1 Struttura dell'applicazione

1.1 Centri di tuple come basi di conoscenza.

Questa applicazione è un esempio di come sfruttare i centri di tuple al pari di basi di conoscenza per teorie logiche. Questo può essere fatto attraverso l'introduzione di primitive in Tucson simulate in questo caso attraverso un'opportuna libreria prolog chiamata ACLTLibrary (ACLT = Agent Coordination through Logic Theories).

Le primitive introdotte sono quattro: demo, demoWait, demoLast, demoWaitLast. Queste consentono ad un agente logico prolog di individuare in un centro di tuple TC una tupla di pattern T. Se quest'ultima non porta ad una soluzione il motore prolog esegue backtracking ritornando alla chiamata della primitiva che restituirebbe un'altra tupla T. In pratica il centro di tuple viene visto come un insieme di fatti di una teoria logica.

"demo(T,TC)": Alla prima chiamata esegue una snapshot del centro di tuple TC e poi sulle tuple registrate ne cerca una che unifichi con T. In caso di backtracking viene selezionata un'atra tupla ma la snapshot non viene mai aggiornata con le nuove tuple di TC. Se il centro di tuple alla chiamata è vuoto allora lo è anche la snapshot e quindi la demo fallisce immediatamente.

```
demo(T,TC) :-
        TC ? inp(rd_all(T,L)),
        demoL(T,L).
demoL(H,[H|T]).
demoL(T,[_|L]) :-
        demoL(T,L).
```

"demoWait(T,TC)": Alla chiamata, demoWait controlla che il centro di tuple non sia vuoto, in caso contrario resta in attesa finché non viene inserita almeno una tupla. Dopodiché si comporta esattamente come demo.

```
demoWait(T,TC) :-
       TC ? inp(empty),
       demoWait(T,TC).
demoWait(T,TC) :-
       demo(T,TC).
```

"demoLast(T,TC)": Alla chiamata esegue una snapshot del centro di tuple, se risulta vuoto allora fallisce subito, altrimenti cerca una tupla che unifichi con il pattern T. A differenza di demo in questo caso le tuple vengono costantemente aggiornate anche in caso di backtracking.

```
demoLast(T,TC) :-
       demoLast(T,TC,[]).
demoLast(T,TC,L) :-
       TC ? inp(rd_all(T,L1)),
       L2=[H|Tail],
       diffp(L2,L1,L),
       demoLastL(T,L2,L2,TC).

demoLastL(T,[T|Tail],L,TC).
demoLastL(T,[_|Tail],L,TC) :-
       demoLastL(T,Tail,L,TC).
demoLastL(T,[],L,TC) :-
       demoLast(T,TC,L).

diffp([],[],_).
diffp([H|T2],[H|T],Lb) :-
       not member(H,Lb),
       diffp(T2,T,Lb).
diffp(L,[H|T],Lb) :-
       member(H,Lb),
       diffp(L,T,Lb).
```

"demoWaitLast(T,TC)": unisce le caratteristiche di demoWait e demoLast.

```
demoWaitLast(T,TC) :-
       TC ? inp(empty),
       demoWaitLast(T,TC).
demoWaitLast(T,TC) :-
```

```
        demoLast(T,TC).

diffp([],[],_).
diffp([H|T2],[H|T],Lb) :-
        not member(H,Lb),
        diffp(T2,T,Lb).
diffp(L,[H|T],Lb) :-
        member(H,Lb),
        diffp(L,T,Lb).
```

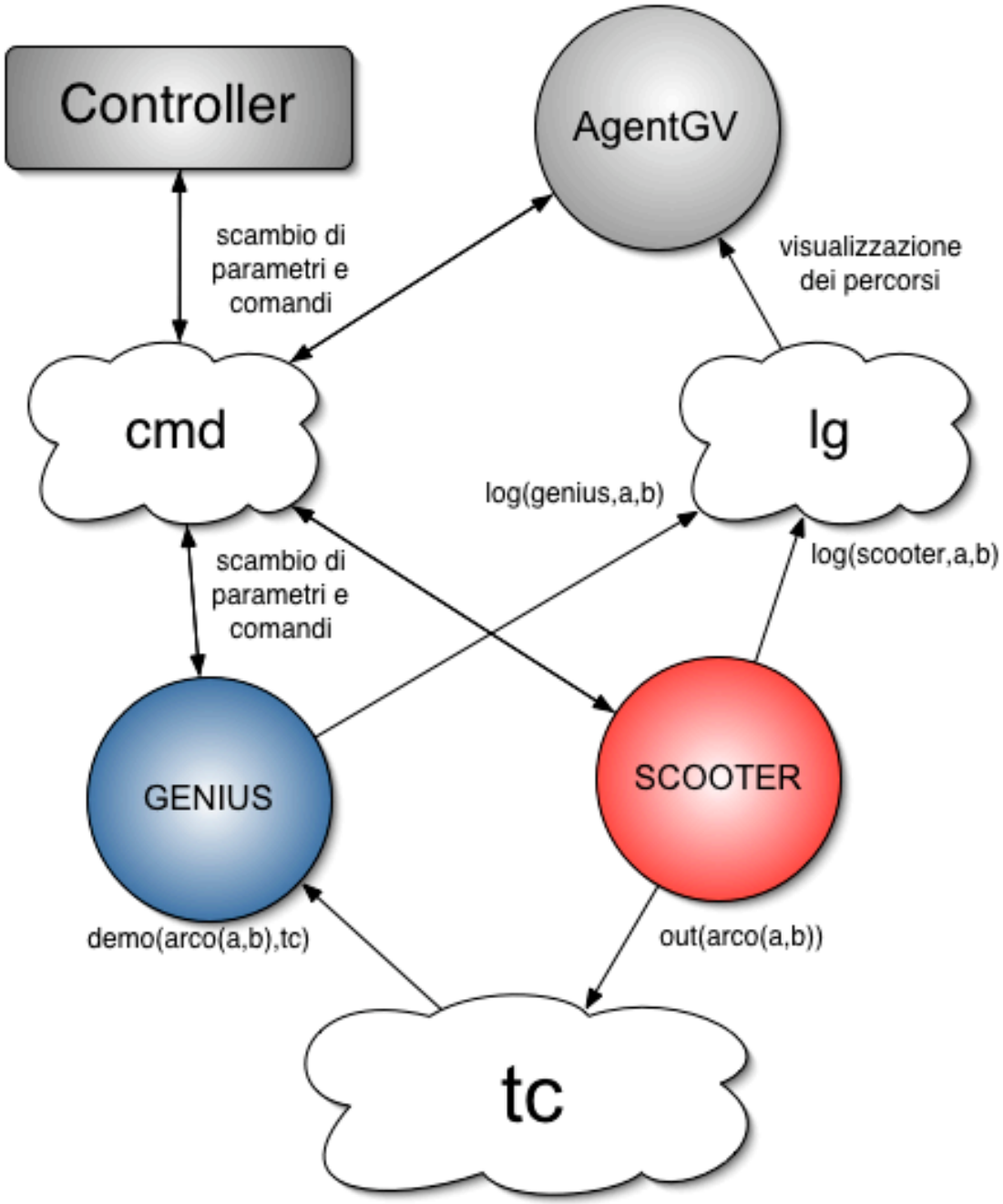
La simulazione di tali primitive è stata realizzata attraverso una libreria per tuProlog chiamata ACLTLibrary. Per funzionare correttamente però è necessario caricare sul centro di tuple delle specifiche in ReSpecT: una simula la primitiva rd_all e l'altra, empty, informa se il centro di tuple è vuoto. “rd_all” è necessaria per realizzare la snapshot al centro di tuple.

Tali primitive consentono di utilizzare agenti logici in grado di valutare la conoscenza di un centro di tuple nel loro complesso, infatti in questo modo si può sfruttare il meccanismo di backtracking di prolog che sarebbe inutilizzabile sruttando le semplici primitive offerte da Tucson.

1.2 Applicazione di prova

Per valutare l'utilizzo delle primitive si è scelto di realizzare una applicazione che simula un ambiente descritto da un grafo orientato. Su tale grafo circolano due agenti: uno veloce (Scooter) che esplora il grafo, uno lento (Genius) che dalla conoscenza acquisita dall'esplorazione di scooter è in grado di determinare un percorso fra due nodi del grafo. Genius chiaramente utilizza le nuove primitive ACLT che essendo puramente logiche gli permettono di dimostrare il goal, ovvero la ricerca di un percorso all'interno del grafo senza passare più volte lo stesso punto. Senza le primitive ACLT sarebbe stato necessario memorizzare in liste o come fatti tutte le tuple del centro di tuple.

L'applicazione stessa è un sistema ad agenti rappresentato qui di seguito.



Allo start-up Controller lancia il Node che rappresenta il mattone fondamentale dell'infrastruttura Tucson, dopodiché crea una finestra con i controlli per la simulazione, esegue il parsing del file xml in cui è descritto il grafo e istanzia un nuovo agente chiamato agentGV. Quest'ultimo è un

agente prolog, anch'esso di controllo, che si occupa dell'output grafico e della gestione della simulazione. AgentGV lancia infatti il GraphicViewer e le finestre di log di Scooter e Genius, inoltre istanzia due altri agenti logici, Scooter e Genius, oggetto della simulazione.
Controller, AgentGV e i due agenti logici si passano comandi e configurazioni attraverso al centro di tuple chiamato "cmd" (command). Una volta che la simulazione ha avuto inizio Scooter inizia a esplorare il grafo che gli è stato passato attraverso cmd da Controller in maniera caotica perseguendo lo scopo di percorrere un certo numero di passi (questo lo rende molto veloce in quanto deve eseguire poche inferenze). Tutta la conoscenza che acquisisce attraverso l'esplorazione viene inserita in "tc", quest'ultimo rappresenta la base di conoscenza per Genius che non ha visione completa del grafo e che quindi si deve basare sulla conoscenza acquisita da Scooter. Entrambi eseguono un log delle proprie attività in "lg" (log) che rappresenta il registro cronologico di tutti gli spostamenti nel grafo eseguiti dai due agenti. Al termine della simulazione AgentGV utilizza le informazioni contenute in "lg" per visualizzare il percorso dei due agenti. (nota: AgentGV sfruttando le primitive ACLT potrebbe eseguire sintesi approfondite dei dati su "lg" che con le primitive attuali sarebbero difficili da compiere).

Analizziamo più in dettaglio quali sono le informazioni scambiate fra i diversi agenti.

“cmd”: Il tuple centre command ha effettivamente un ruolo di coordinazione fra gli agenti, in quanto esso permette di scambiare messaggi di configurazione, in questi sono contenute diverse informazioni: la struttura del grafo (sotto forma di termine prolog); il nodo di partenza e il nodo di arrivo dell’agente Genius; il numero di passi che devono essere svolti dallo Scooter. Su “cmd” viene gestita anche la fase di avvio e di terminazione della simulazione. Quando l'utente avvia la simulazione “cmd” riceve da

Controller una tupla “controller(start)” la quale grazie ad una reaction del centro di tuple viene trasformata in due tuple: “startSim(scooter)” e “startSim(genius)”. Gli agenti Genius e Scooter al termine dell'elaborazione inviano singolarmente a cmd una tupla del tipo “endSim(ID)” dove ID rappresenta il nome dell'agente. Solamente quando entrambi gli agenti hanno terminato è possibile valutare il risultato della simulazione.

“lg”: Il tuple centre log è un puro centenitore di informazioni e non ha un reale ruolo di coordinazione fra agenti. Le reaction in questo caso sono state utilizzate per trasformare le tuple ricevute, senza informazione cronologica, in tuple con un numero di sequenza che permette di stabilire la sequenza delle operazioni svolte dai due agenti.

Il grafo dell'applicazione è descritto da un file xml il quale viene trasformato da Controller in un termine prolog. Da notare che il termine prolog risultante non rappresenta realmente il grafo ma più che altro una rappresentazione prolog di un documento xml. Questo rende tale strumento molto più flessibile, utile al parsing di sorgenti xml direttamente dentro alla teoria prolog. Ciò è stato realizzato grazie all’introduzione di una libreria per tuProlog chiamata SAXParserLibrary.

Se si vuole realizzare un grafo di diverso tipo è sufficiente scrivere un file xml in cui si descrive la posizione dei nodi e i link fra di essi. Attraverso semplici modifiche sarebbe possibile addirittura utilizzare tale strumento per grafi pesati. Tali modifiche sono di lieve entità proprio grazie al fatto che la quasi totalità del sistema si basa su teorie prolog rapidamente modificabili.

2 Considerazioni

2.1 Basi di conoscenza dinamiche

In questa applicazione si può notare come un agente logico (Genius) possa attingere conoscenza da una base di conoscenza dinamica che per definizione è composta da informazioni variabili nel tempo. Ovviamente questo è in contrasto con il principio di correttezza e completezza che deve avere una teoria logica. In questo caso i fatti sono rappresentati dalle tuple che compongono la base di conoscenza, le quali non sono né corrette e né complete. Il problema della completezza viene quindi trascurato, le diverse primitive introdotte (demo, demoWait, demoLast, demoWaitLast) permettono di avere maggiori strumenti per amministrare meglio il problema della completezza. Queste però non danno alcun contributo al grosso problema della corretezza.

Nell'applicazione descritta la conoscenza evolve, in altre parole si può dire che non è completa ma è corretta. Se il grafo rappresentasse il traffico di una città, allora il peso degli archi varierebbe nel tempo. Questo risulta un grosso problema perchè l'agente logico potrebbe svolgere inferenze su fatti che durante l'elaborazione potrebbero modificarsi (il che equivale a renderli falsi), questo implicherebbe che ogni deduzione logica sarebbe necessariamente falsa. Questi fatti vengono definiti transient (passeggeri).

2.2 Caratterizzazione dei fatti transient

Prima di tutto è necessario capire quando un fatto può essere considerato transient. Ad esempio: le leggi fisiche sono fatti incontestabili, mentre la temperatura dell'ambiente è chiaramente variabile nel tempo

quindi transient. In realtà è necessario considerare il punto di vista di chi esegue inferenze sul fatto. Se il fatto nel corso di una dimostrazione non cambia allora è possibile considerarlo fisso, mentre se cambia allora è transient. Pensare al solo vincolo temporale è però fuorviante, il caso della luminosità di un ambiente è emblematico.

Si supponga di dover realizzare un sistema di controllo di una fotocamera. Il sistema è composto da due agenti logici, uno permette il controllo di un impianto di illuminazione dell'ambiente, l'altro gestisce la fotocamera. Entrambi hanno due goal distinti che richiedono la conoscenza della luminosità dell'ambiente con livelli di tolleranza diversi, in un caso una piccola variazione della luminosità non impone di rivedere l'accensione o lo spegnimento dell'impianto di illuminazione, mentre per l'agente fotografo una piccola variazione implica la variazione di diversi parametri sulla fotocamera.

Tale tolleranza è necessaria a stabilire la correttezza dell'informazione. Infatti ogni strumento di misura possiede una certa tolleranza che indica all'utente di quanto la misura può essere errata, questo è noto a priori e quindi ogni misura è corretta a meno di una certa incertezza trascurabile. Ciò può essere esteso a tutti i fatti applicandogli un concetto di tolleranza. In poche parole un fatto è vero fintantoché rimane pressoché invariato dal punto di vista dell'utilizzatore. Sotto questo aspetto è impensabile considerare la veridicità di un fatto come una peculiarità oggettiva.

Da queste considerazioni si evince che l'agente stesso deve avere gli strumenti per valutare la veridicità di un fatto, in altri termini deve tenere sotto costante controllo tutti i fatti utilizzati nella dimostrazione fino al raggiungimento del goal, se durante questo tempo decadesse uno dei fatti, l'agente sarebbe costretto a cercare un altro fatto nella base di conoscenza che si discosti il meno possibile dal precedente (comunque all'interno della tolleranza assegnata). Se il fatto fosse irrimediabilmente perso allora sarebbe

necessario ripartire dall'ultimo passaggio della dimostrazione che si può ritenere ancora corretto. Ne consegue che la tolleranza è un potente strumento di tuning infatti minor tolleranza implica maggiore correttezza ma anche una risposta in genere meno veloce (perchè la dimostrazione verrebbe spesso interrotta e ricorretta), mentre una tolleranza maggiore implica minor correttezza dei risultati ma anche un risposta più veloce.