Introduzione

Questo lavoro ha l'obiettivo di implementare sul simulatore Alchemist il modello ad agenti.

Alchemist è un meta-simulatore estendibile, ispirato alla chimica stocastica e adatto al calcolo pervasivo e ai sistemi distribuiti. Fornisce un meta-modello flessibile, sul quale gli sviluppatori legano le proprie astrazioni, realizzando un'incarnazione.

Il modello ad agenti a cui si fa riferimento è quello BDI (Beliefs, Desires, Intentions) che è ispirato al modello del comportamento umano.

Indice

Elenco delle figure

Codici sorgenti

Capitolo 1

Alchemist

Alchemist fornisce un ambiente di simulazione sul quale è possibile sviluppare nuove incarnazioni, ovvero nuove definizioni di modelli sviluppati su di esso.

1.1 Il meta-modello

Il meta-modello di Alchemist può essere compreso con la figura 1.1.

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 posizinato 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***.

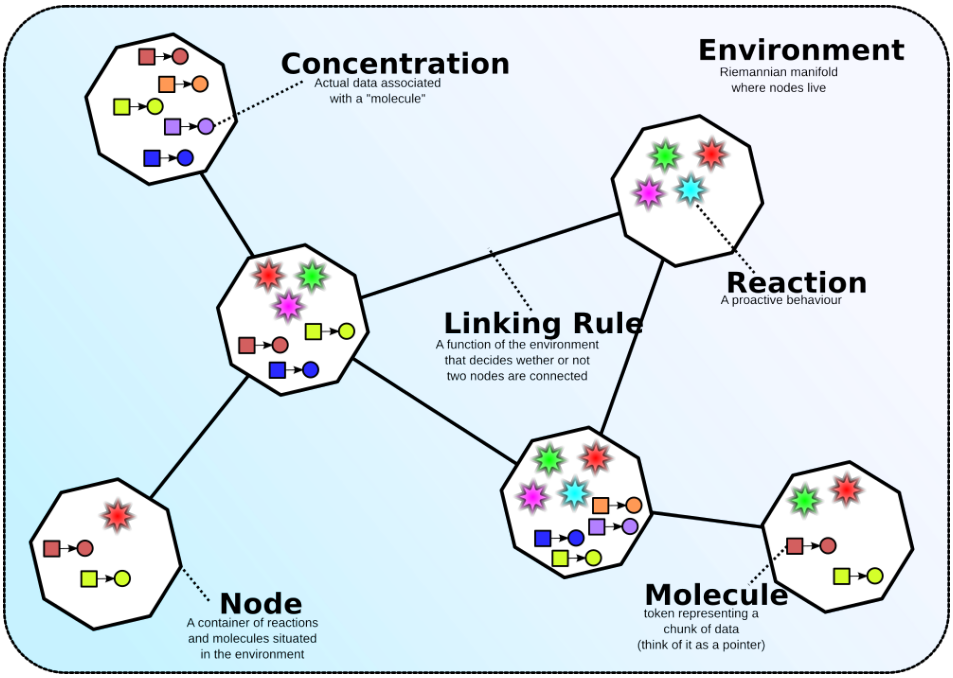

Figura 1.1: Illustrazione meta-modello di Alchemist

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.
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 1.2.

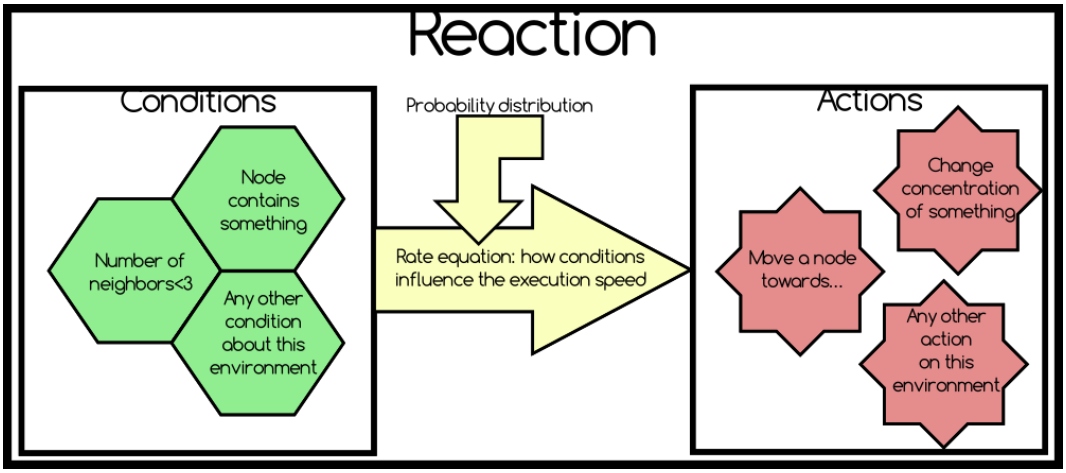

Figura 1.2: Illustrazione modello reazione di Alchemist

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 che si verificano successivamente ed indipendentemente in un dato intervallo di tempo.

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.

1.2 Scrivere una simulazione

Il linguaggio da utilizzare per scrivere le simulazioni in Alchemist è YAML e quello che il parser del simulatore si aspetta in input è una mappa YAML.

Nei prossimi paragrafi verrà mostrato quali sezioni si possono inserire e come utilizzarle per creare la simulazione che si vuole realizzare.

La sezione **incarnation** è obbligatoria. Il parser YAML si aspetta una stringa che rappresenta il nome dell'incarnazione da utilizzare per la simulazione.

```
1  incarnation: agent
```

Codice sorgente 1.1: Incarnazione

Nel resto della sezione, il valore associato alla chiave 'type' fa riferimento al nome di una classe. Se il nome passato non è completo, ovvero non è comprensivo del percorso fino alla classe, Alchemist provvederà a cercare la classe tra i packages.

Per dichiarare variabili che poi potranno essere richiamate all'interno del file di configurazione della simulazione si può procedere in questo modo.

```
2  variables:
3    myVar: &myVar
4      par1: 0
5      par2: "string"
6    mySecondVar: &myVar2
7      par: "value"
```

Codice sorgente 1.2: Variabili simulazione

Utilizzando la keyword **environment** si può scegliere quale definizione di ambiente utilizzare per la simulazione.

```
8  environment:
9    type: OSMEnvironment
10   parameters: [/maps/foo.pbf]
```

Codice sorgente 1.3: Environment

Questo parametro è opzionale e di default è uno spazio continuo bidimensionale: ometterlo equivale a scrivere la seguente configurazione.

```
environment:
  type: Continuous2DEnvironment
```

Codice sorgente 1.4: Default environment

La keyword **positions** consente di specificare il tipo delle coordinate della simulazione. La psizione riflette lo spazio fisico: per esempio non si potrà utilizzare la distanza *Continuous2DEuclidean* se si considera la mappa di una città visto che dati due punti A e B, nel mondo reale la distanza AB è diversa da quella BA.

```
positions:
  type: LatLongPosition
```

Codice sorgente 1.5: Default environment

I collegamenti tra i nodi che verranno utilizzati nella simulazione sono specificati nella sezione **network-model**. Un esempio per la costruzione di collegamenti è il seguente.

```
network-model:
  type: EuclideanDistance
  parameters: [10]
```

Codice sorgente 1.6: Funzione linking-rule

Anche questo è un parametro opzionale e di default non ci sono collegamenti, ovvero i nodi nell'environemnt non sono collegati, ed è descritto con il seguente formalismo.

```
network-model:
  type: NoLinks
```

Codice sorgente 1.7: Default linking-rule

Il posizionamento dei nodi viene gestito dalla sezione **displacements**. Questa sezione può contenere uno o più definizioni di disposizioni per i nodi.

Il parametro 'in' definisce la geometria all'interno del quale verranno disposti i nodi, utilizzando ad esempio punti o figure come cerchi o rettangoli, mentre il parametro 'programs' definisce le reazioni da associare ad ogni nodo di quella certa disposizione.
Esempi di classi utilizzabili nel parametro 'in' sono Point e Circle. La classe Circle necessita di quattro parametri, da passare nel seguente ordine: il numero di nodi da disporre, la coordinata x del centro, la coordinata y del centro, il raggio del cerchio. Per la classe Point è sufficiente fornire in ordine la coordinata x e la coordinata y.

Il parametro 'programs' rappresenta le reazioni da associare ai nodi ed accetta una lista di reazioni, le quali a loro volta sono formate da una lista di parametri. Un'esempio di definizione di una reazione è utilizzando 'time-distribution' (valore utilizzato per settare la freqeunza) e 'program' (parametro che viene passato alla creazione della reazione e che può essere utilizzato per istanziare condizioni e azioni). Un'esempio di displacements è il seguente.

```
20  displacements:
21    - in: {type: Circle, parameters: [5,0,0,2]}
22      programs:
23      -
24        - time-distribution: 1
25          program: "reactionParam"
26        - time-distribution: 2
27          program: "doSomethingParam"
28    - in: {type: Point, parameters: [1,1]}
29      programs:
30      -
31        - time-distribution: 1
32          program: "pointReactionParam"
```

Codice sorgente 1.8: Disposizione nodi e reazioni associate

Capitolo 2

Agenti

Un'agente è un entità che agisce in modo autonomo e continuo in uno spazio condiviso con altri agenti. Le caratteristiche principali di un agente sono: autonomia, proattività e reattività. Gli agenti sono formati da un nome, che è una caratteristica statica, e da componenti dinamici come lo stato.

2.1 Agenti BDI

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 informazioni dello stato dell'agente, ovvero ciò che l'agente sa del mondo (di se stesso e degli altri agenti), e possono comprendere regole di inferenza per permettere l'aggiunta di nuovi beliefs. L'insieme dei belief di un agente è detto 'belief base' o 'belief set' e si può modificare nel tempo.

I ***Desires*** sono tutti i possibili piani che l'agente potrebbe eseguire. Rappresentano gli obiettivi o le situazioni che l'agente vorrebbe realizzare o portare a termine. I **goals** sono desires che l'agente persegue attivamente: per

questo motivo, in generale, i piani desiderabili possono non essere coerenti tra loro mentre i goals è bene che lo siano.

Le ***Intentions*** sono piani a cui l'agente ha deciso di lavorare o a cui sta già lavorando. I piani sono sequenze di azioni che un agente può eseguire per raggiungere una intention. I piani possono contenerne altri al loro interno.

Gli ***Eventi*** innescano le attività reattive degli agenti il cui risultato può essere l'aggiornamento dei beliefs, la chiamata ad altri piani o la modifica di goals.

2.2 Ciclo di ragionamento

Il ciclo di ragionamento, descritto in figura 2.1, è il modo in cui l'agente prende le sue decisioni e mette in pratica le azioni.

In particolare, questo ciclo di ragionamento descrive quello per gli agenti implementati in Jason.

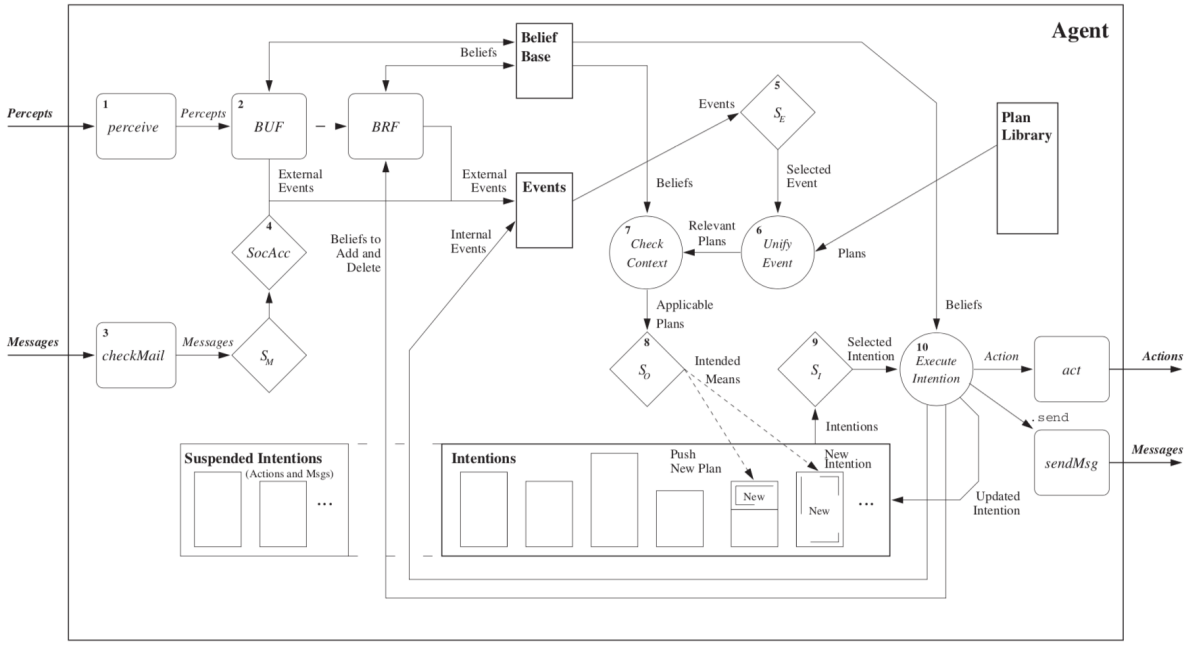

Figura 2.1: Ciclo di ragionamento di un agente

I rettangoli rappresentano lo stato dell'agente. I box arrotondati, i rombi e i cerchi rappresentano le funzioni usate nel ciclo di ragionamento: i primi due identificano funzioni che possono essere personalizzate dal programmatore, mentre i cerchi sono le parti fondamentali dell'interprete che non possono essre modificate. La differenza tra box arrotondati e rombi è che la funzione di quest'ultimi è di selezione: prendono in input una lista di elementi e la funzione ne sceglie uno.

Il ciclo di ragionamento sarà analizzato nei prossimi paragrafi suddividendolo in dieci step. Gli step 1-4 sono quelli che riguardano l'agente per l'aggiornamento dei suoi beliefs relaviti al mondo e agli altri agenti. Gli step 5-10 descrivono la parte principale: uno degli eventi viene selezionato per essere gestito e permettere l'esecuzione di un'intenzione dell'agente.

(1) Percezione dell'ambiente

La prima azione effettuata dall'agente all'interno del ciclo di ragionamento è la precezione di ciò che lo circonda, in modo da poter aggiornare i propri beliefs sullo stato dell'ambiente. L'agente deve utilizzare quindi dei componenti capaci di percepire l'ambiente ed essere interrogati dall'agente.
In ambito applicativo l'agente accederà ai dati dei sensori, prodotti da dispositivi del mondo reale, utilizzando le opportune interfacce.

(2) Aggiornamento dei beliefs

Ottenuta la lista delle percezioni è necessario aggiornare la 'belief base', ovvero l'insieme dei beliefs dell'agente. L'aggiornamento, descritto in figura 2.1 dall'acronimo 'BUF' (Belief Update Function), avviene nella seguente maniera: ogni percezione che non è già presente nella 'belief base' viene aggiunta e viceversa i beliefs che non sono nell'elenco delle percezioni vengono rimossi.
Ognuno dei cambiamenti effettuati nell'aggiunta o rimozione di beliefs produce un evento: quelli generati da percezioni dell'ambiente sono chiaamti *eventi*

esterni. Gli *eventi interni* hanno, in più rispetto agli altri eventi, associata un'intenzione.

(3) Ricezione di comunicazioni da altri agenti

Un'altra importante sorgente di informazioni per un agente in un sistema multi-agente sono gli altri agenti. L'interprete controlla i messaggi che sono arrivati alla casella dell'agente e li rende a lui disponibili: in un ciclo di ragionamento solamente un messaggio può essere processato. Per dare rilevanza a certi messaggi è necessario utilizzare una funzione di selezione (indicata in figura 2.1 da S_M) che permette di aumentarne la priorità. Di default viene utilizzata la politica FIFO (First In First Out).

(4) Selezione dei messaggi 'Socialmente accettabili'

Prima che i messaggi siano processati, passano all'interno di una selezione che determina se possono essere accettatti o meno dall'agente. In figura 2.1 è rappresentata dalla sigla 'SoccAcc'. L'implementazione di default accetta tutti i messaggi da tutti gli agenti. Sovrascrivendo questa funzione è possibile utilizzarla per far ricevere ad un agente solo certi messaggi piuttosto che altri.

(5) Selezione di un evento

Gli agenti BDI operano gestendo continuamente eventi, i quali rappresentano sia la precezione di cambiamenti nell'ambiente sia il cambiamento dei goal dello stesso agente.
In ogni ciclo di ragionamento solo un evento può essere gestito. Ci possono essere vari eventi in attesa ma ne verrà selezionato solamente uno, il quale è scelto dalla funzione di selezione (indicata in figura 2.1 da S_E). Il set di eventi è rappresentato da una lista e i nuovi eventi sono aggiunti in fondo: l'implementazione di base della funzione seleziona il primo elemento della lista, adottando quindi una politica FIFO.

I prossimi step considerano che un evento è stato selezionato e rimosso dalla lista di eventi in attesa. Se la lista di eventi fosse vuota, la selezione dell'evento non avverrebbe e il ciclo di ragionamento salterebbe allo step 9.

(6) Recupero di tutti i paini rilevanti

Selezionato l'evento, è necessario trovare un piano che permetta all'agente di agire in modo tale da gestire quell'evento. La prima cosa da fare è recuperare dalla 'Plan Library' i piani rilevanti, verificando quali tra questi abbia un evento di attivazione che può essere unificato con l'evento selezionato. L'unificazione è il confronto che viene fatto relativo a predicato e termini.
Al fine di questo step si otterrà un set di piani rilevanti per l'evento selezionato e che nello step successivo verrà raffinato per ottenere il set di piani applicabili.

(7) Determinazione dei piani applicabili

Ogni piano ha un contesto che definisce se può essere usato in un certo momento in base alle informazioni che ha l'agente. In questo step si selezionano, tra i piani rilevanti, quelli che, in relazione alla situazione dell'agente, possono avere una possibilità di successo. Per fare questo si controlla se il contesto è una coseguenza logica della 'belief base' dell'agente. Avere più di un piano nel set di quelli applicabili significa che in base alle conoscenza dell'agente e i suoi attuali beliefs qualsiasi di questi piani sarebbe appropriato per gestire l'evento.

(8) Selezione di un piano applicabile

Sebbene qualsiasi dei piani selezionati è adeguato, cioè l'esecuzione di uno di essi sarà sufficiente per gestire l'evento selezionato in questo ciclo di ragionamento, l'agente ne deve selezionarne solamente uno e impegnarsi ad eseguirlo. Vale a dire che l'agente avrà l'intenzione di perseguire l'azione determinata da quel piano e che quindi quest'ultimo sarà presto inserito nel

set di quelli da eseguire.
La selezione del piano è fatta da una funzione di selezione (indicata in figura 2.1 da S_O). Ogni piano applicabile è considerato come una valida alternativa che l'agente ha per la gestione dell'evento. Un'evento rappresenta un particolare goal o un particolare cambiamento percepito nell'ambiente. I goal attualmente nel set degli eventi rappresentano desideri diversi che l'agente può scegliere di perseguire, mentre i piani applicabili, per uno di questi goal, rappresentano le diverse azioni che l'agente può eseguire per raggiungere quello specifico goal. L'ordine con cui il piano è selezionato dalla funzione di selezione è determinato dall'ordine con cui sono scritti nel codice sorgente dell'agente o dall'ordine con cui sono comunicati all'agente.
Ci sono due modi diversi per aggiornare il set delle intenzioni che dipendono dal fatto che l'evento selezionato sia interno (cambiamento nei goals) o esterno (cambiamento percepito nell'ambiente). Se l'agente acquisisce una nuova informazione notificata dall'ambiente viene creata una nuova intenzione per l'agente. Ogni singola intenzione nel set delle intenzioni rappresenta un diverso punto di attenzione per l'agente. Nel prossimo step verrà descritto come una particolare intenzione è scelta per essere eseguita nel ciclo di ragionamento. Per quello che riguarda gli eventi interni, essi sono creati quando l'agente ottiene un nuovo goal da raggiungere. Ciò significa che, prima che venga ripreso il corso dell'azione che ha generato l'evento, è necessario trovare ed eseguire fino al completamento un piano per raggiungere tale goal. In questo caso, non sono create nuove intenzioni ma una di quelle esistenti viene spostata in alto, il che forma una pila di piani che facilita l'interprete poichè l'intenzione da eseguire è quella più in alto.
Quando un piano è scelto dalla libreria, viene creata un'istanza di quel piano per essere inserita nel set delle intenzioni: la libreria dei piani non viene modificata ma è l'istanza che viene manipolata dall'interprete.

(9) Selezione di un intenzione per l'esecuzione

Assumendo di avere un evento da gestire, fino a questo momento nel ciclo di ragionamento abbiamo ottenuto una nuova intenzione. Tipicamente un agente ha più di un'intenzione nel set di intenzioni, ognuna delle quali rappresenta un diverso punto di attenzione e che potrebbe essere eseguita nel prossimo step del ciclo di ragionamento. Ad ogni ciclo avviene l'esecuzione di una sola intenzione, tra quelle che sono in attesa pronte per essere eseguite. Anche in questo caso è utilizzata una funzione di selezione (indicata in figura 2.1 da S_I). Dato che il raggiungimento di certi obiettivi sarà più urgente di altri, la scelta della prossima intenzione è molto importante per come l'agente opererà nell'ambiente. Il meccanismo è di tipo 'round-robin', cioè ogni intenzione è selezionata a turno e, quando viene scelta, viene eseguita solamente un'azione. Come per gli eventi, il set di intenzioni è gestito con politica FIFO: viene preso il primo elemento della lista e, una volta eseguito, viene aggiunto nuovamente alla fine. In questo modo viene garantita un'attenzione equa a tutte le intenzioni.

(10) Esecuzione di uno step di un'intenzione

Un agente ha varie intenzioni che competono tra loro per essere eseguite. Nello step precedente abbiamo scelto l'intenzione da eseguire che non è altro che il corpo di un piano formato da una sequenza di formule: ogni formula eseguita viene rimossa dal corpo dell'istanza del piano. L'intenzione viene sospesa fino a quando l'azione non viene eseguita, in attesa che l'effettore esegua l'azione e confermi al ragionatore se è stata eseguita o meno. L'intenzione sospesa, invece di essere restituita all'insieme di intenzioni, passa a un'altra struttura che memorizza tutte le intenzioni sospese che sono in attesa di un feedback dell'azione o di un messaggio: certi tipi di comunicazione richiedono che l'agente attenda una risposta prima che tale intenzione possa essere ulteriormente eseguita. Dato che un agente ha varie intenzioni e nuovi eventi da gestire, anche se alcune delle intenzioni sono attualmente

sospese, nel prossimo ciclo di ragionamento ci sarà sicuramente qualche altra intenzione da eseguire.

Prima che un'altro ciclo di ragionamento abbia inizio, l'interprete controlla che non ci siano feedback da parte degli attuatori o eventuali nuove risposte. Dopodichè le intenzioni sono aggiornate e incluse di nuovo nel set delle intenzioni, in modo da avere la possibilità di essere nuovamente selezionate nei prossimi cicli di ragionamento.

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.

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.

Librerie e teorie, pur essendo simili, sono gestite diversamente dal motore tuProlog.

Capitolo 3

Incarnazione agenti

Il progetto, come descritto nell'introduzione, ha come obiettivo l'implementazione del meta-modello di Alchemist attraverso la definizione di un incarnazione che modelli gli agenti all'interno del simulatore.

Per la realizzazione del ciclo di ragionamento dell'utente si è utilizzato il motore tuProlog importato e invocato all'interno del simulatore sfruttando la libreria 'aliCE'.

3.1 Mapping dei modelli

Il primo passo nell'evoluzione del progetto è stata l'analisi del mapping tra i due modelli, necessaria per individuare eventuali incongruenze o evidenziare opportunità a livello applicativo e maggiore espressività. Nei mapping effettuati si è cercato quindi di individuare l'entità del meta-modello di Alchemist che offrisse maggiori opportunità espressive per la definizione dell'agente.

Nella prima prova, l'agente è stato riferito ad un nodo, da cui ne deriva che l'environment sarà lo spazio che conterrà tutti gli agenti. Internamente al nodo, le molecole e le concentrazioni saranno utilizzate per gestire i beliefs dell'agente e le reazioni che saranno riferite ai piani utilizzando le condizioni come clausola per far scattare le azioni.
Questo tipo di mapping consente di realizzare simulazioni di sistemi non com-

plessi in cui vi è un solo 'livello' di agenti che interagiscono tra loro. Questa affermazione può essere compresa meglio analizzando il secondo tentativo che è stato effettuato.

Nel secondo mapping, l'agente è stato spostato più internamente al nodo riferendolo ad una reazione facendo diventare il nodo stesso uno spazio per gli agenti. In questo modo l'environment sarà uno spazio in cui possono essere presenti più nodi, i quali a loro volta potranno contenere un set di agenti.
Utilizzando questo secondo caso si riuscirà a creare un sistema con più agenti all'interno di un singolo nodo, che in ambito applicativo può essere riferito ad un device, il quale si muoverà nello spazio insieme ad altri nodi, contenitori di altri agenti.

La frequenza con cui gli eventi di Alchemist sono innescati dipende, oltre che dai parametri passati nella configurazione della simulazione, anche dalle condizioni definite per quello specifico agente: questo influisce sul numero di volte in cui viene eseguita un'azione.

3.2 Fasi di sviluppo

Il passo successivo è stato quello di stilare un piano di sviluppo per affrontare il problema attraverso step incrementali. Le parti in cui è stato deciso di suddividere il lavoro iniziale sono:

1. Scambio di messaggi
2. Gestione flusso di controllo dell'agente
3. Inserimento di condizioni nel flusso di controllo

La scelta di frammentare il probelma in sotto-problemi più semplici è stata presa per facilitare l'integrazione tra due i modelli e capire come utilizzarli.
Al termine degi step sopra descritti si otterrà un'incarnazione che sarà la base per le successive implementazioni. Per completare lo sviluppo del modello ad agenti si dovrà provvedere a sviluppare la possibilità per l'agente di:

- spostarsi, ovvero di muovere l'entità di Alchemist che lo ospita, modificando velocià e direzione
- ottenere la distanza dagli altri agenti
- aggiungere o rimuovere belief dalla 'belief base' e innescare la notifica dell'avvenuta modifica per potervi reagire

3.3 Sviluppo sezioni

Dopo aver esaminato i due modelli e aver analizzato i mapping realizzati si è deciso di implementare la versione che riferisce l'agente alla reazione poichè seguendo lo schema del meta-modello di Alchemist l'implementazione risulta più immediata e espressiva.
Per la definizione della teoria dell'agente verrà utilizzato il tuProlog che sarà richiamato all'interno del ciclo di ragionamento importando la libreria 'alice.tuprolog' che fornisce i costrutti e il motore tuProlog.
All'interno dell'implementazione delle azioni di Alchemist sarà quindi caricata la teoria dell'agente e successivamente utilizzata attraverso la libreria appena descritta.

3.3.1 Definizione incarnazione

Lo sviluppo è partito dalla definizione della classe **AgentIncarnation** che implementa l'interfaccia *Incarnation*. I metodi definiti nell'interfaccia consentono di caratterizzare l'incarnazione nella creazione delle varie entità del modello (nodi, distribuzioni temporali, reazioni, condizioni, azioni).

Per la creazione del **nodo** si è definita la classe **AgentsContainerNode** che estende *AbstractNode*. Questa classe ha tra le sue proprietà il riferiemnto all'environment in cui si trova il nodo e una struttura dati composta da coppie chiave e valore (in cui la chiave è il nome dell'agente e il valore è il riferimento all'azione dell'agente).

La **distribuzione temporale** di ogni reazione è stata realizzata istanziando la classe *DiracComb* inizializzata con il parametro recuperato dal file di configurazione della simulazione. La classe permette di emettere eventi ad un intervallo temporale specificato dal parametro passato.

Per le **reazioni** è stata definita la classe **AgentReaction** che implementa *AbstractReaction* e che rappresenta l'agente e che contiene le condizioni che devono verificarsi per far avvenire le azioni, che sono il fulcro dell'agente. Come proprietà della classe è presente solo una stringa che memorizza il nome dell'agente.
La creazione delle **condizioni** è stata fatta istanziando la classe *AbstractCondition* e implementando i metodi mancanti dell'interfaccia Condition: *getContext* (definisce la profondità della condizione tra GLOBAL, NEIGHBORHOOD, LOCAL), *getPropensityContribution* (permette di influenzare la velocità della reazione che decide se utilizzare o meno questo parametro), *isValid* (definisce la clausola per la valità della condizione).
L'**azione** da creare è passata dalla reazione. Qui verrà gestito il ciclo di ragionamento dell'agente con il metodo *execute* definito nell'interfaccia *Action* e saranno utilizzati i costrutti forniti dalla libreria 'alice.tuprolog' per invocare i piani della teoria dell'agente e poi gestirne il risutlato in Alchemist.

3.3.2 Scambio di messaggi

Per lo scambio di messaggi sono state definite le classi **SimpleAgentAction**, che estende *AbstractAction*, e **PostmanAction** che è una specializzazione della prima. La classe SimpleAgentAction rappresenta la definizione standard di un agente e ha come proprietà il nome dell'agente, una mailbox formata da due code (una per la posta in entrata e una per quella in uscita) e un motore tuProlog. Al suo interno sono implementati i metodi *execute* (che è il metodo principale in cui avviene il ciclo di ragionamento) e i metodi per la gestione delle caselle dei messaggi, le cui strutture sono definite nelle classi innestate InMessage e OutMessage rispettivamente per i messagggi in entrata e in uscita.

All'interno del ciclo di ragionamento dell'agente viene richiamato il piano *receive* che prende, se presente, un messaggio dalla cima della pila di quelli in entrata e ne estrae i termini (mittente e contenuto), i quali vengono passati ad un altro piano, per la gestione del messaggio, definito nella teoria dell'agente. Per quanto riguarda l'invio è stato definito il piano *send* attraveso il quale, specificando destinatario e contenuto, è possibile notificare un messaggio ad un altro agente.

I messaggi sono gestiti utilizzando strutture e metodi Java sviluppati in Alchemist che si occupano anche di aggiungerli e prelevarli dalla teoria dell'agente in modo che esso possa utilizzarli e inserirli nelle opportune code della mailbox.

La classe **PostmanAction** sovrascrive l'implementazione del metodo *execute* in modo da invocare, ad ogni evento lanciato dal simulatore, un metodo che provvederà a prelevare i messaggi dalla coda in uscita da ogni agente e recapitarli ai corretti destinatari inserendoli nella coda di quelli in entrata.

Alla creazione di un'istanza SimpleAgentAction viene caricato il file contenente la teoria tuProlog dell'agente. Per lo scambio di messaggi sono state definite le teorie mostrate qui di seguito, una per l'agente Ping (Codice sorgente 3.1) e una per l'agente Pong (Codice sorgente 3.2).

```
1 init :-
2   send('pong_agent','ping').
3
4 receive :-
5   retract(ingoing(S,M)),
6   handle(S,M).
7
8 handle(S,pong) :-
9   send(S, ping).
10
11 handle(_,go_away) :-
12   act(forward).
13
14 send(R, M) :-
15   self(S),
16   assertz(outgoing(S,R, M)).
```

Codice sorgente 3.1: Teoria agente Ping

```
1
2
3
4 receive :-
5   retract(ingoing(S,M)),
6   handle(S,M).
7
8 handle(S,ping) :-
9   send(S, pong).
10
11 handle(_,go_away) :-
12   act(forward).
13
14 send(R, M) :-
15   self(Sender),
16   assertz(outgoing(Sender,R, M)).
```

Codice sorgente 3.2: Teoria agente Pong

Come si può notare, l'agente Ping ha al suo interno la definizione del piano 'init' che consente l'invio del primo messaggio, che da il via allo scambio con l'agente Pong.

Per avviare una simulazione utilizzando l'incarnazione ad agenti appena descritta sono stati testati due file di confiurazioni diversi.
La simulazione descritta nel Codice sorgente 3.3 prevede la dispozizione di tre agenti (ping_agent, pong_agent e postman) che risiedono all'interno di un unico nodo, mentre quella descritta nel Codice sorgente 3.4 posiziona ogni agente su un nodo diveso. Gli agenti ping_agent e pong_agent sono istanze della classe SimpleAgentAction, mente postman è istanza della classe PostmanAction. Lo spazio in cui sono posizionati i nodi nello spazio è in entrambi i casi un cerchio con centro (0,0) di raggio 2.

```
1 incarnation: agent
2
3 network-model:
```

```
4     type: ConnectWithinDistance
5     parameters: [10]
6
7   displacements:
8     - in: {type: Circle, parameters: [1,0,0,2]}
9       programs:
10        -
11          - time-distribution: 1
12            program: "ping_agent"
13
14          - time-distribution: 1
15            program: "pong_agent"
16
17          - time-distribution: 1
18            program: "postman"
```

Codice sorgente 3.3: Simulazione con agenti sullo stesso nodo

```
1   incarnation: agent
2
3   network-model:
4     type: ConnectWithinDistance
5     parameters: [10]
6
7   displacements:
8     - in: {type: Circle, parameters: [1,0,0,2]}
9       programs:
10        -
11          - time-distribution: 1
12            program: "ping_agent"
13
14    - in: {type: Circle, parameters: [1,0,0,2]}
```

```
15      programs:
16        -
17          - time-distribution: 1
18            program: "pong_agent"
19
20    - in: {type: Circle, parameters: [1,0,0,2]}
21      programs:
22        -
23          - time-distribution: 1
24            program: "postman"
```

Codice sorgente 3.4: Simulazione con agenti su nodi diversi

3.3.3 Spostamento del nodo

Terminato lo sviluppo dello scambio di messaggi ci si è resi conto che lo sviluppo delle parti di gestione del flusso di controllo e di inserimento di condizioni al suo interno fossero realizzabili con poco sforzo. Quindi si è passati alla realizzazione dello spostamento del nodo che ospita l'agente.

Per realizzare lo spostamento sono stati inseriti nella classe AgentsContainerNode tre campi per memorizzare la velocità di spostamento del nodo, la direzione o angolo dello spostamento (rappresentata da un valore espresso in radianti) e il tempo della simulazione in cui è avvenuto l'ultimo aggiornamento della posizione. Oltre a queste proprietà sono stati inseriti anche i metodi per recuperare e aggiornare tali valori.
Il metodo che effettua lo spostamento vero e proprio del nodo è *changeNodePosition* che prende come parametro il tempo della simulazione corrente e ad ogni ciclo ci ragionamento effettua l'aggiornamento della posizone del nodo che ospita l'agente. All'interno del metodo viene costruito un cerchio che ha come centro le coordinate attuali del nodo e come raggio la differenza di tempo rispetto al precedente spostamento moltiplicata per la velocità. La nuova

posizione è un punto della circonferenza che viene individuato utilizzando l'angolo o direzione del nodo.

Per simulare il movimeno del nodo che ospita l'agente all'interno dell'ambiente è stata modificata la teoria degli agenti ping e pong (indicata rispettivamente nei Codici sorgenti 3.1 e 3.2) aggiungendo un belief che rappresenta il bounding box dello spazio all'interno del quale si può muovere l'agente.

```
1  field(5,5,-5,-5).
```

Codice sorgente 3.5: Bounding-box

Dopodichè sono stati aggiunti dei piani per verificare se la posizione del nodo risulta all'interno del bounding box. Dal ciclo di ragionamento si risolve il piano *checkPosition* utilizzando come termini le coordinate della posizone del nodo. I piani *isInFieldX* e *isInFieldY* verificano che le coordinate rientrino rispettivamente nei limiti di ascisse e ordinate, altrimenti viene aggiunto un belief contenente la seguente codifica: T(top), R(right), B(bottom), L(left).

```
18  isInFieldX(X) :-
19    field(T,R,B,L),
20    R =< X,
21    asserta(reachedLimit('R')).
22
23  isInFieldX(X) :-
24    field(T,R,B,L),
25    R > X,
26    true.
27
28  isInFieldX(X) :-
29    field(T,R,B,L),
30    L >= X,
31    asserta(reachedLimit('L')).
32
```

```
33  isInFieldX(X) :-
34    field(T,R,B,L),
35    L < X,
36    true.
37
38  isInFieldY(Y) :-
39    field(T,R,B,L),
40    T =< Y,
41    asserta(reachedLimit('T')).
42
43  isInFieldY(Y) :-
44    field(T,R,B,L),
45    T > Y,
46    true.
47
48  isInFieldY(Y) :-
49    field(T,R,B,L),
50    B >= Y,
51    asserta(reachedLimit('B')).
52
53  isInFieldY(Y) :-
54    field(T,R,B,L),
55    B < Y,
56    true.
57
58  checkPosition(X,Y) :-
59    isInFieldX(X),
60    isInFieldY(Y).
```

Codice sorgente 3.6: Piani per la gestione del bounding-box

Gli eventuali belief aggiunti per il raggiungimento dei limiti del bounding

box, vengono 'consumati' all'interno del ciclo di ragionamento che prevede la modifica della direzione del nodo attraverso la chiamata al metodo *changeDirectionAngle* per riportarlo all'interno dei limiti.

Per la simulazione è stato utilizzato il file di configurazione descritto nel Codice sorgente 3.4 che prevede la disposizione degli agenti su nodi differenti.

3.3.4 Aggiornamento belief base

Per quanto riguarda l'aggiornamento delle 'belief base' si è pensato di incapsulare i predicati 'asserta','assertz' e 'retract' utilizzando strutture dati e piani tuPorlog per svolgere lo stesso comportamento: modifica della 'belief base', notifica del cambiamento tramite evento. In questo modo si è in grado di spezzare l'inserimento o la rimozione del belief dalla notifica del cambiamento, evitando che un agente operi un ciclo di ragionamento senza mai terminare, ovvero che l'agente continui a svolgere compiti lato tuProlog senza far tornare il controllo delle ad Alchemist.
La prima parte, quella dell'aggiunta o rimozione del belief dalla 'belief base', è incapsulata all'interno dei piani 'addBelief(B)' e 'removeBelief(B)' i quali inoltre si occupano di aggiungere un'altro belief fittizio ('added_belief(B)' o 'removed_belief(B)') che verrà utilizzato per divulgare l'evento. La parte di notifica viene svolta da Alchemist ed è divisa in due fasi: inizialmente viene recuperato il belief fittizio e inserito il suo termine all'interno di una mappa, dopodichè quest'ultima viene iterata e per ogni belief viene invocato un evento 'onAddBelief(B)' o 'onRemoveBelief(B)' relativamente alla tipologia di modifica della 'belief base' avvenuta e dove il B è il termine recuperato precedentemente. La definizione dei piani per aggiungere e rimuovere i belief è la seguente.

```
1addBelief(B) :-
2  assertz(belief(B)),
3  assertz(added_belief(B)).
4
```

```
5removeBelief(B) :-
6  retract(belief(B)),
7  assertz(removed_belief(B)).
```

Codice sorgente 3.7: Piani per la gestione del bounding-box

3.4 Definizione del linguaggio

Lo sviluppo del progetto fino a questo momento è stato guidato dalla realizzazione di step volti a produrre una base di partenza e delle conoscenze da poter utilizzare nel proseguimento dell'implementazione. Quindi si è voluto consolidare quanto fatto e riordinare gli sviluppi per avere una gestione uniforme e comune. Si è dunque deciso di sviluppare un 'linguaggio' definendo una serie di piani e belief messi a disposizione del programmatore dell'agente (colui che scriverà le teorie degli agenti) per permettergli di definire i comportamenti degli agenti per le simulazioni. Per fare questo si è partiti dai piani e belief già descritti, riorganizzandoli e poi inserendoli in una 'libreria', ovvero una teoria di base caricata all'interno dell'agente.
Per programmare l'agente sono messi a disposizione i seguenti beliefs:

- **self(A).** Rappresenta il nome dell'agente riferito dal termine A
- **belief(position(X,Y)).** Memorizza nell'agente le coordinate della posizone del nodo che lo ospita
- **belief(movement(S,D)).** Mantiene le informazioni relative a velocità (S) e direzione (D), rappresentata come angolo in radianti, del nodo in cui è posizionato l'agente
- **belief(distance(A,D)).** Ogni agente ha al suo interno N-1 di questi belief dove N è il numero totale degli agenti; rappresenta la distanza (D) di ogni agente del vicinato (A): il vicinato è calcolato con la linking-rule definita nella configurazione della simulazione

Per quanto riguarda i piani disponibili vi sono due tipologie: quelli che innescano un evento e quelli incocati per reagire ad un evento. I piani che fanno parte del primo tipo sono:

- **sendMessage(R,M).** Consente di inviare un messaggio (M) ad un destinatario (R)
- **addBelief(B).** Aggiunge un belief alla 'belief base'
- **removeBelief(B).** Rimuove un belief dalla 'belief base'.

L'altra tipologia riguarda piani invocati per consentire all'agente di reagire ad un evento. Il comportamento degli agenti dipende dall'implementazione del corpo del piano definita dal programmatore dell'agente. Nella teoria dell'agente è possibile definire più piani per ognuna delle seguenti tipologie: solo per 'init' è possibile definire una sola implementazione.

- **init.** Invocato all'inizializzazione dell'agente cioè la prima volta che viene eseguito il ciclo di ragionamento
- **onReceivedMessage(S,M).** Richiamato per consentire di gestire il messaggio (M) ricevuto dal mittente (S)
- **onAddBelief(B).** Ad ogni inserimento di nuovi belief o aggiornamento della 'belief base' (come quello per la posizione o la distanza dagli altri agenti) viene invocato questo piano passando come termine il belief
- **onRemoveBelief(B).** Richiamato ad ogni rimozione di belief dalla 'belief base'

Per definire il linguaggio si è cercato di racchiudere tutte le caratteristiche dell'agente all'interno di una liberia di piani che fornisce una base al programmatore dell'agente e che consente di avere un'espressività maggiore potendo definire il comportamento delle parti scatenate dagli eventi.
I metodi messi a disposizione della libreria sono implementati come mostrato nel Codice sorgente 3.8.

```
1  receiveMessage :-
2    retract(ingoing(S,M)),
3    onReceivedMessage(S,M).
4
5  sendMessage(R, M) :-
6    self(S),
7    assertz(outgoing(S,R, M)).
8
9  addBelief(B) :-
10   assertz(belief(B)),
11   assertz(added_belief(B)).
12
13 removeBelief(B) :-
14   retract(belief(B)),
15   assertz(removed_belief(B)).
```

Codice sorgente 3.8: Libreria agenti

Le parti già sviluppate in precedenza sono state riprese, riorganizzate e adattate alla libreria appena descritta. Il ciclo di ragionamento dell'agente eseguito da Alchemist nelle simulazioni è quindi ora formato dalle chiamate a funzione descritte nel Codice sorgente 3.9.

```
1  handleIncomingMessages();
2
3  readMessage();
4
5  positionUpdate();
6
7  getBeliefBaseChanges();
8
9  notifyBeliefBaseChanges();
```

```
10
11    hanldeOutGoingMessages();
```

Codice sorgente 3.9: Implementazione ciclo di ragionamento

Il metodo ***handleIncomingMessages*** preleva dalla casella di posta in entrata eventuali messaggi presenti e li inserisce nella teoria dell'agente sotto forma di belief 'ingoing(S,M)', dove S è il mittente e M il messaggio.
La chiamata successiva al metodo ***readMessage*** tenta di risolvere il goal 'receive', presente nella libreria dell'agente, che prevede la rimozione del belief 'ingoing' e, come descritto nel Codice sorgente 3.8, conseguentemente invoca il piano 'onReceiveMessage' passandogli i termini appena recuperati: in base alla implementazioni del piano definite per l'agente si avranno comportamenti diversi.
Il terzo metodo invocato, ***positionUpdate***, riguarda l'aggiornamento della posizione. Per prima cosa viene calcolata la nuova posizione, poi rimossa la precedente dalla teoria dell'agente e quindi inserita quella nuova. L'aggiornamento della posizione produce due azioni: la preparazione per la notifica dell'evento di aggiornamento e la chiamata al metodo *getNeighborhoodDistances*. La prima avviene aggiungendo il belief nella mappa delle notifiche dei cambiamenti della 'belief base' mentre la seconda prevede il calcolo della distanza di ogni agente da quello agente corrente: questa seconda azione opera l'aggiornamento dei belief e quindi una modifica della 'belief base', che comporta l'aggiunta dei belief alla mappa per la notifica degli eventi.
Il metodo ***getBeliefBaseChanges*** recupera dalla teoria dell'agente i belief del tipo 'added_belief(B)' o 'removed_belief(B)', ovvero quelli aggiunti da chiamate ai piani 'addBelief(B)' o 'removeBelief(B)'. Per ogni belief aggiunto o rimosso lo aggiunge alla mappa per la notifica degli eventi.
Nel metodo ***notifyBeliefBaseChanges*** viene presa la mappa con tutti i belief aggiunti in precedenza e per ogni belief viene invocato il piano 'onAddBelief(B)' o 'onRemoveBelief(B)' a seconda che il piano sia stato rispettivamente aggiunto o rimosso dalla 'belief base'. Nel caso della posizione e della

distanza tra gli agenti, per gestire l'aggiornamento, viene inserito, e quindi notificato, solamente il belief di aggiunta.
Al termine del ciclo di ragionamento viene invocato il metodo ***hanldeOutGoingMessages*** che recupera dalla teoria dell'agente tutti i belief del tipo 'outgoing(S,R,M)', dove S è il mittente, R il destinatario e M il contenuto del messaggio, e li inserisce nella casella di posta in uscita dell'agene.

Come spiegato nella sezione 3.3.2 lo scambio di messaggi avviene grazie all'agente 'postman' istanza della classe PostmanAction e che nel suo ciclo di ragionamento ha come unico compito quello di prelevare dalla coda in uscita di ogni agente i messaggi e inserendoli nella coda in entrata del destinatario. Per fare ciò sono stati implementati due metodi nella classe SimpleAgentAction per recuperare la lista dei messaggi da inviare, liberando quindi la casella di posta in uscita, e per inserire un nuovo messaggio nella casella di posta in entrata, la quale verrà svuotata dal metodo ***handleIncomingMessages*** al prossimo ciclo di ragionamento.

3.5 Rifattorizzazione incarnazione

Terminata la definizione del linguaggio è stata effettuata una rifattorizzazione delle classi che compongono l'incarnazione, principalmente per quanto rigurda la struttura creata per l'implementazione dell'agente. Lo stato dell'arte prima della rifattorizzazione prevedeva che la classe di Alchemist *AbstractAction* fosse estesa da **SimpleAgentAction**, la quale implementava i metodi per la gestione del ciclo di ragionamento per l'agente. Quest'ultima veniva poi estesa da **PostmanAction**.

Con la rifattorizzazione, si è voluto creare una struttura con una classe principale che implementa i metodi per gestire il comportamento degli agenti e che sia estesa da altre classi, le quali implementino il ciclo di ragionamento invocando i metodi della classe padre.
Seguendo questa logica è stata creata la classe astratta **AbstracAgent** che estende *AbstractAction* e che, dei metodi rimasti da implementare dell'inter-

faccia *Action*, definisce solamente *getContext* lasciando alla classe dell'agente che si vuole creare la capacità di implementare *execute* e *cloneAction*, rispettivamente utilizzati per il ciclo di ragionamento e per clonare l'azione. Nella classe astratta creata, sono state quindi definite le variabili per la gestione dell'agente, tra cui quelle per le code dei messaggi (entrata e uscita), il motore tuProlog e una struttura per la notifica dei cambiamenti della 'belief base'. Nel costruttore viene caricata inoltre la libreria definita nel Codice sorgente 3.8 nella quale sono descritti i piani che definiscono il linguaggio e invocabili nella teoria dell'agente. Inoltre la classe contiene i metodi per il comportamento dell'agente nel ciclo di ragionamento descritti nel Codice sorgente 3.9 e quelli per l'inizializzazione. Quest'ultima Ã¨ identificata come la prima esecuzione del ciclo di ragionamento dell'agente e che quindi racchiude delle istruzioni basilari per la sua corretta operatività: i metodi in questione sono *inizializeAgent* e *initReasoning*.

Estendendo la classe appena descritta è stata creata **SimpleAgent** che racchiude tutte le caratteristiche principali di un agente descritte precedentememte. Alla sua istanzazione, oltre a richiamare il costruttore del padre, carica nel motore tuProlog il file contenente la sua teoria. Gli unici metodi rimasti da definire per creare l'agente sono *execute* e *cloneAction*. L'implementazione del primo è mostrata nel Codice sorgente 3.10 mentre la seconda è semplicemente la creazione di una nuova istanza dell'oggetto che prende il nome 'cloned_' seguito dal nome dell'agente (es. per l'agente 'pong' verrà creato l'agente 'cloned_pong').

```
1   if (!this.isInitialized()) {
2       //Agent's first reasoning cycle
3       this.inizializeAgent();
4
5       this.initReasoning();
6   } else {
7       //Agent's reasoning cycle
8
```

```
9       this.handleIncomingMessages();
10
11      this.readMessage();
12
13      this.positionUpdate();
14
15      this.getBeliefBaseChanges();
16
17      this.notifyBeliefBaseChanges();
18
19      this.hanldeOutGoingMessages();
20  }
```

Codice sorgente 3.10: Simple Agent Reasoning Cycle

Allo stesso modo di **SimpleAgent** è stato creata la classe **PostmanAgent** che, nel costruttore non carica nessuna teoria e, nel suo ciclo di ragionamento esegue una sola operazione, ovvero quella di prelevare i messaggi in uscita dai vari agenti e recapitarli al corretto destinatario. La sua implementazione del metodo *execute* è descritta nel Codice sorgente 3.11.

```
1   getNode().postman();
```

Codice sorgente 3.11: Postman Agent Reasoning Cycle

Le altre parti dell'incarnazione ad agenti sono **AgentReaction** (oggetto che contiene l'agente e la condizione di esecuzione del ciclo di ragionamento) e **AgentsContainerNode** (che rappresenta il nodo che contiene gli agenti). La reazione, creata nell'incarnazione utilizzando la classe **AgentReaction**, è il contenitore più aderente all'agente. Infatti la reazione è composta da una condizione (che è sempre verificata) e l'azione, ovvero l'agente. Il superamento della condizione permette di scatenare il ciclo di ragionamento dell'agente. La classe **AgentsContainerNode** crea un luogo in cui uno o

più agenti posson risiedere. Il nodo ha al suo interno le variabili per impostare la velocià e la direzione (calcolata in radianti), una mappa degli agenti presenti all'interno del nodo e il riferimento all'ambiente in cui è contenuto. Nel nodo sono definiti i metodi per conoscere la posizione del nodo, modificarne velocità e direzione, calcolarne lo spostamento del nodo e recuperare le distanze dagli altri agenti.

Con questa rifattorizzazione si conclude la prima fase del progetto che ha permesso di integrare all'interno del simulatore Alchemist il modello ad agenti. L'implementazione dell'agente realizzata permette di scambiare messaggi con gli altri agenti, spostare il nodo in cui risiede e gestire l'aggiornamento della 'belief base'. È possibile creare simulazioni di agenti e per ognuno caricare una teoria nella quale è definito il loro comportamento.

Capitolo 4

Spatial Tuples

Terminata l'implementazione dei componenti che consentono di definire il comportamento di un agente quali scambio di messaggi, spostamento del nodo e aggiornamenro della 'belief base' si è passati all'implementazione dell'estensione di Spatial Tuples per gli agenti. Spatial Tuples è un'estensione del modello base di tuple per i sistemi distribuiti multi agente dove:

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

4.1 Modello

Spatial Tuples supporta esplicitamente la consapevolezza dello spazio e la coordinazione basata sullo spazio dell'agente in scenari di calcolo pervasivo. In questa estensione le informazioni hanno una posizione e una estensione nello spazio fisico: lo spazio di tuple è quindi concepito come un aumento della realtà fisica dove le tuple rappresentano uno strato di informaizoni associate ad un'informazione spaziale. Una volta che la tupla è associata ad

una regione o posizione, le informazioni sono proprietà che possono essere attribuite a quella porzione di spazio fisico.

Le primitive che permettono la comunicazione sono:

- out(t): emette tuple spaziali t e le associa ad una regione
- rd(tt): ricerca tuple che corrispondano al template tt e le ritorna
- in(tt): ricerca tuple che corrispondano al template tt e le consuma

Le richieste in Spatial Tuples sono sospensive e non deterministiche. Sospensive perchè se non ci sono tuple che fanno match con il template l'operazione è bloccata finchè non viene trovata la tupla. Il non determinismo, invece, deriva dal fatto che se ci fossero più tuple che corrispondono al template ne viene scelta una casualmente.

Le tuple spaziali possono essere associate a posizioni o regioni in modo diretto o indiretto. Le operazioni dirette associano direttamente la tupla spaziale ad una posizione o una regione. Quelle indirette, invece, associano tuple a componenti situati e vale anche se la regione in cui si trova il componente cambia nel tempo: finchè non viene rimossa, la tupla è associata alla regione del componente in cui è situata.

4.2 Implementazione modello

Per inserire il modello appena descritto all'interno dell'incarnazione sono state analizzate le sue parti per decidere l'implementazione delle primitive da fornire al programmatore dell'agente per permettergli di fer comunicare gli agenti con lo spazio di tuple.

Sono stati quindi definiti i seguenti piani che rispecchiano rispettivamente le primitive **out(t)**, **rd(tt)**, **in(tt)** definte dal modello Spaital Tuples e sono state aggiunte alla libreria tuProlog degli agenti dell'incarnazione.

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

```
3
4   readTuple(TT) :-
5       assertz(read(TT)).
6
7   takeTuple(TT) :-
8       assertz(take(TT)).
```

Codice sorgente 4.1: Piani Spatial Tuples

I piani così descritti si occupano semplicemente di creare un wrapper per aggiungere un nuovo belief nella teoria.

Successivamente, per modellare le posizioni degli spazi di tuple, è stata definita la classe **Blackboard** che estende da **AbstractAgent** e implementa una specializzazione dell'agente. Questa classe rappresenta un punto situato nello spazio che può contenere informazioni: gli agenti possono, attraverso le primitive descritte nel Codice sorgente 4.1, scrivere tuple oppure leggere o prendere quelle che corrispondono al template passato. All'interno della classe sono state create due liste una per le richieste in entrata e una per quelle in attesa. La classe espone il metodo *insertRequest* al quale sono passati la tupla/template, l'istanza dell'agente e l'azione da eseguire (*write*/*read*/*take*). La richiesta viene aggiunta alla coda e processata nel ciclo di ragionamento di questo agente. In base alla tipologia di azione vengono invocati i diversi metodi interni *writeOnBlackboard*, *readOnBlackboard*, *takeOnBlackboard* per la gestioni della richiesta. Come detto nella sezione 4.1, se le operazioni richieste non possono essere effettuate vengono sospese e inserite nella coda di quelle in attesa: questa lista viene iterata, come quella delle richieste in entrata, ad ogni ciclo di ragionamento.
Per le richieste di tipo *read* e *take*, se l'operazione di match del template va a successo deve essere notificata all'agente la tupla letta/prelevata. Questa azione è fatta inserendo nella struttura di notifica dei belif dell'agente un nuovo elemento, che è la tupla recuperata dallo spazio di tuple, e che verrà notificata nel prossimo ciclo di ragionamento dell'agente.

Per completare l'implementazione è necessario definire un metodo nella classe AbstracAgent che consente agli agenti, durante il loro ciclo di ragionamento, di recuperari i belief inseriti con i piani descritti nel Codice sorgente 4.1 e di inserirli nello spazio di tuple. È stato quindi implementato il metodo *retrieveTuples* che per prima cosa recupera tutte le tuple dalla teoria dell'agente e poi in base al tipo di azione esegue delle operazioni diverse.
Se l'azione è *write*, recupera l'agente di tipo Blackboard più vicino e se lo trova invoca il metodo per inserire la richiesta.
Diversamente, se l'azione è *read* o *take*, viene recuperata la lista di agenti di tipo Blackboard presenti nel suo vicinato. Per ognuno degli agenti della lista viene inserita la richiesta in modo tale da estendere la ricerca della tupla che corrisponda al template.

4.3 Simulazione

Per testare l'incarnazione con l'implementazione del modello appena descritto è stato utilizzato il pattern di coordinazioen 'breadcrumbs'. L'esempio prodotto utilizza due agenti, Hansel e Gretel, che si muovono all'interno di un'ambiente in cui sono presenti altri agenti utilizzati come spazi di tuple situati. L'agente Hansel si sposterà casualmente nell'ambiente lasciando le briciole negli spazi di tuple più vicini al suo passaggio, mentre l'agente Gretel dovrà muoversi alla ricerca delle briciole e una volta che ne ha individuata una iniziare a seguire le altre per raggiungere Hansel.

Per realizzare questa simulazione è stato scritto il file di configurazione indicato dal Codice sorgente 4.2.

```
1  incarnation: agent
2
3  network-model:
4  type: ConnectWithinDistance
5  parameters: [0.5]
6
```

```
7   displacements:
8   - in: {type: Circle, parameters: [1,-1.8,2,0.2]}
9   programs:
10     -
11       - time-distribution: 1
12         program: "hansel"
13
14  - in: {type: Circle, parameters: [1,-1.5,4,0.2]}
15  programs:
16     -
17       - time-distribution: 1
18         program: "gretel"
19
20  - in: {type: Circle, parameters: [1500,2,2,4.5]}
21  programs:
22     -
23       - time-distribution: 1
24         program: "blackboard"
```

Codice sorgente 4.2: Simulazione modello Spatial Tuples con modello di coordinazione breadcrumbs

Le teorie degli agenti utilizzate in questa simulazione sono indicate qui di seguito.

```
1   init :-
2       writeTuple(breadcrumb(hansel,here)),
3       removeBelief(movement(S,D)),
4       addBelief(movement(0.027,4.8)),
5       addBelief(counter(0,0)),
6       addBelief(spiralCorner(0.08)),
7       takeTuple(stop(hansel)).
8
```

```
9   onAddBelief(position(X,Y)) :-
10      removeBelief(counter(C1,C2)),
11      handlePosition(C1,C2,X,Y).
12
13  handlePosition(C1,C2,X,Y) :-
14      C1 < 300,
15      C2 < 10,
16      C1N is C1 + 1,
17      C2N is C2 + 1,
18      addBelief(counter(C1N,C2N)),
19      writeTuple(breadcrumb(hansel,here)).
20
21  handlePosition(C1,C2,X,Y) :-
22      C1 < 300,
23      C1N is C1 + 1,
24      C2 >= 10,
25      addBelief(counter(C1N,0)),
26      removeBelief(movement(S,D)),
27      belief(spiralCorner(V)),
28      D1 is D + V,
29      addBelief(movement(S,D1)).
30
31  handlePosition(C1,C2,X,Y) :-
32      C1 >= 300,
33      C2N is C2 + 1,
34      addBelief(counter(0,C2N)),
35      removeBelief(spiralCorner(V)),
36      TEMP is V * 0.30,
37      V1 is V + TEMP,
38      addBelief(spiralCorner(V1)).
39
```

```
40
41  onAddBelief(distance(_,_)) :-
42      true.
43
44  onAddBelief(distance(_,_,_)) :-
45      true.
46
47  onAddBelief(movement(_,_)) :-
48      true.
49
50  onRemoveBelief(movement(_,_)) :-
51      true.
52
53  onAddBelief(counter(C1,C2)) :-
54      true.
55
56  onRemoveBelief(counter(C1,C2)) :-
57      true.
58
59  onAddBelief(spiralCorner(V)) :-
60      true.
61
62  onRemoveBelief(spiralCorner(V)) :-
63      true.
64
65  onResponseMessage(msg(stop(hansel),X,Y)) :-
66      removeBelief(movement(_,D)),
67      addBelief(movement(0,D)),
68      writeTuple(stop(gretel)).
```

Codice sorgente 4.3: Teoria agente Hansel

```
1  init :-
2      removeBelief(movement(S,D)),
3      addBelief(movement(0.027,4.3)),
4      takeTuple(breadcrumb(hansel,here)),
5      addBelief(counter(0)),
6      takeTuple(stop(gretel)).
7
8  onAddBelief(position(X,Y)) :-
9      removeBelief(counter(C)),
10     handlePosition(C,X,Y),
11     checkDistance.
12
13 handlePosition(C,X,Y) :-
14     C < 15,
15     C1 is C + 1,
16     addBelief(counter(C1)).
17
18 handlePosition(C,X,Y) :-
19     C >= 15,
20     addBelief(counter(0)),
21     removeBelief(movement(S,D)),
22     D1 is D + 0.08,
23     addBelief(movement(S,D1)).
24
25 onAddBelief(distance(A,ND,OD)) :-
26     true.
27
28 onAddBelief(distance(A,ND)) :-
29     true.
30
31 onAddBelief(movement(_,_)) :-
```

```
32      true.
33
34  onRemoveBelief(movement(_,_)) :-
35      true.
36
37  onAddBelief(counter(C)) :-
38      true.
39
40  onRemoveBelief(counter(C)) :-
41      true.
42
43  onResponseMessage(msg(stop(gretel),X,Y)) :-
44      removeBelief(movement(_,D)),
45      addBelief(movement(0,D)).
46
47  onResponseMessage(msg(breadcrumb(hansel,here),X,Y
    )) :-
48      removeBelief(counter(_)),
49      addBelief(counter(0)),
50      changeDirection(X,Y).
51
52  checkDistance :-
53      belief(distance(hansel,D)),
54      D < 0.08,
55      writeTuple(stop(hansel)).
56
57  checkDistance :-
58      takeTuple(breadcrumb(hansel,here)).
59
60  changeDirection(X2,Y2) :-
61      belief(position(X1,Y1)),
```

```
62        DX is X2 - X1,
63        DY is Y2 - Y1,
64        calculateAtan(DY,DX).
65
66    calculateAtan(DY,DX) :-
67        DX > 0,
68        RAD is atan(DY / DX),
69        removeBelief(movement(S,D)),
70        addBelief(movement(S,RAD)).
71
72    calculateAtan(DY,DX) :-
73        DX < 0,
74        DY >= 0,
75        TMP is atan(DY / DX),
76        RAD is TMP + 3.14,
77        removeBelief(movement(S,D)),
78        addBelief(movement(S,RAD)).
79
80    calculateAtan(DY,DX) :-
81        DX < 0,
82        DY < 0,
83        TMP is atan(DY / DX),
84        RAD is TMP - 3.14,
85        removeBelief(movement(S,D)),
86        addBelief(movement(S,RAD)).
87
88    calculateAtan(DY,DX) :-
89        DX == 0,
90        DY > 0,
91        RAD is 3.14 / 2,
92        removeBelief(movement(S,D)),
```

```
93        addBelief(movement(S,RAD)).
94
95    calculateAtan(DY,DX) :-
96        DX == 0,
97        DY < 0,
98        RAD is -3.14 / 2,
99        removeBelief(movement(S,D)),
100       addBelief(movement(S,RAD)).
```

Codice sorgente 4.4: Teoria agente Gretel

```
1   init :-
2       true.
```

Codice sorgente 4.5: Teoria per gli spazi di tuple

Bibliografia

[1] alchemistsimulator.github.io

[2] Programming Multi-Agent Systems in AgentSpeak using Jason, (Rafael H. Bordini, Jomi Fred Hübner, Michael Wooldridge), Wiley, Interscience (2007)