

# Abstract

TODO Abstract



# Indice

<b>Abstract</b>	<b>1</b>
<b>1 Introduzione</b>	<b>7</b>
1.1 Contesto . . . . .	7
1.1.1 JADE . . . . .	7
1.1.2 SPADE . . . . .	8
1.1.3 Jason . . . . .	9
1.1.4 SARL . . . . .	9
1.1.5 JADEX . . . . .	10
1.1.6 ASTRA . . . . .	10
1.2 Obiettivo del lavoro . . . . .	10
1.2.1 Benefici dell’approccio scelto . . . . .	10
<b>2 Stato dell’arte</b>	<b>11</b>
2.1 Agenti BDI con AgentSpeak . . . . .	11
2.1.1 Definizione . . . . .	12
2.1.2 Descrizione formale . . . . .	12
2.2 Caratteristiche Jason . . . . .	18
2.3 tuProlog . . . . .	19
2.3.1 Caratteristiche tuProlog . . . . .	19
2.4 Alchemist . . . . .	20
2.4.1 Meta-modello . . . . .	20
2.4.2 Esempi (?) . . . . .	23
2.5 LINDA . . . . .	23

2.5.1 Spatial Tuples . . . . .	24
<b>Bibliografia</b>	<b>27</b>
<b>Sitografia</b>	<b>29</b>

## Elenco delle figure

2.1	Illustrazione meta-modello di Alchemist . . . . .	21
2.2	Illustrazione modello reazione di Alchemist . . . . .	22



# Capitolo 1

## Introduzione

TODO DESCRIZIONE INTRODUZIONE

### 1.1 Contesto

TODO DESCRIZIONE CONTESTO

#### 1.1.1 JADE

JADE (Java Agent DEvelopment 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 modello 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 meccanismo di trasporto, in particolare, è come un camaleonte perché 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 buona 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 dà 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 fornisce un meccanismo 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 operativa 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 meccanismo 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 servizi 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 distribuiti/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'environment 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 operativa.

Le specifiche 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), '←' (per implicazione).

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

**Definizione 2.** Se  $g$  è un simbolo di predicato e  $t_1, \dots, t_n$  sono termini, allora  $!g(t_1, \dots, t_n)$  e  $?g(t_1, \dots, t_n)$  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  $t_1, \dots, t_n$  sono termini del primo ordine, allora  $a(t_1, \dots, t_n)$  è un'azione.

**Definizione 5.** Se  $e$  è un evento scatenante,  $b_1, \dots, b_n$  sono letterali di belief e  $h_1; \dots; h_n$  sono goals o azioni, allora  $e : b_1 \wedge \dots \wedge b_m \leftarrow h_1; \dots; h_n$  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 da 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  $[p_1 \ddagger \dots \ddagger p_z]$ , 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.

**Definizione 8.** Il set E è composto da eventi, ognuno 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 dove  $e$  sarà chiamato *evento esterno*, altrimenti è un *evento interno*.

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

**Definizione 10.** Un piano  $p$  è definito da  $e : b_1 \wedge \dots \wedge b_m \leftarrow h_1; \dots; h_n$  è un *piano applicabile* rispetto ad un evento  $e$  se e solo se esiste un identificatore rilevante  $\sigma$  per  $\epsilon$  e esiste una sostituzione  $\theta$  tale che  $\forall (b_1 \wedge \dots \wedge b_m)\sigma\theta$  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  $S_O(O_\epsilon) = p$ , dove  $O_\epsilon$  è il set dei piani applicabili per l'evento  $\epsilon = \langle d, i \rangle$  e  $p$  è  $e : b_1 \wedge \dots \wedge b_m \leftarrow h_1; \dots; h_n$ . 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  $[+!true: true \nmid (e : b_1 \wedge \dots \wedge b_m \leftarrow h_1; \dots; h_n)\sigma] \in I$ .

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

**Definizione 13.** Sia  $S_I(I) = i$ , dove  $i$  è  $[p_1 \nmid \dots \nmid f : c_1 \wedge \dots \wedge c_y \leftarrow !g(t); h_2; \dots; h_n]$ . 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  $[p_1 \nmid \dots \nmid f : c_1 \wedge \dots \wedge c_y \leftarrow ?g(t); h_2; \dots; h_n]$ . 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  $[p_1 \nmid \dots \nmid (f : c_1 \wedge \dots \wedge c_y)\theta \leftarrow h_2\theta; \dots; h_n\theta]$ .

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

**Definizione 16.** Sia  $S_I(I) = i$ , dove  $i$  è  $[p_1 \nmid \dots \nmid p_{z-1} \nmid g(t) : c_1 \wedge \dots \wedge c_y \leftarrow true]$ , dove  $p_{z-1}$  è  $e : b_1 \wedge \dots \wedge b_x \leftarrow !g(s); h_2; \dots; h_n$ . 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  $[p_1 \ddagger \dots \ddagger p_{z-1} \ddagger (e : b_1 \wedge \dots \wedge b_x) \text{ textit} \theta \leftarrow (h_2\theta; \dots; h_n)\theta]$ .

### 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 effettuato 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 priorità 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 applicabile**

Ogni piano ha un contesto che definisce con quali informazioni dell'agente può essere usato. Per piano applicabile si intendono 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 necessario 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.

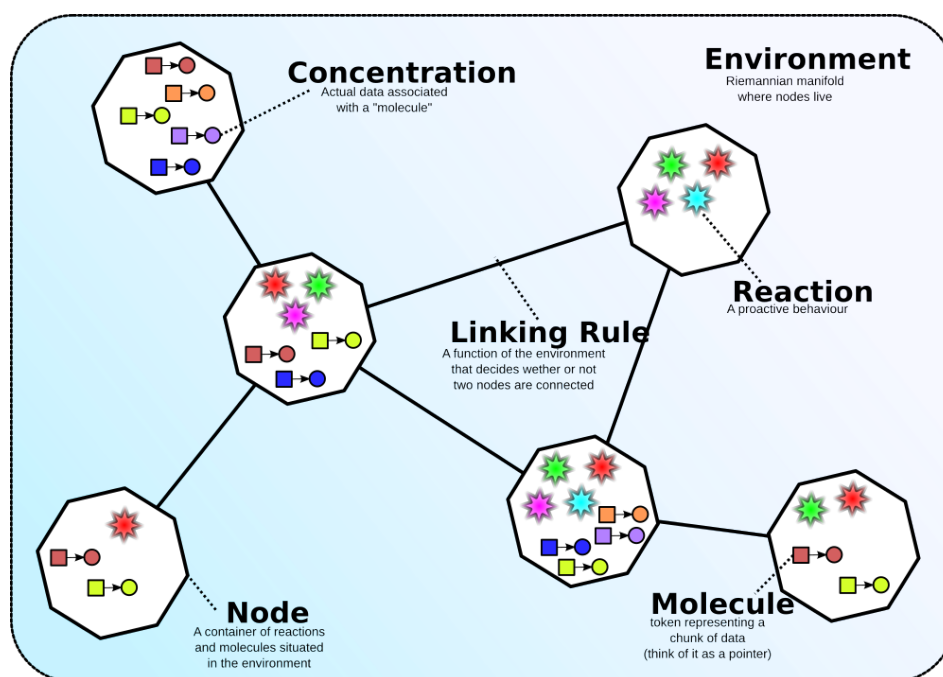


Figura 2.1: Illustrazione meta-modello di Alchemist

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'environment 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.

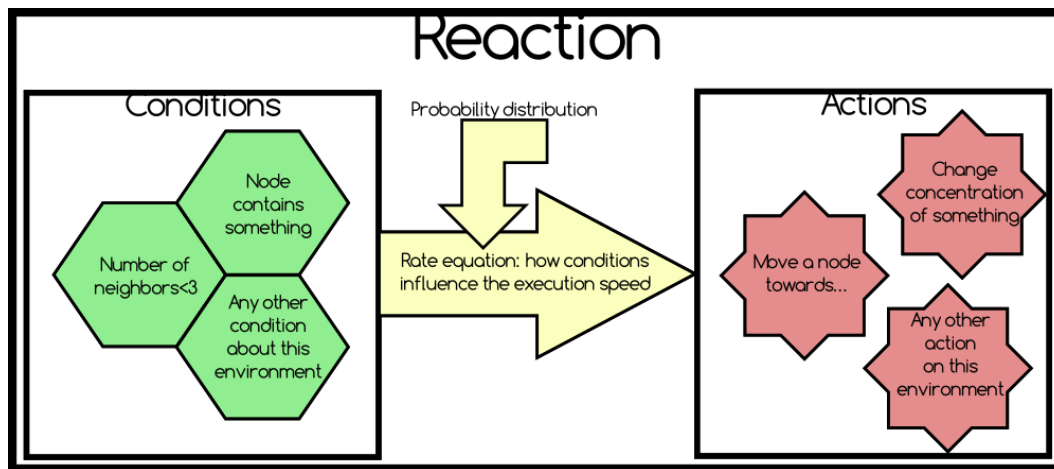


Figura 2.2: Illustrazione modello reazione di Alchemist

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:

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

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 corrispondono 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.



# 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



# 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>