UNIVERSITÀ
DI TORINO

Università degli Studi di Torino
Corso di Laurea Magistrale in Informatica

EDIFICA:
un'architettura estendibile e flessibile per la combinazione di concetti

Tesi di Laurea Magistrale

Relatore:
Prof. Pozzato Gian Luca

Controrelatore:
Prof. Lieto Antonio

Candidato:
Tallone Gioele
matricola 859446

Anno accademico 2022/2023

DICHIARAZIONE DI ORIGINALITÀ

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Estratto

La seguente tesi di laurea magistrale illustra la progettazione e l'implementazione di un software di intelligenza artificiale goal-directed denominato EDIFICA. Tale sistema parte da un altro sistema già esistente: GOCCIOLA, un tool nato per compiere la fusione di concetti espressi tramite Description Logic. Da quest'ultimo eredita EDIFICA eredita la caratteristica di fusione di concetti guidata dal goal e l'idea di utilizzare linguaggi formali nel compiere ragionamento, a cui aggiunge una infrastruttura a moduli estendibile e flessibile che permettono di trattare goal più complessi e in maniera più efficiente.

Indice

1 Introduzione

1.1 Sistemi di Intelligenza Artificiale

Con la seguente tesi di laurea magistrale si presenta il lavoro svolto nell'affrontare alcune criticità che si possono riscontrare nell'ambito dei sistemi di intelligenza artificiale. L'area di competenza in cui questa tesi si immerge è dunque quello dell'Intelligenza Artificiale (IA). Il campo dell'IA è quella disciplina dell'Informatica che ha come argomento quei sistemi informatici, più o meno complessi, che sono in grado di replicare, per un dato compito o task, quella che comunemente chiamiamo intelligenza. L'attenzione non è solo sul comprendere quella che noi chiamiamo "intelligenza", ma anche cercare di costruire entità intelligenti. Osserviamo che il concetto di "intelligenza" non è un concetto definito a priori e in maniera univoca, ma cambia da contesto a contesto. In sostanza, l'intelligenza sta negli occhi di chi osserva: un sistema che svolge un certo compito che comunemente richiederebbe "intelligenza" può essere considerato un sistema di intelligenza artificiale.

Nell'ambito di questa tesi, è stato studiato un sistema di intelligenza artificiale chiamato GOCCIOLA, da cui è stato derivato un nuovo sistema che è stato poi chiamato EDIFICA.

La caratteristica principale di GOCCIOLA è il fatto che il task svolto è un task molto comune per l'essere umano: il processo di combinazione della conoscenza. In particolare, GOCCIOLA organizza la conoscenza come concetti espressi tramite Description Logic e incorpora un meccanismo per andare a fondere tra loro i concetti per determinare nuova conoscenza. Il sistema EDIFICA eredita queste caratteristiche da GOCCIOLA e vi aggiunge una architettura a moduli estendibile e flessibile che permette a un utente di ridefinire certi meccanismi del sistema ma anche di andarne a introdurre di nuovi.

Per poter presentare e discutere le caratteristiche dei seguenti sistemi di intelligenza artificiale dobbiamo tenere presente che l'ambito IA è in continua evoluzione e dunque vi sono ancora molti argomenti che sono dibatti aperti. Per questo motivo, negli anni sono stati proposti diversi modelli per studiare, analizzare e progettare sistemi di intelligenza artificiale.

Ogni modello pone l'accento su certi aspetti dell'ambito IA piuttosto che su altri. Di conseguenza, vi è una vasta moltitudine di possibili modelli da utilizzare per poter definire non solo le caratteristiche principali di un sistema di intelligenza artificiale, ma anche cosa intendiamo per intelligenza in un dato contesto.

Nella seguente tesi, abbiamo preso come testo di riferimento il libro "Cognitive Design for Artificial Minds" di Antonio Lieto [4]. Dal seguente libro sono stati selezionati alcuni modelli che sono stati utilizzati per compiere analisi sui sistemi di intelligenza artificiale presi in esame all'interno di questa tesi. Iniziamo presentandone due: il primo pone l'attenzione sulla definizione di in-

telligenza artificiale, mentre il secondo pone l'attenzione sulla progettazione di un sistema di intelligenza artificiale.

Il primo modello è quello proposto da Stuart Russell e Peter Norvig [12]: tale modello nasce per fornire diverse definizioni di "intelligenza" e poter così studiare i sistemi di intelligenza artificiale in base a tali definizioni. Secondo questi due ricercatori è possibile definire il concetto di Intelligenza Artificiale in svariate maniere ed è possibile osservare che ogni definizione è composta da due fattori:

1. il primo fattore è l'ambito di applicazione del sistema di intelligenza artificiale: intelligenza a livello di processi di pensiero oppure a livello di comportamento desiderato;
2. il secondo fattore è il livello dalla quale osserviamo l'intelligenza: possiamo paragonare l'esecuzione del sistema di intelligenza artificiale con l'esecuzione umana, oppure possiamo paragonare l'esecuzione del sistema di intelligenza artificiale con un concetto ideale di intelligenza che chiamiamo razionalità.

Gategorizzazione di intelligenza secondo Russell & Norvig		
	Confronto con l'uomo	**Confronto con la razionalità**
Focus sul pensiero	Studio del pensare umanamente	Studio del pensare razionalmente
Focus sul comportamento	Studio dell'agire umanamente	Studio dell'agire razionalmente

Prendendo i due fattori definiti da Russell e Norvig e facendone tutte le possibili combinazioni otteniamo la tabella qui sopra riportata, da cui possiamo ricavare quattro possibili approcci con cui definire e studiare l'intelligenza artificiale:

- studio dell'agire umanamente: tramite questo approccio si studia l'intelligenza a livello di comportamento del sistema artificiale paragonandolo con il comportamento dell'essere umano. Potremmo definire questo approccio come un approccio di tipo funzionalista: si tenta di creare un sistema artificiale che di fatto, a livello di confronto con il comportamento umano, riproduca lo stesso comportamento, ma il come questo comportamento viene riprodotto non è di interesse;
- studio del pensare umanamente: tramite questo approccio si studia l'intelligenza a livello di processi di pensiero, paragonandoli a quei processi che intercorrono nella mente umana. Questo approccio è tipico delle scienze

cognitive e l'obiettivo è quello di studiare sistemi di intelligenza artificiale che, nei processi di pensiero, siano confrontabili con quelli dell'essere umano così da indagare come noi possiamo funzionare a livello di processi cognitivi;

- studio del pensare razionalmente: tramite questo approccio si studia l'intelligenza a livello di processi di pensiero, paragonandoli a quella che noi chiamiamo comunemente razionalità. Il pensiero razionale è uno dei modi con cui l'essere umano si mostra intelligente e sin dall'antichità è stato studiato quello che gli antichi greci chiamavano "pensiero corretto". In questo approccio per studiare la razionalità si fa ricorso alla logica;

- studio dell'agire razionalmente: tramite questo approccio si studia l'intelligenza a livello di comportamento paragonandolo con quello che possiamo definire come comportamento razionale. All'interno di questo studio troviamo quelli che vengono comunemente detti "agenti razionali".

Prima di procedere oltre, osserviamo che queste definizioni di intelligenza artificiale non sono mutuamente esclusive tra loro, ovvero più definizioni diverse possono coesistere tra loro in un sistema di intelligenza artificiale. Infatti, un sistema di intelligenza artificiale può nascere per svolgere un determinato task che va a coinvolgere più aspetti diversi ed alcuni possono essere definiti nei termini di una particolare definizione di intelligenza artificiale, mentra altri nei termini di una definizione di intelligenza artificiale alternativa.

Per esempio, si potrebbe studiare un sistema di intelligenza artificiale S in cui gli aspetti principali riguardano l'agire umanamente, ma al cui interno alcune porzioni del sistema risolvono dei sotto-task che potrebbero essere modellati come processi di ragionamento razionale. In questo esempio, il sistema S può essere agevolmente collocato tra i sistemi che studiano l'agire umanamente, ma ha al suo interno delle componenti che riguardano anche lo studio del pensare razionalmente.

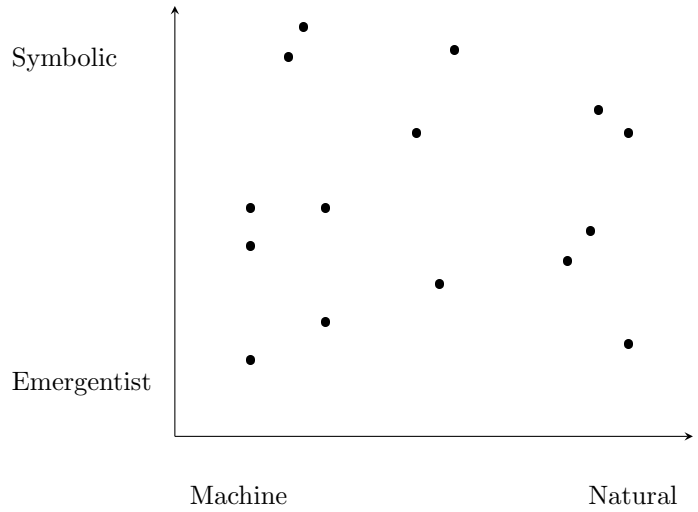
Il secondo modello che andiamo a introdurre riguarda invece il come un sistema di intelligenza artificiale è progettato e costruito. Tale modello è stato proposto nel 2014 da David Vernon [13] e consiste in uno spazio di rappresentazione 2D dei sistemi di intelligenza artificiale. Tale spazio di rappresentazione è composto da due assi:

1. sull'asse delle x si fa riferimento al livello di ispirazione del sistema artificiale. Questo aspetto riguarda lo spettro di scelte che possono essere compiute nel progettare un sistema di intelligenza artificiale. Su questo asse è presente un continuum di scelte contenute in due estremi. Al primo estremo troviamo la progettazione di tipo Natural-Oriented, ovvero ispirato in un qualche modo a un sistema naturale, sia in maniera biologica che cognitiva, mentre all'altro estremo troviamo una progettazione di tipo Machine-Oriented, ovvero orientato a costruire un sistema in cui le uniche ipotesi fatte sono quelle di come è costruita fisicamente la macchina su

cui il sistema di intelligenza artificiale sarà eseguito. In sostanza, questo primo aspetto ci descrive quanto del mondo naturale abbiamo usato come ispirazione per creare il nostro sistema artificiale. Un approccio basato sul Machine-Oriented viene anche detto Funzionalista, mentre un approccio basato sul Natural-Oriented viene anche detto Strutturalista.

2. sull'asse delle y si fa riferimento alla filosofia con cui il sistema di intelligenza artificiale viene implementato. Anche in questo caso si parla di un continuum di scelte in cui possiamo identificare due estremi. Al primo estremo troviamo l'approccio simbolico, mentre al secondo estremo troviamo l'approccio emergentista. L'approccio simbolico consiste nel definire il sistema di intelligenza artificiale come un sistema che manipola simboli e l'ipotesi cardine di questo approccio è che l'intelligenza deriva dalla manipolazione di tali simboli. L'approccio emergentista consiste invece nel definire un sistema come una serie di sistemi elementari che cooperano tra di loro e l'ipotesi cardine di questo approccio è che l'intelligenza è una proprietà che emerge dalla cooperazione di questi sistemi elementari.

Dati questi due assi, possiamo definire uno spazio in due dimensioni al cui interno ogni punto rappresenta un sistema di intelligenza artificiale. A seconda di dove questo punto è collocato nello spazio possiamo affermare quale è la sua tendenza sull'asse dell'ispirazione e sull'asse del paradigma di progettazione.



Rappresentazione in uno spazio 2D delle due componenti introdotte da Vernon. Ogni punto è un sistema di intelligenza artificiale

Anche in questo caso possiamo creare tutte le possibili combinazioni di coppie degli estremi di ciascun asse, ottenendo nuovamente quattro coppie: sistemi **emergentisti & funzionalisti**, sistemi **emergentisti & strutturalisti**, sistemi **simbolici & funzionalisti** e sistemi **simbolici & strutturalisti**.

Per quanto riguarda i sistemi emergentisti si può osservare agevolmente che questo paradigma permette sia di creare sistemi tendenti al funzionalismo che sistemi tendenti allo strutturalismo. In questo ultimo esempio, infatti, l'ipotesi di definire una serie di entità elementari che cooperano si sposa bene con molte teorie neuroscientifiche e biologiche. Per esempio, un tessuto umano può essere visto come una entità le cui proprietà complesse emergono dalla cooperazione delle sue parti: le cellule.

Passando ora ai sistemi simbolici, possiamo anche in questo caso creare sistemi tendenti al funzionalismo, ma anche sistemi tendenti allo strutturalismo. Questo secondo caso potrebbe suonare un po' strano in quanto spesso per quanto riguarda i sistemi simbolici non ci sono teorie neuroscientifiche a supporto. Infatti, non vi sono prove del fatto che l'essere umano operi a livello cerebrale manipolando dei simboli. Potremmo allora pensare che ogni sistema simbolico sia per forza funzionalista, dal momento che non va a includere elementi di ispirazione naturale al suo interno. I sistemi simbolici spesso non considerano vincoli di tipo neurologico (di natura biologica), quindi non possono essere considerati modelli strutturali dell'elaborazione cerebrale, quindi del livello "fisico" (o di hardware). Nonostante ciò, i sistemi simbolici compiono ipotesi a un livello di astrazione più alto: il livello del processamento delle informazioni. Di conseguenza, sebbene i sistemi simbolici non siano adeguati modelli strutturali di come fisicamente è costruito il cervello umano, possono essere dei validi modelli strutturali che riguardano i processi e i meccanismi della mente umana. Un esempio classico che riportiamo è quello della risoluzione di problemi di criptoaritmetica proposto da Newell & Simon [11]. Supponiamo di dover risolvere il seguente problema:

$$DONALD + GERALD = ROBERT$$

in cui ogni lettera corrisponde a una cifra decimale. Supponiamo inoltre che ci venga consigliato di associare alla lettera D il valore 5. Quasi tutti i soggetti umani trovano i valori delle rimanenti lettere in un ordine specifico: prima la lettera T, poi E e così via. Il motivo è il seguente: risolvendo le lettere in questo ordine si deve considerare solo quale valore deve assumere tale lettera, senza dover tenere conto dei valori delle altre lettere: in questo modo non si devono memorizzare più combinazioni di valori e, in caso di fallimento, dover fare un passo indietro e considerare altre combinazioni: dal punto di vista dei processi informativi, l'essere umano in certi contesti lavora come un sistema seriale con memoria a breve termine limitata. Osserviamo che in questo caso abbiamo descritto il sistema intelligente non a livello di come è costruito a un livello "basso", ma a un livello più astratto

Come detto in precedenza, i due modelli presentati si riferiscono ad aspetti diversi dell'ambito IA. Il modello di Russell e Norvig si riferisce al "cosa" fa

il sistema di intelligenza artificiale, ovvero a quale tipo di intelligenza si riferisce, mentre il modello di Vernon si riferisce al "come" il sistema di intelligenza artificiale viene effettivamente implementato. Per questo motivo, possiamo tracciare delle connessioni tra questi due modelli e possiamo osservare come i vari elementi di ciascun modello possono coesistere tra loro.

Per esempio:

- Lo studio del pensare razionalmente può essere visto come un approccio in cui si definisce il concetto di intelligenza artificiale in maniera logica e dunque si vanno a porre dei vincoli sulla progettazione del sistema: i sistemi che nascono in questo studio sono tendenzialmente sistemi simbolici;

- Lo studio del pensare umanamente può essere visto come un approccio in cui si definisce il concetto di intelligenza artificiale in termini di processi mentali e cognitivi e dunque si vanno a porre dei vincoli sulla progettazione del sistema: i sistemi che nascono in questo studio sono tendenzialmente sistemi con una forte ispirazione cognitiva;

- Gli studi dell'agire umanamente e dell'agire razionalmente possono essere visti come approcci meno vincolanti, al cui interno si possono definire diversi vincoli strutturali, siano essi biologici che cognitivi, così come si possono compiere scelte diverse sull'implementazione, potendo scegliere tra sistemi emergentisti oppure simbolici. Per esempio, nel campo degli agenti razionali è possibile definire una branca, quella dei MAS: Multi Agent Systems, ovvero sistemi in cui vi sono più agenti che agiscono in un ambiente e la collaborazione tra tali agenti può essere vista in certi casi come la creazione di un sistema complesso in maniera emergentista. All'interno di questi studi è dunque possibile avere sistemi di ispirazione più machine-oriented e con un paradigma di progettazione emergentista.

Una attenzione particolare va posta sul fatto che non esistono sistemi puramente strutturalisti o puramente funzionalisti: i sistemi che sono creati nell'ambito dello studio del pensare umanamente hanno delle forti componenti di ispirazione cognitiva, ma possono avere anche alcune componenti non ispirate cognitivamente, oppure possono avere componenti di ispirazione biologica su come il cervello umano effettivamente lavora.

1.2 Agenti Intelligenti

Con le premesse fatte nella sezione precedentemente, possiamo fornire una possibile definizione di sistema di intelligenza artificiale a cui faremo riferimento nel corso di questa tesi. Nel nostro ambito, un sistema di intelligenza artificiale S è un sistema informatico che prende in input un task t e quello che fa è andare a risolvere tale task, ovvero va a produrre un determinato risultato r. Gli aspetti importanti di questa definizione sono:

1. il task t non è un qualunque task, bensì un task in cui, convenzionalmente, si ritiene che sia necessario uno sforzo a livello di intelligenza da parte

dell'essere umano. Per esempio, l'accendere o spegnere un interruttore non può essere considerato un task "intelligente", ma il task di riconoscimento e categorizzazione di determinate specie di animali o insetti può essere considerato un task "intelligente";

2. il sistema S è dotato di una propria architettura A e come tale architettura è definita può rendere il sistema S più o meno simile a un sistema naturale e, in caso di performance simili a quelle del sistema naturale di ispirazione, può fornire un modello plausibile di come tale sistema naturale è composto.

Formalmente possiamo definire un sistema di intelligenza artificiale come segue:

$$S(A) : T \rightarrow R$$

In cui S è il sistema di intelligenza artificiale, A è la sua architettura, T è il dominio (l'input del sistema), composto da tutte le possibili istanze t di tale task ed R è il codominio (l'output del sistema), composto dai possibili risultati r che il sistema può produrre.

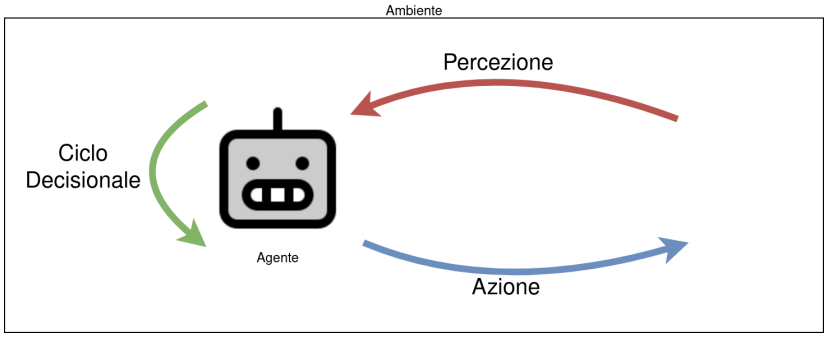
Per comprendere bene questa definizione mostriamo un esempio: supponiamo di voler creare il nostro sistema di intelligenza artificiale: SmartPath, il cui task è la pianificazione di percorsi stradali. Possiamo definire T come $Pianificazione\_Percorsi$ e all'interno troveremo tutte le possibili istanze di tale task. In R, definito come $Percorsi\_Stradali$, invece troveremo verosimilmente i possibili percorsi creati dal sistema di intelligenza artificiale. Riportiamo un esempio:

$SmartPath(A)$:	sistema di intelligenza artificiale per la pianificazione di percorsi stradali
$T : Pinanificazione\_Percorsi =$	{percorso_Roma_Napoli, percorso_Genova_Torino, ...}
$R : Percorsi\_Stradali =$	{<Roma_centro,autostrada A1, ...>, <Genova_centro, autostrada A26, ... >, ...}

La definizione A di architettura per ora la tralasciamo, ma ci tornerà utile in seguito quando parleremo dell'oggetto di questa tesi.

Ora che abbiamo definito cosa intendiamo per sistema di intelligenza artificiale, andiamo a presentare una categoria particolare di sistemi di intelligenza artificiale: gli agenti intelligenti. Un agente può essere definito come un sistema di intelligenza artificiale che è collocato in ambiente e che tiene conto di quest'ultimo nel compiere le proprie scelte. In particolare, un agente ha una serie di sensori che gli permettono di percepire l'ambiente e può, da tali percezioni,

apprendere informazioni nuove dall'ambiente e utilizzare tali informazioni per compiere le proprie scelte ed azioni sull'ambiente tramite sistemi attuatori.



Rappresentazione degli elementi di un agente

Uno stato in cui un agente intelligente può trovarsi è lo stato di impasse: dato un obiettivo da perseguire, l'agente non ha informazioni a sufficienza per risolvere un compito assegnatogli ed entra in una situazione di stallo. Come uscire da tale stallo? L'approccio tipico è quello di ottenere della nuova conoscenza e solitamente si può:

- richiedere direttamente all'utente nuova conoscenza;
- comunicare con un altro agente intelligente ed ottenere da esso nuova conoscenza;
- cercare di ottenere dall'ambiente in cui l'agente è immerso nuove informazioni tramite dei sistemi di percezione di cui l'agente è dotato.

In realtà, è possibile definire anche degli approcci alternativi per risolvere questa condizione di stallo. L'oggetto di questa tesi infatti è un software che tenta di approcciare il problema qui sopra citato in un modo diverso. Tale software si chiama GOCCIOLA, il cui nome sta per "Generating knOwledge by Concept Combination In descriptiOn Logics of typicAlity" [5].

Come suggerisce il nome, questo software si propone di generare nuova conoscenza tramite una combinazione di concetti che in partenza sono disponibili. Nell'ambito degli agenti intelligenti un tool come GOCCIOLA potrebbe essere usato da un agente intelligente per cercare di ottenere nuove informazioni a partire da quelle che già ha. Prendendo la rappresentazione di agente riportata precedentemente, potremmo collocare GOCCIOLA all'interno del ciclo decisionale (in inglese Decision Cycle) dell'agente. Il ciclo decisionale è sostanzialmente l'insieme di istruzioni che un agente compie per decidere quale azione applicare sull'ambiente. GOCCIOLA può essere considerato uno degli elementi utilizzabili dall'agente all'interno del proprio ciclo decisionale per scegliere quale è la prossima azione da applicare sull'ambiente.

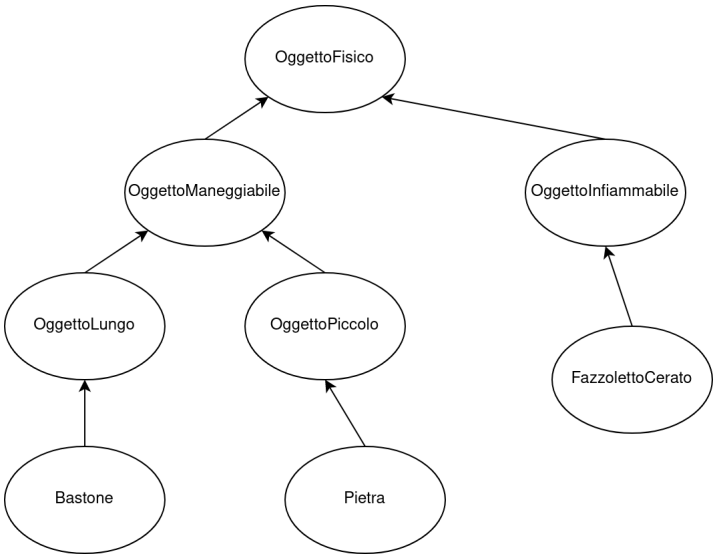
1.3 GOCCIOLA

Iniziamo la presentazione di GOCCIOLA andando a definire cosa di preciso questo tool faccia. In maniera astratta, questo software è dotato di una serie di concetti che "conosce" e il suo compito è, dato un goal, ovvero una serie di proprietà desiderate dall'utente, deve restituire il concetto che soddisfa tale goal. Andando più nello specifico, i concetti sono organizzati dal software come una rete in cui ogni concetto è legato ad altri per mezzo di relazioni di varia natura, mentre il goal è definito come una congiunzione di concetti. Ciò che rende interessante GOCCIOLA è che se il software conosce un concetto che soddisfa i requisiti imposti dal goal, semplicemente restituisce all'utente tale concetto. Se non esiste tale concetto? Il software procede a crearsene uno: prende una coppia di concetti C_1 e C_2 all'interno della propria base di conoscenza e li fonde determinando un nuovo concetto C'. Ovviamente l'aspetto interessante e critico di tutto ciò è il fatto che il software deve decidere bene quali sono i concetti da fondere e come fonderli per poter soddisfare il goal. Inoltre, quando GOCCIOLA è stato introdotto per la prima volta è stato definito anche un altro aspetto che il software si propone di fare: il concetto fusione, ove necessario, deve anche essere "non banale". Presenteremo successivamente come è stata formalizzata la qualità di "non banalità" di un concetto fusione.

Prima di procedere oltre ci chiediamo: il software GOCCIOLA può essere considerato un sistema di intelligenza artificiale? In sostanza, la nostra domanda si riduce a chiederci se il task che il software risolve è un task che richiede una certa "intelligenza". Dal nostro punto di vista, la capacità di andare a determinare nuovi concetti a partire da quelli esistenti cercando di definirli in maniera non banale può essere visto come un task intelligente. Di conseguenza, riportiamo di seguito la rappresentazione formale di GOCCIOLA come sistema di intelligenza artificiale:

$$
\begin{aligned}
GOCCIOLA(A): &\quad \text{sistema di intelligenza artificiale per combinazione di concetti} \\
T : Goal\_Utente = &\quad \{\text{OggettoLungo \& OggettoInfiammabile}, \\
&\qquad \text{Felino \& AnimaleFeroce}, ...\} \\
R : Concetti = &\quad \{\text{Bastone \& FazzolettoCerato}, \\
&\qquad \text{Tigre}, ...\}
\end{aligned}
$$

Fatte queste premesse, presentiamo un esempio per mostrare come questo software si comporta. Supponiamo di aver definito all'interno del software tre concetti: bastone, fazzoletto cerato e pietra. Riportiamo di seguito una organizzazione tassonimica di questi concetti:



Come si può intuire, data la natura tassonomica dell'organizzazione dei concetti, le frecce indicano una nozione di sottoclasse, ovvero: il concetto di fazzoletto cerato appartiene a quella famiglia di oggetti che sono infiammabili. Supponiamo ora che al software vengano dati due goal da risolvere.

Goal 1: supponiamo che venga richiesto al software di proporre un concetto che rappresenti un oggetto lungo. In tal caso, GOCCIOLA restituirà come concetto quello di bastone, in quanto appartenente alla famiglia degli oggetti lunghi.

Goal 2: supponiamo che venga richiesto al software di proporre un concetto che rappresenti un oggetto fisico che può essere maneggiato e che sia infiammabile. In tal caso, il sistema non ha un concetto in grado di soddisfare tale goal, allora procede alla fusione: vengono scelti il concetto di bastone e di fazzoletto cerato per la fusione. Il concetto ottenuto dalla fusione può essere interpretato come quel concetto che rappresenta una torcia.

Come si può evincere, il task che il software si propone di risolvere è un task che solitamente richiede all'essere umano un certo sforzo intellettuale. Per questo motivo possiamo considerare GOCCIOLA come un sistema di intelligenza artificiale. Possiamo anche osservare un'altra cosa: GOCCIOLA non è un agente intelligente in quanto un agente è immerso in un ambiente di cui tiene conto tramite sistemi percettivi e che può anche modificare tramite attuatori. Il seguente software non ha una serie di sistemi percettivi con cui percepisce l'ambiente, né tanto meno una serie di attuatori tramite cui agire sull'ambiente, ma ha solo una serie di concetti a disposizione su cui ragionare per soddisfare i goal richiesti dall'utente.

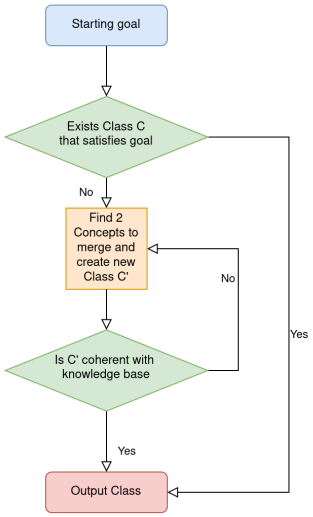
Un altro elemento interessante che verrà presentato in maniera più approfondita nei prossimi capitoli è il fatto che la conoscenza che GOCCIOLA tratta non è solo conoscenza di tipo tassonomica "rigida", ma anche "rilassata".

La relazione che intercorre tra Pietra e OggettoPiccolo è una relazione di sottoclasse "rigida": ogni pietra è un oggetto piccolo, non sono ammesse eccezioni. Ci sono casi in cui può essere utile definire una relazione di sottoclasse che sia "rilassata". Per esempio, possiamo dire che tipicamente il concetto di Uccello è sottoclasse di EssereViventeVolante, perchè sappiamo che solitamente è così, ma non sempre: i pinguini non sanno volare.

Il software GOCCIOLA permette di definire entrambi i tipi di relazione ed utilizza entrambi nella risoluzione del goal proposto dall'utente. In particolare, quando due concetti vengono fusi tra loro, si vanno a prendere le relazioni rilassate di ambo i concetti e il sistema determina quali di queste "sopravvivono" all'interno del concetto fusione e quali invece vengono "soppresse".

1.3.1 La architettura di GOCCIOLA

Sebbene GOCCIOLA non sia un agente intelligente, è comunque un software dotato di una propria architettura e dunque dotato di un proprio flusso di esecuzione. Il flusso di esecuzione prende in input un goal proposto dall'utente e produce in output il concetto che soddisfa tale goal. A questo punto possiamo andare a mostrare quale è la architettura A presente in GOCCIOLA e la riportiamo di seguito. Considereremo come architettura la coppia formata dal flusso di esecuzione e delle strutture utilizzate dal software per compiere le proprie azioni.



Flusso di esecuzione di GOCCIOLA

Possiamo osservare che il flusso di esecuzione di GOCCIOLA goda di una certa semplicità ed eleganza. Infatti, dato un goal in input, il sistema cerca tra i concetti che conosce se esiste un concetto che soddisfa il goal. Se non lo trova, cerca due concetti che conosce e tenta di fondere tali concetti per ottenere un nuovo concetto C'. Se tale concetto C' è coerente con la conoscenza del sistema, viene restituito all'utente.

Per comprendere che cosa è la coerenza di un concetto presentiamo un esempio: supponiamo che nel nostro software venga espresso il fatto che una sostanza possa essere o calda o fredda, non entrambi contemporaneamente. Supponiamo inoltre che, nel risolvere un goal, il software si ritrovi a fondere tra loro la cioccolata calda e il ghiaccio. La fusione di questi concetti sarebbe incoerente perchè staremmo creando un concetto che appartiene contemporaneamente sia a quelle sostanze che sono fredde che a quelle sostanze che sono calde.

La coerenza non è valutata solo a livello di relazioni "rigide", ma anche a livello di relazioni "rilassate". Infatti, durante la fusione per generare il nuovo concetto C' si vanno a generare quelli che sono detti **scenari**: sono istanze in cui si valuta come sarebbe C' se determinate relazioni rilassate fossero vere e altre false. Ogni scenario sostanzialmente è una combinazione di relazioni rilassate che a turno vengono mantenute oppure soppresse. Anche in questo caso possono esistere scenari incoerenti, ovvero scenari in cui affermiamo che solitamente (ovvero con buona probabilità) un concetto C' è contemporaneamente sia caldo che freddo.

Come detto, l'architettura è composta non solo dal flusso di esecuzione, ma anche dalle strutture utilizzate nel software. Possiamo collocare all'interno di questa categoria la ontologia, che è una struttura che serve per rappresentare i concetti noti al software, gli scenari, che sono la struttura con cui il sistema va a vagliare le possibili relazioni rilassate del concetto fusione. Possiamo anticipare che la ontologia è costruita a partire da linguaggi logici, mentre lo scenario può essere visualizzato come una lista, potenzialmente grande a piacere, in cui ogni elemento è una relazione rilassata che possiamo decidere di far valere o meno.

In tutto ciò, il software non solo va a scartare i concetti fusione che sono incoerenti, ma anche quelli che sono considerati banali. Gli scenari banali sono quegli scenari in cui sostanzialmente tutte o quasi tutte le relazioni rilassate dei concetti coinvolti nella fusione vengono mantenute, oppure vengono tutte o quasi tutte eliminate.

1.3.2 Analisi di GOCCIOLA

Andiamo ora brevemente a presentare, tramite i due modelli introdotti nelle sezioni precedenti, dove il software GOCCIOLA si va a collocare nel campo dell'IA.

Per quanto riguarda il modello di Russell e Norvig possiamo affermare che l'approccio principale è quello dello studio del pensare razionalmente. Questo è dovuto al fatto che il software in questione non è un agente, non è immerso in un

ambiente, ed utilizza elementi e strutture logiche per poter compiere il proprio compito: vi è una forte componente che coinvolge il concetto di razionalità. Come sottolineato in precedenza, questa categorizzazione non è rigida, dunque affermare che GOCCIOLA è un sistema che appartiene all'approccio dello studio del pensare razionalmente non implica che GOCCIOLA non possa avere elementi che possano fare parte di altre categorie. Infatti, vi sono aspetti che riguardano questo software che possono essere approcciati tramite lo studio del pensare umanamente. Per esempio, nell'ambito del sistema GOCCIOLA, quando si vanno a fondere due concetti si va ad associare a uno dei due concetti il ruolo di HEAD, ovvero di concetto che maggiormente contribuisce alla definizione del concetto fusione, mentre all'altro concetto il ruolo di MODIFIER, ovvero di concetto che fornisce un supporto al concetto principale.

Per quanto riguarda il modello di Vernon possiamo affermare, e poi lo vedremo nel dettaglio successivamente, che il software GOCCIOLA è un sistema che si basa su una implementazione simbolica e che tendenzialmente è un sistema che si colloca nel continuum tra machine-oriented e natural-oriented, tendendo al machine-oriented: vi sono sia elementi di ispirazione naturale, come il ragionamento basato su classi, che è un processo che noi umani compiamo spesso, ma anche elementi di ispirazione machine-oriented, come il fatto di scegliere la coppia di concetti da fondere in maniera casuale, invece che basarci su euristiche di come un essere umano effettivamente scelga, oppure il fatto di non includere certe restrizioni sulle strutture dati utilizzate per l'esecuzione come il fatto che l'essere umano possa memorizzare contemporaneamente un numero massimo di casistiche.

1.4 Oggetto della tesi: espansione di GOCCIOLA

Procediamo ora a presentare il lavoro che è stato svolto in questa tesi. Il software GOCCIOLA è stato presentato formalmente e implementato in codice per mostrarne il comportamento, di conseguenza questa tesi parte da un software pre-esistente. Tale tool di partenza fornisce uno schema di come avviene in maniera pratica la combinazione di concetti, ma non fornisce un'architettura definita del sistema nè una "regola di decisione" per scegliere come comporre tra loro i concetti. Inoltre, il software iniziale permette "solamente" di combinare tra loro due concetti e non di più e non considera la possibilità di indebolire il goal richiesto dall'utente nel caso in cui tale goal fosse di fatto insoddisfacibile: un utente, per esempio, potrebbe richiedere che il concetto che stiamo cercando è quello che rappresenta la classe degli individui che sono contemporaneamente esseri viventi ed esseri inanimati: il goal richiesto è assurdo! Infine, il software di partenza permette solo di esprimere congiunzioni di proprietà, mentre si potrebbero richiedere anche goal più complessi che permettano di poter trattare anche goal più complessi.

La seguente tesi si è focalizzata sul potenziare il software GOCCIOLA iniziale per poterne derivare un sistema nuovo, dotato di una struttura più definita,

di una flessibilità e una efficienza maggiore.
Il lavoro principale è stato quello di dare una struttura a moduli al software permettendo di definire in maniera flessibile vari aspetti del sistema, come la decisione di quali concetti fondere per primi, oppure definire un tetto massimo ai gruppi di concetti che possono essere presi in esame contemporaneamente dal sistema.
Dopo aver definito la struttura del nuovo sistema, un altro lavoro che è stato svolto è stato quello di definire i vari moduli per le varie fasi del flusso di esecuzione del nuovo sistema. Questi moduli sono stati implementati per due scopi:

- mostrare come la modularizzazione permette di andare a modificare in maniera abbastanza semplice il comportamento del sistema nel suo complesso;
- migliorare determinati aspetti che nella prima versione di GOCCIOLA possiamo ritenere "limitanti". Sono stati infatti definiti moduli di diversa natura, come un modulo per scegliere in maniera più accorta i concetti da fondere, un modulo che permette di esprimere e gestire goal più complessi da parte dell'utente, un modulo che compie la fusione di concetti in maniera più efficiente rispetto alla prima versione di GOCCIOLA.

Il nuovo sistema ottenuto è stato chiamato EDIFICA: ExtenDIble and FlexIble concept Combination Architecture. EDIFICA, secondo lo scema da noi riportato per descrive un sistema di intelligenza artificiale, va a differenziarsi da GOCCIOLA in due punti: sulla architettura del sistema e sul task che il sistema può trattare. Infatti, la architettura nuova sarà più complessa, ma anche più definita, mentre il task nuovo che potrà essere trattato non includerà solo i goal come congiunzione di proprietà, ma anche disgiunzioni e negazioni.

$$
\begin{aligned}
EDIFICA(A): &\quad \text{sistema di intelligenza artificiale per combinazione di concetti, dotato di una struttura più complessa} \\
T : Goal\_Utente = &\quad \{\text{Cane} \mid \text{Gatto}, \text{Felino \& AnimaleFeroce}, ...\} \\
R : Concetti = &\quad \{\text{Gatto, Tigre}, ...\}
\end{aligned}
$$

Nell'ultimo capitolo, verranno fatte una serie di analisi per confrontare il software di partenza, GOCCIOLA, con quello da noi definito, EDIFICA.

2 Definizione della architettura del sistema

2.1 Il linguaggio di programmazione utilizzato

Il software GOCCIOLA è stato scritto in linguaggio Python e la scelta di utilizzare il Python per EDIFICA è stata mantenuta principalmente per tre motivi:

- sebbene il linguaggio Python sia un linguaggio interpretato e quindi meno prestazionale in fase di esecuzione di altri linguaggi, esso è un linguaggio ad alto livello che permette di scrivere codice leggibile e risulta essere dunque un linguaggio comodo ai fini della scrittura del codice;
- è un linguaggio molto supportato, infatti esistono diversi package scaricabili e utilizzabili;
- come ogni linguaggio, fornisce diversi meccanismi semantici che possono essere utilizzati a nostro vantaggio. Nel nostro caso sono stati fondamentali i costrutti di modulo e classe.

2.2 Il linguaggio di rappresentazione della conoscenza e il meccanismo di combinazione

Tra gli elementi fondamentali che hanno caratterizzato la creazione di GOCCIOLA e che manteniamo all'interno di EDIFICA, ritroviamo il linguaggio formale con cui la conoscenza è espressa e il meccanismo con cui essa viene manipolata dal sistema.

Per la rappresentazione è stata utilizzata la logica $\mathbf{T}^{\mathrm{CL}}$. Tale logica appartiene alla famiglia delle DL (Description Logics). Le DL sono logiche che permettono di esprimere concetti e relazioni tra concetti per permettere rappresentazione della conoscenza e ragionamento sui concetti.

La logica $\mathbf{T}^{\mathrm{CL}}$ nasce dalla logica $\mathcal{ALC}$: essa è una DL in cui è possibile esprimere la intersezione di concetti le restrizioni universali e le negazioni su concetti semplici e su concetti complessi. La $\mathbf{T}^{\mathrm{CL}}$ consiste nella logica $\mathcal{ALC}$ arricchita con un formalismo logico atto a catturare della conoscenza tipica. Secondo questa scelta di linguaggio, la conoscenza del sistema viene codificata come una tripla $< R, T, A >$ in cui R è un insieme finito di proprietà rigide, T è un insieme finito di proprietà flessibili e A è un insieme di asserzioni della forma $C(a)$, ovvero l'individuo a appartiene alla classe C, oppure $R(a, b)$, ovvero l'individuo a è legato all'individuo per mezzo della relazione R.

La Description Logic $\mathbf{T}^{\mathrm{CL}}$ ha una caratteristica interessante: appartiene alla stessa classe di complessità della $\mathcal{ALC}$, ma permette di introdurre conoscenza prototipica. Tale conoscenza viene utilizzata per generare nuova conoscenza su richiesta dell'utente.

Per quanto riguarda la nostra tesi, ci siamo concentrati principalmente sulle prime due componenti della tripla, ovvero la conoscenza rigida e quella flessibile.

Riportiamo di seguito un esempio di come la conoscenza potrebbe essere organizzata:

$$Tacchi \sqsubseteq Scarpe$$

$$T(Scarpe) \sqsubseteq Comodo :: 0.9$$

$$T(Tacchi) \sqsubseteq \neg Comodo :: 0.9$$

La prima è una proprietà rigida, che afferma che ogni paio di tacchi è un paio di scarpe. La seconda e la terza sono proprietà flessibili, in cui esprimiamo un valore numerico che può essere interpretato come una probabilità di verità della proprietà.

Data questa rappresentazione, GOCCIOLA introduce altri due aspetti per implementare un meccanismo di combinazione di concetti.

Il primo aspetto consiste nel come fondere due concetti C_1 e C_2 tra loro. Presi due concetti a uno viene assegnato il ruolo di HEAD, ovvero di componente principale della fusione, mentre al secondi viene assegnato il ruolo di MODIFIER, ovvero di componente di supporto della fusione. Questo meccanismo modella una euristica che deriva dalla semantica cognitiva, in cui sostanzialmente viene eletto un elemento che è quello che porta con sé la maggior parte delle informazioni, mentre il secondo, quello di supporto, porta con sé informazioni di carattere "secondario". Questa euristica è implementata in GOCCIOLA, ma in EDIFICA è stata rimossa in quanto GOCCIOLA compie una fusione di coppie di concetti, in cui dunque è possibile definire una HEAD e un MODIFIER, mentre EDIFICA permette di fondere più concetti tra loro, dunque potrebbe non essere scontato definire quali parti della fusione sono quelle principali e quali no.

Il secondo aspetto consiste nella definizione del concetto di scenario: una volta scelti i due concetti da fondere, se ne prendono le caratteristiche tipiche e si valuta quali di queste sopravvivono nel nuovo concetto fusione. Questo meccanismo è necessario in quanto la conoscenza di tipo prototipica non è composizionale: se conosciamo il prototipo di *pet* e quello di *fish*, non siamo in grado di definire il prototipo di *petfish* componendo questi due prototipi. L'idea per implementare il concetto fusione è quella di andare a definire una serie di scenari. Uno scenario può essere visto come un mondo in cui si decide se un proprietà tipica vale o meno. Presi due concetti, si prendono tutte le proprietà tipiche e si vanno a definire tutti i possibili scenari ordinandoli per probabilità. La probabilità di uno scenario è pari al prodotto delle probabilità delle proprietà tipiche coinvolte nello scenario. Fatta questa operazione si selezionano quegli scenari che sono consistenti e si seleziona quello più probabile. Questo meccanismo è stato mantenuto in EDIFICA, ma è stato oggetto di generalizzazione, permettendo di definire gli scenari per N concetti da fondere e non solo due.

Per comprendere come la fusione avviene, supponiamo di dover combinare tra loro i concetti di Studente e di Atleta e che nella base di conoscenza abbiamo a disposizione le seguenti informazioni tipiche:

$$(1)\ T(Studente) \sqsubseteq Giovane :: 0.8$$
$$(2)\ T(Atleta) \sqsubseteq PersonaInForma :: 0.9$$

Nel fondere questi due concetti possiamo definire quattro scenari e ordinarli per probabilità:

$$((1)\ True, (2)\ True) \implies P = 0.8 * 0.9 = 0.72$$
$$((1)\ False, (2)\ True) \implies P = 0.2 * 0.9 = 0.18$$
$$((1)\ True, (2)\ False) \implies P = 0.8 * 0.1 = 0.08$$
$$((1)\ False, (2)\ False) \implies P = 0.2 * 0.1 = 0.02$$

In base a questo ordinamento, sia GOCCIOLA che EDIFICA andranno a prendere lo scenario in cui entrambe le proprietà tipiche sono vere.

Definiti gli strumenti di rappresentazione e di manipolazione della conoscenza, che sostanzialmente sono stati mantenuti in EDIFICA, presentiamo quegli elementi che sono stati introdotti in EDIFICA e che dunque lo differenziano da GOCCIOLA.

2.3 La progettazione del sistema

Scelto il linguaggio di programmazione si è passato a progettare la infrastruttura del nuovo sistema. L'infrastruttura è l'elemento fondamentale da cui sono stati poi sviluppati gli altri aspetti più specifici di EDIFICA. L'obiettivo primario è stato quello di dare una struttura al nostro sistema per poi andare ad aggiungere mano a mano gli elementi che lo caratterizzano.

Per fare ciò, sono stati analizzati diversi aspetti del sistema originale ed è stata fatta una operazione di divisione dei compiti: sono state individuate una serie di funzionalità core e per ognuna è stato definito un modulo python a sé stante. Una volta definiti tutti i moduli sono state generate delle connessioni tra di essi, in base alla struttura che volevamo dare al nuovo sistema. Questa operazione di modularizzazione ha diversi vantaggi:

1. definendo un modulo, di fatto incapsuliamo determinati processi su dati e informazioni e li concentriamo in una porzione del sistema;

2. andando a strutturare un sistema per moduli, andiamo a scomporre il sistema complesso in una serie di sotto-sistemi più semplici e ciò ci permette di poter ragionare autonomamente su ciascuna parte del sistema, senza preoccuparci sul sistema complessivo;

3. un'architettura a moduli permette di mantenere il codice più facilmente, permette di farne manutenzione, come scoprire dei bug, ma permette anche di estendere il sistema più facilmente.

La modularizzazione del sistema è stata un elemento chiave dello sviluppo di EDIFICA: si è partiti dalla scomposizione del sistema GOCCIOLA e dalla analisi delle sue caratteristiche. Una volta aver definito le caratteristiche del nuovo sistema, siano esse derivate da GOCCIOLA oppure nuove, la modularizzazione ci ha permesso di concentrarci su ciascuna caratteristica singolarmente e di migliorarla se era pre-esistente, oppure di definirla rigorosamente se era completamente nuova.

La modularizzazione è stata anche un elemento fondamentale per l'obiettivo principale di questa tesi: estendere il sistema. Infatti, è stato possibile definire una serie di **nuovi** moduli che aggiungono le funzionalità desiderate alla nuova versione di GOCCIOLA.

Infine, la modularizzazione permetterà ad utenti futuri di poter definire nuove funzionalità e di aggiungerle al sistema attuale: basta definire un nuovo modulo M e specificare come questo modulo si relaziona con quelli pre-esistenti nel sistema.

2.4 I processi core e i moduli corrispondenti

Fatte le premesse sui due ingredienti principali dello sviluppo del nuovo sistema, ovvero il linguaggio Python con i meccanismi da esso forniti e la progettazione della architettura del sistema basata su moduli, presentiamo come questi due strumenti sono stati concretamente usati per lo sviluppo di EDIFICA.

Dalla analisi delle funzionalità principali del sistema di partenza, è stato definito, per ciascuna di esse, un modulo all'interno del nuovo sistema. In particolare, i processi considerati core sono:

1. parsificazione e organizzazione delle informazioni di carattere rigido;
2. parsificazione e organizzazione delle informazioni di carattere rilassato;
3. parsificazione e organizzazione del goal definito dall'utente;
4. calcolo dei concetti candidati alla risoluzione del goal;
5. scelta di quali concetti candidati abbiano la precedenza;
6. fusione eventuale di N concetti per la risoluzione del goal.

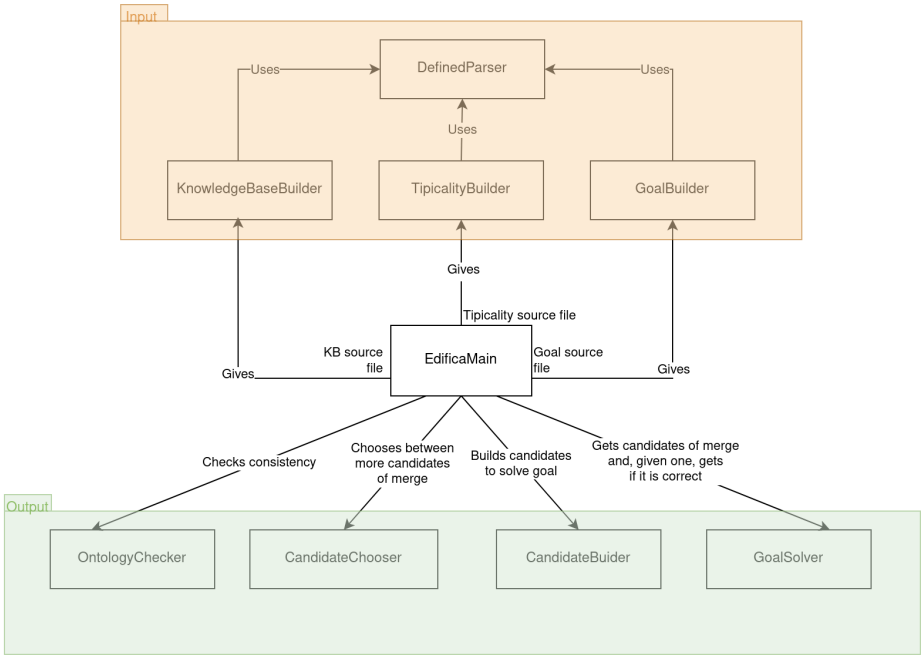
Per strutturare i processi qui sopra indicati è stato definito un modulo Python, chiamato *UtilityInterfaces*, al cui interno sono state definite delle "classi astratte". Ogni classe astratta nasce per svolgere uno specifico compito ed espone all'esterno delle proprietà e delle funzioni. Le classi astratte sono classi incomplete, ovvero classi in cui ci sono delle funzioni che sono state definite, ovvero che hanno un nome e una firma, ma che non sono state implementate, ovvero non hanno corpo. Questa scelta permette a un qualsiasi utente di definire delle classi proprie partendo da quelle astratte andando a implementare il corpo di quelle funzioni che non ce l'hanno. L'idea per questo progetto è la seguente: all'interno di UtilityInterfaces abbiamo una rappresentazione ad alto livello, tramite classi astratte, di come sono fatti i processi core del sistema.

Per ogni processo core viene definito un modulo python apposito andando, all'interno di esso, a definire una classe che estende la classe astratta di interesse. Di conseguenza, possiamo affermare che per ogni classe astratta esiste un modulo corrispondente che lo implementerà. Inoltre, un utente può accedere al modulo UtilityInterfaces e definire all'interno una nuova classe astratta che rappresenta un altro processo che vuole includere all'interno della sua versione del sistema.

Possiamo osservare i primi frutti della modularizzazione: attraverso la scomposizione dei processi possiamo focalizzarci sullo sviluppo di un determinato processo e, grazie alle classi astratte, possiamo anche avere versioni alternative di svolgimento di tale processo rendendo il sistema flessibile. Un utente X potrebbe definire un proprio modulo per il processo di risoluzione dei goal, per esempio basato su statistica, mentre un altro utente Y potrebbe definire il proprio modulo per la risoluzione dei goal, magari basato su risorse semantiche come WordNet o FrameNet. Entrambi gli utenti possono definire come vogliono diverse strategie di risoluzione, mantenendo intatta la struttura del sistema.

2.5 Organizzazione dei moduli nel sistema

Oltre al modulo contente le classi astratte è stato definito il modulo *EdificaMain*, il quale, come vedremo più avanti, svolge il compito di implementare il flusso di esecuzione del sistema. Per fare ciò, tale modulo sfrutta le classi definite in UtilityInterfaces. Riportiamo di seguito lo schema di dipendenze delle classi astratte e del modulo EdificaMain.



Come si può osservare, la maggior parte delle dipendenze partono dal modulo EdificaMain e vanno verso i moduli definiti dalle classi astratte di UtilityInterfaces. Presentiamo di seguito cosa fa ciascuna classe astratta:

1. DefinedParser: è una classe che rappresenta un parsificatore, ovvero un software che traduce un input espresso in un linguaggio L in una serie di informazioni che possono essere strutturate nel linguaggio di codifica del sistema, nel nostro caso il Python;

2. KnowledgeBaseBuilder: è una classe che rappresenta un estrattore di informazioni; in particolare è quella classe che estrae e struttura le informazioni ontologiche di natura rigida. Questo genere di informazioni rappresentano quelle che nel Capitolo 1 abbiamo definito le relazioni rigide tra i concetti. Come si può osservare, tale modulo utilizza, nel compito di traduzione delle informazioni, la classe DefinedParser;

3. TipicalityBuilder: è una classe che rappresenta un estrattore di informazioni; in particolare è quella classe che estrae e struttura le informazioni ontologiche di natura tipica. Questo genere di informazioni rappresentano quelle che nel Capitolo 1 abbiamo definito le relazioni rilassate tra i concetti. Come si può osservare, tale modulo utilizza, nel compito di traduzione delle informazioni, la classe DefinedParser;

4. GoalBuilder: è una classe che rappresenta un estrattore di informazioni; in particolare è quella classe che estrae e struttura le informazioni riguardanti il goal definito da un utente. Come si può osservare, tale modulo utilizza, nel compito di traduzione delle informazioni, la classe DefinedParser;

5. CandidateBuilder: è una classe che, dato il goal da risolvere e le informazioni strutturate in input (sia di natura rigida che di natura rilassata), va a costruire i possibili candidati a risolvere il goal. Un candidato non è altro che un insieme di concetti. Se un candidato è un insieme singoletto, ovvero è un insieme costituito da un solo concetto, diremo che è un candidato semplice (simple-candidate), altrimenti diremo che è un candidato complesso (complex-candidate);

6. CandidateChooser: è una classe che interviene, assieme alla classe GoalSolver, quando non abbiamo simple-candidates. In tal caso, abbiamo candidati composti da molti concetti l'uno e dobbiamo scegliere quali di questi candidati è il più promettente: spetta a questa classe svolgere questo compito;

7. GoalSolver: è una classe che dato un complex-candidate tenta di compiere la fusione tra i concetti all'interno del candidato per generare un nuovo concetto che soddisfi il goal di partenza;

8. OntologyChecker: il compito di questa classe è quello di fornire delle funzionalità di controllo di consistenza dei risultati parziali o finali generati durante il flusso di esecuzione di EDIFICA.

Prima di andare oltre, concentriamoci su due aspetti.

Stessa classe astratta, ma diversi moduli: le tre classi dedicate alla costruzione di informazioni date in input, ovvero KnowledgeBaseBuilder, TipicalityBuilder e GoalBuilder, utilizzano tutte e tre la classe DefinedParser, ma questa è una classe astratta: ciò non significa che i moduli che implementano KnowledgeBaseBuilder, TipicalityBuilder e GoalBuilder debbano usare tutti la stessa implementazione di DefinedParser, anzi, ogni implementazione può usare una implementazione diversa di DefinedParser. Ecco che allora possiamo avere una implementazione di KnowledgeBaseBuilder che utilizza una implementazione di DefinedParser in cui la sintassi di parsificazione è la Manchester Syntax, mentre possiamo avere una implementazione di GoalBuilder che utilizza una implementazione di DefinedParser in cui la sintassi di parsificazione è la Turtle Syntax. In questo senso, tutti e tre i moduli utilizzano la stessa classe astratta, ma possono utilizzarne implementazioni differenti, ciascuna con le proprie peculiarità.

Relazioni minime tra i moduli: nella struttura qui sopra riportata vengono tracciate quelle che sono definite le relazioni minime che possono intercorrere tra i moduli. Ogni implementazione di classe astratta può includere al proprio interno delle relazioni aggiuntive. Per esempio, nell'ambito di questa tesi è stato definito il modulo BaseGoalSolver a partire dalla classe astratta GoalSolver. Tale modulo presenta una relaziona aggiuntiva: un legame con la classe OntologyChecker, utilizzata per controllare la consistenza dei risultati generati dalla fusione di concetti.

2.6 Le meta-informazioni dei moduli

Oltre alle funzioni che vanno implementate dall'utente, ogni classe astratta fornisce una serie di informazioni di meta-livello, ovvero delle informazioni sulla classe in sé, come il tipo di dati trattato dalla classe, oppure il tipo di dato che la classe genera una volta aver assolto il proprio compito. Tali informazioni di meta-livello sono state introdotte per fornire una descrizione di ciascuna classe in quanto tali classi non sono delle "isole", ma sono i mattoncini che compongono il sistema nel suo complesso, e dunque queste informazioni servono per permettere ai diversi moduli di cooperare tra loro.

Il nuovo sistema viene implementato nel seguente modo: date le classi astratte, un utente definisce un modulo per ogni classe astratta, andando a specificare cosa tale modulo nel concreto faccia. Dal momento che i moduli che formano il sistema EDIFICA devono cooperare tra loro, devono condividere tra loro lo schema dei dati. Per esempio, se un modulo A produce come "artefatto" un dato di tipo X e deve comunicare tale "artefatto" al modulo B, con B che però si aspetta in input un dato di tipo Y, questi due moduli non possono cooperare, in quanto hanno schemi di dati differenti.

La descrizione di ogni classe astratta ci permette di definire una serie di informazioni minime che vanno messe a disposizione da parte delle implementazioni delle classi astratte così da poter far cooperare tra loro i vari moduli.

Riportiamo di seguito quali sono le meta-informazioni minime che sono state definite per ciascuna classe astratta e, di conseguenza, presenti in ogni modulo:

- KnowledgeBaseBuilder: tale classe espone come meta-informazioni la struttura del file di input, il formato con cui sono codificate le informazioni singole e il formato utilizzato per codificare le informazioni rigide nella ontologia;
- TipicalitiesBuilder: tale classe espone come meta-informazioni la struttura del file di input, il formato con cui sono codificate le informazioni singole e il formato utilizzato per codificare gli assiomi di tipicalità;
- GoalBuilder: tale classe espone come meta-informazioni la struttura del file di input, il formato con cui sono codificate le informazioni singole e il formato utilizzato per codificare i goal;
- DefinedParser: tale classe espone come meta-informazioni il formato con cui sono codificate le informazioni singole;
- CandidateBuilder: tale classe espone come meta-informazioni il formato dell'ontologia trattata, il formato degli assiomi di tipicalità trattati e il formato dei candidati generati;
- CandidateChooser: tale classe espone come meta-informazioni il formato dei candidati trattati. In questo caso tale meta-informazione rappresenta sia il formato dei candidati in input e sia il formato dei candidati prodotti dalla classe astratta;
- GoalSolver: tale classe espone come meta-informazioni il formato dell'ontologia trattata, il formato degli assiomi di tipicalità trattati e il formato dei candidati trattati per la risoluzione del goal;
- OntologyChecker: tale classe espone come meta-informazioni il formato dell'ontologia che è in grado di trattare e il formato degli assiomi di tipicalità che è in grado di trattare.

Per comprendere a pieno come tali informazioni vengono usate all'interno del sistema, proponiamo un esempio pratico. Supponiamo che nell'implementare la classe KnowledgeBaseBuilder, venga scelto come formato di parsificazione la Manchester Syntax. Essendo che KnowledgeBaseBuilder utilizza per parsificare la classe DefinedParser ed essendo che l'implementazione di KnowledgeBaseBuilder utilizza la Manchester Syntax come formato anche l'implementazione del DefinedParser utilizzato dal KnowledgeBaseBuilder dovrà avere stesso formato, altrimenti i due moduli non potranno coordinarsi.

Come accennato in precedenza, questa serie di meta-informazioni è un insieme minimale, nel senso che ogni utente andando a definire i propri moduli può andare a definirne anche altri. Ovviamente, spetta all'utente definire nuove meta-informazioni e definirne i meccanismi di controllo per permettere la cooperazione tra moduli.

Nei prossimi capitoli analizzeremo i singoli moduli presentandone le varie peculiarità.

3 Flussi di Input

In questo capitolo tratteremo come sono stati gestiti i flussi di input al sistema EDIFICA. Ogni flusso di input verrà trattato a un livello "alto", in cui presenteremo sostanzialmente la semantica dei dati in input, e poi a un livello "basso", in cui presenteremo come praticamente tali informazioni sono trattate e memorizzate in EDIFICA.

I flussi di input in EDIFICA sono tre, che nel nostro caso sono files contenenti informazioni che leggiamo e strutturiamo all'interno del sistema. Andando con ordine abbiamo la base di conoscenza con le relazioni rigide, gli assiomi di tipicalità e il goal da risolvere.

Per ciascun flusso è stato definito un modulo per leggere e strutturare le informazioni basandoci sulle classi astratte definite in fase di modularizzazione.

3.1 Knowledge Base

La Knowledge Base (base di conoscenza) è parte, assieme agli assiomi di tipicalità, della conoscenza di EDIFICA. All'interno della Knowledge Base troviamo tutte quelle informazioni di carattere rigido. La conoscenza del sistema, dunque sia gli assiomi di tipicalità che la base di conoscenza, viene rappresentata all'interno di EDIFICA tramite formalismi logici. Inoltre, le due forme di conoscenza del sistema sono trattate separatamente, per due motivi:

- il primo motivo è di natura semantica: essendo due forme di conoscenza diverse, è ragionevole mantenere queste due forme di conoscenza in file diversi;
- il secondo motivo è di natura pratica: il linguaggio che usiamo per rappresentare la conoscenza del sistema è il linguaggio OWL esteso con gli assiomi di tipicalità. I tool usati per rappresentare la conoscenza, invece, utilizzano OWL puro, senza estensioni. Per questo motivo è utile dividere le due forme di conoscenza: la conoscenza rigida può essere rappresentata con OWL puro, mentre la conoscenza tipica verrà rappresentata con formalismi alternativi che in un secondo momento verranno tradotti in OWL.

3.1.1 OWL e DL

Per Description Logics intendiamo una famiglia di linguaggi di rappresentazione della conoscenza basata sulla logica. Tali linguaggi hanno in comune il fatto che sono espressivamente più potenti della logica proposizionale, ma meno potenti della logica del prim'ordine. Sostanzialmente, più un linguaggio logico è espressivo e più ci permette di catturare determinati aspetti del mondo che rappresentiamo, ma questa espressività comporta anche uno sforzo di computazione maggiore nel compiere del ragionamento logico. I linguaggi appartenenti alla famiglia DL sono linguaggi con buone capacità espressive e con anche una buona complessità in fase di ragionamento logico. OWL è uno di quei linguaggi

che appartiene alla famiglia DL ed è nato per rappresentare conoscenza di tipo ontologico.

Il primo elemento fondamentale è il concetto di ontologia: una ontologia è una rappresentazione astratta di concetti e delle loro relazioni. Le ontologie formali sono quelle ontologie rappresentate secondo un formalismo di rappresentazione esplicito e non ambiguo. Lo scopo delle ontologie formali è quella di condividere una concettualizzazione comune tra individui, organizzazioni e macchine in virtù del linguaggio di rappresentazione non ambiguo utilizzato. Oltre a permettere condivisione, i linguaggi formali permettono anche alle macchine di compiere inferenze sui concetti e a noi di avere certezza della validità di queste inferenze.

Nell'ambito di questa tesi, dunque verrà usato OWL per rappresentare la conoscenza del sistema e per compiere ragionamento sui concetti presenti in tale conoscenza.

In generale, una KB (knowledge base) può essere suddivisa in due porzioni:

- T-Box: porzione terminologica. All'interno di questa porzione vengono definite le classi e le relazioni tra classi;
- A-Box: porzione assertiva. All'interno di questa porzione vengono definiti individui e relazioni tra individui.

Come si può osservare, le entità cardine della rappresentazione della conoscenza sono essenzialmente tre: le **classi**, le **relazioni** e gli **individui**. Per ogni definizione di queste entità verrà fornito un esempio per comprenderne meglio la semantica.

Definizione 3.1 (Individuo)**.** Un individuo è una entità che rappresenta un soggetto particolare che viene trattato nel dominio.
Esempio: supponiamo di essere nel dominio dell'arte. Un individuo che può appartenervi è quello che rappresenta Leonardo Da Vinci, oppure un altro individuo che può appartenervi è quello che rappresenta la sua opera più celebre: La Monnalisa. Come si può osservare queste due sono entità particolari del dominio artistico.

Definizione 3.2 (Classe)**.** Una classe è una entità che rappresenta una famiglia di individui.
Esempio: il concetto di Pittore è quel concetto che rappresenta quell'insieme di individui che sono persone (o più precisamente artisti) e la cui attività principale è quella di dipingere. Vale la pena notare il fatto che una classe non solo identifica delle classi di individui fisici, ma è possibile identificare anche azioni. Per esempio, potremmo definire anche una classe che rappresenta le azioni riguardanti lo spostamento di un corpo nello spazio e all'interno potremmo trovarci il concetto di scivolare, traslare, trascinarsi, correre, rotolare e così via.

Definizione 3.3 (Relazione)**.** Una relazione è un'associazione tra due individui oppure tra due classi.

Esempio: una possibile relazione sempre riferita al dominio dell'arte può essere "produce": è un'associazione tra due concetti, più precisamente legherà tra loro un Artista con una OperaArtistica. Una volta definita questa relazione possiamo anche asserire il seguente fatto:

Leonardo Da Vinci **produce** *La Monnalisa*

Presentiamo più nel dettaglio le caratteristiche delle relazioni in quanto sono molto importanti nella rappresentazione di conoscenza di tipo ontologico. Infatti, gli individui sono collocati principalmente nella A-Box, mentre le classi nella T-Box. Le relazioni si collocano in ambo le parti andando a tracciare una connessione tra A-Box e T-Box.

Relazioni nella T-Box All'interno della T-Box le relazioni sono definite in maniera astratta tramite il concetto di dominio e di codominio. Il dominio può essere visto come il soggetto della relazione, mentre il codominio come l'oggetto della relazione. Nella T-Box dominio e codominio sono classi: le relazioni nella T-Box coinvolgono le classi.
Esempio: la relazione "produce" può avere come dominio la classe Artista e come codominio la classe OperaArtistica.
Inoltre, una classe può essere definita a partire da una relazione.
Esempio: partendo dalla relazione "produce" possiamo definire la classe Pittore come quegli Artisti che producono solo Dipinti. Si può osservare come è stata utilizzata la relazione "produce" e la classe Dipinto per definire come è fatta la classe Pittore.

Relazioni nella A-Box All'interno della A-Box le relazioni sono utilizzate a un livello più pratico: tramite una relazione si stabiliscono legami tra individui.
Esempio: come detto in precedenza, possiamo legare, tramite la relazione "produce", l'individuo Leonardo Da Vinci con l'individuo La Monnalisa.

Come si può osservare, all'interno di OWL si possono scrivere le informazioni tramite triple di entità in cui il primo elemento è detto soggetto, che può essere una classe oppure un individuo, il secondo è il predicato, che è una relazione, e il terzo elemento è l'oggetto, che può essere una classe oppure un individuo.

3.1.2 Ontologie Formali: T-Box

All'interno della T-Box abbiamo i concetti di classe e di relazione. Inoltre, è possibile definire una serie di meta-relazioni, ovvero relazioni definibili solo all'interno della T-Box e che vengono utilizzate per strutturare le relazioni e le classi. Le meta-relazioni che verranno trattate sono tre:

- Relazione di sussunzione: è una meta-relazione tra due classi oppure tra due relazioni ed è una relazione di natura insiemistica. Questa meta-relazione è spesso identificata col nome is-a. Se diciamo che:

  Pittore is-a Artista

stiamo affermando che la classe Pittore è insiemisticamente sottoclasse della classe Artista, ovvero che ogni individuo che appartiene alla classe Pittore appartiene anche alla classe Artista;

- Relazione di equivalenza: è una meta-relazione tra due classi oppure tra due relazioni ed è una relazione di natura insiemistica. Questa meta-relazione è spesso identificata col nome eq-to. Se diciamo che:

  VinoRosso eq-to VinRouge

  stiamo affermando che ogni individuo della classe VinoRosso è anche individuo della classe VinRouge e viceversa;

- Relazione di disgiunzione: è una meta-relazione tra due classi ed è una relazione di natura insiemistica. Questa meta-relazione è spesso identificata col nome disjoint. Se diciamo che:

  VinoRosso disjoint VinoBianco

  stiamo affermando che un individuo che appartiene alla classe VinoRosso non può appartenere alla classe VinoBianco e viceversa.

Possiamo ancora osservare una qualità per ciascuna meta-relazione. La meta-relazione **is-a** è asimmetrica, ovvero che se vale tra A e B non è detto che valga anche tra B ed A. Le meta-relazioni **eq-to** e **disjoint** sono invece simmetriche, ovvero che se valgono tra A e B, allora valgono anche tra B ed A e viceversa.

Tramite gli elementi introdotti possiamo definire nella T-Box le classi e le relazioni e possiamo definire le connessioni tra queste entità tramite le meta-relazioni. In particolare, si dice che OWL è un linguaggio property-centric, ovvero è un linguaggio in cui definiamo le relazioni e poi definiamo le classi a partire dalle relazioni. Come possiamo fare ciò? Nel caso delle classi, le meta-relazioni non sono applicabili solo tra due classi, ma anche tra una classe e quella che viene chiamata class expression. Come detto in precedenza, una classe è una rappresentazione di un insieme di individui a cui viene dato un nome. Una class expression può essere vista come una formula scritta in una determinata sintassi che definisce una classe di individui, ma senza avere un nome, come fosse una classe anonima. Vediamo un esempio:

- la seguente class expression identifica tutti quegli individui che sono artisti e che producono solo ed esclusivamente dipinti:

  (*Artista* **and** (*produce* **only** *Dipinto*))

- come specificato poco fa, una class expression identifica una serie di individui ma non specifica un nome, un modo con cui possiamo identificare questo gruppo di individui. Per questo motivo, possiamo associare un

nome creando un classe e affermando che essa è equivalente a tale class expression:

$$Pittore \equiv (Artista \textbf{ and } (produce \textbf{ only } Dipinto))$$

.

3.1.3 Class Expressions

Come detto precedentemente, possiamo legare tramite meta-relazioni due classi oppure una classe e una class expression. Una class expression è appunto una espressione in un certo linguaggio che permette di definire quali sono i vincoli che una gli individui di tale classe rispettano. Un linguaggio presuppone una sintassi e una semantica. Per quanto riguarda la sintassi, nel nostro caso le class expression sono state espresse in Manchester Syntax, una sintassi molto semplice e leggibile per definire espressioni logiche.

Un elemento fondamentale per poter trattare le informazioni è il parsificatore: è un software il cui compito è quello di leggere informazioni in un certo linguaggio e tradurli in informazioni strutturate all'interno del sistema. Nel nostro caso è stato definito un modulo che implementa la classe astratta DefinedParser. Il compito di questo modulo è quello di leggere le informazioni di input secondo la Manchester Syntax e di generare le strutture dati apposite all'interno del sistema per tali informazioni. Per fare ciò, è stata utilizzata la libreria python boolean-parser che fornisce quello che possiamo definire un parsificatore di base al cui interno possiamo andare a specificare la sintassi che più preferiamo.

Riportiamo di seguito la sintassi delle class expression in Manchester Syntax:

$$\begin{aligned} rop &\leftarrow some \mid only \\ atom &\leftarrow (not\ expr) \mid property\_name\ rop\ expr \mid class\_name \mid (expr) \\ conj &\leftarrow atom \mid atom\ and\ conj \\ expr &\leftarrow conj \mid conj\ or\ expr \end{aligned}$$

Vediamo di preciso il significato di ciascun simbolo:

rop: restriction operator. All'interno di questo simbolo non terminale troviamo due operatori della logica OWL: some, che corrisponde al quantificatore esistenziale della First-Order Logic, e only, che corrisponde al quantificatore universale della First-Order Logic;

conj: rappresenta una congiunzione di atomi oppure un atomo singolo. La congiunzione viene create per emzzo dell'operatore logico AND;

expr: rappresenta una espressione espressa nella Manchester Syntax. Esso può essere una congiunzione oppure una congiunzione legata ad un'altra espressione tramite operatore logico OR;

atom: rappresenta un elemento di base delle espressioni e può essere una negazione di un'altra espressione, il nome di una classe, una espressione compresa tra parentesi oppure una restrizione su proprietà. Una restrizione su proprietà coinvolge un operatore di restrizione, una proprietà e una espressione.

Osserviamo che, per mezzo della definizione di sintassi data, viene in maniera naturale definita una regola di precedenza: l'operatore logico **and** ha la precedenza sull'operatore logico **or**. Per esempio, la espressione riportata di seguito:

$$A \sqcup B \sqcap C$$

Viene parsificata come:

$$A \sqcup (B \sqcap C)$$

Di conseguenza, in fase di parsificazione l'operatore di congiunzione ha la precedenza e "impacchetta" B con C assieme e successivamente l'operatore di disgiunzione "impacchetta" A con $(B \sqcap C)$.

Come specificato, un linguaggio è dotato di una sintassi, ma anche di una semantica, dunque presentiamo brevemente il significato di ogni operatore presentato qui sopra:

- and: è un operatore logico che rappresenta la congiunzione e, più precisamente, in OWL rappresenta l'intersezione insiemistica tra due classi;
- or: è un operatore logico che rappresenta la disgiunzione e, più precisamente, in OWL rappresenta l'unione insiemistica tra due classi;
- not: è un operatore logico che rappresenta la negazione e, più precisamente, in OWL rappresenta l'insieme complemento di una classe;
- only: è un operatore di restrizione che rappresenta la quantificazione universale ed è riferito a una relazione R e a una espressione E. La restrizione sta nel fatto che si va a specificare, tramite la restrizione, quella classe di individui che hanno il ruolo di soggetto all'interno della tripla OWL in cui compare la relazione R come predicato e con oggetto un individuo coerente con la espressione E;
- some: è un operatore di restrizione che rappresenta la quantificazione esistenziale ed è riferito a una relazione R e a una espressione E. La restrizione sta nel fatto che si va a specificare, tramite la restrizione, quella classe di individui in cui esiste almeno un individuo che ha il ruolo di soggetto all'interno della tripla OWL in cui compare la relazione R come predicato e con oggetto un individuo coerente con la espressione E.

3.1.4 Ontologie Formali: A-Box

All'interno della A-Box abbiamo i concetti di individuo e di relazione in cui una relazione serve per associare un soggetto a un oggetto, ovvero per legare tra loro due individui. Anche qui abbiamo a disposizione una meta-relazione: instance-of. Questa meta-relazione ci permette di asserire che un determinato individuo appartiene a una determinata classe: instance-of è una meta-relazione perchè non lega tra loro entità di stessa natura (due classi tra loro o due individui tra loro), ma lega due entità di natura diversa: un individuo e una classe. Questa meta-relazione ci perette di asserire, per esempio, che Michelangelo è un Artista, scrivendo:

Michelangelo *istance_of* **Artista**

Sia nella T-Box che nella A-Box, le meta-relazioni sono utilizzate come predicato per legare tra loro un soggetto e un oggetto.

3.1.5 Ragionamento automatico in OWL

Tramite le entità e le meta-relazioni definite possiamo strutturare i concetti di un dato dominio e poter compiere del ragionamento. Per esempio, un tipo di ragionamento è quello di verificare la consistenza o meno di un individuo. Se infatti affermiamo che le classi VinoRosso e VinoBianco sono disgiunte e affermiamo che il vino V appartiene ad ambo le classi, un software di ragionamento basato su OWL può segnalarci il fatto che l'individuo V è un individuo che genera inconsistenza.

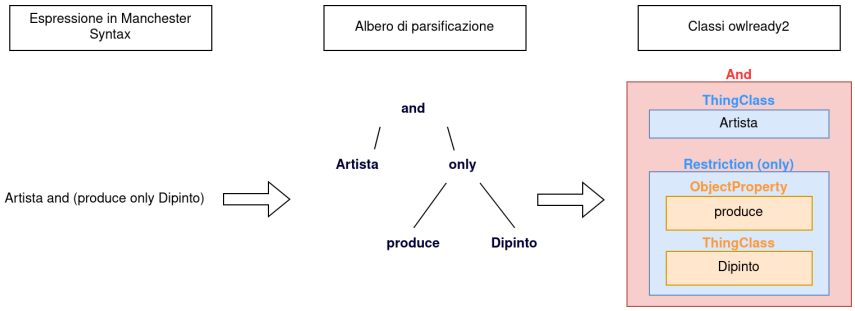
Nella seguente tesi utilizzeremo un reasoner OWL molto diffuso: HermiT. Tale reasoner compie ragionamento tramite ipotesi di mondo aperto. Un reasoner prende un insieme di concetti articolati secondo un formalismo, nel nostro caso un'ontologia, e va a determinare della nuova conoscenza partendo da quella che già ha: andrà a generare dei nuovi fatti. Un reasoner che compie ragionamento basato su ipotesi di mondo aperto inferisce solo quei fatti di cui riesce a dimostrare la veridicità (o alternativamente la falsità). Il caso speculare è il ragionamento basato su ipotesi di mondo chiuso: un reasoner che compie un ragionamento di questo genere asserirà come falso tutto ciò che non riesce a dimostrare.

3.1.6 Trattemento delle class expression in GOCCIOLA

Per trattare le class expression è stato definito un modulo di parsificazione a partire dalla classe astratta *DefinedParser*. Tale modulo si chiama **ManchesterParser** e il suo compito è quello di, data una espressione rappresentata come una stringa in Manchester Syntax, tradurla in entità software strutturate all'interno del sistema. Le strutture utilizzate sono fornite da una libreria python chiamata owlready2. Tale libreria definisce una serie di entità software che ci permettono di definire le classi, le proprietà, le restrizioni e le class expression.

Nel caso di owlready2 le classi definite nella libreria sono: la classe And, la classe Or, la classe Not, la classe Restriction, al cui interno possiamo specificare se la restrizione coinvolge l'operatore **only** oppure l'operatore **some**, e la classe ThingClass per rappresentare classi definite all'interno dell'ontologia.

In pratica, il modulo ManchesterParser prende la espressione e la parsifica, generando l'albero di parsificazione della espressione. Tale albero è puramente sintattico, a ogni parola della espressione associa un nodo dell'albero. Infine, dato l'albero di parsificazione, il modulo parte dalle foglie e, con un approccio bottom-up, costruisce mano a mano le entità software di owlready2 che rappresentano l'espressione in input. Si può osservare come l'oggetto finale abbia una forma "a matriosca".



3.1.7 Trattemento della conoscenza rigida in EDIFICA

Per quanto riguarda la strutturazione della conoscenza, ovvero la definizione di classi, relazioni, individui e meta-relazioni definite nelle sezioni precedenti, la classe astratta di riferimento è *KnowledgeBaseBuilder*, da cui è stato creato il modulo **KnowledgeBaseBuilderManchester**.

La seguente tesi si è concentrata principalmente sulla T-Box di una ontologia. Le T-Box di una ontologia vengono strutturata come un grafo al cui interno vengono definite le classi, le relazioni e le meta-relazioni. Nell'ambito della nostra tesi ci siamo concentrati principalmente sulla strutturazione delle classi, ma una possibile estensione futura è quella di trattare anche estensioni delle relazioni tra classi.

Per strutturare il grafo e per compiere visite al suo interno in maniera efficiente (per ricerca di concetti e per compiere reasoning) è stata utilizzata nuovamente la libreria owlready2. Tramite le entità definite in questa libreria sono state costruite le classi e le relazioni che intercorrono tra esse. La documentazione con esempi di creazione di classi, relazioni e dell'uso delle meta-relazioni di OWL è disponibile al seguente link:
https://owlready2.readthedocs.io/en/v0.42/

Il modulo KnowledgeBaseBuilderManchester va a leggere la base di conoscenza, che è salvata su file, e va a costruire internamente una ontologia "standard",

ovvero contenente solo proprietà rigide, tramite owlready2. Per il modulo in questione è stato reso possibile leggere due tipi di file:

i File in formato owl: tali file possono essere prodotti tramite software appositi di strutturazione di ontologie, come Protégé. In questo caso è possibile utilizzare la libreria owlready2 per caricare automaticamente la base di conoscenza;

ii File in formato user-defined. Nel nostro caso abbiamo definito un formato di file in cui l'utente definisce le relazioni, le classi e le meta-relazioni. In questo caso sono state implementate delle funzioni di parsificazione per leggere il file ed andare a generare, tramite owlready2, la base di conoscenza.

L'esempio del dominio dell'arte viene riportato come esempio qui sotto per mostrare come è strutturato internamente il file in formato user-defined:

```
Properties: produce

Classes:

class Opera:
    subClass of: owl.Thing

class Quadro:
    subClass of: Opera

class Artista:
    subClass of: owl.Thing

class Pittore:
    equivalentClass: Artista and (produce only Quadro)


Properties Domain-Ranges:
    produce: Artista -> Opera

General Axioms:
    Artista disjoint Opera
```

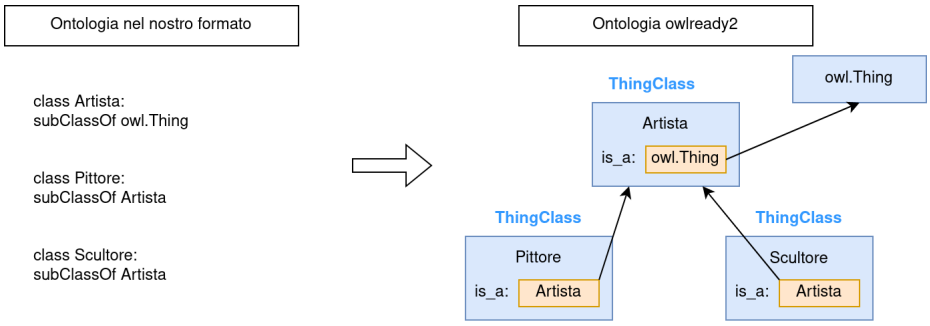
Come si può osservare, il file può essere suddiviso in 4 sezioni:

i Sezione di dichiarazione delle relazioni: per ogni relazione viene dichiarato il nome;

ii Sezione di dichiarazione delle classi: per ogni classe viene dichiarato il nome e a seguito viene dichiarata ogni sua meta-relazione **is-a** e/o **eq-to** verso altre classi;

iii Sezione di dichiarazione di domain e range delle relazioni: per ogni relazione viene definito quale è il suo range e il suo domain;

iv Sezione di dichiarazione di disgiunzioni: possiamo definire all'interno di questa sezione per ogni coppia di concetti la meta-relazione **disjoint**.



Come si può osservare, il seguente modulo va a definire le classi della ontologia in owlready tramite la classe nativa ThingClass e poi a va tracciare le relazioni tra tali concetti tramite le meta-relazioni. Ricordiamo che in questo caso la ontologia generata è una ontologia "standard", ovvero una ontologia formale contenente solo ed esclusivamente relazioni rigide.

3.2 Typicalities

All'interno della base di conoscenza vengono definiti i concetti come classi e tali classi sono legate tra loro tramite le meta-relazioni. Le meta-relazioni sono rigide, nel senso che non ammettono eccezioni: se affermiamo che una classe A è sottoclasse di B, allora ogni singolo individuo di A apparterrà a B. Nella realtà, non sempre le cose sono così: all'interno dello studio del mondo animale definiamo la classe Pinguino come una sottoclasse della classe Uccello. Essendo gli uccelli esseri viventi che sanno volare ed essendo un pinguino un uccello, tramite il linguaggio OWL si può dedurre che anche i pinguini volano, essendo sottoclasse degli uccelli. Ovviamente non è così e questo appunto accade a causa della rigidità della meta-relazione **is-a**. Osserviamo dunque che i concetti tra loro sono legati, ma tali legami hanno valenza differente: alcuni legami sono forti, rigidi, mentre altri legami sono più deboli, più flessibili. L'idea dunque diventa quella di aggiungere una estensione al linguaggio OWL per catturare legami più deboli: introduciamo gli assiomi di tipicalità.

Un assioma di tipicalità può essere interpretato come una meta-relazione tra due concetti in cui viene specificato un valore di tipicalità. Tale valore può essere interpretato come un valore di probabilità. Possiamo esprimere un assioma di tipicalità secondo la seguente sintassi:

$$T(A) \sqsubseteq B : p$$

La semantica di questo assioma è: il tipico esemplare della classe A è una istanza appartenente alla classe B con probabilità p. In questa maniera abbiamo definito una meta-relazione tra le due classi A e B che è meno rigida rispetto a tutte le altre meta-relazioni native di OWL: con probabilità p tale meta-relazione vale. Con il termine "testa dell'assioma" ci riferiremo a $T(A)$, con il termine "corpo dell'assioma" ci riferiremo a B e con il termine "probabilità dell'assioma" ci riferiremo a p.

3.2.1 Trattemento della tipicalità in EDIFICA

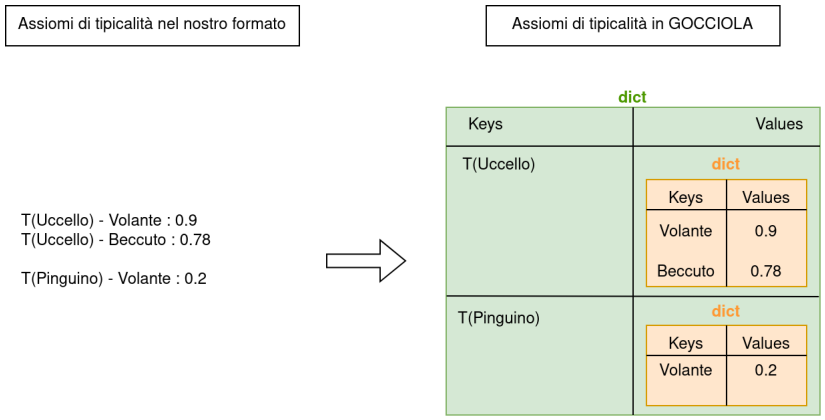
Gli assiomi di tipicalità sono stati gestiti in EDIFICA tramite il modulo **BaseTipicalityBuilder**, creato a partire dalla classe astratta *TipicalityBuilder*. Tale modulo ha il compito di leggere gli assiomi da file e memorizzarli in un dizionario, una struttura nativa del linguaggio Python. Il file è strutturato come una lista di assiomi di tipicalità, in cui a ogni riga corrisponde un assioma. Riportiamo un esempio di seguito:

```
T(Uccello) - Volante : 0.9
T(Pinguino) - Volante : 0.2
T(Studente) - Giovane : 0.75

...
```

Gli assiomi sono memorizzati in un dizionario in Python che corrisponde a una HashTable in cui le chiavi sono la testa dell'assioma (nel nostro esempio T(Uccello), T(Pinguino) e T(Studente)), mentre i valori sono un altro dizionario. In tale dizionario di secondo livello memorizziamo come chiavi il corpo dell'assioma (nel nostro esempio Volante e Giovane) e come valore la probabilità dell'assioma.

Di seguito riportiamo la trasformazione delle informazioni in input in dizionari python all'interno di GOCCIOLA:



Osserviamo la seguente cosa: per come abbiamo definito il nostro modulo di trattamento delle tipicalità, la forma dell'assioma di tipicalità presenta sempre come corpo un concetto. Nel nostro esempio abbiamo che:

$$T(Pinguino) \sqsubseteq Volante : 0.2$$

Tale assioma di tipicalità è equivalente a:

$$T(Pinguino) \sqsubseteq Not(Volante) : 0.8$$

Di conseguenza, se l'utente vuole esprimere che il tipico pinguino non sa volare, deve affermare che tipicamente è volante con una probabilità minore di 0.5.

Un'ultima osservazione che compiamo è il fatto che gli assiomi di tipicalità sono una estensione di OWL, dunque sia le informazioni rigide che quelle flessibili sono di natura ontologica. Vedremo nei prossimi capitoli infatti che gli assiomi di tipicalità possono essere tradotti in entità di OWL.

3.3 Goals

Il terzo ed ultimo input al sistema sono i goal. Un goal viene rappresentato come una class expression che il sistema deve tentare di risolvere. In GOCCIOLA i goal sono class expression in cui sono coinvolte solamente classi definite nell'ontologia e in cui è utilizzato solo ed esclusivamente l'operatore logico di congiunzione. Nella nuova versione è stato reso possibile definire un goal come una qualsiasi class expression di OWL, permettendo di coinvolgere restrizioni, negazioni e disgiunzioni.

La risoluzione del goal consisterà, come vedremo più nel dettaglio nel capitolo 4, nel cercare un concetto che sia equivalente oppure sottoclasse del goal specificato dall'utente; in mancanza di un concetto che soddisfi il goal il sistema combinerà vari concetti per determinare un nuovo concetto che sarà legato al goal tramite la meta-relazione **eq-to**.

3.3.1 Trattemento dei goal in EDIFICA

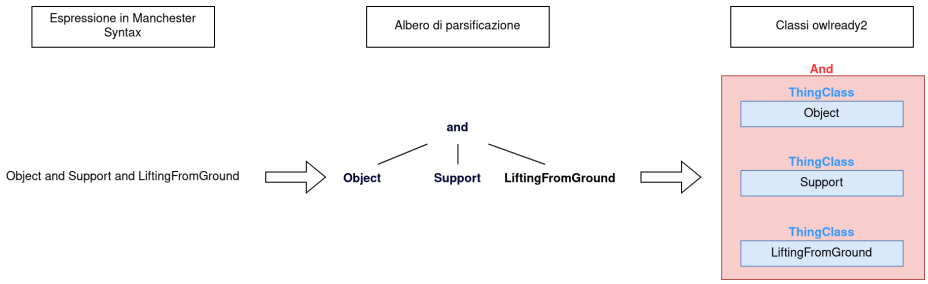
Come nei casi precedenti viene definito un modulo per leggere e strutturare il goal all'interno del sistema: tale modulo è **BaseGoalBuilder** e fa riferimento alla classe astratta *GoalBuilder*.

Il goal è contenuto in un file di testo, che viene parsificato e tradotto in una entità software con struttura ad albero che viene salvata all'interno del sistema. Essendo il goal equivalente a una class expression, il lavoro di lettura, parsificazione e costruzione dell'entità software che rappresenta tale class expression è fatto dal parser presente in BaseGoalBuilder.

Come nei casi precedenti, presentiamo un esempio di come è strutturato il file in input:

```
Object and Support and LiftingFromTheGround
```

Riportiamo inoltre di seguito come viene strutturata all'interno di EDIFICA l'espressione che rappresenta il goal:



3.3.2 Indebolimento dei goal

Sebbene la parte di costruzione della class expression del goal sia fatta in buona parte dal modulo dedicato al parsing, il modulo di gestione del goal fornisce un meccanismo che in GOCCIOLA non era presente: il meccanismo di indebolimento dei goal.

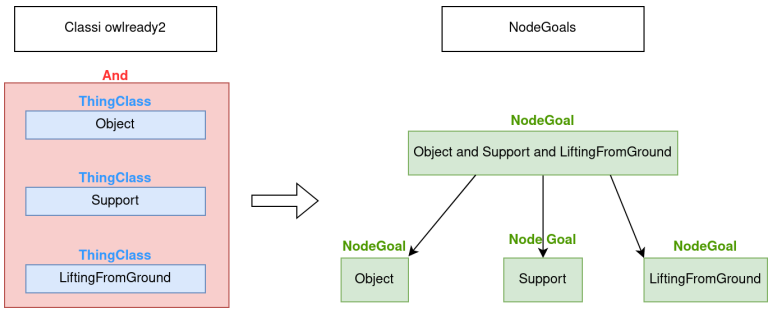
Per indebolimento di un goal G intendiamo un goal G' non vuoto ottenuto andando ad eliminare delle parti del goal di partenza G. A seconda della forma del goal di partenza possiamo definire diversi indebolimenti per tale goal.

Andando più nel dettaglio, il modulo di lettura e di strutturazione dei goal definisce al proprio interno una classe user-defined chiamata NodeGoal. Tale classe rappresenta un nodo dell'albero dei goal. Nella strutturazione ad albero, ogni nodo presenta la class expression a cui fa riferimento e ha annesso una lista di nodi discendenti, ognuno dei quali rappresenta una sotto-espressione del nodo genitore.

Data la class expression E_g che rappresenta il goal da risolvere, si può creare un NodeGoal a partire da E_g. Per come è definita la classe NodeGoal, verrà

creato il nodo corrispondente ad E_g con annessi sotto-nodi. Ogni sotto-nodo corrisponderà a una sotto-espressione di E_g con annessi sotto-nodi. Questo meccanismo ricorsivo di creazione dell'albero continuerà fino a raggiungere un caso base, ovvero quando la espressione da cui si crea il nodo è una classe. A questo punto l'albero rappresentante il goal E_g è completo.

Sostanzialmente, partendo dalla class expression iniziale, rappresentata da una entità di owlready2, si va definire un albero in cui ogni nodo ha associato a sé una sotto-espressione della class expression di partenza.



Come si evince dall'esempio riportato qui sopra, vengono creati un NodeGoal radice, corrispondente alla class expression del goal definito dall'utente, e tre nodi figli, ognuno corrispondente all'argomento della congiunzione: Object, Support e LiftingFromGround.

In sostanza, il modulo BaseGoalBuilder prende un file al cui interno è presente una stringa rappresentante il goal. Utilizza il proprio parser per parsificare tale stringa definendo la corrispondente entità owlready2. Data la entità che rappresenta l'espressione il modulo infine va a generare i vari NodeGoal. Sui NodeGoal si applicherà il meccanismo di indebolimento del goal.

Per poter fare ciò, la classe NodeGoal definisce la funzione getNextGoal(). Tale funzione sfrutta la natura ricorsiva della struttura ad albero di NodeGoal per andare mano a mano ad indebolire il goal di partenza. Questa funzione, quando termina le opzioni di indebolimento, restituisce il valore *None*, che corrisponde a una class expression vuota, altrimenti restituisce il goal indebolito come class expression di owlready2. Di seguito presentiamo come avviene l'indebolimento di un goal in base al tipo di operatore logico o di restrizione che viene applicato.

3.3.3 Indebolimento di classe

Indebolire un goal che corrisponde a una classe definita nella ontologia consiste nel restituire immediatamente None: l'indebolimento di una classe porta a un goal vuoto.

3.3.4 Indebolimento di class restriction

Indebolire un goal che corrisponde a una class restriction corrisponde ad indebolire l'argomento di tale class restriction. Se non è possibile indebolire ulteriormente tale argomento, allora l'indebolimento della class restriction corrisponde a un goal vuoto.

3.3.5 Indebolimento di And

Indebolire un goal che consiste in una congiunzione di N elementi corrisponde al prendere l'indebolimento di ciascun elemento. Dati tutti gli indebolimenti si vanno a creare tutte le possibili combinazioni di unioni insiemistiche di tali indebolimenti. Possiamo enumerare tutti i possibili insiemi creabili, lasciando l'insieme vuoto per ultimo. Quando rimane come indebolimento l'insieme vuoto significa che non è più possibile indebolire il goal e quindi arriviamo al goal vuoto.

3.3.6 Indebolimento di Or

Indebolire un goal che consiste in una disgiunzione di N elementi corrisponde al prendere l'indebolimento di ciascun elemento. Dati tutti gli indebolimenti si vanno a creare tutte le possibili combinazioni di unioni insiemistiche di tali indebolimenti escludendo, per ogni indebolimento, l'insieme vuoto. Possiamo enumerare tutti i possibili sottoinsiemi creabili, aggiungendo l'insieme vuoto per ultimo. Quando rimane come indebolimento l'insieme vuoto significa che non è più possibile indebolire il goal e quindi arriviamo al goal vuoto.

3.3.7 Indebolimento della Not

Indebolire un goal che consiste in una negazione corrisponde a dover indebolire il goal in maniera diversa a seconda dell'argomento della negazione. Ciò che resta vero in ogni caso che vediamo è che se l'indebolimento dell'argomento porta a un goal vuoto, allora anche la negazione nel suo complesso porta a un goal vuoto. Se la negazione ha come argomento:

- una classe;
- un'altra negazione;
- una class expression.

allora semplicemente l'indebolimento consiste nell'indebolire l'argomento e di aggiungerci in seguito la negazione.
Se la negazione ha per argomento una disgiunzione di elementi, allora corrisponde, tramite le regole di DeMorgan, a indebolire una congiunzione in cui ogni elemento è negato.

Infine, se la negazione ha per argomento una congiunzione di elementi, allora corrisponde, tramite le regole di DeMorgan, a indebolire una disgiunzione in cui ogni elemento è negato.

3.3.8 Qualche esempio concreto di indebolimento

Presentiamo tre esempi per comprendere meglio come funziona l'indebolimento dei goal.

Esempio 1:

$$G = A \sqcap \neg(C \sqcup D)$$

Essendo una congiunzione, calcoliamo l'insieme degli indebolimenti di ciascun elemento della congiunzione preso singolarmente:

$$A \rightarrow \{A, \emptyset\}$$

Per la negazione, applichiamo la regola di DeMorgan:

$$\neg(C \sqcup D) \equiv (\neg C) \sqcap (\neg D)$$

Essendo di nuovo una congiunzione, andiamo a calcolare per ogni elemento della congiunzione ogni suo indebolimento:

$$(\neg C) \rightarrow \{\neg C, \emptyset\}$$
$$(\neg D) \rightarrow \{\neg D, \emptyset\}$$

A questo punto combiniamo gli indebolimenti che sono coinvolti nella negazione facendo tutte le combinazioni ottenendo:

$$\neg(C \sqcup D) \equiv (\neg C) \sqcap (\neg D) \rightarrow \{\neg C, \neg D, \neg(C \sqcup D), \emptyset\}$$

Abbiamo ottenuto tutti gli indebolimenti possibili per ogni elemento che compone la congiunzione principale, dunque andiamo a farne tutte le possibili combinazioni ottenendo:

$A \sqcap \neg(C \sqcup D)$	$A \sqcap \neg C$	$A \sqcap \neg D$	A
$\neg(C \sqcup D)$	$\neg C$	$\neg D$	$\emptyset$

Come si può osservare, nel caso della congiunzione, se per un elemento E si raggiunge un indebolimento che crea un insieme vuoto, semplicemente nell'indebolimento della congiunzione totale l'indebolimento di E non comparirà, perchè andiamo a compiere una unione insiemistica tra gli indebolimenti degli elementi che formano la congiunzione.

Esempio 2:

$$G = (A \sqcap C) \sqcup D)$$

Procediamo andando a calcolare l'insieme degli indebolimenti di ciascun elemento della disgiunzione presi singolarmente.

Per quanto riguarda la congiunzione, procediamo nel calcolare tutti gli indebolimenti degli elementi che compongono la congiunzione:

$$A \rightarrow \{A, \emptyset\}$$

$$C \rightarrow \{C, \emptyset\}$$

Possiamo allora affermare che gli indebolimenti del primo elemento della disgiunzione sono:

$$(A \sqcap C) \rightarrow \{A \sqcap C, A, C, \emptyset\}$$

Per quanto riguarda il secondo elemento della disgiunzione, gli indebolimenti sono:

$$D \rightarrow \{D, \emptyset\}$$

A questo punto andiamo a creare tutte le possibili combinazioni tra gli indebolimenti e andiamo ad unire gli insiemi rappresentati da tali indebolimenti. Essendo una disgiunzione dobbiamo prendere, per ciascun elemento, i suoi indebolimenti eccetto l'insieme vuoto. Dobbiamo dunque andare a costruire tutte le possibili combinazioni tra:

$$\{A \sqcap C, A, C\} \text{ e } \{D\}$$

A tutte le combinazioni ottenute aggiungeremo l'insieme vuoto ottenendo:

$$(A \sqcap C) \sqcup D \qquad A \sqcup D$$

$$C \sqcup D \qquad \emptyset$$

Esempio 3:

$$G = p\ some\ (A \sqcap B)$$

Essendo questa una restrizione, andiamo a calcolare tutti i possibili indebolimenti del suo argomento:

$$A \sqcap B$$

Dalla congiunzione prendiamo i suoi elementi e li indeboliamo:

$$A \rightarrow \{A, \emptyset\}$$

$$B \rightarrow \{B, \emptyset\}$$

Poi creiamo ogni possibile combinazione tra gli indebolimenti degli elementi che compongono la congiunzione:

$$A \sqcap B \rightarrow \{A \sqcap B, A, B, \emptyset\}$$

Infine generiamo gli indebolimenti della restrizione tramite gli indebolimenti del suo argomento:

$$p\ some\ (A \sqcap B) \qquad p\ some\ (A)$$

$$p\ some\ (B) \qquad \emptyset$$

4 Flussi di Output

In questo capitolo tratteremo come sono stati gestiti i flussi di output del sistema EDIFICA. Per flussi di output intenderemo tutti quei risultati intermedi o finali che vengono considerati dal software in fase di esecuzione. Per ogni flusso di output è stato definito un modulo e per ciascun modulo verrà fornita una descrizione ad alto livello di quali compiti svolge e una descrizione più tecnica di come svolge in maniera effettiva il proprio lavoro.

Da ora in poi con il termine base di conoscenza ci riferiremo alla conoscenza di natura rigida, con il termine assiomi di tipicalità ci riferiremo alla conoscenza di natura flessibile e con il termine ontologia ci riferiremo a tutta la conoscenza del sistema, ovvero sia quella rigida che quella flessibile, in quanto entrambe le informazioni sono codificabili all'interno di una ontologia formale tramite il linguaggio OWL.

4.1 Ontology Checking

Il primo modulo che è stato definito è quello di controllo di consistenza della conoscenza del sistema. La classe astratta di riferimento è *OntologyChecker*, mentre il modulo di implementazione è **BaseOntologyChecker**.

Il compito di questo modulo è quello di controllare che le informazioni di natura rigida unite con quelle di natura tipica siano consistenti. Le informazioni della base di conoscenza e le informazioni degli assiomi di tipicalità vengono unite e vanno a costituire la ontologia del sistema. In particolare, questi controlli vengono compiuti nei seguenti momenti:

1. dopo aver caricato la ontologia di partenza, dotata delle sole relazioni rigide, e gli assiomi di tipicalità;

2. nel caso in cui si abbia un candidato semplice, bisogna controllare che l'aggiunta di tale candidato come risolutore del goal non generi inconsistenze;

3. nel caso in cui si abbia un candidato complesso, bisogna controllare che l'aggiunta di tale candidato come risolutore del goal non generi inconsistenze.

La classe BaseOntologyChecker in fase di inizializzazione prende la base di conoscenza, definita come una ontologia "standard", ovvero una ontologia con le sole relazioni rigide, e una serie di assiomi di tipicalità. Dopo aver utilizzato i moduli che gestiscono i flussi di input, abbiamo a disposizione in un oggetto owlready2 la base di conoscenza, ovvero le classi legate da relazioni rigide, mentre in un dizionario Python gli assiomi di tipicalità, ovvero le relazioni flessibili. Tali entità software sono utilizzate per inizializzare la classe BaseOntologyChecker.

Controlli dopo il caricamento della conoscenza La prima cosa che viene fatta è controllare che la base di conoscenza letta in input sia consistente. Per fare ciò è stata definita la funzione *check_ontology_consistency* che utilizza il

reasoner Hermit sulla base di conoscenza. Come avviene questo controllo sarà presentato a breve in seguito.

Se la base di conoscenza è consistente, si procede a verificare che gli assiomi di tipicalità lo siano: questo compito è svolto da *check_tipicalities_consistency*. Come avviene questo controllo sarà presentato a breve in seguito.

Controlli per un candidato semplice In questo contesto il sistema trova un candidato semplice per risolvere un goal dato dall'utente. Ricordiamo che un candidato semplice è un insieme di concetti composto da un solo elemento.

Di conseguenza, si deve controllare che una data classe C definita nell'ontologia sia in grado di soddisfare il goal G. Per fare ciò è stata predisposta la funzione *check_single_class* che prende in input la classe C. Il controllo avviene come segue: data la classe C si va a creare una seconda classe C_{bis} che è sottoclasse di C e che è equivalente al goal G che risolve. Una volta creata C_{bis}, la si aggiunge all'ontologia e si controlla che la conoscenza sia ancora consistente: se lo è si restituisce True, altrimenti False.

L'idea di creare una sottoclasse di C ha il seguente significato: si definisce il fatto che C può risolvere il goal G, dunque non per forza tutti gli individui appartenenti a C risolvono G, ma solo alcuni, ovvero una sua sottoclasse C_{bis}.

Controlli per un candidato complesso In questo contesto il sistema trova un candidato complesso *cc*, composto da N concetti. Inoltre, il sistema ha prodotto uno scenario s che rappresenta le proprietà tipiche del concetto fusione nato da *cc*. Siamo in un contesto in cui dobbiamo controllare se il concetto fusione assieme alle sue proprietà tipiche è consistente. Per fare ciò, è stata definita la funzione *add_tipical_combined_attrs*: presi in input *cc* ed s, quello che si va a fare è la seguente cosa: si crea una classe CF che rappresenta il concetto fusione e lo si pone equivalente alla congiunzione dei concetti che compongono *cc* essendone la fusione. Poi, ogni elemento dello scenario, ovvero ogni assioma di tipicalità in s, viene aggiunto alla ontologia completa (con relazioni rigide e rilassate). Una volta fatto ciò, la funzione *is_consistent* controlla se la ontologia arricchita con il nuovo concetto e le sue proprietà tipiche è ancora consistente: tale funzione restituirà True se lo è, False altrimenti.

Per poter aggiungere classi e proprietà alla ontologia sono state definite due funzioni private di servizio: *create_class* e *create_property*.

Presentiamo di seguito come si controllano le inconsistenze nel caso di informazioni di natura rigida e nel caso di informazioni di natura tipica.

4.1.1 Controlli di consistenza della base di conoscenza

Per quanto riguarda il controllo della consistenza della base di conoscenza, il modulo da noi definito utilizza una funzione fornita da owlready2 che ci permette di utilizzare il reasoner HermiT sopra la base di conoscenza memorizzata in EDIFICA. Una volta lanciato, tale reasoner ci restituirà una lista di concetti

che sono stati trovati inconsistenti. Se tale lista è vuota significa che in fase di ragionamento non sono state trovate inconsistenze e che, di conseguenza, la base di conoscenza è consistente.

La libreria owlready2 oltre al reasoner HermiT ne fornisce un secondo, Pellet: tra i due è stato scelto HermiT in quanto è quello fornito di default e, per lo scopo di questa tesi, la scelta di uno o di un altro reasoner non è stato oggetto di ottimizzazione.

Presentiamo rapidamente come funziona un reasoner OWL. Come presentato nel Capitolo 3, OWL è un linguaggio formale che serve per rappresentare la conoscenza e tale linguaggio fornisce una serie di meta-relazioni. Queste ultime sono quelle utilizzate per compiere ragionamento automatico sulle classi presenti nell'ontologia. Un reasoner utilizza le meta-relazioni definite all'interno della base di conoscenza per trovare delle informazioni che non sono state espresse in maniera esplicita e diretta all'interno della conoscenza iniziale.

Per esempio, se affermiamo che la classe $Cane$ è sottoclasse della classe $Mammiferi$ e che quest'ultima è sottoclasse della classe $EssereVivente$, il ragionamento automatico ci può permettere di inferire che $Cane$ sia sottoclasse di $EssereVivente$ a sua volta. Questa informazione non è stata espressa nella base di conoscenza di partenza in maniera diretta, ma in virtù della transitività della meta-relazione **is-a** (essendo equivalente alla relazione di inclusione insiemistica) è stato possibile derivare questa nuova conoscenza.

Quello che un reasoner fa è prendere una base di conoscenza e produrre in output una serie di nuovi fatti che ha potuto dedurre dalla conoscenza di partenza. Quando all'interno dell'insieme di asserzioni prodotte dal reasoner esiste un gruppo di fatti in conflitto tra loro, si dice che la base di conoscenza di partenza è inconsistente, perchè deducendo nuovi fatti a partire da essa troviamo dei fatti in contrasto tra loro.

Mostriamo di seguito tre esempi di inconsistenza.

Inconsistenza dovuta alla definizione delle classi: supponiamo di aver definito nella nostra base di conoscenza tre concetti: Mammifero, Uccello e SpecieX. Supponiamo inoltre di esprimere, tramite la meta-relazione **disjoint** il fatto che le due classi Mammifero e Uccello siano disgiunte, ovvero che non sia possibile avere un individuo che appartiene contemporaneamente alla classe Mammifero e alla classe Uccello. Tramite la meta-relazione **is-a**, supponiamo di affermare che la classe SpecieX, che rappresenta una nuova specie di animale da noi identificata, sia sotto-classe sia di Mammifero che di Uccello. Secondo questa definizione, se eseguiamo un reasoner su tale base di conoscenza, ci verrà segnalata una inconsistenza: nella T-Box abbiamo appena definito una classe di individui, quelli appartenenti alla classe SpecieX, che sarebbero contemporaneamente appartenenti alla classe Mammifero e alla classe Uccello.

Inconsistenza dovuta alla definizione di un individuo: manteniamo, dall'esempio precedente, le due classi Uccello e Mammifero e manteniamo la meta-relazione **disjoint** tra questi due concetti. Supponiamo di definire nella

A-Box un individuo AnimaleMaiVisto, che rappresenta un animale di una specie mai vista. Se, tramite la meta-relazione **istance-of**, affermiamo che AnimaleMaiVisto è istanza sia di Mammifero che di Uccello, un reasoner ci segnalerà una inconsistenza: nella A-Box abbiamo appena definito un individuo che viola la meta-relazione **disjoint**.

Inconsistenza dovuta alla definizione di una classe e di un individuo: per mostrare questa inconsistenza cambiamo scenario: supponiamo di essere di nuovo nel domino dell'arte e di definire la classe Pittore come quella classe Artista che produce solo ed esclusivamente Dipinto. Supponiamo inoltre di definire tre individui: Pippo, appartenente alla classe Pittore, RitrattoDiMarco. appartenente alla classe Dipinto e SculturaDiAntonio appartenente alla classe Scultura. Supponiamo inoltre che nella A-Box specifichiamo il fatto che Pippo ha prodotto sia RitrattoDiMarco che SculturaDiAntonio. In questo caso osserviamo come il ragionamento automatico basato su ipotesi di mondo aperto possa andare a generare dei risultati un po' bizzarri. In questo scenario, non vi sono inconsistenze, in quanto, in assenza di informazioni, nulla ci vieta di assumere che un certo individuo X possa appartenere contemporaneamente alla classe Dipinto e Scultura. Se, però, aggiungiamo il fatto che la classe Dipinto e la classe Scultura devono essere disgiunte, allora si genera una inconsistenza: abbiamo affermato che Pippo, che è un Pittore, produce qualcosa che non è un Dipinto, ovvero SculturaDiAntonio. Per questo motivo si genera una inconsistenza dovuta alla definizione della classe Pittore e alla esistenza di un individuo che ne contraddice la definizione.

4.1.2 Controlli di consistenza degli assiomi di tipicalità

Per quanto riguarda il controllo della consistenza degli assiomi di tipicalità, il modulo da noi definito utilizza una funzione che va a tradurre ciascun assioma in concetti, relazioni e meta-relazioni di OWL.

La traduzione da assiomi di tipicalità ad OWL è stata presentata in [3] e sostanzialmente nel nostro modulo andiamo ad implementare questa traduzione. Presentiamo un esempio per far capire come ciò avviene.

Supponiamo di dover tradurre il seguente assioma di tipicalità:

$$T(Bird) \sqsubseteq Fly : 0.85$$

Per tradurre in OWL andiamo a definire una nuova classe, chiamata Bird1 e andiamo a legare Bird1 con altri concetti tramite la meta-relazione **is-a**:

$$Bird \sqcap Bird1 \sqsubseteq Fly$$
$$Bird1 \sqsubseteq \forall R.(\neg Bird \sqcap Bird1)$$
$$\neg Bird1 \sqsubseteq \exists R.(Bird \sqcap Bird1)$$

Osserviamo le seguenti cose:

- nella seguente notazione il simbolo $\sqsubseteq$ rappresenta la meta-relazione **is-a**. Il simbolo $\equiv$ rappresenta invece la meta-relazione **eq-to**. Infine, il simbolo $\sqcap$ rappresenta la congiunzione in OWL, ovvero l'intersezione insiemistica;
- se abbiamo due assiomi di tipicalità con stessa testa $T(A)$, avremo sempre lo stesso concetto $A1$ di riferimento;
- durante la traduzione andiamo a perdere l'informazione della probabilità di un assioma di tipicalità.

Come presentato nel terzo punto qui sopra, la probabilità di un assioma di tipicalità non viene tradotta in OWL e dunque va memorizzata a parte perchè verrà utilizzata in seguito dal sistema EDIFICA nell'andare a fondere i concetti tra loro. Possiamo però sfruttare tale probabilità anche per fare degli ulteriori controlli di consistenza. Infatti, dati N assiomi di tipicalità con testa $T(A)$, ognuno col proprio corpo B_i non ripetuto, dobbiamo anche controllare la seguente cosa: supponiamo di avere, tra gli N concetti B_i un gruppo G di M concetti B_j che sono tutti disgiunti gli uni dagli altri. Dobbiamo allora controllare che la somma delle probabilità degli assiomi di tipicalità che coinvolgono questi B_j sia al massimo 1. Se tale somma fosse maggiore di 1 significherebbe che ci sono casi in cui due concetti in G hanno qualche istanza in comune, questo sarebbe incoerente con il fatto che gli M concetti in G sono tutti disgiunti tra loro.

Per comprendere bene come si generano inconsistenze nel caso degli assiomi di tipicalità mostriamo un esempio. Supponiamo di avere definito nella nostra base di conoscenza tre concetti che catturano i tre stati della materia: CompostoGassoso, CompostoLiquido e CompostoSolido. Possiamo affermare che questi tre concetti siano tutti disgiunti tra loro. Supponiamo ora di avere definito una classe che rappresenta i composti a base di acqua: CompostoAcquoso. Possiamo affermare che tipicamente un individuo di tale classe è liquido, ma può anche essere solido oppure gassoso. Vediamo due casi di assiomi di tipicalità, uno in cui si generano inconsistenze e uno in cui non se ne generano.

Esempio consistente: supponiamo di specificare i seguenti tre assiomi:

$$T(CompostoAcquoso) \sqsubseteq CompostoGassoso : 0.2$$

$$T(CompostoAcquoso) \sqsubseteq CompostoLiquido : 0.7$$

$$T(CompostoAcquoso) \sqsubseteq CompostoSolido : 0.1$$

Possiamo osservare che sommando le probabilità di ciascun assioma la somma faccia al massimo 1 (nel nostro caso esattamente 1). Questi tre assiomi di per sè non generano inconsistenze, ci dicono come, tipicamente, un CompostoAcquoso si distribuisce a livello di popolazione sui tre stati della materia.

Esempio inconsistente: supponiamo di specificare i seguenti tre assiomi:

$$T(CompostoAcquoso) \sqsubseteq CompostoGassoso : 0.4$$
$$T(CompostoAcquoso) \sqsubseteq CompostoLiquido : 0.7$$
$$T(CompostoAcquoso) \sqsubseteq CompostoSolido : 0.2$$

Possiamo osservare che sommando le probabilità di ciascun assioma la somma sia maggiore di 1: questo significherebbe che ci sono istanze di CompostoAcquoso che tipicamente sono sia liquidi che solidi, andando a contraddire la metarelazione di disgiunzione tra i tre concetti CompostoGassoso, CompostoLiquido e CompostoSolido.

Una volta aver tradotto gli assiomi di tipicalità in OWL, si aggiungono gli assiomi tradotti alla base di conoscenza andando a generare la ontologia finale del sistema. Su questa ontologia si utilizza nuovamente HermiT e si controlla che la lista dei concetti inconsistenti sia vuota. In tal caso la ontologia, ovvero la base di conoscenza arricchita con gli assiomi di tipicalità, è consistente e il sistema EDIFICA può dunque risolvere goal presentati dall'utente.

4.1.3 Criticità della libreria owlready2

Come detto in precedenza, la classe BaseOntologyChecker utilizza al suo interno una serie di funzioni fornite da owlready2 per compiere ragionamento automatico e per controllare la correttezza delle inferenze.

Un aspetto critico delle seguenti funzioni è il seguente: una volta aver compiuto le inferenze, queste vengono direttamente salvate nell'ontologia su cui sono state fatte, di fatto mescolando tra loro fatti pre-esistenti all'interno dell'ontologia con quelle dedotte dal reasoner. Di conseguenza, ogni volta che dobbiamo testare N candidati semplici, oppure dobbiamo testare per un candidato complesso N scenari differenti, dobbiamo compiere una operazione di "rollback" e tornare alla ontologia iniziale, ovvero alla ontologia composta dalla base di conoscenza e dagli assiomi di tipicalità ottenuti dai flussi di input.

Per fare questo, semplicemente è stata salvata la ontologia una volta aver appurato essere consistente. Ogni volta che è stato necessario lanciare il reasoner è stata creata una copia di tale ontologia così da poter creare inferenze senza perdere le informazioni originalmente presenti.

4.2 CandidateBuilder

Presentiamo ora un altro modulo, che aggiunge delle specifiche importanti che non erano presenti in GOCCIOLA. Tale modulo è **BaseCandidateBuilder** che è stato creato a partire dalla classe astratta *CandidateBuilder*. Il modulo in questione va a implementare la funzionalità di trasformazione del goal.

Dato il goal di partenza, strutturato come una class expression di owlready2, come facciamo a dedurre dei possibili gruppi di concetti candidati alla risoluzione di tale goal? Andiamo a tradurre il goal di partenza in un insieme di candidati. Ogni candidato non è altro che un insieme di concetti. Dato un goal

di fatto andiamo a determinare gli insiemi di concetti che possono, potenzialmente, risolvere tale goal. Il compito di questo modulo è quello di colmare un gap semantico: quello che c'è tra il goal di partenza e i possibili candidati alla sua risoluzione. Per fare ciò sono state definite una struttura di supporto e una funzione di supporto.

Classe MyCombination Questa classe rappresenta un insieme di concetti da combinare, ossia una possibile combinazione che può potenzialmente risolvere il goal di partenza. In sostanza, questa struttura rappresenta un candidato.

La funzione search_expression All'interno del modulo BaseCandidateBuilder è stata definita la funzione *search_expression*: è una funzione che, data in input una class expression, cerca quei concetti della base di conoscenza che siano discendenti di tale class expression a seguito di ragionamento automatico tramite HermiT. Questa funzione restituisce solo ed esclusivamente concetti definiti nella base di conoscenza e non altre class expression. All'interno di questa funzione non si utilizza solo la ontologia e il ragionamento automatico per determinare i possibili candidati, ma si utilizzano anche gli assiomi di tipicalità. Infatti, se si viene a sapere che un concetto A è un possibile candidato nella risoluzione del goal e che tipicamente le istanze di A sono anche dei B, allora si includono tra i concetti candidati alla risoluzione del goal anche B.

Data la class expression che rappresenta il goal, il nostro modulo utilizzerà la funzione *search_expression* per cercare i candidati alla risoluzione di tale goal e ogni candidato nuovo trovato corrisponderà alla creazione di una istanza della classe MyCombination.

Il nostro modulo utilizza le informazioni di carattere rigido e di carattere tipico per determinare i candidati alla risoluzione di un goal, ma osserviamo che questa non è l'unica opzione percorribile. Volendo si potrebbe dotare il sistema di un tipo di conoscenza statistica che vada a scegliere quei concetti che, statisticamente parlando, sarebbero quelli più indicati da un essere umano. L'approccio usato nel nostro caso è un approccio volto alla logica: si utilizza il reasoner per andare a decidere quali candidati possono essere plausibili e quali no.

Sebbene questo utilizzo della logica ci possa sembrare distante dal come noi esseri umani pensiamo ed agiamo, in realtà al suo interno contiene delle ipotesi strutturali su come noi esseri umani articoliamo i processi informativi: nel trovare i candidati di un goal complesso si fa uso del sub-goaling, ovvero del processo che consiste nel prendere un problema complesso e scomporlo in una serie di problemi meno complessi da risolvere e provare a risolvere tali problemi meno complessi e, se necessario, scomporli ulteriormente.

Presentiamo di seguito come vengono determinati i concetti candidati in base alla forma di ciascun goal per mezzo della tecnica del sub-goaling. Ricor-

diamo che ogni candidato è formato solo ed esclusivamente da classi definite nella ontologia, dunque non da class expression che, ricordiamo, sono classi che possiamo definire come "anonime".

Candidati per un concetto o una class restriction Dato un concetto semplice o una class restriction in input, i candidati nel risolverlo sono essenzialmente tutti quegli insiemi singoletto in cui ogni concetto contenuto nel candidato è un concetto della ontologia che è equivalente o sottoclasse del concetto/class restriction proposto in input.

Supponiamo che il goal sia la classe C e che A sia sottoclasse di C. Allora i candidati per risolvere il goal sono:

$$candidati_{(C)} \rightarrow \{\{A\}, \{C\}\}$$

Candidati per una disgiunzione Data una disgiunzione di entità, abbiamo più candidati per tale disgiunzione: dobbiamo prendere ogni entità della disgiunzione e valutare quali sono i suoi candidati e i candidati per la disgiunzione non sono altro che l'agglomerazione dei candidati di ciascuna entità della disgiunzione. Vediamo un esempio: supponiamo di avere come concetti le classi A, B e C, di cui sappiamo che A e C sono equivalenti. Supponiamo di dover risolvere il goal $G = A \sqcup B$. Per trovare i candidati della disgiunzione, dobbiamo prendere tutte le entità che compongono la disgiunzione e calcolarne i candidati:

$$A \rightarrow candidati_A = \{\{A\}, \{C\}\}$$

$$B \rightarrow candidati_B = \{\{B\}\}$$

Agglomeriamo i candidati trovati e otteniamo:

$$candidati_{(A \sqcup B)} = \{\{A\}, \{C\}, \{B\}\}$$

Candidati per una congiunzione Dato una congiunzione di entità, abbiamo più candidati per tale congiunzione: dobbiamo prendere ogni entità della congiunzione e valutare quali sono i candidati. Una volta fatto ciò, per ogni entità della congiunzione avremo un insieme di candidati. Per ottenere i candidati della congiunzione si fa il prodotto cartesiano tra questi insiemi, definendo tutte le possibili combinazioni di candidati prendendo un candidato per ciascuna entità della congiunzione. Vediamo l'esempio precedente ma con goal $G = A \sqcap B$. Per risolvere la congiunzione, dobbiamo prendere tutte le entità che compongono la congiunzione e calcolarne i candidati:

$$A \rightarrow candidati_A = \{\{A\}, \{C\}\}$$

$$B \rightarrow candidati_B = \{\{B\}\}$$

A questo punto facciamo tutte le possibili combinazioni di candidati per ogni entità della congiunzione, ottenendo:

$$candidati_{(A \sqcap B)} = \{\{A, B\}, \{C, B\}\}$$

Candidati per una negazione Dato una negazione di una entità, la ricerca degli insiemi di candidati è così articolata:

1. tramite ragionamento automatico si vanno a cercare tutte le classi equivalenti o sottoclassi della negazione completa;
2. nel caso in cui l'argomento fosse una congiunzione o una disgiunzione, si compie un ulteriore passaggio: si va a "spostare" la negazione all'interno della congiunzione/disgiunzione tramite le regole di DeMorgan e si cercano eventuali candidati per la congiunzione/disgiunzione risultante.

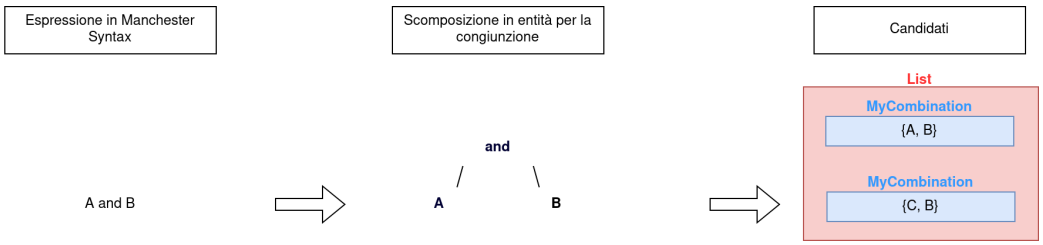
Mostriamo un esempio: supponiamo di avere le classi A e B e di avere la classe C definita come negazione di A e la classe D definita come negazione di B. Supponiamo che il nostro goal sia $G = \neg(A \sqcup B)$. Come primo passaggio chiediamo al resoner di trovare le classi equivalenti al goal. Successivamente, tramite le regole di DeMorgan, possiamo andare a trasformare il goal G in $((\neg A) \sqcap (\neg B))$. In questo modo, possiamo osservare che ci sarebbero due classi che, legate da congiunzione, vanno risolvere questo goal: C e D. Ovvero:

$$candidati_{(\neg A)} = \{\{C\}\}$$
$$candidati_{(\neg B)} = \{\{D\}\}$$

Di conseguenza, si può generare un candidato per questo goal:

$$candidati_{\neg(A \sqcap B)} = \{\{C, D\}\}$$

La funzionalità definita da noi restituirà, dato un goal, una lista di oggetti MyCombination in cui, ricordiamo, ogni MyCombination non è altro che candidato. Riportiamo la rappresentazione del processo di goal transformation per l'esempio di trasformazione del goal $(A \sqcap B)$ presentato negli esempi precedenti:



4.3 GoalSolver

Una volta aver determinato quali sono i candidati per risolvere il goal dato dall'utente bisogna eleggerne uno come risolutore. Nel caso avessimo candidati semplici, ovvero candidati composti da un solo concetto, controlliamo direttamente la consistenza di tali candidati semplici. In assenza di candidati semplici dobbiamo passare in rassegna i candidati complessi. In tal caso, dobbiamo selezionare un candidato e poi cercare di estrarre un nuovo concetto da tale candidato. La selezione del candidato complesso sarà presentata al termine di questo capitolo: ci sarà un modulo che, data una lista di candidati complessi, ci restituirà quello più promettente. Dato un candidato complesso, dobbiamo fondere i concetti al suo interno e vedere se questo concetto fusione soddisfa il goal. Per fare questo abbiamo definito il modulo **BaseGoalSolver** facendo riferimento alla classe astratta *GoalSolver*. Il compito di questo modulo è, dato un candidato complesso, generare un nuovo concetto in grado di risolvere il goal imposto dall'utente e che sia consistente con l'ontologia del sistema.

Il meccanismo di base di GOCCIOLA nel risolvere questo compito consiste nel generare scenari per determinare le caratteristiche del concetto fusione. La generazione degli scenari in questa tesi è stato oggetto di ottimizzazione.

Di seguito andiamo a presentare come questa funzionalità è stata implementata.

4.3.1 Goal Resolution

Dato un candidato complesso si tenta di combinare i concetti contenuti al suo interno e di ottenere un nuovo concetto che risolva il goal. Inoltre, nel fondere i concetti del candidato complesso, quelle che vengono anche determinate sono le proprietà tipiche del nuovo concetto. Se la fusione non avviene con successo, semplicemente il goal non è stato risolto e, di conseguenza, il sistema EDIFICA dovrà trovare un altro candidato complesso su cui tentare di fare la fusione. Il modulo di risoluzione del goal da noi definito al suo interno utilizza una serie di moduli di supporto che sono stati oggetto di ottimizzazione. Presentiamo brevemente come avviene concettualmente la fusione e poi presentiamo nei capitoli successivi come i moduli di supporto vanno a performare tale fusione.

Dato un candidato complesso $Candidate = \{C_1, ..., C_n\}$, cerchiamo di determinare un concetto fusione FC. Definiamo innanzitutto le meta-relazioni "rigide" per FC:

$$FC \equiv (C_1 \sqcap ... \sqcap C_n)$$

Osserviamo che possiamo già determinare un primo controllo per verificare se il concetto fusione esiste per $Candidate$: se la congiunzione dei concetti in $Candidate$ risulta essere inconsistente, allora possiamo affermare che per $Candidate$ non è possibile creare un concetto fusione.

Presentiamo un esempio per comprendere quando ciò avviene; supponiamo che:

$$Candidate = \{EssereVivente, EssereNonVivente\}$$

In questo caso, il modulo scarterebbe subito questo candidato, in quanto stiamo cercando di definire un nuovo concetto fusione FC come congiunzione di due classi disgiunte tra loro.

Se il concetto fusione FC risulta essere consistente dal punto di vista delle caratteristiche "rigide", passiamo a definire le sue caratteristiche più "flessibili": dobbiamo determinare le sue proprietà tipiche.

La fusione è una fase critica che coinvolge le caratteristiche tipiche dei concetti da fondere. Infatti, è importante dover definire quali sono le caratteristiche tipiche del concetto fusione per far sì che la fusione sia plausibile agli occhi di una persona e che soprattutto non sia inconsistente con l'intera conoscenza del sistema. La determinazione delle caratteristiche tipiche di un concetto fusione parte dal collezionare le caratteristiche tipiche dei concetti che lo compongono. Date queste caratteristiche tipiche si determinano gli scenari: sono combinazioni in cui per ogni caratteristica tipica viene deciso se essa vale o meno. Uno scenario può essere visto come una combinazione di valori di verità sulle caratteristiche tipiche. Se il valore di verità è Falso per una data caratteristica in un dato scenario, allora essa non sarà vera in tale scenario. Se il valore è invece Vero per una data caratteristica in un dato scenario, allora essa sarà vera in tale scenario. In sostanza, vengono create tutte le possibili combinazioni di presenza/assenza delle caratteristiche tipiche coinvolte e si va a scegliere, tra quelle consistenti con l'ontologia, quella più probabile.

Presentiamo un esempio di come concettualmente questo avviene e nelle prossime sezioni andiamo nel dettaglio a vedere come gli scenari sono definiti formalmente e come sono implementati. Supponiamo di dover fondere due concetti: A e B. Supponiamo inoltre di avere i seguenti assiomi di tipicalità:

$$Axiom_1 : \ T(A) \sqsubseteq C : 0.7$$

$$Axiom_2 : \ T(A) \sqsubseteq D : 0.82$$

$$Axiom_3 : \ T(B) \sqsubseteq F : 0.55$$

Nel fondere A e B, andiamo a definire quali sono le proprietà tipiche del concetto fusione. Per fare ciò, enumeriamo tutti i possibili scenari. Avendo 3 assiomi, i possibili scenari sono 8, ciascuno corrispondente a una diversa combinazione di verità/falsità di ciascun assioma. Otteniamo dunque:

$$Axiom_1, Axiom_2, Axiom_3 \qquad Axiom_1, Axiom_2, \neg Axiom_3$$

$$Axiom_1, \neg Axiom_2, Axiom_3 \qquad Axiom_1, \neg Axiom_2, \neg Axiom_3$$

$$\neg Axiom_1, Axiom_2, Axiom_3 \qquad \neg Axiom_1, Axiom_2, \neg Axiom_3$$

$$\neg Axiom_1, \neg Axiom_2, Axiom_3 \qquad \neg Axiom_1, \neg Axiom_2, \neg Axiom_3$$

Idealmente, questi scenari vengono ordinati per probabilità e si parte dal più probabile al meno probabile. Scelto uno scenario, si compie un test di consistenza. Se tale test viene superato, allora la fusione dei due concetti può essere considerata andata a buon fine e le proprietà tipiche del concetto fusione sono quelle dello scenario scelto. Se tale test non viene superato, allora si passa al prossimo scenario in ordine decrescente di probabilità.

Se il candidato non riesce a risolvere il goal, si passerà ad un altro candidato tra quelli disponibili. Ricordiamo che la fusione dei concetti presenti in un candidato non ha successo quando:

1. i concetti che formano il candidato sono inconsistenti tra loro se uniti tra loro da congiunzione: il concetto fusione è inconsistente dal punto di vista delle caratteristiche "rigide";

2. i concetti che formano il candidato sono consistenti tra loro se uniti tra loro da congiunzione, ma non esiste uno scenario consistente che esprime le caratteristiche "rilassate" del concetto fusione.

4.4 Scenario Management e Consistenza di Concetti

Per il modulo di risoluzione del goal sono stati definiti una serie di moduli di supporto, ognuno che svolge uno specifico compito. Il modulo BaseGoalSolver, dato in input un candidato, ovvero un insieme di concetti, utilizza il modulo **ConceptScenarioBuilder**, il cui compito è generare mano a mano gli scenari di tale insieme di concetti ed eleggere lo scenario più plausibile che rappresenterà quali sono le caratteristiche tipiche del concetto derivato dalla fusione dei concetti nel candidato. Inoltre questo modulo di supporto controllerà se lo scenario generato è consistente con la ontologia tramite il modulo di controllo della consistenza BaseOntologyChecker. Presentiamo ora il modulo **ConceptScenarioBuilder** e poi esploriamo le due funzionalità su cui si basa, ciascuna implementata in un modulo a parte:

- controllo a priori delle consistenze tra concetti: dato il candidato, si prendono tutti gli assiomi di tipicalità coinvolti. Di tali assiomi se ne prende il corpo e si controllano quali corpi sono in conflitto così da poter stabilire quali assiomi di tipicalità sono potenzialmente oppure sicuramente in conflitto;

- Produzione efficiente degli scenari: tramite la conoscenza delle consistenze tra gli assiomi tipicalità è possibile generare gli scenari consistenti senza doverli creare tutti e controllarli uno per uno tramite ragionamento automatico.

4.4.1 Gestione del candidato

Il modulo ConceptScenarioManager ha il compito di andare a compiere la fusione dei concetti contenuti nel candidato in input e di controllare che tale concetto fusione sia consistente con l'ontologia del sistema. Per compiere la fusione, se questa è possibile, dobbiamo definire il concetto di scenario. Dato un insieme di concetti $Concepts$, per ciascun concetto possiamo prendere gli assiomi di tipicalità ottenendo l'insieme $T_{Concepts}$, ovvero l'insieme di tutti quegli assiomi di tipicalità che hanno come testa uno dei concetti di $Concepts$:

$$T_{Concepts} = \{axiom_t | head(axiom_t) \in Concepts\}$$

Dato un assioma di tipicalità possiamo definire una tripla, che chiameremo scenario element (SE), con la seguente forma:

$$SE = (body(axiom_t),\ TRUE,\ prob(axiom_t))$$

Chiameremo questa forma di SE la forma canonica di uno scenario element. Tale forma rappresenta la verità di un dato assioma di tipicalità, in virtù del secondo valore della tripla, che vale $TRUE$. Dato un assioma di tipicalità, possiamo definire anche un SE in forma alternativa, che rappresenta la falsità dell'assioma:

$$SE = (body(axiom_t),\ FALSE,\ 1 - prob(axiom_t))$$

Dato un assioma di tipicalità, possiamo dunque sempre definire due scenario element: $SE_{canonical}$ e $SE_{alternative}$.

Scelta una sequenza (dunque dotata di ordine) composta da tutti gli assiomi di tipicalità in $T_{Concepts}$ senza ripetizioni possiamo definire ora il concetto di scenario.

$$s :=< at_0, at_1, ..., at_n > con\ at_i\ \in T_{Concepts}\ e\ |T_{Concepts}| = n$$

Uno scenario, data la sequenza di assiomi s, consiste nell'assegnare a ciascun assioma uno scenario element, sia esso uno scenario element in forma canonica che uno sceario element in forma alternativa.

$$scenario =< se_0, se_1, ..., se_n > con\ se_i\ \in \{SE_{canonical}(at_i), SE_{alternative}(at_i)\}$$

Come si può osservare, dato un insieme di assiomi di tipicalità $T_{Concepts}$ composto da n assiomi di tipicalità, i possibili scenari generabili sono 2^n. Inoltre, dato uno scenario è possibile associargli un valore di plausibilità, moltiplicando le probabilità di ciascun scenario element che lo compone. Possiamo quindi ordinare l'insieme degli scenari generabili da $T_{Concepts}$ per probabilità.

Quello che essenzialmente fa il modulo ConceptScenarioBuilder è:

1. prendere ogni possibile coppia di concetti appartenenti al candidato, ottenere il corpo degli assiomi di tipicalità di tali concetti e confrontarli tra loro per determinare se i due corpi possono essere in conflitto. Facendo ciò si definisce un grafo dei conflitti il cui compito è quello di rappresentare quali corpi sono in conflitto con altri corpi. Tale informazione sarà utilizzata in fase di generazione degli scenari per scartare a priori determinati scenari;

2. prendere per ogni concetto del candidato gli assiomi di tipicalità e per ciascuno assioma definire il corrispondente scenario element. Dati tutti gli scenario element, si calcola lo scenario più probabile: per ciascuno scenario element si prende quello, tra la forma canonica e quella alternativa, con probabilità maggiore;

3. dati il grafo dei conflitti e lo scenario più probabile, si vanno a definire tutti i possibili scenari consistenti ordinati in maniera decrescente per probabilità dello scenario;

4. controllare ogni scenario generato in ordine decrescente di probabilità finchè non se ne trova uno che sia non banale e che, tramite ragionamento automatico, sia consistente con l'ontologia.

Nella versione originale di GOCCIOLA questa procedura di definizione degli scenari veniva già fatta, ma è stata oggetto di ottimizzazione in questa tesi. Infatti, nella versione originale venivano definiti tutti i possibili scenari e controllati uno ad uno tramite ragionamento automatico, risultando in un grande costo computazionale. Nella versione di EDIFICA, invece, sono state adottate tecniche che a priori permettono di andare ad eliminare sin da subito scenari palesemente incoerenti ed è stata definita una procedura di ordinamento a priori degli scenari. In questo modo, ogni scenario è oggetto di ragionamento automatico solo se necessario, ovvero solo se gli scenari che lo precedono, ovvero con probabilità maggiore, sono risultati incoerenti oppure troppo banali. Riassumendo: in GOCCIOLA ogni scenario generato è soggetto al controllo del reasoner in maniera sistemica. In EDIFICA vi è un primo filtro in cui vengono eliminati sin da subito determinati scenari e su quelli rimanenti viene eventualmente compiuto il ragionamento automatico. Ciò significa che, nel caso pessimo in cui nessuno scenario soddisfi la consistenza con l'ontologia, EDIFICA dovrà controllare ogni scenario possibile, ma questo solo nel caso pessimo. Nel caso di GOCCIOLA, invece, si dovranno controllare tutti i possibili scenari sempre, sia nel caso ottimo in cui il primo scenario più probabile è consistente con la base di conoscenza, sia nel caso pessimo in cui nessuno scenario lo è.
Nello specifico, i punti 1. e 3. sono i passaggi che ci permettono di ottimizzare il processo presentato in GOCCIOLA per mezzo della definizione e dell'utilizzo di un grafo dei conflitti.

Inoltre, come si può osservare dalla definizione data, abbiamo definito il concetto di scenario riferendoci a un candidato di n concetti: la nuova versione del software rilassa il vincolo sul numero di concetti coinvolti nella fusione, che da due passa a un generico n.

4.4.2 Consistency Graph

Il compito di definire quali assiomi di tipicalità sono in conflitto o meno per poter andare ad escludere a priori determinati scenari è stato implementato in un modulo di utilità chiamato **consistency_checker3**.
Per trattare ciò che tale modulo va a fare presentiamo cosa intendiamo per avere due concetti in conflitto.

Per definire la nozione di concetti in conflitto andiamo a definire ricorsivamente la nozione di derivazione di concetto. Intuitivamente, diciamo che da un concetto C possiamo derivare un concetto C' se C è discendente di C', oppure se è equivalente a C'. Formalmente definiamo induttivamente questa relazione di derivazione e affermiamo che questa è definita sia per i concetti presenti nella ontologia che per i concetti definiti da class expression, ovvero quelle che nei capitoli precedenti abbiamo definito classi "anonime". Di conseguenza, col termine "concetto" ci riferiremo sia a concetti definiti nella ontologia, sia a concetti definiti da una class expression, ovvero dato un concetto C esso può:

- essere definito come concetto della ontologia;
- essere definito a partire da una espressione logica.

Presentiamo degli esempi per mostrare formalmente come esprimere un concetto.

Concetto di una ontologia Supponiamo di avere una ontologia e di definire al suo interno il concetto di Artista. Il concetto Artista allora è un concetto della nostra base di conoscenza:

$$Artista \in Concepts_{ONTOLOGY}$$

Concetto di una ontologia equivalente a una class expression Supponiamo di definire la classe Pittore come equivalente alla classe degli artisti che creano solo dipinti. Definiamo di conseguenza un concetto che rappresenta una classe "anonima":

$$C \mathrel{\widehat{=}} (Artista \sqcap (crea\ only\ Dipinto))$$

E poi definiamo un concetto appartenente alla ontologia come equivalente a tale classe "anonima":

$$(Pittore \equiv C) \wedge (Pittore \in Concepts_{ONTOLOGY})$$

Una cosa analoga può essere fatta se vogliamo definire un concetto della ontologia come sottoclasse di una class expression.

Prima di fornire la definizione induttiva ricordiamo che, per come sono state definite le meta-relazioni **is-a** ed **eq-to**, possiamo dire che:

$$A \equiv B \iff A \sqsubseteq B \wedge B \sqsubseteq A$$

Definizione di derivazione di concetti:

Caso base

$$C \vdash C_1 \iff C \sqsubseteq C_1$$

Caso induttivo

$$C \vdash C_1 \iff \exists C_2 (C \sqsubseteq C_2 \wedge C_2 \vdash C_1)$$

In virtù della definizione di **eq-to**, possiamo affermare che, essendo $C \equiv C$ e dunque $C \sqsubseteq C$, $C \vdash C$.

Questa relazione di derivabilità può essere estesa anche alle proprietà.

Caso base

$$P \vdash P_1 \iff P \sqsubseteq P_1$$

Caso induttivo

$$P \vdash P_1 \iff \exists P_2 (P \sqsubseteq P_2 \wedge P_2 \vdash P_1)$$

Data la definizione di derivabilità per un singolo concetto, andiamo a definire il concetto di inconsistenza. Diciamo che una coppia di concetti (C_1,C_2) genera inconsistenza quando questi due concetti, se legati da una congiunzione, generano una classe inconsistente, ovvero equivalente alla classe *owl.Nothing*. Indicheremo come segue la inconsistenza:

$$(C_1, C_2) \vdash \bot$$

Per come abbiamo definito la coppia (C_1,C_2), essendo che la inconsistenza deriva dalla loro congiunzione, possiamo dire che:

$$(C_1, C_2) \vdash \bot \iff (C_2, C_1) \vdash \bot$$

Definiamo di seguito in maniera induttiva le regole di derivazione di inconsistenza tra due concetti:

Caso base: due concetti C_1 e C_2 sono in conflitto se esiste una coppia di antenati, un antenato di C_1 e un antenato di C_2, che sono disgiunti tra loro.

$$(C_1, C_2) \vdash \bot \iff \exists C_1' \exists C_2' (C_1 \vdash C_1' \wedge C_2 \vdash C_2' \wedge disjoint(C_1', C_2'))$$

Caso Or: supponiamo che nella analisi del primo concetto accada che da esso possiamo derivare un concetto definito a partire da una class expression, in particolare da un or. Si genera inconsistenza con il secondo concetto se e solo se tutti i componenti dell'**or** generano inconsistenza col secondo concetto.

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_2 \vdash C_2') \wedge (C_1' \widehat{=} (E_1 \sqcup E_2 \sqcup ... \sqcup E_n)) \wedge \\ \forall E_A \in \{E_1, E_2, ..., E_n\} (C_1'' \widehat{=} E_A \wedge (C_1'', C_2') \vdash \bot)$$

Caso And: supponiamo che nella analisi del primo concetto accada che da esso possiamo derivare un concetto definito a partire da una class expression, in particolare da un **and**. Si genera inconsistenza con il secondo concetto se e solo se esiste almeno un componente dell'and che genera inconsistenza col secondo concetto.

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_2 \vdash C_2') \wedge (C_1' \widehat{=} (E_1 \sqcap E_2 \sqcap ... \sqcap E_n)) \wedge \\ \exists E_A \in \{E_1, E_2, ..., E_n\} (C_1'' \widehat{=} E_A \wedge (C_1'', C_2') \vdash \bot)$$

Caso Not: nel caso in cui uno dei due concetti consiste in una negazione, bisogna controllare diverse combinazioni. Supponiamo che il primo concetto sia quello con la negazione:

- se il secondo concetto consiste nell'argomento della negazione, allora abbiamo una inconsistenza;
- se il primo concetto ha come argomento una congiunzione, si utilizzano le regole di DeMorgan per trasformare la congiunzione in disgiunzione in cui ogni entità della disgiunzione ha una negazione al suo interno;
- se il primo concetto ha come argomento una disgiunzione, si utilizzano le regole di DeMorgan per trasformare la disgiunzione in congiunzione in cui ogni entità della congiunzione ha una negazione al suo interno;
- se il primo concetto ha come argomento una class restriction di tipo **only**, ovvero con quantificatore universale, si trasforma il quantificatore universale in esistenziale e l'argomento della quantificazione eredita la negazione;
- se il primo concetto ha come argomento una class restriction di tipo **some**, ovvero con quantificatore esistenziale, si trasforma il quantificatore esistenziale in universale e l'argomento della quantificazione eredita la negazione.

Il caso della negazione può essere oggetto di ulteriori analisi, in quanto non viene approfondito il caso in cui entrambi i concetti consistono in una negazione. Questo rende la nostra trattazione delle inconsistenze incompleta, ma comunque

corretta, infatti il compito del modulo di gestione delle inconsistenze è quello di andare ad eliminare gli scenari più comuni e palesemente errati, lasciando quindi spazio a implementazioni future di controlli più approfonditi.

Esprimiamo formalmente la nozione di inconsistenza per la negazione riportate qui sopra:

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_1' \widehat{=} \neg C2)$$

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_1' \widehat{=} \neg(E_1 \sqcup E_2 \sqcup ... \sqcup E_n)) \wedge$$
$$(C_1'' \widehat{=} (\neg E_1 \sqcap \neg E_2 \sqcap ... \sqcap \neg E_n)) \sqcap (C_1'', C_2) \vdash \bot)$$

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_1' \widehat{=} \neg(E_1 \sqcap E_2 \sqcap ... \sqcap E_n)) \wedge$$
$$(C_1'' \widehat{=} (\neg E_1 \sqcup \neg E_2 \sqcup ... \sqcup \neg E_n)) \wedge (C_1'', C_2) \vdash \bot)$$

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_1' \widehat{=} \neg(P \; only \; E1)) \wedge$$
$$(C_1'' \widehat{=} P \; some \; (\neg E1)) \wedge (C_1'', C_2) \vdash \bot$$

$$(C_1, C_2) \vdash \bot \iff (C_1 \vdash C_1') \wedge (C_1' \widehat{=} \neg(P \; some \; E1)) \wedge$$
$$(C_1'' \widehat{=} P \; only \; (\neg E1)) \wedge (C_1'', C_2) \vdash \bot$$

Caso restriction: Per quanto riguarda le class restriction, sono state definite due regole di inconsistenze che sono valide solo se entrambi i concetti risultano essere delle class restriction sulla stessa proprietà:

- se entrambe sono delle quantificazioni universali sulla stessa proprietà, allora i due concetti sono inconsistenti se gli argomenti delle rispettive quantificazioni lo sono;
- se un concetto è una quantificazione esistenziale e l'altra è una quantificazione universale sulla stessa proprietà, i due concetti sono inconsistenti se gli argomenti delle rispettive quantificazioni lo sono;

Anche in questo caso, non è stata fatta una trattazione approfondita che copre ogni possibile caso. Per esempio, in questo caso è stata posta come fissa la proprietà per le due restrizioni, ma si possono fare degli ulteriori studi su casi in cui abbiamo una quantificazione su una proprietà P e una quantificazione su una sotto-proprietà P' di P. Un altro argomento di approfondimento potrebbe essere quello di coinvolgere un concetto che è una restrizione e l'altro concetto che è una classe o una disgiunzione o una congiunzione o una negazione.

Riportiamo formalmente le regole di inconsistenza per le class restriction di seguito:

$$(C_1, C_2) \vdash \bot \iff C_1 \widehat{=} (P \; only \; E_1) \wedge C_2 \widehat{=} (P \; only \; E_2) \wedge C_1' \widehat{=} E_1 \wedge C_2' \widehat{=} E_2 \wedge (C_1', C_2') \vdash \bot$$

$$(C_1, C_2) \vdash \bot \iff C_1 \widehat{=} (P \; only \; E_1) \wedge C_2 \widehat{=} (P \; some \; E_2) \wedge C_1' \widehat{=} E_1 \wedge C_2' \widehat{=} E_2 \wedge (C_1', C_2') \vdash \bot$$

Osserviamo come la commutatività degli elementi della coppia ci permetta di definire queste regole di derivazione ponendo l'attenzione sul primo elemento della coppia: C_1. Se l'elemento interessante della derivazione fosse C_2, ovvero il secondo elemento della coppia, tramite la commutatività possiamo scambiare di posizione C_1 e C_2 e trattare C_2 come se fosse il primo elemento dalla coppia:

$$(C_1, C_2) \vdash \bot \iff (C_2, C_1) \vdash \bot$$

Mostriamo ora come queste regole formali sono state implementate all'interno del nostro sistema. All'interno di consistency_checker3 sono state definite due funzioni mutuamente ricorsive.

La prima funzione definita è explore(), che prende in input due concetti c_1 e c_2. Il compito di questa funzione è di confrontare c_1 e ogni suo possibile antenato con c_2.

La seconda funzione definita è explore_max(), che prende in input due concetti c_1 e c_2 e quello che sostanzialmente va a fare è:

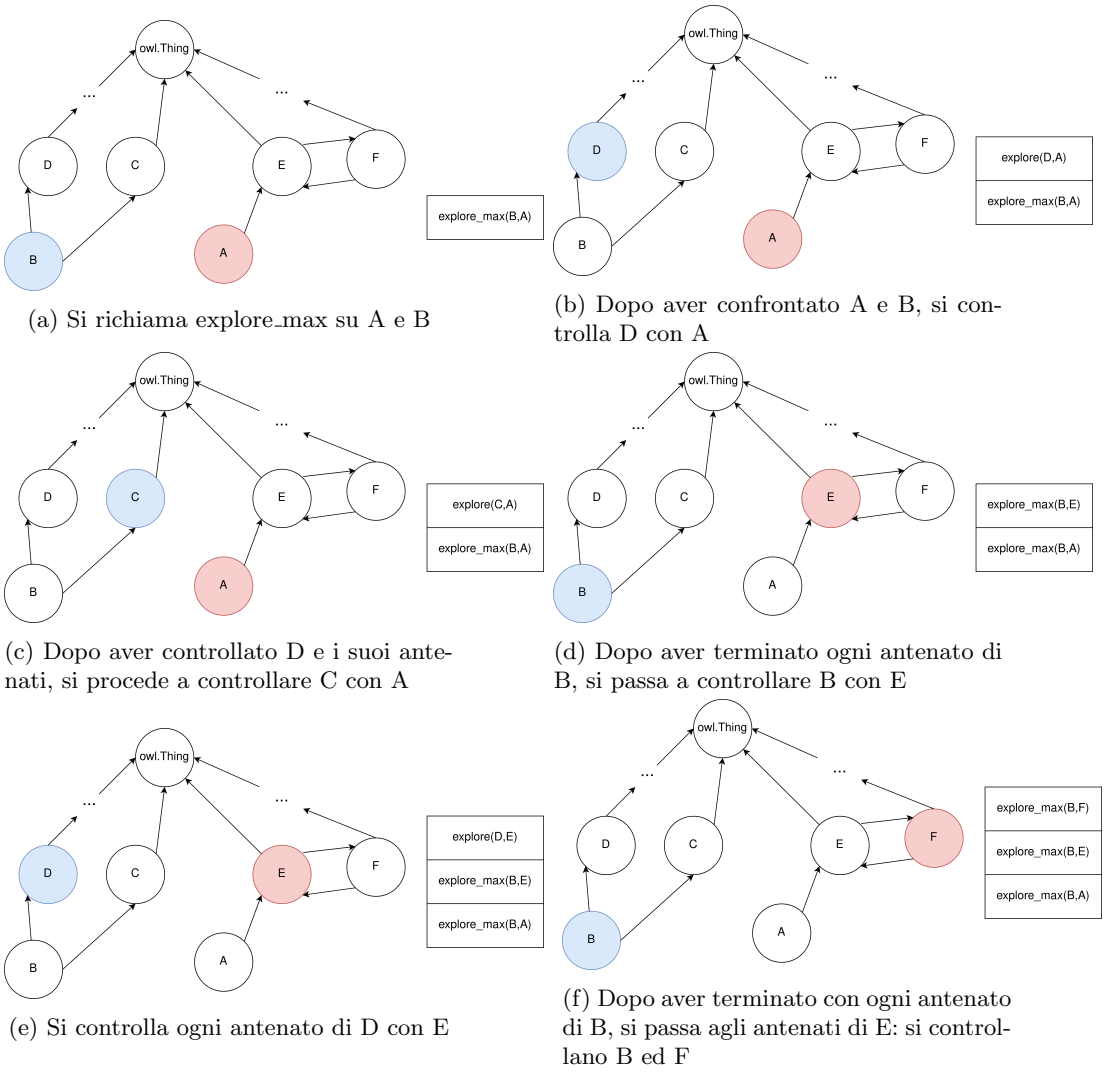
1. tramite la definizione di explore(), compie una ricerca di conflitti tra c_2 e tutti gli antenati di c_1 (c_1 compreso);
2. dopo aver terminato il punto precedente si prendono i diretti "parenti" di c_2, ovvero le classi equivalenti a c_2 e le super-classi di c_2. In maniera ricorsiva si applica explore_max() su c_1 e su ciascun "parente" di c_2.

Come si può evincere, explore_max() utilizza al proprio interno explore(), ma anche explore() usa al proprio interno explore_max(): questo avviene nel caso delle ultime due regole di inconsistenza presentate in precedenza, ovvero nel caso in cui c_1 e c_2 siano entrambe delle class restriction definite sulla stessa proprietà P. In tal caso, come specificato nella regola di inconsistenza, si devono andare a controllare gli argomenti delle due class restriction: questo consiste nel prendere arg_{c1} e arg_{c2} e di compiere un controllo completo dall'inizio: utilizziamo explore_max() su arg_{c1} e arg_{c2}. In questo senso, le due funzioni qui definite explore() ed explore_max() sono mutuamente ricorsive: ci sono casi in cui la prima chiama la seconda e successivamente la seconda chiama la prima.

Per entrambe le funzioni è stato definito inoltre un meccanismo per evitare definizioni cicliche di concetti. Mostriamo con un esempio perchè tale meccanismo è essenziale per le due funzioni ricorsive. Supponiamo di applicare explore_max() su due classi: A e B. Supponiamo inoltre che B sia equivalente a una classe C. Per come abbiamo definito la nozione di inconsistenza, dobbiamo poi esplorare tramite explore_max() A e C. Così facendo, dobbiamo esplorare anche le classi equivalenti di C, in cui ritroviamo B: ci ritroveremo a rimbalzare tra B e C in eterno!
Per fare ciò, semplicemente teniamo traccia dei concetti che stiamo attualmente visitando, così, quando arriveremo a C, non esploreremo B.

Tramite il modulo da noi definito ora abbiamo la possibilità di definire un grafo dei conflitti: ogni nodo rappresenta un concetto, sia esso una classe definita nella ontologia o una class expression. Per ogni nodo abbiamo la nozione di sottoclasse e ogni nodo ha una lista di concetti con cui entra in conflitto. Tale grafo sarà utilizzato per andare a definire quali informazioni tipiche possono coesistere all'interno di uno scenario e quali no.

Nella pagina successiva presentiamo un esempio di come le due funzioni explore_max() ed explore() collaborano nell'andare a ricercare le inconsistenze. Nell'esempio in questione vengono riportate le classi A e B e i loro diretti parenti: per A abbiamo E ed F, con E ed F equivalenti, mentre per B abbiamo C e D.
Nella rappresentazione riportata di seguito vengono riportati alcuni dei passaggi di esecuzione del modulo consistency_checker3 e per ciascun passaggio sono evidenziati i concetti presi in esame in quel passaggio e sulla destra di ciascun passaggio viene riportato lo stack con le chiamate ricorsive.



(a) Si richiama explore_max su A e B

(b) Dopo aver confrontato A e B, si controlla D con A

(c) Dopo aver controllato D e i suoi antenati, si procede a controllare C con A

(d) Dopo aver terminato ogni antenato di B, si passa a controllare B con E

(e) Si controlla ogni antenato di D con E

(f) Dopo aver terminato con ogni antenato di B, si passa agli antenati di E: si controllano B ed F

Figura 1: Fasi del processo di analisi delle inconsistenze

Come si può evincere dall'esempio, le relazioni di equivalenza tra due concetti sono trasformate in relazioni di sottoclasse: i concetti E ed F sono equivalenti, allora E è sottoclasse di F ed F è sottoclasse di E.

4.4.3 Scenario Management

Per quanto riguarda la generazione e la gestione degli scenari, è stato definito un modulo che, dato lo scenario più probabile e il grafo dei conflitti, va a generare tutti i possibili scenari consistenti in ordine decrescente di probabilità. L'attività principale di questo modulo sta nel creare gli scenari in maniera efficiente e di utilizzare le informazioni di consistenza tra i concetti per evitare di creare scenari che sono palesemente inconsistenti.

Per implementare questo modulo sono state create due strutture di supporto: Node e ScenarioNode.

Classe Node: rappresenta uno scenario element, ovvero una tripla:

$$(concetto, valore\ di\ verità, probabilità)$$

All'interno di questa classe non vi è solo l'informazione riguardante lo scenario element, ma anche una informazione riguardante i legami che intercorrono tra i vari scenario elements. Le istanze della classe Node infatti sono dotate anche dell'attributo **coherence**, che corrisponde a un dizionario Python al cui interno sono state definite tre chiavi:

- *opposite*: all'interno di questa categoria troviamo tutti quei Node il cui concetto risulta essere in conflitto con il concetto del Node attuale;
- *both_false*: all'interno di questa categoria troviamo tutti quei Node il cui valore di verità deve essere falso se il valore di verità del Node attuale è falso;
- *both_true*: all'interno di questa categoria troviamo tutti quei Node il cui valore di verità deve essere vero se il valore di verità del Node attuale è vero.

Dal momento che la classe Node rappresenta uno scenario element, da esso possiamo ottenere sia la forma canonica che la forma alternativa. Ricordiamo che la forma canonica è quella in cui il valore di verità è $TRUE$, mentre la forma alternativa è quella in cui il valore di verità è $FALSE$.

Infine, la classe Node è stata dotata di una funzione che si focalizza sul restituire, in base alla richiesta dell'utente, quella forma, sia essa canonica oppure alternativa, di probabilità maggiore o di probabilità minore, a seconda della richiesta. L'input a tale funzione può essere 0, che significa che l'utente vuole la forma con probabilità maggiore, oppure 1, che significa che l'utente vuole la forma con probabilità minore.

Classe ScenarioNode: rappresenta quella che possiamo definire una classe Wrapper: preso un Node, che ha le informazioni circa uno scenario element e le sue relazioni con altri scenario elements, vi aggiunge una funzionalità inizialmente non definita in Node. In Node, data una opzione, che ci rappresenta la

volontà di avere la forma più probabile o quella meno probabile, ci viene restituita la forma corrispondente dello scenario element. In ScenarioNode è stata definita una funzione "inversa": dato il valore di verità, ci viene detta quale è l'opzione corrispondente, se quella della forma con probabilità maggiore o quella della forma con probabilità minore.

Presentati gli elementi fondamentali del modulo andiamo a vedere come vengono generati i vari scenari consistenti in ordine di probabilità decrescente.

Il modulo ScenarioManager prende in input lo scenario più probabile e il grafo dei conflitti dei concetti. Uno scenario viene codificato come una lista di triple in cui il primo elemento è il corpo dell'assioma di tipicalità, il secondo è il valore di verità e il terzo è la probabilità. In sostanza ogni tripla ha la seguente forma:

$$(body_{axiom}, truth\_value, probability)$$

Dato lo scenario più probabile in input, il modulo definisce un Node e uno ScenarioNode per ogni tripla dello scenario. Successivamente, tramite il grafo dei conflitti si vanno a definire le relazioni che intercorrono tra i vari Node. Tali relazioni vengono definite come segue:

1. per ogni tripla dello scenario si prende il $body_{axiom}$ e si cercano nel grafo dei conflitti quei body riferiti ad altri assiomi di tipicalità con cui è in conflitto. Per ogni $body_{conflict}$ in conflitto con $body_{axiom}$ si va ad aggiungere $body_{conflict}$ nella categoria *opposite* del Node che si riferisce a $body_{axiom}$ e viceversa;

2. se ci sono delle triple con lo stesso $body_{axiom}$, allora ci ritroviamo ad avere più Node con stesso body, di conseguenza questi nodi devono variare in maniera coerente: o sono tutti con $truth\_value$ a True oppure lo sono tutti con $truth\_value$ a False: si aggiornano le categorie *both_true* e *both_false* per ciascun Node;

3. si creano tutte le possibili coppie ($body_1$,$body_2$) tra gli assiomi di tipicalità presenti nello scenario e si controlla se uno dei due è sottoclasse dell'altro. In tal caso vanno poste delle relazioni per mantenere la coerenza. Supponiamo nel nostro caso che $body_1$ sia sottoclasse di $body_2$, allora si aggiungeranno due relazioni tra i rispettivi Node:

   - se $body_1$ è vero, allora anche $body_2$ deve esserlo: nel Node riferito a $body_1$ si aggiunge nella categoria *both_true* il Node riferito a $body_2$:
   - se $body_2$ è falso, allora anche $body_1$ deve esserlo: nel Node riferito a $body_2$ si aggiunge nella categoria *both_false* il Node riferito a $body_1$.

Dopo questo primo passaggio di inizializzazione del modulo, presentiamo ora le quattro funzioni il cui compito è quello di generare gli scenari consistenti in ordine decrescente di probabilità: generateNextScenario, update, merge e my_sum.

generateNextScenario La seguente funzione è quella che possiamo considerare come principale: per generare gli scenari consistenti ordinati per probabilità decrescente si richiama questa funzione, mentre le altre tre saranno utilizzate come funzioni di supporto.

In input prende una lista di valori interi chiamata *bits* e un intero chiamato *pos*. L'idea è la seguente: dato lo scenario più probabile, che avrà forma $< se_1, se_2, ..., se_n >$, definiamo *bits* come una lista lunga n in cui, per ogni posizione i della lista abbiamo uno dei seguenti valori:

- 0: si prende, dello scenario element di posizione i, la forma più probabile;
- 1: si prende, dello scenario element di posizione i, la forma meno probabile;
- -1: lo scenario element di posizione i non è ancora stato valutato.

Questa lista rappresenta uno scenario che può essere incompleto, ovvero uno scenario in cui alcuni scenario elements sono stati fissati e altri non ancora, oppure completo, ovvero con tutti gli scenario elements che sono stati fissati. Da uno scenario incompleto possiamo estrarre diversi scenari completi, andando a fissare a 0 oppure a 1 quegli scenario elements che hanno valore -1 in *bits*. Ricordiamo inoltre che il modulo ScenarioManager fa corrispondere a ogni scenario element un oggetto ScenarioNode, il quale contiene le informazioni di consistenza con altri scenario element.

Il secondo parametro della funzione generateNextScenario, ovvero *pos*, indica invece l'attuale posizione che si sta analizzando per generare gli scenari.

Il compito della funzione generateNextScenario è la seguente: dato in input uno scenario, rappresentato da *bits*, restituisce in output la lista degli scenari completi coerenti generabili da *bits*.

Per fare ciò, questa funzione agisce come segue: dati *bits* e *pos*, analizza lo scenario rappresentato di *bits* ponendo l'attenzione sullo ScenarioNode nella posizione *pos*. Una volta aver terminato questa analisi si passa, in maniera ricorsiva, alla posizione successiva, ovvero $pos + 1$. Vediamo di seguito come viene fatta l'analisi di *bits* in riferimento allo ScenarioNode in *pos*.

Possiamo scomporre l'analisi in due casi.

- se *pos* eccede la dimensione di *bits*, allora abbiamo terminato gli scenario element da analizzare e restituiamo, come scenari possibili, la lista singoletto in cui all'interno troviamo *bits*. In sostanza questo è il caso in cui abbiamo in input uno scenario completo e abbiamo ecceduto la posizione di ricerca;
- se *pos* non eccede la dimensione di *bits*, vuol dire che abbiamo un possibile scenario element da valutare e vuol dire anche che abbiamo in input uno scenario che può essere completo, ma ancora da verificare, oppure incompleto. Dato uno scenario incompleto possiamo derivare tanti possibili scenari completi. In tal caso distinguiamo due sottocasi:
  - se lo scenario element specificato da *pos* risulta essere già stato fissato (avrà dunque valore 0 oppure 1), si deve controllare che lo scenario

rappresentato da *bits* sia coerente: uno scenario, sia esso completo che incompleto, è coerente se lo è prendendo in considerazione gli scenario element che sono stati fissati. Per compiere questi controlli di coerenza sono state definite la funzione di supporto **update** e la funzione di supporto **my_sum**, che verranno presentate di seguito. Se i controlli di queste funzioni vanno a buon termine, allora lo scenario è coerente e lo scenario element in *pos* è già fissato, dunque si può eseguire ricorsivamente generateNextScenario sulla posizione successiva. Se tali controlli non vanno a buon termine, allora vuol dire che nel fissare i valori dello scenario siamo arrivati a una situazione inconsistente e dunque il risultato è una lista vuota. Possiamo osservare che se abbiamo in input uno scenario incompleto la cui analisi di coerenza non porta a successo possiamo sin da subito affermare che tale scenario incompleto non genererà scenari completi, senza dover andare avanti nella generazione di scenari;

– se lo scenario element specificato da *pos* non è ancora stato fissato si generano due scenari, siano essi completi che incompleti: uno in cui esso sarà posto a 0, che chiameremo $bits_0$, e l'altro in cui sarà posto a 1, che chiameremo $bits_1$. Per ambo gli scenari si compieranno i controlli dovuti tramite **update** e **my_sum**: se questi hanno esito positivo, allora si mantiene lo scenario e si calcolano gli scenari completi che possiamo estrarre da tale esso richiamando ricorsivamente generateNextScenario sulla prossima posizione, altrimenti gli scenari completi estraibili dallo scenario corrispondono alla lista vuota, ovvero non ne esistono. Tramite le chiamate ricorsive otteniamo due liste di scenari completi coerenti: una riferita a quegli scenari completi generabili da $bits_0$ e l'altra riferita agli scenari completi generabili da $bits_1$. Date queste due liste possiamo fonderle e restituire la loro unione come lista di scenari completi generabili da *bits*. La fusione delle due liste avviene tramite la terza funzione di supporto da noi definita: **merge**.

Quando questa funzione viene richiamata da un modulo esterno a ScenarioManager, i parametri avranno i seguenti valori:

- *bits* sarà uno scenario incompleto in cui non è stato fissato nessuno scenario element: ovvero tutti i valori in *bits* saranno -1;
- *pos* rappresenta la prossima posizione con cui compiere l'analisi, dunque varrà 0, ovvero lo scenario element di prima posizione.

Presentiamo ora le tre funzioni di supporto: my_sum, update e merge.

my_sum La seguente funzione è di supporto e nasce per controllare la consistenza degli assiomi di tipicalità dati la lista *bits* e un nodo n. Preso n, che

rappresenta uno ScenarioNode di cui fissiamo il valore a $TRUE$, si prendono tutti i nodi con cui è in conflitto, ovvero quelli appartenenti alla categoria "opposite":

$$conflict\_nodes_n = \{node \in opposite(n)\}$$

Di questi nodi si selezionano quelli che in *bits* rappresentano la forma canonica, ovvero quella con valore di verità $TRUE$:

$$canonic\_conflict\_nodes_n = \{node \in conflict\_nodes_n \mid form(node) = canonical\}$$

Presi questi nodi, che rappresentano scenario elements in forma canonica, ovvero assiomi di tipicalità posti a $TRUE$ e in conflitto con n, dobbiamo controllare che, sommando le probabilità di ciascuno scenario element, la somma sia minore o uguale a 1.

Questa funzione sostanzialmente controlla che, arrivati ad n, ovvero allo scenario element di posizione *pos*, gli assiomi di tipicalità con corpo in conflitto n non creino una probabilità maggiore di 1, ovvero non creino uno scenario inconsistente. Questa funzione viene sempre utilizzata prima della funzione update per controllare che i conflitti riferiti ad n siano coerenti.

update La seguente funzione è di supporto e nasce per compiere una serie di controlli sulla consistenza dello scenario rappresentato da *bits* facendo riferimento allo scenario element di posizione *pos*. Tali controlli coinvolgono, in input, cinque parametri:

- n: lo ScenarioNode che rappresenta lo scenario element in *pos*;
- opt: l'opzione che specifica quale forma prendere di n. Tale valore è sempre fissato a 0 o a 1;
- k: lo ScenarioNode di cui controllare la consistenza con n;
- y: la posizione dello ScenarioNode k all'interno dello scenario;
- bits: la lista di bits che rappresenta lo scenario incompleto.

La funzione update viene richiamata in due condizioni:

1. stiamo analizzando uno scenario, sia esso completo che incompleto, e la posizione dell'attuale scenario element risulta essere già stato fissato, di conseguenza dobbiamo controllare se lo scenario può essere potenzialmente inconsistente;
2. stiamo analizzando uno scenario, sia esso completo che incompleto, e la posizione dell'attuale scenario element non è stato fissato, allora creiamo due scenari incompleti alternativi, uno in cui tale scenario element è posto a 0 e l'altro in cui è posto a 1 e controlliamo la consistenza di ambo gli scenari generati.

Il controllo effettivo viene fatto confrontando il nodo che rappresenta lo scenario element attuale con ciascun altro nodo dello scenario: per questo motivo in input abbiamo bisogno delle informazioni sul nodo attuale, ovvero n e opt, e le informazioni sull'altro nodo da controllare, ovvero k ed y. Quando confrontiamo i due nodi n e k possono verificarsi tre condizioni:

1. il nodo n e il nodo k sono entrambi con valore fissato e controllando le loro consistenze, sono consistenti tra loro. In questo caso la funzione update si limita a restituire come valore di ritorno True, indicando che n e k sono consistenti tra loro;

2. il nodo n e il nodo k sono entrambi con valore fissato e controllando le loro consistenze, sono inconsistenti. In questo caso la funzione update si limita a restituire come valore di ritorno False, indicando che n e k sono inconsistenti tra loro. Un esempio di inconsistenza tra n e k può essere il seguente: n è fissato a 0 e k è fissato a 1. Scopriamo che k fissato a 1 significa che k ha valore di verità True, mentre n fissato a 0 ha valore di verità False, ma abbiamo che tra n e k intercorre una relazione da n a k di tipo *both_false*, dunque se n è falso, allora anche k dovrebbe esserlo;

3. il nodo n è, per definizione, fissato a un valore, ma il nodo k può non esserlo, ovvero può avere valore -1 (essendo che siamo in uno scenario incompleto). Se sappiamo che tra n e k intercorre una relazione, sia essa di *both_true* o di *both_false*, possiamo fissare subito il valore di k, andando a tagliare via delle opzioni per i prossimi scenari: sappiamo che per k può valere solo un valore e non più due come nel caso standard. In questo caso aggiorniamo il valore del nodo k andando ad aggiornare la lista bits passata in input. In questa condizione la funzione update restituirà nuovamente True.

Nel controllare che i due nodi n e k siano consistenti viene utilizzata la funzione **my_sum**. Infatti, essa viene utilizzata prima di **update** per controllare che n sia consistente. Nella **update** viene utilizzata su k qualora k fosse in conflitto con n. Il perchè è il seguente: supponiamo che k ed n siano in conflitto, e che ad n venga assegnato il valore di verità $TRUE$. Essendo posto a $TRUE$, per k abbiamo ora il nodo n tra quei nodi che si aggiungono ai nodi con cui è in conflitto e che sono anche veri: avere n a $TRUE$ potrebbe generare delle inconsistenze se ci sono altri nodi con valore $TRUE$ in conflitto con k. Fatto ciò, la funzione compie i controlli sulle proprietà *both_true* e *both_false* andando a fissare, ove possibile, i valori di quei nodi che in *bits* non sono ancora stati fissati.

merge La seguente funzione è di supporto e serve per compiere la fusione di due liste già ordinate. Il risultato è la lista composta da tutti gli elementi delle due liste in input ed è ordinata.

Il codice della funzione è riportato di seguito.

```
def merge(self,list1,list2):
        p_l1 = 0
        p_l2 = 0

        p1 = -1
        p2 = -1

        res = []

        if p_l1 < len(list1):
            p1 = calculate_probability(list1[p_l1])
        if p_l2 < len(list2):
            p2 = calculate_probability(list2[p_l2])

        for i in range(0,len(list1 + list2)):
            if p1 >= p2:
                res.append(list1[p_l1])
                p_l1 = p_l1 + 1
                if p_l1 < len(list1):
                    p1 = calculate_probability(list1[p_l1])
                else:
                    p1 = -1
            else:
                res.append(list2[p_l2])
                p_l2 = p_l2 + 1
                if p_l2 < len(list2):
                    p2 = calculate_probability(list2[p_l2])
                else:
                    p2 = -1

        return res
```

Come possiamo osservare dal codice, oltre alle due liste di input $list1$ e $list2$, per ipotesi già ordinate, sono presenti anche due puntatori: p_l1 per la lista $list1$ e p_l2 per la lista $list2$. Inoltre, sono state definite due variabili $p1$, riferita a $list1$ e $p2$ riferita a $list2$. Infine, vi è una quinta variabile: res, che rappresenta la lista risultato.

Dimostriamo che se in input $list1$ e $list2$ sono ordinate, anche l'output res lo è. Partiamo dalla inizializzazione delle variabili. Abbiamo rispettivamente:

- p_l1 e $p1$: p_l1 è l'indice del prossimo elemento di $list1$ da analizzare, mentre $p1$ è la probabilità di tale elemento. Ogni elemento della lista è uno scenario, di conseguenza $p1$ è la probabilità dello scenario di $list1$ nella posizione indicata da $pl1$. Se la lista è vuota, $p1$ vale -1, ovvero una probabilità irrealizzabile, altrimenti varrà quanto la probabilità del primo elemento di $list1$;

- p_l2 e $p2$ la loro semantica è analoga a quella di p_l1 e $p1$, con la differenza che fa riferimento a $list2$;

- res: è la variabile al cui interno verrà memorizzata la lista risultante dalla concatenazione di $list1$ e $list2$, ma ordinata in ordine decrescente di probabilità. All'inizio tale lista è vuota.

Proseguendo nell'analisi del codice, possiamo osservare che è stato definito un ciclo for. Analizziamone prima il corpo: al suo interno vi è una istruzione if-then-else. Se la probabilità $p1$ è maggiore o uguale a quella $p2$, significa che lo scenario in posizione p_l1 ha probabilità maggiore o uguale di quella in p_l2. Di conseguenza si aggiunge lo scenario di indice p_l1 a res mettendolo in coda e si fa avanzare l'indice p_l1 in avanti di una posizione e si aggiorna di conseguenza $p1$ assegnando la probabilità del nuovo scenario se esso esiste, -1 altrimenti. Il ramo else è lo speculare del ramo then, in cui sostanzialmente si va ad aggiungere lo scenario in p_l2 a res (essendo quello di probabilità massima) e andando poi ad aggiornare p_l2 e $p2$ di conseguenza.

Analizzando il corpo del ciclo, possiamo affermare che esattamente uno tra p_l1 e p_l2 verrà sempre fatto avanzare di una posizione. Quando uno di questi arriva al fondo della lista a cui si riferisce, esso avrà la probabilità a cui è legata, sia essa $p1$ o $p2$, con valore -1 e quindi non avrà più possibilità di avere probabilità massima. Facendo partire l'iteratore i del ciclo for dal valore 0 al valore pari alla lunghezza delle due liste concatenate, siamo sicuri che così facendo, dato che esattamente un indice tra i due avanza sempre, arriveremo alla fine del ciclo che entrambi gli indici avranno appena raggiunto la fine della propria lista: p_l1 sarà pari alla lunghezza di $list1$ e p_l2 sarà pari alla lunghezza di $list2$ e di conseguenza $p1$ e $p2$ varranno -1.

Per affermare che questo codice alla fine ci restituisce una lista di scenari, in particolare tutti gli scenari in $list1$ e $list2$, ordinata in maniera decrescente di probabilità dobbiamo trovare una proprietà che produce questo ordinamento desiderato e che si mantiene per tutto il ciclo for: ci serve un invariante di ciclo. L'invariante in questione è il seguente:

$$\begin{gathered}\forall s \in res\ \forall s' \in list1[p\_l1 :]\ p(s) \geq p(s') \wedge \\ \forall s \in res\ \forall s' \in list2[p\_l2 :]\ p(s) \geq p(s') \wedge \\ \forall i \in \mathbb{N}\ \forall j \in \mathbb{N}((i < j) \wedge (i < |res|) \wedge (j < |res|) \implies p(res[i]) \geq p(res[j]))\end{gathered}$$

L'invariante è vero all'inizio: l'invariante sopra riportato è vero a inizio ciclo in quanto, essendo res vuoto all'inizio, il primo quantificatore universale dei primi due congiunti è trivialmente vero, non avendo scenari s da confrontare. Per quanto riguarda il terzo congiunto, anch'esso è vero in quanto non esistono indici i e j naturali che siano minori di 0 (ovvero la lunghezza di res)

Se l'invariante è vero prima di eseguire il corpo ciclo, allora lo è anche dopo l'esecuzione del corpo del ciclo: prendiamo una generica iterazione del ciclo for. Ipotizziamo che, prima di eseguire il corpo l'invariante sia vero,

dunque che:

$$\forall s \in res\ \forall s' \in list1[p\_l1 :]\ p(s) \geq p(s') \wedge$$
$$\forall s \in res\ \forall s' \in list2[p\_l2 :]\ p(s) \geq p(s') \wedge$$
$$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}((i < j) \wedge (i < |res|) \wedge (j < |res|) \implies p(res[i]) \geq p(res[j]))$$

Dopo l'esecuzione del corpo, abbiamo generato res' e dimostriamo che l'invariante continua a valere anche per esso.

Partiamo osservando che res' consiste in res in cui, al fondo, viene aggiunto lo scenario di probabilità massima tra $list1[p\_l1]$ e $list2[p\_l2]$ (dovuto al costrutto if-then-else nel corpo del ciclo), ovvero:

$$res' = res + [x]$$

$$\text{con } x = \begin{cases} list1[p\_l1], & \text{se } p1 \geq p2 \\ list2[p\_l2] & \text{altrimenti} \end{cases}$$

Dimostriamo che il primo congiunto vale per res':

$$\forall s \in res'\ \forall s' \in list1[p\_l1' :]\ p(s) \geq p(s')$$

Essendo res' pari a $res + [x]$, sappiamo che per ogni scenario s in res, per ipotesi di verità a inizio ciclo, continua a valere il congiunto. Bisogna verificare che valga ancora per x. Per fare ciò dobbiamo chiederci cosa succede al nuovo indice p_l1' che otteniamo dopo l'esecuzione del corpo del ciclo. Possono avvenire due casi per p_l1':

- rimanere invariato: dunque $list2[p\_l2]$ è il massimo ed è maggiore o uguale a $list1[p\_l1]$, ed essendo $list1$ ordinata in maniera decrescente di probabilità, x è maggiore o uguale di ogni elemento in $list1[p\_l1' :]$;
- aumenta di uno: dunque $list1[p\_l1]$ è il massimo ed è maggiore o uguale di tutti i suoi successori, essendo $list1$ ordinata in maniera decrescente di probabilità, di conseguenza x è maggiore o uguale di ogni elemento di $list1[p\_l1' :]$.

In maniera analoga si può dimostrare la stessa cosa per il secondo congiunto:

$$\forall s \in res'\ \forall s' \in list2[p\_l2' :]\ p(s) \geq p(s')$$

Dimostriamo ancora che vale il terzo congiunto:

$$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}((i < j) \wedge (i < |res'|) \wedge (j < |res'|) \implies p(res'[i]) \geq p(res'[j]))$$

Sappiamo, per ipotesi di validità dell'invariante a inizio ciclo, che questo enunciato vale per tutti gli elementi di res. Sappiamo inoltre che res' è pari a $res + [x]$. Dobbiamo dimostrare sostanzialmente che per ogni elemento di res, esso è maggiore o uguale, come probabilità, di x, essendo x in fondo alla lista. Per farlo sfruttiamo il primo e il secondo congiunto dell'invariante a inizio ciclo.

Sappiamo infatti che ogni elemento di res, a inizio ciclo, è maggiore o uguale, come probabilità, di ogni scenario contenuto in $list1[p\_l1 :]$ e $list2[p\_l2 :]$. Di conseguenza, scegliendo x pari allo scenario di probabilità massimo tra $list1[p\_l1]$ e $list2[p\_l2]$, sappiamo che x ha probabilità minore o uguale a ogni elemento di res: il terzo congiunto per res' è valido.

Alla fine della funzione res è ordinato: Essendo l'invariante vero all'inizio e continua a mantenersi a ogni esecuzione del corpo del ciclo for, quando eseguiamo per l'ultima volta il ciclo, al termine di esso l'invariante sarà vero e in res avremo tutti gli elementi di $list1$ e di $list2$ ordinati per probabilità decrescente. Dal momento che una volta usciti dal ciclo la lista res non viene più modificata, l'invariante è valido. Possiamo affermare che alla fine della funzione **merge** la lista risultante è ordinata in maniera decrescente di probabilità e contiene tutti gli elementi che erano presenti in $list1$ e in $list2$.

Ora che abbiamo approfondito le tre funzioni di supporto, andiamo a mostrare in maniera approfondita la semantica della funzione **generateNextScenario** e dimostriamone la correttezza. La semantica è la seguente:
"Dato in input uno scenario, sia esso completo o incompleto, viene restituita una lista ordinata di tutti gli scenari consistenti e completi generati a partire da tale scenario. Il concetto di consistenza viene definito dalle funzioni **update** e **my_sum** e l'ordinamento degli scenari è in maniera decrescente secondo la probabilità di ogni singolo scenario".

Dato uno scenario completo, ovvero uno scenario in cui per ogni scenario element viene fissata una forma, canonica oppure alternativa, la probabilità di tale scenario è il prodotto delle probabilità di ogni scenario element che lo compone.

Dimostriamo la semantica di **generateNextScenario** per induzione sulla lunghezza della lista di scenario elements ancora da analizzare:

Caso bits è uno scenario completo: in questo caso, ogni scenario element della lista di $bits$ è stata fissato e:

- pos eccede le dimensioni dello scenario;
- pos non eccede le dimensioni dello scenario.

Il primo caso è quello di base, allora:

$$generateNextScenario(bits, pos) = [bits]$$

In questo caso, essendo una lista di un elemento solo, è trivialmente ordinata.

Il secondo caso è quello in cui applichiamo il passo induttivo:

Caso 1: lo scenario, sebbene sia completo, non supera per lo scenario element in posizione *pos* i test di consistenza: allora lo scenario intero è incoerente. In questo caso restituiamo la lista vuota: tale lista è trivialmente ordinata:

$$generateNextScenario(bits, pos) = []$$

Caso 2: lo scenario supera per lo scenario element in posizione *pos* i test di consistenza: allora si passa alla posizione $pos + 1$. La chiamata ricorsiva di **GenerateNextScenario** genererà o la lista singoletto con lo scenario completo all'interno, trivialmente ordinata, oppure la lista vuota, anch'essa trivialmente ordinata, se un qualche nodo da $pos + 1$ in avanti genera incoerenza.

Caso bits è uno scenario incompleto: in questo caso la lista di *bits* non è del tutto fissata e *pos* non eccede le dimensioni dello scenario.

Caso 1: *pos* si riferisce a uno scenario element la cui forma è già stata fissata. In tal caso allora:

1. se i bit fissati generano inconsistenza allora:

   $$generateNextScenario(bits, pos) = [\,]$$

   in questo caso la lista vuota è trivialmente ordinata ed è trivialmente costituita da scenari consistenti (non avendone);

2. se i bit fissati non generano inconsistenza, allora, essendo la posizione attuale fissata, passiamo alla prossima:

   $$generateNextScenario(bits, pos) = generateNextScenario(bits, pos + 1)$$

   Essendo la chiamata ricorsiva applicata a una lista di scenario element ancora da analizzare più piccola, vale l'ipotesi induttiva: la chiamata ricorsiva su $(pos + 1)$ genererà una lista ordinata di scenari consistenti.

Caso 2: *pos* si riferisce a uno scenario element che va fissato. In tal caso, si prende lo ScenarioNode in posizione *pos* e si generano due scenari incompleti s_1 ed s_2, uno che corrisponde a *bits* ma in cui fissiamo il nodo in *pos* a 0 e l'altro che corrisponde a *bits* ma in cui fissiamo il nodo in *pos* a 1. A questo punto vengono compiuti una serie di controlli su entrambi gli scenari incompleti presi in maniera indipendente:

1. se i controlli falliscono su s_i allora tale scenario incompleto è inconsistente e la lista degli scenari completi generabili da esso è la lista vuota:

   $$generateNextScenario(s_i, pos) = [\,]$$

in questo caso la lista vuota è trivialmente ordinata ed è anche composta da scenari consistenti, in quanto non è possibile generare scenari consistenti partendo da s_i;

2. se i controlli vanno a buon fine su s_i, allora si compie la chiamata ricorsiva:

$$generateNextScenario(s_i, pos) = generateNextScenario(s_i, pos + 1)$$

per ipotesi induttiva, essendo che la lista di scenario elements che dobbiamo analizzare è diminuita in lunghezza, la chiamata ricorsiva su $(pos + 1)$ genererà una lista ordinata di scenari consistenti.

Alla fine dei controlli, sia per s_1 che per s_2 otteniamo le rispettive liste di scenari completi ordinati e consistenti. Ricordiamo che s_1 ed s_2 sono rispettivamente due varianti dello scenario incompleto di partenza *bits* in cui andiamo a fissare un valore per lo scenario element in posizione *pos*.

Essendo che, per definizione, in *bits* uno scenario element può essere fissato solo a 0 o a 1, gli scenari completi di *bits* saranno tutti quegli scenari completi ottenuti fissando 0 in *pos* più tutti quegli scenari completi ottenuti fissando 1 in *pos*. In sostanza, per calcolare gli scenari completi di *bits*, calcoliamo gli scenari completi di s_1 ed s_2. Sappiamo che gli scenari completi di queste due varianti sono già liste ordinate e consistenti, di conseguenza ci basta fare un merge delle due liste: se utilizziamo la funzione **merge** otterremo per *bits* nuovamente una lista ordinata, composta dagli scenari consistenti ottenuti da s_1 ed s_2.

$$generateNextScenario(bits, pos) =$$
$$merge(generateNextScenario(s_1, pos + 1), generateNextScenario(s_2, pos + 1))$$

Anche in questo ultimo caso otteniamo, per definizione dei valori possibili degli scenario elements e per ipotesi induttiva, che tutti gli scenari completi e consistenti generabili da *bits* sono l'unione degli scenari completi generabili da s_1 ed s_2. Tramite la funzione **merge** siamo sicuri che la lista di scenari completi e consistenti generabili da *bits* è anche ordinata.

4.5 CandidateChooser

L'ultimo modulo che presentiamo è quello dedicato alla politica di priorità dei candidati. Come detto in precedenza, dato un goal possiamo andare a determinare svariati candidati per la sua risoluzione. Una volta aver appurato che non ci sono candidati semplici oppure che tutti i candidati semplici risultano essere in conflitto con l'ontologia, dobbiamo passare ad analizzare i candidati complessi. La domanda che ci poniamo è: quale candidato ci conviene analizzare per primo? Il seguente modulo definisce un ordine tra questi candidati. La classe astratta di riferimento è *CandidateChooser* e il modulo definito si chiama **ComplexCandidateChooser**. All'interno di quest'ultimo sono definite due importanti funzioni:

- data una serie di candidati complessi, va a compiere una operazione di "semplificazione". Infatti, dato un candidato complesso della seguente forma:

  $$CC = \{C_1, C_2, ..., C_n\}$$

  Si va a determinare una forma semplificata CC_S, con la seguente caratteristica: non esiste una coppia (C_A, C_B) di concetti all'interno di CC_S tale per cui C_A sia antenato di C_B. Questa operazione ci permette di andare ad ottenere dei candidati con meno concetti, quindi potenzialmente degli scenari più piccoli da trattare, andando però a sacrificare delle informazioni tipiche degli antenati. Infatti, supponiamo che C_A sia antenato di C_B e che C_A abbia un assioma di tipicalità che ci permette di catturare meglio delle proprietà tipiche del concetto fusione, queste informazioni vengono perse all'interno della semplificazione del candidato. In sostanza, un candidato semplificato può essere visto come un candidato che fornisce un concetto "abbozzato". In questa semplificazione può avverarsi uno dei seguenti scenari:

  1. il candidato semplificato risulta essere un candidato semplice. In tal caso va tenuto da parte e successivamente testato prima di passare a quelli complessi;
  2. il candidato semplificato risulta essere un candidato complesso. In tal caso va aggiunto ai candidati complessi pre-esistenti.

- dati i candidati complessi, siano essi pre-esistenti che dovuti al processo di semplificazione, viene definita una funzione che ordina tali candidati, così da poter eleggere quello che reputiamo il candidato più promettente per l'analisi.

La seconda funzionalità è il cuore del modulo e può essere implementata in vari modi. Nel nostro caso è stato deciso di adottare un approccio semplice per rendere l'idea di come tale funzionalità può essere calata nel contesto del sistema GOCCIOLA nel suo complesso. In particolare, il modulo da noi definito prende come informazioni discriminatorie ai fini dell'ordinamento gli assiomi di tipicalità. Dato un candidato complesso CG, sia esso semplificato che pre-esistente, a esso viene associata una coppia (x, y) con la seguente semantica:

- x rappresenta quanti concetti formano tale candidato complesso;
- y rappresenta quanti assiomi di tipicalità sono coinvolti.

Dati due candidati complessi CG_1 e CG_2, si calcolano le rispettive coppie (x_1, y_1) e (x_2, y_2) e si confrontano tali coppie come segue:

1. si confronta prima la componente x: viene data priorità al candidato composto da meno concetti, ovvero più semplice in fatto di concetti diversi coinvolti;

2. si confronta la componente y in caso di parità di componente x: viene data priorità al candidato che coinvolge più assiomi di tipicalità, questo per poter esprimere in maniera più dettagliata le caratteristiche tipiche del potenziale concetto fusione.

Per comprendere meglio come il modulo da noi definito funziona, presentiamo un esempio. Supponiamo di avere tre candidati per risolvere il nostro goal:

$$\{Orso, Toro, Topo\}, \{Cane, Squalo\}, \{Gatto, Aquila\}$$

Supponiamo inoltre che il primo candidato coinvolga sei proprietà tipiche, il secondo ne coinvolga tre e la terza ne coinvolga quattro.

Il nostro modulo fa una primo ordinamento basandosi sulla cardinalità di ogni candidato, generando il seguente ordinamento:

$$\{Cane, Squalo\}, \{Gatto, Aquila\}, \{Orso, Toro, Topo\}$$

Infine, a parità di cardinalità del candidato si ordinano i candidati preferendo quei candidati che coinvolgono più proprietà tipiche. Il risultato è il seguente:

$$\{Gatto, Aquila\}, \{Cane, Squalo\}, \{Orso, Toro, Topo\}$$

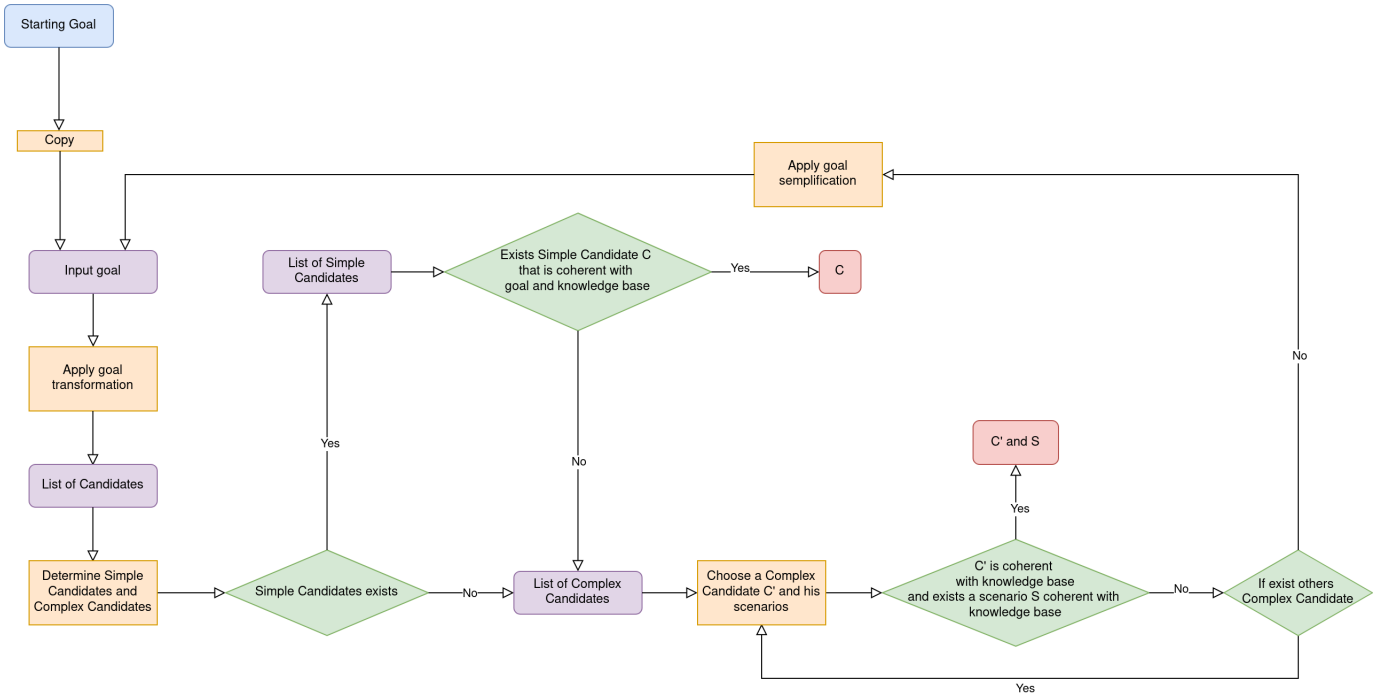
Concludiamo osservando che questo modulo può essere oggetto di varie aggiunte. Per esempio, si potrebbe dotare questo modulo di un tipo di conoscenza statistica oppure basata su risorse semantiche per compiere l'ordinamento. Potremmo anche dotare questo modulo di un meccanismo di "memoria interna" in cui memorizzare perchè un certo candidato complesso C ha fallito così da andare ad aggiornare l'ordinamento dei candidati complessi e svalutare quei candidati che in un qualche modo sono simili C.

5 Analisi e conclusioni

In questo capitolo presentiamo il sistema da noi sviluppato nel suo complesso e analizziamo alcune delle sue caratteristiche. Ricordiamo che il nostro sistema nasce partendo dal tool GOCCIOLA. Tra le caratteristiche principali del nostro sistema abbiamo un'architettura astratta di base per il nuovo sistema e di istanziare tale struttura per mezzo di moduli software. Come questi moduli vengono coordinati tra loro è definito nel modulo EdificaMain, un modulo il cui compito è definire il nuovo flusso di esecuzione del sistema.

5.1 La architettura di EDIFICA

Nel primo capitolo abbiamo presentato GOCCIOLA rappresentando il suo flusso di esecuzione, ora riportiamo di seguito il flusso di esecuzione di EDIFICA.



Nella immagine qui sopra viene riportato il flusso di esecuzione del nuovo sistema in cui:

- i rettangoli gialli rappresentano operazioni su dati in input che possono produrre dati in output;
- i rettangoli viola rappresentano dei dati;

- i rombi verdi rappresentano punti di scelta: a seconda della condizione che si verifica si percorrerà un arco piuttosto che un altro;
- il rettangolo blu rappresenta l'input iniziale al sistema: il goal da risolvere;
- i rettangoli rossi rappresentano l'output del sistema: un concetto C se è stato trovato un concetto semplice che risolve il goal, oppure una coppia (C', S) che rappresentano rispettivamente il goal fusione e le sue proprietà tipiche.

Come si può osservare, tale ciclo è stato arricchito rispetto al ciclo di GOCCIOLA includendo elementi che non erano presenti nel sistema di partenza. Andiamo a presentare tali nuovi elementi che, assieme agli elementi pre-esistenti di GOCCIOLA, formano la architettura di EDIFICA.

Trasformazinoe del goal In GOCCIOLA non era specificato come prendere i concetti per soddisfare il goal. Nel nostro sistema è stata definita una funzione di trasformazione che, dato un goal G va a determinare i possibili candidati per risolvere tale goal.

Candidati Come specificato nel paragrafo precedente, non vi era una nozione di candidato in GOCCIOLA, mentre in questa è stata introdotta suddividendola inoltre in due categorie: candidati semplici, ovvero insiemi composti da un concetto solo, e candidati complessi, ovvero insiemi composti da almeno due concetti.

Precedenza ai concetti semplici All'interno del flusso di esecuzione del nostro sistema si può vedere che, una volta aver definito una serie di candidati, si inizia esplorando i concetti semplici e nel caso non ce ne fossero, oppure fallissero tutti nel soddisfare il goal, si passa a considerare concetti complessi.

Candidati complessi con più di 2 concetti Una ulteriore estensione rispetto al sistema di partenza sta in una maggiore efficienza nel trattare gli scenari per produrre le proprietà tipiche dei concetti fusione, ma anche nella possibilità di trattare non più solo due concetti per volta, ma la possibilità di trattarne un numero arbitrario.

Indebolimento del goal Infine, è stato aggiunto un meccanismo di indebolimento del goal qualora fosse necessario. Supponiamo che nel tentativo di risolvere il goal G si scopre che non si è in grado di fondere dei concetti tra loro senza generare inconsistenze o, a monte, il goal G sia inconsistente. In tal caso il nuovo sistema prende il goal G e va a compiere una operazione di indebolimento, che possiamo sostanzialmente visualizzare come una semplificazione del goal, andando di fatto a rilassare dei vincoli. Dal goal G possiamo ottenere diversi

goal indeboliti G' e il sistema, nel caso peggiore in cui nessun indebolimento sia possibile, li andrà ad analizzare tutti e tenterà di risolverli uno per uno.

Dato questo quadro generale del nuovo sistema, possiamo osservare che di GOCCIOLA è stato mantenuto l'approccio di organizzare la conoscenza tramite una ontologia formale. Inoltre, è stato mantenuto l'approccio al problema: generare nuova conoscenza tramite combinazione di concetti per mezzo della creazione di scenari. Questo approccio è stato però oggetto di modifica che si è aggiunta alla dotazione di un'architettura al nostro nuovo sistema. Le caratteristiche principali di questa architettura sono:

- flessibilità: la architettura definisce una struttura in cui ogni elemento della struttura può essere personalizzato;
- estendibilità: la stessa struttura definita dalla architettura può essere espansa aggiungendo nuovi elementi.

Date queste premesse comprendiamo meglio quale è il significato dell'acronimo EDIFICA, ovvero "ExtenDIble & FlexIble Combination Architecture".

Con EDIFICA ci riferiremo al sistema da noi creato, con le scelte implementative presentate nei capitoli precedenti. Nonostante ciò, faremo spesso riferimento alla sua architettura e di come questa possa permettere al nostro EDIFICA di mutare il suo comportamento a fronte di modifiche di moduli e di aggiunta di altre funzionalità.

5.2 Analisi tramite i livelli di Marr

Come spiegato nel primo capitolo, vi sono diversi modelli che possono essere utilizzati per definire cosa intendiamo per intelligenza artificiale e come andiamo a creare un sistema di intelligenza artificiale. In questo capitolo andiamo ad analizzare GOCCIOLA ed EDIFICA tramite un modello proposto da David Marr nel 1977: la gerarchia di Marr [10]. Questa gerarchia è modello per poter analizzare sistemi computazionali. La prima caratteristiche fondamentale di questo modello è che riguarda i sistemi computazionali in generale, ovvero non solo i sistemi computazionali che nascono per replicare l'intelligenza umana, ma un qualsiasi sistema in grado di compiere calcoli, come una calcolatrice.

Questo modello propone l'idea di analizzare un sistema computazionale secondo tre livelli:

- Livello di teoria computazionale: è il livello più astratto e consiste nel dare una specifica del task da svolgere. Nel caso dei sistemi di intelligenza artificiale questo task è associato un certo fenomeno cognitivo. In questo livello non si specificano i processi e i meccanismi cognitivi coinvolti;
- Livello di algoritmi e rappresentazioni: è il livello intermedio in cui si specifica come un certo task è svolto. In questo livello si specificano le procedure e le strutture di rappresentazione coinvolte in tale task;

- Livello di implementazione: è il livello più concreto e riguarda il come fisicamente vengono implementate le strutture di rappresentazione e gli algoritmi.

Marr stesso ci fornisce un esempio per comprendere come analizzare un sistema computazionale attraverso questi tre livelli: analisi di un registratore di cassa.

A livello di teoria computazionale siamo nel campo dell'aritmetica, più precisamente nella teoria delle addizioni. A questo livello è importante definire la funzione che sarà implementata, ovvero la addizione, e le sue proprietà, come la commutatività e la associatività.

A livello di algoritmi e rappresentazioni, ci si concentra sulla forma della rappresentazione dei dati e il processo di elaborazione dei dati. Per quanto riguarda la rappresentazione possiamo usare i numerali arabi, mentre per il processo di elaborazione per la addizione possiamo utilizzare lo schema della somma in colonna con riporto.

A livello di implementazione, ci si concentra su come la rappresentazione e il processo di elaborazione sono fisicamente realizzati. Se si implementa su un moderno circuito elettronico possiamo implementare la rappresentazione di un numero tramite una codifica binaria in complemento a 2. Per quanto riguarda il processo di elaborazione si può definire il circuito che svolge la sommatoria bit a bit tra due numeri binari.

Presentiamo ora GOCCIOLA ed EDIFICA alla luce di questi tre livelli.

5.2.1 Analisi di GOCCIOLA

Livello di teoria computazionale il sistema può essere calato nel contesto della logica, in particolare della logica descrittiva $\mathcal{ALC}$ arricchita con assiomi di tipicalità. Possiamo definire il task svolto dal sistema come un task di deduzione, ovvero come quel task che consiste nel partire da un'ontologia con delle informazioni già presenti ed estrarre da essa nuova conoscenza. Tale nuova conoscenza viene richiesta da un utente.

Livello di rappresentazione e di algoritmi il sistema rappresenta le informazioni come classi, individui e proprietà per mezzo di rappresentazioni ontologiche espresse in OWL. Questa rappresentazione fornisce di default delle meta-relazioni. Inoltre, i goal sono rappresentate come congiunzioni logiche di concetti. Per quanto riguarda l'algoritmo, questo consiste nel prendere casualmente due concetti e di tentare di compiere una fusione semantica. Tale fusione consiste nel prendere le informazioni di carattere tipico di ciascun concetto coinvolto nella fusione e di generare quelli che nel capitolo precedente abbiamo definito scenari.

Livello di implementazione il sistema rappresenta ogni informazione di base, ovvero classi, individui, proprietà, meta-relazioni, come delle entità software complesse, ognuna con le proprie caratteristiche. Ogni entità software viene poi codificata in linguaggio binario e in modo tale da poter essere memorizzata e gestita su un moderno computer. Per quanto riguarda l'algoritmo, quest'ultimo vine implementato come programma che viene tradotto anch'esso in una serie di bit che indicano operazioni che la CPU deve svolgere.

5.2.2 Analisi di EDIFICA

Passiamo ora ad analizzare tramite i livelli di Marr il sistema nato dalla seguente tesi di laurea: EDIFICA.

Livello di teoria computazionale il sistema EDIFICA risulta essere equivalente a GOCCIOLA dal punto di vista della teoria computazionale utilizzata: utilizziamo sempre la logica $\mathcal{ALC}$ arricchita con assiomi di tipicalità e il task è sempre quello di compiere deduzione dato un obiettivo stabilito dall'utente.

Livello di rappresentazione e di algoritmi il sistema EDIFICA rappresenta le informazioni come classi, individui e proprietà per mezzo di rappresentazioni ontologiche espresse in OWL. Un aspetto interessante riguarda la rappresentazione dei goal: non sono più espressi solo ed esclusivamente come congiunzioni, ma possono includere negazioni, disgiunzioni e class restriction. Inoltre, vi è un altro elemento importante nella rappresentazione: il concetto di candidato. Per quanto riguarda l'algoritmo, questo consiste in tre macro-passaggi:

- un passaggio di ricerca basato sulla logica $\mathcal{ALC}$ per vagliare i candidati alla risoluzione del goal;
- un passaggio di ordinamento che permetta di esprimere quale candidato è preferibile esplorare per primo;
- un passaggio di fusione semantica di due o più concetti. Quest'ultimo passaggio è semanticamente equivalente al passaggio di fusione di concetti presenti in GOCCIOLA.

Livello di implementazione il sistema EDIFICA risulta essere una estensione di GOCCIOLA dal punto del livello di implementazione: oltre a codificare in binario le rappresentazioni già presenti in GOCCIOLA, codifichiamo anche il concetto di candidato come entità software che vengono a loro volta tradotte in codice binario. Per quanto riguarda l'algoritmo, quest'ultimo vine implementato come programma che viene tradotto anch'esso in una serie di bit che indicano operazioni che la CPU deve svolgere.

A seguito della nostra analisi per mezzo del modello proposto da Marr, possiamo affermare che il sistema EDIFICA a livello di rappresentazione e di algoritmi presenta una definizione più dettagliata e articolata in più passaggi. L'elemento

interessante rimane il fatto che EDIFICA permette di specificare le funzioni di ricerca dei candidati e di ordinamento dei candidati in maniera libera, senza andare a porre vincoli forti sul come queste debbano essere implementate

5.3 EDIFICA e il modello di Russell&Norvig

Nel primo capitolo abbiamo presentato le caratteristiche di GOCCIOLA secondo il modello di Russell&Norvig. In tale capitolo abbiamo osservato in particolare il fatto che GOCCIOLA sia un tool che si va a collocare nella famiglia dei software che hanno come argomento il pensare razionalmente. Avevamo anche fatto notare il fatto che questa categorizzazione non è mutuamente esclusiva, facendo notare che in fondo GOCCIOLA può avere elementi che appartengono alla famiglia del pensare umanamente, come l'idea di scartare determinati scenari perchè, dal punto di vista umano, considerabili banali.

GOCCIOLA secondo Russell & Norvig	
Studio del pensare umanamente	Studio del pensare razionalmente
Studio dell'agire umanamente	Studio dell'agire razionalmente

Qui sopra viene riportata la tabella con le quattro categorie definite dal modello di Russell&Norvig con una colorazione di tipo heat-map (mappa di calore). Più il colore della cella è tendente a un colore caldo, più sono gli elementi che il sistema ha in comune con tale cella. Più il colore della cella tende a un colore freddo, meno saranno gli elementi che il sistema ha in comune con tale cella.

Osserviamo che GOCCIOLA risulta essere particolarmente legata alla categoria dello **Studio del pensare razionalmente**, un po' meno con lo **Studio del pensare umanamente** e ha pochi legami con lo **Studio dell'agire umanamente** e lo **Studio dell'agire razionalmente**.

Possiamo affermare che EDIFICA, come il suo predecessore, ha sempre elementi di ambo le categorie:

- pensare razionalmente: si fa ampio uso del ragionamento ontologico all'interno della funzione di ricerca dei candidati, così come si utilizzano aspetti del linguaggio logico utilizzato per ottimizzare la creazione degli scenari;
- pensare umanamente: si fa uso di aspetti che riguardano la cognizione per quanto può riguardare la funzione di ordinamento dei candidati, così come sul come strutturare la ricerca dei candidati. In questo ultimo caso, infatti, si è fatto uso del sub-goaling: dato un goal di cui cerchiamo i candidati, non solo utilizziamo il ragionamento ontologico per trovarne i

candidati, ma compiamo una frammentazione del goal in sotto-goal così da determinare nuovi candidati fondendo tra loro i candidati dei diversi sotto-goals generati.

EDIFICA secondo Russell & Norvig	
Studio del pensare umanamente	Studio del pensare razionalmente
Studio dell'agire umanamente	Studio dell'agire razionalmente

Come si può osservare, i legami di EDIFICA sono più forti per quanto riguardano lo **Studio del pensare umanamente** e lo **Studio del pensare razionalmente**. Per quanto riguarda le rimanenti categorie, possiamo affermare che EDIFICA ha gli stessi legami che ha GOCCIOLA con tali categorie.

Ricordiamo che la proprietà interessante di EDIFICA è l'architettura flessibile: un utente può definire altre funzioni di ricerca e altre funzioni di ordinamento potendo includere all'interno di ognuna diversi meccanismi basati su logica oppure basati sulla cognizione oppure anche misti, includendo aspetti di una parte e dell'altra.

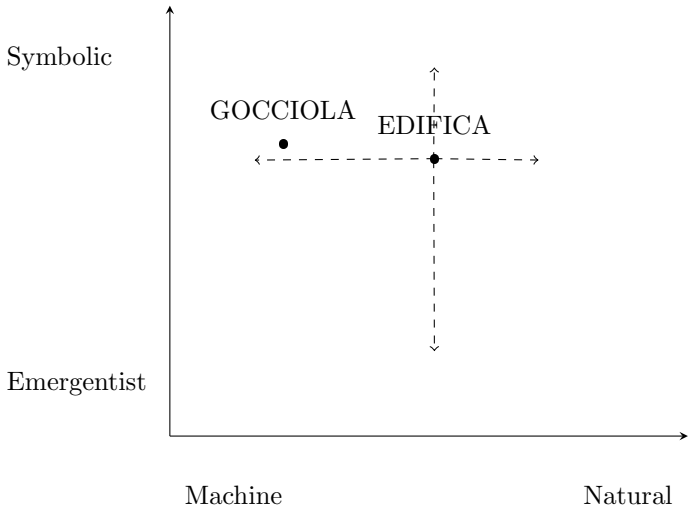
5.4 EDIFICA e il modello di Vernon

Presentiamo ora come EDIFICA si va a collocare all'interno del modello di Vernon. Ricordiamo che GOCCIOLA secondo tale modello è un sistema che si basa su una implementazione simbolica e che tendenzialmente è un sistema che si colloca nel continuum tra machine-oriented e natural-oriented, tendendo al machine-oriented.

EDIFICA, dal canto suo, è sempre un modello simbolico, perchè basato sulla definizione formale di concetti secondo un linguaggio prestabilito e sulla manipolazione di tali concetti per mezzo di relazioni e meta-relazioni. L'elemento interessante di EDIFICA sta sempre nella flessibilità della sua architettura, che permette di andare a includere all'interno del sistema le ipotesi di carattere strutturale che più preferiamo. Nel caso della nostra tesi, abbiamo dotato il sistema EDIFICA di un approccio a sub-goaling per la costruzione dei candidati, così come abbiamo dotato il sistema di un approccio che possiamo ritenere ragionevole per quanto riguarda l'ordinamento di tali candidati. Non è esclusa la possibilità anche di poter aggiungere euristiche per andare ad escludere direttamente determinati scenari nella loro costruzione in base a quelle che possiamo considerare le euristiche utilizzate dall'essere umano.

In sostanza, il sistema EDIFICA può essere visto un po' come un punto "mobile" all'interno degli assi del modello di Vernon, che può andare a tende-

re verso il funzionalismo così come allo strutturalismo. Tramite la flessibilità della sua architettura, EDIFICA può "muoversi" orizzontalmente all'interno di tale piano cartesiano. Inoltre, in virtù della sua architettura sostanzialmente aperta e modulare, si possono includere anche nuovi moduli che possono anche prevedere l'interazione di diversi altri sotto-sistemi e che possono dunque introdurre una componente emergentista. Tale punto può anche muoversi sull'asse verticale del modello di Vernon. Su questo ultimo aspetto poniamo però luce su una criticità: per come è stata definita la sua architettura, EDIFICA tende ad avere un approccio simbolico e strutturato e quindi introdurre elementi di natura emergentista o comunque non basati sulla logica può essere più complesso, rispetto all'introdurre versioni dei moduli pre-esistenti che includono ipotesi più o meno strutturaliste.



Rappresentazione di GOCCIOLA e di EDIFICA nello spazio 2D di Vernon.

5.5 Conclusioni

Alla luce di ciò che è stato presentato in questa tesi possiamo affermare che l'elemento principale che è stato introdotto nel sistema da noi creato sta nella sua architettura. Inoltre, il sistema da noi sviluppato combina la architettura da noi definita con un approccio logico e simbolico al task di risoluzione di un goal espresso dall'utente.

5.5.1 Pro e Contro

Tra gli elementi interessanti che possiamo trovare in EDIFICA abbiamo il fatto che è ha un'architettura che si basa sull'utilizzo di sistemi simbolici, legati quindi a un approccio di tipo "Classical AI". Inoltre, l'elemento interessante è come

utilizzare una logica non classica per poter trattare aspetti non rigidi e ferrei della realtà e come sfruttare questa logica per poter compiere ragionamenti per selezionare i candidati alla risoluzione di un goal e per eseguire una fusione semantica tra concetti.

Un altro elemento interessante di EDIFICA è la sua natura modulare, dunque flessibile ed estendibile, che permetta la possibilità di inserire altre tipologie di vincoli a piacere e di poter anche inserire moduli che possano avere al proprio interno un approccio di tipo emergentista.

Tra i limiti del sistema riportiamo il fatto che la manipolazione di simboli è tipicamente costoso computazionalmente, infatti nella produzione degli scenari sebbene abbiamo compiuto delle ottimizzazioni, nel caso peggiore rimaniamo appartenenti nella famiglia degli algoritmi EXP.

Un altro limite è costituito dal fatto che questo sistema tratta come conoscenza solo ed esclusivamente conoscenza esprimibile come classi, di fatto non permettendo di gestire anche altre tipologie di conoscenza. Il limite principale risiede nel fatto che questa forma di conoscenza è strutturata, ben definita a priori e nota. Il sistema EDIFICA, in sostanza, dipende molto dalla base di conoscenza e, in assenza di questa, non è banale definire tramite procedure automatiche una base di conoscenza a partire dai soli dati osservati nella realtà. Inoltre, vi è sempre il rischio che le logiche descrittive, su cui si basa la rappresentazione della conoscenza, non siano sufficientemente espressive per la realtà che vogliamo rappresentare.

5.5.2 Estensioni di EDIFICA

Presentiamo in questa ultima sezione alcuni spunti per possibili estensioni di EDIFICA.

Introdurre ulteriori vincoli strutturali: una possibile estensione del sistema EDIFICA consiste nell'andare a fornire dei vincoli strutturati più forti per i singoli moduli del sistema. Per esempio, il sistema attuale compie ragionamento automatico unito a una tecnica di sub-goaling per andare a determinare una serie di candidati, che possono variare da alcuni fino a migliaia. Un possibile vincolo potrebbe essere quello di definire un tetto massimo di candidati generabili. In effetti, l'essere umano in genere non va a determinare a priori ogni possibilità, solitamente genera alcune opzioni e se nessuna di queste funziona passa a generarne di altre. Inoltre, non è detto che l'essere umano compia in dei casi dei ragionamenti espliciti. Per esempio, potremmo istintivamente pensare a un determinato concetto C per un goal G per poi accorgerci che C sarebbe inutile ai fini della risoluzione dei goal: istintivamente abbiamo proposto il concetto C, senza per forza compiere un ragionamento logico esplicito. Nel caso di EDIFICA, non esiste questa "deduzione intuitiva", ma è una deduzione formale tramite ragionamento automatico. Si potrebbe pensare allora di includere una parte nuova al modulo di trasformazione del goal che permetta al sistema di

fare anche delle scelte un po' bizzarre, che non sono deducibili con il ragionamento automatico, ma che vadano ad imitare quelle scelte a volte bizzarre che noi umani compiamo.

Considerare anche individui e proprietà nel ragionamento: un'altra possibile estensione di EDIFICA potrebbe essere quella di estendere l'analisi dei candidati, il loro ordinamento e la generazione degli scenari anche a proprietà ed individui. Infatti, tutto ciò che è stato presentato di EDIFICA riguarda le classi e come possiamo andare a organizzarle, ma possiamo anche compiere ragionamenti più raffinati tramite proprietà e individui. Per esempio, in OWL è possibile anche definire una classe C come una enumerazione di individui. La classe $PuntiCardinali$ può essere definita come la classe composta dai seguenti individui: $Nord$, Sud, $Ovest$ ed Est. In questo modo, potremmo avere per una classe sia una definizione tramite class expression, ma anche una definizione tramite enumerazione di individui. Questa nozione di individui legati a una classe potrebbe essere utilizzata per permettere non solo di compiere ragionamento prototipico, basato sulle proprietà tipiche di una classe, ma anche ragionamento esemplare, ovvero basato su un'analisi degli esemplari che compongono tale classe. Si potrebbe pensare di poter attribuire anche un peso a ciascun esemplare che compone la classe, per indicare quanto sia comune osservare tale individuo nella realtà.

Considerare semantiche alternative per gli assiomi di tipicalità: una possibile estensione che può essere compiuta, infine, potrebbe riguardare gli assiomi di tipicalità. Un primo spunto interessante potrebbe essere quello di definire metodi empirici per stimare la probabilità dell'assioma di tipicalità e di far variare questo metodo di stima in base al contesto in cui andiamo a calare il sistema. Per esempio, nella savana la probabilità di trovare, tra gli uccelli, un pinguino sarebbe zero, mentre al circolo polare artico tale probabilità sarebbe quasi del 100%. Potrebbero essere interessanti da studiare dei meccanismi con cui il sistema può avere più insiemi di assiomi di tipicalità che va a intercambiare in base al contesto in cui è calato magari deducendo, dalle asserzioni proposte dall'utente, quale sia il contesto in questione. Un secondo spunto interessante potrebbe essere quello di andare a studiare più approfonditamente la semantica dell'assioma di tipicalità. Come detto, in questa tesi abbiamo associato a un assioma un numero che consiste nella sua probabilità, ma questo potrebbe non per forza essere un numero o, se lo è, non per forza potrebbe avere il valore di una probabilità, potrebbe essere inteso come indice di gradimento (una sorta di valutazione che noi andiamo a inserire nel sistema), così come potrebbe essere un certainty factor, ovvero un indicatore di quanto siamo certi che un dato evento si manifesti.

Riferimenti bibliografici

[1] Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, and Simone Albani. A distribution semantics for probabilistic ontologies. In Fernando Bobillo, Rommel N. Carvalho, Paulo Cesar G. da Costa, Claudia d'Amato, Nicola Fanizzi, Kathryn B. Laskey, Thomas Lukasiewicz, Trevor Martin, and Matthias Nickles, editors, *Proceedings of the 7th International Workshop on Uncertainty Reasoning for the Semantic Web (URSW 2011), Bonn, Germany, October 23, 2011*, volume 778 of *CEUR Workshop Proceedings*, pages 75–86. CEUR-WS.org, 2011.

[2] Eleonora Chiodino, Antonio Lieto, Federico Perrone, and Gian Luca Pozzato. A goal-oriented framework for knowledge invention and creative problem solving in cognitive architectures. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2893–2894. IOS Press, 2020.

[3] Laura Giordano, Valentina Gliozzi, Gian Luca Pozzato, and Riccardo Renzulli. An efficient reasoner for description logics of typicality and rational closure. In Alessandro Artale, Birte Glimm, and Roman Kontchakov, editors, *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017*, volume 1879 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[4] A. Lieto. *Cognitive Design for Artificial Minds*. Routledge, first edition, 2021.

[5] Antonio Lieto, Federico Perrone, and Gian Luca Pozzato. GOCCIOLA: generating new knowledge by combining concepts in description logics of typicality (short paper). In Alberto Casagrande and Eugenio G. Omodeo, editors, *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019*, volume 2396 of *CEUR Workshop Proceedings*, pages 157–166. CEUR-WS.org, 2019.

[6] Antonio Lieto, Federico Perrone, Gian Luca Pozzato, and Eleonora Chiodino. Beyond subgoaling: A dynamic knowledge generation framework for creative problem solving in cognitive architectures. *Cogn. Syst. Res.*, 58:305–316, 2019.

[7] Antonio Lieto and Gian Luca Pozzato. A description logic framework for commonsense conceptual combination integrating typicality, probabilities and cognitive heuristics. *J. Exp. Theor. Artif. Intell.*, 32(5):769–804, 2020.

[8] Antonio Lieto, Gian Luca Pozzato, and Federico Perrone. A dynamic knowledge generation system for cognitive agents. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, pages 676–681. IEEE, 2019.

[9] Antonio Lieto, Gian Luca Pozzato, and Alberto Valese. COCOS: a typicality based concept combination system. In Paolo Felli and Marco Montali, editors, *Proceedings of the 33rd Italian Conference on Computational Logic, Bolzano, Italy, September 20-22, 2018*, volume 2214 of *CEUR Workshop Proceedings*, pages 55–59. CEUR-WS.org, 2018.

[10] D. Marr. *Vision*. W. H. Freeman, 1982.

[11] A. Newell and H.A. Simon. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[12] S. J. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice-Hall, second edition, 2002.

[13] D. Vernon. *Artificial Cognitive Systems: A Primer*. MIT Press, 2014.