# Abstract

TODO Abstract

# Indice

# Elenco delle figure

# Codici sorgenti

# Capitolo 1

# Introduzione

TODO DESCRIZIONE INTRODUZIONE

## 1.1 Contesto

TODO DESCRIZIONE CONTESTO

### 1.1.1 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.

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. Il meccaniscmo di trasporto, in particolare, è come un camaleonte perchè si adatta ad ogni situazione scegliendo trasparentemente il miglior protocollo disponibile.

### 1.1.2 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.

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). 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.

### 1.1.3 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. 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.

### 1.1.4 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.

### 1.1.5 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 serviziw e consumatori. 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.

### 1.1.6 ASTRA

ASTRA è un linguaggio di programmazione ad agenti per creare sistemi intelligenti distrubuiti/concorrenti costruiti su Java. 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.

## 1.2 Obiettivo del lavoro

TODO

### 1.2.1 Benefici dell'approccio scelto

TODO

# Capitolo 2

# Stato dell'arte

In questo capitolo sono mostrati 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 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: il suo insieme è chiamato 'belief base' o 'belief set'.
I **desires** rappresentano tutti i possibili piani che un agente potrebbe eseguire. 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.

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.

### 2.1.1 Definizione

**AgentSpeak** è un linguaggio di programmazione basato su un linguaggio del primo ordine con eventi e azioni. 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. L'adozione di programmi per soddisfare tali stimoli è detta intenzioni.

### 2.1.2 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).

**Definizione 1.** Se $b$ è un simbolo di predicato e $t_1,...,t_n$ sono termini, allora $b(t_1,...,t_n)$ 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*.

**Definizione 2.** Se $g$ è un simbolo di predicato e $t_1,...,t_n$ sono termini, allora $!g(t_1,...,t_n)$ e $?g(t_1,...,t_n)$ sono *goals*.

**Definizione 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*.

**Definizione 4.** Se $a$ è un simbolo di azione e $t_1,...,t_n$ sono termini del primo ordine, allora $a(t_1,...,t_n)$ è un'azione.

**Definizione 5.** Se $e$ è un *eventi di attivazione*, $b_1,...,b_m$ sono letterali di belief e $h_1;...;h_n$ sono goals o azioni, allora $e : b_1 \wedge ... \wedge b_m \leftarrow h_1;...;h_n$ è un piano.

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).

**Definizione 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.

**Definizione 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 $[p_1‡...‡p_z]$, dove $p_1$ è il fondo dello stack e $p_z$ la testa. Gli elementi sono delimitati da ‡. Per convenienza [+!true : true ← true] è detta *true intention* e viene definita con T.

**Definizione 8.** Il set E è composto da eventi, oguno delle quali è una tupla $\langle e, i\rangle$, dove $e$ è l'evento e $i$ un'intenzione. Se l'intenzione è di tipo *true intention* allora $e$ sarà chiamato *evento esterno*, altrimenti è un *evento interno*.

**Definizione 9.** Sia $S_E(\mathrm{E}) = \epsilon = \langle d, i \rangle$ e sia $p = e : b_1 \wedge ... \wedge b_m \leftarrow h_1;...;h_n$, il piano $p$ è rilevante per l'evento $e$ se e solo se esiste un unificatore $\sigma$ tale per cui $\mathrm{d}\sigma = \mathrm{e}\sigma$. $\sigma$ è detto *unificatore rilevante* per $\epsilon$.

**Definizione 10.** Un piano $p$ è definito da $e : b_1 \wedge ... \wedge b_m \leftarrow h_1;...;h_n$ è un *piano applicabile* rispetto ad un evento $\epsilon$ se e solo se esite un identificatore rilevante $\sigma$ per $\epsilon$ e esite una sostituzione $\theta$ tale che $\forall(b_1 \wedge ... \wedge b_m)\sigma\theta$ è una conseguenza logica di B. La composizione $\sigma\theta$ è riferita all'*unificatore applicabile* per l'evento $\epsilon$ e $\theta$ è riferita alla sostituzione della corretta risposta.

**Definizione 11.** Sia $\mathrm{S_O}(\mathrm{O}_\epsilon) = p$, dove $\mathrm{O}_\epsilon$ è il set dei piani applicabili per l'evento $\epsilon = \langle d, i \rangle$ e $p$ è $e : b_1 \wedge ... \wedge b_m \leftarrow h_1;...;h_n$. Il piano $p$ è destinato all'evento $\epsilon$, dove $i$ è la *true intention* se e solo se esiste un *unificatore applicabile* $\sigma$ per cui $[+!\mathrm{true} : \mathrm{true} \leftarrow \mathrm{true} \ddagger (e : b_1 \wedge ... \wedge b_m \leftarrow h_1;...;h_n)\sigma\ ] \in \mathrm{I}$.

**Definizione 12.** Sia $\mathrm{S_O}(\mathrm{O}_\epsilon) = p$, dove $\mathrm{O}_\epsilon$ è il set dei piani applicabili per l'evento $\epsilon = \langle d, [p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow !g(t);\mathrm{h_2};...;\mathrm{h_n}] \rangle$, e $p$ è $+!g(s) : \mathrm{b_1} \wedge ... \wedge \mathrm{b_m} \leftarrow \mathrm{k_1};...;\mathrm{k_j}$. Il piano $p$ è destinato all'evento $\epsilon$ se e solo se esiste un *unificatore applicabile* $\sigma$ tale che $[p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow !g(t);\mathrm{h_2};...;\mathrm{h_n} \ddagger (+!g(s) : \mathrm{b_1} \wedge ... \wedge \mathrm{b_m})\ \sigma \leftarrow (\mathrm{k_1};...;\mathrm{k_j})\ \sigma;\ (\mathrm{h_2};...;\mathrm{h_n})\sigma] \in \mathrm{I}$.

**Definizione 13.** Sia $\mathrm{S_I}(\mathrm{I}) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow !g(t);\mathrm{h_2};...;\mathrm{h_n}]$. L'intenzione $i$ si dice che è eseguita se e solo se $\langle +!g(t), i \rangle \in \mathrm{E}$.

**Definizione 14.** Sia $\mathrm{S_I}(\mathrm{I}) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow ?g(t);\mathrm{h_2};...;\mathrm{h_n}]$. L'intenzione $i$ si dice che è eseguita se e solo se esiste una sostituzione $\theta$ tale che $\forall \mathrm{g(t)}\theta$ è una conseguenza logica di B e $i$ è rimpiazzato da $[p_1 \ddagger ... \ddagger (f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y})\theta \leftarrow \mathrm{h_2}\theta;...;\mathrm{h_n}\theta]$.

**Definizione 15.** Sia $\mathrm{S_I}(\mathrm{I}) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow a(t);\mathrm{h_2};...;\mathrm{h_n}]$. L'intenzione $i$ si dice che è eseguita se e solo se $a(t) \in \mathrm{A}$, e $i$ è rimpiazzato da $[p_1 \ddagger ... \ddagger f : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow \mathrm{h_2};...;\mathrm{h_n}]$.

**Definizione 16.** Sia $\mathrm{S_I}(\mathrm{I}) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger p_{z\text{-}1} \ddagger g(t) : \mathrm{c_1} \wedge ... \wedge \mathrm{c_y} \leftarrow true]$, dove $p_{z\text{-}1}$ è $e : \mathrm{b_1} \wedge ... \wedge \mathrm{b_x} \leftarrow !g(s);\ \mathrm{h_2};...;\mathrm{h_n}$. L'intenzione $i$ si dice

che è eseguita se e solo se esiste una sostituzione $\theta$ tale che g(t)$\theta$ = g(s)$\theta$ e $i$ è rimpiazzato da [$p_1$ ‡ ... ‡ $p_{z-1}$ ‡ (e : $b_1 \wedge ... \wedge b_x)\theta \leftarrow (h_2;...;h_n)\theta$].

### 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 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. 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: 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'agente. Questa funzione può essere implementata ad esempio per far ricevere solo i messaggi un certo agente.

**d. Selezione evento**

Gli eventi rappresentano la percezione del cambiamento nell'ambiente o dello stato interno dell'agente, 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, 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. 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. 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: 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. 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. 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: 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.

## 2.2 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.

## 2.3 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.

### 2.3.1 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.

## 2.4 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).

### 2.4.1 Meta-modello

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

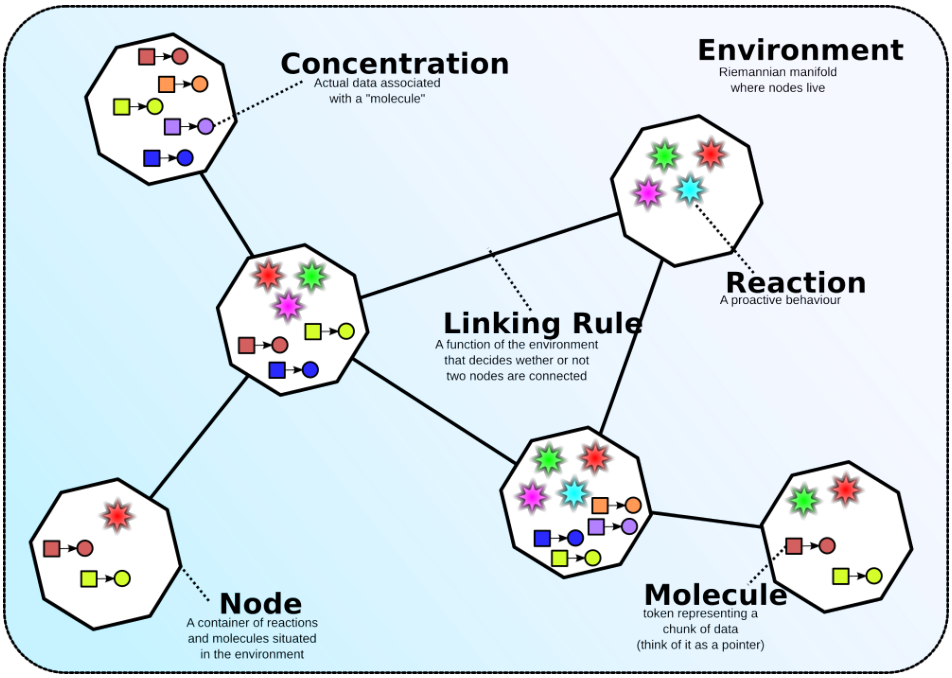


Figura 2.1: Illustrazione meta-modello di Alchemist

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.

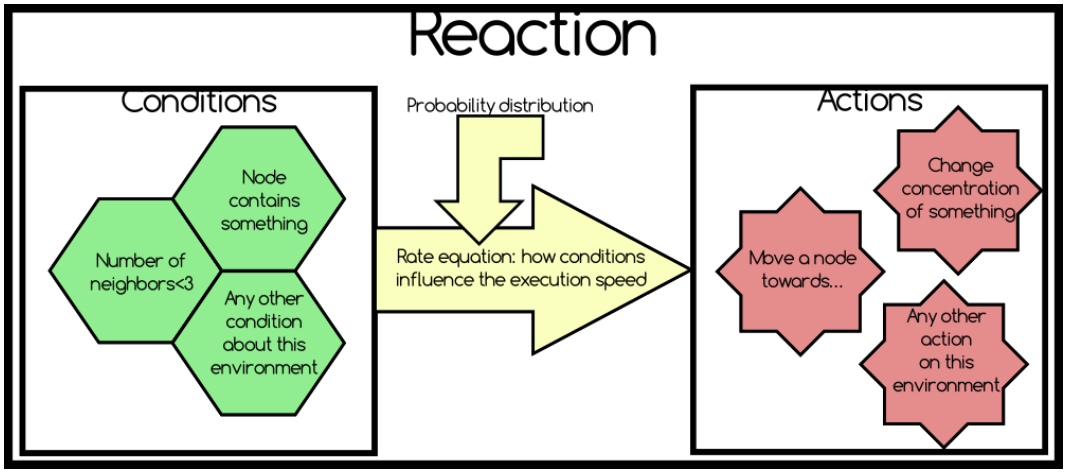


Figura 2.2: Illustrazione modello reazione di Alchemist

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.

### 2.4.2 Esempi (?)

## 2.5 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.

La sintassi del processo di calcolo in LINDA è definito tramite la seguente grammatica:

$P,\ Q ::= t \mid \text{outeval}(P).Q \mid \text{rd}(t).P \mid \text{in}(t).P \mid P + Q \mid \text{rec}X.P \mid X$

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.

### 2.5.1 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. Quindi è stato definito come una libreria senza nessun riferimento all'ambiente. Nei capitoli successivi verrà mostrato come, partendo dalla libreria, 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 definire questo linguaggio per permettere di poterlo utilizzare facilemente colmando la distanza tra l'ambiente e la libreria tuProlog.

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

**Definizione 1.** Se $b$ è un simbolo di predicato e $t_1,...,t_n$ sono termini, allora *belief*$(b(t_1,...,t_n))$ è un atomo di belief. Un atomo di belief oppure la

sua negazione sono riferiti al letterale del belief. Un atomo di belief base sarà chiamato *belief base*.

**Definizione 2.** Se $g$ è un simbolo di predicato e $t_1,...,t_n$ sono termini, allora achievement($g(t_1,...,t_n)$) e test($g(t_1,...,t_n)$) sono *goals*.

**Definizione 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*.

**Definizione 4.** Se $a$ è un simbolo di azione e $t_1,...,t_n$ sono termini del primo ordine, allora $a(t_1,...,t_n)$ è un'azione.

**Definizione 5.TODO RIVEDERE DEFINIZIONE PIANO (sotto meglio)** Se $e$ è un *eventi di attivazione* e $h_1,...,h_n$ sono goals o azioni, allora $e$ :- $h_1,...,h_n$. è un piano. L'espressione a sinistra di ':-' è la testa, quella sulla destra il corpo: se quest'ultimo è vuoto viene definito con l'espressione *true*. Se si utilizzano le guardie, ad esempio $b_1,...,b_m$, allora '$\leftarrow$'($e$, [$b_1,...,b_m$], [$h_1,...,h_n$]) è un piano.

**Definizione 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$(i, [p_1,...,p_n])$, dove $i$ è l'identificativo univoco dell'intenzione e [$p_1$,...,$p_n$] è lo stack formata da azioni, belief o goal: $p_1$ è la coda e $p_n$ è la testa.

**Definizione 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.

**Definizione 8.** Dato un evento $\epsilon$ ed un piano $p$ = '$\leftarrow$'($e$, [$b_1,...,b_m$], [$h_1,...,h_n$]), allora $p$ è rilevante per l'evento $\epsilon$ se e solo se esite un unificatore $\sigma$ tale per cui d$\sigma$ = e$\sigma$. $\sigma$ è detto *unificatore rilevante* per $\epsilon$.

**Definizione 9.** Un piano $p$ è definito da '$\leftarrow$'($e$, [$b_1,...,b_m$], [$h_1,...,h_n$]) è un *piano applicabile* rispetto ad un evento $\epsilon$ se e solo se esite un identificatore

rilevante $\sigma$ per $\epsilon$ e esite una sostituzione $\theta$ tale che $\forall(b_1,...,b_m)\sigma\theta$ è una conseguenza logica di B.

**TODO**

**selezione intenzione (13,14,15,16)**

**Definizione 13.** Sia $S_I(I) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : c_1 \wedge ... \wedge c_y \leftarrow !g(t); h_2;...;h_n]$. L'intenzione $i$ si dice che è eseguita se e solo se $\langle +!g(t), i \rangle \in E$.

**Definizione 14.** Sia $S_I(I) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : c_1 \wedge ... \wedge c_y \leftarrow ?g(t); h_2;...;h_n]$. L'intenzione $i$ si dice che è eseguita se e solo se esiste una sostituzione $\theta$ tale che $\forall g(t)\theta$ è una conseguenza logica di B e $i$ è rimpiazzato da $[p_1 \ddagger ... \ddagger (f : c_1 \wedge ... \wedge c_y)\theta \leftarrow h_2\theta;...;h_n\theta]$.

**Definizione 15.** Sia $S_I(I) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger f : c_1 \wedge ... \wedge c_y \leftarrow a(t); h_2;...;h_n]$. L'intenzione $i$ si dice che è eseguita se e solo se $a(t) \in A$, e $i$ è rimpiazzato da $[p_1 \ddagger ... \ddagger f : c_1 \wedge ... \wedge c_y \leftarrow h_2;...;h_n]$.

**Definizione 16.** Sia $S_I(I) = i$, dove $i$ è $[p_1 \ddagger ... \ddagger p_{z-1} \ddagger g(t) : c_1 \wedge ... \wedge c_y \leftarrow true]$, dove $p_{z-1}$ è $e : b_1 \wedge ... \wedge b_x \leftarrow !g(s); h_2;...;h_n$. L'intenzione $i$ si dice che è eseguita se e solo se esiste una sostituzione $\theta$ tale che $g(t)\theta = g(s)\theta$ e $i$ è rimpiazzato da $[p_1 \ddagger ... \ddagger p_{z-1} \ddagger (e : b_1 \wedge ... \wedge b_x)\theta \leftarrow (h_2;...;h_n)\theta]$.

## 3.2 Interfacce 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 le keyword del linguaggio e, successivamente, vengono analizzate ed esposte:

- init
- self(A)
- agent

- node
- addBelief(B)
- removeBelief(B)
- onAddBelief(B)
- onRemoveBelief(B)
- onReceivedMessage(S, M)
- achievement
- test
- concurrent
- belief(position(X,Y))
- belief(distance(A, ND, OD)) — o — belief(distance(A, ND))

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 belief '**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 keyword '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 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**', 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 stati definiti 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.

### Estensione Spatial Tuples

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

- writeTuple(T)
- readTuple(TT)
- takeTuple(TT)
- onResponseMessage(M)

Le keyword 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 sorgente 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

TODO intro

## 4.1 Mapping modello agenti su meta-modello

TODO

## 4.2 Descrizione interprete linguaggio

TODO

# Bibliografia

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

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

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

**41**

# Sitografia


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

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

[3] http://www.sarl.io/

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

[5] http://astralanguage.com/wordpress/

[6] http://jason.sourceforge.net/wp/

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

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