Abstract

In questo lavoro di tesi si vuole presentare un nuovo linguaggio che fonda le sue basi su AgentSpeak e tuProlog permettendo quindi di implementare il modello ad agenti, ereditato da AgentSpeak, su ambienti o piattaforme diverse, grazie alla flessibilità di tuProlog. Verrà preso in esame l'utilizzo del linguaggio in combinazione con il meta-simulatore Alchemist e ne verranno descritti i dettagli implementativi.

Indice

Elenco delle figure

Codici sorgenti

Capitolo 1

Introduzione

In questo lavoro di tesi ci si è concentrati sulla realizzazione di un nuovo linguaggio ad agenti che permettesse di essere utilizzato in ambienti o piattaforme differenti.

Obiettivo del lavoro

L'obiettivo del lavoro è quello utilizzare la definizione di agenti BDI fatta da AgentSpeak per definire un nuovo linguaggio ad agenti al quale, inoltre, si è voluto aggiungere anche una caratteristica di flessibilità. Quest'ultima è stata raggiunta grazie all'utilizzo della libreria tuProlog che ha permesso di definire un linguaggio che possa essere utilizzato da interpreti realizzati su ambienti e piattaforme differenti accomunate dall'utilizzo di questa libreria. Si vuole mostrare, inoltre, come è possibile realizzare un interprete che lavori con il nuovo linguaggio definito.

Benefici dell'approccio scelto

Il beneficio del linguaggio è quindi un'architettura ad agenti che possa essere eseguita sia in un ambiente simulato che in uno reale, per tentare di esprimere tutto il potenziale del modello ad agenti. In questo lavoro verrà presentata una soluzione che implementa l'interprete del linguaggio in un ambiente simulato, il meta-simulatore Alchemist, ma si sarebbe potuta sce-

gliere una qualsiasi altra piattaforma o ambiente sulla quale fosse possibile utilizzare la libreria tuProlog.

Capitolo 2

Stato dell'arte

In questo capitolo sono mostrati alcuni lavori correlati ed i modelli e i linguaggi utilizzati per svolgere il lavoro di tesi. Per ognuno verrà fatta una descrizione per esporre le caratteristiche principali ed un particolare di quelle che sono state utilizzate in questo progetto.

2.1 Lavori correlati

Qui di seguito sono descritti alcuni modelli e linguaggi correlati al lavoro proposto in questo progetto di tesi. Sono presi in esame singolarmente e per ognuno è fatta una presentazione generale e accenni alle loro caratteristiche principali.

2.1.1 JADE

JADE (Java Agent DEvelopement Framework) è un software implementato in Java che semplifica l'implementazione del sistema multi-agente attraverso un middleware che è conforme alle specifiche FIPA e uno strumento grafico che supporta le fasi di debug e distribuzione [Sit. 1].

FIPA è una società di standard IEEE, il cui scopo è la promozione di tecnologie e specifiche di interoperabilità che facilitino l'interworking end-

to-end di sistemi di agenti intelligenti in moderni ambienti commerciali ed industriali.

Un middleware è un software che fornisce servizi per applicazioni che permette agli sviluppatori di implementare meccanismi di comunicazione e di input/output. Viene usato particolarmente in software che necessitano di comunicazione e gestione di dati in applicazioni distribuite.

Un sistema basato su JADE può essere distribuito su diverse macchine (anche con sistemi operativi differenti) e la configurazione può essere controllata da un'interfaccia remota. La configurazione può essere anche cambiata durante l'esecuzione spostando gli agenti da una macchina ad un'altra in base alle necessità.

L'architettura di comunicazione offre uno scambio di messaggi privati di ogni agente flessibile ed efficiente, dove JADE crea e gestisce una coda di messaggi ACL in entrata. È stato implementato il modelllo FIPA completo e i suoi componenti sono stati ben distinti e pienamente integrati: interazione, protocolli, preparazione pacchetti, ACL, contenuto dei linguaggi, schemi di codifica, ontologie e protocollo di trasporto [Sit. 1]. Il meccaniscmo di trasporto, in particolare, è come un camaleonte perchè si adatta ad ogni situazione scegliendo trasparentemente il miglior protocollo disponibile.

2.1.2 SPADE

SPADE (Smart Python multi-Agent Development Environment) è una piattaforma per sistemi multi-agente scritta in Python e basata sui messaggi istantanei (XMPP). Il protocollo XMPP offre una buon architettura per la comunicazione tra agenti in modo strutturato e risolve eventuali problemi legati al design della piattaforma, come autenticazione degli utenti (agenti) o creazione di canali di comunicazione.

Il modello ad agenti è composto da un meccanismo di connessione alla piattaforma, un dispatcher di messaggi e un set di comportamenti differenti a cui il dispatcher da i messaggi. Ogni agente ha un identificativo (JID) e una password per autenticarsi al server XMPP.

La connessione alla piattaforma è gestita internamente tramite il protocollo XMPP, il quale fonisce un meccaniscmo per registrare e autenticare gli utenti al server XMPP. Ogni agente potrà quindi mantenere aperta e persistente uno stream di comunicazioni con la piattaforma.

Ogni agente ha al suo interno un componente dispatcher per i messaggi che opera come un postino: quando arriva un messaggio per l'agente, lo posizione nella corretta casella di posta; quando l'agente deve inviare un messaggio, il dispatcher si occupa di inserirlo nello stream di comunicazione [Sit. 2].

Un agente può avere più comportamenti simultaneamente. Un comportamento è un operazione che l'agente può eseguire usando il pattern di ripetizione. Spade fornisce alcuni comportamenti predefiniti: Cyclic, Periodic (utili per eseguire operazioni ripetitive); One-Shot, Time-Out (usati per eseguire operazioni casuali); Finite State Machine (permette di costruire comportamenti complessi) [Sit. 2]. Quando un messaggio arriva all'agente, il dispatcher lo indirizza alla coda del comportamento corretto: il dispatcher utilizza il template di messaggi di ogni comportamento per capire qual è il giusto destinatario. Quindi un comportamento può definire il tipo di messaggi che vuole ricevere.

2.1.3 Jason

Jason è un interprete per la versione estesa di AgentSpeak che implementa la semantica operazionale di tale linguaggio e fornisce una piattaforma per lo sviluppo di sistemi multi-agente [Sit. 3]. AgentSpeak è uno dei principali linguaggi orientati agli agenti basati sull'architettura BDI. Il linguaggio interpretato da Jason è un'estensione del linguaggio di programmazione astratto AgentSpeak(L). Gli agenti BDI (Belief-Desires-Intentions) forniscono un meccaniscmo per separare le attività di selezione di un piano, fra quelli disponibili, dall'esecuzione del piano attivo, permettendo di bilanciare il tempo per la scelta del piano e quello per eseguirlo.

2.1.4 SARL

SARL è un linguaggio di programmazione ad agenti tipizzato staticamente. SARL mira a fornire le astrazioni fondamentali per affrontare la concorrenza, la distribuzione, l'interazione, il decentramento, la reattività e la riconfigurazione dinamica. Queste funzionalità di alto livello sono adesso considerate i principali requisiti per un'implementazione facile e pratica delle moderne applicazioni software complesse [Sit. 4].

2.1.5 JADEX

Il framework di componenti attivi Jadex fornisce funzionalità di programmazione e di esecuzione per sistemi distribuiti e concorrenti. L'idea generale è di considerare che il sistema sia composto da componenti che agiscono come fornitori di servizi e consumatori [Sit. 5]. Rispetto a SCA (Service Component Architecture) i componenti sono sempre entità attive, mentre in confronto agli agenti la comunicazione è preferibilmente eseguita utilizzando chiamate ai servizi.

2.1.6 ASTRA

ASTRA è un linguaggio di programmazione ad agenti per creare sistemi intelligenti distrubuiti/concorrenti costruiti su Java [Sit. 6].
ASTRA è basato su AgentSpeak(L), ovvero fornisce tutte le stesse funzionalità base, ed inoltre le aumenta con una serie di feature orientate a creare un linguaggio di programmazione ad agenti più pratico.

2.2 Agenti BDI con AgentSpeak

Il modello BDI consente di rappresentare le caratteristiche e le modalità di raggiungimento di un obiettivo secondo il paradigma ad agenti. Gli agenti BDI forniscono un meccanismo per separare le attività di selezione di un piano, fra quelli presenti nella sua teoria, dall'esecuzione del piano attivo,

permettendo di bilanciare il tempo speso nella scelta del piano e quello per eseguirlo.

I **beliefs** sono quindi informazioni dello stato dell'agente, ovvero ciò che l'agente sa del mondo [Bib. 3] il suo insieme è chiamato 'belief base' o 'belief set'.
I **desires** rappresentano tutti i possibili piani che un agente potrebbe eseguire [Bib. 3]. Rappresentano ciò che l'agente vorrebbe realizzare o portare a termine: i *goals* sono desideri che l'agente persegue attivamente ed è quindi bene che tra loro siano coerenti, cosa che non è obbligatoria per quanto riguarda il resto dei desideri.
Le **intentions** identificano i piani a cui l'agente ha deciso di lavorare o a cui sta già lavorando e a loro volta possono contenere altri piani [Bib. 3].

Gli **eventi** innescano le attività reattive, ovvero la loro caratteristica di proattività degli agenti, come ad esempio l'aggiornamento dei beliefs, l'invocazione di piani o la modifica dei goals.

2.2.1 Definizione

AgentSpeak è un linguaggio di programmazione basato su un linguaggio del primo ordine con eventi e azioni [Bib. 1]. Il comportamento degli agenti è dettato da quanto definito nel programma scritto in AgentSpeak. I beliefs correnti di un agente sono relativi al suo stato attuale, l'enviroment e gli altri agenti. Gli stati che un agente vuole determinare sulla base dei suoi stimoli esterni e interni sono i desideri [Bib. 1]. L'adozione di programmi per soddisfare tali stimoli è detta intenzioni.

2.2.2 Descrizione formale

Vengono ora mostrate le definizioni che formalizzano questo linguaggio di programmazione. Di seguito, nelle prime cinque definizioni, viene formalizzato il linguaggio e, nelle restanti, la semantica operazionale.

Le specifche del linguaggio consistono in un set di beliefs e in un set di piani. Quest'ultimi sono sensibili al contesto e richiamati da eventi che permettono la scomposizione gerarchica degli obiettivi e l'esecuzione di azioni.

L'alfabeto del linguaggio formale consiste in variabili, costanti, simboli di funzione, simboli di predicati, simboli di azioni, quantificatori e simboli di punteggiatura. Oltre alle logiche del primo ordine, sono usati '!' (per achievement), '?' (per test), ';' (per operazioni sequenziali), '$\leftarrow$' (per implicazione) [Bib. 1].

Definition 2.1. Se b è un simbolo di predicato e $t_1, \ldots, t_n$ sono termini, allora $b(t_1, \ldots, t_n)$ un atomo di belief. Se $b(t)$ e $c(s)$ sono atomi di belief, allora $b(t) \wedge c(s)$ e $\neg b(t)$ sono beliefs. Un atomo di belief oppure la sua negazione sono riferiti al letterale del belief. Un atomo di belief base sarà chiamato *belief base*.

Definition 2.2. Se g è un simbolo di predicato e $t_1, \ldots, t_n$ sono termini, allora $!g(t_1, \ldots, t_n)$ e $?g(t_1, \ldots, t_n)$ sono goals.

Definition 2.3. Se $b(t)$ è un atomo di belief, $!g(t)$ e $?g(t)$ sono goals, allora $+b(t)$, $-b(t)$, $+!g(t)$, $-!g(t)$, $+?g(t)$, $-?g(t)$ sono eventi di attivazione.

Definition 2.4. Se a è un simbolo di azione e $t_1, \ldots, t_n$ sono termini del primo ordine, allora $a(t_1, \ldots, t_n)$ è un'azione.

Definition 2.5. Se e è un *evento di attivazione*, $b_1, \ldots, b_m$ sono letterali di belief e $h_1; \ldots; h_n$ sono goals o azioni, allora $e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$ è un piano. L'espressione alla sinistra della freccia è la testa del piano e l'espressione alla destra è il corpo del piano. L'espressione sulla destra dei due punti, nella testa del piano, è il contesto. Per convenienza si può definire un corpo vuoto con *true*.

Durante l'esecuzione un agente è composto di un set di beliefs B, un set di piani P, un set di intenzioni I, un set di eventi E, un set di azioni A e un set di funzioni di selezione S, il quale è formato da S_E (funzione di selezione degli eventi), S_O (funzione di selezione del piano), S_I (funzione di selezione dell'intenzione).

Definition 2.6. Un *agente* è formato da $\langle E, B, P, I, A, S_E, S_O, S_I \rangle$, dove E è un set di eventi, B è una 'belief base', P è un set di piani, I è un set di intenzioni, A è un set di azioni. La funzione S_E sceglie un evento da E; la funzione S_O sceglie un piano dal set di quelli applicabili; la funzione S_I sceglie l'intenzione da eseguire dal set I.

Definition 2.7. Il set I è composto da intenzioni, ognuna delle quali è una pila di piani parzialmente istanziati (dove alcune variabili sono state istanziate). Un'intenzione è definita da $[p_1 \ddagger \ldots \ddagger p_z]$, dove p_1 è il fondo dello stack e p_z la testa. Gli elementi sono delimitati da $\ddagger$. Per convenienza $[+!true : true \leftarrow true]$ è detta *true intention* e viene definita con T.

Definition 2.8. Il set E è composto da eventi, oguno delle quali è una tupla $\langle e, i \rangle$, dove e è l'evento di attivazione e i un'intenzione. Se l'intenzione è di tipo *true intention* allora e sarà chiamato *evento esterno*, altrimenti è un *evento interno*.

Definition 2.9. Sia $S_E(E) = \epsilon = \langle d, i \rangle$ e sia $p = e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$, il piano p è rilevante per l'evento e se e solo se esiste un unificatore σ tale per cui $d\sigma = e\sigma$. σ è detto *unificatore rilevante* per ϵ.

Definition 2.10. Un piano p è definito da $e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$ è un *piano applicabile* rispetto ad un evento ϵ se e solo se esite un unificatore rilevante σ per ϵ e esite una sostituzione θ tale che $\forall(b_1 \wedge \ldots \wedge b_m)\sigma\theta$ è una conseguenza logica di B. La composizione $\sigma\theta$ è riferita all'*unificatore applicabile* per l'evento ϵ e θ è riferita alla sostituzione della corretta risposta.

Definition 2.11. Sia $S_O(O_\epsilon) = p$, dove O_ϵ è il set dei piani applicabili per l'evento $\epsilon = \langle d, i\rangle$ e p è $e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$. Il piano p è destinato all'evento ϵ, dove i è la *true intention* se e solo se esiste un *unificatore applicabile* σ per cui $[+!true : true \leftarrow true \ddagger (e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n)\sigma] \in I$.

Definition 2.12. Sia $S_O(O_\epsilon) = p$, dove O_ϵ è il set dei piani applicabili per l'evento $\epsilon = \langle d, [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots; h_n]\rangle$, e p è $+!g(s) : b_1 \wedge \ldots \wedge b_m \leftarrow k_1; \ldots; k_j$. Il piano p è destinato all'evento ϵ se e solo se esiste un *unificatore applicabile* σ tale che $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots; h_n \ddagger (+!g(s) : b_1 \wedge \ldots \wedge b_m)\sigma \leftarrow (k_1; \ldots; k_j)\sigma; (h_2; \ldots; h_n)\sigma] \in I$.

Definition 2.13. Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots; h_n]$. L'intenzione i si dice che è eseguita se e solo se $\langle +!g(t), i\rangle \in E$.

Definition 2.14. Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow ?g(t); h_2; \ldots; h_n]$. L'intenzione i si dice che è eseguita se e solo se esiste una sostituzione θ tale che $\forall g(t)\theta$ è una conseguenza logica di B e i è rimpiazzato da $[p_1 \ddagger \ldots \ddagger (f : c_1 \wedge \ldots \wedge c_y)\theta \leftarrow h_2\theta; \ldots; h_n\theta]$.

Definition 2.15. Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow a(t); h_2; \ldots; h_n]$. L'intenzione i si dice che è eseguita se e solo se $a(t) \in A$, e i è rimpiazzato da $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow h_2; \ldots; h_n]$.

Definition 2.16. Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger p_{z-1} \ddagger g(t) : c_1 \wedge \ldots \wedge c_y \leftarrow true]$, dove p_{z-1} è $e : b_1 \wedge \ldots \wedge b_x \leftarrow !g(s); h_2; \ldots; h_n$. L'intenzione i si dice che è eseguita se e solo se esiste una sostituzione θ tale che $g(t)\theta = g(s)\theta$ e i è rimpiazzato da $[p_1 \ddagger \ldots \ddagger p_{z-1} \ddagger (e : b_1 \wedge \ldots \wedge b_x)\theta \leftarrow (h_2; \ldots; h_n)\theta]$.

Ciclo di ragionamento

Il ciclo di ragionamento è il modo in cui l'agente prende le sue decisioni e mette in pratica le azioni. Esso è composto di otto fasi: le prime tre sono quelle che riguardano l'aggiornamento dei belief relativi al mondo e agli altri agenti, mentre altre descrivono la selezione di un evento che permette l'esecuzione di un'intenzione dell'agente.

a. Percezione ambiente

La percezione effettuata dall'agente all'interno del ciclo di ragionamento è utilizzata per poter aggiornare il suo stato. L'agente interroga dei componenti capaci di rilevare i cambiamenti nell'ambiente [Bib. 3] e di emettere dati consultabili utilizzando opportune interfacce.

b. Aggiornamento beliefs

Ottenuta la lista delle percezioni è necessario aggiornare la 'belief base'. Ogni percezione non ancora presente nel set viene aggiunta e al contrario quelle presenti nel set e che non sono nella lista delle percezioni vengono rimosse [Bib. 3]. Ogni cambiamento effettato nella 'belief base' produce un evento: quelli generati da percezioni dell'ambiente sono detti eventi esterni; quelli interni, rispetto agli altri, hanno associata un'intenzione.

c. Ricezione e selezione messaggi

L'altra sorgente di informazioni per un agente sono gli altri agenti presenti nel sistema. L'interprete controlla i messaggi diretti all'agente e li rende a lui disponibili [Bib. 3]: ad ogni iterazione del ciclo può essere processato solo un messaggio. Inoltre,può essere assegnata una priotità ai messaggi in coda definendo una funzione di prelazione per l'agente.
Prima di essere processati i messaggi passano all'interno di una funzione di selezione che definisce quali messaggi possano essere accettati dall'agen-

te [Bib. 3]. Questa funzione può essere implementata ad esempio per far ricevere solo i messaggi di un certo agente.

d. Selezione evento

Gli eventi rappresentano la percezione del cambiamento nell'ambiente o dello stato interno dell'agente [Bib. 3], come il goal. Ci possono essere vari eventi in attesa ma in ogni ciclo di ragionamento può esserne gestito uno solo, il quale viene scelto dalla funzione di selezione degli eventi che ne seleziona uno dalla lista di quelli in attesa. Se la lista di eventi fosse vuota si passa direttamente alla penultima fase del ciclo di ragionamento [Bib. 3], ovvero la selezione di un'intenzione.

e. Recupero piani rilevanti

Una volta selezionato l'evento è necessario trovare un piano che permetta all'agente di agire per gestirlo. Per fare ciò viene recuperata dalla 'Plan Library' la lista dei piani rilevanti, verificando quali possano essere unificati con l'evento selezionato [Bib. 3]. L'unificazione è il confronto relativo a predicati e termini. Al termine di questa fase si otterrà un set di piani rilevanti per l'evento selezionato che verrà raffinato successivamente.

f. Selezione piano appplicabile

Ogni piano ha un contesto che definisce con quali informazioni dell'agente può essere usato. Per piano applicabile si intendeno quelli che, in relazione allo stato dell'agente, possono avere una possibilità di successo. Viene quindi controllato che il contesto sia una conseguenza logica della 'belief base' dell'agente [Bib. 3]. Vi possono anche essere più piani in grado di gestire un evento ma l'agente deve selezionarne uno solo ed impegnarsi ad eseguirlo.
La selezione viene fatta tramite un'apposita funzione che inoltre tiene conto dell'ordinamento dei piani in base alla loro posizione nel codice sorgente oppure dall'ordine di inserimento. Quando un piano è scelto, viene creata

un'istanza di quel piano che viene inserita nel set delle intenzioni [Bib. 3]: sarà l'istanza ad essere manipolata dall'interprete e non il piano nella libreria.

Ci sono due possibili modalità per la creazione di un'intenzione e dipende dal fatto che l'evento selezionato sia esterno o interno [Bib. 3]. Nel primo caso viene semplicemente creata l'intenzione, altrimenti viene inserita un'altra intenzione in testa a quella che ha generato l'evento, poichè è necessario eseguire fino al completamento un piano per raggiungere tale goal.

g. Selezione intenzione

A questo punto, se erano presenti eventi da gestire, è stata aggiunta un'altra intenzione nello stack. Un agente ha tipicamente più di un'intenzione nel set delle intenzioni che potrebbe essere eseguita, ognuna delle quali rappresenta un diverso punto di attenzione [Bib. 3]. Ad ogni ciclo di ragionamento avviene l'esecuzione di una sola intenzione, la cui scelta è importante per come l'agente opererà nell'ambiente.

h. Esecuzione intenzione

L'intenzione, scelta nello step precedente, non è altro che il corpo di un piano formato da una sequenza di istruzioni, ognuna delle quali, una volta eseguita, viene rimossa dall'istanza del piano. Terminata l'esecuzione un'intenzione, quest'ultima viene restituita al set delle intenzioni a meno che non debba aspettare un messaggio o un feedback dell'azione [Bib. 3]: in questo caso viene memorizzata in una struttura e restituita una volta ricevuta la risposta. Se un'intenzione è sospesa non può essere selezionata per l'esecuzione nel ciclo di ragionamento.

Scambio di messaggi

Lo scambio di messaggi è la comunicazione standard che avviene tra agenti per comunicare tra loro e operare in base al contenuto ricevuto. La comunicazione definita da AgentSpeak utilizza tre parti. La prima è la coda dei

messaggi in input, ovvero una lista contenente tutti i messaggi che il sistema o interprete riceve e che sono destinati all'agente. La seconda è la coda dei messaggi di output che si allunga ogni volta che l'agente vuole inviare un messaggio ad un altro agente. L'ultima è una struttura all'interno della quale vengono memorizzate le intenzioni che sono sospese dall'esecuzione poichè aspettano una risposta dal canale di comunicazione dei messaggi.
L'interprete è il mezzo per il quale i messaggi trasmessi. Esso infatti ha il compito di recuperare tutti i messaggi nella coda in uscita di ogni agente e successivamente recapitarli. Per la consegna viene recuperato l'agente destinatario di ogni messaggio e poi quest'ultimo viene posizionato nella coda di quelli in input dell'agente, in modo tale che possa recuperarne il contenuto al prossimo ciclo di ragionamento.

2.3 Caratteristiche Jason

Jason è la maggiore implementazione di AgentSpeak e, oltre ad implementare le sue semantiche operazionali, lo estende dichiarando il linguaggio per definire gli agenti. Jason aggiunge un set di meccanismi potenti per migliorare le abilità degli agenti ed, inoltre, mira a rendere più pratico il linguaggio di programmazione ad agenti. Alcuni dei meccanismi aggiunti da Jason sono:

- negazione forte;
- gestione del fallimento dei piani;
- atti linguistici basati sulla comunicazione inter-agente;
- annotazioni sulle etichette del piano che possono essere utilizzate mediante funzioni di selezione elaborate;
- supporto per gli ambienti di sviluppo;
- possibilità di eseguire un sistema multi-agente distribuito su una rete;

- funzioni di selezione personalizzabili, funzioni di trust, architettura generale dell'agente;

- estensibilità mediante azioni interne definite dall'utente.

2.4 tuProlog

tuProlog è un interprete Prolog per le applicazioni e le infrastrutture Internet basato su Java. È progettato per essere facilmente utilizzabile, leggero, configurabile dinamicamente, direttamente integrato in Java e facilmente interoperabile. tuProlog è sviluppato e mantenuto da 'aliCE' un gruppo di ricerca dell'Alma Mater Studiorum - Università di Bologna, sede di Cesena. È un software Open Source e rilasciato sotto licenza LGPL.

2.4.1 Caratteristiche tuProlog

tuProlog ha diverse caratteristiche e qui di seguito verranno illustrate solo alcune di esse, ovvero quelle utilizzate all'interno di questo lavoro. Il motore tuProlog fornisce e riconosce i seguenti tipi di predicati:

- predicati built-in: incapsulati nel motore tuProlog;

- predicati di libreria: inseriti in una libreria che viene caricata nel motore tuProlog. La libreria può essere liberamente aggiunta all'inizio o rimossa dinamicamente durante l'esecuzione. I predicati della libreria possono essere sovrascritti da quelli della teoria. Per rimuovere un singolo predicato dal motore è necesssario rimuovere tutta la libreria che contiene quel predicato;

- predicati della teoria: inseriti in una teoria che viene caricata nel motore tuProlog. Le teorie tuProlog sono semplicemente collezioni di clausole Prolog. Le teorie possono essere liberamente aggiunte all'inizio o rimosse dinamicamente durante l'esecuzione.

In questo lavoro è stato utilizzato il motore tuProlog, fornito tramite la libreria Java 'alice.tuprolog', e le funzionalità collegate per monitorare e modificare la teoria di ogni agente.

Una peculiare modalità di utilizzo di tuProlog sfruttata in questo progetto è stata la registrazione di oggetti Java all'interno della teoria dell'agente. In questo modo è possibile utilizzare uno stesso oggetto sia nella parte Java che in quella tuProlog. Nello specifico, come verrà mostrato nella parte relativa all'implementazione, questa funzionalità è stata utilizzata per registrare la classe stessa dell'agente in tuProlog e consentendo l'invocazione di metodi implementati lato Java direttamente dalla teoria dell'agente.

2.5 Alchemist

Alchemist è un simulatore per il calcolo pervasivo, aggregato e ispirato alla natura. Esso fornisce un ambiente di simulazione sul quale è possibile sviluppare nuove incarnazioni, cioè nuove definizioni di modelli implementati su di esso. Ad oggi sono disponibili le funzionalità per:

- simulare un ambiente bidimensionale;
- simulare mappe del mondo reale, con supporto alla navigazione e importazione di tracciati in formato gpx;
- simulare ambienti indoor importando immagini in bianco e nero;
- eseguire simulazioni biologiche utilizzando reazioni in stile chimico;
- eseguire programmi Protelis, Scafi, SAPERE (scritti in un linguaggio basato su tuple come LINDA).

2.5.1 Meta-modello

Il meta-modello di Alchemist può essere compreso osservando la figura 2.1.

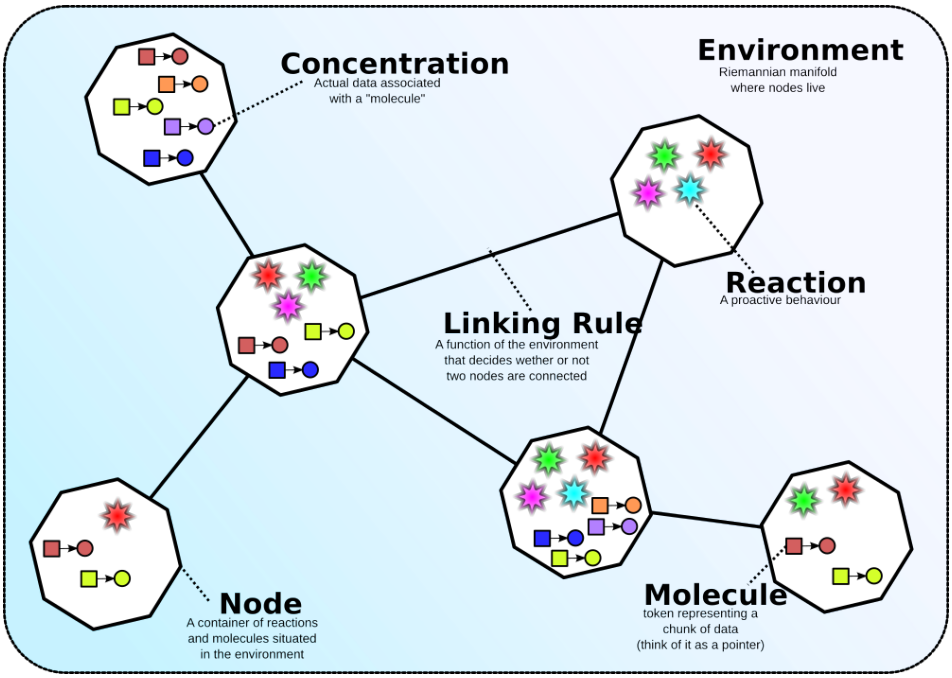

Figura 2.1: Illustrazione meta-modello di Alchemist

L'***Environment*** è l'astrazione dello spazio ed è anche l'entità più esterna che funge da contenitore per i nodi. Conosce la posizione di ogni nodo nello spazio ed è quindi in grado di fornire la distanza tra due di essi e ne permette inoltre lo spostamento.

È detta ***Linking rule*** una funzione dello stato corrente dell'environemnt che associa ad ogni nodo un ***Vicinato***, il quale è un entità composta da un nodo centrale e da un set di nodi vicini.

Un ***Nodo*** è un contenitore di molecole e reazioni che è posizionato all'interno di un environment.

La ***Molecola*** è il nome di un dato, paragonabile a quello che rappresenta il nome di una variabile per i linguaggi imperativi. Il valore da associare ad una molecola è detto ***Concentrazione***.

Una ***Reazione*** è un qualsiasi evento che può cambiare lo stato dell'environment ed è definita tramite una distribuzione temporale, una lista di condizioni e una o più azioni.

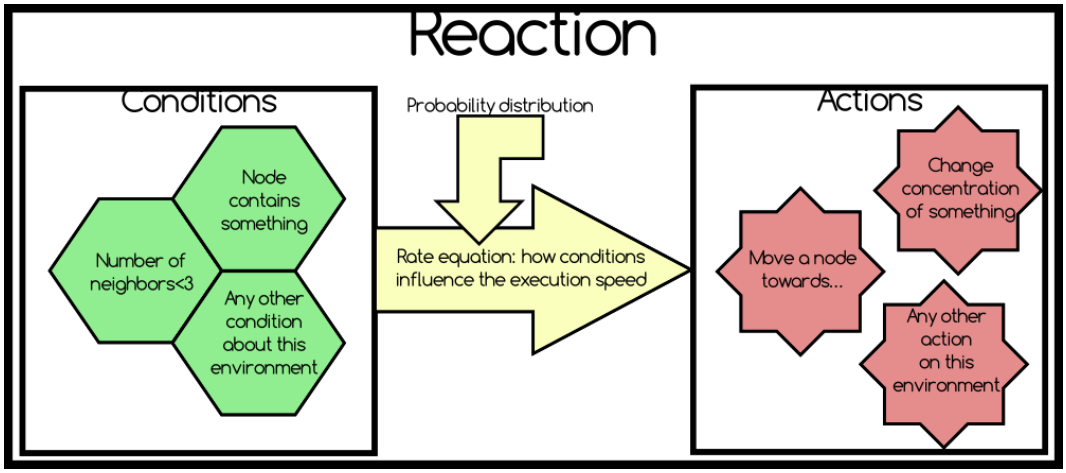

Figura 2.2: Illustrazione modello reazione di Alchemist

La frequenza con cui avvengono dipende da:

- un parametro statico di frequenza;
- il valore di ogni condizione;
- un'equazione di frequenza che combina il parametro statico e il valore delle condizioni restituendo la frequenza istantanea;
- una distribuzione temporale.

Ogni nodo contiene un set di reazioni che può essere anche vuoto.

Per comprendere meglio il meccanismo di una reazione si può osservare la figura 2.2.

Una ***Condizione*** è una funzione che prende come input l'environment corrente e restituisce come output un booleano e un numero. Se la condizione non si verifica, le azioni associate a quella reazione non saranno eseguite. In relazione a parametri di configurazione e alla distribuzione temporale, una condizione potrebbe influire sulla velocità della reazione.

La ***Distribuzione temporale*** indica il numero di eventi, in un dato intervallo di tempo, generati da Alchemist e che innescano la verifica delle condizioni che possono portare alla potenziale esecuzione delle azioni.

Un'***Azione*** è la definizione di una serie di operazioni che modellano un cambiamento nel nodo o nell'environment.

In Alchemist un'incarnazione è un'istanza concreta del meta-modello appena descritta e che implementa una serie di componenti base come: la definizione di una molecola e del tipo di dati della concentrazione, un set di condizioni, le azioni e le reazioni. Incarnazioni diverse possono modellare universi completamente differenti.

2.5.2 Esempi (?)

2.6 LINDA

LINDA è un modello di coordinazione e comunicazione tra diversi processi paralleli che operano su oggetti immagazzinati e recuperati dalla memoria associativa, virtuale, condivisa. Nel modello diverse primitive operano su una sequenza ordinata di oggetti, le 'tuple', che vengono aggiunte ad un linguaggio sequenziale e una memoria associativa logica globale, detta spazio di tuple, nel quale i processi immagazzinano e recuperano le tuple.

Il modello LINDA originale definisce quattro operazioni consentite sulle tuple e lo spazio di tuple:

- *in*: legge una tupla e la consuma dallo spazio di tuple
- *rd*: legge una tupla senza consumarla dallo spazio di tuple
- *out*: inserisce una tupla nello spazio di tuple
- *eval*: crea un processo per valutare le tuple e lo inserisce nello spazio di tuple.

LINDA è un modello di coordinazione utilizzato per definire altri modelli e tecnologie di coordinazione, dove gli agenti distribuiti interagiscono e si coordinano tramite scambio di messaggi utilizzando spazi di informazione condivisa e sfruttando la comunicazione generativa.

Di seguito è descritta un'estensione del modello appena descritto chiamata Spatial Tuples dove le informazioni base assumono una posizione e un'estensione nello spazio fisico.

2.6.1 Spatial Tuples

Spatial Tuples è un estensione del modello base di tuple per i sistemi distribuiti multi-agente, dove

- le tuple sono posizionate nel mondo fisico e si possono muovere;
- il comportamento delle primitive di coordinamento può dipendere dalle proprietà spaziali del coordinamento degli agenti;
- lo spazio di tuple può essere concepito come un livello virtuale che aumenta la realtà fisica.

Spatial Tuples supporta esplicitamente la consapevolezza dello spazio e la coordinazione basata sullo spazio dell'agente in scenari di calcolo pervasivo.

Questo modello può risultare molto utile in scenari dove gli utenti si spostano all'interno di un ambiente fisico aumentato e devono coordinarsi con altri utenti, che siano persone o agenti.

Modello e linguaggio

Spatial Tuples si occupa prima di tutto di tuple spaziali. Una tupla spaziale è una tupla associata ad un'informazione spaziale. Le informazioni spaziali possono essere, ad esempio, GPS, amministrative, organizzative: in ogni caso la tupla viene associata a qualche luogo o regione dello spazio fisico. Una tupla spaziale decora lo spazio fisico e può funzionare come meccanismo base per aumentare la realtà con informazioni di ogni sorta. Una volta che la tupla è associata ad una regione o posizione, le sue informazioni possono essere pensate come proprietà attribuite a quella porzione di spazio fisico. Accedendo alle tuple con i meccanismi di Spatial Tuples, l'informazione può essere osservata da qualsiasi agente che si occupa dello spazio fisico specifico

in modo tale da comportarsi di conseguenza.
Inoltre, una tupla può essere associata anche ad un componente situato. In questo caso, se il componente cambia la sua posizione nel tempo, finchè non viene rimossa, anche la tupla si sposterà con esso.

In Spatial Tuples viene introdotto un linguaggio di descrizione dello spazio per specificare le informazioni spaziali che decorano le truple. Questo linguaggio è ortogonale al linguaggio di comunicazione e ha lo scopo di fornire l'ontologia di base che definisce i concetti spaziali.

Primitive spaziali

Gli operatori base di Spatial Tuples sono: $out(t), rd(tt), in(tt)$ dove t è la tupla e tt è un template di tupla. Il funzionamento delle primitive è il seguente:

- *out*, permette di associare la tupla ad una regione o posizione;
- *rd*, cerca le tuple che corrispondono al template e ne ritorna una copia;
- *in*, come *rd*, cerca le tuple che corrispondo al template ma poi ne restituisce una consumandola dalla sorgente.

Le primitive *rd* e *out* sono dette 'getter' e, in Spatial Tuples, sono:

- sospensive, se non ci sono tuple che fanno match con il template l'operazione è bloccata finchè non viene trovata una tupla
- non deterministiche, se vi sono più tuple che fanno match con il template una è scelta in modo non deterministico.

Capitolo 3

AgentSpeak in tuProlog

Nel capitolo precedente è stato descritto lo stato attuale di lavori correlati che utilizzano il modello ad agenti BDI per costruirne altri più complessi ed espressivi o implementano linguaggi basati sugli agenti. Inoltre, è stato mostrato lo stato dell'arte delle tecnologie che sono state utilizzate.

In questo lavoro di tesi si è voluto definire un nuovo linguaggio che fosse *'platform indipendent'*, ovvero indipendente dall'ambiente sul quale viene utilizzato: è stato definito al pari di una libreria, senza nessun riferimento all'ambiente. Nei capitoli successivi verrà mostrato come, partendo dal linguaggio, è stata colmata la distanza con l'ambiente scelto.

3.1 Definizione linguaggio

Essendo tuProlog un interprete che opera su piattaforme differenti, si è voluto utilizzarlo nella definizione del linguaggio per permettere di utilizzare quest'ultimo facilemente, potendo sfruttare la libreria tuProlog per colmare la distanza tra l'ambiente e il linguaggio.

Seguendo quella che è la struttura di AgentSpeak, è stato formalizzato questo linguaggio di programmazione ad agenti, e, di seguito, sono mostrate le definizioni.

Definition 3.1. Se b è un simbolo di predicato e $t_1, \ldots, t_n$ sono termini, allora $belief(b(t_1, \ldots, t_n))$ è un atomo di belief. Se $b(t)$ e $c(s)$ sono atomi di belief, allora $b(t) \wedge c(s)$ e $\neg b(t)$ sono beliefs. Un atomo di belief oppure la sua negazione sono riferiti al letterale del belief. Un atomo di belief base sarà chiamato *belief base*.

Definition 3.2. Se g è un simbolo di predicato e $t_1, \ldots, t_n$ sono termini, allora $achievement(g(t_1, \ldots, t_n))$ e $test(g(t_1, \ldots, t_n))$ sono *goals*.

Definition 3.3. Se $b(t)$ è un atomo di belief e $g(t)$ un goal, allora $onAddBelief(b(t))$, $onRemoveBelief(b(t))$, $onReceivedMessage(b(t))$, $onResponseMessage(b(t))$, $concurrent(achievement(g(t)))$, $concurrent(test(g(t)))$ sono *eventi di attivazione*.

Definition 3.4. Se a è un simbolo di azione e $t_1, \ldots, t_n$ sono termini del primo ordine, allora $a(t_1, \ldots, t_n)$ è un'azione.

Definition 3.5. Se e è un *evento di attivazione*, $b_1, \ldots, b_m$ sono belief o guardie e $h_1, \ldots, h_n$ sono goals o azioni, allora ' $\leftarrow'$ $(e, [b_1, \ldots, b_m], [h_1, \ldots, h_n])$ è un piano.

Definition 3.6. Ogni intenzione ha al suo interno uno stack di piani parzialmente istanziati, ovvero dove alcune delle variabili sono state istanziate. Un'intenzione è definita come $intention(i, [p_1, \ldots, p_n])$, dove i è l'identificativo univoco dell'intenzione e $[p_1, \ldots, p_n]$ è lo stack formata da azioni, belief o goal: p_1 è la coda e p_n è la testa.

Definition 3.7. Un *agente* è formato da $\langle B, P, I, A, S_O, S_I \rangle$, dove B è una 'belief base', P è un set di piani, I è un set di intenzioni, A è un set di azioni. La funzione S_O sceglie un piano dal set di quelli applicabili; la funzione S_I sceglie l'intenzione da eseguire dal set I.

Definition 3.8. Dato un evento ϵ ed un piano $p = \text{`} \leftarrow' (e, [b_1, \ldots, b_m], [h_1, \ldots, h_n])$, allora p è rilevante per l'evento ϵ se e solo se esite un unificatore σ tale per cui $d\sigma = e\sigma$. σ è detto *unificatore rilevante* per ϵ.

Definition 3.9. Un piano p è definito da $\text{`} \leftarrow' (e, [b_1, \ldots, b_m], [h_1, \ldots, h_n])$ è un *piano applicabile* rispetto ad un evento ϵ se e solo se esite un identificatore rilevante σ per ϵ e esite una sostituzione θ tale che $\forall(b_1, \ldots, b_m)\sigma\theta$ è una conseguenza logica di B.

3.2 API del linguaggio

Il linguaggio appena definito è ciò che è messo a disposizione del programmatore dell'agente per descrivere il suo comportamento. Oltre questo, sono state definite altre sintassi che permettano al programmatore di gestire ogni evento o situazione per l'agente. Qui di seguito sono citate regole, variabili e fatti del linguaggio:

- $init :- \ldots$
- $self(A)$.
- $agent$
- $node$
- $addBelief(B)$.
- $removeBelief(B)$.
- $onAddBelief(B) :- \ldots$
- $onRemoveBelief(B) :- \ldots$
- $onReceivedMessage(S, M) :- \ldots$
- $achievement(t)$.

- $test(t)$.
- $concurrent(t)$.
- $belief(position(X, Y))$.
- $belief(distance(A, ND, OD))$. oppure $belief(distance(A, ND))$.

Qui di seguito vengono analizzate ed esposte. Per ogni regola è lasciata l'implementazione del corpo al programmatore dell'agente.

La regola '$init$' è messa a disposizione per permettere di far effettuare una configurazione iniziale dell'agente. Infatti, questa regola verrà invocata solo ed esclusivamente la prima volta che viene azionato l'agente, al posto del ciclo di ragionamento. In questo modo il programmatore dell'agente è in grado di far eseguire all'agente una serie di operazioni iniziali per impostare la 'belief base' dell'agente.

Il fatto '$self(A)$' permette all'agente di recuperare il suo nome. In questo modo il nome dell'agente può essere recuperato anche all'interno della teoria tuProlog.

I due letterali '$agent$' e '$node$' sono due variabili di tuProlog alle quali sono collegati gli oggetti dell'agente e del nodo, implementati nell'ambiente su cui si è scelto di utilizzare il linugaggio. Se costruiti correttamente, dalla teoria dell'agente sarà possibile richiamare metodi implementati nella classe corrispondente. La variabile '$agent$' fa riferimento all'oggetto dell'agente stesso, mentre '$node$' si riferisce all'oggetto che rappresenta lo spazio sul quale l'agente è inserito. In questo modo possono essere gestite le azioni interne ed esterne dell'agente.

Le regole '$addBelief(B)$' e '$removeBelief(B)$' sono utilizzabili per aggiungere o rimuovere elementi dalla 'belief base'. Il loro utilizzo scatena un evento che va ad invocare '$onAddBelief(B)$', '$onRemoveBelief(B)$'. Più precisamente '$onAddBelief(B)$' viene invocato quando viene aggiunto un belief, mentre '$onRemoveBelief(B)$' è chiamato in seguito alla rimozione di un belief dalla 'belief base'. In entrambi i casi la variabile B corrisponde al

belief inserito o rimosso.
Diversamtene, quando viene letto un messaggio ricevuto da un altro agente (o anche da se stesso), è invocato '$onReceivedMessage(S, M)'$, dove S rappresente il mittente e M il contenuto del messaggio, che consente all'agente di reagire quando viene letto un messaggio tra quelli presenti nella sua coda di ingresso.

Come visto precedentemente nella Definizione 3.2, i letterali '*achievement*', '*test*' sono utilizzati per impostare dei goal nell'agente. Ciò che viene scatenato è l'inserimento della serie di operazioni definita dal goal in testa allo stack dell'intenzione. In combinazione, i due letterali appena citati possono essere usati in combinazione con '*concurrent*', mostrato nella Definizione 3.3, che permette di inserire le operazioni definite nel goal in una nuova intenzione. In questo modo, la nuova intenzione può essere eseguita in modo concorrente o parallelo rispetto a quella 'padre'.

Per rendere disponibile al programmatore dell'agente varie possibilità per accedere a informazioni quali la posizione dell'agente e la distanza degli altri agenti rispetto alla propria posizione, sono utilizzati due belief che saranno aggiornati direttamente dall'ambiente sul quale viene utilizzato il linguaggio. La posizione dell'agente viene aggiornata una volta per ogni ciclo di ragionamento e, al termine, sono modificati anche i valori dei belief appena citati.
Per quanto riguarda la posizione dell'agente, potrà essere invocato '$belief(position(X, Y))'$ dove X è la coordinata relative alle ascisse o longitudine e Y è la coordinata relativa alle ordinate o latitudine.
La distanza da altri agenti può essere molto utile per far scegliere all'agente di effettuare o meno una certa azione. Se nella lista del vicinato entra un nuovo agente viene inserito il belief '$belief(distance(A, ND))'$, dove A è il nome dell'agente nel vicinato e ND è la distanza che li separa. Se, invece, un agente era già nella lista del vicinato e vi rimane, allora viene inserito il belief '$belief(distance(A, ND, OD))'$, dove A è il nome dell'agente nel vicinato, ND è la nuova distanza che li separa e OD è la distanza che li divideva precedentemente.

3.2.1 Gestione intenzioni

Le intenzioni sono la modalità con cui l'agente opera le sue azioni. Come descritto in precedenza, nel ciclo di ragionamento alla sezione 2.2.2, l'agente esegue una serie di passi che portano all'esecuzione di un'azione. Qui di seguito è descritto come avviene il ciclo di ragionamento utilizzando questo linguaggio. La spiegazione terrà conto solamente degli aspetti relativi alla parte tuProlog e quindi sarà incompleta fino al raggiungimento della sezione 4.2. Le funzioni di selezione per i piani applicabili e le intenzioni non sono trattate in questa parte, poichè sono relative all'implementazione dell'interprete.

L'agente lato tuProlog definisce il suo comportamento tramite una serie di regole e fatti che gli permettono di reagire ad eventi sia esterni che interni. Una percezione dell'ambiente può essere scatenata ad esempio da uno spostamento o una modifica della 'belief base': quando questo avviene l'ambiente sul quale è utilizzato il linguaggio invoca una delle regole che, se implementata correttamente nella teoria dell'agente, consente all'agente di reagire all'evento. Un altro tipo di input che può ricevere l'agente è la ricezione di un messaggio. In tuProlog l'agente può reagire alla lettura del contenuto del messaggio poichè l'implementazione e la gestione delle code e la selezione dei messaggi viene fatta dall'interprete.

La frequenza dell'esecuzione del ciclo di ragionamento dipende dall'ambiente sul quale viene utilizzato il linguaggio. All'interno del ciclo, una volta selezionato il piano applicabile per l'evento avvenuto, l'interprete invoca delle regole per ottenere la lista delle operazioni presenti nel corpo del piano (o regola) per poterle inserire nell'intenzione. Le regole per recuperare la lista eseguono per ogni elemento del corpo una lettura e un'inserimento all'interno di una lista, la quale poi viene restituita.

La creazione dell'intenzione viene fatta dall'interprete ma salvata come fatto nella teoria dell'agente. Come descritto nella definizione 3.6, l'intenzione $intention(id, [op_1, \ldots, op_n])$ è composta da un identificativo univoco id e da una lista di operazioni $[op_1, \ldots, op_n]$. All'interno della teoria dell'agente

possono essere presenti più intenzioni contemporaneamente ma ad ogni ciclo di ragionamento solo una verrà selezionata per l'esecuzione. Come detto precedentemente, anche la selezione dell'intenzione è gestita dall'interprete ma, lato tuProlog viene gestita l'esecuzione. Infatti, l'interprete si limita a invocare la regola $execute(I)$ all'interno della quale viene gestita l'esecuzione della prima operazione sullo stack dell'intenzione con identificativo I, ovvero quella che è stata precedentemente selezionata.
La regola *execute* si occupa di recuperare l'intenzione riferita all'identificativo passato e quindi prendere la testa dello stack delle operazioni. Quest'ultima viene valutata ed in base alla sua natura vengono eseguite azioni diverse:

- le azioni vengono eseguite direttamente;
- i goal vengono recuperati lo stack di operazioni collegate viene successivamente aggiunto in testa all'intenzione di cui faceva parte il goal;
- i goal espressi all'interno di *concurrent* creano una nuova intenzione che potrà essere eseguita in parallelo rispetto a quella da cui ha avuto origine la chiamata al goal.

3.2.2 Estensione Spatial Tuples

Il linguaggio appena descritto è stato esteso per permettere di utilizzare il modello Spatial Tuples. Sono state quindi inserite le seguenti regole:

- $writeTuple(T)$.
- $readTuple(TT)$.
- $takeTuple(TT)$.
- $onResponseMessage(M) :- \ldots$

Le regole dell'elenco sono tutte riferite all'inserimento, all'interno del linguaggio, del modello di coordinazione LINDA e più precisamente del modello Spatial Tuples. Con questa estensione, viene data la possibilità agli agenti di

poter inserire e recuperare informazioni posizionate nello spazio. Le regole messe a disposizione mappano le primitive dei modelli che vogliono implementare '*in*', '*rd*', '*out*' rispettivamente in '$writeTuple(T)'$, '$readTuple(TT)'$, '$takeTuple(TT)'$ dove T è la tupla da inserire e TT è il template da ricercare.
Utilizzando '$writeTuple(T)'$ il programmatore è in grado di inserire informazioni posizionate nello spazio degli agenti e con le quali gli stessi agenti possono interagire. Per leggere le informazioni si possono utilizzare due diverse modalità: '$readTuple(TT)'$ e '$takeTuple(TT)'$. Nel primo caso viene utilizzato il template passato per confrontarlo con le tuple nell'intorno dell'agente e se ci sono risultati che combaciano con il template allora uno di questi viene restituito. Per quanto riguarda invece '$takeTuple(TT)'$, si comporta ugualmente per quanto riguarda la ricerca della tupla con il template ma poi, una volta trovati i risultati ne sceglie uno e prima di restituirlo lo elimina dallo spazio di tuple in cui era presente.
Entrambe le due modalità di getter seguono la semantica standard dei modelli basati su tuple, e quindi sono:

- sospensive: se non ci sono tuple che si abbinano al template l'operazione è bloccata finchè non viene trovata una tupla;

- non deterministiche: se ci sono più tuple che si abbinano al template una è scelta in modo non deterministico.

Per dare la possibilità di gestire la risposta e gestire la tupla restituita è stata introdotta '$onResponseMessage(M)'$ che viene invocata ogni qualvolta che una tupla viene restituita dallo spazio di tuple. Il contenuto M è la tupla incapsulata in un belief in modo che si possano gestire tuple provenienti da diversi spazi di tuple e con contenuti differenti.

3.3 Esempi linguaggio

In questa sezione verranno mostrate alcuni casi d'uso del linguaggio appena descritto. Nello specifico verrà mostrato un primo scenario dove sono stati configurati gli agenti per realizzare un semplice scambio di messaggi (o Ping Pong). Nel secondo esempio, invece, viene illustrato come poter utilizzare l'estensione Spatial Tuples supportata dal linguaggio.

Ping Pong

In questo primo esempio è presentato il problema del Ping Pong. In questo esempio sono definiti due agenti, Ping e Pong, ognuno dei quali risponde ad un messaggio ricevuto. L'agente Ping, alla ricezione del messaggio '*pong*' da parte dell'agente Pong risponderà con un messaggio '*ping*'. Viceversa, l'agente Pong, alla ricezione del messaggio '*ping*' da parte dell'agente Ping risponderà con un messaggio '*pong*'.

Per far iniziare lo scambio di messaggi è stato utilizzato 'init' per impostare all'interno di uno dei due agenti, nello specifico l'agente Ping, un'intenzione iniziale. In questo modo, al primo ciclo di ragionamento, l'agente eseguirà l'intenzione e invierà il primo messaggio.

```
init :-
  addBelief(intention(0,[iSend('pong_agent','ping')])),
  agent <- insertIntention(0).

onReceivedMessage(S,pong) :-
  iSend(S, ping).
```

Codice sorgente 3.1: Agente Ping

In entrambe le teorie dei due agenti è stata richiamata '$iSend(S, M)$', dove S è il destinatario e M è il messaggio, che è un'azione interna dichiarata e gestita nell'ambiente sul quale è utilizzato il linguaggio. Nel Codice sorgen-

te 3.1 viene inviato all'agente Pong il messaggio 'ping', mentre nel Codice sorgente 3.2 il messaggio inviato all'agente Ping è 'pong'.

```
onReceivedMessage(S,ping) :-
  iSend(S, pong).
```

Codice sorgente 3.2: Agente Pong

Message passing through Spatial Tuples

In questo esempio viene mostrato come possono essere utilizzate le primitive del modello Spatial Tuples incorporate nel linguaggio descritto in precedenza. Nello specifico viene mostrato come tre agenti (Alice, Bob e Carl) comunicano tra loro inserendo messaggi negli spazi di tuple a loro vicini, usandoli come 'lavagna'. L'agente Alice nel suo ciclo di configurazione, esegue due scritture sulla 'lavagna' (spazio di tuple) inserendo messaggi per Bob e Carl e successivamente effettua altre due richieste allo spazio di tuple richiedendo due messaggi a lei destinati senza conoscerne il contenuto. Una volta ricevuti i messaggi non fa niente.

```
init :-
  writeTuple(blackboard,msg(bob,hello)),
  writeTuple(blackboard,msg(carl,hello)),
  takeTuple(blackboard,msg(alice,X)),
  takeTuple(blackboard,msg(alice,X)).

onResponseMessage(msg(X,Y)) :- true.
```

Codice sorgente 3.3: Alice

L'agente Bob, nel cuo ciclo di configurazione, effettua una richiesta allo spazio di tuple per ricevere messaggi a lui destinati. Inoltre, nella sua teoria, è definito un comportamento in caso di ricezione del messaggio: manda ad Alice lo stesso messaggio che ha ricevuto.

```
init :-
  takeTuple(blackboard,msg(bob,X)).

onResponseMessage(msg(bob,X)) :-
  writeTuple(blackboard,msg(alice,X)).
```

Codice sorgente 3.4: Bob

Come Bob, l'agente Carl esegue lo stesso comportamento di Bob.

```
init :-
  takeTuple(blackboard,msg(carl,X)).

onResponseMessage(msg(carl,X)) :-
  writeTuple(blackboard,msg(alice,X)).
```

Codice sorgente 3.5: Carl

Capitolo 4

AgentSpeak in tuProlog su Alchemist

In questo capitolo verrà esposta la parte di implementazione mancante nel capitolo precedente. Più precisamente è descritto come è stato scelto di implementare il modello ad agenti su Alchemist, fornendo un'analisi del mapping, e di come è stato utilizzato il linguaggio per definire l'interprete, scendendo nel dettaglio di come è stata realizzata la gestione delle intenzioni, lo spostamento dell'agente e l'estensione Spatial Tuples.

La scelta della piattaforma è ricaduta su Alchemist poichè fonisce un meta-modello molto adattabile a vari ambiti applicativi e una struttura di simulazione già consolidata ed efficace.
Come detto in precedenza è possibile realizzare implementazioni del modello ad agenti utilizzando il linguaggio definito in questo lavoro di tesi anche sfruttando altre piattaforme sulle quali lavorare con la libreria tuProlog.

4.1 Mapping modelli

In precedenza sono stati descritti il modello ad agenti e il meta-modello di Alchemist. Ora, dopo aver definito il linguaggio, per procedere all'implementazione è necessario capire quale risulta il migliore modo, in termini di

performace e espressività, per unire i due modelli. In questa fase si vuole quindi pensare come realizzare sul meta-modello fornito da Alchemist il modello ad agenti cercando eventuali incongruenze o opportunità per massimizzare il risultato.

Si è partiti analizzando le entità del meta-modello e per ognuna è stato posto l'interrogativo sul fatto che potesse essere un'agente. Fin da subito sono state ritenute inadatte l'Environment e la Molecola: il primo perchè è esso stesso lo spazio e non avrebbe potuto rappresentare lo spazio degli agenti, mentre le molecole perchè forniscono un livello di dettaglio troppo elevato e non hanno una struttura per che consente di contenere lo stato dell'agente.

Le entità rimaste da analizzare sono quindi il Nodo e la Reazione. Mappando il Nodo come agente ne deriva che l'Environment corrisponderà allo spazio degli agenti mentre, all'interno dell'agente, le Molecole e le Concentrazioni potranno essere utilizzate per gestire la 'belief base' e le reazioni saranno riferite ai piani, utilizzando le Condizioni come clausola per scatenare le Azioni. Questo tipo di mapping consente di realizzare simulazioni di sistemi non complessi, in cui agenti allo stesso livello operano e comunicano tra loro.

Posizionando l'agente nella Reazione, quindi più internamente rispetto al precedente mapping, il Nodo sarà quindi un contenitore di agenti e l'Environment lo spazio nel quale si muovono i gruppi di agenti. Ogni agente avrà il riferimento ad una Condizione e ad una Azione: quest'ultima conterrà il ciclo di ragionamento dell'agente mentre la Condizione, sempre vera, ne determinerà la frequenza di esecuzione. Utilizzando questa seconda ipotesi sarà possibile realizzare simulazioni di sistemi complessi, nei quali dei nodi, che potrebbero essere dispositivi mobili (ad esempio cellulari), si muovo nello spazio ed ognuno dei quali al suo interno contiene un gruppo di agenti che possono interagire sia internamente che esternamente.

Dopo aver analizzato le due possibili alternative presentate, è stato scelto il mapping in cui l'agente è posizionato nella Reazione poichè permette una

maggiore espressività e un'apertura verso più scenari applicativi.

4.2 Descrizione interprete linguaggio

Una volta scelto il mapping si è iniziato lo sviluppo dell'interprete del linguaggio sul meta-simlatore al fine di creare una 'incarnazione', ovvero il nome con cui sono chiamate le implementazioni dei modelli in Alchemist.

Alchemist è un meta-simulatore che, proprio per la sua natura di simulatore, ha un meccanismo di generazione degli eventi e fornisce quindi la possibilità di gesire lo scheduling dei cicli di ragionamento degli agenti. Più precisamente, in Alchemist è possibile definire una distribuzione temporale per ogni reazione, la quale nell'incarnazione che si vuole realizzare e secondo il mapping scelto corrisponde ad un agente. È quindi possibile decidere quante volte viene programmata l'iterazione del ciclo di ragionamento di ogni singolo agente. La scelta fatta per gestire la distribuzione temporale è ricaduta sull'utilizzo di un pettine di Dirac che è una distribuzione periorica degli eventi costruita da una somma di delta di Diract, la quale è una funzione generalizzata che dipende da un parametro reale utilizzata per rappresentare dei picchi, gli eventi.

Si è pensato come organizzare le invocazioni delle regole base, definite dal linguaggio, per migliorare l'usabilità l'interprete. È stato deciso di creare una classe astratta che raggruppasse le implementazioni delle funzionalità principali dell'agente, le quali saranno utili al programmatore dell'agente nella realizzazione delle classi specifiche degli agenti poichè sarà sufficiente richiamare queste funzioni dove non necessari comportamenti specifici.

4.2.1 Implementazione del linguaggio

Il primo passo è stato quello di definire la libreria dell'agente, ovvero l'implementazione delle chiamate messe a disposizione del programmatore dell'agente. Sono state quindi definite, utilizzando le funzionalità base di tuProlog, le regole per aggiungere o rimuovere un belief, per eseguire un'intenzione e

per recuperare la lista di operazioni dal corpo di un piano, quest'ultima solo per uso interno. La definizione di queste regole è stata necessaria per permettere la gestione dei cicli di ragionamento, cioè per consentire ad Alchemist di riprendere il controllo alla fine del ciclo di ragionamento di ogni agente evitando così che un agente possa eseguire le sue operazioni entrando in un loop infinito. Ad esempio la definizione per l'aggiunta e la rimozione dei belief è quella mostrata nel Codice sorgente 4.1 nella quale sono eseguite due operazioni: la prima riguarda l'effettiva modifica della 'belief base' con il belief passato, mentre la seconda inserisce un belief fittizio recuperato dall'interprete per permettere di innescare l'evento di modifica del relativo belief al prossimo ciclo di ragionamento.

```
addBelief(B) :-
  assertz(belief(B)),
  assertz(added_belief(B)).

removeBelief(B) :-
  retract(belief(B)),
  assertz(removed_belief(B)).
```

Codice sorgente 4.1: Implementazione regole modifica della 'belief base'

Una modalità analoga è stata utilizzata per gestire anche altri eventi, sempre per poter separare l'invocazione della regola dall'attivazione dell'evento. Gli eventi in questione sono quelli relativi alla realizzazione dell'estensione Spatial Tuples e che sono stati implementati come mostrato nel Codice sorgente 4.2. In questo caso sono aggiunti dei fatti il cui contenuto è la tupla o il template da utilizzare nella richiesta verso lo spazio di tuple.

```
writeTuple(T) :-
  assertz(write(T)).

readTuple(T) :-
  assertz(read(T)).
```

```
takeTuple(T) :-
  assertz(take(T)).
```

Codice sorgente 4.2: Implementazione regole estensione Spatial Tuples

4.2.2 Invocazione regole ed esecuzione intenzioni

Terminata la definizione delle regole per la gestione degli eventi è stata aggiunta la libreria 'alice.tuprolog' dalla quale è stato importato all'interno della classe astratta dell'agente il motore tuProlog per la realizzazione delle invocazioni verso la teoria dell'agente. Ogni agente ha un motore tuProlog al cui interno è caricata la libreria che definisce l'implementazione del linguaggio e una teoria specifica scritta dal programmatore dell'agente che descrive il comportamento dell'agente. L'agente nel meta-modello è mappato con la Reazione ma implementativamente il ciclo di ragionamento dell'agente, ovvero il cuore, risiederà nell'Azione e quindi la classe astratta dell'agente sarà un'estensione dell'interfaccia relativa a tale entità. Sono stati definiti due costruttori, uno che imposta solamente il nome dell'agente e il generatore random per l'agente e un altro che prende in ingresso anche la reazione a cui è riferito l'agente e che è utilizzata per ottenere la distribuzione temporale.

Per fare un'invocazione vengono utilizzate alcune classi messe a disposizione dalla libreria 'alice.tuprolog' che sono descritte qui di seguito.
La prima cosa da fare è costruire il template del fatto o della regola che si vuole ottenere, sfruttando le classi della libreria tuProlog, e per fare questo si possono utilizzare due modi: creazione del termine o composizione della struttura del termine. Nel primo caso viene invocata la funzione statica $Term.createTerm(t)$ dove il parametro passato è una stringa che descrive la struttura del template. Diversamente, la composizione della struttura del termine utilizza la classe Struct. Per creare un oggetto di questo tipo si devono definire almeno un funtore, di tipo stringa, e un termine, che può essere un numero, una variabile o un'altra struttura. In questo modo la creazione

del termine del template risulta molto più efficiente.
Terminata la costruzione del template è il momento di effettuare l'interrogazione all'interno della teoria dell'agente. Per fare questo viene utilizzata la funzione $solve(term)$ definita all'interno del motore tuProlog e che restituisce un oggetto dal quale è possibile ricavare i risultati dell'interrogazione, come ad esempio:

- controllare se la richiesta è andata a successo;
- recuperare il valore di una singola variabile inserita nel template;
- ottenere la soluzione del template, ovvero dove tutte le variabili utilizzate sono sostituite con il valore del fatto o della regola ricavato;
- verificare se ci fossero altre possibili alternative che corrispondono al template utilizzato;

Per quanto riguarda invece l'esecuzione dell'intenzione sono state definite una serie di regole che, dato l'identificativo, recuperano l'intenzione e ne prelevano la lista di operazioni collegate. Se la lista è vuota allora l'intenzione viene rimossa. Altrimenti si preleva l'operazione in testa che viene eseguita e restituisce una lista con eventuali operazioni da eseguire scaturite dal suo processamento (ed esempio se l'azione è un goal). La nuova lista di operazioni viene quindi aggiunta in testa alla lista delle operazioni già presenti nell'intenzione e poi quest'ultima viene aggiornata. Quanto appena descritto è mostrato nel Codice sorgente 4.3.

È inoltre necessario descrivere in che modo vengono selezionati i piani applicabili e le intenzioni per completare la descrizione degli step che sono effettuati.
Per quanto riguarda la selezione di piani applicabili, in base all'implementazione attuale, viene utilizzata la selezione del motore tuProlog poichè sfrutta l'ordinamento con cui sono definite le regole e i fatti nella teoria per scegliere quello giusto, anche attraverso l'utilizzo del contesto.
Diversamente per quanto riguarda le intenzioni avviene la seguente gestione.

Nella teoria dell'agente sono salvate tutte le intenzioni mentre nell'interprete viene salvata una lista contenente solamente gli identificativi delle rispettive intenzioni presenti nella teoria. La tipologia di selezione è Round-Robin, ogni intenzione avrà lo stesso spazio di esecuzione delle altre. L'intenzione da eseguire viene presa dalla testa dello stack e una volta finita la sua esecuzione viene posizionata in coda: in questo modo si assicura che ogni intenzione possa essere eseguita.

```
execute(I) :-
    intention(I, []),
    !,
    agent <- removeCompletedIntention(I).

execute(I) :-
    retract(intention(I, [ACTION | STACK])),
    execute(I, ACTION, TOP),
    !,
    append(TOP, STACK, NEWSTACK),
    assertz(intention(I, NEWSTACK)).
```

Codice sorgente 4.3: Implementazione regole per invocazione esecuzione di un'intenzione

4.2.3 Gestione azioni

L'esecuzione vera e propria dell'intenzione è gestita da altre regole che sono definite ciascuna per ogni tipologia: *achievement*, *test*, *concurrent*, azione interna e azione esterna; qui di seguito verrà mostrato come sono state implementate ognuna di esse.
In caso l'azione da eseguire fosse un *achievement* allora, una volta verificata la correttezza del contesto (detto anche guardia), il contenuto del corpo di quella regola verrà recuperato e restituito per essere aggiunto in testa allo

stack da cui è pervenuta l'invocazione di quell'azione. Lo stesso comportamento verrà tenuto per azioni di tipo *test*.
Diversamente, se si tratta dell'azione *concurrent*, sia che essa racchiuda *achievement* o *test*, il suo obiettivo è quello di creare un intenzione concorrente a quella dalla quale è pervenuta l'invocazione dell'azione. Per fare questo, per prima cosa è ottenuta la regola verificandone la correttezza del contesto e quindi si recupera la sua lista di operazioni, contenuta nel corpo. A questo punto viene generato un nuovo identificativo univoco per l'intenzione e poi è creato il fatto dell'intenzione così formato: $intention(id, [op_1, \ldots, op_n])$.
Per quanto riguarda le azioni interne ed esterne, quindi che rispettivamente accadono nell'agente o nell'ambiente, il programmatore dell'interprete è in grado di definirne di nuove in base al contesto applicativo in cui si deve calare il linguaggio e l'interprete. Un'esempio di definizione di un'azione interna è quello relativo all'azione per inviare dei messaggi. Per realizzarlo è stata definita la sintassi $iSend(S, M)$ dove S è il mittente e M messaggio. Quando un'azione interna o esterna viene richiamata per l'esecuzione deve essere verificata la sua esistenza prima che possa essere eseguita: per fare questo possono essere definiti dei fatti, come il seguente $is\_internal(iSend(S, M))$, che sono utilizzati come template per verificare che la sintassi sia corretta. Il controllo funziona come una guardia e se ha successo allora l'operazione interna può essere invocata tramite l'apposita funzione implementata all'interno della classe dell'agente, riferita all'oggetto *agent*. Per comodità si potrebbe implementare all'interno dell'agente un'unica funzione, ad esempio $executeInternalAction(action)$ che viene sempre richiamata per l'esecuzione di azioni interne e che, in base al parametro in ingresso, esegue l'azione corretta. In modo analogo possono essere gestire le azioni esterne che possono essere gestite dal nodo, lo spazio dell'agente. Infatti, la scelta di fornire anche l'oggetto del nodo nella teoria dell'agente permette di poter implementare azioni che l'agente possa compiere e che abbiano effetto nell'ambiente.

4.2.4 Spostamento del nodo

Precedentemente nella sezione 4.1 si è trattato dello spazio, ovvero dell'ambiente in cui gli agenti si muovono. Con il mapping scelto per l'implementazione del meta-modello sono presenti due livelli di spazio: uno internamente al nodo e che contiene gli agenti ed un altro che è l'ambiente globale dove si possono muovere i nodi. Il nodo quindi è visto non come un singolo agente ma come un contenitore di agenti che è in grado di muoversi nello spazio sia fisico che simulato. Si può quindi implementare all'interno della classe del nodo la funzionalità per gestire il movimento.
Nell'implementazione che è stata effettuata si sono utilizzate due variabili per la gestione dello spostamento: la direzione (calcolata in radianti) e la velocità. Entrambe le variabili sono memorizzate come proprietà della classe. Inoltre, è stato utilizzato un altro parametro, il momento dell'ultimo aggiornamento della posizione, per poter calcolare l'esatta posizione finale del nodo trascorso un certo arco temporale. In questo modo, avendo possibilità di gestire tutti i parametri (direzione, velocità, tempo) è possibile muovere il nodo in una qualsiasi posizione.
La funzione che calcola la nuova posizione del nodo è descritta qui di seguito. Date la posizione attuale del nodo, la direzione in radianti, la velocità del nodo e il tau (tempo della simulazione) dell'ultimo aggiornamento. Viene costruito un cerchio che ha come centro la posizione attuale del nodo e il cui raggio ha distanza uguale a $V * (T_1 - T_0)$, dove V è la velocità del nodo, T_1 è il tempo attuale della simulazione e T_0 è il tempo dell'ultimo aggiornamento della posizione. Sulla circonferenza appena creata viene individuata la prossima posizione del nodo calcolando il punto che corrisponde alla direzione attualmente memorizzata nel nodo.

```
1 void changeNodePosition(Time t1) {
2   Position currPos = this.getNodePosition();
3
4   double radius = (t1.toDouble() -
     this.t0.toDouble()) * this.speed;
```

```
5
6   double coordX = currPos.getCoordinate(0) + radius
     * Math.cos(this.radAngle);
7
8   double coordY = currPos.getCoordinate(1) + radius
     * Math.sin(this.radAngle);
9
10  this.environment.moveNodeToPosition(this,
     this.environment.makePosition(coordX, coordY));
11
12  this.t0 = t1;
13 }
```

Codice sorgente 4.4: Implementazione aggiornamento posizione nodo

Per rispettare il fatto che il movimento sia una funzione basilare dell'implementazione ad agenti si è pensato a come realizzarlo. Richiamare la funzione mostrata nel Codice sorgente 4.4 dentro ogni agente non è il modo migliore poichè potrebbe essere richiamata più di una volta ed essere quindi aggiornata troppo frequentemente.
L'idea che è stata poi realizzata è quella di inserire all'interno di ogni nodo un agente che abbia come unico compito quello di innescare l'aggiornamento della posizione del nodo e di comunicare la variazione a tutti gli altri agenti all'interno del nodo. Per fare ciò, il modo migliore è stato quello di inserire durante la creazione del nodo, tramite il codice dell'interprete, un agente il quale, come detto precedenemente, avrà il compito di innescare l'aggiornamento della posizione del nodo e una volta terminato di far partire l'aggiornamento interno di ogni agente riguardo la sua posizione (che è quella del nodo) e la distanza rispetto a tutti gli altri agenti (sia residenti nello stesso nodo che in altri).
Questo agente che gestisce il movimento ha una distribuzione temporale uguale a 1, ovvero il suo ciclo di ragionamento viene eseguito 1 volta ogni

'clock' dello scheduler. In questo modo, definendo opportunamente la distribuzione temporale degli altri agenti della simulaizone si può ottenere il risultato come se l'aggiornamento della posizione provenisse da un componente esterno, ad esempio un apparato GPS, che periodicamente aggiorna la posizione.

4.2.5 Implementazione Spatial Tuples

La realizzazione lato Alchemist del modello Spatial Tuples è stata fatta appongiandosi al lavoro già svolto per la realizzazione degli agenti. Partendo dalla classe astratta definita per gli agenti è stato quindi deciso di estenderla per creare una nuova classe, anch'essa astratta, con la quale sarà possibile creare implementazioni specifiche per gli spazi di tuple. Per implementare correttamente il modello Spatial Tuples e gestire opportunamente le sue primitive è necessario però completare due parti: la prima è quella appena descritta riguardo l'implementazione di una classe astratta, mentre la seconda riguarda la definizione di funzioni nella classe dell'agente per poter gesistire le invocazioni delle primitive del linguaggio.

Sfruttando l'ereditarietà della classe astratta degli agenti, anche lo spazio di tuple è implementato come un agente anche se con le sue precise caratteristiche. Per implementare la classe sono state definite come proprietà due liste: una per i messaggi in entrata e una per quelli in attesa. Il resto delle proprietà utilizzate come il motore tuProlog, il nome dell'agente, la libreria sono ereditate dal padre e lo stesso vale anche per il costruttore.
Ogni richiesta è composta da un template, un identificativo dell'azione e l'oggetto dell'agente al quale poi comunicare la risposta. Per la gestione delle due code aggiunte sono state implementate due funzioni, una per ogni coda, ed in entrambe, in base all'azione della richiesta ricevuta(in, rd, out), viene invocato il metodo per effettuare quell'operazione. Le operazioni vanno a modificare la teoria dello spazio di tuple scrivendo, leggendo o rimuovendo fatti dalla teoria tramite il motore tuProlog.

L'implementazione dell'operazione *in* viene sempre eseguita immediatamente e al termine richiama la funzione che controlla se nella lista delle richieste pendenti vi siano template che corrispondano alla tupla appena inserita.

Le operazioni *rd* e *out* sono gestite in modo simile, cambia solo la natura del termine contenente il template che viene passato al motore tuProlog: nel primo caso contiene solo il template mentre nel secondo anche la sintassi per rimuovere la tupla. In entrambi i casi, se il motore restituisce una tupla, questa viene presa e inviata tramite messaggio diretto all'agente.

In questo modo si è implementato lo spazio di tuple in modo astratto, lasciando alla classe specifica il compito di implementare le funzioni mancanti, come ad esempio quella del ciclo di ragionamento. Ora è necessario parlare della parte da definire nella classe dell'agente.

Nella classe dell'agente rimane da realizzare la parte per inviare una richiesta allo spazio di tuple e per ricevere in risposta le tuple richieste: per fare questo si sono implementati due funzioni. La prima funzione è quella che recupera i fatti a seguito dell'invocazione di una delle regole definite nel Codice sorgente 4.2 e successivamente li utilizza per creare la richiesta. Nell'implementazione realizzata si sono voluti tenere due approcci diversi per la scrittura e per la lettura di tuple: nel primo caso è stato scelto di farlo nella tupla più vicina, mentre nel secondo caso sono recuperati tutti gli spazi di tuple nell'intorno dell'agente e su questi viene richiesto il template. La possibilità di recuperare uno o più spazi di tuple è fornita dal nodo che è stato implementato in modo tale da poter recuperare la lista degli agenti presenti in ogni nodo del suo vicinato che poi può essere opportunamente filtrata in base alle necessità.
L'altra funzione da implementare, è quella per ricevere il messaggio di ritorno dallo spazio di tuple quando viene trovata la tupla che corrisponde al template passato. Come parametro in ingresso richiede un termine valido così da limitare il suo operato al recupero della regola $onResponseMessage(T)$, dove T è il parametro, idonea al contesto e quindi creare una nuova intenzione

pronta per essere eseguita.

Bibliografia

[Bib. 1] AgentSpeak(L): BDI Agents speak out in a logical computable language, Anand S. Rao, 1996

[Bib. 2] Spatial Tuples: Augmenting reality with tuples, Ricci et al., 2017

[Bib. 3] Programming multi-agent systems in AgentSpeak using Jason, Rafael H. Bordini, Jomi Fred Hubner, Michael Wooldridge, 2007

57

Sitografia

[Sit. 1] https://jade.tilab.com/

[Sit. 2] https://pypi.org/project/spade/

[Sit. 3] http://jason.sourceforge.net/wp/

[Sit. 4] http://www.sarl.io/

[Sit. 5] https://www.activecomponents.org/#/project/news

[Sit. 6] http://astralanguage.com/wordpress/

[Sit. 7] http://apice.unibo.it/xwiki/bin/view/Tuprolog/

[Sit. 8] http://alchemistsimulator.github.io