, le premesse avranno almeno un contesto in comune.

La corretta implementazione di questa funzione è essenziale affinché il processo di saturazione avvenga completamente. Inoltre, per sfruttare al meglio utilizzo di più cores, i contesti risultanti non devono essere pochi.

Il reasoner assegna a ogni concept name o nominal un insieme di contesti. I vari tipo di contesti usati dal reasoner si differenziano in base alla regola di inferenza, quindi i vari tipi di contesti ne sono in totale 6 (`ContextCR1`, `ContextCR2`, `ContextCR3`, `ContextCR4`, `ContextCR5` e `ContextCR6`).

Il reasoner alloca i contesti prima di iniziare il processo di saturazione in questo modo (chiamiamo questa procedura `allocateContexts(axioms)`):

- Per ogni concept name C, istanzia i contesti `ContextCR1(`C`)`, `ContextCR2(`C`)`, `ContextCR3(`C`)`, `ContextCR4(`C`)` e `ContextCR5(`C`)`
- Per ogni nominal $\{a\}$, istanzia i contesti `ContextCR1(`$\{a\}$`)`, `ContextCR2(`$\{a\}$`)`, `ContextCR3(`$\{a\}$`)`, `ContextCR4(`$\{a\}$`)`, `ContextCR5(`$\{a\}$`)` e `ContextCR6(`$\{a\}$`)`

La funzione `getContexts(axiom)` restituisce dunque i contesti il base al tipo di assioma e l'implementazione usata dal reasoner è mostrata nell'Algoritmo 5.

Algorithm 5: getContexts(axiom)

```
Data: input: axiom in forma normale
Output: un insieme di contesti
contexts ← ∅
if axiom is of type A1 ⊑ B then
    CR1CR2CR3Contexts ← CR1CR2CR3ContextsMap.get(A1)
    CR4Contexts ← CR4ContextsMap.get(A1)
    if B is of type ⊥ then
        CR5Contexts ← CR5ContextsMap.get(A1)
        contexts.addAll(CR5Contexts)
    CR6Contexts ← CR6ContextsMap.values()
    contexts.addAll(CR1CR2CR3Contexts)
    contexts.addAll(CR4Contexts)
    contexts.addAll(CR6Contexts)
if axiom is of type A1 ⊑ ∃r.A2 then
    CR4Contexts ← CR4ContextsMap.get(A2)
    CR5Contexts ← CR5ContextsMap.get(A2)
    CR6Contexts ← CR6ContextsMap.values()
    contexts.addAll(CR4Contexts)
    contexts.addAll(CR5Contexts)
    contexts.addAll(CR6Contexts)
return contexts
```

dove A_1, A_2 sono concept names, nominal oppure il top concept $\top$ e B può anche essere il bottom concept $\bot$.

L'idea è quella di individuare un concept name (o nominal) rappresentativo per ogni completion rule e assegnare quel concetto al corrispondente contesto per quella completion rule. La funzione `getContexts(axiom)` mostrata nell'Algoritmo 5 assegna a ogni assioma i contesti che rappresentano quell'assioma in ogni completion rule.

3.1.5 Inizializzazione dei contesti

Un contesto $C(A)$ per un concept name (o nominal) A è *inizializzato* quando le seguenti due regole sono state applicate:

$$\frac{}{A \sqsubseteq A} \qquad \frac{}{A \sqsubseteq \top}$$

L'Algoritmo 6 inizializza un contesto producendo (se non già inizializzato) le conclusioni delle regole di inferenza sopra mostrate.

Algorithm 6: $C(A)$.initialize()

```
Output: {A ⊑ A, A ⊑ ⊤} oppure ∅
if not C(A).isInitialized then
    C(A).isInitialized ← true
    return {A ⊑ A, A ⊑ ⊤}
return ∅
```

3.1.6 Strutture dati per gli assiomi processati usate dai contesti

Per alcune regole di completamento, la ricerca delle premesse non si limita soltanto agli assiomi di input, ma anche agli assiomi derivati.

Per velocizzare la ricerca delle premesse all'interno di un contesto risulta conveniente partizionare l'insieme degli assiomi processati all'interno di un contesto in più strutture dati che offrono metodi di ricerca efficienti. Di seguito vengono mostrati gli indici sugli assiomi di input assegnati ai vari tipi di contesto e le loro strutture dati interne per gli assiomi processati all'interno di quel contesto:

- `ContextCR1`: il contesto per la regola `CR1` ha bisogno soltanto dell'indice `toldSups` per ricercare la seconda premessa contenuta negli assiomi di input. Gli assiomi processati dal contesto `CR1` hanno la forma $A_1 \sqsubseteq B$ (dove A_1 è il concetto che rappresentativo del contesto) e vengono salvati nel seguente insieme:

$$\texttt{ContextCR1}(A_1).\texttt{processedAxioms} = \{A_1 \sqsubseteq B \mid A_1 \sqsubseteq B \text{ è stato processato } \}$$

- `ContextCR2`: il contesto per la regola `CR2` ha bisogno dell'indice `leftIntersections` per ricercare la prima premessa appartenente agli assiomi di input. Per ogni premessa così trovata, la regola che processa il contesto `CR2` controlla se nell'insieme `CR2.processedAxioms` è contenuta la regola $C \sqsubseteq C_2$. Gli assiomi processati dal contesto `CR2` hanno la forma $A_1 \sqsubseteq B$ (dove A_1 è il concetto che rappresentativo del contesto) e vengono salvati nel seguente insieme:

$$\texttt{ContextCR2}(A_1).\texttt{processedAxioms} = \{A_1 \sqsubseteq B \mid A_1 \sqsubseteq B \text{ è stato processato } \}$$

- `ContextCR3`: il contesto per la regola `CR3` ha bisogno dell'indice `existentialsBySubclass` per ricercare la seconda premessa tra gli assiomi di input. Gli assiomi processati dal contesto `CR3` hanno la forma $A_1 \sqsubseteq B$ (dove A_1 è il concetto che rappresentativo del contesto) e vengono salvati nel seguente insieme:

$$\texttt{ContextCR3}(A_1).\texttt{processedAxioms} = \{A_1 \sqsubseteq B \mid A_1 \sqsubseteq B \text{ è stato processato } \}$$

- `ContextCR4`: il contesto per la regola `CR3` ha bisogno degli indici `superclassesByRoleFiller` e `superclassesByFillerRole` per ricercare la prima premessa tra gli assiomi di input. Gli assiomi processati in un contesto di tipo `CR4` possono assumere due forme: $A_2 \sqsubseteq \exists r.A_1$ oppure $A_1 \sqsubseteq B$ (dove A_1 è il concetto che rappresentativo del contesto). Vengono quindi mantenute due strutture dati di assiomi processati:

$$\texttt{ContextCR4}(A_1).\texttt{simpleProcessedAxioms} = \{A_1 \sqsubseteq B \mid A_1 \sqsubseteq B \text{ è stato processato}\}$$

$$\texttt{ContextCR4}(A_1).r.\texttt{subclassesByRoleProcessedAxioms} = \{A_2 \sqsubseteq \exists r.A_1 \mid A_2 \sqsubseteq \exists r.A_1 \text{ è stato processato}\}$$

- `ContextCR5`: il contesto per la regola `CR5` non ha bisogno di nessuna premessa appartenente agli assiomi di input. Gli assiomi processati in un contesto di tipo `CR5` possono assumere due forme: $A_2 \sqsubseteq \exists r.A_1$ e $A_1 \sqsubseteq \bot$ (dove A_1 è il concetto che rappresentativo del contesto). Siccome l'assioma $A_1 \sqsubseteq \bot$ può apparire soltanto una volta in un contesto di tipo `CR5`, viene mantenuta l'informazione se è stato già incontrato o meno. Gli assiomi processati vengono salvati nel seguente insieme:

$$\texttt{ContextCR5}(A_1).\texttt{processedAxioms} = \{A_2 \sqsubseteq \exists r.A_1 \mid A_2 \sqsubseteq \exists r.A_1 \text{ è stato processato}\}$$

- `ContextCR6`: il contesto `CR6` ha come concetto rappresentativo un nominal $\{a\}$. Ogni contesto di tipo `CR6` deve ricevere tutti gli assiomi derivati dalla forma $A_1 \sqsubseteq B$ e $A_1 \sqsubseteq \exists r.A_2$ indipendentemente dal tipo di contesto a cui questi assiomi appartengono. Il contesto di tipo `CR6` deve tenere traccia di tutti gli assiomi dalla forma $A_1 \sqsubseteq \{a\}$, per questo motivo viene salvata la seguente struttura dati:

$$\texttt{ContextCR6}(\{a\}).\texttt{processedIndividualSubclasses} = \{A_1 \sqsubseteq \{a\} \mid A_1 \sqsubseteq \{a\} \text{ è stato processato } \}$$

  Gli assiomi dalla forma $A_1 \sqsubseteq A_2$ vengono salvati in questo indice:

$$\texttt{ContextCR6}(\{a\}).A_1.\texttt{simpleSuperclassesBySubclassProcessedAxioms} = \{A_1 \sqsubseteq A_2 \mid A_1 \sqsubseteq A_2 \text{ è stato processato}\}$$

  Per tenere traccia delle relazioni $A_1 \rightsquigarrow_R A_2$ viene mantenuto un grafo orientato dove i nodi sono concept names o nominals processati e un arco da (A_1, A_2) esiste se e soltanto se l'assioma $A_1 \sqsubseteq \exists r.A_2$ è stato processato. Infine, per la definizione della relazione $\rightsquigarrow_R$ è necessario mantenere tutti i nominals incontrati:

$$\texttt{ContextCR6}(\{a\}).\texttt{processedIndividuals} = \{\{b\} \mid \{b\} \text{ appare in qualche assioma processato}\}$$

  Ogni contesto differente di tipo `CR6` mantiene una copia di queste strutture dati.

3.2 Algoritmo di saturazione

Il processo di saturazione eseguito dal reasoner è riportato nell'Algoritmo 7.

Algorithm 7: `saturate`($\mathcal{T}$)

```
   Data: input: T in forma normale
   Output: TBox T* saturata
 1 activeContexts ← ∅
 2 allocateContexts(T)
 3 for axiom ∈ T do
 4     for context ∈ getContexts(axiom) do
 5         if not context.isInitialized() then
 6             initialAxioms ← context.initialize()
 7             for initialAxiom ∈ initialAxioms do
 8                 for initialContext ∈ getContexts(initialAxiom) do
 9                     initialContext.addTodoAxiom(initialAxiom)
10         context.addTodoAxiom(axiom)
11         activeContexts.activate(context)
12 Loop
13     context ← activeContexts.poll()
14     if context == null then break
15     Loop
16         axiom ← activeContexts.poll()
17         if axiom == null then break
18         if context.containsProcessedAxiom(axiom) then break
19         context.addProcessedAxiom(axiom)
20         for conclusion ∈ deriveConclusions(context, axiom) do
21             for conclusionContext ∈ getContexts(conclusion) do
22                 conclusionContext.addProcessedAxiom(conclusion)
23                 activeContexts.activate(conclusionContext)
24     activeContexts.deactivate(context)
```

L'Algoritmo 7 prima alloca i contesti come descritto in Sezione 3.1.4. Successivamente, per ogni assioma in $\mathcal{T}$, ottiene i contesti associati a quell'assioma, aggiunge l'assioma alla coda `todoAxioms` del contesto e attiva il contesto. Se il contesto non è stato inizializzato, viene inizializzato e gli assiomi iniziali prodotti vengono assegnati ai corrispondenti contesti. Questa è la fase di "inizializzazione dei contesti".

Il loop principale (riga 12-24) può essere processato concorrentemente da differenti workers. Ogni worker, una volta ottenuto un contesto, può processarlo in maniera indipendente. Il worker ripetutamente consuma gli assiomi da processare del contesto. L'assioma da processare viene prima inserito negli assiomi processati del contesto e poi vengono applicate le regole di inferenza. Ogni conclusione così ottenuta viene poi assegnata ai contesti interessati. Il contesto, una volta processato, viene disattivato e l'Algoritmo 7 termina quando non esistono più contesti attivi.

La funzione `deriveConclusions` applica la giusta regola di inferenza in base al contesto e alla tipologia di assioma in input. Nelle successive sottosezioni vengono mostrati gli pseudo-codici delle regole di inferenza utilizzate dal reasoner.

3.2.1 Algoritmo regola `CR1`

Algorithm 8: `applyCR1`(context, $A_1 \sqsubseteq B$)

Data: `input`: `context` di tipo `ContextCR1` e $A_1 \sqsubseteq B$ da processare
Output: un insieme di GCIs (conclusioni)

```
1 conclusions ← ∅
2 if not context.toldSups.containsKey(B) then return ∅
3 for B ⊑ C ∈ context.toldSups.get(B) do
4     conclusions.add(A1 ⊑ C)
5 return conclusions
```

3.2.2 Algoritmo regola `CR2`

Algorithm 9: `applyCR2`(context, $A_1 \sqsubseteq B$)

Data: `input`: `context` di tipo `ContextCR2` e $A_1 \sqsubseteq B$ da processare
Output: un insieme di GCIs (conclusioni)

```
1 conclusions ← ∅
2 if not context.leftIntersections.containsKey(B) then return ∅
3 for B ⊓ C ⊑ D ∈ context.leftIntersections.get(B) do
4     if context.processedAxioms.contains(A1 ⊑ C) then
5         conclusions.add(A1 ⊑ D)
6 return conclusions
```

3.2.3 Algoritmo regola `CR3`

Algorithm 10: `applyCR3`(context, $A_1 \sqsubseteq B$)

Data: `input`: `context` di tipo `ContextCR3` e $A_1 \sqsubseteq B$ da processare
Output: un insieme di GCIs (conclusioni)

```
1 conclusions ← ∅
2 if not context.existentialsBySubclass.containsKey(B) then return ∅
3 for B ⊑ ∃r.D ∈ context.existentialsBySubclass.get(B) do
4     conclusions.add(A1 ⊑ ∃r.D)
5 return conclusions
```

3.2.4 Algoritmo regola `CR4`

Algorithm 11: `applyCR4`(context, $A_2 \sqsubseteq \exists r.A_1$)

Data: `input`: `context` di tipo `ContextCR4` e $A_2 \sqsubseteq \exists r.A_1$ da processare
Output: un insieme di GCIs (conclusioni)

```
1 conclusions ← ∅
2 if not context.superclassesByRoleFiller.containsKey(r) then return ∅
3 for ∃r.D ⊑ E ∈ context.superclassesByRoleFiller.get(r) do
4     if context.simpleProcessedAxioms.contains(A1 ⊑ D) then
5         conclusions.add(A2 ⊑ E)
6 return conclusions
```

Algorithm 12: `applyCR4`(context, $A_1 \sqsubseteq B$)

Data: `input`: `context` di tipo `ContextCR4` e $A_1 \sqsubseteq B$ da processare
Output: un insieme di GCIs (conclusioni)

```
1 conclusions ← ∅
2 if not context.superclassesByFillerRole.containsKey(B) then return ∅
3 for ∃r.B ⊑ E ∈ context.superclassesByFillerRole.get(B) do
4     if context.subclassesByRoleProcessedAxioms.contains(r) then
5         for D ⊑ ∃r.A1 ∈ context.subclassesByRoleProcessedAxioms.get(r) do
6             conclusions.add(D ⊑ E)
7 return conclusions
```

3.2.5 Algoritmo regola CR5

Algorithm 13: applyCR5(context, $A_2 \sqsubseteq \exists r.A_1$)

Data: input: context di tipo ContextCR5 e $A_1 \sqsubseteq B$ da processare
Output: un insieme di GCIs (conclusioni)

```
conclusions ← ∅
if context.containsProcessedAxiom(A1 ⊑ ⊥) then
    conclusions.add(A2 ⊑ ⊥)
return conclusions
```

Algorithm 14: applyCR5(context, $A_1 \sqsubseteq \bot$)

Data: input: context di tipo ContextCR5 e $A_1 \sqsubseteq \bot$ da processare
Output: un insieme di GCIs (conclusioni)

```
conclusions ← ∅
for A2 ⊑ ∃r.A1 context.processedAxioms do
    conclusions.add(A2 ⊑ ⊥)
return conclusions
```

3.2.6 Algoritmo regola CR6

Algorithm 15: applyCR6(context, axiom)

Data: input: context di tipo ContextCR6 e axiom da processare
Output: un insieme di GCIs (conclusioni)

```
conclusions ← ∅
if context.processedIndividualSubclasses.size() < 2 then return ∅
for C ⊑ {a} ∈ context.processedIndividualSubclasses do
    for D ⊑ {a} ∈ context.processedIndividualSubclasses do
        if C == D then continue
        if context.graph.reachability(C, D) then for
          D ⊑ E ∈ context.superclassesBySubclassProcessedAxioms.get(D) do
            conclusions.add(C ⊑ E)
return conclusions
```

3.3 Il package `saturation`

La Figura 3 sintetizza i componenti principali del package `saturation`.

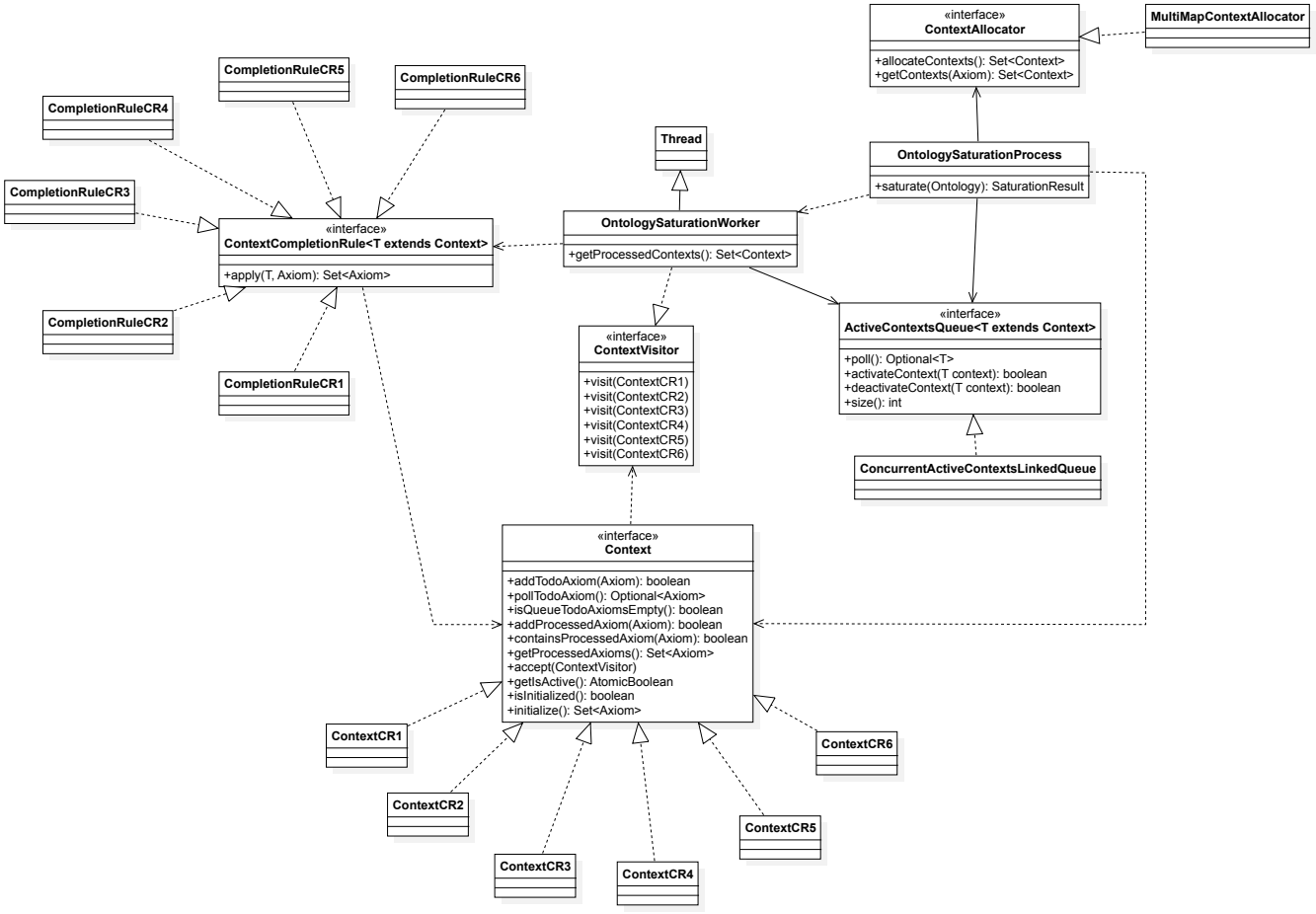


Figura 3: Classi del package `saturation`

- `Context`: è una interfaccia che rappresenta un generico contesto ed espone tutte le fondamentali operazioni viste nei paragrafi precedenti;
- `ContextCompletionRule`: è una interfaccia (una `FunctionalInterface`) che rappresenta una regola di inferenza;
- `ContextVisitor`: design pattern Visitor. Consente di separare la logica di processamento di un contesto dal contesto. Viene implementato da `OntologySaturationWorker`;
- `ActiveContextQueue`: rappresenta la coda dei contesti attivi;
- `ContextAllocator`: è una interfaccia che definisce la funzione che assegna a ogni assioma un contesto (la funzione `getContexts` vista nei paragrafi precedenti);
- `OntologySaturationWorker`: rappresenta un worker che processa i contesti attivi. Estende la classe `Thread` e implementa l'interfaccia `ContextVisitor`.

4 Taxonomy Construction Phase

Il risultato della classificazione viene rappresentato nella forma di una gerarchia di sussunzione (*subsumption hierarchy*), cioè un grafo i cui nodi sono etichettati utilizzando i concept names di $\mathcal{T}$ e un arco da un nodo A a un nodo B esiste se e soltanto se A è sussunto da B rispetto a $\mathcal{T}$.

La fase di saturazione computa la relazione di sussunzione completa transitivamente chiusa. È necessario quindi ridurre transitivamente le sussunzioni computate. L'output della classificazione è una tassonomia che contiene soltanto sussunzioni dirette tra i nodi che rappresentano classi di equivalenza di concetti atomici. La relazione di sussunzione $\sqsubseteq_{\mathcal{T}}$ è una relazione riflessiva e transitiva. È possibile rappresentare il risultato della classificazione considerando *l'ordine parziale stretto* $\sqsubset_{\mathcal{T}}$.

In questo capitolo viene mostrato l'algoritmo usato dal reasoner per costruire la subsumption hierarchy.

4.1 Algoritmo di costruzione della gerarchia di sussunzione

Il seguente algoritmo computa due mapping. Dato un concetto atomico A, computa A.`equivalentClasses` e A.`directSuperClasses`. Il primo insieme contiene tutti i concetti che sono equivalenti ad A, incluso A stesso. Il secondo insieme contiene esattamente un elemento da ogni classe di equivalenza di superconcetti diretti di A. L'Algoritmo 16 riceve in input la TBox $\mathcal{T}^*$ saturata e un mapping `superConcepts` che associa a ogni concetto

Algorithm 16: `transitiveReduction`($\mathcal{T}^*$, `superConcepts`)

```
   Data: input: TBox T* saturata e mapping superConcepts
   Output: equivalentConcepts, directSuperConcepts
 1 equivalentConcepts ← ∅
 2 directSuperConcepts ← ∅
 3 for A ∈ sig(T*) do
 4     for C ∈ superConcepts.get(A) do
 5         if A ∈ superConcepts.get(C) then
 6             equivalentConcepts.add(C)
 7         else
 8             isDirect ← true
 9             for B ∈ directSuperConcepts.get(B) do
10                 if C ∈ superConcepts.get(B) then
11                     isDirect ← false
12                     break
13                 if B ∈ superConcepts.get(C) then
14                     directSuperConceptsA ← directSuperConcepts.get(A)
15                     directSuperConceptsA.remove(B)
16             if isDirect then
17                 directSuperConceptsA ← directSuperConcepts.get(A)
18                 directSuperConceptsA.add(C)
19 return (equivalentConcepts, directSuperConcepts)
```

$A \in \mathtt{sig}(\mathcal{T}^*)$[1] le sue superclassi. Il mapping `superConcepts` può essere facilmente computato a partire dagli assiomi appartenenti a $\mathcal{T}^*$.

Una volta computato i due mappings `equivalentConcepts` e `directSuperConcepts` per ogni A, la costruzione della tassonomia è semplice. Viene introdotto un nodo per ogni classe distinta di concetti equivalenti, e vengono connessi i nodi in accordo alla relazione diretta con i superconcetti. Alla fine, vengono inseriti il top e il bottom node nelle giuste posizioni, anche se $\top$ o $\bot$ non appaiono in $\mathtt{sig}(\mathcal{T}^*)$.

[1] Con $\mathtt{sig}(\mathcal{T})$ si intende dire l'insieme di tutti i concept names e role names che appaiono in $\mathcal{T}$ (la *signature* di $\mathcal{T}$). Indichiamo con A un concept name e con r un role name. Con $A \in \mathtt{sig}(\mathcal{T})$ intendiamo dire tutti i concept names che appaiono in $\mathcal{T}$.

5 Testing

In questo capitolo verranno documentati i casi di test eseguiti sul reasoner per verificare il suo corretto funzionamento.

5.1 Testing normalisation phase

In questa sezione verranno testate le regole di normalizzazione. I casi di test sono presenti in `src/test/java` e sono contenuti nella classe `OntologyNormalisationTest`.

5.1.1 Test case 1 (SUCCESS)

Sia $\mathcal{T}$ la seguente TBox:

$$\mathcal{T} := \{\exists r.A \sqcap \exists r.\exists s.A \sqsubseteq A \sqcap B\}$$

L'applicazione esaustiva delle regole di normalizzazione mostrate in Sezione 1.2 sulla TBox $\mathcal{T}$ produce la seguente TBox $\mathcal{T}_{\text{norm}}$ normalizzata:

$$\mathcal{T}_{\text{norm}} = \{\exists r.A \sqsubseteq B_1, B_1 \sqcap B_2 \sqsubseteq B_0, \exists s.A \sqsubseteq B_3, \exists r.B_3 \sqsubseteq B_2, B_0 \sqsubseteq A, B_0 \sqsubseteq B\}$$

Il seguente test case verifica se il reasoner produce il giusto risultato $\mathcal{T}_{\texttt{norm}}$ data la TBox $\mathcal{T}$ in input:

```
@DisplayName("Ontology Normalisation Test Suite")
public class OntologyNormalisationTest {
    @Nested
    @DisplayName("Normalisation Test Case 1")
    class NormalisationTestCase1 {

        OWLOntology abnormalOntology;

        @BeforeEach
        void createEmptyOntology() {
            // Prima di iniziare il test case, crea una ontologia vuota
            abnormalOntology = OntologyUtils.createEmptyOWLOntology();
        }

        @Test
        @DisplayName("Normalisation Test Case 1")
        void normalisationTest() {
            // Costruisci T
            OWLSubClassOfAxiom abnormalSubClassOfAxiom = buildAbnormalSubClassOfAxiomTestCase1();
            abnormalOntology.add(abnormalSubClassOfAxiom);

            // Normalizza T applicando le regole di normalizzazione esaustivamente su T
            OntologyNormaliser ontologyNormaliser = new OntologyNormaliser(abnormalOntology);
            OWLOntology normalisedOntology = ontologyNormaliser.createNormalisedOntology();

            // Verifica che il risultato sia esattamente uguale a T_norm
            assertThat(normalisedOntology.getTBoxAxioms(Imports.INCLUDED), hasSize(6));
        }

        // Metodo che costruisce TBox anormale T
        private OWLSubClassOfAxiom buildAbnormalSubClassOfAxiomTestCase1() { ... }

    }

    ...

}
```

5.1.2 Test case 2 (SUCCESS)

Sia $\mathcal{T}$ la seguente TBox che rappresenta una università:

$$\begin{aligned}\mathcal{T} := \{&\texttt{Course} \sqcap \texttt{Person} \sqsubseteq \bot,\\ &\texttt{UGC} \sqsubseteq \texttt{Course},\\ &\texttt{PGC} \sqsubseteq \texttt{Course},\\ &\texttt{Teacher} \equiv \texttt{Person} \sqsubseteq \exists \mathit{teaches}.\texttt{Course},\\ &\exists \mathit{teaches}.\top \sqsubseteq \texttt{Person},\\ &\texttt{Student} \equiv \texttt{Person} \sqcap \exists \mathit{attends}.\texttt{Course},\\ &\exists \mathit{attends}.\top \sqsubseteq \texttt{Person}\}\end{aligned}$$

L'applicazione esaustiva delle regole di normalizzazione mostrate in Sezione 1.2 sulla TBox $\mathcal{T}$ produce la seguente TBox $\mathcal{T}_{\text{norm}}$ normalizzata:

$$\begin{aligned}\mathcal{T}_{\text{norm}} := \{&\exists \mathit{attends}.\texttt{Course} \sqsubseteq B_0,\\ &\exists \mathit{teaches}.\texttt{Course} \sqsubseteq B_1,\\ &\texttt{Student} \sqsubseteq \exists \mathit{attends}.\texttt{Course},\\ &\texttt{Teacher} \sqsubseteq \exists \mathit{teaches}.\texttt{Course},\\ &B_1\texttt{Person} \sqsubseteq \texttt{Teacher},\\ &\exists \mathit{attends}.\top \sqsubseteq \texttt{Person},\\ &B_0 \sqcap \texttt{Person} \sqsubseteq \texttt{Student},\\ &\texttt{PGC} \sqsubseteq \texttt{Course},\\ &\texttt{Course} \sqcap \texttt{Person} \sqsubseteq \bot,\\ &\texttt{Teacher} \sqsubseteq \texttt{Person},\\ &\texttt{UGC} \sqsubseteq \texttt{Course},\\ &\exists \mathit{teaches}.\top \sqsubseteq \texttt{Person},\\ &\texttt{Student} \sqsubseteq \texttt{Person}\}\end{aligned}$$

Il seguente test case verifica se il reasoner produce il giusto risultato $\mathcal{T}_{\text{norm}}$ data la TBox $\mathcal{T}$ in input:

```
@DisplayName("Ontology Normalisation Test Suite")
public class OntologyNormalisationTest {

    ...

    @Nested
    @DisplayName("Normalisation Test Case 2 (Simple University)")
    class NormalisationTestCase2 {

        @Test
        @DisplayName("Normalise simple ontology university")
        void university() {
            // Costruisci TBox \mathcal{T} anormale
            OWLOntology ontologyUniversity = OntologyUtils.createSimpleOntologyUniversity();
            OntologyNormaliser ontologyNormaliser = new OntologyNormaliser(ontologyUniversity);

            // Costruisci expected TBox \mathcal{T}_\text{norm}
            Set<OWLSubClassOfAxiom> expectedTBox = buildExpectedTBox();

            // Normalizza \mathcal{T} applicando le regole di normalizzazione esaustivamente su \mathcal{T}
            OWLOntology normalisedOntology = ontologyNormaliser.createNormalisedOntology();

            // Verifica che il risultato sia esattamente uguale a \mathcal{T}_norm
            Set<OWLAxiom> outputTBox = normalisedOntology.getTBoxAxioms(Imports.INCLUDED);
            assertThat(outputTBox, hasSize(equalTo(13)));
            assertThat(outputTBox, is(equalTo(expectedTBox)));
        }
```

```
        // Metodo per costruire expected TBox T_norm
        Set<OWLSubClassOfAxiom> buildExpectedTBox() { ... }

    }

    ...

}
```

5.2 Testing saturation phase & taxonomy construction phase

In questa sezione verranno testati i processi di saturazione e di costruzione della gerarchia di sussunzione. Per costruire l'output aspettato, verrà utilizzato come oracolo un noto reasoner chiamato HermiT [?]. I casi di test sono presenti in `src/test/java` e sono contenuti nella classe `OntologySaturationTest`.

5.2.1 Test case 1 (SUCCESS)

Sia $\mathcal{T}$ la seguente TBox:

$$\begin{aligned}\mathcal{T} := \{ & A \sqsubseteq B \sqcap \exists r.C, \\ & B \sqcap \exists r.B \sqsubseteq C \sqcap D, \\ & C \sqsubseteq \exists r.A \sqcap B \\ & \exists r.\exists r.B \sqcap D \sqsubseteq \exists r.(A \sqcap B), \\ & \{io\} \sqsubseteq C, \\ & D \sqsubseteq \exists r.(A \sqcap C)\}\end{aligned}$$

Denotiamo con $\mathcal{T}^*$ la TBox che si ottiene applicando prima il processo di normalizzazione e poi il processo di saturazione. Utilizziamo HermiT come oracolo per fare asserzioni sull'output prodotto dal reasoner da testare. Il seguente test case verifica se il reasoner produce il giusto risultato $\mathcal{T}^*$ e la giusta gerarchia di sussunzione data la TBox $\mathcal{T}$ in input:

```
public class OntologySaturationTest {

    @Test
    @DisplayName("Test Case 1 - Ontology A")
    public void testCase1() {
        // Costruisci TBox T
        OWLOntology ontologyA = OntologyUtils.createOWLOntologyA();

        // Normalizza TBox T ottenendo T_norm
        OntologyNormaliser ontologyNormaliser = new OntologyNormaliser(ontologyA);
        OWLOntology normalisedOntologyA = ontologyNormaliser.createNormalisedOntology();

        // Crea indice su T_norm
        OntologyIndexer ontologyAIndexer = new OntologyIndexer(normalisedOntologyA);
        OntologyIndex ontologyAIndex = ontologyAIndexer.buildIndex();

        // Esegui processo di saturazione su T_norm ottenendo T*
        OntologySaturationProcess ontologyASaturationProcess =
            new OntologySaturationProcess(normalisedOntologyA, ontologyAIndex);
        SaturationResult saturation = ontologyASaturationProcess.saturate();

        // Costruisci gerarchia di sussunzione
        SubsumptionHierarchyProcess subsumptionHierarchyProcess = new SubsumptionHierarchyProcess();
        SubsumptionHierarchy subsumptionHierarchy =
            subsumptionHierarchyProcess.buildHierarchy(saturation);

        // Costruisci oracolo (reasoner HermiT)
        ReasonerFactory reasonerFactory = new ReasonerFactory();
        OWLReasoner hermiT = reasonerFactory.createReasoner(normalisedOntologyA);
```

```
        // Computa expected output usando HermiT
        hermiT.precomputeInferences(InferenceType.CLASS_HIERARCHY);

        // Ottieni sig(T*)
        Set<OWLClass> signature = normalisedOntologyA.getClassesInSignature(Imports.INCLUDED);

        // Tutti gli assiomi in T* sono sussunti anche da HermiT
        saturation.getAllAxioms().forEach(axiom ->
            assertThat(hermiT.isEntailed(axiom),
                is(equalTo(true))));

        // Stesse classi di equivalenza di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getEquivalentClasses(owlClass),
                is(equalTo(hermiT.getEquivalentClasses(owlClass)))));

        // Stesse sottoclassi dirette di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getDirectSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass, InferenceDepth.DIRECT)))));

        // Stesse superclassi dirette di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getDirectSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass, InferenceDepth.DIRECT)))));

        // Stesse superclassi di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass)))));

        // Stesse sottoclassi di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass)))));
    }

    ...

}
```

5.2.2 Test case 2 (SUCCESS)

Sia $\mathcal{T}$ la seguente TBox:

$$\begin{aligned}\mathcal{T} := \{&A \sqsubseteq B \sqcap \exists r.C,\\ &B \sqcap \exists r.B \sqsubseteq C \sqcap D,\\ &C \sqsubseteq \exists r.A \sqcap B\\ &\exists r.\exists r.B \sqcap D \sqsubseteq \exists r.(A \sqcap B),\\ &\{io\} \sqsubseteq C,\\ &D \sqsubseteq \exists r.(A \sqcap C),\\ &C \sqsubseteq \{io\},\\ &A \sqsubseteq \{io\}\}\end{aligned}$$

La TBox $\mathcal{T}$ è simile al quella usata nel Test Case 1 (Sezione 5.2.1). In questa nuova TBox, vengono aggiunti due assiomi $C \sqsubseteq \{io\}$ e $A \sqsubseteq \{io\}$ per rendere possibile l'attivazione della regola `CR6`.

Denotiamo con $\mathcal{T}^*$ la TBox che si ottiene applicando prima il processo di normalizzazione e poi il processo di saturazione. Utilizziamo HermiT come oracolo per fare asserzioni sull'output prodotto dal reasoner da testare.

Il seguente test case verifica se il reasoner produce il giusto risultato $\mathcal{T}^*$ e la giusta gerarchia di sussunzione data la TBox $\mathcal{T}$ in input:

```
public class OntologySaturationTest {

    @Test
    @DisplayName("Test Case 2 - Ontology B")
    public void testCase2() {
        // Costruisci TBox T
        OWLOntology ontologyA = OntologyUtils.createOWLOntologyB();

        // Normalizza TBox T ottenendo T_norm
        OntologyNormaliser ontologyNormaliser = new OntologyNormaliser(ontologyA);
        OWLOntology normalisedOntologyA = ontologyNormaliser.createNormalisedOntology();

        // Crea indice su T_norm
        OntologyIndexer ontologyAIndexer = new OntologyIndexer(normalisedOntologyA);
        OntologyIndex ontologyAIndex = ontologyAIndexer.buildIndex();

        // Esegui processo di saturazione su T_norm ottenendo T*
        OntologySaturationProcess ontologyASaturationProcess =
            new OntologySaturationProcess(normalisedOntologyA, ontologyAIndex);
        SaturationResult saturation = ontologyASaturationProcess.saturate();

        // Costruisci gerarchia di sussunzione
        SubsumptionHierarchyProcess subsumptionHierarchyProcess = new SubsumptionHierarchyProcess();
        SubsumptionHierarchy subsumptionHierarchy =
            subsumptionHierarchyProcess.buildHierarchy(saturation);

        // Costruisci oracolo (reasoner HermiT)
        ReasonerFactory reasonerFactory = new ReasonerFactory();
        OWLReasoner hermiT = reasonerFactory.createReasoner(normalisedOntologyA);

        // Computa expected output usando HermiT
        hermiT.precomputeInferences(InferenceType.CLASS_HIERARCHY);

        // Ottieni sig(T*)
        Set<OWLClass> signature = normalisedOntologyA.getClassesInSignature(Imports.INCLUDED);

        // Tutti gli assiomi in T* sono sussunti anche da HermiT
        saturation.getAllAxioms().forEach(axiom ->
            assertThat(hermiT.isEntailed(axiom),
                is(equalTo(true))));

        // Stesse classi di equivalenza di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getEquivalentClasses(owlClass),
                is(equalTo(hermiT.getEquivalentClasses(owlClass)))));

        // Stesse sottoclassi dirette di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getDirectSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass, InferenceDepth.DIRECT)))));

        // Stesse superclassi dirette di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getDirectSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass, InferenceDepth.DIRECT)))));

        // Stesse superclassi di HermiT
        signature.forEach(owlClass ->
```

```
            assertThat(subsumptionHierarchy.getSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass))));

        // Stesse sottoclassi di HermiT
        signature.forEach(owlClass ->
            assertThat(subsumptionHierarchy.getSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass)))));
    }
```

5.3 Testing della classificazione dei concetti OWL 2 API

Il reasoner può essere utilizzato direttamente con le OWL 2 API tramite le implementazioni delle interfacce `OWLReasonerFactory` e `OWLReasoner`. In questa sezione, vengono documentati dei casi di test che, oltre a verificare il corretto funzionamento del reasoner, mostra come utilizzare il reasoner tramite le suddette interfacce. Ancora una volta, per ottenere il risultato desiderato, viene utilizzato come oracolo il reasoner HermiT. I casi di test sono presenti in `src/test/java` e sono contenuti nella classe `OWLOntologyClassHierarchyInferenceTest`.

5.3.1 Test case 1 (SUCCESS)

L'input è la seguente TBox $\mathcal{T}$:

$$\begin{aligned}\mathcal{T} := \{&A \sqsubseteq B \sqcap \exists r.C,\\ &B \sqcap \exists r.B \sqsubseteq C \sqcap D,\\ &C \sqsubseteq \exists r.A \sqcap B\\ &\exists r.\exists r.B \sqcap D \sqsubseteq \exists r.(A \sqcap B),\\ &\{io\} \sqsubseteq C,\\ &D \sqsubseteq \exists r.(A \sqcap C)\}\end{aligned}$$

Il caso di test è il seguente:

```
public class OWLOntologyClassHierarchyInferenceTest {

    @Test
    public void testOntologyA() {
        // Crea ontologia T
        OWLOntology ontologyA = OntologyUtils.createOWLOntologyA();

        // Crea reasoner da testare tramite OWL API
        OWLReasonerFactory owlReasonerFactory = new ELPPReasonerFactory();
        OWLReasoner myReasoner = owlReasonerFactory.createReasoner(ontologyA);

        // Crea reasoner HermiT (oracolo)
        OWLReasonerFactory oracleFactoryHermiT = new ReasonerFactory();
        OWLReasoner hermiT = oracleFactoryHermiT.createReasoner(ontologyA);

        // Computa la gerarchia di sussunzione dei concepts names in T
        myReasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY);
        hermiT.precomputeInferences(InferenceType.CLASS_HIERARCHY);

        // Confronta il risultato prodotto da myReasoner con quello prodotto da hermiT
        for (OWLClass owlClass: ontologyA.getClassesInSignature()) {

            // Stesse classi di equivalenza di HermiT
            assertThat(myReasoner.getEquivalentClasses(owlClass),
                is(equalTo(hermiT.getEquivalentClasses(owlClass))));

            // Stesse sottoclassi di HermiT
            assertThat(myReasoner.getSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass))));

            // Stesse sottoclassi dirette di HermiT
```

```
            assertThat(myReasoner.getSubClasses(owlClass, InferenceDepth.DIRECT),
                is(equalTo(hermiT.getSubClasses(owlClass, InferenceDepth.DIRECT))));

            // Stesse superclassi di HermiT
            assertThat(myReasoner.getSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass))));

            // Stesse superclassi dirette di HermiT
            assertThat(myReasoner.getSuperClasses(owlClass, InferenceDepth.DIRECT),
                is(equalTo(hermiT.getSuperClasses(owlClass, InferenceDepth.DIRECT))));
        }

    }

    ...

}
```

5.3.2 Test case 2 (SUCCESS)

La TBox $\mathcal{T}$ di input è la seguente:

$$\begin{aligned}\mathcal{T} := \{&A \sqsubseteq B \sqcap \exists r.C,\\ &B \sqcap \exists r.B \sqsubseteq C \sqcap D,\\ &C \sqsubseteq \exists r.A \sqcap B\\ &\exists r.\exists r.B \sqcap D \sqsubseteq \exists r.(A \sqcap B),\\ &\{io\} \sqsubseteq C,\\ &D \sqsubseteq \exists r.(A \sqcap C),\\ &C \sqsubseteq \{io\},\\ &A \sqsubseteq \{io\}\}\end{aligned}$$

Il caso di test è il seguente ed è simile a quello precedente:

```
public class OWLOntologyClassHierarchyInferenceTest {

    @Test
    public void testOntologyA() {
        // Crea ontologia T
        OWLOntology ontologyB = OntologyUtils.createOWLOntologyB();

        // Crea reasoner da testare tramite OWL API
        OWLReasonerFactory owlReasonerFactory = new ELPPReasonerFactory();
        OWLReasoner myReasoner = owlReasonerFactory.createReasoner(ontologyB);

        // Crea reasoner HermiT (oracolo)
        OWLReasonerFactory oracleFactoryHermiT = new ReasonerFactory();
        OWLReasoner hermiT = oracleFactoryHermiT.createReasoner(ontologyB);

        // Computa la gerarchia di sussunzione dei concepts names in T
        myReasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY);
        hermiT.precomputeInferences(InferenceType.CLASS_HIERARCHY);

        // Confronta il risultato prodotto da myReasoner con quello prodotto da HermiT
        for (OWLClass owlClass: ontologyB.getClassesInSignature()) {

            // Stesse classi di equivalenza di HermiT
            assertThat(myReasoner.getEquivalentClasses(owlClass),
                is(equalTo(hermiT.getEquivalentClasses(owlClass))));

            // Stesse sottoclassi di HermiT
```

```
            assertThat(myReasoner.getSubClasses(owlClass),
                is(equalTo(hermiT.getSubClasses(owlClass))));

            // Stesse sottoclassi dirette di HermiT
            assertThat(myReasoner.getSubClasses(owlClass, InferenceDepth.DIRECT),
                is(equalTo(hermiT.getSubClasses(owlClass, InferenceDepth.DIRECT))));

            // Stesse superclassi di HermiT
            assertThat(myReasoner.getSuperClasses(owlClass),
                is(equalTo(hermiT.getSuperClasses(owlClass))));

            // Stesse superclassi dirette di HermiT
            assertThat(myReasoner.getSuperClasses(owlClass, InferenceDepth.DIRECT),
                is(equalTo(hermiT.getSuperClasses(owlClass, InferenceDepth.DIRECT))));
        }

    }

    ...

}
```

5.3.3 Test case 3 SUCCESS

Sia $\mathcal{T}$ la seguente TBox:

$$\begin{aligned}\mathcal{T} := \{&\texttt{Course} \sqcap \texttt{Person} \sqsubseteq \bot,\\ &\texttt{UGC} \sqsubseteq \texttt{Course},\\ &\texttt{PGC} \sqsubseteq \texttt{Course},\\ &\texttt{Teacher} \equiv \texttt{Person} \sqsubseteq \exists \mathit{teaches}.\texttt{Course},\\ &\exists \mathit{teaches}.\top \sqsubseteq \texttt{Person},\\ &\texttt{Student} \equiv \texttt{Person} \sqcap \exists \mathit{attends}.\texttt{Course},\\ &\exists \mathit{attends}.\top \sqsubseteq \texttt{Person}\}\end{aligned}$$

Per verificare la corretta classificazione dei concetti della TBox $\mathcal{T}$ confrontiamo il reasoner con HermiT. Il test case è stato eseguito con successo e può essere trovato nella classe `OWLOntologyClassHierarchyInferenceTest.java`.

5.4 Test delle prestazioni

In questa sezione verranno mostrati i risultati di un piccolo confronto delle prestazioni del processo di saturazione del reasoner costruito rispetto a ELK [3]. I due reasoner verranno messi a confronto su una versione modificata della Galen Ontology[2] già normalizzata contenente 81.220 assiomi. Il processo di saturazione produce 3.500.000+ assiomi. L'ontologia può essere trovata in `src/test/resources/full-galen-modified.owl`. Per eseguire i test delle performance è stata utilizzata la libreria Java Microbenchmark Harness (JMH)[3]. I test delle prestazioni si trovano nel package `src/test/java/performance`. La Figura 4 mostra i risultati ottenuti.

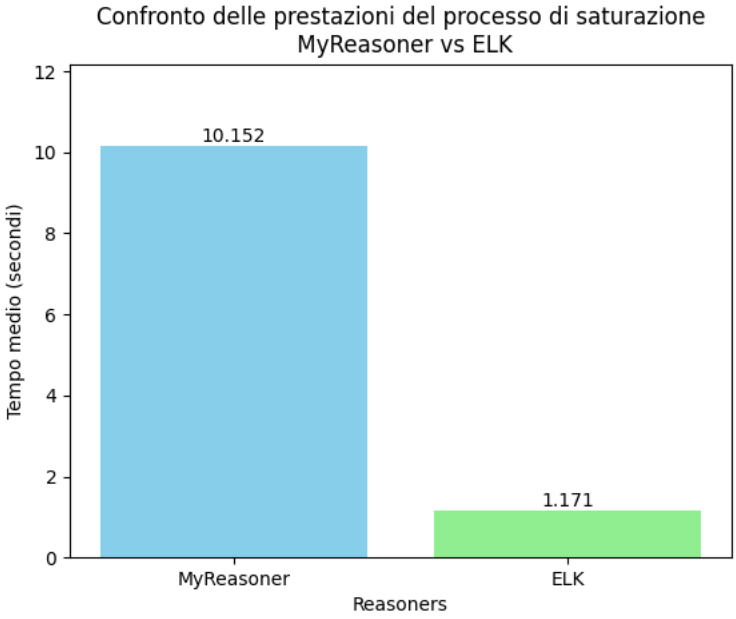


Figura 4: MyReasoner vs ELK

È bene notare che, per un confronto più equo, i test delle performance sul reasoner costruito misurano soltanto le prestazioni del processo di indicizzazione dell'ontologia insieme al processo di saturazione. Il processo di costruzione della tassonomia è inefficiente perché l'algoritmo implementato dal reasoner è sequenziale. L'Algoritmo 16 che costruisce la gerarchia di sussunzione dei concetti può essere eseguito facilmente in parallelo. Chiaramente, il reasoner ELK risulta essere decisamente più performante.

[2]https://bioportal.bioontology.org/ontologies/GALEN
[3]https://github.com/openjdk/jmh

Riferimenti

[1] Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05). pp. 364–369. Professional Book Center (2005)

[2] Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Proc. 10th Int. Semantic Web Conf. (ISWC'11). LNCS, vol. 7032, pp. 305–320. Springer (2011)

[3] Kazakov, Yevgeny et al. "ELK Reasoner: Architecture and Evaluation." International Workshop on OWL Reasoner Evaluation (2012).

[4] Glimm, B., Horrocks, I., Motik, B. et al. HermiT: An OWL 2 Reasoner. J Autom Reasoning 53, 245–269 (2014). https://doi.org/10.1007/s10817-014-9305-1

[5] Singh, Gunjan Bhatia, Sumit Mutharaju, Raghava. (2020). OWL2Bench: A Benchmark for OWL 2 Reasoners. 10.1007/978-3-030-62466-8_6.